Master Thesis in Computer Science

User Interface Components to Support Simple and Efficient Use and Control of Large, High Resolution Displays

Daniel Stødle

11th February 2005

Faculty of Science
Department of Computer Science
University of Tromsø, N-9037 Tromsø, Norway

Abstract

This thesis investigates basic user interface abstractions and tools, with the purpose of simplifying and enhancing their use on large, high resolution tiled displays (display walls). The hardware platform comprises 24 projectors driven by 12 commodity x86 PCs, interconnected via gigabit ethernet. The underlying software platform is based on Linux and Mac OS X, with X Windows and VNC forming the display wall's backend.

A control management interface and tool is developed and implemented to simplify the process of booting the many components of the display wall, supporting hardware and software control of individual projectors and computers. Support for multiple cursors is added to the X Windows-driven interface in an application-agnostic manner by multiplexing the system cursor, providing different users with individually controllable virtual cursors. Interaction is simplified by making new windows appear at the current user's cursor, and by allowing windows to be grouped and moved together.

The management and user interface proposals developed and implemented in this thesis have been deployed in the display wall lab at the Department of Computer Science, University of Tromsø, and are currently in daily use by both teachers and students.

Acknowledgements

Thanks to Otto J. Anshus for his help, support, ideas and suggestions during both my period working with this master thesis and for the work I have carried out in the past as a grad student - his help has been instrumental in motivating me to complete my work. I would also like to thank John Markus Bjørndalen for providing valuable input, assistance and discussing different approaches to my work. I also thank Kai Li and Grant Wallace for providing inspiration for working with the display wall and tackling the multi-input problem. Thanks also go out to the technical staff at the Department of Computer Science in Tromsø (Torfinn Holand, Ken-Arne Jensen, Jon Ivar Kristiansen and Kai-Even Nilssen), and Jan Fuglesteg for his help on different administrative issues along the way. Finally, thanks to my family and my girlfriend for supporting me, and to the many people who have endured the beta stages of the software development, at a time when things weren't working, crashed or just misbehaved.

This thesis has used resources funded in part by NFR project No. 159936/V30, "SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices", and resources from the NFR and University of Tromsø supported project "Display Wall with Compute Cluster".

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the summer of 2004, the Tromsø Display Wall became operational. Since then it has been in use for demonstrations and various educational purposes, mostly with good results. During this time, we have made a number of experiences with its use. We have discovered how the way one works changes radically when a large display surface is available, and have painfully observed how many applications fail to adapt gracefully to such a large resolution.

We have seen how regular users often come to view "turning the display wall on" as something of a black art, resulting in the need for a simple and robust way to bring the display wall up. The demand for this has steadily risen as the display wall as a demonstration tool has grown in popularity.

In working with the display wall, we have also noted how the lack of support for multiple cursors prevents efficient and simultaneous use of the large, shared surface. We have also observed how the placement of windows can become a pain to deal with, as windows pop up everywhere *but* where one would want them to be, and how the need for a simple mechanism for moving more than one window around grows with the display's size.

This master thesis aims to ease and rectify these problems. A user friendly management system for the display wall is developed, with a simple GUI to perform the most common tasks (turning the display wall on and off). Multi-cursor/multi-input support is implemented for the display wall, and the way one works with windows is improved by utilizing window groups as a method for easily moving windows.

The thesis begins by describing the hardware and software that drives the current incarnation of the display wall. It then moves on to deal with two facets of working with the display wall, first describing the design and implementation of the display wall management software, before detailing the development of a multi-cursor, high resolution aware window manager. Finally, some related work is presented, limitations of the developed software considered, and some conclusions are drawn.

1.1 Background

The display wall in Tromsø currently consists of 24 projectors back-projecting an image of a large desktop onto a cinema-sized silver screen surface. Behind the screen a cluster of 12 computers is deployed, with each node driving two projectors. The computers are interconnected with a switched gigabit ethernet network. In addition, one computer (called `ctrl`) provides serial interfaces to control each of the 24 projectors, allowing the projectors' power state and many other settings to be programmatically controlled. The `ctrl` computer is also connected to a camera that looks at the front of the silver screen. The camera allows tasks such as automatic projector-to-node mapping to be performed, in addition to currently unused features related to capturing images for use in automatic projector alignment. A state-of-the-art surround sound system has also been deployed in the lab, though it has not yet been put to use. A schematic of the setup can be seen in Figure 1.2.

On the user-end of the lab, a number of workstations are available. One of them runs the VNC server, and is responsible for many of the other tasks required for "powering" the wall. These will be further detailed later. The workstation also has a gyro-mouse and wireless keyboard attached, in addition to a wired keyboard and mouse.

A PowerMac G5 with dual 30” screens and a tablet computer is also available in the lab. The G5 runs the GUI, developed as part of this thesis, for easily starting and stopping the display wall. The tablet computer gives the user control of one cursor on the display wall, and provides a simple means of drawing figures on the wall. These computers are all connected via gigabit ethernet to the display wall cluster, with the exception of the tablet, which usually communicates over a wireless network.

Figure 1.1: The projectors creating the display wall.

1.1.1 Software

The display wall is powered mainly by computers running some flavor of Linux. The display wall cluster runs RedHat Linux 9[1], while the workstations run Fedora Core 2. In order to display a unified desktop, modified versions of RealVNC’s “VNC for Unix 4.0” [1] [2] server and viewer software are used. Attempts were first made both at using the Princeton-modified Tileviewer as well as the original Tileviewer [3] distribution for creating the virtual desktop, though these efforts failed due to the packages being very unstable and prone to crashes.

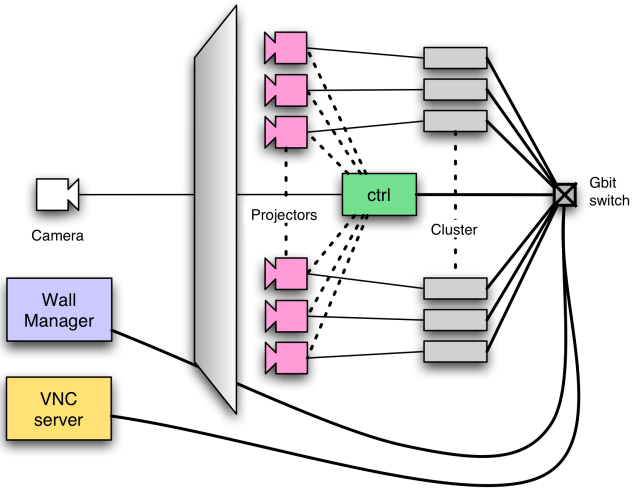

Figure 1.2: A schematic of the hardware/software setup for the display wall.

The modifications made to RealVNC’s server and viewer software mainly aimed at adding support for displaying and serving only parts of the virtual desktop (i.e., display region support), although support for distorting the displayed image using OpenGL was also ported over from the Princeton Tileviewer distribution. This code (known as the “alignment code”) is currently not in use due to major performance problems, probably caused by driver issues.

The projectors are controlled with a simple application that talks to the various serial interfaces connected to the projectors. This program was written by Ken-Arne Jensen, one of the engineers at the Depart-

[1] During the spring semester of 2005, the cluster will be upgraded with a new Linux distribution, a new column of projectors will be added and the cluster will be expanded to 24 nodes.

Figure 1.3: The display wall running the multi-cursor window manager and some applications.

ment of Computer Science in Tromsø. Regular users will not use this application, as this functionality has been embedded in the developed GUI (the “Wall Manager” box in Figure 1.2).

The various display wall nodes are controlled with a master-slave based python script (this script is also invisible to end-users, as it is controlled by the GUI). The details of its development and operation will be treated later in this thesis, as it is part of the display wall management software.

1.1.2 Discussion

The current hardware and software setup for driving and supporting the display wall works well. Although projector failures occasionally occur, the remaining hardware and software has proven stable and dependable. In particular, the VNC-driven backend creating the large virtual desktop works surprisingly well. Performance is naturally a problem, as the VNC server is not easily distributable and thus must run on a single computer, creating a bottleneck. Keeping well over 18 megapixels of content updated[2] also puts a strain on the network and prevents straightforward display of moving content, such as movies or graphics-intensive visualizations or demonstrations.

Some experiences with replacing the VNC backend with a software package called Xdmx will be detailed in section 2.2, though no major focus is placed on this aspect of the display wall in this thesis.

[2]The wall currently has a resolution of 6144x3072 pixels.

Chapter 2

Display wall challenges

This chapter deals with some of the challenges in creating a display wall, presenting some of the existing display walls and contrasting them to the one built in Tromsø. An overview is also given of the software backend driving the display wall in Tromsø, before an alternative backend is considered.

As mentioned in the introduction, a display wall consists of a cluster and a means for displaying output from the cluster nodes. The output can be displayed either by projectors, or using tiled LCD displays; the former is more common, as it allows the individual images to stitch seamlessly together. LCD-based solutions are less popular due to the borders surrounding the individual displays, but have the benefit that they are more space- and heat-efficient.

Although display walls are becoming more and more popular, their basic design has not changed much since their inception. Examples of these are the display wall developed for the iRoom, which is part of the Stanford Interactive Workspaces project [4], and the Scalable Display Wall project at Princeton University[1]. The software solutions are different - the iRoom relies on much custom software for supporting collaboration, whereas the display wall at Princeton is based on the same software as in Tromsø, using VNC for providing a very large desktop. Princeton's display wall cluster, however, runs Windows, and also has some applications that do not use VNC, such as some 3D demonstrations and a parallel MPEG player for displaying movies on the entire display wall.

Using VNC as the backend for creating a large desktop has both advantages and disadvantages. The primary advantage is that it is very simple to set up and configure. Using VNC also avoids the entire problem of synchronized program execution, as all applications running on the display wall are executing on *one* computer. The disadvantage is that VNC does not support any kind of 3D acceleration, and that it needs a lot of bandwidth for keeping the display wall updated.

2.1 The display wall backend

The display wall at the Department of Computer Science in Tromsø is driven using a VNC server running on a user-configurable machine, with viewers configured to run from the cluster. The viewers are started by the management software, detailed in chapter 3. The VNC server's resolution must match that of the display wall, which is 6144x3072 pixels[2]. The server can be running in either 16- or 32-bit mode, depending on the desired performance characteristics. 32-bit mode is preferable when large, detailed, colorful and *static* images or models are to be viewed. 16-bit should be used when many users are interacting concurrently, resulting in many pixels needing to be moved around on screen.

The VNC server and viewer software has been modified to support serving and displaying only a part of the virtual desktop[3]. Code has also been ported from Princeton's VNC viewer software to gain OpenGL distortion support, allowing software alignment of projector edges. This code turned out not to work very well due to some issues with the NVIDIA drivers and/or video accelerators, and is due to this not in use.

[1] http://www.cs.princeton.edu/omnimedia/

[2] 6x4 projectors, with each projector running at a resolution of 1024x768.

[3] These changes are outside the scope of this thesis, and will not be detailed further.

2.2 Alternative display wall backends

During our work with the display wall, we have observed how our use of VNC for driving the wall both works and doesn't work. Achieving a simple and fairly robust setup comes at the cost of functionality and performance. Investigating further, there are some other potential candidates for powering the display wall. The most prominent of these, Xdmx [5], will be described and explored in this section, before naming some other options.

Xdmx, or Distributed, multi-head X, is an entirely different beast compared to VNC. While they both provide a single virtual display running on a single computer, their approaches for sharing that virtual desktop with other computers differ wildly. Where VNC shares pixels, Xdmx shares the actual drawing operations. For a discussion of the benefits and disadvantages of sharing pixels and drawing operations, see [6]. VNC was designed to share a single desktop with a potentially large number of users, whereas Xdmx manages a large desktop that *isn't* shareable with others.

Xdmx works by connecting to a number of "slave" X servers, and then distributing X protocol requests and replies to the slaves. In the display wall lab, this translates to the X servers running on the display wall cluster. When configuring Xdmx, the hostnames of the slaves are specified according to their location in the virtual display (the `wall_ctrl` master script, part of the developed management software, has a method for exporting the current display wall configuration to a configuration that Xdmx can understand).

By sharing drawing operations (or more precisely, distributing X protocol requests), Xdmx can achieve support for some things that VNC can not, OpenGL being the most useful of these extra features. In trying out Xdmx, focus was put mainly on how it handled the things we already do with VNC: Move windows, show, zoom and drag images around, and finally standard browsing of PDFs and the web.

The reason for these, perhaps somewhat limited, tasks, is that they are the tasks the wall is most frequently used for during lectures and demonstrations, and as such need to at least remain at their current performance levels, if not surpass them. Improving other areas with a new backend, while at the same time destroying performance in another area, would only serve to frustrate the current users of the wall. It is essential that the main tasks at least maintain the performance we observe in VNC (even if that performance is far from what should be considered good enough).

Benchmarking GUI performance is a very difficult task, and it is not made simpler by working with performance that is already sub-optimal. The following deliberations are based wholly on subjective opinions from the author and comments solicited from others in the lab while the Xdmx solution was being tested. It was also discovered, by accident, how big an impact small changes to the underlying operating system on the node running the Xdmx server can have on performance.

After using Xdmx for a while, browsing images, moving windows and testing the OpenGL support, it turns out that Xdmx lacks in a few key areas. First, its update mechanism is inferior to that of VNC. VNC manages to reuse the existing pixels it already has available to a much greater extent during dragging operations compared to Xdmx.

Xdmx was also prone to crashing, and didn't adequately support very large windows. As soon as windows got bigger than approximately 4096 pixels wide, pixel artifacts resulted. At this point, window performance was also a lot worse than the VNC counterpart. VNC also has an advantage in that it isn't affected by its viewers exiting or crashing.

The one area where Xdmx really shone, was in OpenGL performance. VNC can not match this, as it doesn't support OpenGL. The initial test of Xdmx' OpenGL support was in running GLgears, a simple OpenGL demo application showing spinning, interlocking cogs in red, green and blue. The demo runs superbly, but as one of the workstations were upgraded with newer Fedora packages, performance dropped by more than 90% when Xdmx was using that node as the front-end. The underlying reason for this drop has not yet been uncovered. GLgears, previously capable of well over 300 frames per second in a window approximately 3000 by 3000 pixels, suddenly stuttered along at around 20-30 fps. It is possible that the upgrades caused Xdmx to run OpenGL unaccelerated, although this doesn't make much sense as the OpenGL commands aren't executed on the Xdmx' front-end node, but rather on the cluster.

Xdmx was also tested by attempting to play the first-person shooter game Enemy Territory on the display wall. While the game started correctly and rendered correctly, it turned out to be difficult to make the game utilize the entire resolution offered by the display wall, in addition to mouse control being extremely sensitive. Due to this, it was difficult to obtain any meaningful performance data.

One of the popular demos on the display wall during the entire past semester has been the rollercoaster, a 3D visualization of a rollercoaster ride on many different tracks. The rollercoaster easily ran, although also this had trouble filling the entire display wall. Some artifacting was visible with large window sizes,

but performance was very good, and clearly comparable to the parallel version of the rollercoaster, which renders directly on the 12 display wall cluster nodes[4].

Trying the modified window manager with Xdmx was a worthwhile exercise, and uncovered a number of issues. The first problem is related to the way the window manager ends up interacting with Xdmx. In cases where it was necessary to kill the window manager, the port used for listening for multi-input was not closed by the operating system, making it impossible to re-launch the window manager; Xdmx itself had to be killed for the port to be released. This is is not a problem when the window manager manages windows served by the Xvnc server.

The performance experienced with using Xdmx and a number of virtual cursors was also far from impressive, and yet again demonstrated how VNC and Xdmx differ in their strategies for updating pixels. In conclusion, despite the stellar OpenGL performance, Xdmx is not yet ready for prime-time use on the display wall. Its problems keeping content refreshed (at least compared to VNC) makes it unsuited for the standard display wall uses. Only when OpenGL support is required, should Xdmx be used, as Xdmx also crashed numerous times during testing, indicating that it is still a beta product. Xdmx may still be very useful on smaller display walls, such as the mini-wall that was recently installed in the lab, consisting of four LCD displays.

2.2.1 Other options

There currently aren't many other solutions available for driving a display wall; most are simply improvements or re-implementations of VNC- or Xdmx-like applications. One such example is NoMachine's NX server and client software [7], which promises to deliver much improved performance over VNC. A free implementation of the server component is available, and testing this application for driving the wall is currently left as future work.

Despite this lack of solutions for driving the wall, the goal of using OpenGL on the display wall is still within reach. Using Chromium and/or WireGL (WireGL has been rolled into Chromium, so in reality one only needs to consider Chromium) [8] [9] for OpenGL and VNC for the remaining tasks is one potential solution. Conducting experiments with Chromium/WireGL is also left as future work.

[4]Note that the "single-threaded" version of the rollercoaster, running under Xdmx, also renders directly on the 12 cluster nodes. The difference is that the rendering commands pass through the network first, distributed by Xdmx, as opposed to being generated locally on each node and synchronized using MPI.

Chapter 3

Management software

This chapter deals with the design and implementation of the management software for the display wall, with the management GUI being detailed in the next chapter. The software consists of a number of discrete components. The components, shown in Figure 3.1, are:

- A GUI for starting and stopping the display wall
- Display wall master/slave software
- Projector control software
- Linux boot scripts on the display wall cluster
- xpattern and ppmsub

The GUI, named *Wall Manager*, interacts with all parts of the system, including the VNC server, to provide a simple means of starting and stopping the wall. As one of the chief goals of this thesis is to make the display wall simple to use, it also aids in discovering where a problem may be rooted in cases where it fails to start the wall.

The master/slave software, called `wall_ctrl`, consists of a master script that sends commands to slave instances running on each computer in the display wall cluster. The commands supported are very versatile, allowing simple tasks such as starting VNC viewers and displaying patterns with various colors and geometric shapes, to more complex tasks for creating the projector-node mapping, or executing arbitrary command-line executables.

The projector control software was written by Ken-Arne Jensen. It can be executed manually from the `ctrl` machine, or indirectly through the Wall Manager application. Unfortunately, the projectors provide no status information, not even rudimentary information indicating whether a given projector is on or off. This creates some problems for the management software, as there is no way to tell whether a given projector has started successfully (see section 4.5).

The Linux boot scripts serve to bring the cluster into a known state, by re-defining run level 5 on the node from starting an X login session, to running the slave script instead. The slave script, in turn, brings up the additional necessary components (an X Window server and a black blanking window). Finally, the xpattern application allows custom display of various colors and geometric shapes, and the ppmsub application allows for fast subtraction of one PPM image from another. The xpattern application is used when determining the projector-to-host mapping, for displaying a black blanking window, manual color calibration and warp configuration[1]. The ppmsub application is used for speeding up the process of subtracting a background image from a foreground image during the various calibration phases. The implementation and functionality of xpattern and ppmsub, being as simple as they are, will not be further discussed in this thesis; their source code is included in the appendix, however (see pages 81 and 42).

3.1 Requirements

This and the following sections detail the requirements, design and implementation of the `wall_ctrl` management scripts. The `wall_ctrl` software is divided into two parts: A master script and a number of

[1] The display wall no longer uses the warp configuration, as the performance of the wall while warping was in use was terrible.

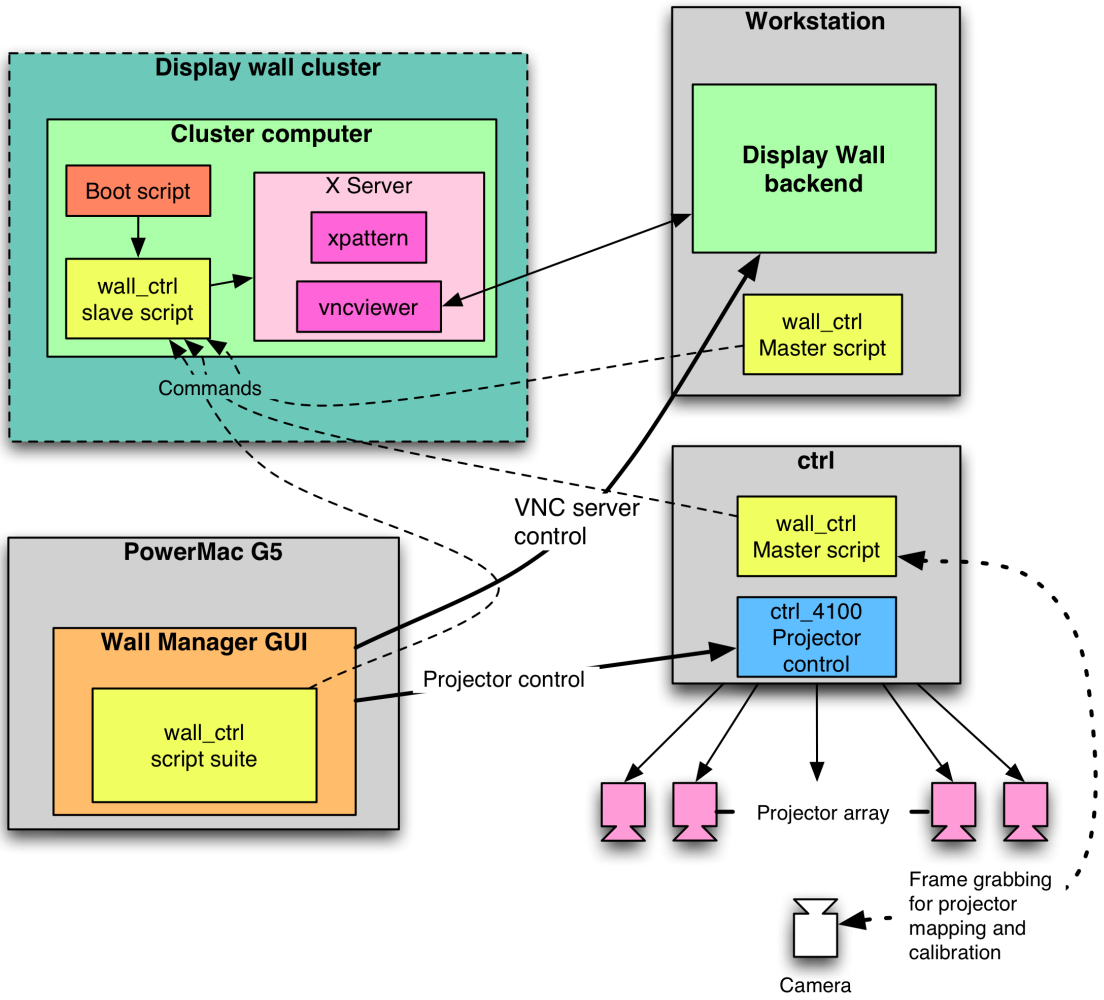

Figure 3.1: An overview of the wall management components.

slave scripts. Note that an implicit assumption regarding the control of the display wall has already been made, namely that the software will be organized according to the master-slave pattern. The reason for this is that it is the most intuitive organization for this kind of software - anything else would needlessly complicate the design and implementation. A slave needs to be able to do the following:

1. Start the X Window System on the node
2. Receive commands from the network
3. Perform various actions in response to commands from the master
4. Reload configuration and code on-the-fly

Starting X Windows is important as X windows needs to run directly, with no login box and with full access from any node (i.e., no restrictions on who may display windows to the node). Receiving commands is essential for controlling a slave, and the actions performed will naturally depend on the commands the slave receives. Finally, the ability to reload configuration and code at runtime is important for the following reasons. First, reloading the configuration is necessary whenever the master changes the configuration. Although this doesn't happen often, restarting the slave only to reload the configuration results in wasted time. Second, reloading the code that drives the slave is necessary both during testing/bug fixing, and in

cases were additional features need to be incorporated into the slave[2] without manually restarting the slave each time.

The master must be able to instruct the slaves to perform tasks over the network, and receive status messages from the slaves. The master does *not* need to be running continuously, it only needs to run for the duration it takes to relay a command to the slaves, and receive any replies.

A common requirement for both the master and slaves, is that they should be implemented in a platform independent way. This is required in order to allow the underlying platform to change, and to allow the slaves to be controlled from any platform - be it a PDA, a Macintosh or an x86 computer. Python is an interpreted language that fulfills this requirement, and has been chosen as the implementation language for the master and slaves for this reason.

3.2 Design

The master script is designed to simply relay commands to the slave scripts, running on the cluster. As such, it shouldn't run continuously, but only on demand. The slaves, however, need to run continuously, and be sufficiently stable to avoid causing problems with the operation of the display wall. The following design descriptions are based on the fact that Python is to be used as the implementation language.

The network operation of the master/slave scripts is designed around using multicast for sending requests and replies. While multicast in general is not reliable, it works well on the LAN in the display wall lab as long as the network is not completely saturated. Also, as no operation performed by the master has devastating consequences should a packet not make it to one or more of the cluster nodes, it is safe for the user to simply re-issue the request if it fails.

The messages exchanged should also be human-readable, in order to simplify the implementation and ease debugging. Considering that the code is to be written in Python, Python dictionaries are a simple way to implement the message structure, while also being easily readable. This also has the benefit of avoiding problems with marshaling/unmarshaling (specifically, converting to and from network byte order), as all integers/floats are expressed as text strings. While this may be slightly less efficient than packing the information tightly, the loss of efficiency isn't a big issue due to the small number of messages being exchanged. The only real problem with this approach is that as the dictionaries get larger, problems with the lack of multicast reliability may arise, and messages may begin to stretch the size limit of individual multicast datagrams. This is an indication of an inherent scalability problem with the multicast approach - as more projectors/hosts are added, the size of the datagrams will increase. Despite this potential problem, the decision to use multicast remains, due to the reduced complexity of the implementation.

3.3 Implementation

The `wall_ctrl` software has been implemented in Python. In addition to the above-mentioned platform independence requirements, a Python-implementation made it simple to get the wall up and running, which was a big priority during the initial development of the package.

The implementation consists of a master-script and a slave-script. The master-script's main responsibility is to take commands given by the user, and broadcast them to the slaves running on the same local network. The master is also responsible for creating the initial wall configuration, and updating the configuration as necessary. The configuration may also be updated by hand.

The master sends commands to the slaves using IP multicast, at the cost of some potential reliability problems. The justification for this lies primarily in the simplified implementation, as opposed to implementing reliable multicast. Experiences from the past semester have shown that packet loss almost never occurs, yielding small potential benefits from such an effort, in addition to complicating the code base a lot.

The multicast address used by the master and slaves for communication is 224.10.20.30, on port 10101. The choice of port number was random, with a slight bias towards an interesting-looking port number. The messages exchanged follow a simple format, using serialized Python dictionaries both for requests and replies. The general form of these dictionaries are shown below, with serialization and de-serialization as the next two steps:

```
# Create a message of type <type> with <params>:
```

[2] This feature has proven extremely useful on several occasions, usually for adding features.

```
message    = {"type":<type>, "params":<params>}
# Serialize it:
data       = repr(message)
mysock.send(data, ..)

# On the other end:
data       = mysock.recv(..)
message    = eval(data)
# message now contains our dictionary.
```

`<type>` is expected to be a string describing the message type. `<params>` is also a dictionary, which can either be empty, or contain parameters for the message type in question. `repr` and `eval` are Python built-ins for marshaling different Python objects.

The following are the currently supported message types sent from the master, along with their interpretation. The parameters are described further below.

`identify`	Requests that every slave identify itself by hostname and number of available projectors. This can be thought of as a "ping" request.
`reload_config`	Requests that every slave reloads the configuration file, updating their own view of the world.
`reset`	Requests that every slave resets their state completely. This involves completely reloading their own code from disk.
`execute`	Makes the slaves execute a given command.
`terminate`	Terminates a command previously executed with the `execute` directive.
`die`	Causes the slaves to exit.
`set_projector_state`	A broad command instructing a projector to enter a certain state.

Table 3.1: Messages sent by the master

The `identify` request is used initially (before any configuration is present) by the master to build what is essentially a random host-to-projector mapping using the available hosts (i.e., the hosts replying to the identify command), and does not require any parameters. After this initial mapping has been created, the master will issue a series of directed `set_projector_state` commands, instructing the owner of a given projector to display a small, white square in the projector's center. For every projector, an image is captured, in addition to a background image where all projectors are turned on, but displaying black. The background is subtracted from the captured images using the ppmsub utility, before an analysis of the resulting images is performed. This analysis basically consists of looking for the first pixel whose intensity exceeds a given threshold, and note its location in the image.

The entire analysis process yields a list sorted by projector location from top-left to bottom-right, indexed by projector ID. The projector IDs are then re-assigned to the correct host based on the knowledge gained when randomly assigning the projector IDs, and then grabbing an image of the projector corresponding to that ID, yielding a correct projector-to-host mapping. Once this mapping is complete, the master instructs the slaves to reload their configuration. The `identify` request is also used by the Wall Manager to discover the current state of the cluster.

Due to the use of multicast, the user is not guaranteed that a request from the master is performed by all (or any) of the slaves, as multicast packets may be dropped. This applies in particular to the `identify`, `reload_config` and `reset` commands. Losing such a command may pose a problem for casual users. However, since these commands are usually executed by administrators, the actual users of these commands will be people already familiar with the wall. Verifying that the command actually completed can then be done manually, if necessary.

The `set_projector_state` request is the most complicated request, featuring a number of different states, and a fairly complicated parameter structure. It has two required parameters: The requested state, and the set of projectors to which the state should be applied. A slave will decide whether to execute the request or not by determining if the projector ID(s) it owns is part of the set given in the message. The dictionary keys in the parameter structure for these two parameters are `state` and `projectors`. The `projectors` key is expected to resolve into a list of numbers, each identifying one projector ID that the command applies to.

The following table lists the different states a projector can be placed in:

`identify_image`	Displays a white square 100 pixels wide on the projector.
`calib_image_horiz`	Displays an image with horizontal lines, as part of the automatic projector alignment process (out of scope for this thesis).
`calib_image_vert`	Same as above, except vertical lines are displayed.
`calib_image_mesh`	Displays a mesh; essentially the two states above at once.
`calib_image_rect`	Displays a rectangle encompassing the borders of the projector. This state is used when the projectors are being manually aligned (i.e., by physically moving the projectors around).

`calib_image_white`
`calib_image_red`
`calib_image_green`
`calib_image_blue`
	Displays a solid white, red, green or blue color.

`rgb`	Displays a given RGB color on the projector. The color is part of the parameter dictionary, stored under the keys `red`, `green`, `blue`.
`on`	Turns the projector "on" - that is, starts a VNC viewer covering the projector.
`off`	Displays all black on the projector.
`kill`	Kills any projector-state binary (VNC viewer, xpattern), displaying the "raw" X 11 desktop on the projector.

Table 3.2: Messages sent by the slaves

The `execute` and `terminate` commands are used to execute arbitrary applications on the nodes, either for displaying to the display wall, or for producing debug output that can be read from a slave's standard out. The parameters to `execute` is a dictionary containing the name of an executable and the arguments to be passed to it.

The `die` command will only be performed by slaves whose hostname matches the `hostname` parameter, or if the hostname is "all". The remaining commands display various patterns and colors on the projector(s) in question.

3.3.1 Slave implementation

This section describes the implementation of the slaves, including how a slave responds to the requests listed above. When a slave starts, it begins by setting up a multicast socket for receiving commands, before it reads the current configuration and examines its environment. The environment examination consists mainly of checking for a running instance of X Windows. If no instance is found, the slave will start an instance of X Windows[3]. Once this is done, the slave will set its projectors to state "off". Note that this state is not related to the *power state* of the projectors, only what the projector displays once it has been turned on.

Checking for X11 is done by executing the `ps` command, and parsing its output, looking for an executable named "X" or "X11"[4]. If no X server is detected, it is started by executing `startx`, and then waiting for 40 seconds - the approximate time it takes for the X server to start completely. The slave then proceeds to export the `DISPLAY` environment variable to any applications it will later execute (such as VNC viewers or xpattern), and then executes some applications for configuring the X environment. This configuration consists of disabling energy saving (so that the projectors won't be blanked by the X server), disabling the terminal bell, disable the X server's screensaver and running `xhost +`, allowing everyone to display applications to the server.

Once this has been done, the slave enters a loop where it waits for incoming traffic on the multicast socket, and responds to requests if necessary. The slave uses the xpattern application to set the projectors to the various `calib_image_*`, `rgb` and `off` states, and `vncviewer` for the `on` state.

In response to an `identify` request, the slave will send an `identity` message, containing the hostname of the node and number of projectors as parameters. A `reset` request is processed by performing an

[3]Note that this requires that special permissions are enabled on the host computer, as starting X11 usually requires either console or root access.

[4]In retrospect, a much simpler technique for accomplishing this exists, by simply attempting to open a TCP connection to port 6000, similar to the probing phase of the Wall Manager implementation (see section 4.3.2).

`execl` call, re-executing the Python interpreter with the slave script as the argument. This is sufficient to reload the slave's code, and as a side-effect also reloads the slave's configuration.

3.3.2 Configuration

The display wall configuration is stored as a simple Python source file. It is loaded by the scripts, incorporating the variables named in the file into Python's global namespace. These variables can then be accessed by the scripts whenever details about the configuration is required. Both the master and slaves load the configuration using the following method:

```
def read_config(self):
    # Get the location where this script is stored
    folder, ourname = os.path.split(__file__)
    # Append the remaining path to the config file path
    conf_file       = os.path.join(folder, "conf/"+wall_common.wall_config_file)
    # Open the file, read it and close it
    cf              = open(conf_file, "r")
    data            = cf.read()
    cf.close()
    # Instruct python to parse and execute the data we just read
    exec(data, globals())
```

The current configuration used by the display wall is shown below:

```
wall = [6, 4]
proj_ctrl_hostname = "ctrl"
resolution_pr_projector = [1024, 768]
vnchost = "wks1:1"
mapping =
{'d045.Cluster.cs.UiT.No': [14, 15], 'd039.Cluster.cs.UiT.No': [2, 3],
 'd040.Cluster.cs.UiT.No': [4, 5], 'd043.Cluster.cs.UiT.No': [10, 11],
 'd048.Cluster.cs.UiT.No': [20, 21], 'd047.Cluster.cs.UiT.No': [18, 19],
 'd041.Cluster.cs.UiT.No': [6, 7], 'd044.Cluster.cs.UiT.No': [12, 13],
 'd046.Cluster.cs.UiT.No': [16, 17], 'd038.Cluster.cs.UiT.No': [0, 1],
 'd042.Cluster.cs.UiT.No': [8, 9], 'd049.Cluster.cs.UiT.No': [22, 23]
}
warp = {}
```

The `wall` variable defines the geometry of the wall: 6 projectors along the X axis, and 4 projectors along the Y axis, for a total of 24 projectors. `proj_ctrl_hostname` sets the name of the host controlling the projectors (this variable is primarily used by the Wall Manager application), `vnchost` contains the hostname and screen number of the machine running the VNC server, and `resolution_pr_projector` describes the pixel resolution used by each projector. The `mapping` variable describes the projector-to-host mapping (or vice versa), with projectors named sequentially from 0, starting with the upper-left projector, moving right. See Figure 3.2 for a small projector-to-host mapping example. The `warp` variable (which is not shown here in its entirety), consists of the corner-points used to align the VNC viewers, emulating hardware projector alignment.

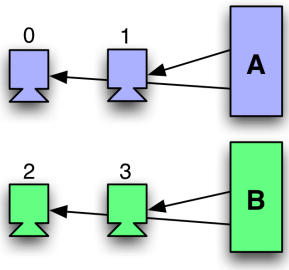

Figure 3.2: A simple example of projector-to-host mapping. Node A is connected (and thus, mapped) to projectors 0 and 1, and node B is connected to projectors 2 and 3.

3.4 Discussion

The current incarnation of the `wall_ctrl` script suite was the first component of the management software to reach a stable state, having been in use for most of the past semester. It has also gone through a number of minor changes, bug fixes and feature enhancements, where the reset-functionality has proven itself very valuable. The architecture is also versatile enough to easily add new commands to both slave and master, something which has been done on numerous occasions (the `rgb` command is one of these added commands, after it was discovered that the white, red, green and blue commands were not sufficient to achieve good color calibration of the display wall).

Chapter 4

Wall Manager

This chapter deals with the design and implementation of the GUI for controlling the display wall. The Wall Manager GUI has as its main goal the ability to easily turn the display wall on and off. This means that it must do more than just become a pretty front-end for the `wall_ctrl` master script, as starting the wall also entails turning the projectors on and making sure a VNC server is available to create the display wall's virtual desktop.

4.1 Requirements

The Wall Manager application has the following requirements. First, it must be able to interface with the master script, in order to instruct the slaves to perform the following common tasks: Start/stop VNC viewers, display a user-selected color or pattern on the entire display wall and check the slave status. This requirement stems from the desire to re-use the existing Python code, avoiding a duplication of coding effort.

Second, Wall Manager must be able to start a VNC server remotely, and turn the projectors on and off. This also requires that Wall Manager is able to login to the hosts that are to provide this functionality, and execute commands in a well-defined environment. An interface for controlling individual projectors is also necessary, as communication with the projectors occasionally fails[1].

Wall Manager must also be as independent from the hardware configuration of the display wall as possible. It must be able to reflect changes in the number of projectors and computers in use in its interface. Finally, Wall Manager must be able to detect which components potentially can be the cause of a failure, in cases where the display wall fails to start, providing the user with suggestions as to where the error might be located. Wall Manager should consolidate all these functions so that it, in essence, can provide the user with a single-click interface for starting and stopping the display wall.

4.2 Design

The Wall Manager's GUI was designed in Apple's Interface Builder, an application for creating user interfaces on the Mac OS X platform. This locks the Wall Manager to the Apple Macintosh platform. The main reason for this decision was based on the fact that creating and prototyping GUIs on Mac OS X is very simple, meaning that development time could be spent on creating functionality, rather than tweaking pixel offsets for the various GUI elements. Also, the display wall lab already has a top-of-the-line Macintosh installed, and not putting it to use seemed like a great shame. A screenshot of Wall Manager can be seen in Figure 4.1.

The user interface is laid out in two parts - a "simple" view, and a "detailed" view. The simple view contains a field for authenticating the user (which is necessary for logging into the computers controlling the projectors and VNC server), as well as three buttons titled "Start", "Stop" and "Open Terminal". The first two buttons are used for respectively starting and stopping the display wall, whereas the last button is used for opening up a standard terminal. This is useful for cases where the user needs to do tasks that are

[1] Note that this failure in communication is *not* due to the use of multicast for master/slave communication, but a problem with the serial interface on the computer connected to the projectors.

