

Extending the Access Grid environment to enhance usability

Internship with VisLab at the
University of Queensland
27/07/2005 - 15/01/2006

Cihan Altinay
Student of Computer Science
Technische Universität Ilmenau
Student number 31609

Table of Contents

1. Introduction.....	3
1.1. VisLab.....	3
1.2. The Access Grid.....	3
2. My tasks at VisLab.....	4
2.1. Adjustments to the 8MP tiled display.....	4
2.2. Cursor-Applet.....	7
2.3. Accelerated graphics with Chromium.....	12
2.4. X2X-mod.....	15
2.5. Unifying the display & capture machines.....	18
2.6. Echo canceler setup under Linux.....	23
2.7. “Shared PowerPoint” under Linux.....	28
3. Conclusion.....	31
4. References.....	32

1. Introduction

1.1. VisLab

Established in 1992 by Prof. Bernard Pailthorpe, the Visualization and High-Performance Computing Laboratory (VisLab) at the University of Sydney [1] is, together with its newly founded counterpart at the University of Queensland (UQ) [2], leading in Australia in the areas of advanced visualization, grid computing and computational science. Both sites are equipped with high-end hardware such as digital projectors, high-resolution displays, SGI Altix supercomputers, RAID arrays and workstations for the use of both, staff and students. Sydney VisLab is also home of the first Australian Access Grid (AG) [3] (see section 1.2) node which was installed in 2001. Today, several nodes are available at the University of Queensland. With their own node at UQ VisLab, staff members and interns have excellent facilities to develop and enhance AG software. Other activities include consulting and installing new AG nodes for interested parties.

1.2. The Access Grid

Aiming to boost productivity and ease collaboration between multiple sites, the AG is still expanding at a high rate. With over 20 room nodes in Australia alone and hundreds more in 47 countries it offers a unique environment that goes far beyond common video conferencing. Anybody can join the Access Grid as long as the equipment requirements are met. For a room node this means that besides the computers that run the software and have a fast Internet connection several video cameras, projectors, speakers, microphones, echo cancelers, appropriate lighting and space for desks and chairs have to be available [4] (see Figure 1). Once a node is set up the operator can use the Venue Client which is part of the multi-platform open source AG software to navigate to a virtual venue.

Similar to real life meetings, all sites that are located in the same virtual venue can see each others video streams and talk to each other. It is also possible to share data by uploading it to a venue, but the most distinct feature of the AG software is the ability to run shared applications. An example for a useful shared application is *Shared Presentation*. Every site that joins a Shared Presentation session sees the same slide which is loaded into a *local* presentation software. A master (usually, but not necessarily, the

presenter) can advance slides which are then synchronized for all participants. With its modular design, the software enables developers to add new shared applications to the numerous existing ones. Interesting new applications include a remote temperature monitoring tool and sharing of an electronic microscope.



Figure 1: A typical Access Grid set up with 3 projectors, front and ceiling cameras, microphones and the control machine on the right hand side

2. My tasks at VisLab

Sections 2.1 to 2.7 detail my work at VisLab in chronological order. Although they were generally started one after another, I changed or improved the results of projects that were considered completed as I received feedback or gained new knowledge throughout my time at VisLab.

2.1. Adjustments to the 8MP tiled display

UQ VisLab houses a high resolution rear projected tiled display that is driven by a 2x3 projector array. The projectors are mounted on a customized positioner behind the screen which allows fine grained movements of each individual projector in all 3 dimensions as well as tilting. With a total resolution of 3840x2048 pixels (or 8 mega pixels) the idea is to have virtually one large screen for visualization purposes as demonstrated in Figure 2. The JVC projectors are fitted with short throw lenses to get a usable screen size of approximately 130x64cm.