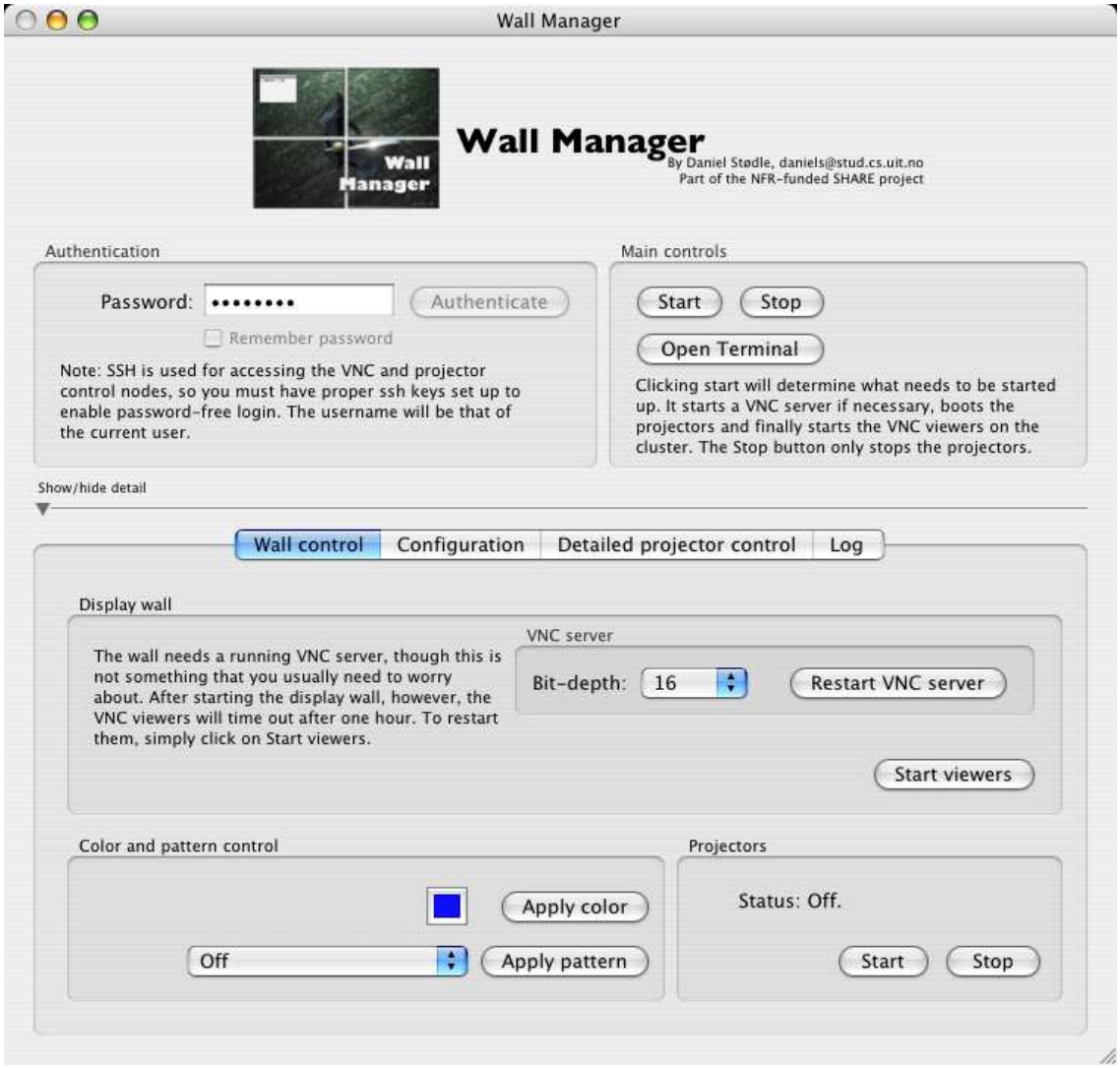

Figure 4.1: A screenshot of the Wall Manager application.

not supported by Wall Manager, and thus may require the use of a terminal. It is also a convenience for users that are not familiar with Mac OS X, and as such may not know how to open a terminal.

The detailed view consists of a number of tabs. The tabs are: A detailed control tab, a tab for viewing the current wall configuration, a tab for detailed projector control and finally a tab containing a log for troubleshooting. The first tab offers more control over the wall, allowing the user to start or restart the VNC server and VNC viewers. It also allows the user to set a custom color on the entire display wall (for color calibrating purposes) or set one of a number of predefined patterns on the display wall (for manually aligning the projectors). Finally, the user can start or stop the projectors.

The configuration tab shows the current configuration of the display wall: The current resolution per projector, resolution of the entire display wall, projector geometry, the number of cluster nodes and the hostnames of the computers running the VNC server and controlling the projector.

The detailed projector control tab allows the user to "manually" turn given projectors on or off, in cases where the automatic projector control fails. This unfortunately happens quite often, and is related to problems with the serial communications link between the projector controlling computer, and the different projectors. The buttons representing the different projectors should be created dynamically based on the wall's configuration.

The log tab contains a log over actions taken by the Wall Manager. The log includes generic information about events, and also details about where failures in the system might be located.

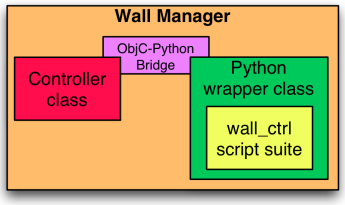

Figure 4.2: Diagram of the Wall Manager design.

The design for the Wall Manager's code has been kept as simple as possible, as can be seen in Figure 4.2. In essence, it consists of two parts. The first part is a controller class written in Objective-C, responsible for keeping the user interface up-to-date and for responding to the user's actions (clicks, menu selections, etc.). The second part is a wrapper-class written in Python, that is responsible for calling through to the various functions in the `wall_ctrl` script. The wrapper-class should be written according to the calling conventions established by the PyObjC bridge[2], allowing it to be treated as a proper Objective-C class, and accessible from the controller class.

4.3 Implementation

Wall Manager was implemented in Objective-C, utilizing PyObjC 1.1 [10] to communicate between Python and Objective-C[3]. PyObjC is an open source project that makes Python classes available as Objective-C classes, and vice versa. This module allows the `wall_ctrl` master script to be effectively reused from within the Objective-C/Cocoa-based GUI.

While the bridge between Python and Objective-C made implementing the GUI simpler, it wasn't by far enough to complete the GUI. As the GUI also needed to start a VNC server and control the projectors, code was required that would allow these tasks to complete. The choice quickly fell upon using the existing secure shell infrastructure to execute the commands. The difficulty here, however, turned out to be getting `ssh` to successfully authenticate the current user (or, rather, supply a password to `ssh` for authentication). The developers of `ssh` have made this task very difficult (or at least cumbersome) to perform. In the end, the following approach was settled on, and then implemented in Python:

1. Start an instance of `ssh-agent`, exporting the resulting environment variables to the GUI's environment.
2. Export three additional environment variables: `SSH_ASKPASS`, `DISPLAY` and `ASKPASS_PASSWORD`. The two first environment variables are required by `ssh` to solicit special behaviour from its password entry mechanism. The last variable is justified below.
3. Start an instance of `ssh-add`, connecting pipes to its input and output (this causes `ssh-add` to use the output from the executable in `$SSH_ASKPASS` for authentication).
   (a) `ssh-add` starts the `askpass` executable, developed as part of Wall Manager.
   (b) `askpass` reads the environment variable `$ASKPASS_PASSWORD`, and outputs it to standard out. The reason for using an environment variable here is that `ssh` does not allow arguments to be passed to the `$SSH_ASKPASS` executable.
   (c) If the password is correct, `ssh-add` returns immediately. If the password is incorrect, `ssh-add` hangs.
4. Should the authentication take longer than approximately one second, Wall Manager assumes that the authentication failed, and kills the `ssh-add` process[4], before informing the user what happened and requiring the user to re-attempt the authentication. Otherwise, authentication has succeeded.

[2] An introduction to PyObjC can be found here: http://pyobjc.sourceforge.net/doc/intro.php
[3] Towards end of work with this thesis, PyObjC 1.2 was released - compatibility with this release has not been verified.
[4] This shouldn't be necessary, as ssh-add *should* exit with a non-zero status code on failure. For some reason, it doesn't.

5. The `$SSH_ASKPASS` variable is reset, to prevent leaking the password[5].

Assuming that authentication succeeded, `ssh` can now be used for executing commands on remote hosts where the key is authorized for password-free login (i.e., it's public component is stored in `/.ssh/authorized_keys2`). The setup in the display wall lab is such that this works without problems, due in part to the underlying shared filesystem. When the user quits Wall Manager, the `ssh-agent` instance is terminated.

4.3.1 Controller class implementation

The controller class, named `wall_manager`, is implemented in the standard Cocoa fashion. It has a number of action methods, each the "target" for one or more buttons or menu items in the user interface. These connections are established in Interface Builder. It also has a number of "outlets", which are connected to various parts in the user interface, allowing the controller to set the properties of these objects. An example of this is the outlet `depth_menu`, which gives the controller object access to the popup-menu used to select the bit depth of the VNC server.

The controller class mainly consists of these action methods, along with logic logging events and producing user-friendly error messages when failures occur (for instance, trying to start a VNC server without being authenticated). The most complicated method is probably the `start_everything` method, which itself relies on a number of function in the Python wrapper-script. See below for a detailed description of how this function operates.

4.3.2 Probing the system

Before Wall Manager attempts to turn the display wall on, it will probe the system. It is also possible to manually probe the system, by clicking the "Probe system" button, accessible from the Log tab. The probing phase performs the following actions (in order):

1. Check that we are authenticated.
2. Check if we can ssh to the computer controlling the VNC server.
3. Check for a running VNC server.
4. Check for ssh access to the computer controlling the projectors.
5. Check the status of each individual cluster node using the master script.

If any of the above tests fail, a note is made in the log, and the user is informed. The implementation of the tests is fairly straight-forward; checking for ssh access simply involves opening a connection to the machine in question on port 22. Similarly, the presence of a VNC server is easily checked by opening a connection on port 6000 + the VNC server's display number (usually 1).

The cluster status is verified using multicast messages sent in response to a "ping" message from the master script. If no reply is received from a given computer in the cluster within 3 seconds, the slave software on that computer is assumed to be down. The most straightforward way to fix this problem is to simply reboot the node in question, although it is also possible to restart the slave software manually[6]. Also, it is possible that the lack of a response is a false negative, in that the response packet has been dropped by the network. This is why Wall Manager offers to try starting the wall even if it doesn't receive a reply from each cluster node.

4.3.3 Booting the display wall

After having probed the system, Wall Manager knows which components it needs to start in order to bring the display wall into the "on" state. It proceeds to start a VNC server (if necessary), before starting the projectors (which are assumed to be off when Wall Manager starts). Wall Manager uses `ssh` to perform

[5] Before this step was added, password leakage was possible. This was due to the fact that the terminal application inherits Wall Manager's environment, and can display it if launched after successful authentication.

[6] The Wall Manager does not currently support remote rebooting of either the entire cluster or individual cluster nodes, as this requires root access.

these two tasks, instructing it to log in to the host, perform a command and log out again. The following listing illustrates how the projectors are started using ssh:

```
def startProjectors_whichProjector_(self, projhost, which_proj):
    print "Will attempt to start projectors.."
    if self.has_authenticated:
        cmd = "cd wallctrl/bin/ ; ./p_start.sh"
        if which_proj != None:
            cmd += " "+which_proj
        os.spawnvpe(os.P_WAIT, "ssh", ["ssh", projhost, cmd], os.environ)
    else:
        print "We are not authenticated yet."
```

The VNC server is started by first setting the `PATH` environment variable, and then running the "vnc-server" script (assumed to be located in `/wallctrl/bin/`, see section 4.6) supplying geometry and bit depth as appropriate for the current hardware configuration. The projectors are started using a small script called `p_start.sh`.

After starting the projectors, Wall Manager proceeds to call upon the master script (using the PyObjC bridge), which in turn instructs the cluster nodes to start the VNC viewers. This finishes the display wall boot sequence.

Stopping the display wall currently consists only of stopping the projectors. The remaining software can be left running; in fact, it has been our experience that the user rarely wants to kill the VNC server, as it may be displaying web sites, pictures or documents that the user wants to return to next time the wall is in use. The VNC viewers currently time out after one hour of use; this behaviour removes the necessity of stopping them. Note, however, that the user can directly control the viewers in the detailed Wall Manager view.

4.4 Using the wall management software

Over the course of the past semester, the wall management code has been extensively tested through real use by multiple people involved with the display wall, including both teachers and students. It has been well received, although there has to some extent been the feeling of there being a "black art" to getting the display wall up and running. The author of this thesis was previously often asked to start the display wall, for others to give lectures or demonstrations. With the GUI in place, this task has been reduced from taking minutes to being done in seconds, and even better, can be performed not only by a select few, but by anyone wanting to use the wall. The GUI has been in use for approximately a month at the time this thesis is delivered, and is currently in active use by the students in one of the courses offered by the University.

The Wall Manager application has also been tested to verify that it indeed reports the various failure scenarios correctly, and gives meaningful error messages to the users. This testing consisted of in turn disabling various components, and seeing how Wall Manager responded to and reported the conditions. The results from these tests show that Wall Manager successfully isolates the errors it is supposed to recognize. In conclusion, the Wall Manager application has proven itself to be a stable and worthy addition to the current crop of applications related to the wall, significantly lowering the bar for casual users wanting to use the wall.

4.5 Discussion

Development of the Wall Manager GUI is an excellent example of applying the end-to-end principle [11] to the particular task of developing a control interface for the display wall. The Wall Manager is capable of performing a fixed set of tasks, but in the end, it is the user who will be able to see if the tasks are performed correctly. Only a very large amount of engineering can, for instance, overcome the problem of detecting whether a projector is on or off, by for instance integrating the camera present in the display wall lab with the Wall Manager GUI, to accurately report projector state. Such a solution is still bound to fail in some circumstances, whereas the user operating the GUI will have detailed knowledge of projector state simply by looking at the display wall.

Similar arguments apply to issues such as deciding what to do if some part of the system ends up being unresponsive. Applying the end-to-end argument again, it is clear that no matter how a failure is detected,

and whatever complicated schemes can be devised to rectify the error, in the end, it is simpler for the user to do something about it by herself, rather than scripting many "tailored" solutions into the Wall Manager application.

Wall Manager fulfills its purpose of providing users with a one-click way of starting the wall, but lacks one desirable piece of functionality. After starting the wall, users need a way of controlling it. This currently entails logging in to one of the Linux workstations and executing either `x2x` or `x2wmx` (see section 5.4.1). These applications are responsible for forwarding cursor and keyboard events, enabling users to interact with the display wall. Adding "one-click" support for this as well should be a simple matter, but is limited since the PowerMac is not attached to a remote keyboard and mouse[7]. Controlling the wall from the PowerMac would thus become a somewhat tedious task, especially considering that the PowerMac is positioned exactly opposite to the display wall. Apart from this one short-coming, Wall Manager fulfills its requirements.

Future versions of Wall Manager should support the above-mentioned cursor forwarding mechanism, as well as integrate support for performing administrator-level operations on the cluster. An example of such an operation is to remotely reset the runlevel of each computer in the cluster, effectively restarting the cluster software (slave scripts, X server, etc). This will require more work on the authentication code, since the cluster currently doesn't support ssh-access.

4.6 Deployment

This section describes how the various components have been deployed and the interactions between them, in essence giving a blueprint for a possible software foundation for future display walls. The figures given in the introduction to this thesis and in chapter 3, Figures 1.2 and 3.1, visualize the deployment. The overall software setup is as follows:

- An instance of the slave script runs on each node in the display wall cluster.
- The master script can run from any computer on the LAN, but must run from the `ctrl` computer to perform tasks related to frame grabbing, as these tasks require direct access to the camera attached to the `ctrl` computer.
- A VNC server runs on the `wks1` computer.
- The GUI runs on the PowerMac G5.

When deploying the slave scripts on the cluster, the following directory structure is expected:

```
~/wallctrl/
~/wallctrl/conf/
~/wallctrl/bin/
```

This directory structure must exist in the home directory of any user wanting to use the Wall Manager GUI for controlling the display wall. The reason for this requirement is that Wall Manager needs to know where its supporting scripts and binaries are located. If the user does not intend to use the GUI, the `wallctrl` directory can be moved and renamed at will.

The source code for both the slave and master scripts, along with some supporting code, is located in the `wallctrl` directory. The `conf` directory holds the configuration for the display wall (see section 3.3.2), and the `bin` directory holds x86 copies of the binaries important for driving the wall. The binaries are the modified VNC server and viewer (`Xvnc`, `vncserver` and `vncviewer`), the xpattern executable, the `ctrl_4100` application for controlling the projectors, a small tool for subtracting ppm-images from each other called `ppmsub` (used to speed up image analysis), as well as a couple of scripts utilizing the `ctrl_4100` executable.

The cluster nodes have been configured to execute the slave script when entering run level 5 (which is the default run level on the cluster nodes), which in turn takes care of starting the X server. The scripts are stored under a special user, with the password for the VNC server stored in that user's `.vnc/` directory. The "password" is simply a copy of the VNC server's password file (which does not store a plaintext password), allowing password-free login to the VNC server.

[7] The mouse is in fact wireless, but has a very limited range.

Whenever the VNC server is started (usually on `wks1`, though this is configurable), the modified VNC server located in the above directory structure, is used. The `Xdmx` alternative is also run from `wks1`, although not in an automated fashion, as it currently needs to be started manually.

The Wall Manager application has been installed on the Power Macintosh, and a user with correctly configured SSH keys has been set up to allow others to easily use the display wall. This user shares the same login as the standard user for display wall usage in the rest of the lab.

Deploying the various pieces of the software turns out to be a fairly simple task, as the cluster and workstations all share a filesystem where the users' home directories are stored. Setting everything up from scratch thus merely entails copying the `wallctrl` directory into place, configuring SSH keys and setting up the proper permissions and boot scripts on the cluster nodes.

Chapter 5

Implications of large, high resolution displays on basic user interface abstractions

This chapter deals with extending an existing window manager for X Windows, aiming to better support its use on the display wall. During the past semester, we have experienced how using the display wall occasionally can be a painful experience, either due to lack of performance, functionality or simply that the applications being run on the wall don't scale well to the resolution offered by it.

One illuminating example of this is the placement of windows. While most window managers offer preferences to guide the placement of new windows on screen, these preferences are rarely suited for use on the display wall. In addition, they don't seem to apply to every window, which ends up frustrating users instead of helping them. Many programs, for instance, ask users if the document they are working on should be saved. The standard way of doing this is to open a dialog, containing Save and Discard buttons. Now, this dialog tends to pop up in the center of the screen. This is very nice for a one- or dual-monitor setup, where you will be likely to notice the dialog quickly, and make a decision.

On the display wall, however, a dialog popping up in the center of the virtual display, can be "miles" from where the user's current focus is. At the very least, the user will be annoyed at having to move the cursor from her current area of focus, in order to dismiss the dialog. At worst, the user won't notice the dialog for some time (it might even pop up under another window), and begin wondering why the application has stopped responding to clicks or keyboard input.

A second issue revolves around the concept of multiple cursors. A display wall can be an excellent tool for collaboration, but its usefulness as a collaborative tool can, in some respects, be proportional to the number of concurrent users it supports. Naturally, the wall can support as many viewers as can be in the room at the same time. The same can unfortunately not be said for interaction.

User interfaces today are in a large part designed and implemented with a single user in mind. While multi-user interfaces are becoming more common (for instance in applications such as Microsoft NetMeeting [12], where application sharing between multiple users is possible), the fundamental fact is that these applications were never designed with more than one (simultaneous) user in mind. This aspect makes it evident that any attempt at bringing multiple cursors to the wall has to support legacy applications; any effort that only supports applications specifically written or modified to support multiple cursors, will at best see only limited use.

5.1 Requirements

As a basis for experimentation with different "display wall friendly" user interface concepts, it was decided that an existing, open-source window manager should be used, as developing a full-fledged window manager from scratch would be too much work, and outside the scope of the thesis. Our experiences with the display wall before work on the thesis started lead the author to believe that the following features could be useful:

- Multi-cursor support

- Moving multiple windows simultaneously
- Make new windows appear where the user's cursor is
- Larger cursors
- Larger window borders

The motivation for the different features vary. Multi-cursor support is clearly motivated by the fact that it is desirable to have more than one person interact with the display wall simultaneously. Moving many windows at once might not seem like an important feature, but the previous semester has shown how it would be very useful to move a group of (possibly related) windows from one area of the display to another. The alternative is to move the windows one by one, a task that quickly becomes tedious to perform on the wall; both due to performance problems, and due to the long pixel distances involved. Moving one window is okay, moving four or five is tedious and difficult.

Having new windows appear where the current user's cursor is, appears to be the simplest solution to the problem of windows appearing a long pixel-distance away from the user's current focus. It can be viewed as a variation of the technique used in some operating systems, where the cursor is moved to the location of the default button in a dialog. Here the technique is turned around, and the window is moved under the cursor instead.

A larger cursor is necessary because the standard cursor is only 16x16 pixels large, a practically invisible quantity on a large display. Larger window borders are useful because hitting small/thin targets on the display wall is difficult - much of the time spent resizing a window is often spent aiming the cursor at the window's "sweet spot". The aim is to make these regions of the windows easier to hit.

A system for dynamically magnifying areas of the display wall would also be a nice addition to the usability of the display wall. The best approach here would probably either be based on pixel-magnification or using the projector hardware to zoom (this part of the projectors is programmable via the serial interface, so this is not impossible). Developing this functionality has been left for future work.

Finally, a surprisingly simple enhancement that very much improves the quality of work with the display wall is to simply set the desktop background to black. This drastically reduces the strain on the eye compared to using a lighter background, and gives the added benefit that the seams between the different projectors become more difficult to spot. Black has become the default desktop background in all the setups utilizing the wall.

5.2 Design

This section describes the design of the two main components added to the window manager: Multi-cursor support and window grouping. The other modifications either appear as part of these components, or as small patches elsewhere in the window manager's source code and are, as such, not "designed".

The overall view of this part of the system is shown in Figure 5.1.

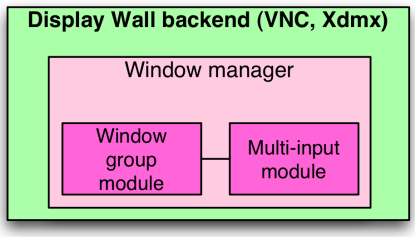

Figure 5.1: Components in the window manager.

5.2.1 Multi-cursor design

In order to grasp the solution space in which a multi-cursor solution can operate, it is necessary to know how input is handled in existing graphical user interfaces, such as those found in Mac OS X, Windows and Linux. These systems traditionally support only one user providing keyboard and mouse input. The mechanism used to alert applications about the different kinds of input has always been events. A keypress, for instance, causes the window system to generate a key-down event, followed by any number of key-repeat events and finally a key-up event. Similarly, a mouse will cause the window system to produce mouse-moved, mouse-down and mouse-up events. The first work to deal with both mice and multiple cursors was the classical paper by Douglas C. Engelbart and William K. English, on man-computer interaction [13].

The multi-cursor component is aimed at achieving the following goals:

- Non-invasive
- Modularizable
- "Unlimited" number of cursors
- Good performance
- Integration with the window group implementation

The multi-cursor component needs to be as non-invasive as possible. This ties in with the goal of it being modularizable, as it should be simple to just "drop into" a different window manager, without requiring too many changes. As will be explained in section 5.4.1, this goal is difficult to achieve. Non-invasiveness also implies that multi-cursor support should not require existing applications to be re-compiled for the specific goal of "understanding" multiple input sources.

There have been some previous attempts at adding multi-cursor support to X. One approach is based on multiplexing the existing system cursor to emulate a "real" multi-cursor environment [14]. The particular implementation in [14] combines X cursor multiplexing with utilizing unused bits in the existing X protocol and a modified window manager, making it possible to catch, parse and re-route multi-cursor events before they reach the client applications. This works reasonably well, but has the disadvantage that applying the modification to different window managers can be a bit of work, and that it only allows up to seven cursors (eight if counting the system cursor).

Another approach is to add multi-cursor support in the X server, with appropriate extensions exposing them to other applications. The disadvantage to this is the fact that modifications are needed to the X server, a component that can be difficult to replace in many settings. A third approach, which can be thought of as application-level multi-input, is explored in works such as MMM (Multi-Device, Multi-User, Multi-Editor) [15] and PebblesDraw [16]. MMM is based on building applications from the ground up with multi-input taken into consideration, focusing on concurrent text editing and truly handling multiple different input sources (different keyboards and different cursors). PebblesDraw is another example of application-level multi-input. Its input sources were PDAs running a program called Remote Commander, which relayed input to PebblesDraw. Remote Commander also supported relaying input to any other application, although without proper support for simultaneous usage.

The approach taken for the multi-cursor design in this thesis is a variation of the system cursor multiplexing approach. However, rather than attempting to piggy-back multi-cursor information to the existing X Windows event structures (as in [14]), a new protocol is developed. The protocol is designed only to send events from a source to a server, with the sources being the different input clients, and the server being the window manager. The protocol's client side is implemented in an application that works much like x2x[1], while the server-side is implemented in the window manager, forwarding the events in the best way possible.

The combination of the system cursor multiplexing approach and a simple protocol for forwarding events has many advantages. The implementation is no longer hampered by the legacy X protocol, and does not need to make such compromises resulting in, for instance, at most seven cursors being supported. Whether more than seven cursors are necessary in practice remains to be seen - at least it won't be a limitation for future development. Also, the multi-cursor module will be simpler to port to other window managers, as it (in theory) no longer needs the sort of hooks into the window manager's existing event processing system as are present in [14], fulfilling in part the goal of modularizability. Finally, it provides system-wide multi-cursor support, not requiring modifications to existing applications.

There are disadvantages as well. Since the design effectively necessitates an additional network connection, the multi-cursor module needs to run in its own thread, as it can't rely on the window manager to give it time to check for traffic on the sockets it listens to and communicates on. While this design also aids in making it simple to port to different window managers, it also necessitates a second connection to the X server, as Xlib isn't thread safe. It also creates a number of problems related to thread safety in interacting with the window manager's internal data structures, which turns out to be essential in making the system usable.

[1] x2x is an application that allows a mouse to be used on more than one X display.

Multi-cursor input

Multiplexing the existing X cursor is not sufficient to create a usable multi-cursor enabled window manager, as it becomes impossible for two users to simultaneously move windows, select text or perform similar actions. The solution is to add a number of special-cased behaviours to the multi-cursor design, where the system cursor is not needed. This currently applies to moving and selecting windows - a future version will also allow window resizing as part of these specially programmed behaviours. Keyboard input does not suffer from the problems associated with cursor multiplexing, as long as the correct window is focused at the time the keyboard event is processed by the X server.

Overall, the approach taken for multi-input is similar to the one employed in [14], but differs in that it does not attempt to piggyback multi-cursor information using the existing X event system. The two also differ in their strategy for managing the focus window - the solution in this thesis does not use the cursor to emulate the user changing the window focus, but rather uses the window manager's own focus management functions for making sure that keyboard events get routed to the correct window.

The decision to not use the X protocol combined with x2x for forwarding input events also allows the implementation to partially bypass the overhead of having events propagate twice through the X server[2], as they are being received directly by the multi-cursor implementation. This point is true at least for modifying the cursor location on screen, as well as the remaining cursor events and mouse-button processing. For keyboard events, the situation is slightly different; see section 5.4.1 for details.

Multi-cursor protocol

The multi-cursor protocol is a simple, acknowledgement-based protocol. For the following discussion, the *client* will be a user manipulating a cursor on a shared display, called the *server*. The client initiates the transaction by connecting to the server (which, by default, runs on port 5000 + display number). The server, upon receiving the client's connection, sends the client a message containing the cursor ID allocated for the client, as well as the width and height of the screen the cursor will be moving in. The choice of port number was meant to correlate somewhat with the ports on which a regular X server runs (6000 + display number), while also using a port that is available in the display wall lab.

The client uses the width and height of the display to intelligently scale cursor movements, as well as knowing when the cursor leaves the virtual display (meaning that the user wants her cursor back). The cursor ID is currently not used by the client, but may in the future be used to indicate what color or shape the virtual cursor has. Once the message containing this information has been received, the client and server enter a loop, where the server responds to any message sent by the client with an acknowledgement message.

The messages the client can send contain either information about a mouse motion event, mouse button event, or keyboard event. Motion events indicate that the client wants the virtual cursor to move; button events indicate that a mouse button is pressed or released (scroll wheel events are also treated as button events) and keyboard events send information about key presses and releases.

The client will, at most, send four mouse motion messages, before waiting for an acknowledgement from the server. Allowing four motion messages prevents the acknowledgment system from becoming a bottleneck (and thus, creating jerky mouse movements on the display wall), while still preventing the server from becoming swamped with motion traffic in cases where it is not able to respond quickly enough to mouse motion messages.

Keyboard and button event messages are *always* sent, even if no acknowledgements have been received. The reason that acknowledgements are used in this protocol is to prevent the server from being swamped with mouse motion messages, as these can be very frequent. They aren't otherwise necessary, as the underlying transport protocol in use is TCP.

A previous incarnation of the protocol did not require acknowledgements, and lead to the virtual cursor continuing to move for several seconds after the user stopped moving her mouse, indicating that the server was still busy processing old mouse motion messages. Mouse button and keyboard events do not require acknowledgements for two important reasons: First, they do not occur as frequently as mouse motion events, and as such rarely risk flooding the server. Second, and most importantly, it would be catastrophic for the user experience if a mouse button or keyboard event message was simply lost. A user typing "hello"

[2] In [14], the event is first received by the X server, which passes it to the window manager. The window manager decodes it, and if it was a multi-cursor event, passes it back to the X server, this time without the multi-cursor bits.

expects the full text to appear, not “helo”[3], “ello”, “heo” or any other combination of dropped keyboard packages. In the case of a lost “key released” message, the user could end up seeing “heeeee” on the display, which is just as bad.

The protocol ends whenever either side closes the connection. The server can naturally handle multiple clients simultaneously.

5.2.2 Window groups

The window group functionality has its roots in the requirement that a user should be able to move many windows simultaneously. Every cursor has its own group, with group membership indicated by drawing a colored border around the windows belonging to a particular group. The border’s color is equal to the color of the cursor owning the group - for instance, a red cursor implies that all windows belonging to the cursor’s group have a red border.

A window becomes a member of a group in one of two ways. Whenever a user clicks a window, it will be added to that user’s group. The user can also click and drag on the desktop (root window), producing a selection rectangle. Any windows inside this rectangle will replace the windows in the current user’s group (see Figure 5.2). To deselect all windows, the user can simply click on the desktop.

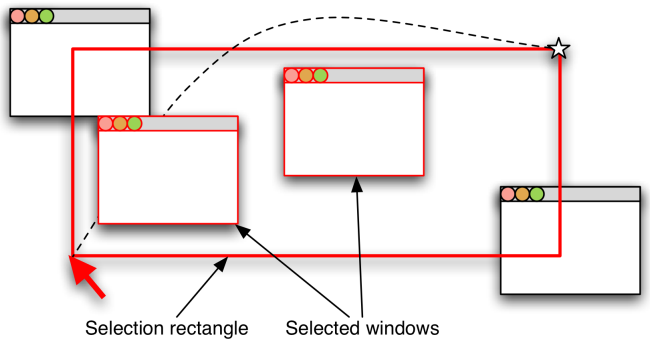

Figure 5.2: How window selection works.

The selection rectangle is also used when windows are moved, and a selection rectangle is naturally maintained for each group. When a window in a group is attempted moved beyond the (now invisible) selection rectangle, the entire group will move with it (including the selection rectangle); moving a window inside the group’s selection rectangle does not affect the positions of the other windows in the group.

The benefit with this behaviour is that it is possible to rearrange windows within the group, without moving all the other windows at the same time. When, however, the user needs to shift her “working area” from one end of the display wall to the other, all the windows can come with her. Changing working area often happens as the user moves around the room, making windows far away harder to manipulate.

The window group mechanism also supports some keyboard equivalents, as this part of the window manager not necessarily requires multi-cursor support. They serve the purpose of allowing the user to assign windows to groups and teleport groups to the current cursor position.

Window teleportation works by moving the selected group of windows to the current cursor position (or, when multi-cursor support is available, to the position of the group’s cursor), when the user presses the F1 key[4]. This is useful when many windows are selected and need to move a long distance, as dragging many windows (and thus many pixels) tends to be slow on the display wall.

Window placement

The algorithm for placing windows is designed to be simple and provide for “convenient” positioning of new windows. It works by simply positioning new windows at the current cursor position. Windows that don’t belong to a group are placed at the system cursor’s position, while windows spawned from a window

[3] Except, maybe, if the server happens to be a mail server. This is not the case here.

[4] When the window grouping mechanism is used outside the multi-cursor implementation, the different groups are teleported using the F1-F10 keys.

belonging to a group (that is, windows whose parent window belongs to a group) are positioned at the location of that group's virtual cursor. The new window is also added to that group.

5.3 Selecting a window manager

Deciding on a window manager to modify was a difficult task. There are many window managers for the X Window System to choose from, but in order for there to be a reasonable chance of completing an implementation complying to the requirements and design outlined above, it can not be to complex in its implementation. That is, finding a suitable location to hook into the existing implementation is the key to success.

Unfortunately, this conflicts with the next desirable quality of the window manager: That it is sufficiently feature rich to be comfortably used on the display wall. Among the window managers considered before the choice fell on Window Maker [17], were twm, FluxBox (a BlackBox spinoff), Enlightenment, IceWM, kwm (the KDE window manager) and CDE. This list is by no means comprehensive. The main reasons for rejecting a window manager were stability, code complexity and feature set. FluxBox, for instance, does not handle the resolution offered by the display wall, never managing to start up. twm simply was too simple in its implementation, and kwm was rejected because of its complexity and tight integration with KDE.

In the end, Window Maker was chosen. The window manager offers a respectable set of features, providing compatibility with both Gnome and KDE applications, while at the same time retaining a "no-frills" approach to managing windows. The source code is acceptably complex and structured given the features it provides, and seemed fairly simple to hook into. Also, Window Maker has support for moving multiple windows - however, the mechanism used in Window Maker is slightly different from the solution sought in this thesis. The current approach in Window Maker only allows moving many windows while retaining their relative distances - this thesis aims to allow a single window to move independently within a well-defined area, and the entire group once the window is moved outside this area.

Finally, Window Maker was chosen because the author of this thesis found it to work well on the display wall, while providing a good-looking graphical user interface mixed with a reasonable set of useful features.

5.4 Implementation

This section describes the implementation of the multi-cursor and window group modules, developed for the Window Maker window manager. The code was written in C, and compiles with Window Maker 0.80.2. Figure 5.3 shows a screenshot of the completed implementation.

5.4.1 Multi-cursor implementation

Multi-cursor support is initialized from the window group init function. It also depends upon some of the data structures used by the window group implementation, primarily for selecting the cursor color. Initialization consists of the following steps:

1. Open a new connection to the X server
2. Set a blank system cursor
3. Check that the X server supports the XTest extension
4. Create a graphics context for the window selection rectangles
5. Start the processing thread

A new connection is necessary due to the fact that the multi-cursor component runs in its own thread, and thus risks interfering with the window manager's Xlib calls. The blank system cursor is used to avoid flickering, caused by the X cursor multiplexing mechanism. (The system cursor jumps between many different positions to emulate the virtual cursors.) The XTest extension (which is supported by nearly all X servers these days) provides a way to post keyboard, button and mouse events that look like real events to the applications receiving them, as opposed to the `XSendEvent` function, which produces events that in

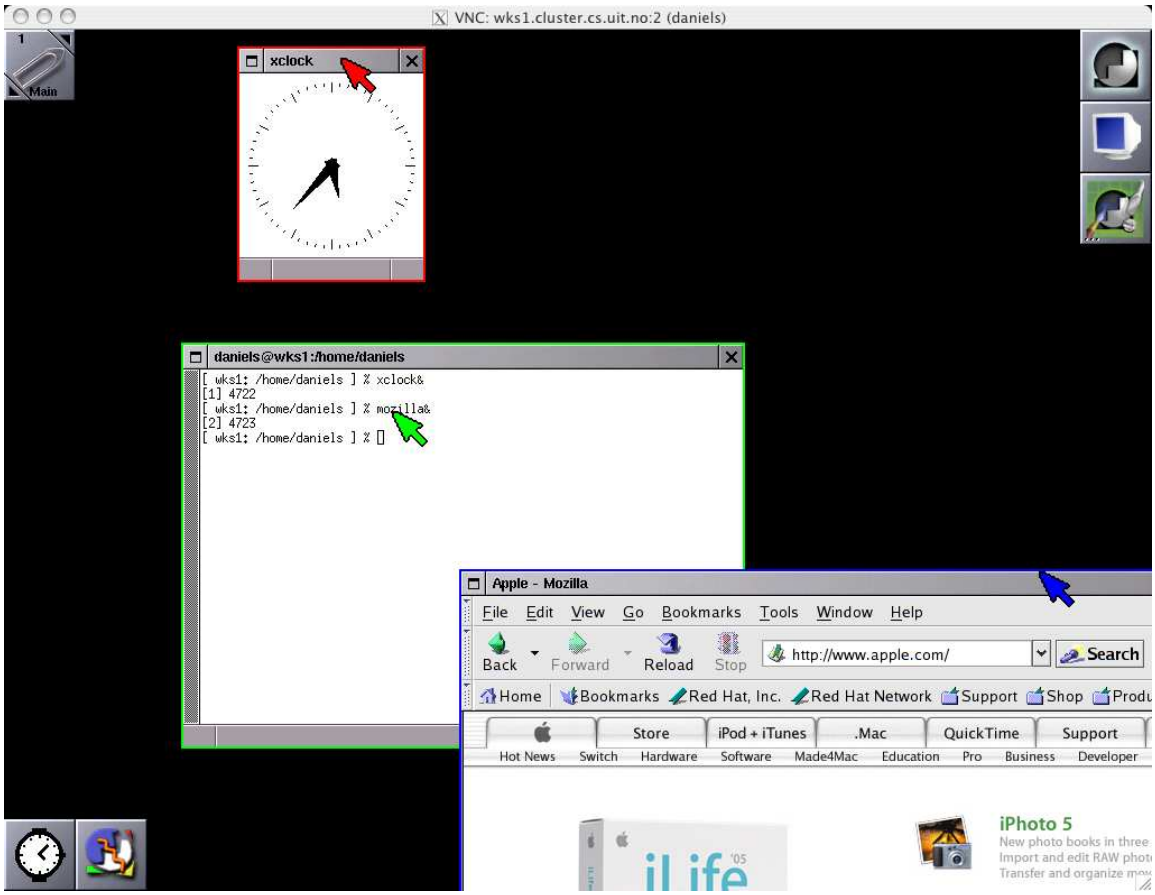

Figure 5.3: A screenshot of the modified window manager, featuring three concurrent users, each owning one window.

most cases are masked by applications. The graphics context created is used to draw the window selection rectangles. This task is handled by the multi-cursor implementation, as the group implementation does not have any knowledge per se of more than one cursor (group support can work independently of multi-cursor support).

Once the processing thread is up and running, one additional initialization step is taken, by creating a socket listening to incoming multi-cursor clients. If successful, the code goes into a loop, processing requests from clients as they arrive (see Figure 5.4).

As alluded to in section 5.2.1, attaining the goal of window manager non-invasiveness in the multi-cursor implementation proved to be a difficult task. The reason for this is that the multi-cursor implementation needs access to a number of window manager-internal data structures, as well as use a number of functions in the window manager for moving windows. This implies a tighter dependency on the window manager's implementation than what is strictly desired. Unfortunately, this integration is necessary to achieve the required performance and usability.

Request processing

This section describes how the server responds to the various requests described in section 5.2.1, and in general how clients are managed by the server.

When a new client connects to the multi-cursor server, the server assigns the first available cursor ID to the new client. This means that the cursor ID a user ends up with for most purposes is random, and that a user can't rely on always receiving, say, the green cursor. Although it would be possible to extend the multi-cursor protocol with messages allowing the user to choose a cursor ID, it was decided that this functionality was unnecessary, as it needlessly complicates the user's multi-cursor experience. A user just want's to have a cursor on the display wall, and not worry about details such as cursor IDs. Also, since the

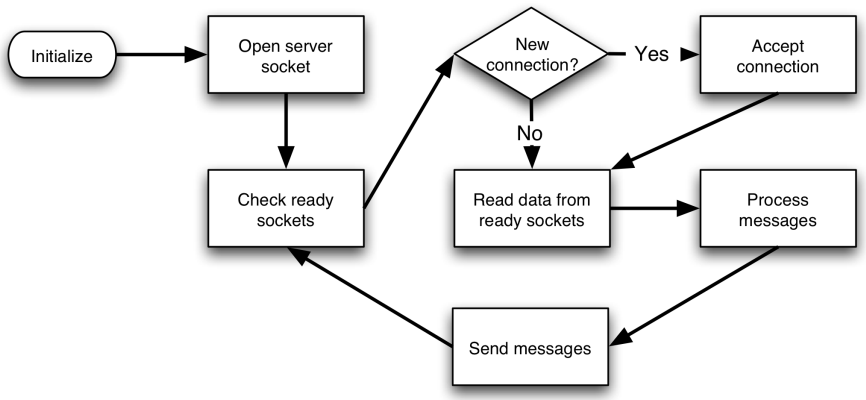

Figure 5.4: The multi-input server loop.

window grouping functionality is fairly non-persistent in both design and implementation, users need not worry about "losing their group" or similar, as they can simply "re-color" it when they resume interaction with the display wall.

After having received the connection and assigned the cursor ID, the server creates a window representing the virtual cursor. The window is shaped as a cursor, and filled with the color associated with the cursor. The server proceeds to receive keyboard, button and mouse input events from the client. The way these events are handled depends on the current state of the client. The client can be in the following states:

1. Default
2. Selecting
3. Moving
4. Dragging

In the default state, clicks and keyboard events are simply relayed using the XTest extension, although keyboard events are handled a bit differently than clicks (see 5.4.1). The multi-cursor module maintains a focus window for each cursor, which is used when keyboard events are posted. During keyboard event posting, the cursor window is hidden automatically. This prevents typed text from becoming obscured, and also prevents the cursor windows fighting with the "real" window for focus, as the virtual cursors *are* windows.

Mouse movements are echoed to the user by moving the "cursor-window" associated with the cursor, giving the appearance of an actual cursor moving on screen. The window is created and maintained in such a way that it is always on top, by listening for `VisibilityNotify` events produced by the X server. This ensures that the cursor is always visible. The cursor windows are 32x32 pixels large, 4 times more than the standard cursor size of 16x16 pixels.

The selecting state is used when a client attempts to select a number of windows and group them. A client enters the selecting state whenever the left mouse button is pressed with the virtual cursor residing in the screen's root window. While the client is in this state, a selection rectangle is drawn between the point where the click began, and the point where the virtual cursor currently is. Once the left mouse button is released, the client leaves the selecting state. Any window completely inside the two corner-points of the selection rectangle at this time are grouped and associated with the client.

The moving state is used when a user wants to move a window, and is entered when a user clicks and holds on a window's title bar. The reason this state exists is that multiplexing the system cursor doesn't work when multiple users attempt to move windows at the same time, resulting in the different windows jumping between the different positions the system cursor ends up moving to and from. The moving state ends when the user releases the left mouse button. A similar problem exists for users simultaneously attempting to resize windows, and it can be solved by adding a new cursor state (this is left for future work). Note that the moving state uses some internal window manager functions for performing the actual window

repositioning (to achieve things like “sticky” windows and prevent windows from moving offscreen - things already implemented by the window manager).

Finally, the dragging state is used for emulating drags using the system cursor, and applies regardless of what button is being clicked. The system cursor is moved to the location of the virtual cursor, and a drag is emulated by first posting a mouse button down event, followed by a series of mouse motion events. The drag is concluded once the user releases the mouse button in question, and an emulated mouse button released event is sent. Only one user can be in the dragging state at any given point in time, to prevent interference with their current action from other users.

When the user leaves the display wall (i.e., regains control over her local cursor, rather than having cursor movements forwarded to the display wall), the server proceeds to hide the virtual cursor window, and remove the cursor from the server loop.

Keyboard event delivery

Correctly delivering keyboard events turns out to be very tricky to get right. The initial attempt, combining `XSetInputFocus` with a call to `XTestFakeKeyboardEvent`, did not work as expected. The difficulty does not lie in synthesizing the keyboard event - the `XTestFakeKeyboardEvent` call handles this part beautifully. Rather, the problem is making sure that the event gets posted to the correct window. The overall design of multi-input event processing is illustrated in Figure 5.5.

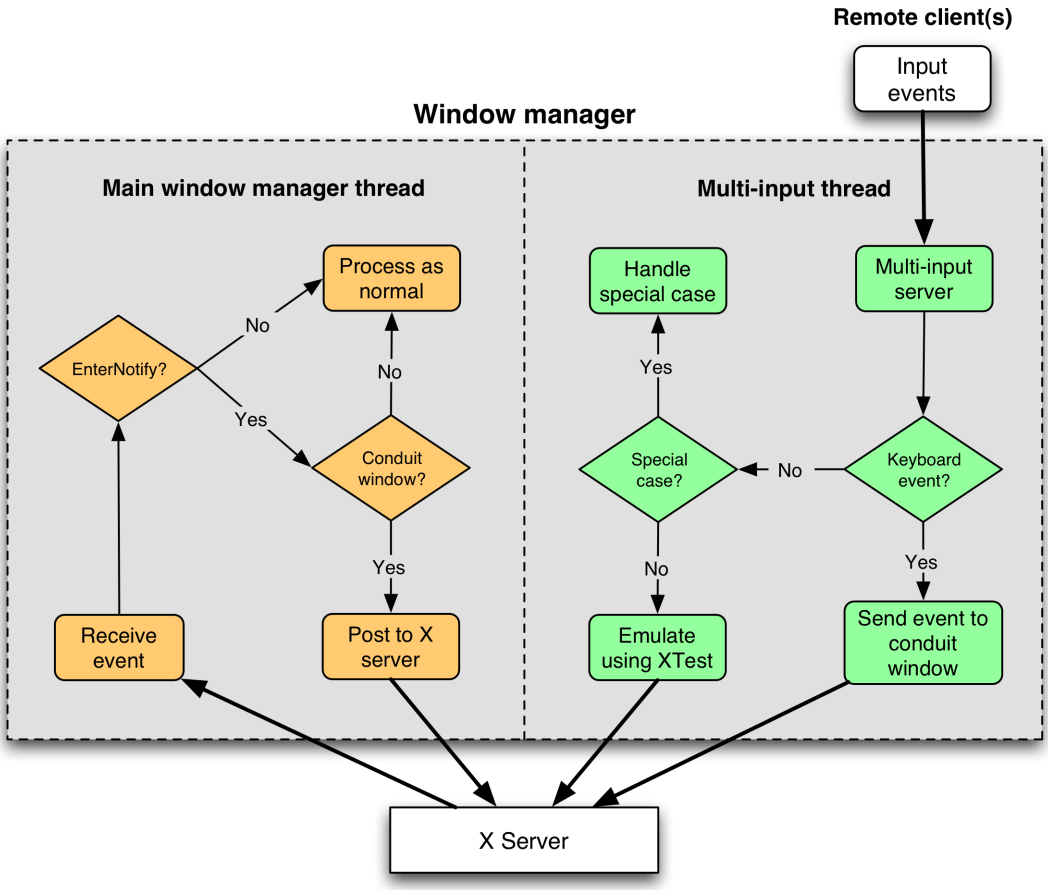

Figure 5.5: Multi-input event handling and forwarding.

Realizing that the problem lies with the window manager’s focus management, a second attempt was made where the correct window is focused, using the window manager’s own functions for focus management (`wSetFocusTo`). This attempt was mildly successful, as keyboard input now worked mostly as

expected. The problem with this approach was with Xlib and thread safety, as the window manager's focus management code uses the window manager's own connection to the X server. The first attempt at resolving this was based on simply adding the X server connection as a parameter to the `wSetFocusTo` routine - a strategy that had proven itself useful for utilizing other window manager functions. It was quickly discovered, however, that this was not a feasible task, as the routine relies on a number of other functions that would also require the same treatment, which in turn rely on other functions, and so on.

The only solution was to post the events from the main thread, avoiding all of the thread safety issues. Doing this became another exercise in working with and around the X event system. The general idea of the approach is as follows. First, a window is created in the main thread (using the window manager's X server connection). This window will never be mapped on screen, and serves only as a conduit to the window manager.

The second step involves using `XSendEvent` to send an event to the window constructed in the first step. This event will be delivered to the window manager, as that is the client owning the conduit window. The window manager's event handling system was then patched to catch any events destined for the conduit window. These events are then parsed for the keyboard and focus window information, before the focus window is set and the event posted using the XTest extension.

The obvious choice of event to send was the `ClientMessage` event, which allows users to send 32 bytes of data to any window. After testing this, it turns out that the window manager frequently runs "sub-event loops" in many locations, were `ClientMessage` events are not processed as expected (often because these loops mask out most events). As it happens, the `ClientMessage` events can not be "masked in", resulting in the need for a different event type which does not suffer from this problem. The need arises because the sub-event loops have the ability to effectively deadlock the window manager. A simple example illustrates this: Processing the "alt-tab" keyboard equivalent. A user attempting to "alt-tab" to a different window would end up deadlocking the window manager, as the keyboard release event never is posted. The reason is that the `ClientMessage` containing the event isn't processed by the sub-loop handling the "alt-tab" combination, resulting in the window manager waiting forever for the release event.

The only event available to remedy this problem is the `EnterNotify` event (or it's sibling, `LeaveNotify`). It has a sufficient number of available fields to pack the conduit window, focus window, keycode and keyboard status (press or release), while still propagating to the point where the event hook in the window manager has been installed. The event type can also be "masked in" in the previously mentioned sub-loops, allowing it to be handled correctly also in these contexts. Using this event, keyboard events are now transported from the multi-cursor module to the window manager, where they are "executed". The strategy has paid off and works well, while at the same time avoiding numerous thread safety issues.

x2wmx

As part of the multi-cursor module, a client application to forward mouse and keyboard input in a multi-cursor compatible way was needed. This section details the implementation of this application, called `x2wmx`.

`x2wmx` works on the same principle as the, until now, standard application for forwarding such information: `x2x`. `x2x` is the de-facto way to forward X11 events from one computer to another, also outside the display wall realm. `x2x` has many more features than what was required for use in a multi-cursor environment (such as support for "shadow displays" and different interfaces for grabbing and releasing the cursor), which makes its codebase fairly difficult to port in a sane manner to the multi-cursor realm[5]. This is the reason a new application was developed from scratch, rather than building on the existing `x2x` codebase.

`x2wmx` is a very simple application. On launch, it parses its arguments, before creating a thin, 1-pixel wide window along the left edge of the screen. Whenever the cursor enters this window (see Figure 5.6), it is grabbed and hidden on the local display. All subsequent movements, keyboard and mouse button events are forwarded to the remote end. When the remote cursor shifts off the other edge of the display wall, control is returned to the local cursor. Event forwarding is done using the previously described multi-cursor event protocol (see section 5.2.1), sharing the messaging code with the Window Manager's multi-cursor module.

Local events are processed in a loop together with the messaging socket used to communicate with the remote multi-cursor server. The messaging socket is opened once the cursor enters x2wmx' border window, and closed when the cursor returns to the local display.

[5] In [14], `x2x` is modified to include the necessary bits indicating the cursor it is controlling. These modifications are small enough to easily be applied to `x2x`.

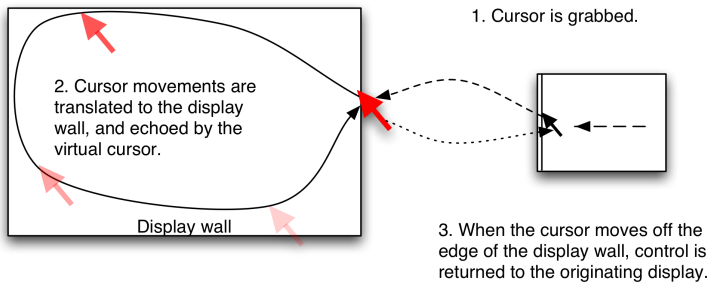

Figure 5.6: x2wmx grabbing, forwarding and releasing the cursor.

`x2wmx` has three different forwarding modes for translating cursor movements on the local computer to cursor movements on the display wall: Absolute, relative and pad. Absolute positioning is the simplest mode, where cursor movements on the local computer are scaled to the coordinate space of the display wall:

$scale\_x = remote\_width / local\_width$
$scale\_y = remote\_height / local\_height$
$x' = x * scale\_x$
$y' = y * scale\_y$

The problem with absolute mode is that it doesn't always provide sufficiently good precision. The relative mode rectifies this problem, by using relative distances to calculate the new location of the remote cursor, using the difference between the current and previous local cursor locations to calculate the delta values. The relative and absolute modes work well for computers connected to standard mouse input devices, where no sudden "jumps" in the local cursor position can occur. This is not the case for the tablet computer present in the lab, which uses a stylus as its primary mouse input device. The stylus input is processed in an absolute fashion by the tablet, meaning that the cursor is kept at the tip of the stylus.

This kind of cursor positioning creates problems for the relative forwarding mode, but continues to work correctly for the absolute forwarding mode. Absolute mode is not ideal, however, when working on the tablet, as it prompts extensive hand movements to move the cursor. Users also tend to gravitate towards keeping the stylus roughly centered on the tablet's display, confusing users who all appear to expect a relative forwarding mode. The same problem with lack of precision also plagues absolute mode on the tablet. In sum, these two factors prompted development of a different forwarding mechanism for use on the tablet.

The model for developing this forwarding mode was to emulate a touchpad, as present on many laptops. After a lot of experimentation, a suitable way to emulate touchpad behaviour was found. It works by maintaining a "center point". The center point is used to calculate dx and dy:

$dx = current\_x - center\_x$
$dy = current\_y - center\_y$

The deltas are then scaled to prevent enormous jumps in the cursor position on the display wall, using the following function:

$scale(x) = (1 + (\ln x)^3) * scaling\_factor$

Two plots of the function can be seen in Figure 5.7. The input value (a pixel delta in either the X or Y direction) is along the X axis, with the function's output along the Y axis. This function was chosen because it maps large deltas to smaller deltas, while slightly accelerating the cursor when the deltas are small (less than 96). A number of other functions were also experimented with, but they all produced behaviour that felt wrong to work with on the display wall.

The linear scaling factor is used to increase or decrease the standard speed obtained from moving the stylus on the tablet. The scaling factor should lie in the interval $[1, 2>$, to allow for precise cursor movements using the stylus (factors greater or equal to 2 will effectively prevent cursor movements with deltas less than the integer value of the scale factor). The cursor position is stored as a floating point number in the implementation, allowing the fractional parts of the deltas to accumulate correctly. Note that the cursor position is sent to the server as integers.

For all this to work, however, the center point needs to be maintained in an intelligent manner. The center point is changed in two ways, as illustrated by the pseudo-code below:

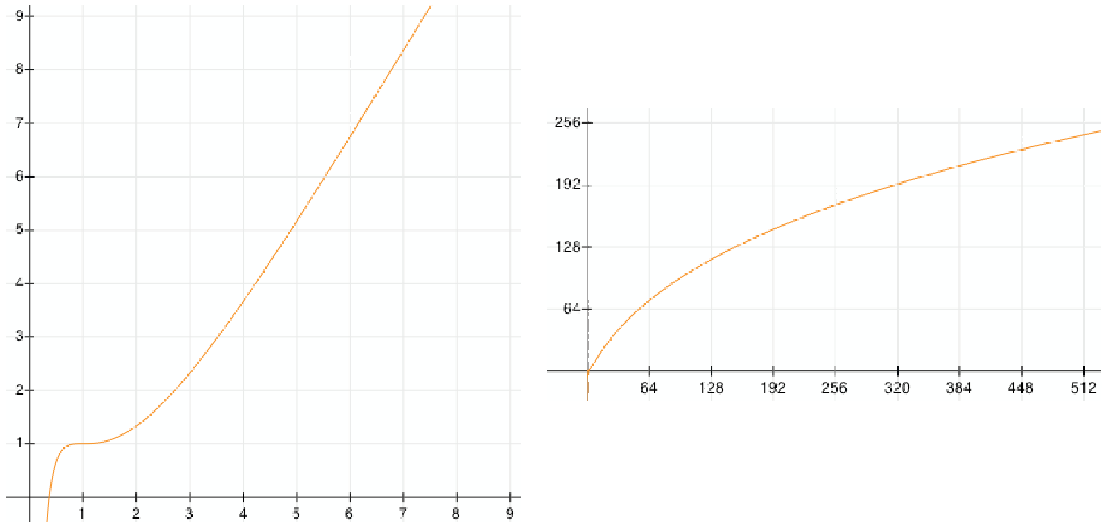

Figure 5.7: Plots of the function used for scaling mouse deltas.

```
if (current time - last event time > threshold)
   center point = current cursor location

last event time = current time
delta           = current cursor location - center point
center point    = average(current cursor location, center point)
delta           = scale(delta)
```

First, if the amount of time since the last event exceeds a certain threshold, the center point is reset to the current cursor position. This has the effect that the user can drag the stylus, lift it and return to the original position, continuing the drag, without adversely affecting the virtual cursor position (this is impossible in the standard relative mode). Second, the gradual convergence of the center point and current cursor location allows the user to move the stylus to a position on the tablet's display and keep it there, without producing a continuous stream of large deltas.

In testing, the pad forwarding algorithm has proven itself to be both precise and simple to use on the tablet computer, although different users end up preferring different levels of "acceleration", which is why the scaling factor is allowed as an argument to `x2wmx`. It should be noted that using the tablet computer requires some practice, as the stylus needs to be lifted a rather long distance from the tablet's display before not producing further local cursor events. The user needs to take this distance into account when moving the stylus back and forth in an attempt to move the cursor a long distance in one direction.

5.4.2 Window group implementation

The window group implementation works by first initializing its group-array. Currently, only 10 groups are supported, but this can easily be changed. The limitation stems primarily from the number of common, named colors available in X windows. Initialization consists of zeroing a number of fields, and allocating colors to the various groups. These colors are also used by the multi-cursor implementation to color the various cursors. Once the colors have been allocated, the group implementation calls the multi-cursor initialization routine.

The remaining part of the group implementation consists of functions to manipulate groups; adding one or several windows to a group, removing one window or clearing all windows from a group and manipulating the bounding rectangle encompassing all windows in a group. These functions are called both from the multi-cursor implementation, as well as from various places in the existing window manager code, usually in response to a keypress. Note that this does not introduce problems with thread safety, as the calls are only come from either the window manager (i.e., when multi-input is not enabled) or the multi-input code. The

multi-input code *does not* pass along the keyboard events that trigger these calls from within the window manager's main thread.

In order to associate windows with groups, it was also necessary to add an identifier to Window Maker's central `WWindow` structure. This variable is used in the window manager's `doWindowMove` function to determine whether the window belongs to a group, and if so fetch the group's selection rect, and move multiple windows if more than one window belong to the group. It is also used to implement group inheritance - that is, new windows whose parent is assigned to a group, is assigned to the same group. This mechanism is essential in having new windows pop up at the requesting user's cursor position.

Intelligent window placement is implemented by calling a custom placement function after the window has been initially positioned by the window manager, but before the window is updated. The placement is performed after having figured out if the window should belong to a group.

Each group maintains a focus window, a bounding rectangle, a group color and a list of windows belonging to the group. The focus window is mainly used when the multi-cursor implementation is *not* present, in order to "focus" a group. The bounding rectangle is the union of the initial selection rectangle (if any) and the bounding rectangle of all windows belonging to the group.

As part of the window group implementation, modifications were also made to the window manager's existing window movement code. These modifications hook into the group implementation, and check whether the entire group should be moved, or just a single window. This allows existing window movement code to be leveraged, meaning that features such as "sticky windows"[6] continue to work even in the presence of window groups.

As part of the requirements, the window manager should also provide larger window borders to more easily allow users to resize windows. This need exists because hitting thin borders on a large screen quickly becomes a very delicate and difficult task. Implementing this requirement was done by changing a constant in the window manager sources, controlling the thickness of the resize border.

5.4.3 Multithreading concerns

Both the multi-cursor implementation and the window group implementation give rise to some multithreading concerns in their integration with the host window manager, although these problems plague the multi-cursor implementation more than the window grouping component. The underlying reason for these multithreading concerns is simply the fact that central data structures in the window manager may be modified by the window manager at the same time as the two components attempt to access them. The multi-cursor module also needs to be careful to avoid calling functions in the window manager utilizing the window manager's connection to the X server.

A problem also related to thread safety is the problem of stale pointers. Since the code written is an add-on to an existing, fairly complex window manager, it is important to ensure that windows are not freed without the two components' knowledge. This has been solved by incorporating hooks into the window manager's window destruction function, allowing clean-up also in the group and multi-cursor modules. Apart from this, the window grouping code is not subject to multithreading problems to such a large extent as the multi-cursor component, as it runs in the same thread as the window manager.

Currently, the multi-cursor module does a number of things to ensure some level of thread safety. First, a number of the window manager's functions have been modified to take the X server connection as an argument, rather than using Window Maker's global `dpy` variable. These changes prevent problems related to out-of-sync X connections, and allow the multi-cursor module to use some of the window manager's functions without too many problems. This change does not completely ensure thread safety though, as the list of windows may still change while the multi-cursor module is executing code in somewhere in the window manager's bowels. This happens mainly when the user selects or moves windows, but also every time the user performs a mouse click.

To avoid dangling references and windows being removed by the window manager while they are being worked on by the multi-cursor implementation, a mutex has been introduced that is locked whenever a message is received and processed, and unlocked when processing is done. The only other place the mutex is taken is in the code responsible for destroying windows - `wWindowDestroy` in the existing window manager. At this point, two methods are called in the group and multi-cursor implementations, removing references to the window being destroyed. The lock is not released until the window has been removed

[6] When a window is moved close to a different window, it will "snap" to that window when the distance between the windows' borders are less than a certain pixel threshold.

from the window manager's internal list of windows. Although this locking strategy is oriented towards the "one giant lock" approach, it was the only feasible solution to implement in the time available for this thesis, as the window manager is too complex for it to be possible to introduce fine-grained locking without investing a large amount of effort.

There are still undetermined locations in the code where race conditions can (and do) occur. In using the window manager, however, it turns out that the possible race conditions rarely occur[7], and the window manager has proven remarkably stable considering that it has not been made entirely thread safe. This is likely due to extreme care in the design of the multi-cursor module, avoiding techniques where race conditions would be made more likely. Also, keyboard event processing is executed in the main thread, avoiding the problem altogether.

The problems of thread safety, however, illuminate one area that was not considered sufficiently well when deciding on a window manager. Ideally, the selected window manager should have been multithreaded already (and thus, presumably, thread safe), or at the very least consistently use data structures that are simple to make thread safe. A simple structure, such as a linked list, can be made thread safe relatively easily by only using wrapper functions to perform list management and iteration. This is not done consistently in the Window Maker implementation, thus thwarting any attempt at making such list management functions thread safe.

5.5 Experiences using the software

The software developed to implement multi-cursor and window group support has been continuously tested during its development, and is now in a stable state. The multi-cursor implementation has been exercised by having 3 users simultaneously interacting with the wall. This testing uncovered a couple of interesting behaviours and performance problems with the multi-cursor implementation.

First, as expected, multiplexing the system cursor works, but is not without flaws. In an early version, users ended up noticing this particularly well when attempting to resize windows or select menu items from different menus. If another user attempted to use the system cursor at the same time in a similar operation, the result was that neither user got their task performed as expected. This is the main shortcoming of the cursor multiplexing paradigm, and has been partially resolved by preventing other users from interacting with the system cursor when it is in use by another user. The resizing problems can also be relieved by implementing a fifth cursor state, a feature currently left for future work.

An unexpected performance problem with the multi-cursor responsiveness became evident when multiple cursors were in use on the display wall. If one user started dragging many windows totaling a large area on the display wall, the other cursors would begin stuttering, and move in an extremely jerky fashion. This is not a shortcoming of the multi-cursor implementation per se, but an artifact from using VNC for driving the wall. Since the cursors are just windows (and thus pixels) being moved around, they end up being updated less often when large windows are moved, causing the apparent stuttering effect. For the users, though, the root cause isn't the issue - the lack of responsiveness is.

Multi-input has also been very extensively tested, due to the many attempts required before getting the keyboard event delivery mechanisms right. While it usually works, there are corner cases where input from multiple clients end up getting posted to the same window, or not at all. One example of this is cases where users share the same focus window (even though it only belongs to one group). A similar problem exists when applications enter fullscreen mode. This causes the multi-cursor module to incorrectly set the focus window, resulting in keyboard input being lost.

The focus mechanism also ends up continuously flashing the different windows receiving keyboard events between "focused" and "unfocused" appearances, a visually displeasing effect. Solving this problem is currently left as future work.

The window placement algorithm works well, and usually positions windows where a user would expect the window to appear. The algorithm is not foolproof, however, as there are cases where it is not possible to determine who the window should belong to. In general, though, this feature has improved the user experience more than it has diminished it, and as such should be considered a success.

Testing also uncovered that the modified window manager still occasionally experiences problems with Xlib and what appears to be Xlib calls from the multi-cursor thread utilizing the window manager's X server connection. The root cause of these problems has yet to be determined, as the problem occurs very rarely and is difficult to reproduce for debugging.

[7] One of the hallmarks of race conditions.

Chapter 6

Related work

Much of the related work in the fields studied in this thesis does not deal with the problems examined in this thesis. They focus instead on conducting user studies or perform different kinds of thought experiments - things that are quite irrelevant when it fore instance comes to looking for interesting, working multi-input implementations. Despite this, there are a number of articles that are relevant to the discussions in this thesis.

The first work in which multiple cursors were present was in an article by Douglas C. Engelbart and William K. English from 1968 [13]. This work, in addition to presenting the mouse as an input device, also allowed collaboration by giving one user a controlling cursor, leaving the rest of the users with cursors that could only be used for pointing.

Since then, a number of interesting systems have been developed in the field of multi-input. Some of them also have features similar to the window grouping feature developed in this thesis. MMM [15] and PebblesDraw [16] have already been mentioned - see section 5.2.1. Tivoli [18] is a third example featuring a shared whiteboard-style solution, where users interact with the shared surface using special pens. Tivoli is also interesting because they have made similar observations regarding window placement as in this thesis, although their motivation is different, in that users simply can't reach dialog boxes or other controls placed too high up on the "liveboard". The RoomWare "DynaWall" [19] is another example of a shared whiteboard, and also carries similarities to the work developed in this thesis, especially in their window placement policy: The DynaWall dialog boxes always pop up in front of the current user(s). The DynaWall also supports simultaneously interacting users, but is limited in that it doesn't allow generic multi-input - users have to use the application(s) provided, and nothing else, whereas the work presented here allows multi-input to any X application.

Most similar to the multi-input solution developed in this thesis is the multi-input window manager developed at Princeton University [14]. The main difference between the two is how focus is managed and the mechanism by which input is received by the multi-input implementation.

In Dynamo [20], the developers have created a feature called "carves". A carve is an area of the screen where only the user creating the carve can interact, and optionally grant access to other users later on. Carves differ in their goals from the window grouping developed in this thesis. While carves primarily were created for the purpose of persistence and access-control, the window grouping mechanism is aimed at easing interaction with many windows on the display wall. Window groups are not as persistent as the carves in Dynamo, nor are they meant to be.

In terms of display wall management, there has also been some previous work. Display Wall-in-a-Box [3] was tested, but found lacking in several areas. It does not provide projector control, and the VNC server and viewers shipped with it (Tileviewer) are unstable. DwallGUI [21] provides a feature set similar to Wall Manager, but differs in many key areas. Where Wall Manager requires Mac OS X to run, DwallGUI runs on Windows. The cluster being controlled by DwallGUI also needs to run Windows. Wall Manager also differs in that it helps the user isolate and locate the source of errors, a service not provided by DwallGUI.

Chapter 7

Conclusions

This thesis has had its main focus on two parts of the display wall: The software driving and managing the wall, and the experience end users have in working with the wall, and how that user experience can be improved. A sophisticated system for operating the wall, both for administrators, power-users and end-users alike has been developed. Over time the system has proven itself to be stable and simple to extend when the need presented itself. The master-slave organization of the wall-controlling scripts is robust and intuitive, both from an end-user standpoint and from a development perspective.

The Wall Manager software has further elaborated on this, and is currently the best and simplest way of operating the display wall. It has successfully achieved the goal of giving the display wall a simple "on-off" switch, while still providing power-users with enough control to be able to diagnose errors. The more advanced functions of the master script have also been successfully exposed through a simple interface.

The modifications made to the Window Maker window manager have succeeded in their two primary goals of bringing multiple cursors to the display wall, and easing the manipulation of many windows for end users. Multi-cursor support works reliably, and has already contributed to making the display wall a canvas suited for many users working together at once. As expected, the window group feature has eased window movement and alleviated frustration as users shift their focus areas on the display wall. The simple modification of making windows pop up near the current user's cursor also aids in improving the user experience.

Investigating different ways of powering the display wall was also instructive, both in watching how the modified window manager would work with the alternative Xdmx solution, and examining the differing performance characteristics of a VNC-driven approach versus Xdmx. While Xdmx currently doesn't work well enough, it shows promise.

In conclusion, the work carried out in this thesis has eased interaction with the wall, and lowered the bar for casual users wanting to use it. The multi-input implementation promises to allow for greater "parallelism" among people wanting to interact with the wall. The window grouping feature tends to become addictive after using it for a while, an effect that surely indicates that the feature is useful. Having multiple cursors available also feels "liberating" in interacting with the wall, as one no longer has to wait for one's own turn using it. Wall Manager is already in day-to-day use, and the modified window manager will be configured as the default once the remaining issues are solved.

7.1 Limitations and future work

The current window group implementation is limited to at most 10 groups. The primary reason for this limitation stems from the most common "named" colors available in X Windows, and can easily be rectified by dynamically creating colors based on RGB values or adding further entries to the list of named colors. This also imposes a limit of 10 cursors on the multi-cursor component, which gets its colors from the window group code. Fixing this is not urgent, as there currently aren't even enough computers equipped with mice in the display wall lab to use all 10 cursors.

Although the window manager currently is largely thread safe, there may be undiscovered problems or race conditions that need addressing. Verifying the thread safety of the window manager was unfortunately too much work for this thesis (and not to mention far outside the scope of it), but should be done in the future. Also, the window manager currently produces a number of graphical glitches during tasks such as

window resizing or window selection. The problems stem from calls to `XGrabServer` being removed from the window manager, as they would effectively result in a deadlock between the multi-cursor module and the window manager[1]. These problems should be fixed, but were not prioritized during development for this thesis, as the goal was prototyping functionality, not flawless presentation.

The display wall could also be made easier to use, by implementing zoomability of various parts of the display. Users often have the need to enter or read small type on the display wall, but find themselves standing or sitting on the wrong side of the room, being too far from the wall to read the text. This could easily be solved by implementing a zooming mechanism for X Windows. Doing this correctly may be somewhat difficult, however, and has been left for future work.

Related to both window groups and multiple cursors, one can see the need for making ones window(s) "private". That is, preventing other users from interacting with windows that oneself has claimed. Similar features already exist in systems such as the carve-based system mentioned above. Exploring this topic further will be done in future experiments, once the grouping implementation has been modified to support "window locking".

The window group support will also be extended in the future with an audio-based window movement mechanism, allowing a user to stand in front of the screen and clap her hands to move a bunch of grouped windows from one end of the display to the other, or something similar to this, as an experiment in using sound to interact with large surfaces. This will be a first step towards Asimov's vision of a fully interactive display wall, where objects can be manipulated by gestures and voice commands on surfaces that effectively are display walls.

The x2wmx "pad" implementation currently doesn't support real acceleration, only scaling. A future version of x2wmx should implement a proper acceleration mechanism, taking the swiftness of the stroke into account when reporting deltas to the multi-input server.

Wall Manager in its current state does not need much further work, although it would be desirable to implement support for allowing users to control the wall from the PowerMac, either directly from Wall Manager, or using `x2x` or `x2wmx`. Support for even better control over the cluster nodes should also be considered. Note, however, that `x2x` does not work correctly on Mac OS X, and that a similar utility called `osx2x` can be used in its place.

[1]The problem is strange, though, as the XTest extension provides a function for making the calling client impervious to server grabs. For some reason, this call does not seem to work.

References

[1] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), January 1998.

[2] VNC for Unix 4.0. http://www.realvnc.com/.

[3] Dave Semeraro and NCSA. Display Wall-in-a-Box. http://www.ncsa.uiuc.edu/Projects/AllProjects/Projects83.html.

[4] Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, 1(2):67–74, 2002.

[5] Rickard E. Faith and Kevin E. Martin. Xdmx: Distributed, multi-head X. http://dmx.sourceforge.net/.

[6] Daniel Stødle. Collaborative Sharing of Windows Between Mac OS X, the X Window System and Windows, July 2004. Term paper, University of Tromsø.

[7] NoMachine. NX server and client. http://www.nomachine.com/.

[8] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702. ACM Press, 2002.

[9] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: a scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140. ACM Press, 2001.

[10] PyObjC, an open source bridge between Objective-C and Python. http://pyobjc.sourceforge.net/.

[11] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[12] Microsoft Netmeeting. http://www.microsoft.com/windows/NetMeeting/.

[13] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference*, pages 395–410, December 1968.

[14] Grant Wallace, Peng Bi, Kai Li, and Otto Anshus. A MultiCursor X Window Manager Supporting Control Room Collaboration. Technical Report TR-707-04, Princeton University, Computer Science, July 2004.

[15] Eric A. Bier and Steven Freeman. MMM: A user interface architecture for shared editors on a single screen. In *UIST '91: Proceedings of the 4th annual ACM symposium on User interface software and technology*, pages 79–86. ACM Press, 1991.

[16] Brad A. Myers, Herb Stiel, and Robert Gargiulo. Collaboration using multiple PDAs connected to a PC. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 285–294. ACM Press, 1998.

[17] Window Maker, an open-source window manager for the X Window System. http://www.windowmaker.org/.

[18] Elin Rønby Pedersen, Kim McCall, Thomas P. Moran, and Frank G. Halasz. Tivoli: An electronic whiteboard for informal workgroup meetings. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 391–398. ACM Press, 1993.

[19] Norbert A. Streitz, Jörg Geißler, Torsten Holmer, Shin'ichi Konomi, Christian Müller-Tomfelde, Wolfgang Reischl, Petra Rexroth, Peter Seitz, and Ralf Steinmetz. i-land: an interactive landscape for creativity and innovation. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 120–127. ACM Press, 1999. See also: http://www.ipsi.fraunhofer.de/ambiente/english/projekte/projekte/dynawall.html.

[20] Shahram Izadi, Harry Brignull, Tom Rodden, Yvonne Rogers, and Mia Underwood. Dynamo: A public interactive surface supporting the cooperative sharing and exchange of media. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 159–168. ACM Press, 2003.

[21] Grant Wallace. Display wall cluster management. IEEE Visualization 2002 Workshop on Commodity-Based Visualization Clusters. http://www.cs.princeton.edu/omnimedia/papers.html.

Appendix A

CD-ROM

The included CD-ROM contains the source code for both the Wall Manager application and the `wall_ctrl` scripts. In addition, some supporting software (notably the PyObjC distribution) and the modified VNC server and viewer source has been included.

This thesis is also included in both PDF format and as LaTeXsource. More information about the CD-ROM's contents can be found in the CD-ROM's read me.

Appendix B

Source code

This appendix contains (almost) all the source code developed as part of this master thesis. Please note that while some effort has been made to make lines fit within the “standard” page margins, this style has not been used everywhere, as the author prefers source code with longer lines. For best viewing, the digital copies should be studied, with the tab length set to 4 spaces.