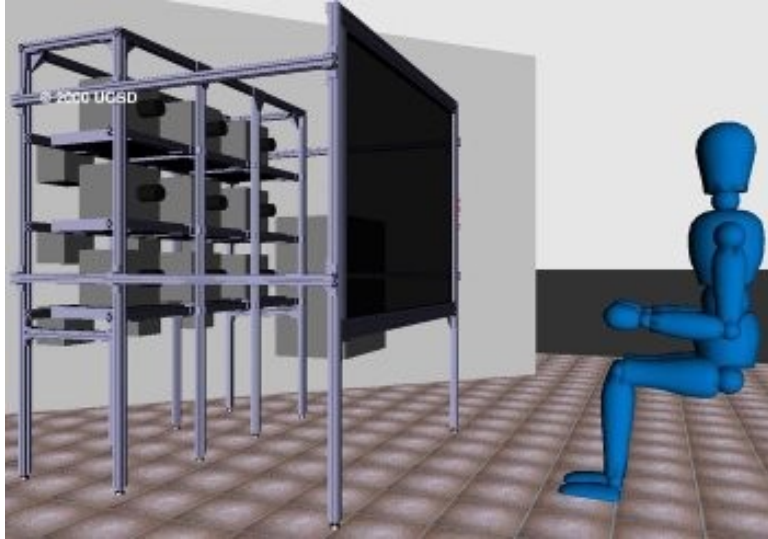


Figure 2: Concept of a rear projected tiled display, here with 9 projectors mounted on a frame

However, with the pre-mounted lenses a noticeable pincushion effect makes the viewer see the edges of each tile rather than one large display as intended (see Figure 3, left). My task was thus to find a way to remove the effect. With a large set of adjustable menu options, it was likely that the projectors have a built-in setting to counteract pincushion distortion. Studying the manual and finding a setting was therefore a first attempt. Unfortunately, no suitable setting was found.

It is essential that the projection plane is exactly perpendicular to the projectors and that the distance between each projector and the plane is equal. To verify that this is the case, the positioner was used to realign the projectors and look for improvements. I soon realized that it is a very time consuming task to keep all 6 projectors aligned while moving or changing settings of just one.

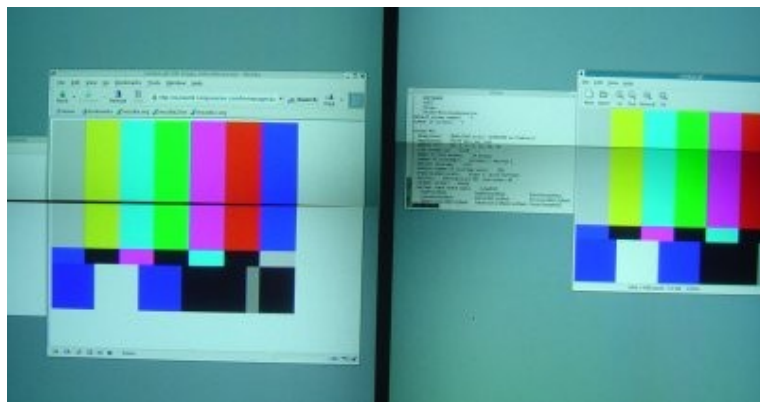


Figure 3: Clearly visible pincushion effect with the original lenses (left) but nearly invisible with the new ones (right)

The idea was that increasing the distance between projector and screen should theoretically make the pincushion effect less visible. After more than one day of adjust-align-measure iterations no satisfactory result was achieved and the conclusion was made that the only way to improve the situation was to use different lenses.

JVC was informed about the problem and sent two replacement lenses with a different throw ratio in response. If these would perform better, all 6 projectors could be equipped with these lenses. My delicate task was to exchange one lens at a time of the \$10,000 projectors. Having removed the case, it was not clear at first how the new lens would fit in the same place as the old one as both, diameter and length varied significantly. Without any labels it was well possible that the lenses were not suitable for the projector model. Therefore, before detaching the existing lens the mount instructions for the replacement were read carefully to confirm they can be applied to this projector. It turned out that additional elements that were shipped with the new lens had to be attached to the projector case first. Only then was it possible to affix the larger new lens at the right place. What was not documented though, was the necessity to change the pin-assignment of the cable that connects the focus motor with the main projector board to make the motor work. Also, I didn't succeed attaching the top cover back on as the motor of the new lens sticks out.