For the window manager development, much code has been incorporated into existing source files. In many cases the changes amount to just one- or two-line additions/modifications. These files are not included in the following source listings, but can be found on the accompanying CD-ROM. Other files have had more extensive additions or changes. For these files, a lot of unrelated “junk” has been removed to emphasize the added code. In some cases, this also includes removing the function prologues and/or epilogues.

ppmsub.c 1/3

```
/*  ppmsub.c
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no

    A small utility to subtract one PPM file (the background) from another.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void    read_ppm(char *name, unsigned char **out_data, int *w, int *h, int *d);

typedef struct {
    int     r, g, b;
} rgb_t;

int main(int argc, char *argv[]) {
    unsigned char       *bg_data, *src_data, buf[50];
    int         bw, bh, bd, sw, sh, sd, increment, max_col_val, i, max_len, last
_pct, avg;
    FILE        *out;
    rgb_t       src_rgb, bg_rgb;

    if (argc < 3) {
        printf("Usage: %s <bg> <src> [dst]\n", argv[0]);
        printf("If no destination is given, the source is overwritten.\n");
        exit(1);
    }
    //  Read input, and verify that they are of the same format
    read_ppm(argv[1], &bg_data, &bw, &bh, &bd);
    read_ppm(argv[2], &src_data, &sw, &sh, &sd);
    if (bw != sw || bh != sh || bd != sd) {
        printf("Error: PPMs have mismatching height, width or depth : %d<->%d, %d<->%d, %d<->%d\n"
, bw, sw, bh, sh, bd, sd);
        exit(1);
    }

    //  Are we dealing with RGB values in the range 0-255 or 0-65535?
    if (sd < 256) {
        increment   = 3;
        max_col_val = 255;
    }
    else {
        increment   = 6;
        max_col_val = 65535;
        printf("Error: ppmsub currently doesn't support ppm files with RGB values in the range 0-65535.\n")
;
        exit(1);
    }

    //  Open output
    if (argc > 3)
        out = fopen(argv[3], "w+");
    else
        out = fopen(argv[2], "w+");
    sprintf(buf, "P6\n%d %d\n%d\n", sw, sh, sd);
    fwrite(buf, 1, strlen(buf), out);

    //  Process files
    i               = 0;
    max_len         = sw*sh*increment;
    last_pct        = 0;
    while (i < max_len) {
        if (increment == 6) {
            //  Not implemented - wasn't necessary .. :/
```

ppmsub.c 2/3

```
        }
        else {
            src_rgb.r   = src_data[i];
            src_rgb.g   = src_data[i+1];
            src_rgb.b   = src_data[i+2];
            bg_rgb.r    = bg_data[i];
            bg_rgb.g    = bg_data[i+1];
            bg_rgb.b    = bg_data[i+2];
        }
        //  Subtract background from source, and ensure the result is valid
        src_rgb.r   -= bg_rgb.r;
        src_rgb.g   -= bg_rgb.g;
        src_rgb.b   -= bg_rgb.b;
        src_rgb.r   = (src_rgb.r < 0 ? 0 : src_rgb.r);
        src_rgb.g   = (src_rgb.g < 0 ? 0 : src_rgb.g);
        src_rgb.b   = (src_rgb.b < 0 ? 0 : src_rgb.b);
        //  Average the results (this produces the best results for our purposes
)
        avg         = src_rgb.r + src_rgb.g + src_rgb.b;
        avg         /= 3;
        if (increment == 6) {
            //  Again, not implemented..
        }
        else {
            buf[0]      = (unsigned char)avg;
            buf[1]      = (unsigned char)avg;
            buf[2]      = (unsigned char)avg;
        }
        //  Update output
        fwrite(buf, 1, increment, out);
        i               += increment;
        if (i*100/max_len >= last_pct+10) {
            printf("\r%d..", i*100/max_len);
            fflush(stdout);
            last_pct    = i*100/max_len;
        }
    }
    printf("Done!\n");
    fflush(out);
    fclose(out);
    return 0;
}


void    read_ppm(char *name, unsigned char **out_data, int *w, int *h, int *d) {
    FILE    *f;
    int     len;
    unsigned char   *data, buf[20];

    f       = fopen(name, "r");
    *out_data   = 0;
    *w          = 0;
    *h          = 0;
    *d          = 0;
    if (!f) {
        printf("File %s not found\n", name);
        exit(1);
    }
    fseek(f, 0, SEEK_END);
    len     = ftell(f);
    rewind(f);
    data        = malloc(len+1);
    fread(data, 1, len, f);
    fclose(f);
    data[len]   = 0;
```

ppmsub.c 3/3

```
    //  A small state machine for parsing the input PPM file.
    if (data[0] == 'P' && data[1] == '6') {
        int state = 0, i = 0, j = 0;
        i   = 3;
        while (state < 3) {
            if (data[i] == ' ' || data[i] == '\n' || data[i] == '\t') {
                if (j) {
                    buf[j++]    = 0;
                    if (state == 0)
                        *w      = atoi(buf);
                    else if (state == 1)
                        *h      = atoi(buf);
                    else if (state == 2)
                        *d      = atoi(buf);

                    state++;
                }
                j   = 0;
            }
            else
                buf[j++] = data[i];
            i++;
        }
        *out_data   = &data[i];
    }
}
```

askpass.c 1/1

```
/*  askpass.c
    This simple program reads an environment variable called "ASKPASS_PASSWORD"
    and prints it to stdout, suitable for use in automated calls to ssh-add.

    By Daniel Stodle, daniels@stud.cs.uit.no
*/
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

extern char **environ;

const char  *env_var_name = "ASKPASS_PASSWORD";

int main(int argc, char *argv[]) {
    int     i = 0;
    char    *pw;

    while (environ[i]) {
        if (strncasecmp(env_var_name, environ[i], strlen(env_var_name)) == 0) {
            //  We found our password
            pw  = environ[i];
            pw  += strlen(env_var_name)+1;
            fprintf(stdout, "%s\n", pw);
            return 0;
        }
        i++;
    }
    fprintf(stderr, "No password found\n");
    fprintf(stdout, "\n");
    return -1;
}
```

main.m 1/1

```
//
//  main.m
//  Wall Manager
//
//  Created by Daniel St¿dle on 14-12-04.
//  Copyright __MyCompanyName__ 2004. All rights reserved.
//

#import <Cocoa/Cocoa.h>
#import "PythonGlue.h"

int main(int argc, char *argv[])
{
    NSAutoreleasePool   *p = [[NSAutoreleasePool alloc] init];

    [[PythonGlue alloc] init];
    return NSApplicationMain(argc,  (const char **) argv);
}
```

ProjControlButton.h 1/1

```
/*  ProjControlButton.h
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
*/
#ifndef PROJCONTROLBUTTON_H
    #define PROJCONTROLBUTTON_H

//  Imports
#import <Cocoa/Cocoa.h>
#import "wall_manager.h"

enum {
    kTag_turn_on        = 1,
    kTag_turn_off,
    kTag_toggle_eco_mode,
};


@interface ProjControlButton : NSButton {
    NSMenu          *popup;
    wall_manager    *ctrl;
}
- (id)initWithFrame:(NSRect)frame controller:(wall_manager*)controller andMenu:(
NSMenu*)menu;
- (void)modify_state:(id)sender;
@end
#endif
```

ProjControlButton.m 1/2

```
/*  ProjControlButton.m
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no

    Implementation for a simple button, which displays a popup menu when right-
    clicked, and ignores all other clicks, allowing the button state to indicate
    whether the projector it represents is on or off.
*/


#import "ProjControlButton.h"

@implementation ProjControlButton
- (id)initWithFrame:(NSRect)frame controller:(wall_manager*)controller andMenu:(
NSMenu*)menu {
    popup   = menu;
    ctrl    = controller;
    return [super initWithFrame:frame];
}


//  Ignore mouseUp and mouseDown events
- (void)mouseDown:(NSEvent*)evt { }
- (void)mouseUp:(NSEvent*)evt { }


//  ..But process rightMouseDown events, displaying our popup menu :)
- (void)rightMouseDown:(NSEvent*)evt {
    id <NSMenuItem> item;
    int             i;

    //  Find the first item of the popup menu. We use this item to indicate to t
he
    //  user which projector the popup menu belongs to.
    item    = [popup itemAtIndex:0];
    //  Set the item's title to something like "Projector 1.2"
    [item setTitle:[NSString stringWithFormat:@"Projector %@",[self title]]];
    [item setEnabled:NO];
    //  Set target and action methods of the popup menu to point to this object
instance.
    for (i=1;i<[popup numberOfItems]-1;i++) {   //  Eco mode config currently no
t supported
        item    = [popup itemAtIndex:i];
        [item setTarget:self];
        [item setAction:@selector(modify_state:)];
    }
    //  Finally, show the popup menu. We don't have to worry about it from this
    //  point.
    [NSMenu popUpContextMenu:popup withEvent:evt forView:self];
}


//  modify_state: This function takes care of receiving the menu item selected
//  by the user, and then taking the appropriate action.
- (void)modify_state:(id)sender {
    switch ([sender tag]) {
        case kTag_turn_on:
            //  Projector control is implemented by sending a message to the
            //  controller class, which then takes care of forwarding the messag
e
            //  to the python module. The reason we go via the controller class,
            //  is to allow the controller to update any status items (such as
            //  the text boxes indiciating projector state).
            if ([ctrl control_proj:[self title] turnOn:YES])
                [self setState:NSOnState];
            break;
```

ProjControlButton.m 2/2

```
        case kTag_turn_off:
            if ([ctrl control_proj:[self title] turnOn:NO])
                [self setState:NSOffState];
            break;
        case kTag_toggle_eco_mode:
            //  Unimplemented for now.
            break;
    }
}

@end
```

PythonGlue.h 1/1

```
#import <Cocoa/Cocoa.h>
@interface PythonGlue : NSObject
{
}
- init;
@end
```

PythonGlue.m 1/1

```
/*
PythonGlue is part of the PyObjC bridge.

PythonGlue is a class implementing a singleton object that does
nothing, but it has one side effect: it initializes Python (which
should be linked into the bundle containing this class) and executes
Contents/Resources/PythonGlue.py from the main bundle.

No error checking is done, but Python errors will result in messages
on standard error (or the console, for programs started from the Finder).
*/

#import <Foundation/Foundation.h>
#import "PythonGlue.h"
#import <Python/Python.h>
#import <stdio.h>

@implementation PythonGlue

- init
{
    static id _singleton;
    NSString *path;
    const char *c_path;
    FILE *fp;

    if (_singleton) return _singleton;
    _singleton = self;
    path = [[[NSBundle mainBundle] resourcePath]
              stringByAppendingPathComponent: @"PythonGlue.py"];
    c_path = [path cString];
    if ((fp=fopen(c_path, "r")) == NULL) {
        perror(c_path);
        return self;
    }
    Py_Initialize();
    PyRun_SimpleFile(fp, c_path);
    return self;
}

@end
```

wall_communicator.h 1/1

```
/*  wall_communicator.h
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no

    This file defines the interface to the wall_communicator class,
    which is implemented in python (see wall_communicator.py). Its
    purpose is mainly to declare the various interfaces exported by
    wall_communicator.py, so we avoid compiler warnings and get typesafety
    at the same time.
*/

#ifndef WALL_COMMUNICATOR_H
    #define WALL_COMMUNICATOR_H

//  Includes
    #import <Cocoa/Cocoa.h>

//  Class
@interface wall_communicator : NSObject {
}
- (IBAction)getConfig:(id)sender;
- (IBAction)startWall:(id)sender;
- (IBAction)stopWall:(id)sender;
- (NSNumber*)authenticate:(NSString*)user withPassword:(NSString*)password;
- (void)startVncServer:(NSString*)vnchost withDepth:(NSNumber*)depth andGeometry
:(NSString*)geometry onScreen:(NSNumber*)screen;
- (void)startProjectors:(NSString*)projhost whichProjector:(NSString*)which_proj
;
- (void)stopProjectors:(NSString*)projhost whichProjector:(NSString*)which_proj;
- (void)shutdown;
- (void)setColor:(NSNumber*)red green:(NSNumber*)green blue:(NSNumber*)blue;
- (void)setPattern:(NSNumber*)pat;
- (NSArray*)probeCluster;
- (void)startXdmx:(NSString*)host onScreen:(NSNumber*)screen;
@end
#endif
```

wall_manager.h 1/2

```
/*  wall_manager.h
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
    Best viewed with tabs = 4.
*/

#ifndef WALL_MANAGER_H
    #define WALL_MANAGER_H

//  Includes
    #import <Cocoa/Cocoa.h>
    #import "wall_communicator.h"

//  Constants
enum {
    kProjector_cool_down_interval   = 60,   //  Number of seconds to keep the
                                            //  projector control buttons
                                            //  disabled.

    kProbe_auth                     = 0,    //  Probe constants, used for
    kProbe_vnc_ssh,                         //  accessing the probe results
    kProbe_vnc,                             //  array after conducting a system
    kProbe_proj_ctrl,                       //  probe. Also used to control the
    kProbe_cluster_up,                      //  progress bar.
    kProbe_cluster_software_up,
    kNum_probes,

    kStart_vnc_viewer_action        = 1,    //  Actions to perform in case some
    kStart_projectors_action,               //  problems are encountered that
    kStop_projectors_action,                //  are not fatal. See the
    kStart_it_anyway_action,                //  sheet_ended method.
};

//  Class
@interface wall_manager : NSObject {
    IBOutlet wall_communicator      *wall_comm;     //  instance of the
                                                    //  wall_communicator object
    IBOutlet NSTextField            *geometry_fld,  //  outlets to various parts
                                    *res_fld,       //  of the Wall Manager
                                    *num_nodes_fld, //  interface.
                                    *vnchost_fld,
                                    *projhost_fld,
                                    *password_fld,
                                    *proj_status,
                                    *cluster_status,
                                    *wall_status;
    IBOutlet NSProgressIndicator    *auth_pgs,
                                    *wall_pgs,
                                    *proj_pgs,
                                    *cluster_pgs,
                                    *overall_pgs,
                                    *probe_pgs;
    IBOutlet NSPopUpButton          *depth_menu,
                                    *pattern_menu;
    IBOutlet NSTabView              *tab_view;
    IBOutlet NSTextView             *log_text;
    IBOutlet NSButton               *start_proj_btn,
                                    *stop_proj_btn,
                                    *toggle_detail_btn;
    IBOutlet NSView                 *proj_container_view;
    IBOutlet NSMenu                 *proj_control_menu;
    IBOutlet NSColorWell            *color_well;

    int             wall[2],    //  Number of projectors (X, Y)
                    res[2],     //  Resolution pr projector.
                    vncscreen;  //  Which display the VNC server should run on,
```

wall_manager.h 2/2

```
                                //  for instance host:1 => vncscreen = 1.
    BOOL            is_authenticated,   //  True when user has authenticated.
                    proj_screen_on;     //  True if we think the projectors are
on.
    NSString        *vnchost,   //  Hostname of the hosts running the VNC server
                    *projhost;  //  and controlling the projectors.
    NSDictionary    *mapping;   //  The mapping between hostnames and projector
IDs.
    time_t          last_projector_state_change;// Time when we last did
                                                // anything to the projectors.
    int             probe_state[kNum_probes];   // Results from the system probe
}
- (IBAction)reload_config:(id)sender;       //  Reloads the configuration from w
all_conf.py
- (IBAction)start_vnc_viewers:(id)sender;
- (IBAction)start_vnc_server:(id)sender;
- (IBAction)start_xdmx:(id)sender;          //  Not implemented.
- (IBAction)authenticate:(id)sender;
- (IBAction)toggle_detail:(id)sender;
- (IBAction)start_projectors:(id)sender;
- (IBAction)stop_projectors:(id)sender;
- (IBAction)start_everything:(id)sender;
- (IBAction)stop_everything:(id)sender;
- (IBAction)probe:(id)sender;
- (IBAction)open_terminal:(id)sender;
- (IBAction)set_color:(id)sender;
- (IBAction)set_pattern:(id)sender;

//  Helper methods for the system probe.
- (int)try_connect:(const char*)host port:(int)p;
- (int)probe_cluster;
- (void)sheet_ended:(NSWindow*)sheet returnCode:(int)code contextInfo:(void *)ct
x;

//  GUI stuff
- (void)update_fields;
- (void)reenable_projector_control:(NSTimer*)timer;
- (void)display_needs_authentication_msg;

//  Logs a message to the textfield in the log-tab.
- (void)log:(NSString*)msg;

- (void)update_proj_control;
- (NSButton*)create_proj_btn:(NSRect)frame;
- (BOOL)control_proj:(NSString*)name turnOn:(BOOL)on;
- (void)set_proj_button_state;
@end
#endif
```

wall_manager.m 1/10

```
/*  wall_manager.m
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
    Best viewed with tabs = 4.

    This file contains the implementation of the controller class for the
    Wall Manager application.

*/

#import     "wall_manager.h"
#import     "ProjControlButton.h"
#include    <netinet/in.h>
#include    <arpa/inet.h>
#include    <netdb.h>

@implementation wall_manager
//  awakeFromNib: Called by the runtime when the application starts. Used
//  to set the font for the log field, and reload the display wall config.
- (void)awakeFromNib {
    [log_text setFont:[NSFont fontWithName:@"Monaco" size:9]];
    [self reload_config:self];
}

//  reload_config: Reloads the display wall config from the wall_conf.py file.
//  Uses the PyObjC bridge to get access to the new configuration.
- (IBAction)reload_config:(id)sender {
    NSMutableDictionary     *dict = 0;

    dict    = [wall_comm getConfig:self];
    if (dict) {
        NSArray         *array;
        NSDictionary    *d2;
        NSNumber        *num;
        NSString        *str;

        //  Get config info from the returned dictionary
        array       = [dict objectForKey:@"wall"];
        wall[0]     = [[array objectAtIndex:0] intValue];
        wall[1]     = [[array objectAtIndex:1] intValue];
        array       = [dict objectForKey:@"resolution"];
        res[0]      = [[array objectAtIndex:0] intValue];
        res[1]      = [[array objectAtIndex:1] intValue];
        vnchost     = [dict objectForKey:@"vnchost"];
        vncscreen   = [[dict objectForKey:@"vncscreen"] intValue];
        projhost    = [dict objectForKey:@"projhost"];
        mapping     = [dict objectForKey:@"mapping"];
        //  Retain these things so we don't lose them
        [vnchost retain];
        [projhost retain];
        [mapping retain];
        //  Update text fields and projector buttons.
        [self update_fields];
        [self update_proj_control];
    }
}


- (IBAction)start_vnc_viewers:(id)sender {
    [self log:@"Starting VNC viewers.\n"];
    [wall_comm startWall:self];
}


- (IBAction)start_vnc_server:(id)sender {
    if (is_authenticated) {
```

wall_manager.m 2/10

```
        [self log:[NSString stringWithFormat:@"Starting VNC server on %@.\n", vnchost]]
;
        [wall_pgs startAnimation:self];
        [wall_comm startVncServer:vnchost withDepth:[NSNumber numberWithInt:[[de
pth_menu selectedItem] tag]]
                    andGeometry:[NSString stringWithFormat:@"%dx%d",res[0]*wall[
0], res[1]*wall[1]]
                    onScreen:[NSNumber numberWithInt:vncscreen]];
        [wall_pgs stopAnimation:self];
    }
    else
        [self display_needs_authentication_msg];
}


- (IBAction)start_xdmx:(id)sender {
    [self log:@"Xdmx must be started manually, for the time being.\n"];
/*  if (is_authenticated) {
        [self log:[NSString stringWithFormat:@"Starting Xdmx on %@.\n", vnchost]
];
        [wall_pgs startAnimation:self];
        [wall_comm startXdmx:vnchost onScreen:[NSNumber numberWithInt:vncscreen]
];
        [wall_pgs stopAnimation:self];
    }
    else
        [self display_needs_authentication_msg];
*/
}


- (IBAction)authenticate:(id)sender {
    NSNumber    *result;
    //  Give user feedback - start progress indicator
    [auth_pgs startAnimation:self];
    //  Authenticate with the given password
    result      = [wall_comm authenticate:@"username-not-used" withPassword:[passwo
rd_fld stringValue]];
    //  Stop progress indicator
    [auth_pgs stopAnimation:self];
    //  Check result
    if (![result boolValue]) {
        NSBeginAlertSheet(@"Failed to authenticate", @"OK", 0, 0, [NSApp mainWindow], 0
, 0, 0, 0,
        @"Authentication failed! Verify your password, check that you have generated ssh-keys "
        "and verify the permissions on your home directory and the .ssh directory.");
    }
    else {
        [self log:@"User authenticated.\n"];
        is_authenticated    = YES;
        //  Disable the authenticate-button - no need to authenticate more than
once!
        [sender setEnabled:NO];
    }
}


//  toggle_detail: Resizes the window to either show or hide the "detail" portio
n
//  of the window.
- (IBAction)toggle_detail:(id)sender {
    NSWindow        *w = [NSApp mainWindow];
    NSRect          f, sf;
    int             y, h;
```

wall_manager.m 3/10

```
    f   = [w frame];
    if ([sender state] == NSOffState) {
        sf              = [sender frame];
        y               = f.size.height - sf.origin.y;
        h               = f.size.height;
        f.size.height   = y+10;
        f.origin.y      = f.origin.y + h - f.size.height;
        [tab_view setHidden:YES];
        [w setFrame:f display:YES animate:YES];
    }
    else {
        sf              = [tab_view frame];
        h               = f.size.height;
        f.size.height   += sf.size.height+10;
        f.origin.y      = f.origin.y + h - f.size.height;
        [w setFrame:f display:YES animate:YES];
        [tab_view setHidden:NO];
    }
}


- (IBAction)start_projectors:(id)sender {
    if (is_authenticated) {
        //  Verify that we can indeed change the state at this point in time
        if (last_projector_state_change+kProjector_cool_down_interval <= time(0)
) {
            [self log:@"Starting projectors..\n"];
            //  Disable start and stop buttons. The projectors need to wait for
about 60
            //  seconds after being started or stopped, before their state can b
e re-altered.
            [start_proj_btn setEnabled:NO];
            [stop_proj_btn setEnabled:NO];
            //  Start progress feedback, and start the projectors.
            [proj_pgs startAnimation:self];
            [wall_comm startProjectors:projhost whichProjector:0];
            [proj_pgs stopAnimation:self];
            //  Schedule a timer to fire in 60 seconds, reenabling the projector
 control buttons.
            last_projector_state_change = time(0);
            [NSTimer scheduledTimerWithTimeInterval:60.0 target:self selector:@s
elector(reenable_projector_control:) userInfo:0 repeats:NO];
            proj_screen_on  = YES;
            [proj_status setStringValue:@"Status: On."];
            [self set_proj_button_state];
        }
        else {
            NSBeginAlertSheet(@"Too soon!", @"OK", 0, 0, [NSApp mainWindow], 0, 0
, 0, @"You need to wait a little while before doing anything with the projectors.");
        }
    }
    else
        [self display_needs_authentication_msg];
}


- (IBAction)stop_projectors:(id)sender {
    if (is_authenticated) {
        if (last_projector_state_change+kProjector_cool_down_interval <= time(0)
) {
            [self log:@"Stopping projectors..\n"];
            [start_proj_btn setEnabled:NO];
            [stop_proj_btn setEnabled:NO];
            [proj_pgs startAnimation:self];
            [wall_comm stopProjectors:projhost whichProjector:0];
```

wall_manager.m 4/10

```
            [proj_pgs stopAnimation:self];
            last_projector_state_change = time(0);
            [NSTimer scheduledTimerWithTimeInterval:60.0 target:self selector:@s
elector(reenable_projector_control:) userInfo:0 repeats:NO];
            proj_screen_on  = NO;
            [proj_status setStringValue:@"Status: Off."];
            [self set_proj_button_state];
        }
        else {
            NSBeginAlertSheet(@"Too soon!", @"OK", 0, 0, [NSApp mainWindow], 0, 0
, 0, @"You need to wait a little while before doing anything with the projectors.");
        }
    }
    else
        [self display_needs_authentication_msg];
}


//  start_everything: Attempts to start all the necessary components in order to
//  get the display wall up and running. Starts out by probing the system,
//  inspecting the probe results and then deciding on what needs to be done.
- (IBAction)start_everything:(id)sender {
    int     i;
    BOOL    ready_to_go = YES;

    [self log:@"Starting display wall.\n"];
    //  Set user feedback progress bar's max value
    [overall_pgs setDoubleValue:0.0];
    [overall_pgs setMaxValue:2+kNum_probes];
    [overall_pgs setNeedsDisplay:YES];
    [overall_pgs displayIfNeeded];
    //  Perform the system probe
    [self probe:self];
    [overall_pgs setDoubleValue:1.0];
    [self log:@"Figuring out what we need to boot.\n"];
    for (i=0;i<kNum_probes && ready_to_go;i++) {
        [overall_pgs setDoubleValue:1.0+i];
        [overall_pgs displayIfNeeded];
        //  The probe_state array contains a non-zero value if the given probe f
ailed.
        if (probe_state[i] != 0) {
            switch (i) {
                case kProbe_auth:
                    if (probe_state[kProbe_vnc]) {
                        NSBeginAlertSheet(@"Authentication requrired!", @"OK",
                        0, 0, [NSApp mainWindow], 0, 0, 0, 0,
                        @"You need to authenticate before starting the wall, as "
                        "there is no VNC server currently running. Authentication "
                        "is also required for starting the projectors.");
                    }
                    else {
                        NSBeginAlertSheet(@"Authentication requrired!", @"OK",
                        @"Start viewers", 0, [NSApp mainWindow], self, 0,
                        @selector(sheet_ended:returnCode:contextInfo:),
                        (void*)kStart_vnc_viewer_action,
                        @"You need to authenticate before starting the wall. "
                        "If all you need to do is start the vnc viewers, click "
                        "Start viewers (a VNC server already seems to be running).");
                    }
                    [self log:@"Boot not complete - not authenticated.\n"];
                    ready_to_go = NO;
                    break;
                case kProbe_vnc_ssh:
                    NSBeginAlertSheet(@"Link to VNC computer down", @"Will do!",
                    0, 0, [NSApp mainWindow], 0, 0, 0, 0,
```

wall_manager.m 5/10

```
                    [NSString stringWithFormat:@"The vnc computer (%@) is down "
                    "or not responding. Can you please reboot it before "
                    "continuing?", vnchost]);
                    [self log:@"VNC computer seems to be down. "
                    "Boot not completed.\n"];
                    ready_to_go = NO;
                    break;
                case kProbe_vnc:
                    //  No VNC server - try starting one.
                    [self start_vnc_server:self];
                    break;
                case kProbe_proj_ctrl:
                    NSBeginAlertSheet(@"Link to projector computer down", @"OK",
                    0, 0, [NSApp mainWindow], 0, 0, 0, 0,
                    [NSString stringWithFormat:@"The projector computer (%@) is "
                    "down or not responding. It will need to be rebooted in "
                    "order to start the projectors.", projhost]);
                    [self log:@"The projector computer seems to be down. "
                    "Boot not completed.\n"];
                    ready_to_go = NO;
                case kProbe_cluster_software_up:
                    //  Cluster is partially down. Let the user decide if she
                    //  wants to proceed with the display wall startup, or not.
                    NSBeginAlertSheet(@"Cluster partially down",
                    @"Abort start-up", @"Start wall anyway", 0, [NSApp mainWindow],
                    self, 0, @selector(sheet_ended:returnCode:contextInfo:),
                    (void*)kStart_it_anyway_action,
                    @"One or more of the cluster computers are not responding! "
                    "Please check the log for detailed information. You can "
                    "choose to abort the boot process, or start the components "
                    "that can be started anyway.");
                    ready_to_go = NO;
            }
        }
    }
    if (ready_to_go) {
        [overall_pgs setDoubleValue:1.0+i];
        if (!probe_state[kProbe_proj_ctrl]) {
            if (!proj_screen_on)
                [self start_projectors:self];
            else
                NSBeginAlertSheet(@"Turn on projectors?", @"Leave projectors as is",
                @"Start projectors", 0, [NSApp mainWindow], self, 0,
                @selector(sheet_ended:returnCode:contextInfo:),
                (void*)kStart_projectors_action,
                @"The projectors appear to be already running. If this is not "
                "the case, click Start projectors.");
        }
        [self start_vnc_viewers:self];
    }
    [overall_pgs setDoubleValue:[overall_pgs maxValue]];
    NSBeginAlertSheet(@"Everything OK", @"Alright!", 0, 0, [NSApp mainWindow],
    0, 0, 0, 0,
    @"The display wall has been successfully started.");
}


- (void)sheet_ended:(NSWindow*)sheet returnCode:(int)code contextInfo:(void *)ct
x {
    if (code == NSAlertAlternateReturn) {
        switch((int)ctx) {
            case kStart_vnc_viewer_action:
                [self start_vnc_viewers:self];
                break;
            case kStart_projectors_action:
```

wall_manager.m 6/10

```
                [self start_projectors:self];
                break;
            case kStop_projectors_action:
                [self stop_projectors:self];
                break;
            case kStart_it_anyway_action:
                [self start_projectors:self];
                [self start_vnc_viewers:self];
                NSBeginAlertSheet(@"Everything done", @"OK", 0, 0, [NSApp mainWindo
w], 0, 0, 0, 0,
                @"As much of the display wall as possible has been started.");
                break;
        }
    }
}


- (IBAction)stop_everything:(id)sender {
    if (proj_screen_on)
        [self stop_projectors:self];
    else {
        NSBeginAlertSheet(@"Turn off projectors?", @"Leave projectors as is",
        @"Stop projectors", 0, [NSApp mainWindow], self, 0,
        @selector(sheet_ended:returnCode:contextInfo:),
        (void*)kStop_projectors_action,
        @"The projectors appear to be stopped already. If this is not the case, "
        "click Stop projectors.");
    }
}


//  probe: Conducts the system probe.
- (IBAction)probe:(id)sender {
    int                     i, vncport = vncscreen + 6000;

    [probe_pgs setMaxValue:kNum_probes];
    [probe_pgs setDoubleValue:0.0];
    for (i=0;i<kNum_probes;i++)
        probe_state[i]  = 0;

    i   = 0;
    [self log:@"Running probe..\n"];
    [self log:[NSString stringWithFormat:@"     Authenticated: %s\n", (is_authenticat
ed ? "YES" : "NO")]];
    probe_state[kProbe_auth]        = is_authenticated ? 0 : -1;
    [probe_pgs setDoubleValue:++i];

    //  Try connecting on port 22 to the VNC computer (checks for SSH):
    probe_state[kProbe_vnc_ssh]     = [self try_connect:[vnchost cString] port:2
2];
    [self log:[NSString stringWithFormat:@"   SSH to VNC computer: %s\n", (!probe_stat
e[kProbe_vnc_ssh] ? "YES" : "NO")]];
    [probe_pgs setDoubleValue:++i];

    //  Try connecting on port 6000+vncscreen to the VNC computer (checks for ru
nning VNC server):
    probe_state[kProbe_vnc]         = [self try_connect:[vnchost cString] port:v
ncport];
    [self log:[NSString stringWithFormat:@"  VNC server is running: %s\n", (!probe_state
[kProbe_vnc] ? "YES" : "NO")]];
    [probe_pgs setDoubleValue:++i];

    //  Try connecting on port 22 to the projector control computer (checks for
SSH):
    probe_state[kProbe_proj_ctrl]   = [self try_connect:[projhost cString] port:
```

wall_manager.m 7/10

```
22];
    [self log:[NSString stringWithFormat:@"SSH to projector control: %s\n", (!probe_state
[kProbe_proj_ctrl] ? "YES" : "NO")]];
    [probe_pgs setDoubleValue:++i];

    //  Probe cluster software.
    probe_state[kProbe_cluster_software_up] = [self probe_cluster];
    [self log:[NSString stringWithFormat:@"Cluster software running: %s\n", (!probe_state
[kProbe_cluster_software_up] ? "YES" : "NO")]];
    [probe_pgs setDoubleValue:++i];

    while (i<kNum_probes)
        [probe_pgs setDoubleValue:++i];

    [self log:@"Probing complete.\n"];
}


- (IBAction)open_terminal:(id)sender {
    [[NSWorkspace sharedWorkspace] launchApplication:@"Terminal"];
}


- (IBAction)set_color:(id)sender {
    NSColor     *color;
    NSNumber    *red, *green, *blue;

    color   = [color_well color];
    red     = [NSNumber numberWithInt:([color redComponent]*65535)];
    green   = [NSNumber numberWithInt:([color greenComponent]*65535)];
    blue    = [NSNumber numberWithInt:([color blueComponent]*65535)];

    [wall_comm setColor:red green:green blue:blue];
}


- (IBAction)set_pattern:(id)sender {
    [wall_comm setPattern:[NSNumber numberWithInt:[[pattern_menu selectedItem] t
ag]]];
}


//  try_connect: Attempts to open a connection to <host> on port <p>, closing th
e
//  socket after the attempt. A very very simple port scanner..
- (int)try_connect:(const char*)host port:(int)p {
    int     sock, ret_val = 0;
    struct sockaddr_in      addr;
    struct hostent          *he;
    in_addr_t               ip;

    sock    = socket(AF_INET, SOCK_STREAM, 0);
    if (!sock)
        return -1;

    he      = gethostbyname(host);
    if (he) {
        memset(&addr, 0, sizeof(struct sockaddr_in));
        addr.sin_family         = AF_INET;
        addr.sin_port           = htons((unsigned short)p);
        ip                      = *((in_addr_t*)he->h_addr_list[0]);
        addr.sin_addr.s_addr    = ip;
        if (connect(sock, (struct sockaddr*)&addr, sizeof(struct sockaddr)) != 0
)
            ret_val = -1;
```

wall_manager.m 8/10

```
    }
    else
        ret_val = -1;
    close(sock);
    return ret_val;
}


//  probe_cluster: Check which cluster nodes are responding to "identify" reques
ts.
- (int)probe_cluster {
    NSDictionary    *dict;
    NSArray         *array, *map_array;
    NSNumber        *proj;
    NSString        *str1;
    int             proj_count = 0, i, j, *found;

    //  Get hosts that respond to "identify" request:
    array   = [wall_comm probeCluster];
    //  Count number of projectors:
    for (i=0;i<[array count];i++) {
        dict    = [array objectAtIndex:i];
        proj    = [dict objectForKey:@"num_projs"];
        proj_count  += [proj intValue];
    }
    printf("Projectors found: %d\n", proj_count);
    //  If we have the projectors we need - good! No further checking is done.
    if (proj_count == wall[0]*wall[1])
        return 0;
    else {
        //  Figure out which node(s) are missing
        [self log:@"\nWarning: One or more cluster nodes are not responding!\n"];
        //  This algorithm is inefficient, but luckily computers are fast these
        //  days, so a worstcase N^2 algo doesn't matter in this case. Basically
,
        //  we compare every element in the array of hosts we have discovered, t
o
        //  all the hostnames in the mapping dictionary, checking off matches as
 we go.
        map_array   = [mapping allKeys];
        found       = calloc([map_array count], sizeof(int));
        for (i=0;i<[array count];i++) {
            dict    = [array objectAtIndex:i];
            str1    = [dict objectForKey:@"hostname"];
            for (j=0;j<[map_array count];j++) {
                if (found[j])
                    continue;
                if ([str1 caseInsensitiveCompare:[map_array objectAtIndex:j]] ==
 NSOrderedSame) {
                    found[j]    = 1;
                    break;
                }
            }
        }
        //  List any hosts that aren't responding
        for (i=0;i<[map_array count];i++) {
            if (!found[i])
                [self log:[NSString stringWithFormat:@"Not responding: %@\n", [map_a
rray objectAtIndex:i]]];
        }
        free(found);
        [self log:@"The machines listed above are either powered off, or the "
        "slave software is not running. Have an administrator investigate, and "
        "set the run level on the nodes to 5.\n\n"];
        return -1;
```

wall_manager.m 9/10

```
    }
}


//  update_fields: Updates the fields in the configuration tab according to the
//  current configuration.
- (void)update_fields {
    [geometry_fld setStringValue:[NSString stringWithFormat:@"%dx%d projectors", wa
ll[0], wall[1]]];
    [res_fld setStringValue:[NSString stringWithFormat:@"%dx%d pixels (%dx%d per proje
ctor)", res[0]*wall[0], res[1]*wall[1], res[0], res[1]]];
    [vnchost_fld setStringValue:vnchost];
    [projhost_fld setStringValue:projhost];
    [num_nodes_fld setIntValue:[mapping count]];
}


//  reenable_projector_control: Enables the projector control buttons once the
//  timer to do so expires.
- (void)reenable_projector_control:(NSTimer*)timer {
    [start_proj_btn setEnabled:YES];
    [stop_proj_btn setEnabled:YES];
}


- (void)display_needs_authentication_msg {
    NSBeginAlertSheet(@"Not authenticated", @"OK", 0, 0, [NSApp mainWindow],
    0, 0, 0, 0, @"You are not yet authenticated to perform this operation!");
}


- (void)log:(NSString*)msg {
    NSRange     end_range;

    end_range.location  = [[log_text textStorage] length];
    end_range.length    = 0;
    [log_text replaceCharactersInRange:end_range withString:msg];

    end_range.length    = [msg length];
    [log_text scrollRangeToVisible:end_range];
}


//  update_proj_control: This badly named method creates the line-up of buttons
//  in the Detailed projector control tab, adjusting their sizes and positions
//  to fit the current projector geometry.
- (void)update_proj_control {
    NSButton        *btn;
    NSRect          btn_rect, content_rect, f;
    int             i, j;

    content_rect    = [proj_container_view bounds];
    btn_rect.origin.x       = 0;
    btn_rect.origin.y       = 0;
    btn_rect.size.width     = (content_rect.size.width - 20 - 5*wall[0]) / wall[
0];
    btn_rect.size.height    = (content_rect.size.height - 20 - 5*wall[1]) / wall
[1];

    for (i=0;i<wall[1];i++) {
        for (j=0;j<wall[0];j++) {
            f           = btn_rect;
            f.origin.x  = 10 + j*(5+btn_rect.size.width);
            f.origin.y  = content_rect.size.height - (10 + btn_rect.size.height
+ i*(5+btn_rect.size.height));
```

wall_manager.m 10/10

```
            btn     = [self create_proj_btn:f];
            [btn setTitle:[NSString stringWithFormat:@"%d.%d",i+1,j+1]];
            //[btn setMenu:proj_control_menu];
            [proj_container_view addSubview:btn];
            [btn setFrame:f];
        }
    }
}


//  create_proj_button: Helper method to create a projector control button.
- (NSButton*)create_proj_btn:(NSRect)frame {
    NSButton        *btn;
    btn = [[ProjControlButton alloc] initWithFrame:frame controller:self andMenu
:proj_control_menu];
    [btn setButtonType:NSOnOffButton];
    [btn setImagePosition:NSNoImage];
    [btn setBordered:YES];
    [btn setBezelStyle:NSShadowlessSquareBezelStyle];
    return btn;
}


//  control_proj:turnOn: Turns the given projector on or off, according to the
//  turnOn parameter.
- (BOOL)control_proj:(NSString*)name turnOn:(BOOL)on {
    if (is_authenticated) {
        if ((proj_screen_on && !on) || (!proj_screen_on && on))
            [proj_status setStringValue:@"Status: Mixed"];
        [self log:[NSString stringWithFormat:@"Changing state for projector %@ to %s\n", na
me, (on ? "on" : "off")]];
        if (on)
            [wall_comm startProjectors:projhost whichProjector:name];
        else
            [wall_comm stopProjectors:projhost whichProjector:name];
        return YES;
    }
    else {
        [self display_needs_authentication_msg];
        return NO;
    }
}

- (void)set_proj_button_state {
    int         state = (proj_screen_on ? NSOnState : NSOffState);
    NSButton    *btn;
    NSArray     *array;
    int         i;

    array   = [proj_container_view subviews];
    for (i=0;i<[array count];i++) {
        btn = [array objectAtIndex:i];
        [btn setState:state];
    }
}


//  Called by the runtime when the user has asked Wall Manager to quit. Used to
//  call the python-bridge, and have it terminate any running ssh-agent.
- (void)applicationWillTerminate:(NSNotification*)notif {
    [wall_comm shutdown];
}

@end
```

event.c 1/2

```
/* event.c- event loop and handling

    NOTE: This file has been cut down to show the relevant modifications to
    the Window Maker source code. Please see the CD-ROM for the complete source
    listing.
*/

//  DST: Group includes
#include "group.h"
#define XK_MISCELLANY
#include "X11/keysymdef.h"
extern WMRect       selected_rect;


//  DST: Add group support
static int try_group_key(XEvent *event) {
    KeySym  ks;
    int     fkey;
    WScreen *scr        = wScreenForRootWindow(event->xkey.root);
    WWindow *wwin       = scr->focused_window;
    int     modifiers   = event->xkey.state & ValidModMask;

    ks      = XKeycodeToKeysym(dpy, event->xkey.keycode, 0);
    fkey    = ks - XK_F1;
    //printf("%X %d\n", ks, fkey);
    if (fkey >= 0 && fkey < kNum_groups) {
        printf("Caught FKEY: %d\n", fkey);

        if (modifiers & ShiftMask) {
            if (scr->selected_windows) {
                WWindow             *tmpw;
                WMArrayIterator     iter;
                free_group(fkey);
                WM_ITERATE_ARRAY(scr->selected_windows, tmpw, iter) {
                    assign_group(fkey, &selected_rect, tmpw, 0);
                }
            }
            else {
                free_group(fkey);
                assign_group(fkey, 0, wwin, 0);
            }
        }
        else if (modifiers & ControlMask)
            select_group(fkey);
        else if (modifiers == 0)
            teleport_group(fkey, event->xkey.x_root, event->xkey.y_root);
        else
            return 0;
        return 1;
    }
    return 0;
}


void
DispatchEvent(XEvent *event)
{
    //  NOTE: First part of function snipped - see CD-ROM for complete source
    switch (event->type) {
        //  Snipped lots of case's for handling different event types, again,
        //  see CD-ROM for complete source
        //  DST: Added checking for the conduit window here, and process the eve
nt!
        case EnterNotify:
            if (event->xcrossing.window == conduit_win) {
```

event.c 2/2

```
                if (event->xcrossing.focus) {
                    wSetFocusTo(wScreenForWindow(event->xcrossing.subwindow), wW
indowFor(event->xcrossing.subwindow));
                    XFlush(dpy);
                }
                XTestFakeKeyEvent(dpy, (event->xcrossing.x << 16) | event->xcros
sing.y, event->xcrossing.same_screen, CurrentTime);
                XFlush(dpy);
                break;
            }
            handleEnterNotify(event);
        break;
        //  Snipped rest of case statements
    }
}


static void
handleKeyPress(XEvent *event)
{
    WScreen *scr = wScreenForRootWindow(event->xkey.root);
    WWindow *wwin = scr->focused_window;
    int i;
    int modifiers;
    int command=-1, index;
#ifdef KEEP_XKB_LOCK_STATUS
    XkbStateRec staterec;
#endif /*KEEP_XKB_LOCK_STATUS*/
    // DST: Check if the key is a group assignment key
    if (try_group_key(event))
        return;

    //  REST OF FUNCTION SNIPPED - see complete source on CD-ROM.
}
```

group.c 1/4

```
/*  group.c
    (c) 2004-2005 Daniel Stodle, daniels@stud.cs.uit.no

    Window groups.
*/

#include "group.h"
#include "multi.h"
#define XK_MISCELLANY
#include "X11/keysymdef.h"


group_t             *groups;
Window              conduit_win;
static char         *names[kNum_groups] = {"red","green","blue","cyan",
                                           "magenta","yellow","navy","violet",
                                           "brown","pink","wheat","salmon"};
void        init_groups(void) {
    int     i, idx, value;
    XColor  col, dummy;
    WScreen *scr;

    printf("Group support initializing!\n");
    scr     = wScreenWithNumber(0);
    groups  = calloc(kNum_groups, sizeof(group_t));
    for (i=0;i<kNum_groups;i++) {
        //  Allocate array for windows
        groups[i].windows   = WMCreateArray(5);
        //  Allocate color for group
        //  Make sure we don't create a black color (always have one of the lowe
r three bits set)
        XAllocNamedColor(dpy, scr->w_colormap, names[i], &col, &dummy);

        groups[i].color = col;
    }
    init_multi_input();
    Window                  win;
    XSetWindowAttributes    attr;
    int                     mask;
    mask                    = CWOverrideRedirect | CWSaveUnder;
    attr.override_redirect  = 1;
    attr.save_under         = True;
    conduit_win             = XCreateWindow(dpy, RootWindow(dpy,0), 0, 0, 2, 2,
2, CopyFromParent, InputOutput, CopyFromParent, mask, &attr);
}


void        assign_group(int gid, WMRect *area, WWindow *win, int remove_others)
 {
    if (gid >= 0 && gid < kNum_groups && win) {
        if (remove_others)
            free_group(gid);

        if (win->gid != -1)
            remove_from_group(win);

        WMAddToArray(groups[gid].windows, win);
        win->gid    = gid;
        set_border_considering_group(win);
        if (!groups[gid].focused_win)
            groups[gid].focused_win = win;
        if (area)
            groups[gid].area    = *area;
        else {
            //  Expand the group rect, if necessary.
```

group.c 2/4

```
            WMArrayIterator iter;
            WWindow         *tmpw;
            int             l = win->frame_x, t = win->frame_y,
                            b = win->frame_y+win->frame->core->height,
                            r = win->frame_x+win->frame->core->width;

            WM_ITERATE_ARRAY(groups[gid].windows, tmpw, iter) {
                if (tmpw->frame_x < l)
                    l   = tmpw->frame_x;
                if (tmpw->frame_y < t)
                    t   = tmpw->frame_y;
                if (tmpw->frame_y+tmpw->frame->core->height > b)
                    b   = tmpw->frame_y+tmpw->frame->core->height;
                if (tmpw->frame_x+tmpw->frame->core->width > r)
                    r   = tmpw->frame_x+tmpw->frame->core->width;
            }
            //  Take the existing rect into account, if it exists
            if (groups[gid].area.size.width && groups[gid].area.size.height) {
                if (groups[gid].area.pos.x < l)
                    l   = groups[gid].area.pos.x;
                if (groups[gid].area.pos.y < t)
                    t   = groups[gid].area.pos.y;
                if (groups[gid].area.pos.x+groups[gid].area.size.width > r)
                    r   = groups[gid].area.pos.x+groups[gid].area.size.width;
                if (groups[gid].area.pos.y+groups[gid].area.size.height > b)
                    b   = groups[gid].area.pos.y+groups[gid].area.size.height;
            }
            set_rect(groups[gid].area, l, t, r-l, b-t);
        }
    }
}


void        remove_from_group(WWindow *win) {
    if (win && win->gid >= 0 && win->gid < kNum_groups) {
        //  Remove it
        WMRemoveFromArrayMatching(groups[win->gid].windows, 0, win);
        if (win == groups[win->gid].focused_win)
            groups[win->gid].focused_win = WMGetFromArray(groups[win->gid].windo
ws, 0);

        win->gid    = -1;
        set_border_considering_group(win);
    }
}


void        free_group(int gid) {
    WMArrayIterator iter;
    WWindow *tmpw;
    set_rect(groups[gid].area,0,0,0,0);
    if (gid >= 0 && gid < kNum_groups) {
        WM_ITERATE_ARRAY(groups[gid].windows, tmpw, iter) {
            tmpw->gid   = -1;
            set_border_considering_group(tmpw);
        }
        WMEmptyArray(groups[gid].windows);
        groups[gid].focused_win = 0;
        set_rect(groups[gid].area,0,0,0,0);
    }
}

WMArray*    windows_for_group(int gid) {
    if (gid >= 0 && gid < kNum_groups)
        return groups[gid].windows;
```

group.c 3/4

```
    return 0;
}


WMRect*     area_for_group(int gid) {
    if (gid >= 0 && gid < kNum_groups)
        return &groups[gid].area;
    return 0;
}



void        bind_group_shortcuts(Window w) {
    int     i;
    KeySym  fkey = XK_F1;

    for (i=0;i<10;i++)
        XGrabKey(dpy, XKeysymToKeycode(dpy, fkey+i), AnyModifier, w, True, GrabM
odeAsync, GrabModeAsync);
}


void        set_border_considering_group(WWindow *win) {
    int     border_width    = 2;
    if (win->gid != -1) {
        //if (WFLAGP(win, no_border))
        XSetWindowBorderWidth(dpy, win->frame->core->window, border_width);
        XSetWindowBorder(dpy, win->frame->core->window, groups[win->gid].color.p
ixel);
    }
    else {
        if (win->flags.selected)
            XSetWindowBorder(dpy, win->frame->core->window, win->screen_ptr->whi
te_pixel);
        else {
            XSetWindowBorder(dpy, win->frame->core->window, win->screen_ptr->fra
me_border_pixel);
            XSetWindowBorderWidth(dpy, win->frame->core->window, 0);
        }
    }
}


void        teleport_group(int gid, int x, int y) {
    WWindow *win;
    int     rel_x, rel_y, i, min_x, min_y, count;
    WMRect  area;

    printf("Teleporting group %d to %d, %d\n", gid, x, y);
    if (gid >= 0 && gid < kNum_groups) {
        count   = WMGetArrayItemCount(groups[gid].windows);
        if (count <= 0) {
            printf("Group is empty - nothing to teleport\n");
            return;
        }
        win     = WMGetFromArray(groups[gid].windows, 0);
        min_x   = win->frame_x;
        min_y   = win->frame_y;
        for (i=1;i<count;i++) {
            win     = WMGetFromArray(groups[gid].windows, i);
            if (win->frame_x < min_x)
                min_x   = win->frame_x;
            if (win->frame_y < min_y)
                min_y   = win->frame_y;
        }
```

group.c 4/4

```
        rel_x   = x - min_x;
        rel_y   = y - min_y;
        doWindowMove(dpy, win, groups[gid].windows, rel_x, rel_y, &groups[gid].a
rea, 1);
    }
}


void        select_group(int gid) {
    printf("Focusing group %d\n", gid);
    if (gid >= 0 && gid < kNum_groups) {
        if (groups[gid].focused_win) {
            printf("Group has window with focus, raising it and setting input focus\n");
            wSetFocusTo(groups[gid].focused_win->screen_ptr, groups[gid].focused
_win);
            wRaiseFrame(dpy, groups[gid].focused_win->frame->core);
            //XRaiseWindow(dpy, groups[gid].focused_win->frame->core->window);
            //XSetInputFocus(dpy, win->frame->core->window, RevertToParent, Curr
entTime);
        }
    }
}


void        set_focused_win(WWindow *win) {
    printf("Set focused win\n");
    if (win->gid >= 0 && win->gid < kNum_groups) {
        printf("Setting group %d focus\n", win->gid);
        groups[win->gid].focused_win    = win;
        multi[win->gid].focus           = win->client_win; //win->frame->core->w
indow;
    }
    else
        printf("Warning, window doesn't belong to a group!\n");
}
```

group.h 1/1

```
/*  group.h
    (c) 2004-2005 Daniel Stodle, daniels@stud.cs.uit.no
    Header file supporting window groups.
*/

#ifndef GROUP_H
    #define GROUP_h

//  Includes
    #include "window.h"
    #include "wcore.h"
    #include "framewin.h"
    #include "WindowMaker.h"
    #include "screen.h"

//  Constants
enum {
    kNum_groups     = 10,
};

//  Macros
#define set_rect(r, xx, yy, ww, hh) (r.pos.x=xx,r.pos.y=yy,r.size.width=ww,r.siz
e.height=hh)


//  Structs
typedef struct _group_t {
    WMRect      area;
    WMArray     *windows;
    WWindow     *focused_win;
    XColor      color;
} group_t;

//  Exported vars
extern group_t  *selected_group;
extern group_t  *groups;
extern Window   conduit_win;

//  Prototypes
void        init_groups(void);
void        assign_group(int gid, WMRect *area, WWindow *win, int remove_others)
;
void        remove_from_group(WWindow *win);
void        free_group(int gid);
WMArray*    windows_for_group(int gid);
WMRect*     area_for_group(int gid);
void        bind_group_shortcuts(Window w);
void        set_border_considering_group(WWindow *win);
void        teleport_group(int gid, int x, int y);
void        select_group(int gid);
void        set_focused_win(WWindow *win);
#endif
```

moveres.c 1/2

```
/* moveres.c
    NOTE: This file has been cut down to show the relevant modifications to
    the Window Maker source code. Please see the CD-ROM for the complete source
    listing.
 */

//  DST: Group includes
#include "group.h"
#include "moveres.h"

/*  DST
    Added rectangle to keep track of where the selection happened, so that we ca
n
    move individual windows within that rectangle without moving _all_ of them.
*/

WMRect      selected_rect = {0,0,0,0};


//  DST: Modified to take display-var as argument, in addition to handling group
s.
void
doWindowMove(Display *which_dpy, WWindow *wwin, WMArray *array, int dx, int dy,
WMRect *move_rect, int retain_relative_positions)
{
    WWindow *tmpw;
    int x, y;
    int scr_width = wwin->screen_ptr->scr_width;
    int scr_height = wwin->screen_ptr->scr_height;

    if (!array || !WMGetArrayItemCount(array)) {
        wWindowMove(which_dpy, wwin, wwin->frame_x + dx, wwin->frame_y + dy)
;
    } else {
        WMArrayIterator iter;

        //  DST: Add code to support individual win movement within selection re
ct
        int move_all    = 0,
            add_x       = 0,
            add_y       = 0,
            w, h;

        x = wwin->frame_x + dx;
        y = wwin->frame_y + dy;
        w   = (int)(wwin->frame->core->width);
        h   = (int)(wwin->frame->core->height);
        if (move_rect && !retain_relative_positions) {
            if (x < move_rect->pos.x ||
                (x+w) > (move_rect->pos.x+move_rect->size.width)) {
                move_all    = 1, add_x = 1;
            }
            if (y < move_rect->pos.y ||
                (y+h) > (move_rect->pos.y+move_rect->size.height)) {
                move_all    = 1, add_y = 1;
            }
        }
        else
            move_all    = 1;
        if (!move_all)
            wWindowMove(which_dpy, wwin, x, y);
        else {
            if (add_x && move_rect)
                move_rect->pos.x    += dx;
            if (add_y && move_rect)
```

moveres.c 2/2

```
                move_rect->pos.y    += dy;
            WM_ITERATE_ARRAY(array, tmpw, iter) {
                x = tmpw->frame_x + dx;
                y = tmpw->frame_y + dy;

                /* don't let windows become unreachable */

                if (x + (int)tmpw->frame->core->width < 20)
                    x = 20 - (int)tmpw->frame->core->width;
                else if (x + 20 > scr_width)
                    x = scr_width - 20;

                if (y + (int)tmpw->frame->core->height < 20)
                    y = 20 - (int)tmpw->frame->core->height;
                else if (y + 20 > scr_height)
                    y = scr_height - 20;

                wWindowMove(which_dpy, tmpw, x, y);

                if (move_rect) {
                    w   = (move_rect->pos.x+move_rect->size.width) - (tmpw->fram
e_x + tmpw->frame->core->width);
                    h   = (move_rect->pos.y+move_rect->size.height) - (tmpw->fra
me_y + tmpw->frame->core->height);
                    if (move_rect->pos.x > x)
                        move_rect->pos.x = x;
                    else if (w<0)
                        move_rect->pos.x    -= w;
                    if (move_rect->pos.y > y)
                        move_rect->pos.y = y;
                    else if (h<0)
                        move_rect->pos.y    -= h;
                }
            }
        }
    }
}

//  DST: Modified to take display-var as argument, in addition to handling group
s.
void
updateWindowPosition(Display *which_dpy, WWindow *wwin, MoveData *data, Bool doR
esistance,
                     Bool opaqueMove, int newMouseX, int newMouseY)
{
    //  NOTE: The function prologue and epilogue have been cut. See CD-ROM, etc.
    if (opaqueMove) {
        //  DST: Win group support
        if (wwin->gid != -1)
            doWindowMove(which_dpy, wwin, windows_for_group(wwin->gid), newX - w
win->frame_x, newY - wwin->frame_y, area_for_group(wwin->gid), 0);
        else
            doWindowMove(which_dpy, wwin, scr->selected_windows, newX - wwin->fr
ame_x, newY - wwin->frame_y, &selected_rect, 0);
    } else {
        /* erase frames */
        if (wwin->gid != -1)
            drawFrames(wwin, windows_for_group(wwin->gid), data->realX - wwin->f
rame_x, data->realY - wwin->frame_y);
        else
            drawFrames(wwin, scr->selected_windows, data->realX - wwin->frame_x,
 data->realY - wwin->frame_y);
    }
}
```

multi.c 1/16

```
/*  multi.h
    (c) 2004-2005 Daniel Stodle, daniels@stud.cs.uit.no

    This file contains the bulk of the multi-input implementation, with some
    additional parts residing in event.c (for handling keyboard input in the
    main thread).
*/

#include    "multi.h"
#include    "group.h"
#include    "X11/extensions/shape.h"
#include    <X11/extensions/XTest.h>
#include    <pthread.h>
#include    "multi_msg.h"

#define XK_MISCELLANY
#include    <X11/keysymdef.h>

static int  can_use_multi = 1;

/*  The following constants describe the bits necessary for drawing a pointer,
    and the bits for a 16x16 blank cursor.
*/
const unsigned char cursor_bits[] = {
   0x00, 0x00, 0x06, 0x00, 0x1e, 0x00, 0x7c, 0x00, 0xfc, 0x01, 0xf8, 0x07,
   0xf8, 0x1f, 0xf0, 0x07, 0xf0, 0x03, 0xe0, 0x07, 0xe0, 0x0e, 0x40, 0x1c,
   0x40, 0x38, 0x00, 0x70, 0x00, 0x20, 0x00, 0x00};

const unsigned char cursor_mask_bits[] = {
   0x07, 0x00, 0x1f, 0x00, 0x7f, 0x00, 0xfe, 0x01, 0xfe, 0x07, 0xfc, 0x1f,
   0xfc, 0x3f, 0xf8, 0x1f, 0xf8, 0x07, 0xf0, 0x0f, 0xf0, 0x1f, 0xe0, 0x3e,
   0xe0, 0x7c, 0x40, 0xf8, 0x00, 0x70, 0x00, 0x20};

const unsigned char cursor_blank_bits[] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

unsigned char *cursor_bits_x2, *cursor_mask_bits_x2;;

//  This struct contains descriptors for all multi-input clients. Allocated
//  at runtime.
multi_input_t   *multi;

extern char *DisplayName;   // in main.c
extern WPreferences wPreferences;

Display         *multi_display; //  We need our own connection to the X server
Window          root_win;       //  The root window (we assume that it doesn't c
hange!)
Cursor          blank_cursor;   //  An invisible cursor
GC              frame_gc;       //  GC used for drawing selection rectangles in
the root window
uint32_t        black_pixel;    //  Cached value for the X server's black pixel
value.

/*  The mutex below protects against windows being used by the multi-input threa
d
    after being freed by the window manager's main thread. It is taken in
    wUnmanageWindow.
*/
pthread_mutex_t win_lock    = PTHREAD_MUTEX_INITIALIZER;


/*  init_multi_input:
```

multi.c 2/16

```
    Called from init_groups(). Takes care of initializing the multi-input stuff,
    and startin the multi-input thread.
*/
void        init_multi_input(void) {
    int         i, xtest_evt_base, xtest_err_base, major, minor;
    XGCValues   gcv;

    printf("Multi-input starting up with display: %s\n", (DisplayName ? DisplayName : "localhost:
0"));
    multi_display   = XOpenDisplay(DisplayName);
    if (!multi_display) {
        printf("Warning: Failed to open display for multi-cursor. Will use the other one.. DANGER DANGER
!\n");
        multi_display   = dpy;
    }
    root_win        = RootWindow(multi_display, 0);
    create_blank_cursor();
    XDefineCursor(multi_display, RootWindow(multi_display, 0), blank_cursor);
    scale_cursors_2x();

    multi   = 0;
    if (!XTestQueryExtension(multi_display, &xtest_evt_base, &xtest_err_base, &m
ajor, &minor)) {
        printf("Multi-input: XTestExtension not present, multi-input will be disabled.\n");
        return;
    }
    XTestGrabControl(multi_display, True);
    multi   = (multi_input_t*)calloc(kNum_groups, sizeof(multi_input_t));
    for (i=0;i<kNum_groups;i++) {
        multi[i].color  = groups[i].color;
        multi[i].cursor = create_cursor_win(multi[i].color);
    }

    black_pixel             = BlackPixel(multi_display, 0);
    gcv.plane_mask          = AllPlanes;
    gcv.foreground          = 0;
    gcv.line_width          = 2;
    gcv.subwindow_mode      = IncludeInferiors;
    gcv.graphics_exposures  = False;
    gcv.function            = GXxor;
    frame_gc                = XCreateGC(multi_display, root_win, GCForeground|GC
GraphicsExposures |GCFunction|GCLineWidth | GCPlaneMask, &gcv); //GCSubwindowMod
e|
    if (!frame_gc)
        printf("No frame gc!\n");
    else
        printf("Frame GC allocated.\n");
    if (can_use_multi) {
        pthread_t   pid;
        pthread_create(&pid, 0, multi_input_thread, 0);
    }
}


/*  multi_input_thread: The entry point for the multi-input thread.
    (Big surprise!). Opens a listening socket, and runs the input loop if
    successful.
*/
void*       multi_input_thread(void *args) {
    int     sock, yes = 1, port, d = 0, s = 0;
    char    *str;
    struct sockaddr_in addr;

    printf("Starting multi-input server..\n");
    //  Get the display number, if available, and store it in 'd'
```

multi.c 3/16

```
    str = (DisplayName ? strchr(DisplayName, ':') : 0);
    if (str && sscanf(str, ":%i.%i", &d, &s)<1)
        d   = 0;

    //  Open listening socket
    sock                = socket(AF_INET, SOCK_STREAM, 0);
    if (!sock) {
        perror("Failed creating socket!\n");
        return 0;
    }
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
        close(sock);
        perror("Error creating input socket.\n");
        return 0;
    }
    memset(&addr, 0, sizeof(struct sockaddr_in));
    addr.sin_family         = AF_INET;
    addr.sin_port           = htons(5000+d);
    addr.sin_addr.s_addr    = INADDR_ANY;
    if (bind(sock, (struct sockaddr*)&addr, sizeof(struct sockaddr)) == -1) {
        perror("Failed to bind server socket\n");
        close(sock);
        return 0;
    }
    run_input_loop(sock);
    return 0;
}


/*  run_input_loop: This is the mainloop of the multi-input thread, which takes
    care of handling incoming input requests and messages, as well as handle
    any X events.
*/
void        run_input_loop(int server_sock) {
    int             new_sock, max_sock, i;
    fd_set          fds;
    struct timeval  timeout;
    XEvent          evt;

    printf("Multi-input loop awaiting connections!\n");
    listen(server_sock, 10);
    while (1) {
        //  Set bits in the fd_set
        FD_ZERO(&fds);
        FD_SET(server_sock, &fds);
        max_sock    = server_sock;
        for (i=0;i<kNum_groups;i++) {
            if (multi[i].sock) {
                FD_SET(multi[i].sock, &fds);
                if (multi[i].sock > max_sock)
                    max_sock    = multi[i].sock;
            }
        }
        max_sock++;
        //  Check for traffic
        timeout.tv_sec  = 0;
        timeout.tv_usec = 100;
        if (select(max_sock, &fds, 0, 0, &timeout) > 0) {
            if (FD_ISSET(server_sock, &fds))
                accept_new_multi_client(server_sock);
            for (i=0;i<kNum_groups;i++) {
                if (multi[i].sock && FD_ISSET(multi[i].sock, &fds))
                    handle_multi_input(i);
            }
        }
```

multi.c 4/16

```
        //  Handle pending X events (these will mostly be related to our cursor
windows)
        while (XPending(multi_display)) {
            XNextEvent(multi_display, &evt);
            switch (evt.type) {
                case Expose:
                    //  We only need the first expose event.
                    if (evt.xexpose.count == 0) {
                        XClearWindow(multi_display, evt.xexpose.window);
                        XSetWindowBorder(multi_display, evt.xexpose.window, blac
k_pixel);
                    }
                    break;
                case VisibilityNotify:
                    //  The problem with this event is that multiple overlapping
                    //  cursor windows end up fighting each other for who gets t
o
                    //  be on top. This should probably be fixed in some clever
way
                    //  at a later time.
                    XRaiseWindow(multi_display, evt.xvisibility.window);
                    break;
                default:
                    //  Keep track of events we receive but don't handle, so we
                    //  can add support for them (if necessary).
                    printf("Event of type %d\n", evt.type);
                    break;
            }
        }
    }
}


/*  accept_new_multi_client: Accepts a connection on the given server socket,
    assigns the connection a cursor ID and sends a message to the remote end
    with size of display and assigned cursor ID.
*/
void        accept_new_multi_client(int server_sock) {
    int     cid = 0, sock, len = sizeof(struct sockaddr), flag;
    struct sockaddr     addr;

    sock    = accept(server_sock, &addr, &len);
    flag    = 1;
    //  Input over TCP really performs a lot better with the TCP_NODELAY flag se
t.
    if (setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int))
== -1)
        printf("Warning: Failed setting TCP_NODELAY on socket.\n");

    //  Find an available cursor ID. Available cursors will have their socket se
t to 0.
    for (cid=0;cid<kNum_groups;cid++) {
        if (!multi[cid].sock)
            break;
    }
    if (cid>=kNum_groups) {
        //  The protocol should be extended to give the remote clients info abou
t
        //  why their connection is just rudely closed.
        printf("Warning: Too many input clients! Discarding incoming connection..\n");
        close(sock);
        return;
    }
    //  Assign the cursor, and map the cursor window.
    multi[cid].sock = sock;
```

multi.c 5/16

```
    map_cursor_if_needed(cid);
    //  Inform the remote end about the size of the display, and the cursor ID
    //  it has been allocated.
    if (multi_send_msg(multi[cid].sock, kMulti_info_msg, 3, cid, WidthOfScreen(D
efaultScreenOfDisplay(multi_display)), HeightOfScreen(DefaultScreenOfDisplay(mul
ti_display))) != 0)
        terminate_multi_client(cid);
}


/*  terminate_multi_client: Cleans up a multi-cursor client by closing its socke
t,
    and releasing any data associated with the cursor. Also hides the cursor win
dow
    associated with the client.
*/
void        terminate_multi_client(int cursor_id) {
    printf("Removing multi-input client %d\n", cursor_id);
    if (multi[cursor_id].sock)
        close(multi[cursor_id].sock);

    unmap_cursor_if_mapped(cursor_id);
    multi[cursor_id].sock   = 0;
    if (multi[cursor_id].wwin) {
        pthread_mutex_lock(&win_lock);
        multi_move_window(cursor_id, 1);
        pthread_mutex_unlock(&win_lock);
    }
    multi[cursor_id].cursor_state   = 0;
    multi[cursor_id].btn_state      = 0;
    multi[cursor_id].focus          = 0;
    multi[cursor_id].old_focus      = 0;
}


/*  handle_multi_input: Receives multi-input messages from the given client,
    and handles them.
*/
void        handle_multi_input(int cursor_id) {
    fd_set          fds;        //  Used for polling
    struct timeval  timeout;
    multi_msg_t msg;        //  We receive one message at a time
    m_point_t   win_loc,    //  Dummy variable for XQueryPointer
                last_warp,  //  Used to cache the last location the cursor was w
arped to, but only on a per-client basis
                old_cursor_loc; //  Stores the old system-cursor position, in ca
se we should reset it.
    Window      root,           //  Used for XQueryPointer
                dummy;          //  Used for XQueryPointer
    int         fkey,           //  Holds which fkey (if any) is pressed. Fkeys
are used for window teleportation.
                mask,           //  Dummy mask for XQueryPointer
                can_move_result,//  Holds the result from check_move_window()
                can_post,       //  Controls whether we will post button events.
                received_events = 0,    //  Number of processed events.
                i;

    set_pt(last_warp,-1,-1);
    /*  To ensure fair event processing, we only process up to
        kMulti_evt_window_size*3 events from a cursor client at a time.
    */
    do {
        //  Receive one message
        if (multi_recv_msg(multi[cursor_id].sock, &msg) != 0) {
            terminate_multi_client(cursor_id);
```

multi.c 6/16

```
            return;
        }
        //  WARNING: If anything happens below that causes a window to be destro
yed,
        //  we may end up deadlocking!
        pthread_mutex_lock(&win_lock);
        switch (msg.type) {
            case kMulti_info_msg:
                printf("WM should not receive an info message!\n");
                break;
            case kMulti_button_msg:
                //  The user is trying to perform a mouse-up or mouse-down. Firs
t,
                //  show the cursor if necessary
                map_cursor_if_needed(cursor_id);

                //  Figure out where the system cursor is right now
                XQueryPointer(multi_display, root_win, &dummy, &dummy, &old_curs
or_loc.x, &old_cursor_loc.y, &i, &i, &i);

                //  Warp the system cursor to the location of this virtual curso
r,
                //  and store the old focus window
                warp_if_needed(&last_warp, root_win, multi[cursor_id].x, multi[c
ursor_id].y);
                multi[cursor_id].old_focus  = multi[cursor_id].focus;

                //  Figure out which window the cursor now hovers above
                XQueryPointer(multi_display, root_win, &root, &multi[cursor_id].
focus, &win_loc.x, &win_loc.y, &win_loc.x, &win_loc.y, &mask);

                //  Do we have a focus window?
                if (multi[cursor_id].focus) {
                    //  Yes, figure out where the cursor is within the window. A
lthough the focus location
                    //  is currently not used, it might be useful in the future
(and the previous event posting
                    //  mechanism, based on XSetInputFocus and XWarpPointer, use
d the focus location extensively).
                    XQueryPointer(multi_display, multi[cursor_id].focus, &root,
&root, &win_loc.x, &win_loc.y, &multi[cursor_id].fx, &multi[cursor_id].fy, &mask
);
                    //  Assign the window to this cursor.
                    assign_group(cursor_id, 0, wWindowFor(multi[cursor_id].focus
), 0);
                }
                /*  Scan cursors and check for anyone else in the dragging state
.
                    We do this here to avoid doing it twice below.
                */
                can_post    = 1;
                for (i=0;i<kNum_groups;i++) {
                    if (i != cursor_id && (multi[i].cursor_state & kCursor_dragg
ing) != 0) {
                        //  We can't interfere with someone else's drag,
                        //  so we drop this event.
                        can_post = 0;
                        break;
                    }
                }
                //  If we can't post the button event, we restore the system cur
sor's
                //  old position.
                if (!can_post)
                    warp_if_needed(&last_warp, root_win, old_cursor_loc.x, old_c
```

multi.c 7/16

```
ursor_loc.y);

                if (multi[cursor_id].cursor_state || msg.data[0] == 1) {
                    /*  The cursor is either selecting, moving, dragging or the
click
                        is with the left mouse button, so we need to check where
 the
                        click happens and whether we need to handle it specially
. First
                        we check if the current cursor position is in a location
 where
                        we can move the window.
                    */
                    can_move_result = check_move_window(cursor_id);

                    if ((multi[cursor_id].focus == 0 || multi[cursor_id].cursor_
state & kCursor_selecting) && !(multi[cursor_id].cursor_state & kCursor_dragging
)) {
                        /*  Focus-window == 0 means that the virtual cursor isn'
t located above any window,
                            so the click translates to a drag in the root window
. We also get here if we're
                            already in the selecting state.

                            Drag in root window, perform selection.
                        */

                        if (msg.data[1]) {
                            //  Mouse-button is down => begin window selection
                            multi[cursor_id].cursor_state   |= kCursor_selecting
;
                            multi[cursor_id].selection.x    = multi[cursor_id].x
;
                            multi[cursor_id].selection.y    = multi[cursor_id].y
;
                            multi[cursor_id].selection.w    = 2;
                            multi[cursor_id].selection.h    = 2;
                            update_selection_rect(cursor_id, 0);
                        }
                        else if (!msg.data[1]) {
                            //  Mouse-button is up => end window selection
                            multi[cursor_id].cursor_state   &= ~kCursor_selectin
g;
                            //  First update: Resize the visible selection rect
                            update_selection_rect(cursor_id, 0);
                            //  Second update: End the selection, and select any
                            //  windows inside the rect.
                            update_selection_rect(cursor_id, 1);
                        }
                    }
                    else if (msg.data[1] == 1 && can_move_result == kCursor_can_
move_win) {
                        //  Moust-button is down, and the cursor is in the
                        //  titlebar of a window => begin window move
                        multi[cursor_id].cursor_state   |= kCursor_moving;
                        multi_move_window(cursor_id, -1);
                    }
                    else if (multi[cursor_id].cursor_state & kCursor_moving) {
                        //  Mouse-button is up, and we are moving a window =>
                        //  end window move
                        multi_move_window(cursor_id, 1);
                        multi[cursor_id].cursor_state   &= ~kCursor_moving;
                    }
                    else if (can_move_result != kOther_cursor_moving_win && can_
post) {
```

multi.c 8/16

```
                        //  We can post "raw" mouse-button events, and no-one el
se is
                        //  currently having their mouse-button down. Also, we a
re
                        //  not attempting to click in a window's titlebar (whic
h would
                        //  interfere with someone else attempting to move the w
indow).
                        XTestFakeButtonEvent(multi_display, msg.data[0], msg.dat
a[1], 0);
                        if (msg.data[1]) {
                            multi[cursor_id].btn_state      |= 1 << msg.data[0];
                            multi[cursor_id].cursor_state   |= kCursor_dragging;
                        }
                        else {
                            multi[cursor_id].btn_state      &= ~(1 << msg.data[0
]);
                            multi[cursor_id].cursor_state   &= ~kCursor_dragging
;
                        }
                    }
                    else if (can_post) {
                        //  Post a raw button event.
                        XTestFakeButtonEvent(multi_display, msg.data[0], msg.data[1]
, 0);
                        if (msg.data[1]) {
                            multi[cursor_id].btn_state      |= 1 << msg.data[0];
                            multi[cursor_id].cursor_state   |= kCursor_dragging;
                        }
                        else {
                            multi[cursor_id].btn_state      &= ~(1 << msg.data[0]);
                            multi[cursor_id].cursor_state   &= ~kCursor_dragging;
                        }
                    }
                    if (msg.data[1]) {
                        //  reset the blank system cursor. Other apps tend to change
                        //  it after clicks, so we need to do this fairly often.
                        XDefineCursor(multi_display, root_win, blank_cursor);
                    }
                    else
                        update_cursor(cursor_id);
                    break;
                case kMulti_motion_msg:
                    //  Handle mouse movements.
                    multi[cursor_id].lx = multi[cursor_id].x;
                    multi[cursor_id].ly = multi[cursor_id].y;
                    multi[cursor_id].x  = msg.data[0];
                    multi[cursor_id].y  = msg.data[1];
                    //  Did the cursor actually move since last time?
                    if (multi[cursor_id].x != multi[cursor_id].lx || multi[cursor_id
].y != multi[cursor_id].ly) {
                        //  Yes, map it if needed, and move the window representing
the cursor.
                        map_cursor_if_needed(cursor_id);
                        XMoveWindow(multi_display, multi[cursor_id].cursor, multi[cu
rsor_id].x, multi[cursor_id].y);
                        //  Depending on the cursor state, either post a fake motion
 event, update the selection
                        //  rect, or move a (number of) window(s).
                        if (multi[cursor_id].btn_state && !(multi[cursor_id].cursor_
state & ~kCursor_dragging))
                            XTestFakeMotionEvent(multi_display, 0, multi[cursor_id].
x, multi[cursor_id].y, 0);
                        else if (multi[cursor_id].cursor_state & kCursor_selecting)
```

multi.c 9/16

```
{
                        update_selection_rect(cursor_id, 0);
                        multi[cursor_id].selection.w    = multi[cursor_id].x - m
ulti[cursor_id].selection.x;
                        multi[cursor_id].selection.h    = multi[cursor_id].y - m
ulti[cursor_id].selection.y;
                        update_selection_rect(cursor_id, 0);
                    }
                    else if (multi[cursor_id].cursor_state & kCursor_moving)
                        multi_move_window(cursor_id, 0);
                    //update_cursor(cursor_id);
                }
                break;
            case kMulti_keyboard_msg:
                //  Handle keyboard events. First, hide the cursor if necessary.
                unmap_cursor_if_mapped(cursor_id);
                //  Is this the FKEY corresponding to this cursor ID? If so,
                //  teleport the currently selected windows to the current posit
ion.
                fkey    = XKeycodeToKeysym(dpy, msg.data[0], 0) - XK_F1;
                if (fkey == 0)
                    teleport_group(fkey, multi[cursor_id].x, multi[cursor_id].y)
;
                else
                    multi_send_key_event_to_main_thead(multi[cursor_id].focus, m
sg.data[0], msg.data[1], 1);
                break;
        }
        pthread_mutex_unlock(&win_lock);
        received_events++;

        //  Poll for new socket traffic
        FD_ZERO(&fds);
        FD_SET(multi[cursor_id].sock, &fds);
        timeout.tv_sec  = 0;
        timeout.tv_usec = 0;
    } while (select(multi[cursor_id].sock+1, &fds, 0, 0, &timeout) > 0 && receiv
ed_events <= kMulti_evt_window_size*3);
    //  Send acknowledgement. One acknowledgement is sufficient.
    if (multi_send_msg(multi[cursor_id].sock, kMulti_event_ack_msg, 1, cursor_id
) != 0)
        terminate_multi_client(cursor_id);
}

#pragma mark -

/*  update_cursor: Raises the cursor window, sets its border to black and clears
    the cursor's window (in effect, filling it with the cursor's color).
*/
void        update_cursor(int cid) {
    XRaiseWindow(multi_display, multi[cid].cursor);
    XSetWindowBorder(multi_display, multi[cid].cursor, black_pixel);
    XClearWindow(multi_display, multi[cid].cursor);
}


//  map_cursor_if_needed: Checks if the cursor is mapped, and if not, maps it.
void        map_cursor_if_needed(int cid) {
    if (!multi[cid].mapped) {
        multi[cid].mapped   = 1;
        XMapRaised(multi_display, multi[cid].cursor);
    }
}
```

multi.c 10/16

```
//  unmap_cursor_if_mapped: Checks if the cursor is mapped, and if so, unmaps it
.
void        unmap_cursor_if_mapped(int cid) {
    if (multi[cid].mapped) {
        multi[cid].mapped   = 0;
        XLowerWindow(multi_display, multi[cid].cursor);
        XUnmapWindow(multi_display, multi[cid].cursor);
    }
}


/*  warp_if_needed: Compares <pt> with x and y, and warps the system cursor's
    position if they differ, updating <pt> with the new values.
*/
void        warp_if_needed(m_point_t *pt, Window w, int x, int y) {
    if (pt->x != x || pt->y != y) {
        pt->x   = x;
        pt->y   = y;
        XWarpPointer(multi_display, 0, w, 0, 0, 0, 0, x, y);
    }
}


/*  focus_if_needed: Sets input forcus to the given window, if it is different
    from *w, and raises it. Note that this function is not currently used,
    as focus handling has been moved to the main thread in event.c.
*/
void        focus_if_needed(Window *w, Window focus) {
    if (*w != focus) {
        *w  = focus;
        XSetInputFocus(multi_display, focus, RevertToParent, CurrentTime);
        XRaiseWindow(multi_display, focus);
    }
}

#pragma mark -

/*  update_selection_rect: Draws the selection rect for the given cursor, with
    that cursor's color. If select is true, it erases the drawn rect and selects
    any windows inside the selection rect, assigning them to the cursor's group.
*/
void        update_selection_rect(int cid, int select) {
    int         x, y, w, h, x1, x2, y1, y2, group_freed = 0;
    WMRect      sel_rect;
    WScreen     *scr;

    x   = multi[cid].selection.x;
    y   = multi[cid].selection.y;
    w   = multi[cid].selection.w;
    h   = multi[cid].selection.h;
    if (w < 0) {
        x   += w;
        w   = -w;
    }
    if (h < 0) {
        y   += h;
        h   = -h;
    }
    XGrabServer(multi_display);
    if (select) {
        //  Warning: The thread safety of the code below has not been completely
        //  verified. It _should_ work..
        WWindow *tmpw;
        sel_rect.pos.x          = x;
        sel_rect.pos.y          = y;
```