Figure 4: Connecting the motor of the large replacement lens

However, the greatest concern was the fact that there was no way of getting a sharp image at first. Obviously the distance between the light source and the lens was incorrect. After various attempts to change that we found the washers that were responsible. Amazingly the width of just some millimeters made a great difference and we were able to get a sharp picture after removing the washers. With that knowledge I exchanged the lens of a second projector to be able to see whether the pincushion distortion was still visible. Both

projectors were put back into the frame and switched on. But without moving the whole frame away from the screen it was impossible to join the two projections because their size changed to approximately half the size of the ones generated by the original lenses. So a new cycle of moving, adjusting, measuring and checking started with different results at different distances. In the best case the gap between the two projections was barely visible which was good news (see Figure 3, right). For the case where the projectors are mounted side-by-side the gap was a bit larger. But even after days of adjustments I did not manage to obtain a picture with no gap (or no distortion) at all. Having achieved such a picture with different projectors at Sydney's VisLab before, it was a surprise that these *newer* projectors performed worse. After reporting back to JVC we were told that a technician would come over and take a look for himself. But that didn't happen during my time at VisLab.

2.2. Cursor-Applet

Typically, an Access Grid room node has a large tiled display driven by several projectors. The three projectors at UQ VisLab are combined to a 3840x1024 pixel tiled display that is projected onto an area of approximately 5.3 by 1.6 meters. The operator usually sits several meters away from the wall adjusting settings using dialogs via a flat panel monitor while the video windows are projected onto the wall (see Figure 1). But with all windows originating from the same Venue Client the tiled display and the monitor have to be "connected" so that windows can be dragged from the monitor over to the wall. In fact, as participants of a meeting appear and disappear the operator often has to maneuver application and video windows using the mouse from his control monitor to the wall or vice versa. DMX (Distributed Multihead X) is part of modern X servers and enables merging several X servers so that they act as one large desktop. It is successfully in use at VisLab for some time. However, at resolutions and distances as those mentioned, the projected default X11 mouse cursor has a diameter of no more than 1cm and is simply invisible. Even the largest available size of this cursor is often not large enough to be seen clearly on the wall. On the other hand, if this large cursor is moved across to the control screen its size is too disturbing to be usable. An application was needed that can change the mouse cursor size on the fly whenever the operator wishes to.

I started my work on this by researching where cursors are stored and how they are loaded by the X server. Beyond that, I tried to find out what was needed to exchange the current

cursor (theme) by a new one on the fly. The way it works is that the X server reads a cursor theme (whose name is specified in a configuration file) when it is started and each window that is created inherits the current theme. A theme consists of several Xcursor files (see below), one for each shape that the cursor can have (left pointer, hourglass, hand etc.). Once a theme is loaded, an application can change the shape by calling appropriate library functions. Alternatively, it is possible to set the shape to a custom bitmap. But in both cases the changes only affect the current application window and there is no way of changing the theme of the server directly so that the changes are reflected in all running applications. With this knowledge I downloaded the X.org 6.8.1 source tree to see if I can find a way of doing it anyway. The search gave me an insight to the lowest levels of the X server and helped understanding how the components work together. Changes to the server code would only be a last resort as that would involve a lot of work for users of the application. Instead, the utility programs were inspected that are included in the tree to find clues for a solution. Inspired by the utility *xwininfo* the idea was born to recurse over all windows starting from the root and change the cursor shape to a custom image. A first C program *xswitchcur* was written to confirm that it works this way. After starting up, it loads an Xcursor file with the *XcursorFilenameLoadCursor* function and then sets the current cursor to this bitmap recursively in all windows via *XDefineCursor*.