multi.c 11/16

```
        sel_rect.size.width     = w;
        sel_rect.size.height    = h;
        x1  = x;
        x2  = x+w;
        y1  = y;
        y2  = y+h;
        /* select the windows and put them in the selected window list */
        scr = wScreenForWindow(root_win);
        tmpw = scr->focused_window;
        free_group(cid);
        while (tmpw != NULL) {
            if (!(tmpw->flags.miniaturized || tmpw->flags.hidden)) {
                if ((tmpw->frame->workspace == scr->current_workspace ||
                     IS_OMNIPRESENT(tmpw)) && (tmpw->frame_x >= x1) &&
                     (tmpw->frame_y >= y1) &&
                     (tmpw->frame->core->width + tmpw->frame_x <= x2) &&
                     (tmpw->frame->core->height + tmpw->frame_y <= y2)) {
                    assign_group(cid, &sel_rect, tmpw, 0);
                }
            }
            tmpw = tmpw->prev;
        }
    }
    else {
        XSetForeground(multi_display, frame_gc, groups[cid].color.pixel);
        XDrawRectangle(multi_display, root_win, frame_gc, x, y, w, h);
    }
    XUngrabServer(multi_display);
}


/*  multi_move_window: Moves windows associated with the given cursor. If done
    is 1, finishes moving and frees move-data. The thread-safety of this code
    also remains under scrutiny, as it interacts quite a bit with the window
    manager's internals..
*/
void        multi_move_window(int cid, int done) {
    WScreen                 *scr;
    XSetWindowAttributes    set_attr;
    XWindowAttributes       get_attr;

    if (done == 1) {
        if (multi[cid].reset_save_under) {
            set_attr.save_under = False;
            XChangeWindowAttributes(multi_display, multi[cid].wwin->frame->core-
>window, CWSaveUnder, &set_attr);
        }
        freeMoveData(&multi[cid].move_data);
        multi[cid].wwin = 0;
        return;
    }
    else if (!multi[cid].wwin) {
        multi[cid].wwin = wWindowFor(multi[cid].focus);
        if (!multi[cid].wwin) {
            printf("Couldn't find wwindow for move.\n");
            return;
        }
        initMoveData(multi[cid].wwin, &multi[cid].move_data);
        multi[cid].move_data.mouseX = multi[cid].x;
        multi[cid].move_data.mouseY = multi[cid].y;
        multi[cid].opaque_move      = wPreferences.opaque_move;
        multi[cid].reset_save_under = 0;
        if (done == -1)
            wRaiseFrame(multi_display, multi[cid].wwin->frame->core);
        if (multi[cid].opaque_move) {
```

multi.c 12/16

```
            XGetWindowAttributes(multi_display, multi[cid].wwin->frame->core->wi
ndow, &get_attr);
            if (!get_attr.save_under) {
                set_attr.save_under = True;
                XChangeWindowAttributes(multi_display, multi[cid].wwin->frame->c
ore->window, CWSaveUnder, &set_attr);
                multi[cid].reset_save_under = 1;
            }
        }
    }
    scr = multi[cid].wwin->screen_ptr;
    if (multi[cid].x != multi[cid].lx || multi[cid].y != multi[cid].ly)
        updateWindowPosition(multi_display, multi[cid].wwin, &multi[cid].move_da
ta, scr->selected_windows == NULL && wPreferences.edge_resistance > 0, multi[cid
].opaque_move, multi[cid].x, multi[cid].y);
}


/*  check_move_window:
    Returns 0 if the given cursor doesn't intersect with any (known)
    window's titlebar, 1 if it is okay to move the window, and 2 if the
    window is already being moved by a different multi-cursor.

    Assumes that win_lock is held.
*/
int     check_move_window(int cid) {
    WWindow     *win;
    int         x, y, w, h, i;

    //  If there is no focus window, there is nothing to move.
    if (!multi[cid].focus)
        return 0;

    //  Check if the given window is being moved by someone else.
    win = wWindowFor(multi[cid].focus);
    if (win) {
        x   = win->frame_x;
        y   = win->frame_y;
        w   = win->frame->titlebar->width;
        h   = win->frame->titlebar->height;
        if (multi[cid].x > x && multi[cid].x < x+w &&
            multi[cid].y > y && multi[cid].y < y+h) {
            for (i=0;i<kNum_groups;i++) {
                if (i==cid)
                    continue;
                if (multi[i].focus == multi[cid].focus && (multi[i].cursor_state
 & kCursor_moving))
                    return kOther_cursor_moving_win;
            }
            return kCursor_can_move_win;
        }
    }
    return 0;
}


//  query_multi_cursor: Returns the position of the given multi-cursor.
void        query_multi_cursor(int cid, int *x, int *y) {
    if (cid >= 0 && cid < kNum_groups) {
        *x  = multi[cid].x;
        *y  = multi[cid].y;
    }
}
```

multi.c 13/16

```
/*  multi_send_key_event_to_main_thead: Sends an xcrossing event, targeted to th
e
    conduit window, to the window manager's main thread, facilitating keyboard
    event posting.
*/
void    multi_send_key_event_to_main_thead(Window focus, uint32_t keycode, uint3
2_t state, uint32_t should_focus) {
    XEvent event;

    memset(&event, sizeof(XEvent), 0);
    event.xcrossing.type                = EnterNotify;
    event.xcrossing.window              = conduit_win;
    event.xcrossing.subwindow           = focus;
    event.xcrossing.x                   = keycode >> 16;
    event.xcrossing.y                   = keycode & 0xFFFF;
    event.xcrossing.same_screen         = state;
    event.xcrossing.focus               = should_focus;
    XSendEvent(multi_display, conduit_win, False, NoEventMask, &event);
    XSync(multi_display, False);
}


/*  multi_remove_window: Called by the window manager to indicate that the
    given window is going away. Called while win_lock is held by the window
    manager, in wWindowDestroy.
*/
void    multi_remove_window(Window win) {
    int     i;

    for (i=0;i<kNum_groups;i++) {
        if (multi[i].old_focus == win)
            multi[i].old_focus  = 0;
        if (multi[i].focus == win)
            multi[i].focus      = 0;
        if (multi[i].wwin) {
            /*  We only get here if this cursor is in the middle of moving a win
dow.
                We have to free the move data as well and reset the cursor state
.
            */
            if (multi[i].wwin->client_win == win) {
                multi[i].wwin           = 0;
                multi[i].cursor_state   &= ~kCursor_moving;
                freeMoveData(&multi[i].move_data);
            }
        }
    }
}

#pragma mark -

/*  create_cursor_win:
    This code is a slightly modified version of the code written by
    Grant Wallace for creating a cursor shaped window.
*/
Window      create_cursor_win(XColor cursor_color) {
    Window                  win;
    XSetWindowAttributes    attr;
    int                     mask;
    int                     shape_event_base,
                            shape_error_base;
    Pixmap                  pixmap, pmask;
    XWMHints                *hints;

    if (!XShapeQueryExtension (multi_display, &shape_event_base, &shape_error_ba
```

multi.c 14/16

```
se)) {
        printf("XShapeExtension not found - multi-input will be disabled.\n");
        can_use_multi   = 0;
        return 0;
    }

    mask                    = CWOverrideRedirect | CWBackPixel | CWBorderPixel |
 CWSaveUnder;
    attr.override_redirect  = 1;
    attr.background_pixel   = cursor_color.pixel;
    attr.border_pixel       = BlackPixel(multi_display, 0);
    attr.save_under         = True;
    win                     = XCreateWindow(multi_display, root_win, 0, 0, kCurs
or_size, kCursor_size, 2, CopyFromParent, InputOutput, CopyFromParent, mask, &at
tr);

    if (!win) {
        printf("Error! Failed to create multi-cursor window..\n");
        return 0;
    }

    // TODO: Create the pixmap once on startup and reuse
    pixmap                  = XCreateBitmapFromData(multi_display, win,  (const
char *) cursor_bits_x2, kCursor_size, kCursor_size);
    pmask                   = XCreateBitmapFromData(multi_display, win, (const c
har *) cursor_mask_bits_x2, kCursor_size, kCursor_size);

    XShapeCombineMask(multi_display, win, ShapeClip, 0, 0, pixmap, ShapeSet);
    XShapeCombineMask(multi_display, win, ShapeBounding, 0, 0, pmask, ShapeSet);
    XSelectInput(multi_display, win, VisibilityChangeMask | ExposureMask);

    hints                   = XAllocWMHints();
    hints->flags            |= InputHint;
    XSetWMHints(multi_display, win, hints);
    XFree(hints);

    return win;
}


//  create_blank_cursor: Creates an invisible cursor.
void        create_blank_cursor(void) {
    Pixmap      pixmap, pmask;
    XColor      black, white;

    pixmap                  = XCreateBitmapFromData(multi_display, root_win,  (c
onst char *) cursor_blank_bits, 16, 16);
    pmask                   = XCreateBitmapFromData(multi_display, root_win, (co
nst char *) cursor_blank_bits, 16, 16);
    bzero(&black, sizeof(XColor));
    bzero(&white, sizeof(XColor));
    black.pixel     = BlackPixel(multi_display, DefaultScreen(multi_display));
    white.pixel     = WhitePixel(multi_display, DefaultScreen(multi_display));
    XQueryColor(multi_display, DefaultColormap(multi_display, DefaultScreen(mult
i_display)), &black);
    XQueryColor(multi_display, DefaultColormap(multi_display, DefaultScreen(mult
i_display)), &white);
    blank_cursor            = XCreatePixmapCursor(multi_display, pixmap, pixmap,
 &black, &white, 16, 16);
}


/*  scale_cursors_2x: Scales the bitmaps (cursor_bits, cursor_mask_bits) from
    a 16x16 to a 32x32 cursor.
*/
```

multi.c 15/16

```
void        scale_cursors_2x(void) {
    int     x, y, src_major, src_minor, dst_major, dst_minor;
    int     xx, yy;
    char    *mask_copy;

    cursor_bits_x2      = (unsigned char*)calloc(1, 32*32/8);
    cursor_mask_bits_x2 = (unsigned char*)calloc(1, 32*32/8);
    for (y=0;y<16;y++) {
        for (x=0;x<16;x++) {
            src_major   = x/8 + (y*2);
            src_minor   = x%8;
            if (cursor_bits[src_major] & (1 << src_minor)) {
                xx  = x*2;
                yy  = y*2;
                cursor_bits_x2[xx/8+yy*4]   |= 1 << (xx%8);
                cursor_bits_x2[xx/8+yy*4]   |= 1 << ((xx+1)%8);
                cursor_bits_x2[xx/8+(yy+1)*4]   |= 1 << (xx%8);
                cursor_bits_x2[xx/8+(yy+1)*4]   |= 1 << ((xx+1)%8);
            }
            if (cursor_mask_bits[src_major] & (1 << src_minor)) {
                xx  = x*2;
                yy  = y*2;
                cursor_mask_bits_x2[xx/8+yy*4]  |= 1 << (xx%8);
                cursor_mask_bits_x2[xx/8+yy*4]  |= 1 << ((xx+1)%8);
                cursor_mask_bits_x2[xx/8+(yy+1)*4]  |= 1 << (xx%8);
                cursor_mask_bits_x2[xx/8+(yy+1)*4]  |= 1 << ((xx+1)%8);
            }
        }
    }
    //  Smooth scaled copies
    smooth_bit_buffer(cursor_bits_x2, 32, 32, 1);
    smooth_bit_buffer(cursor_mask_bits_x2, 32, 32, 1);
}


/*  smooth_bit_buffer: Executes a simple smoothing algorithm to make the
    scaled cursors appear more beautiful. If use_copy is 0, the buffer
    will be smoothed in-place, giving a different result.
*/
void        smooth_bit_buffer(char *data, int sx, int sy, int use_copy) {
    int     count, x, y, a, b, c, aa, bb, cc, i;
    char    *copy;

    if (use_copy) {
        copy    = (unsigned char*)calloc(1, sx*sy/8);
        memcpy(copy, data, sx*sy/8);
    }
    else
        copy    = data;
    for (y=1;y<sy-1;y++) {
        for (x=1;x<sx-1;x++) {
            aa          = (x-1)/8 + (y*4);
            bb          = x/8 + (y*4);
            cc          = (x+1)/8 + (y*4);
            a           = (x-1)%8;
            b           = x%8;
            c           = (x+1)%8;
            count       = 0;
            for (i=-1;i<2;i++) {
                if (copy[aa+(i*4)] & (1 << a))
                    count++;
                if (copy[bb+(i*4)] & (1 << b))
                    count++;
                if (copy[cc+(i*4)] & (1 << c))
                    count++;
```

multi.c 16/16

```
            }
            if (count >= 5)
                data[bb]    |= (1 << b);
        }
    }
    if (use_copy)
        free(copy);
}
```

multi.h 1/2

```
/*  multi.h
    (c) 2004-2005 Daniel Stodle, daniels@stud.cs.uit.no
    Header file supporting multi-input.
*/

#ifndef MULTI_H
    #define MULTI_h

//  Includes
    #include "window.h"
    #include "wcore.h"
    #include "framewin.h"
    #include "WindowMaker.h"
    #include "screen.h"
    #include "moveres.h"
    #include <stdint.h>

//  Constants
enum {
    kCursor_size    = 32,

    kCursor_default     = 0,
    kCursor_selecting   = 1 << 0,
    kCursor_moving      = 1 << 1,
    kCursor_dragging    = 1 << 2,

    kCursor_can_move_win        = 1,
    kOther_cursor_moving_win    = 2,
};

#define set_pt(p, a, b) (p.x=a,p.y=b)

//  Typedefs
typedef struct {
    int         x, y;
} m_point_t;

typedef struct {
    Window      cursor,     // The window representing a cursor
                old_focus,  // Old focus window
                focus;      // Current focus window
    int         sock,       // Input socket
                x, y,       // Current x and y coordinates
                lx, ly,     // Previous x and y coordinates
                fx, fy,     // Coordinates for focus location - not used now
                btn_state,  // Which buttons are pressed? 1 bit pr button.
                mapped,     // Indicates if the cursor window mapped
                cursor_state;   // Current cursor state.
    struct {
        int     x, y, w, h; // This cursor's selection rectangle.
    } selection;
    XColor      color;      // Cursor color.
    WWindow     *wwin;      // Window associated with move operation.
    MoveData    move_data;  // Move-data, used by the window manager during move
s.
    int         opaque_move,
                reset_save_under;
} multi_input_t;

//  Export some globals!
extern multi_input_t    *multi;
extern pthread_mutex_t  win_lock;

//  Prototypes
    void        init_multi_input(void);
```

multi.h 2/2

```
    void*       multi_input_thread(void *args);
    void        run_input_loop(int server_sock);
    void        update_cursor(int cid);
    void        update_selection_rect(int cid, int select);

    void        map_cursor_if_needed(int cid);
    void        unmap_cursor_if_mapped(int cid);

    void        warp_if_needed(m_point_t *pt, Window w, int x, int y);
    void        focus_if_needed(Window *w, Window focus);

    void        accept_new_multi_client(int server_sock);
    void        terminate_multi_client(int cursor_id);
    void        handle_multi_input(int cursor_id);
    int         check_move_window(int cid);
    void        multi_move_window(int cid, int done);
    void        query_multi_cursor(int cid, int *x, int *y);
    void        multi_send_key_event_to_main_thead(Window focus, uint32_t keycod
e, uint32_t state, uint32_t should_focus);

    void        multi_remove_window(Window win);

    Window      create_cursor_win(XColor cursor_color);
    void        create_blank_cursor(void);
    void        scale_cursors_2x(void);
    void        smooth_bit_buffer(char *data, int sx, int sy, int use_copy);
#endif
```

placement.c 1/2

```
/* placement.c - window and icon placement on screen
    NOTE: This file has been cut down to show the relevant modifications to
    the Window Maker source code. Please see the CD-ROM for the complete source
    listing.
 */

//  DST: Include necessary header
#include "multi.h"


//  DST: Add my own placement routine
extern Display *dpy;
Bool    DanielPlaceWindow(WWindow *wwin, int *x_ret, int *y_ret, unsigned width,
 unsigned height) {
    Window      root, child;
    int         root_x, root_y, win_x, win_y, mask;
    WScreen     *scr = wwin->screen_ptr;

    //  If the window is associated with a group, place it at the group's cursor
    //  location. Otherwise, use the current system cursor location.
    if (wwin->gid != -1)
        query_multi_cursor(wwin->gid, x_ret, y_ret);
    else {
        XQueryPointer(dpy, DefaultRootWindow(dpy), &root, &child, &root_x, &root
_y, &win_x, &win_y, &mask);
        *x_ret  = root_x;
        *y_ret  = root_y;
    }
    //  Keep window on screen
    if (*x_ret + width > scr->scr_width)
        *x_ret = scr->scr_width - width;
    if (*x_ret < 0)
        *x_ret = 0;

    if (*y_ret + height > scr->scr_height)
        *y_ret = scr->scr_height - height;
    if (*y_ret < 0)
        *y_ret = 0;

    return True;
}



void
PlaceWindow(WWindow *wwin, int *x_ret, int *y_ret,
            unsigned width, unsigned height)
{
    WScreen *scr = wwin->screen_ptr;
    int h = WMFontHeight(scr->title_font) + (wPreferences.window_title_clearance
 + TITLEBAR_EXTEND_SPACE) * 2;

    switch (wPreferences.window_placement) {
     case WPM_MANUAL:
    InteractivePlaceWindow(wwin, x_ret, y_ret, width, height);
    break;

     case WPM_SMART:
    smartPlaceWindow(wwin, x_ret, y_ret, width, height);
    break;

     case WPM_AUTO:
        // DST: Call my own placement routine when "auto" is selected
        if (DanielPlaceWindow(wwin, x_ret, y_ret, width, height))
            break;
```

placement.c 2/2

```
    if (autoPlaceWindow(wwin, x_ret, y_ret, width, height, 0)) {
        break;
    } else if (autoPlaceWindow(wwin, x_ret, y_ret, width, height, 1)) {
        break;
    }
    /* there isn't a break here, because if we fail, it should fall
       through to cascade placement, as people who want tiling want
       automagicness aren't going to want to place their window */

     case WPM_CASCADE:
        if (wPreferences.window_placement == WPM_AUTO)
            scr->cascade_index++;

    cascadeWindow(scr, wwin, x_ret, y_ret, width, height, h);

        if (wPreferences.window_placement == WPM_CASCADE)
            scr->cascade_index++;
    break;

     case WPM_RANDOM:
    {
        int w, h, extra_height;
        WArea usableArea = scr->totalUsableArea;

            if (wwin->frame)
                extra_height = wwin->frame->top_width + wwin->frame->bottom_widt
h + 2;
            else
                extra_height = 24; /* random value */

        w = ((usableArea.x2-X_ORIGIN(scr)) - width);
        h = ((usableArea.y2-Y_ORIGIN(scr)) - height - extra_height);
        if (w<1) w = 1;
        if (h<1) h = 1;
        *x_ret = X_ORIGIN(scr) + rand()%w;
        *y_ret = Y_ORIGIN(scr) + rand()%h;
    }
    break;

#ifdef DEBUG
     default:
    puts("Invalid window placement!!!");
    *x_ret = 0;
    *y_ret = 0;
#endif
    }

    if (*x_ret + width > scr->scr_width)
    *x_ret = scr->scr_width - width;
    if (*x_ret < 0)
        *x_ret = 0;

    if (*y_ret + height > scr->scr_height)
    *y_ret = scr->scr_height - height;
    if (*y_ret < 0)
        *y_ret = 0;
}
```

window.c 1/3

```
/* window.c - client window managing stuffs
    NOTE: This file has been cut down to show the relevant modifications to
    the Window Maker source code. Please see the CD-ROM for the complete source
    listing.
 */

//  DST: Group/mc stuff
#include "group.h"
#include "multi.h"

WWindow*
wWindowCreate()
{
    WWindow *wwin;

    wwin = wmalloc(sizeof(WWindow));
    wretain(wwin);

    memset(wwin, 0, sizeof(WWindow));

    wwin->client_descriptor.handle_mousedown = frameMouseDown;
    wwin->client_descriptor.parent = wwin;
    wwin->client_descriptor.self = wwin;
    wwin->client_descriptor.parent_type = WCLASS_WINDOW;

    //  DST: Add group support
    wwin->gid   = -1;
    return wwin;
}


void
wWindowDestroy(WWindow *wwin)
{
    int i;

    //  DST: Add cleanup code, so we don't end up with dangling pointers
    //  WARNING: We assume that win_lock has been taken at this point! This
    //  shouldn't be a problem, since this method is only called from
    //  wUnmanageWindow, which takes the lock for us. We also assume that the
    //  caller releases the lock.
    if (wwin->gid >= 0) {
        remove_from_group(wwin);
    }
    multi_remove_window(wwin->client_win);

    //  Rest of function snipped. No prologue appears before my code insertion
    //  above.
}



/*
 *----------------------------------------------------------------------
 * wManageWindow--
 *  reparents the window and allocates a descriptor for it.
 * Window manager hints and other hints are fetched to configure
 * the window decoration attributes and others. User preferences
 * for the window are used if available, to configure window
 * decorations and some behaviour.
 *  If in startup, windows that are override redirect,
 * unmapped and never were managed and are Withdrawn are not
 * managed.
 *
```

window.c 2/3

```
 * Returns:
 *  the new window descriptor
 *
 * Side effects:
 *  The window is reparented and appropriate notification
 * is done to the client. Input mask for the window is setup.
 *  The window descriptor is also associated with various window
 * contexts and inserted in the head of the window list.
 * Event handler contexts are associated for some objects
 * (buttons, titlebar and resizebar)
 *
 *----------------------------------------------------------------------
 */
WWindow*
wManageWindow(WScreen *scr, Window window)
{
    //  SNIP function prologue - see CD-ROM

    //  DST: Ensure that window initially doesn't belong to a group
    wwin->gid   = -1;

    //  SNIP: A lot of code between the above and the following:

    //  DST: Get parent window, and figure out if this window should be added to
 the group
    Window  root, parent, *children = 0;
    int     numc;

    printf("Querying tree..\n");
    XQueryTree(dpy, window, &root, &parent, &children, &numc);
    if (children)
        XFree(children);

    if (scr->focused_window) {
        printf("Found parent, checking for grouped win.. %d\n", scr->focused_window->gid);
        if (scr->focused_window->gid != -1) {
            printf("Assigning group!\n");
            assign_group(scr->focused_window->gid, 0, wwin, 0);
            set_focused_win(wwin);
            DanielPlaceWindow(wwin, &x, &y, width, height);
            wWindowMove(dpy, wwin, x, y);
        }
    }
    //  Function epilogue snipped. See CD-ROM.
}



/*
 *----------------------------------------------------------------------
 * wUnmanageWindow--
 *  Removes the frame window from a window and destroys all data
 * related to it. The window will be reparented back to the root window
 * if restore is True.
 *
 * Side effects:
 *  Everything related to the window is destroyed and the window
 * is removed from the window lists. Focus is set to the previous on the
 * window list.
 *----------------------------------------------------------------------
 */
void
wUnmanageWindow(WWindow *wwin, Bool restore, Bool destroyed)
{
    WCoreWindow *frame = wwin->frame->core;
```

window.c 3/3

```
    WWindow *owner = NULL;
    WWindow *newFocusedWindow = NULL;
    int wasFocused;
    WScreen *scr = wwin->screen_ptr;

    //  DST: We take the lock here, since we'll soon begin modifying the window
list
    pthread_mutex_lock(&win_lock);

    //  Snip a lot of code dealing with closing the window, before it is finally
    //  destroyed, and we can release the lock:
    wWindowDestroy(wwin);
    pthread_mutex_unlock(&win_lock);
    XFlush(dpy);
}
```

multi_msg.c 1/2

```
/*  multi_msg.c
    (c) 2004-2005 Daniel Stodle, daniels@stud.cs.uit.no

    Small functions for sending and receiving multi-input messages. Code is
    shared with the x2wmx implementation.
*/

#include    "multi_msg.h"
#include    <stdarg.h>
#include    <stdio.h>

/*  multi_msg_send: Sends a message on the given socket, using the
    given message type. All values are treated as 32-bit values, and converted
    to network byte order before put on the network. num_params indicates how
    many parameters are in the message. Returns 0 on success, -1 on failure.
*/
int         multi_send_msg(int sock, uint32_t msg_type, int num_params, ...) {
    multi_msg_t msg;
    va_list     args;
    int         i, sent = 0, res;
    char        *data;

    if (num_params > kMax_msg_params && num_params < 0) {
        printf("Invalid parameter count!\n");
        return -1;
    }
    msg.type    = htonl(msg_type);
    va_start(args, num_params);
    for (i=0;i<num_params;i++)
        msg.data[i] = htonl(va_arg(args, uint32_t));
    va_end(args);

    data    = (char*)&msg;
    do {
        res = send(sock, data+sent, sizeof(multi_msg_t)-sent, 0);
        if (res > 0)
            sent    += res;
        else if (res < 0)
            return -1;
        //  Loop until everything is sent.
    } while (sent < sizeof(multi_msg_t));
    return 0;
}


/*  multi_recv_msg: Receive a multi-input message. Makes sure to always
    return complete messages. Returns 0 on success, -1 on failure.
*/
int         multi_recv_msg(int sock, multi_msg_t *msg) {
    int         i, rcvd = 0, res;
    char        *data;

    data            = (char*)msg;
    do {
        res = recv(sock, data+rcvd, sizeof(multi_msg_t)-rcvd, 0);
        if (res > 0)
            rcvd    += res;
        else if (res <= 0)
            return -1;
        //  Loop until everything is received.
    } while (rcvd < sizeof(multi_msg_t));
    msg->type   = ntohl(msg->type);
    for (i=0;i<kMax_msg_params;i++)
        msg->data[i]    = ntohl(msg->data[i]);
```

multi_msg.c 2/2

```
    return 0;
}
```

multi_msg.h 1/1

```
/*  multi_msg.h
    (c) 2004-2005 Daniel Stodle, daniels@stud.cs.uit.no
    Header file supporting low-level multi-input messaging functions.
*/

#ifndef MULTI_MSG_H
    #define MULTI_MSG_H

//  Includes
    #include    <sys/types.h>
    #include    <sys/socket.h>
    #include    <netinet/tcp.h>
    #include    <arpa/inet.h>
    #include    <netinet/in.h>
    #include    <netdb.h>
    #include    <stdint.h>
    #include    <unistd.h>

//  Constants
enum {
    kMulti_input_port   = 5000,     //  and soon: port+display (ie, 5001 for dis
play :1, etc)

    //  Message types
    kMulti_info_msg     = 0,        //  data[0] = cursor id, data[1] = screen wi
dth, data[2] = screen height
    kMulti_button_msg,              //  data[0] = button number, data[1] = is pr
ess
    kMulti_motion_msg,              //  motion: data[0] = x, data[1] = y (global
).
    kMulti_keyboard_msg,            //  data[0] = keycode, data[1] = is press
    kMulti_going_away_msg,          //  sent to indicate that user is going away
    kMulti_event_ack_msg,           //  sent to acknowledge the last event.

    kMulti_evt_window_size  = 4,    //  number of events a client can send befor
e waiting for an ack. The client will only filter mouse events.

    kMax_msg_params     = 4,
};

//  Typedefs
typedef struct {
    uint32_t    type,
                data[kMax_msg_params];
} multi_msg_t;


//  Prototypes
    int         multi_send_msg(int sock, uint32_t msg_type, int num_params, ...)
;
    int         multi_recv_msg(int sock, multi_msg_t *msg);
#endif
```

x2wmx.c 1/8

```
/*  x2wmx.c
    (c) 2004-2005 Daniel Stoedle, daniels@stud.cs.uit.no

    A simpler version of x2x, supporting only mouse and keyboard events.
    Works ONLY with the modified Window Maker WM.
*/


//  Includes
#include    "x2wmx.h"
#include    "multi_msg.h"
#include    "X11/Xlib.h"
#include    <string.h>
#include    <stdlib.h>
#include    <stdio.h>
#include    <X11/extensions/XTest.h>
#include    <signal.h>
#include    <math.h>
#include    <sys/time.h>
#include    <string.h>

//  Globals
Display     *dpy;
Window      grab_win; // The thin window along the edge, used to grab the cursor
fwd_info_t  fwd;
Cursor      blank_cursor;
int         relative    = 1,
            should_warp = 1,
            pad         = 0;
float       accel       = 1.0;

static unsigned char cursor_blank_bits[] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

int     main(int argc, char *argv[]) {
    XSetWindowAttributes        attr;
    XEvent                      evt;
    XColor                      white, black;
    Pixmap                      pixmap, pmask;
    fd_set                      set;
    struct timeval              t;
    multi_msg_t                 msg;
    uint32_t                    event_mask;
    double                      last_stroke_evt, cur_time;
    float                       rx, ry;
    char                        *hostname;
    int                         scr_height, scr_width, opt, dx, dy, port, i;

    //  Parse arguments
    opt = kOpt_unknown;
    for (i=1;i<argc;i++) {
        if (strcmp(argv[i], "-x") == 0)
            opt = kOpt_set_accel;
        else if (strcmp(argv[i], "-to") == 0)
            opt = kOpt_set_dest;
        else if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
            usage(argv[0]);
            return 0;
        }
        else if (strcmp(argv[i], "-abs") == 0)
            relative    = !relative;
        else if (strcmp(argv[i], "-pad") == 0)
            pad         = !pad;
```

x2wmx.c 2/8

```
        else if (strcmp(argv[i], "-w") == 0)
            should_warp = !should_warp;
        else {
            switch (opt) {
                case kOpt_set_dest:
                    hostname    = strtok(argv[i], ":\n");;
                    port        = atoi(strtok(0, ":\n"));
                    port        += kMulti_input_port;
                    break;
                case kOpt_set_accel:
                    accel       = atof(argv[i]);
                    break;
                default:
                    usage(argv[0]);
                    return 0;
            }
            opt = kOpt_unknown;
        }
    }
    //  Initialize
    printf("x2wmx starting up. Acceleration: %.2f Absolute: %s Pad? %s\n", accel, (relative?"No":
"Yes"), (pad?"Yes":"No"));
    dpy = XOpenDisplay(0);
    if (!dpy) {
        printf("Failed to open display!\n");
        return 1;
    }
    //  We don't want to worry about broken pipes
    signal(SIGPIPE, SIG_IGN);

    //  Create the capturing window. The window is 2 pixels wide, and we capture
    //  the cursor whenever it enters this window.
    attr.override_redirect  = True;
    attr.background_pixel   = BlackPixel(dpy, DefaultScreen(dpy));
    grab_win                = XCreateWindow(dpy, RootWindow(dpy, DefaultScreen(d
py)), 0, 0, 2, HeightOfScreen(DefaultScreenOfDisplay(dpy)), 0, 0, InputOutput, 0
, CWBackPixel | CWOverrideRedirect, &attr);
    event_mask  = ButtonPressMask | ButtonReleaseMask | EnterWindowMask | KeyPre
ssMask | KeyReleaseMask;
    XSelectInput (dpy, grab_win, event_mask);
    XMapRaised(dpy, grab_win);

    //  Create a pixmap for the blank cursor.
    pixmap  = XCreateBitmapFromData(dpy, grab_win,  (const char *) cursor_blank_
bits, 16, 16);
    pmask   = XCreateBitmapFromData(dpy, grab_win, (const char *) cursor_blank_b
its, 16, 16);
    bzero(&black, sizeof(XColor));
    bzero(&white, sizeof(XColor));
    black.pixel     = BlackPixel(dpy, DefaultScreen(dpy));
    white.pixel     = WhitePixel(dpy, DefaultScreen(dpy));
    XQueryColor(dpy, DefaultColormap(dpy, DefaultScreen(dpy)), &black);
    XQueryColor(dpy, DefaultColormap(dpy, DefaultScreen(dpy)), &white);
    blank_cursor    = XCreatePixmapCursor(dpy, pixmap, pixmap, &black, &white, 1
6, 16);

    //  Make us impervious to any clients grabbing the X server.
    XTestGrabControl(dpy, True);

    //  Get dimensions of the local display
    scr_height  = HeightOfScreen(DefaultScreenOfDisplay(dpy));
    scr_width   = WidthOfScreen(DefaultScreenOfDisplay(dpy));
    //  Start the forwarding loop.
    while (1) {
        //  Check if we're forwarding (ie, we have a socket)
```

x2wmx.c 3/8

```
        if (fwd.sock) {
            FD_ZERO(&set);
            FD_SET(fwd.sock, &set);
            //  Poll the socket
            t.tv_sec    = 0;
            t.tv_usec   = 0;
            if (select(fwd.sock+1, &set, 0, 0, &t) > 0) {
                //  Receive a message
                if (multi_recv_msg(fwd.sock, &msg) != 0) {
                    printf("Lost connection?\n");
                    end_forward(event_mask);
                }
                else {
                    //  Process it - we only expect acknowledgements, all
                    //  other messages we can receive are processed during
                    //  connection setup.
                    if (msg.type == kMulti_event_ack_msg) {
                        //  Reset acknowledgement counter.
                        fwd.need_ack                = 0;
                        fwd.mouse_evt_since_last_ack    = 0;
                    }
                }
            }
        }
        //  Any local X events waiting to be processed?
        if (XPending(dpy)) {
            //  Get it and process it
            XNextEvent(dpy, &evt);
            switch (evt.type) {
                case EnterNotify:
                    //  We receive this event when the user's cursor has entered
                    //  our forwarding window. If no socket exists, we start
                    //  a new forwarding session.
                    if (fwd.sock == 0) {
                        start_forward(hostname, port, event_mask);
                        if (pad)
                            last_stroke_evt = current_time();
                    }
                    break;
                case MotionNotify:
                    //  The local cursor moved. Let's relay that info to the
                    //  remote end. We only care about movement events when
                    //  we're actually forwarding.
                    if (fwd.sock) {
                        //  Figure out the amount the cursor has moved since the
                        //  last motion event
                        dx  = evt.xmotion.x_root - fwd.local.x;
                        dy  = evt.xmotion.y_root - fwd.local.y;
                        //  If we're in relative mode, we may need to warp the
                        //  local cursor back to the center of the display
                        //  occasionally. This is because we don't receive motio
n
                        //  events when the cursor is packed up into one of the
                        //  corners/edges, and attempted moved "beyond" the
                        //  corner/edge. This behaviour can be disabled using
                        //  the -w switch.
                        if (relative && should_warp &&
                           (evt.xmotion.x_root-10 <= 0 || evt.xmotion.x_root+10
> scr_width ||
                            evt.xmotion.y_root-10 <= 0 || evt.xmotion.y_root+10
> scr_height)) {
                            XWarpPointer(dpy, None, RootWindow(dpy, DefaultScree
n(dpy)), 0,0,0,0, fwd.center_pt.x, fwd.center_pt.y);
                            fwd.local.x = fwd.center_pt.x;
                            fwd.local.y = fwd.center_pt.y;
```

x2wmx.c 4/8

```
                        }
                        else {
                            //  No warping, so just remember the new position
                            //  for future delta calculations.
                            fwd.local.x = evt.xmotion.x_root;
                            fwd.local.y = evt.xmotion.y_root;
                        }
                        if (pad) {
                            cur_time    = current_time();
                            //  How much time since last time we received a moti
on event?
                            //  If "too much", then we reset the center point.
                            if (cur_time - last_stroke_evt > 0.005) {
                                fwd.center_pt.x = evt.xmotion.x_root;
                                fwd.center_pt.y = evt.xmotion.y_root;
                            }
                            //  Update time of last motion event. We only use th
is
                            //  variable in the context of pad-forwarding, so se
tting
                            //  it here is sufficient.
                            last_stroke_evt = cur_time;
                            //  Calculate deltas using the center point as refer
ence
                            dx  = evt.xmotion.x_root - fwd.center_pt.x;
                            dy  = evt.xmotion.y_root - fwd.center_pt.y;
                            //  Move the center point towards the actual pointer
 position
                            fwd.center_pt.x = (evt.xmotion.x_root + fwd.center_p
t.x) / 2;
                            fwd.center_pt.y = (evt.xmotion.y_root + fwd.center_p
t.y) / 2;
                            //  Convert deltas to floats, and scale them
                            rx  = dx;
                            ry  = dy;
                            //  We don't do negatives!
                            if (rx < 0)
                                rx  = -rx;
                            if (ry < 0)
                                ry  = -ry;

                            //  Scale according to f(x) = (1 + (ln x)^3)*scale
                            if (dx != 0) {
                                rx  = log(rx);
                                rx  *= rx*rx;
                                rx  += 1;
                                rx  *= accel;
                            }
                            if (dy != 0) {
                                ry  = log(ry);
                                ry  *= ry*ry;
                                ry  += 1;
                                ry  *= accel;
                            }
                            //  Give correct sign again
                            if (dx < 0)
                                rx  = -rx;
                            if (dy < 0)
                                ry  = -ry;
                            //  Update remote cursor position. We use the floati
ng point
                            //  values locally for greater precision, and conver
t those
                            //  values to integers afterwards.
                            fwd.f_remote.x  += rx;
```

x2wmx.c 5/8

```
                            fwd.f_remote.y  += ry;
                            fwd.remote.x    = fwd.f_remote.x;
                            fwd.remote.y    = fwd.f_remote.y;
                        }
                        else if (relative) {
                            //  Relative mode, simply add the deltas to the remo
te cursor pos
                            fwd.remote.x    += dx;
                            fwd.remote.y    += dy;
                        }
                        else {
                            //  Absolute - simply scale the local position to a
location remotely.
                            fwd.remote.x    = ((evt.xmotion.x_root * fwd.width)
/ scr_width);
                            fwd.remote.y    = ((evt.xmotion.y_root * fwd.height)
 / scr_height);
                        }
                        //  Is the user about to end forwarding?
                        if (fwd.remote.x > fwd.width-2) {
                            end_forward(event_mask);
                            break;
                        }
                        //  No, but prevent remote cursor from going offscreen!
                        else if (fwd.remote.x < 0)
                            fwd.remote.x    = 0;
                        if (fwd.remote.y > fwd.height)
                            fwd.remote.y    = fwd.height;
                        else if (fwd.remote.y < 0)
                            fwd.remote.y    = 0;

                        //  Finally, only send the event if we don't need an
                        //  acknowledgment first.
                        if (!fwd.need_ack) {
                            if (fwd.mouse_evt_since_last_ack > kMulti_evt_window
_size)
                                fwd.need_ack    = 1;
                            if (multi_send_msg(fwd.sock, kMulti_motion_msg, 2, f
wd.remote.x, fwd.remote.y) != 0)
                                end_forward(event_mask);
                            fwd.mouse_evt_since_last_ack++;
                        }
                    }
                    break;
                case ButtonPress:
                case ButtonRelease:
                    //  Forward button press, if necessary
                    if (fwd.sock) {
                        if (multi_send_msg(fwd.sock, kMulti_button_msg, 2, evt.x
button.button, evt.type == ButtonPress) != 0)
                            end_forward(event_mask);
                    }
                    break;
                case KeyPress:
                case KeyRelease:
                    //  Forward keyboard event, if necessary
                    if (fwd.sock) {
                        if (multi_send_msg(fwd.sock, kMulti_keyboard_msg, 2, evt
.xkey.keycode, evt.type == KeyPress) != 0)
                            end_forward(event_mask);
                    }
                    break;
                case Expose:
                    //  Handle expose events by clearing the window.
                    XClearWindow(dpy, grab_win);
```

x2wmx.c 6/8

```
                    break;
                case MappingNotify:
                    break;
                }
            }
            else {
                //  Nothing to do? Sleep a bit.
                usleep(5000);
            }
        }
        return 0;
}


void        usage(char *name) {
    printf("Usage: %s -to <host:screen> [-x <acceleration>] [-abs] [-w] [-pad]\n", name);
    printf("  -w: Disable cursor warping.\n");
    printf("  -abs: Calculate remote cursor position using local position.\n");
    printf("  -pad: Optimize for usage on a table computer.\n");
    printf("  -x: Set the acceleration when used in pad-mode, a floating point number between 1 and 2.\n");
}

/*  start_forward: Begins cursor and keyboard forwarding to the given host
    on the given port, using the event mask for selecting input to the forwardin
g
    window.
*/
void        start_forward(char *hostname, int port, uint32_t event_mask) {
    multi_msg_t     msg;
    Window          r, w;
    int             wx, wy, mask;

    //  Store the current local cursor position, so we can restore it later.
    XQueryPointer(dpy, RootWindow(dpy, 0), &r, &w, &fwd.orig_pos.x, &fwd.orig_po
s.y, &wx, &wy, &mask);
    fwd.orig_pos.x      += 3;

    //  Connect to the remote screen, and initiate the protocol.
    printf("Beginning forwarding session to %s:%d\n", hostname, port);
    fwd.sock    = connect_to_host(hostname, port);
    if (!fwd.sock)
        return;

    //  We need to know the size of the remote display, so wait for that to
    //  come through.
    if (multi_recv_msg(fwd.sock, &msg) != 0) {
        printf("Error getting message!\n");
        close(fwd.sock);
        fwd.sock    = 0;
        return;
    }
    //  Initialize the forwarding struct.
    fwd.cursor_id       = msg.data[0];
    fwd.width           = msg.data[1];
    fwd.height          = msg.data[2];
    fwd.remote.x        = fwd.width-32;
    fwd.remote.y        = fwd.height/2;
    fwd.f_remote.x      = fwd.remote.x;
    fwd.f_remote.y      = fwd.remote.y;
    fwd.center_pt.x = WidthOfScreen(DefaultScreenOfDisplay(dpy)) / 2;
    fwd.center_pt.y = HeightOfScreen(DefaultScreenOfDisplay(dpy)) / 2;
    fwd.local.x     = fwd.center_pt.x;
    fwd.local.y     = fwd.center_pt.y;

    fwd.need_ack                    = 0;
```

x2wmx.c 7/8

```
    fwd.mouse_evt_since_last_ack    = 0;

    printf("Connected to remote screen. Cursor id is %d, size is %dx%d.\n", fwd.cursor_id, fwd.wid
th, fwd.height);
    //  Grab keyboard and pointer, and warp the local pointer to the center of t
he display.
    XWarpPointer(dpy, None, RootWindow(dpy, DefaultScreen(dpy)), 0,0,0,0, fwd.ce
nter_pt.x, fwd.center_pt.y);
    XGrabPointer(dpy, grab_win, True, PointerMotionMask | ButtonPressMask | Butt
onReleaseMask, GrabModeAsync, GrabModeAsync, None, blank_cursor, CurrentTime);
    XGrabKeyboard(dpy, grab_win, True, GrabModeAsync, GrabModeAsync, CurrentTime
);
    XSelectInput(dpy, grab_win, event_mask | PointerMotionMask);
    XFlush(dpy);
}


/*  end_forward: Closes down the forwarding socket and releases the cursor
    and keyboard. Also restores the local cursor position.
*/
void        end_forward(uint32_t event_mask) {
    printf("Ending forwarding session for cursor %d\n", fwd.cursor_id);
    if (fwd.sock) {
        multi_send_msg(fwd.sock, kMulti_going_away_msg, 0);
        close(fwd.sock);
        fwd.sock    = 0;
    }
    fwd.cursor_id   = -1;
    XWarpPointer(dpy, 0, RootWindow(dpy, 0), 0, 0, 0, 0, fwd.orig_pos.x, fwd.ori
g_pos.y);
    XUngrabKeyboard(dpy, CurrentTime);
    XUngrabPointer(dpy, CurrentTime);
    XSelectInput(dpy, grab_win, event_mask);
    XFlush(dpy);
}


//  connect_to_host: Opens a TCP connection to the given host and port.
int         connect_to_host(char *hostname, int port) {
    struct hostent          *hInfo;
    in_addr_t               ip;
    int                     sock = 0, yes = 1;
    struct sockaddr_in      addr;

    hInfo                   = gethostbyname(hostname);
    if (!hInfo) {
        perror("Hostname lookup failed.\n");
        return 0;
    }
    ip                      = *(in_addr_t*)hInfo->h_addr_list[0];
    sock                    = socket(AF_INET, SOCK_STREAM, 0);
    if (!sock) {
        perror("Unable to create socket.\n");
        return 0;
    }
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
        perror("Unable to set SO_REUSEADDR");
        return 0;
    }
    yes = 1;
    if (setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &yes, sizeof(int)) =
= -1)
        printf("Couldn't set TCP_NODELAY - error is not fatal.\n");

    addr.sin_family         = AF_INET;
```

x2wmx.c 8/8

```
    addr.sin_port           = htons(port);
    addr.sin_addr.s_addr    = ip;
    memset(&(addr.sin_zero), '\0', 8);
    if (connect(sock, (struct sockaddr*)&addr, sizeof(struct sockaddr)) == -1) {
        perror("Unable to connect socket to remote host!\n");
        return 0;
    }
    return sock;
}


/*  current_time: Returns the current time of day down to the precision offered
    by gettimeofday as a double.
*/
double  current_time(void) {
    struct timeval  t;
    double          res;
    gettimeofday(&t, 0);
    res                     = (double)t.tv_sec + (((double)t.tv_usec)/10e6f);
    return res;
}
```

x2wmx.h 1/1

```
/*  x2wmx.h
    (c) 2004-2005 Daniel Stoedle, daniels@stud.cs.uit.no

    Header file for x2wmx.
*/

#ifndef X2WMX_H
    #define X2WMX_H

//  Includes
    #include    <stdint.h>
    #include    <unistd.h>

//  Constants
enum {
    kOpt_unknown        = 0,
    kOpt_set_dest,
    kOpt_set_accel,
    kOpt_help,
};


//  Typedefs
typedef struct {
    int         x, y;
} point_t;

typedef struct {
    float       x, y;
} fpoint_t;

typedef struct {
    int         sock,
                cursor_id,
                width,
                height,
                need_ack,
                mouse_evt_since_last_ack;
    point_t     orig_pos,   //  used to restore local cursor position after grab
bing it.
                center_pt,
                local,      //  last position in root window on local display
                remote;     //  current position on remote display
    fpoint_t    f_remote;   //  remote position in floating point, for the pad m
ode
} fwd_info_t;

//  Prototypes
    void        usage(char *name);
    void        start_forward(char *hostname, int port, uint32_t event_mask);
    void        end_forward(uint32_t event_mask);
    int         connect_to_host(char *hostname, int port);
    double      current_time(void);
#endif
```

xpattern.c 1/5

```
/*  xpattern.c
    (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
    A simple app to display different patterns in an X window.
*/

#include <X11/Xlib.h>
#include <X11/cursorfont.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

//  Macros
#define setrect(r, a, b, c, d) (r.x=a,r.y=b,r.w=c,r.h=d)

//  Typedefs
typedef struct {
    int     x, y, w, h;
} xrect_t;

//  Prototypes
void    usage(char *name);
Window  create_window(Display *dpy, xrect_t size);
void    hide_cursor(Display *dpy, Window w);
void    draw_pattern(Display *dpy, Window w, GC gc, int pat,uint32_t color, int
weight, xrect_t bounds);

//  Pattern constants
enum {
    kNo_pattern     = -1,
    kPattern_rect   = 0,
    kPattern_square,
    kPattern_horiz,
    kPattern_vert,
    kPattern_mesh,
    kNum_patterns,
};

//  Supported patterns
const static char *patterns[kNum_patterns] = { "-rect", "-square", "-horiz", "-vert",
"-mesh" };

int main(int argc, char *argv[]) {
    int             i, j, bg_color, pat, weight, set_color = 0;
    char            *data;
    Display         *dpy;
    Window          w;
    xrect_t         size;
    Colormap        screen_colormap;
    XColor          alloc_color;
    Status          rc;
    GC              gc;
    XEvent          e;
    //  Open display and set some default pattern and color values
    dpy             = XOpenDisplay(getenv("DISPLAY"));
    if (!dpy) {
        printf("Error: Couldn't open display: %s\n", getenv("DISPLAY"));
        exit(1);
    }
    bg_color        = BlackPixel(dpy, DefaultScreen(dpy));
    setrect(size, 0, 0, 400, 300);
    pat             = kNo_pattern;
    weight          = 1;
    alloc_color.red     = 0;
```

xpattern.c 2/5

```
    alloc_color.green   = 0;
    alloc_color.blue    = 0;

    //  Parse arguments
    for (i=1;i<argc;i++) {
        if (strcmp(argv[i], "-black") == 0)
            bg_color   = BlackPixel(dpy, DefaultScreen(dpy));
        else if (strcmp(argv[i], "-white") == 0)
            bg_color   = WhitePixel(dpy, DefaultScreen(dpy));
        else if (strcmp(argv[i], "-red") == 0) {
            alloc_color.red = 65535;
            set_color           = 1;
        }
        else if (strcmp(argv[i], "-green") == 0) {
            alloc_color.green   = 65535;
            set_color           = 1;
        }
        else if (strcmp(argv[i], "-blue") == 0) {
            alloc_color.blue    = 65535;
            set_color           = 1;
        }
        else if (strcmp(argv[i], "-rgb") == 0) {
            if (i+3 >= argc) {
                printf("Error: -rgb requires three arguments.\n");
                usage(argv[0]);
                exit(1);
            }
            alloc_color.red     = atoi(argv[i+1]);
            alloc_color.green   = atoi(argv[i+2]);
            alloc_color.blue    = atoi(argv[i+3]);
            set_color           = 1;
            i                   += 3;
        }
        else if (strcmp(argv[i], "-geometry") == 0) {
            if (i+1>=argc) {
                usage(argv[0]);
                exit(1);
            }
            //  Parse size. We're not very tolerant of the format here, we
            //  just want 4 numbers. Negative positions/sizes are not handled
            //  (they are treated as positive values)
            data    = strtok(argv[i+1], "x+-");
            size.w  = atoi(data);
            data    = strtok(0, "x+-");
            size.h  = atoi(data);
            data    = strtok(0, "x+-");
            size.x  = atoi(data);
            data    = strtok(0, "x+-");
            size.y  = atoi(data);
            i++;
        }
        else if (strcmp(argv[i], "-weight") == 0) {
            if (i+1>=argc) {
                usage(argv[0]);
                exit(1);
            }
            weight  = atoi(argv[i+1]);
            i++;
        }
        else {
            for (j=0;j<kNum_patterns;j++) {
                if (strcmp(argv[i], patterns[j]) == 0)
                    pat = j;
            }
        }
```

xpattern.c 3/5

```
    }
    //  Create window and hide cursor
    w       = create_window(dpy, size);
    hide_cursor(dpy, w);
    size.x  = 0;
    size.y  = 0;
    if (set_color) {
        //  Set background color to whatever the user requested
        screen_colormap = XCreateColormap(dpy, w,
                            DefaultVisual(dpy, DefaultScreen(dpy)), AllocNone);
        rc              = XAllocColor(dpy, screen_colormap, &alloc_color);
        bg_color        = alloc_color.pixel;
    }
    //  Draw the window
    gc  = XCreateGC(dpy, w, 0, 0);
    draw_pattern(dpy, w, gc, pat, bg_color, weight, size);
    XSelectInput(dpy, w, KeyPressMask | ExposureMask | PointerMotionMask);
    while (1) {
        XNextEvent(dpy, &e);
        if (e.type == KeyPress)
            break;
        if (e.type == Expose)
            draw_pattern(dpy, w, gc, pat, bg_color, weight, size);
    }
    XUndefineCursor(dpy, w);
    XCloseDisplay(dpy);
    return 0;
}


void    usage(char *name) {
    printf("Usage: %s [options]", name);
    printf("Options:\n");
    printf("-black, -white, -red, -green, -blue: Control the pattern color\n");
    printf("-rgb <red> <green> <blue>: Pattern color as RGB, range 0-65535\n");
    printf("-rect, -square, -horiz, -vert, -mesh: Pattern to display\n");
    printf("-weight <wgt>: Set line thickness in pixels\n");
    printf("-geometry <AxB+C+D>: Set size and position of window\n");
}


Window  create_window(Display *dpy, xrect_t size) {
    int                     black   = BlackPixel(dpy, DefaultScreen(dpy));
    Window                  w;
    uint32_t                value_mask;
    XSetWindowAttributes    attribs;

    w   = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), size.x, size.y,
                              size.w, size.h, 0, black, black);

    //  Use override-redirect to precisely control position, and avoid wm border
s
    value_mask                  = CWOverrideRedirect;
    attribs.override_redirect   = 1;
    XChangeWindowAttributes(dpy, w, value_mask, &attribs);

    // We want to get MapNotify and Expose events
    XSelectInput(dpy, w, StructureNotifyMask|ExposureMask);

    // Map the window (that is, make it appear on the screen)
    XMapWindow(dpy, w);

    // Wait for the MapNotify event
    for(;;) {
        XEvent e;
```

xpattern.c 4/5

```
        XNextEvent(dpy, &e);
        if (e.type == MapNotify)
            break;
    }
    return w;
}


/*  hide_cursor: Assigns a blank cursor to the given window, effectively
    hiding it.
*/
void    hide_cursor(Display *dpy, Window w) {
    Pixmap      fg, mask;
    int         scr;
    Cursor      crs;
    XColor      white, black;
    char        fg_data[]   = { '\0' },
                mask_data[] = { '\0' };

    //  This simplified cursor hiding (using only one byte for the bitmap)
    //  should really be incorporated into the multi-cursor implementation..
    scr     = DefaultScreen(dpy);
    fg      = XCreatePixmapFromBitmapData(dpy, RootWindow(dpy, scr), fg_data,
                1, 1, BlackPixel(dpy, scr), WhitePixel(dpy, scr), 1);
    mask    = XCreatePixmapFromBitmapData(dpy, RootWindow(dpy, scr), mask_data,
                1, 1, BlackPixel(dpy, scr), WhitePixel(dpy, scr), 1);

    bzero(&black, sizeof(XColor));
    bzero(&white, sizeof(XColor));
    black.pixel     = BlackPixel(dpy, scr);
    white.pixel     = WhitePixel(dpy, scr);
    XQueryColor(dpy, DefaultColormap(dpy, scr), &black);
    XQueryColor(dpy, DefaultColormap(dpy, scr), &white);
    crs             = XCreatePixmapCursor(dpy, fg, mask, &black, &white, 1, 1);
    XDefineCursor(dpy, w, crs);
}


/*  draw_pattern: This function performs the magic of drawing a pattern into
    the given window.
*/
void    draw_pattern(Display *dpy, Window w, GC gc, int pat, uint32_t color, int
 weight, xrect_t bounds) {
    int     i, x, y, interval, fallthrough = 0;

    XSetForeground(dpy, gc, color);
    XSetLineAttributes(dpy, gc, weight, LineSolid, CapButt, JoinMiter);
    switch (pat) {
        case kNo_pattern:
            break;
        case kPattern_rect:
            XDrawRectangle(dpy, w, gc, bounds.x, bounds.y, bounds.w-1, bounds.h-
1);
            break;
        case kPattern_square:
            bounds.x    = (bounds.w/2) - weight;
            bounds.y    = (bounds.h/2) - weight;
            bounds.w    = weight * 2;
            bounds.h    = weight * 2;
            XFillRectangle(dpy, w, gc, bounds.x, bounds.y, bounds.w, bounds.h);
            break;
        case kPattern_mesh:
            //  Mesh = horiz + vert. The fallthrough var prevents the break belo
w
            //  from executing.
```

xpattern.c 5/5

```
        fallthrough = 1;
    case kPattern_horiz:
        x           = bounds.x;
        y           = bounds.y+80;
        interval    = (bounds.h-160) / 3;
        for (i=0;i<4;i++)
            XDrawLine(dpy, w, gc, 0, y+i*interval, bounds.w, y+i*interval);
        if (!fallthrough)
            break;
    case kPattern_vert:
        x           = bounds.x + 80;
        y           = bounds.y;
        interval    = (bounds.w - 160) / 4;
        for (i=0;i<5;i++)
            XDrawLine(dpy, w, gc, x+i*interval, 0, x+i*interval, bounds.h);
        break;
    }
    XFlush(dpy);
}
```

camera_control.py 1/3

```
# camera_control.py
# -*- coding: latin-1 -*-
# (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
#
# This class provides the user with the ability to remote-control a PTZ camera,
# over the network. It needs a running DEVSERV daemon, which handles the serial
# control of the camera. See
# http://dsd.lbl.gov/OldMisc/mbone/devserv/homepage.html for more info about
# the DEVSERV, and http://dsd.lbl.gov/OldMisc/mbone/devserv/Remcam.txt for the
# remote camera protocol description. DEVSERV 1.2 has been modified by me to
# also support controlling the focus of the camera, in accordance with the
# camera's hardware manual. (DEVSERV supported it in its protocol, but did not
# implement focus control.)

# vicmod is the frame grabber module, written by John Markus Bjørndalen.
import mcast, time, string, socket, vicmod

# These are the addresses and ports specified by DEVSERV. Communication goes
# over both multicast and UDP.
camera_mc_addr  = '224.35.36.37'
camera_mc_port  = 5556
camera_udp_port = 5555

class camera:
    mc_sock                 = 0
    udp_sock                = 0
    pkt                     = ""
    camera_server_address   = ""
    vicmod_opened           = 0

    def __init__(self):
        # Open a multicast socket to discover a device server
        self.mc_sock    = mcast.mcast_listener(camera_mc_addr, camera_mc_port)
        self.mc_sock.sendto(self.create_pkt("devserv description"),
                            (camera_mc_addr, camera_mc_port+2))
        # Open a UDP socket for sending commands
        self.udp_sock   = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.udp_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.udp_sock.bind(('', 0))

    # discover: This must be called before any other operations have been
    # performed. It sends out a camera discovery request, and waits for a reply,
    # periodically resending the discovery request.
    def discover(self):
        print "Discovering camera control server.."
        while 1:
            self.pkt = self.mc_sock.poll(do_eval=0)
            if self.pkt != None:
                cam_server                = self.parse_pkt()
                self.camera_server_address = cam_server[0][2]
                print "Camera server detected:",self.camera_server_address
                self.camera_server_address = string.split(self.camera_server_add
ress, ".")[0]
                print "Using short name:",self.camera_server_address
                # The DEVSERV protocol specs allow control over many cameras.
                # We only need the first camera.
                self.cam_name              = "cam 1 "
                break
            else:
                time.sleep(1)

    # parse_pkt: Parses a DEVSERV packet.
    def parse_pkt(self):
        lines       = string.split(self.pkt, '#')
        words       = []
```

camera_control.py 2/3

```
        for i in range(len(lines)):
            words.append(string.split(lines[i], " "))

        print words
        return words

    # create_pkt: Creates a packet suitable for sending to devserv.
    def create_pkt(self, content):
        data        = str(int(time.time())) + " " + socket.gethostbyname(socket.
gethostname()) + "\#" + content
        data        = str(len(data)+1) + " " + data
        return data

    # home: Commands the camera to return to the "home" position.
    def home(self):
        #print "Home!"
        self.send_cmd(self.cam_name + "home")
        self.zoom("A",1)

    # pan: Pans the camera. Method is either A, R or F, for absolute, relative
    # or fractional. The method can also be "home", in which case the camera's
    # pan will return to the home-position. Amount is specified in degrees,
    # except for fractional, which is specified as a float. Speed is an integer
    # controlling how fast the motion is executed.
    def pan(self, method, amount,speed=None):
        #print "Pan",method,amount
        cmd = self.cam_name + "pan " + method + " " + str(amount)
        if speed != None:
            cmd += str(speed)
        self.send_cmd(cmd)

    # tilt: Tilts the camera; see pan for param description.
    def tilt(self, method, amount,speed=None):
        #print "Tilt",method,amount
        cmd = self.cam_name + "tilt " + method + " " + str(amount)
        if speed != None:
            cmd += str(speed)
        self.send_cmd(cmd)

    # zoom: Zooms the camera.
    def zoom(self, method, amount):
        #print "Zoom",method,amount
        cmd = self.cam_name + "zoom " + method + " " + str(amount)
        self.send_cmd(cmd)

    # focus: Guess. Just take a guess. ;)
    def focus(self, method, amount):
        #print "Focus",method,amount
        cmd = self.cam_name + "focus " + method + " " + str(amount)
        self.send_cmd(cmd)

    # send_cmd: Sends the given command on the UDP socket to DEVSERV.
    def send_cmd(self, cmd):
        print self.create_pkt(cmd)
        self.udp_sock.sendto(self.create_pkt(cmd),
                             (self.camera_server_address, camera_udp_port))

    # program: Just a small test program to observe the camera's behaviour.
    def program(self):
        self.home()
        self.zoom("A", 3)
        time.sleep(1)
        self.tilt("A", -15)
        time.sleep(3)
        self.pan("A", -12)
```

camera_control.py 3/3

```
        time.sleep(1)
        self.pan("A", -6)
        time.sleep(1)
        self.pan("A", 0)
        time.sleep(1)
        self.pan("A", 6)

    # grab: Grabs whatever the camera is currently looking at, and saves it in
    # <file_name>.
    def grab(self, file_name):
        self.vicmod_init()
        vicmod.fg_grab_to_file(file_name)

    # vicmod_init: Initializes the vicmod framegrabber module.
    def vicmod_init(self):
        if not self.vicmod_opened:
            vicmod.fg_open(1,1)
            self.vicmod_opened = 1
```

wall_conf.py 1/1

```
#!/usr/bin/python
wall = [6, 4]
proj_ctrl_hostname = "ctrl"
resolution_pr_projector = [1024, 768]
vnchost = 'wks1:1'
mapping = {'d045.Cluster.cs.UiT.No': [14, 15], 'd043.Cluster.cs.UiT.No': [10, 11], 'd040.Cluste
r.cs.UiT.No': [4, 5], 'd039.Cluster.cs.UiT.No': [2, 3], 'd048.Cluster.cs.UiT.No': [20, 21], 'd0
47.Cluster.cs.UiT.No': [18, 19], 'd041.Cluster.cs.UiT.No': [6, 7], 'd044.Cluster.cs.UiT.No': [12,
 13], 'd046.Cluster.cs.UiT.No': [16, 17], 'd038.Cluster.cs.UiT.No': [0, 1], 'd042.Cluster.cs.UiT.N
o': [8, 9], 'd049.Cluster.cs.UiT.No': [22, 23]}
#warp = {}
warp = {\
        0:"0.000000x-31.677347,983.861535x-46.449766,983.707775x726.433762,-6.502571x729.284640",
\
        1:"983.379105x-49.115679,1988.113118x-57.825425,1989.381771x722.342493,982.960506x725.04632
8", \
        2:"1986.005093x-60.636704,3001.293352x-63.098651,3001.975060x721.606635,1988.483244x720.836
019", \
        3:"3000.798422x-61.685690,4013.787997x-57.228114,4016.555574x723.091742,3002.520151x722.852
075", \
        4:"4013.953624x-52.246055,5012.837699x-46.912589,5018.922211x726.086941,4014.225371x723.881
426", \
        5:"5014.555085x-47.629419,6000.000000x-35.935351,6004.967253x731.806276,5015.943123x727.011
093", \
        6:"-8.744840x732.350161,982.541608x726.243429,984.555574x1504.672899,-10.301726x1500.524189
", \
        7:"982.038087x728.166827,1987.194147x724.165190,1992.439249x1502.433650,983.991436x1503.314
702", \
        8:"1989.018741x724.370785,3000.815920x722.571936,3002.604455x1504.670325,1991.143020x1502.4
11211", \
        9:"3001.821739x721.934419,4013.830185x724.808717,4013.900571x1504.681922,3002.813768x1502.9
73030", \
        10:"4013.667557x725.361728,5016.284843x727.236295,5021.440570x1500.703498,4015.134523x1504.
083066", \
        11:"5017.608259x728.385090,6000.862494x734.120128,6005.522363x1496.430151,5021.264316x1502.
458906", \
        12:"-9.614979x1504.127951,982.761709x1504.373415,988.428422x2279.893841,-8.579325x2271.464
848", \
        13:"984.890280x1502.744769,1989.672413x1500.753442,1989.956501x2279.596659,988.567325x2275.
284300", \
        14:"1990.102768x1503.299918,3002.301166x1502.412254,3000.208747x2282.563918,1990.155538x22
77.966377", \
        15:"3001.683845x1503.040151,4012.516240x1504.922013,4011.642361x2282.652915,3000.765032x22
81.040798", \
        16:"4011.416912x1502.321469,5018.605442x1498.282021,5019.493194x2275.691260,4012.903510x22
81.539007", \
        17:"5019.727777x1502.630120,6005.090300x1495.971776,6007.145055x2264.733546,5021.609476x22
73.322702", \
        18:"-6.357116x2268.116895,989.147265x2273.877296,992.026890x3048.586239,-1.764068x3031.712
524", \
        19:"988.157537x2271.500539,1999.524124x2277.640580,1997.360739x3058.825057,988.562271x3045.
907213", \
        20:"1993.152228x2278.380413,3002.891739x2281.412294,3003.246104x3054.588395,1999.503365x30
53.899259", \
        21:"3002.114244x2280.555732,4012.154286x2282.634024,4007.323906x3057.341804,3000.602112x30
54.706312", \
        22:"4007.801044x2279.687847,5014.109818x2275.736901,5013.544731x3050.452244,4010.117772x30
58.726636", \
        23:"5017.235390x2273.088559,6002.526627x2265.969946,6001.377080x3031.677347,5014.112718x30
43.900399", \
        "dummy":"yeah" }
```

mcast.py 1/1

```
# -*- coding: latin-1 -*-
# Code adapted from John Markus Bjørndalen's python multicast class, modified
# only to support successful usage on BSD-ish systems requiring the use of
# the SO_REUSEPORT flag.
import struct, socket, threading, select

mc_addr = '224.1.1.8'
mc_port = 8025
mc_addr_port = (mc_addr, mc_port)

def create_mc_socket(mc_addr=mc_addr, port=mc_port):
    "Creates a multicast socket, with loopback and reuseaddr set"

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    if "SO_REUSEPORT" in dir(socket):
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

    # This is a little hacked, but it reflects the memory layout of the
    # ip_mreq struct (linux: /usr/include/bits/in.h).
    #maddr = socket.inet_aton(mc_addr)
    maddr = socket.inet_aton(socket.gethostbyname(mc_addr))
    iaddr = socket.inet_aton('0.0.0.0') # Any interface
    mreq = maddr + iaddr

    sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_LOOP, 1)
    sock.bind(('', port))
    return sock


class mcast_listener:
    def __init__(s, addr=None, port=None):
        s.mcs       = create_mc_socket(mc_addr=addr, port=port)
        s.mcs_poll  = select.poll()
        s.mcs_poll.register(s.mcs.fileno(), select.POLLIN)

    def poll(s, timeout = 0, do_eval = 1):
        fds = s.mcs_poll.poll(timeout)
        if len(fds) < 1:
            return
        for found in fds:
            if s.mcs.fileno() == found[0]:
                pkt = s.mcs.recv(1500)
                if do_eval:
                    msg = eval(pkt)
                else:
                    msg = pkt
                return msg

    def sendto(s, msg, addr):
        return s.mcs.sendto(msg, addr)
```

ppm_utils.py 1/2

```
# ppm_utils.py
# -*- coding: latin-1 -*-
# (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
# Simple collection of utilties to handle dealing with ppms. That is, it
# handles the PPMs produced by the framegrabber, and nothing more - only P6's.

import string, sys, struct

# read: Reads the given PPM file, and returns its contents as a tuple of size
# and data.
def read(file):
    fd          = open(file, "r")
    fmt         = fd.readline()
    if fmt != "P6\n":
        print "Invalid image format, ppm [P6] expected,", fmt, "found in source"
        fd.close()
        return None,None

    while 1:
        size    = fd.readline()
        if size[0] != '#' and len(size) > 3:
            break

    tmp     = size.split()
    if len(tmp) == 2:
        size    += fd.readline()

    size        = size.split()
    size        = [int(size[i]) for i in range(len(size))]
    data        = fd.read()
    fd.close()
    return size,data

# save: Saves the PPM file.
def save(file, size, data):
    fd          = open(file, "w+")
    fd.write("P6\n"+str(size[0])+" "+str(size[1])+" "+str(size[2])+"\n");
    fd.write(data)
    fd.flush()
    fd.close()

# subtract: Subtracts one set of PPM data from another. This method is dog-slow,
# so use the ppm_sub utility instead.
def subtract(bg_size, bg_data, src_size, src_data, dest):
    if src_size[0:2] != bg_size[0:2]:
        print "Error: Size of source and background differ:", repr(src_size), repr(bg_size)
        return

    x           = min(int(src_size[0]), int(bg_size[0]))
    y           = min(int(src_size[1]), int(bg_size[1]))

    if (src_size[2] > 255 and bg_size[2] < 255) or (src_size[2] < 255 and bg_siz
e[2] > 255):
        print "Error: Differing image depths."
        return

    if int(src_size[2]) > 255:
        increment   = 6
        pack_str    = "HHH"
        max_col_val = 65535
    else:
        increment   = 3
        pack_str    = "BBB"
        max_col_val = 255
```

ppm_utils.py 2/2

```
    i               = 0
    max_len         = len(src_data)
    result          = ""
    fd              = open(dest, "w+")
    fd.write("P6\n"+str(src_size[0])+" "+str(src_size[1])+" "+str(src_size[2])+"\n
")
    last_pct        = 0
    while i < max_len:
        pix     = src_data[i:i+increment]
        src_rgb = struct.unpack(pack_str, pix)
        pix     = bg_data[i:i+increment]
        bg_rgb  = struct.unpack(pack_str, pix)
        src_rgb = [(src_rgb[j]-bg_rgb[j]) for j in range(3)]
        src_rgb = [max(0,src_rgb[j]) for j in range(3)]
        avg     = sum(src_rgb) / 3
        fd.write(struct.pack(pack_str, avg, avg, avg)) #src_rgb[0],src_rgb[1],sr
c_rgb[2])
        i       += increment
        if i*100/max_len >= last_pct+10:
            print str(i*100/max_len)+"%..",
            last_pct    = int(i*100/max_len)

    print "Done."
    fd.flush()
    fd.close()
    return
```

projector_location.py 1/2

```
# projector_location.py
# -*- coding: latin-1 -*-
# (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
# This code simply returns the (x,y) location of the first pixels beyond a
# certain intensity in a number of PPM files.

import time, string, socket, sys, struct

# get_projector_grid: Returns a list of (x,y) locations, one for each image.
# The basename is used to construct the complete image pathnames. A pixel is
# selected if its intensity is > threshold. The intensity is calculated as the
# average of the red, green and blue components of the PPM. The images should
# be pre-processed by having had a background image subtracted first, to prevent
# noise from becoming a problem.
def get_projector_grid(num_projs, basename,threshold):
    print "Analyzing images.."
    locations   = []
    for p in range(num_projs):
        f       = basename+str(p)+".ppm"
        fd      = open(f, "r")
        fmt     = fd.readline()
        size    = fd.readline()
        data    = fd.read()
        fd.close()
        if fmt != "P6\n":
            print "Invalid image format, ppm [P6] expected,",fmt,"found"
            return

        size    = size.split()
        x       = int(size[0])
        y       = int(size[1])
        max_val = int(size[2])

        l       = len(data)
        i       = 0
        ok      = 0
        if max_val > 255:
            while i < l:
                pix     = data[i:i+6]
                rgb     = struct.unpack("HHH", pix)
                rgb     = [socket.ntohs(rgb[j]) for j in range(len(rgb))]
                avg     = rgb[0]+rgb[1]+rgb[2]
                avg     /= 3
                if avg/65535.0 >= threshold:
                    #print "Value found at",x,"x",y,"for rgb:",repr(rgb)
                    locations.append([(i/6) % x,(i/6) / x])
                    ok  = 1
                    break
                i       += 6
            if not ok:
                locations.append([-1,-1])
        else:
            while i < l:
                pix     = data[i:i+3]
                rgb     = struct.unpack("BBB", pix)
                avg     = rgb[0]+rgb[1]+rgb[2]
                avg     /= 3
                if avg/255.0 >= threshold:
                    #print "Value found at",i,"for rgb:",repr(rgb)
                    locations.append([(i/3) % x,(i/3) / x])
                    ok  = 1
                    break
                i       += 3
            if not ok:
                locations.append([-1,-1])
```

projector_location.py 2/2

```
        img = 0
        print "Image",p,"complete"
    print "Locations:",repr(locations)
    return locations

if __name__ == "__main__":
    get_projector_grid(2,"./data/projector_", 0.95)
```

wall_cmd.py 1/1

```
# wall_cmd.py
# Contains only a small method for creating a command. This should be somewhere
# else, it doesn't need its own file.

def create_wall_cmd(type, params):
    cmd = {"type":type, "params":params}
    return cmd
```

wall_common.py 1/1

```
# wall_common.py
# This file contains some common definitions used by both the master and slave.

# Multicast group and port
wall_mc_ip      = '224.10.20.30'
wall_mc_port    = 10101
wall_mc_addr    = (wall_mc_ip, wall_mc_port)

# Name of config file, path relative to the ./conf/ directory.
wall_config_file  = "wall_conf.py"
```

wall_master.py 1/9

```
#!/bin/env python
# -*- coding: latin-1 -*-
# wall_master.py
# (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
#
# This file contains the code for controlling the wall-slaves.

import mcast, time, string, socket, sys, os
import wall_common, wall_cmd, projector_location, ppm_utils
#import camera_control

class wall_master:
    mc_sock = 0 # The socket used for sending commands and receiving replies

    def __init__(self):
        self.cam        = None
        self.mc_sock    = mcast.mcast_listener(wall_common.wall_mc_ip, wall_comm
on.wall_mc_port)
        self.read_config()

    # start_wall: Starts the display wall (ie, starts the VNC viewers). proj is
    # a list of projector IDs to start. If noalign is true, the software
    # alignment mechanism is disabled.
    def start_wall(self, proj=None, noalign=None):
        if proj != None:
            projectors  = [proj]
        else:
            projectors  = [i for i in range(wall[0]*wall[1])]

        if noalign != None:
            self.send_cmd("set_projector_state", {"state":"on", "projectors":projectors, "no
align":1})
        else:
            self.send_cmd("set_projector_state", {"state":"on", "projectors":projectors})

    # stop_wall: Stops the display wall (ie, stops the VNC viewers).
    def stop_wall(self):
        self.set_state("off")

    # kill_wall: Stops any executable started by the slaves (xpattern, VNC viewe
r)
    def kill_wall(self):
        self.set_state("kill")

    # reset_wall: Instructs the display wall slaves to reload their codebase.
    def reset_wall(self):
        self.send_cmd("reset")

    # die_wall: Instructs the slaves to commit suicide. Hostname is the hostname
    # of the slave to instruct, or "all", to instruct all to die.
    def die_wall(self, hostname):
        self.send_cmd("die", {"hostname":hostname})

    # rect_state: Sets the slaves to display a rectangle.
    def rect_state(self):
        self.set_state("calib_image_rect")

    # set_state: Sets the given projector state on all projectors.
    def set_state(self, s):
        projectors      = [i for i in range(wall[0]*wall[1])]
        self.send_cmd("set_projector_state", {"state":s, "projectors":projectors})

    # color_state: Sets the given projectors to the given color, or all if no
    # projectors are specified. Color is a string; either red, green, blue or
    # white.
```

wall_master.py 2/9

```
    def color_state(self, color, proj=None):
        if proj != None:
            projectors      = [int(proj)]
        else:
            projectors      = [i for i in range(wall[0]*wall[1])]
        if color == "black":
            self.send_cmd("set_projector_state", {"state":"off", "projectors":projectors})
        else:
            self.send_cmd("set_projector_state", {"state":"calib_image_"+color, "projectors":p
rojectors})

    # rgb_state: Sets the projectors to the given rgb color. r,g and b are in
    # the interval [0, 65535]
    def rgb_state(self, r, g, b, proj=None):
        if proj != None:
            projectors      = [int(proj)]
        else:
            projectors      = [i for i in range(wall[0]*wall[1])]

        self.send_cmd("set_projector_state", {"state":"rgb", "red":int(r), "green":int(g),
"blue":int(b), "projectors":projectors})

    # execute: Starts the given executable on the specified host, with the
    # specified params. <args> is taken directly from the command line arguments
.
    def execute(self, args):
        if len(args) < 2:
            print "Must have both hostname and command to execute.."
        host            = args[0]
        cmd             = args[1]
        params          = args[2:]
        self.send_cmd("execute", {"hostname":host, "cmd":cmd, "cmd_params":params})

    # terminate: Stops execution of a previously 'execute'd command.
    def terminate(self, host):
        self.send_cmd("terminate", {"hostname":host})

    # configure: Creates and builds the slave configuration. Configuration
    # proceeds in two steps; first by identifying the available slaves, and then
    # building the projector-to-host mapping for each slave. If no vnchost is
    # specified, the default "wks1:1" will be used. projs and res is specified
    # as 6x4 and 1024x768 respectively.
    def configure(self, projs, res, vnchost=None):
        print "Configuring.."
        # Get slave identities
        hosts       = self.receive_identities()
        # Begin creating the config data
        config      = "#!/usr/bin/python\n"
        projs       = projs.split("x")
        projs       = [int(projs[i]) for i in range(len(projs))]
        res         = res.split("x")
        res         = [int(res[i]) for i in range(len(res))]
        config      += "wall = " + repr(projs) + "\n"
        config      += "resolution_pr_projector = " + repr(res) + "\n"
        if vnchost != None:
            config  += "vnchost = "+repr(vnchost)+"\n"
        else:
            config  += "vnchost = 'wks1:1'\n"

        # TODO: The hostname of the control computer is currently hardcoded
        config      += "proj_ctrl_hostname = 'ctrl'\n"
        num_projs   = projs[0]*projs[1]
        config_map  = {}
        cur_id      = 0
        # Create a random projector-to-host mapping
```

wall_master.py 3/9

```
        for host in hosts:
            projectors  = []
            avail       = host["num_projs"]
            while avail > 0 and cur_id < num_projs:
                projectors.append(cur_id)
                cur_id  += 1
                avail   -= 1

            key             = host["hostname"]
            config_map[key] = projectors

        map_start   = len(config)
        config      += "mapping = " + repr(config_map) + "\n"
        map_end     = len(config)
        config      += "warp = {}\n"
        # Write configuration, and instruct slaves to reload it
        self.write_config(config)
        self.read_config()
        self.send_cmd("reload_config")
        # Wait for the slaves to do our bidding!
        time.sleep(0.5)

        #self.rearrange_projectors(projs)

        # Create the mapping
        self.build_projector_mapping()
        # Remove old mapping, re-add it and then rewrite the config file
        config      = config[0:map_start]+config[map_end:]
        config      += "mapping = " + repr(mapping) + "\n"
        self.write_config(config)
        self.send_cmd("reload_config")
        time.sleep(1)
        # And that's it, configuration is updated

    # receive_identities: Sends out a request to identify the available slaves,
    # and returns a list with hostnames and number of projectors controlled by
    # that host.
    def receive_identities(self):
        print "Sending identity request"
        self.send_cmd("identify")
        print "Receiving identities.."
        start           = time.time()
        hosts           = []

        # Receive replies for the next two seconds.
        while time.time()-start < 2:
            reply       = self.mc_sock.poll()
            if reply != None:
                if reply["type"] == "identity":
                    #print "Add host:", repr(reply)
                    hosts.append(reply["params"])
            else:
                time.sleep(0.02)

        print "Found",len(hosts),"hosts."
        return hosts

    # rearrange_projectors:
    # This method is called to determine the correct host-to-projector
    # mapping. It will call the available slaves, asking each slave to
    # illuminate its display, and then instruct the camera to take a snapshot of
    # the display wall. For new display walls, the camera pan, tilt and zoom
    # settings will need to be readjusted.
    def rearrange_projectors(self, projs):
        self.init_camera()
```

wall_master.py 4/9

```
        print "Finding correct host-to-projector mapping.."
        self.cam.zoom("A",1)
        self.cam.tilt("A",-19)
        time.sleep(1)
        self.cam.pan("A",-1)
        time.sleep(1)
        background  = "./data/projector_background.ppm"
        self.cam.grab(background)
        bg_size, bg_data    = ppm_utils.read(background)
        for y in range(projs[1]):
            for x in range(projs[0]):
                proj_id             = x+(y*projs[0])
                self.send_cmd("set_projector_state", {"state":"identify_image", "projectors":[p
roj_id]})
                time.sleep(1.75)
                name                = "./data/projector_"+str(proj_id)+".ppm"
                self.cam.grab(name)
                self.send_cmd("set_projector_state", {"state":"off", "projectors":[proj_id]}
)
                src_size, src_data  = ppm_utils.read(name)
                ppm_utils.subtract(bg_size, bg_data, src_size, src_data, name)

        self.build_projector_mapping()

    # build_projector_mapping:
    # This method figures out which projector is being controlled by which
    # host, by analyzing the images taken in rearrange_projectors. Basically,
    # each host is instructed to display a small, filled, white square. This
    # square (actually just the first white pixel) is then searched for, and
    # identifies the position of the projector on the wall.
    def build_projector_mapping(self):
        print "Building projector mapping"
        # The projector location module takes care of analyzing the pictures
        # for us, and returns a list of pixel locations. The pictures contain
        # a small white square, and the positions between the different squares
        # is used to figure out which particular computer is connected to which
        # projector(s).
        locations   = projector_location.get_projector_grid(wall[0]*wall[1],"./dat
a/projector_",0.80)

        # Append each item's index to the list. This is necessary as the list
        # will be sorted in the next steps.
        for i in range(len(locations)):
            locations[i].append(i)

        # Sort by y-coordinate
        vert_sort       = locations
        vert_sort.sort(lambda x,y: x[1]-y[1])
        horiz_sort      = []
        # Sort every row by x coordinate
        for y in range(wall[1]):
            horiz_sort.append(vert_sort[y*wall[0]:(y+1)*wall[0]])
            horiz_sort[y].sort(lambda x,y: x[0]-y[0])

        # Rebuild the list
        locations       = []
        for loc in horiz_sort:
            locations   += loc

        # Append current logical ID
        for i in range(len(locations)):
            locations[i].append(i)

        # Re-sort locations list to correspond with original order
        locations.sort(lambda x,y:x[2]-y[2])
```

wall_master.py 5/9

```
        # Rebuild the mapping accordingly
        for host in mapping.keys():
            host_projs      = mapping[host]
            new_projs       = []
            for p in host_projs:
                new_projs.append(locations[p][3])

            mapping[host]   = new_projs
        # Phew, and we're done :)

    # init_camera: Resets the camera's position, and creates the camera object,
    # if necessary.
    def init_camera(self):
        print "Initializing camera.."
        if self.cam == None:
        #   self.cam    = camera_control.camera()
            pass
        self.cam.discover()
        self.cam.zoom("A",1)
        self.cam.home()
        time.sleep(1)

    # write_config: Writes the passed-in configuration to disk.
    def write_config(self, config):
        cf  = open(wall_common.wall_config_file, "w")
        cf.write(config)
        cf.flush()
        cf.close()

    # read_config: Reads the configuration and hands it off to the python
    # parser, placing the configuration in our global namespace.
    def read_config(self):
        folder, ourname = os.path.split(__file__)
        conf_file       = os.path.join(folder, "conf/"+wall_common.wall_config_fi
le)
        cf              = open(conf_file, "r")
        data            = cf.read()
        cf.close()
        exec(data, globals())

    # get_conf_dict: This method is used by Wall Manager to access our entire
    # configuration as one dictionary.
    def get_conf_dict(self):
        conf    = { "wall":wall, "vnchost":vnchost, "mapping":mapping, "resolution":res
olution_pr_projector, "projhost":proj_ctrl_hostname }
        return conf

    # send_cmd: This method sends a command with the given parameters (if any)
    # on our multicast socket.
    def send_cmd(self, cmd, params=None):
        if params==None:
            data    = repr(wall_cmd.create_wall_cmd(cmd, {}))
        else:
            data    = repr(wall_cmd.create_wall_cmd(cmd, params))
        self.mc_sock.sendto(data, wall_common.wall_mc_addr)

    # calibrate: This method is used to gather pictures used for the software
    # calibration of the VNC viewers. The final step of the calibration is
    # manual, and must be performed in Matlab. Note that calibration is not
    # used.
    def calibrate(self, basename, grab_loc=None):
        self.init_camera()
        self.stop_wall()
        time.sleep(1)
```

wall_master.py 6/9

```
        print "Beginning calibration run for "+repr(wall)+" wall.."
        # Camera position format consists of one, two or three numbers.
        # When only one or two numbers are present, they are interpreted as the
        # absolute pan and tilt values. If a third value is also present,
        # this third value indicates the zoom. These values need to be re-tuned
        # for new display walls.

        # -15, -10, 6 : -6, 9 : [-15,-11.5,5]
        camera_pos  = {0: [-15,-8.5,7], 1: [-8], 2: [0], 3: [7], 4: [14], \
                       5: [-15,-12],  6: [-8], 7: [0], 8: [7], 9: [14], \
                       10: [-15,-17],11: [-8],12: [0],13: [7],14: [14] }

        self.move_camera(camera_pos[0])
        print "Focusing.."
        #self.cam.focus("A","01A0") # for [-15,-11.5,5]
        self.cam.focus("A","01F5") # for [-15,-8.5.5,7]
        time.sleep(1)
        # The following loop basically moves the camera into position for gettin
g
        # a clear image of the given projector, and then instructs the projector
        # to display the calibration patterns. For each pattern, a picture is
        # grabbed.
        for y in range(wall[1]-1):
            for x in range(wall[0]-1):
                loc = x+y*(wall[0]-1)
                if grab_loc != None and grab_loc != loc:
                    self.move_camera(camera_pos[loc], no_sleep=1)
                    continue
                else:
                    self.move_camera(camera_pos[loc])

                if grab_loc == loc:
                    time.sleep(3)
                    self.move_camera(camera_pos[loc+1])
                    self.move_camera(camera_pos[loc])

                print "Grabbing %.3d.." % loc

                name    = "./data/%s_%.3d_background.ppm" % (basename, loc+1)
                self.cam.grab(name)
                for yy in range(2):
                    for xx in range(2):
                        proj_id     = (x+xx)+((y+yy)*wall[0])
                        name        = "./data/%s_%.3d_%.2d_h.ppm" % (basename, loc+1,
 (xx+yy*2)+1)
                        self.send_cmd("set_projector_state", {"state":"calib_image_horiz", "p
rojectors":[proj_id]})
                        time.sleep(1.75)
                        self.cam.grab(name)
                        self.send_cmd("set_projector_state", {"state":"calib_image_vert", "pr
ojectors":[proj_id]})
                        time.sleep(1.75)
                        name        = "./data/%s_%.3d_%.2d_v.ppm" % (basename, loc+1,
 (xx+yy*2)+1)
                        self.cam.grab(name)
                        self.send_cmd("set_projector_state", {"state":"off", "projectors":[p
roj_id]})
                        time.sleep(1)

    # process_calib_images: This method removes background noise from the
    # captured calibration images, by subtracting a background image from the
    # calibration image. It uses a small C tool for performing the subtraction
    # work, as doing it in python proved to be very very slow.
    def process_calib_images(self, basename, process_loc=None):
        print "Removing background noise from calibrated images.."
```

wall_master.py 7/9

```
        for y in range(wall[1]-1):
            for x in range(wall[0]-1):
                loc = x+y*(wall[0]-1)
                if process_loc != None and process_loc != loc:
                    continue

                print "Preprocessing %.3d.." % (loc+1)
                bg_name         = "./data/%s_%.3d_background.ppm" % (basename, loc+1)
                for yy in range(2):
                    for xx in range(2):
                        name                = "./data/%s_%.3d_%.2d_h.ppm" % (basename
, loc+1, (xx+yy*2)+1)
                        os.spawnl(os.P_WAIT, "./bin/ppmsub", "ppmsub", bg_name, nam
e, name)
                        name                = "./data/%s_%.3d_%.2d_v.ppm" % (basename
, loc+1, (xx+yy*2)+1)
                        os.spawnl(os.P_WAIT, "./bin/ppmsub", "ppmsub", bg_name, nam
e, name)

    # move_camera: Moves the camera to the vector given by <where>, where <where
>
    # contains the values for absolute pan, tilt and zoom values. Only pan needs
    # to be specified, the remaining values are optional.
    def move_camera(self, where, no_sleep=0):
        t   = None
        z   = None
        p   = where[0]
        if len(where) > 1:
            t   = where[1]
            if len(where) > 2:
                z   = where[2]
        self.cam.pan("A",p)
        if t != None:
            self.cam.tilt("A",t)
        if z != None:
            self.cam.zoom("A",z)
            if no_sleep == 0:
                time.sleep(3)
        self.cam.pan("A",p+1)
        if no_sleep == 0:
            time.sleep(0.1)
        self.cam.pan("A",p)
        if no_sleep == 0:
            time.sleep(1)

    # convert_config: Converts the wall configuration into a format suitable for
    # use with Xdmx.
    def convert_config(self, name):
        data    = "virtual displaywall {\n\twall "+str(wall[0])+"x"+str(wall[1])+"\n"

        host    = []
        id  = []
        for p in mapping:
            id.append([min(mapping[p]), p])

        print id
        id.sort(lambda x,y: x[0]-y[0])
        for x in id:
            print x
            data += x[1]+":0 "
            host.append(x[1]+":0")

        data    += ";\n}"
        cf  = open(name, "w")
        cf.write(data)
```

wall_master.py 8/9

```
        cf.flush()
        cf.close()

def usage():
    print "Usage:", sys.argv[0], " <command> [params]"
    print "Where command is one of:"
    print " configure <projectors-horiz>x<projectors-vert> <projector-res-x>x<projector-res-y>"
    print " exec <hostname or all> <path-to-executable> [params]"
    print " term <hostname or all>"
    print " start -- starts a vncviewer on all slaves"
    print " stop -- stops running vncviewers, and brings up a black screen"
    print " kill"
    print " reset -- reloads slave python code on-the-fly"
    print " die -- kills everything, including the slave and X11"
    print " white, red, green, blue, black [projector id] -- set color of all or one projector"
    print " calibrate [basename [grab_loc]] -- grabs calibration images"
    print " process_calib [basename [grab_loc]] -- process calibration images"
    print " rect -- draw a large white rectangle around every projector"
    print "Examples:"
    print " configure 2x1 1024x768"

# Let's get on with the show!
if __name__ == '__main__':
    if len(sys.argv) < 2:
        usage()
        sys.exit(1)

    # Parse arguments and figure out what we need our master to do!
    master  = wall_master()
    if sys.argv[1] == "configure":
        if len(sys.argv) < 4:
            usage()
            sys.exit(1)
        if len(sys.argv) == 5:
            master.configure(sys.argv[2],sys.argv[3],sys.argv[4])
        else:
            master.configure(sys.argv[2],sys.argv[3])
    elif sys.argv[1] == "start":
        proj=None
        noalign=None
        #if len(sys.argv) > 2:
        #   proj = int(sys.argv[2])
        if "noalign" in sys.argv:
            noalign=1
        master.start_wall(proj, noalign)
    elif sys.argv[1] == "stop":
        master.stop_wall()
    elif sys.argv[1] == "exec":
        master.execute(sys.argv[2:])
    elif sys.argv[1] == "term":
        master.terminate(sys.argv[2])
    elif sys.argv[1] == "kill":
        master.kill_wall()
    elif sys.argv[1] == "reset":
        master.reset_wall()
    elif sys.argv[1] == "die":
        if len(sys.argv) < 3:
            hostname = "all"
        else:
            hostname = sys.argv[2]
        master.die_wall(hostname)
    elif sys.argv[1] == "ifconfig":
        master.send_cmd("ifconfig")
    elif sys.argv[1] == "calibrate":
        if len(sys.argv) < 3:
```

wall_master.py 9/9

```
            basename    = "calib"
        else:
            basename    = sys.argv[2]
        if len(sys.argv) > 3:
            grab_loc    = int(sys.argv[3])
        else:
            grab_loc    = None

        master.calibrate(basename, grab_loc)
    elif sys.argv[1] == "process_calib":
        if len(sys.argv) < 3:
            basename    = "calib"
        else:
            basename    = sys.argv[2]
        if len(sys.argv) > 3:
            process_loc = int(sys.argv[3])
        else:
            process_loc = None
        master.process_calib_images(basename, process_loc)
    elif sys.argv[1] == "rect":
        master.rect_state()
    elif sys.argv[1] == "white" or sys.argv[1] == "red" or sys.argv[1] == "green" o
r sys.argv[1] == "blue" or sys.argv[1] == "black":
        if (len(sys.argv) > 2):
            master.color_state(sys.argv[1], sys.argv[2])
        else:
            master.color_state(sys.argv[1])
    elif sys.argv[1] == "rgb":
        if (len(sys.argv) > 5):
            master.rgb_state(sys.argv[2], sys.argv[3], sys.argv[4], sys.argv[5])
        else:
            master.rgb_state(sys.argv[2], sys.argv[3], sys.argv[4])
    elif sys.argv[1] == "reload_config":
        master.send_cmd("reload_config")
    elif sys.argv[1] == "convert":
        if len(sys.argv) > 2:
            name    = sys.argv[2]
        else:
            name    = "wall_conf.converted"
        master.convert_config(name)
    else:
        print "Unrecognized option"
```

wall_slave.py 1/6

```
#!/bin/env python
# -*- coding: latin-1 -*-
# wall_slave.py
# (c) 2004-2005 Daniel Stødle, daniels@stud.cs.uit.no
#
# This file contains the slave implementation for the wall controlling script.

import mcast, time, string, socket, sys, os, signal, posix
import wall_common, wall_cmd

class wall_slave:
    mc_sock                 = 0     # multicast socket for communication
    hostname                = socket.gethostname()
    projectors              = []    # logical IDs of this slave's projectors
    projector_pid           = []    # pid of application controlling projector(s
)
    last_exec_pid           = 0     # pid of application started with "exec"

    def __init__(self, num_projs):
        # Init consists of setting up multicast, reading number of projectors
        # under our control, load configuration, boot X windows and setting the
        # projectors to "off".
        self.mc_sock        = mcast.mcast_listener(wall_common.wall_mc_ip, wall_
common.wall_mc_port)
        self.num_projs      = num_projs
        self.read_config()
        self.check_for_x11()
        self.set_projector_state_for_all("off")

    # start: The slave's main loop.
    def start(self):
        print "Slave on host",socket.gethostname(),"up and running"
        while 1:
            # Check for commands
            cmd         = self.mc_sock.poll()
            if cmd != None:
                # Handle command
                print "Got cmd:",cmd["type"]
                if cmd["type"] == "identify":
                    # Send our hostname and number of projectors as the reply
                    self.hostname   = socket.gethostname()
                    params          = {"hostname":self.hostname, "num_projs":self.n
um_projs}
                    self.send_cmd("identity", params)
                elif cmd["type"] == "reload_config":
                    print "Reloading configuration"
                    self.read_config()
                    self.set_projector_state_for_all("off")
                elif cmd["type"] == "set_projector_state":
                    # Check if our projector(s) is/are in the param set, and if
                    # so, set state.
                    prm             = cmd["params"]
                    state           = prm["state"]
                    for i in range(len(self.projectors)):
                        if self.projectors[i] in prm["projectors"]:
                            self.set_projector_state(i, state, prm)
                elif cmd["type"] == "execute" or cmd["type"] == "terminate":
                    # Execute/terminate a command. If any previous command
                    # is running, it is killed first.
                    prm             = cmd["params"]
                    if self.hostname == prm["hostname"] or prm["hostname"] == "all":
                        print "I will execute:", repr(prm)
                        if self.last_exec_pid != 0:
                            os.kill(self.last_exec_pid, signal.SIGKILL)
                            # Prevent zombies!
```

wall_slave.py 2/6

```
                            os.waitpid(self.last_exec_pid, 0)
                            self.last_exec_pid  = 0
                        # If we got an execute (not a terminate), proceed to
                        # start the new process
                        if cmd["type"] == "execute":
                            cmd_params          = prm["cmd_params"]
                            cmd_params.insert(0, prm["cmd"])
                            self.last_exec_pid  = os.spawnve(os.P_NOWAIT, prm["c
md"], cmd_params, self.environ)
                elif cmd["type"] == "ifconfig":
                    # Simply executes ifconfig. Used for debugging
                    fd      = os.popen("/sbin/ifconfig")
                    data    = fd.read()
                    print data
                elif cmd["type"] == "reset":
                    # Reload our codebase.
                    for i in range(len(self.projectors)):
                        self.set_projector_state(i, "kill")
                    return "reset"
                elif cmd["type"] == "die":
                    # Just die if our hostname (or "all") equals the hostname
                    # param.
                    prm             = cmd["params"]
                    if self.hostname == prm["hostname"] or prm["hostname"] == "all":
                        return None
            else:
                time.sleep(1)

    # check_for_x11: Checks to see whether X is running or not. If an X server
    # isn't detected, one is automatically started up. Once the X server is
    # running, the environment is configured, and X's energy saving features are
    # disabled.
    def check_for_x11(self):
        # This check could really be performed in a much simpler manner, by
        # simply attempting to connect to localhost, port 6000. Should've
        # thought of that before! This technique, however, also works
        # sufficiently well. It basically checks the output of ps to determine
        # if a process named X or X11 is running.
        ps_cmd  = "/bin/ps axco pid,command --noheaders"
        stdout  = os.popen(ps_cmd)
        ps      = stdout.read()
        stdout.close()
        ps_list = ps.split()
        if "X" in ps_list or "X11" in ps_list:
            print "X is running."
        else:
            print "No X server detected. Will start one.."
            self.x11_pid        = os.spawnl(os.P_NOWAIT, "/usr/X11R6/bin/startx", "start
x")
            print "Waiting for X to finish starting.."
            time.sleep(40)

        # Prepare the environment
        print "Setting DISPLAY to :0.0"
        self.environ                = os.environ
        self.environ["DISPLAY"] = ":0.0"
        print "Disabling energy saving, screensaver and terminal bell.."
        # Force display on
        os.spawnle(os.P_WAIT, "/usr/X11R6/bin/xset", "xset", "dpms", "force", "on", self.
environ)
        # Disable engery saving
        os.spawnle(os.P_WAIT, "/usr/X11R6/bin/xset", "xset", "dpms", "0", "0", "0", sel
f.environ)
        # Disable terminal bell
        os.spawnle(os.P_WAIT, "/usr/X11R6/bin/xset", "xset", "-b", self.environ)
```

wall_slave.py 3/6

```
        # Disable screensaver
        os.spawnle(os.P_WAIT, "/usr/X11R6/bin/xset", "xset", "s", "reset", self.environ)
        os.spawnle(os.P_WAIT, "/usr/X11R6/bin/xset", "xset", "s", "off", self.environ)
        os.spawnle(os.P_WAIT, "/usr/X11R6/bin/xhost", "xhost", "+", self.environ)
        print "Done configuring X."

    # get_vnc_args: Gets the appropriate arguments to pass to vnc for the given
    # projector. This is currently a bit hacked to support running one VNC
    # viewer on two projectors simultaneously.
    def get_vnc_args(self, p, params=None):
        p_id    = self.projectors[p]
        print "Getting VNC for projector",p_id
        # Determine where the projector is positioned
        x       = p_id % wall[0]
        y       = p_id / wall[0]
        print "Location is",x,  "x",y,"for wall:",wall[0],"x",wall[1]
        x       *= 1024
        y       *= 768
        args    = []
        args.append("-passwd")
        # TODO: The following argument needs to be modularized somehow. The
        # password file should probably just be placed in the same directory
        # as the script, though this would create problems when the password
        # is changed.
        args.append("/home/daniels/.vnc/passwd")
        args.append("UseLocalCursor=0")
        args.append("DotWhenNoCursor=0")
        if params.has_key("noalign") or not warp.has_key(p_id):
            # We want to run the viewer in "full" fullscreen, covering both
            # projectors. This works because our projectors are connected to
            # the "right" computers, but fails if this is not the case.
            if p_id % 2 != 0:
                return None
            args.append("-rx")
            args.append(str(x))
            args.append("-ry")
            args.append(str(y))
            args.append("-rw")
            args.append("2048")
            args.append("-rh")
            args.append("768")
        elif warp.has_key(p_id):
            args.append("-gl")
            args.append("-align")
            args.append(warp[p_id])
        else:
            print "Error getting args!"
            return None

        args.append("preferredencoding=hextile")
        args.append("-noborder")
        args.append("-FullColour")
        print "Launching vncviewer with:",repr(args)
        return args

    # set_projector_state_for_all: Sets all the projectors controlled by this
    # slave to the given state.
    def set_projector_state_for_all(self, state):
        for i in range(len(self.projectors)):
            self.set_projector_state(i, state)

    # set_projector_state: Sets the given projector to the given state. States
    # are one of: identify_image, calib_image_horiz, calib_image_vert,
    # calib_image_mesh, on, off, kill, rgb, calib_image_{red,green,blue,white}.
    def set_projector_state(self, p, state, params=None):
```

wall_slave.py 4/6

```
        print "Setting projector",p,"to state",state
        geometry    = self.get_geometry_for_projector(p)
        if state == "identify_image":
            self.fork_process(p, "./bin/xpattern", ["-white", "-square", "-weight", "50",
 "-geometry", geometry])
        elif state == "calib_image_horiz":
            self.fork_process(p, "./bin/xpattern", ["-white", "-horiz", "-weight", "1", "
-geometry", geometry])
        elif state == "calib_image_vert":
            self.fork_process(p, "./bin/xpattern", ["-white", "-vert", "-weight", "1", "-
geometry", geometry])
        elif state == "calib_image_mesh":
            self.fork_process(p, "./bin/xpattern", ["-white", "-mesh", "-weight", "5", "
-geometry", geometry])
        elif state == "calib_image_rect":
            self.fork_process(p, "./bin/xpattern", ["-white", "-rect", "-weight", "1", "-
geometry", geometry])
        elif state == "calib_image_white":
            self.fork_process(p, "./bin/xpattern", ["-white", "-square", "-weight", "1024"
, "-geometry", geometry])
        elif state == "calib_image_red":
            self.fork_process(p, "./bin/xpattern", ["-red", "-square", "-weight", "1024",
 "-geometry", geometry])
        elif state == "calib_image_green":
            self.fork_process(p, "./bin/xpattern", ["-green", "-square", "-weight", "1024"
, "-geometry", geometry])
        elif state == "calib_image_blue":
            self.fork_process(p, "./bin/xpattern", ["-blue", "-square", "-weight", "1024",
 "-geometry", geometry])
        elif state == "rgb":
            self.fork_process(p, "./bin/xpattern", ["-rgb", str(params["red"]), str(pa
rams["green"]), str(params["blue"]), "-square", "-weight", "1024", "-geometry", geometry
])
        elif state == "on":
            vncargs = self.get_vnc_args(p, params)
            if vncargs != None:
                geometry    = self.get_geometry_for_projector_dual(p)
                vncargs.append("-geometry")
                vncargs.append(geometry)
                vncargs.append(vnchost)
                self.fork_process(p, "./bin/vncviewer", vncargs)
        elif state == "off":
            self.fork_process(p, "./bin/xpattern", ["-geometry", geometry])
        elif state == "kill":
            self.fork_process(p,sig=signal.SIGKILL)

    # fork_process: Forks a process, storing the pid in the appropriate slot in
    # the projector_pid array. If cmd is none, any current process running
    # for the given projector will be killed, but no new process started.
    def fork_process(self, p, cmd=None, args=[],sig=signal.SIGTERM):
        if len(self.projector_pid) < p:
            print "Warning: Ignoring fork_process for unknown projector",p,"and command '",cmd,"'"
            return

        old_pid = self.projector_pid[p]
        self.projector_pid[p]   = 0
        if cmd != None:
            pid = os.fork()
            if pid == 0:
                args.insert(0, cmd)
                print "Execing…",repr(args)
                os.execve(cmd, args, self.environ)
            else:
                self.projector_pid[p]   = pid
                if old_pid != 0:
```

wall_slave.py 5/6

```
                    print "Killing old process.. Will wait after kill"
                    os.kill(old_pid, sig)
                    print "Waiting..",
                    os.waitpid(old_pid, 0)
                    print "OK!"
        elif old_pid != 0:
            print "Killing old process.. Will wait after kill"
            os.kill(old_pid, sig)
            print "Waiting..",
            os.waitpid(old_pid, 0)
            print "OK!"

    # read_config: Reads the configuration file, and resets the projector state.
    def read_config(self):
        cf      = open(wall_common.wall_config_file, "r")
        data    = cf.read()
        cf.close()
        exec(data, globals())
        if len(self.projector_pid) > 0:
            self.set_projector_state_for_all("kill")

        # Are we included in the new mapping?
        if mapping.has_key(self.hostname):
            self.projectors = mapping[self.hostname]
            self.projector_pid  = []
            for i in range(len(self.projectors)):
                self.projector_pid.append(0)

        print "Projectors:", repr(self.projectors)
        for p_id in self.projectors:
            x       = p_id % wall[0]
            y       = p_id / wall[0]
            print "Projector",p_id,"is at",x,"x",y


    # get_geometry_for_projector: Gets the placement of windows in order to
    # display the window on projector p, in a format suitable for the
    # -geometry switch accepted by most X windows programs.
    def get_geometry_for_projector(self, p):
        geometry    = "1024x768"
        x           = p*1024
        geometry    += "+"+str(x)+"+0"
        return geometry

    # get_geometry_for_projector_dual: Same as above, but this time covering
    # both projectors.
    def get_geometry_for_projector_dual(self, p):
        geometry    = "2048x768"
        x           = p*1024
        geometry    += "+"+str(x)+"+0"
        return geometry

    # send_cmd: Sends a command on the multicast socket
    def send_cmd(self, cmd, params=None):
        if params==None:
            data    = repr(wall_cmd.create_wall_cmd(cmd, {}))
        else:
            data    = repr(wall_cmd.create_wall_cmd(cmd, params))

        self.mc_sock.sendto(data, wall_common.wall_mc_addr)

def usage():
    print "Usage:", sys.argv[0], "<number of projectors>"

if __name__ == '__main__':
```

wall_slave.py 6/6

```
    if len(sys.argv) < 2:
        usage()
        sys.exit(1)

    slave   = wall_slave(int(sys.argv[1]))
    result  = slave.start()
    print result
    # Should we reload ourselves?
    if result == "reset":
        os.execl("/usr/bin/python", "/usr/bin/python", "./wall_slave.py", sys.argv[1])
    print "Slave exiting."
```

PythonGlue.py 1/1

```
# PythonGlue.py, part of PyObjC.

# Skeleton Python source for embedding Python into ObjC programs.
# This source file expects to be run by the ObjC code in PythonGlue.m
# and it expects to live in Contents/Resources of some .app bundle.
# It will add the Resources folder and its PyObjC subfolder to
# sys.path and import any modules found in Resources (which in
# turn makes any PyObjC classes in these modules available to the
# ObjC runtime system).
import os, sys

DEBUG = 0

def main():
    # First find the Resource folder of the current application
    resource_folder, ourname = os.path.split(__file__)
    if DEBUG:
        print "PythonGlue: resource folder:", resource_folder

    # Add this folder and the PyObjC subfolder to sys.path
    sys.path.append(resource_folder)
    sys.path.append(os.path.join(resource_folder, "PyObjC"))

    # Now import all modules from the resource folder
    count = 0
    for filename in os.listdir(resource_folder):
        if filename[-3:] == ".py" and filename != ourname:
            module_name = filename[:-3]
            if DEBUG:
                print "PythonGlue: import", module_name
            __import__(filename[:-3])
            count = count + 1
    if count == 0:
        print "PythonGlue: Warning: no Python modules found"

if __name__ == '__main__':
    main()
```

wall_communicator.py 1/4

```
# wall_communicator.py
# (c) 2004-2005 Daniel Stoedle, daniels@stud.cs.uit.no
# This file consists of bridge-functions between Objective-C and Python.
# The class is instantiated from the PythonGlue.py file, which imports all
# the python code necessary for the wall_ctrl script.

import wall_master
from objc import YES, NO
from Foundation import *
from AppKit import *
from PyObjCTools import NibClassBuilder
import os, sys, popen2, time, signal

NibClassBuilder.extractClasses("MainMenu")
class wall_communicator(NibClassBuilder.AutoBaseClass):
    # init: Creates a master-object, and initializes authentication state.
    def init(self):
        self                    = super(wall_communicator, self).init()
        self.master             = wall_master.wall_master()
        self.ssh_agent          = None
        self.ssh_agent_pipe     = None
        self.has_authenticated  = False
        return self

    # getConfig: Returns the current display wall configuration, as given in
    # conf/wall_conf.py.
    def getConfig_(self, obj):
        conf    = self.master.get_conf_dict()
        vnchost = conf["vnchost"]
        conf["vnchost"]   = vnchost.split(":")[0]
        conf["vncscreen"] = vnchost.split(":")[1]
        return conf

    # startWall: Instructs the master to start the display wall.
    def startWall_(self, obj):
        self.master.start_wall(None, 1)

    # authenticate_withPassword_: Authenticates the user. The <user> variable
    # is currently unused, since authentication is based upon the currently
    # logged-in user's ssh key.
    def authenticate_withPassword_(self, user, password):
        # Check if we have an ssh-agent running
        if self.ssh_agent == None:
            # No agent running, start one and parse its output. The output
            # from ssh-agent indicates the environment variables that need be
            # exported in order for ssh to "know about" the ssh-agent.
            sin, sout, serr = os.popen3("ssh-agent")
            data    = sout.read()
            sin.close()
            sout.close()
            serr.close()
            data    = data.split(";")
            data    = [data[0], data[2][1:]]
            pipe    = data[0].split("=")
            pid     = data[1].split("=")
            os.environ[pipe[0]] = pipe[1]
            os.environ[pid[0]]  = pid[1]
            # Set up the environment for making ssh use an external application
            # for providing a password. We use the askpass executable for this
            # purpose, an application that simply copies the contents of the
            # $ASKPASS_PASSWORD env variable to stdout.
            askpass,file    = os.path.split(__file__)
            askpass         += "/askpass"
            os.environ["SSH_ASKPASS"]       = askpass
            os.environ["DISPLAY"]           = ":0.0"
```

wall_communicator.py 2/4

```
            self.ssh_agent  = int(pid[1])
            self.ssh_agent_pipe = pipe[1]

        # Export password for askpass
        os.environ["ASKPASS_PASSWORD"] = password
        # Check if we need to authenticate
        if not self.has_authenticated and self.ssh_agent != None:
            print "Attempting to authenticate.."
            obj     = popen2.Popen3("ssh-add")
            pid     = obj.pid
            print "Waiting for result.."
            for i in range(10):
                time.sleep(0.2)
                res = os.waitpid(pid, os.WNOHANG)
                if res[0] == pid:
                    if res[1] == 0:
                        self.has_authenticated  = True
                    else:
                        print "Wrong password."
                        self.has_authenticated  = False
                    return self.has_authenticated
            print "Authenticate seems to have failed. Killing:", pid
            os.kill(pid, signal.SIGKILL)
            os.waitpid(pid, 0)

        # Remove password from environment. This is required, as we would
        # otherwise have a rather large security hole, allowing others to see
        # the plaintext password through the terminal application, when launched
        # by Wall Manager.
        os.environ["ASKPASS_PASSWORD"] = "wedontwantnosecurityleaks"
        return self.has_authenticated

    # startVncServer...: Starts a VNC server on the given host, using the specifi
ed
    # geometry, depth and display. Will kill any VNC server already running on
    # the host, assuming that it has privileges to do so. Uses ssh for starting
    # the server.
    def startVncServer_withDepth_andGeometry_onScreen_(self, vnchost, depth, geo
metry, screen):
        print "Will attempt to start VNC server.."
        if self.has_authenticated:
            cmd = 'ssh '+vnchost+' "export PATH=/usr/sbin/:/sbin/:/usr/local/'
            cmd += 'bin/:/usr/local/sbin/:/usr/local/bin:/bin:/usr/bin:/usr/'
            cmd += 'X11R6/bin:$PATH ; cd wallctrl/bin/ ; ./vncserver -kill :'
            cmd += str(screen)+' ; ./vncserver -alwaysshared -depth '
            cmd += str(depth)+' -geometry '+geometry+'"'
            print "Executing:",cmd
            os.system(cmd)
        else:
            print "We are not authenticated yet."

    # startProjectors...: Starts the given projectors. which_proj is a list
    # containing projector locations (either "all" or "x.y"). Uses a small
    # script on the server side to turn the projectors on - this turns out
    # to be a lot faster than doing it "manually" here.
    def startProjectors_whichProjector_(self, projhost, which_proj):
        print "Will attempt to start projectors.."
        if self.has_authenticated:
            cmd = "cd wallctrl/bin/ ; ./p_start.sh"
            if which_proj != None:
                cmd += " "+which_proj
            os.spawnvpe(os.P_WAIT, "ssh", ["ssh", projhost, cmd], os.environ)
        else:
            print "We are not authenticated yet."
```

wall_communicator.py 3/4

```
    # stopProjectors..: Same as above, except the projectors are stopped instead
.
    def stopProjectors_whichProjector_(self, projhost, which_proj):
        print "Will attempt to stop projectors.."
        if self.has_authenticated:
            cmd = "cd wallctrl/bin/ ; ./p_stop.sh"
            if which_proj != None:
                cmd += " "+which_proj
            os.spawnvpe(os.P_WAIT, "ssh", ["ssh", projhost, cmd], os.environ)
        else:
            print "We are not authenticated yet."

    # setColor: Sets the given r,g,b color on the wall. The rgb values are
    # expected to lie in the domain 0-65535.
    def setColor_green_blue_(self, r, g, b):
        print "Set color",r,g,b
        self.master.rgb_state(r,g,b)

    # setPattern: Sets a pattern on the wall. pat is a constant between 0 and 5,
    # whose value corresponds to the various possible patterns: vertical lines,
    # horizontal lines, mesh (vert+horiz), bounding rectangle, small white squar
e
    # and off (ie, an all black screen).
    def setPattern_(self, pat):
        print "Setting pattern",pat
        if pat == 0:
            self.master.set_state("calib_image_vert")
        elif pat == 1:
            self.master.set_state("calib_image_horiz")
        elif pat == 2:
            self.master.set_state("calib_image_mesh")
        elif pat == 3:
            self.master.set_state("calib_image_rect")
        elif pat == 4:
            self.master.set_state("identify_image")
        elif pat == 5:
            self.master.set_state("off")

    # shutdown: Called when Wall Manager quits, and is responsible for cleaning
    # up. This clean up only consists of killing the ssh-agent process, if it is
    # running.
    def shutdown(self):
        if self.ssh_agent != None:
            os.kill(self.ssh_agent, signal.SIGKILL)
            self.ssh_agent  = None

    # probeCluster: Uses the "identify" message to locate the cluster nodes on
    # the local network. The return value is an array of dictionaries, each
    # dictionary containing two keys: "hostname" and "num_projs". Wall Manager
    # compares the list to the expected hostnames, and reports any missing clust
er
    # nodes to the user.
    def probeCluster(self):
        return self.master.receive_identities()

    # startXdmx: An attempt at allowing Xdmx to be started from the GUI. It does
    # not work yet, because Xdmx does not daemonize itself after starting up. A
    # possible solution to this problem is to run Xdmx in a screen or using nohu
p,
    # though these approaches have not yet been implemented nor experimented wit
h.
    def startXdmx_onScreen_(self, host, screen):
        print "Will attempt to start Xdmx.."
        if self.has_authenticated:
            self.master.stop_wall()
```

wall_communicator.py 4/4

```
            cmd = 'ssh '+vnchost+' "export PATH=/usr/sbin/:/sbin/:/usr/local/'
            cmd += 'bin/:/usr/local/sbin/:/usr/local/bin:/bin:/usr/bin:/usr/'
            cmd += 'X11R6/bin:$PATH ; cd wallctrl/bin/ ; ./vncserver -kill :'
            cmd += str(screen)+' ; Xdmx :1 -configfile ./conf/xdmx.conf -config'
            cmd += ' displaywall +xinerama -ignorebadfontpaths -ac"'
            print "Executing:",cmd
            os.system(cmd)
        else:
            print "We are not authenticated yet."
```