Although this worked quite well I soon realized that more had to be done. Two obvious problems were that new windows did not inherit my custom cursor and all but the "left pointer" shape still used the theme that was currently set. The first fact meant that either all cursors of the current theme are cached by the X server or they are loaded whenever a window is opened. To see in which way it works the theme name to use was changed during an X session. This is easily done for the current user by specifying the desired name in the file *~/.icons/default/index.theme*. One of the alternative cursor themes that ship with the current X server "redglass" was chosen because its red color is easily distinguished from the default black cursor. Next, a random application with a window was started and, surprisingly, the cursor turned red when it entered the window. This meant that the cursor files are not cached but each and every time a window is created they are reloaded for this window. All that was left to do was updating all "old" windows but that's exactly what *xswitchcur* does.

Now that I was able to switch themes on the fly I had to look into the main problem of changing the cursor size. The well-documented Xcursor specification states that an Xcursor file is made up of one or more bitmaps with different sizes. Similar to telling the X server which theme to use there is a way to tell it which cursor size is requested.

But unlike the trick with the `index.theme` file it is not possible to change the size while the X server is running. This was not a problem as I first thought because the size is simply a "nominal" size. Thus, it is possible to have two different cursors which both have a *nominal* size of 16 but where one is an 8x8 pixel bitmap whereas the other is 80x80 pixels large. In other words, the size 16 is not related to the real bitmap size. So this was exactly what I did next, namely creating a new cursor with a huge size of 80x80 pixels but the same nominal size of 16 that the currently active cursor had. The program *xcursorgen* which also ships with the X server can be used to convert images into Xcursor files with the help of a configuration file. Loading the large redglass template into the Gimp a simple resize operation was needed to get the desired size. Then, an appropriate configuration file was created and the bitmap was converted into an Xcursor file with *xcursorgen*. These steps had to be repeated with all 26 shapes that are used and took several hours.

Unfortunately, there was no way of automating them because with every shape a hot spot position had to be saved in the configuration file. That position differs from shape to shape and can not be automatically derived from the bitmap.

Now that my new theme "redglass64" was ready it had to be copied into the `~/.icons` folder. A new program *xcursorsize* was created using elements from *xswitchcur*. It is invoked with a command line parameter to specify whether to use a large cursor (redglass64) or a small cursor (redglass). Instead of changing the `index.theme` file, a simpler way was found leaving the theme name as is but linking a different folder to the theme directory. For example if X is configured to use the redglass theme, it looks for cursor files inside the directory `~/.icons/redglass/cursors`. This directory may be a symbolic link which is convenient because when setting the cursor "size" all that had to be done was to relink that folder to point to either redglass64 or redglass. After that, the same recursive method is used to update existing windows and the program can exit. Both, program and cursor files were transferred to the 3-projector machine for a preliminary test. *xcursorsize* worked nearly as expected but instead of seeing a large red transparent cursor (redglass theme) the cursor looked pixelated and black. Because this was not the case on the development computer I suspected a differing X server version to be the reason so I investigated. Disabling the DMX extension revealed that the problem was related to this extension because the cursor looked correct without it. The fact that enabling DMX also caused icons on the Gnome desktop to disappear and the main menu to look wrong led to the assumption that there is a bug in the DMX extension. I started an Internet search on this and although the DMX project page on sourceforge [5] is rather outdated since DMX became part of the X server, a bug report there showed that others had similar problems

when using DMX with Gnome or KDE. One member provided a workaround, namely disabling the *XRender* extension (using the `-norender` option) when using DMX. Having done that, the icons and menu worked fine and the cursor stopped looking pixelated. However, instead of being red it was still black. Further investigation and research didn't reveal the reason but the fact that all cursors looked black and white no matter which colors they were really using. I posted this issue in a separate bug report. Unfortunately, nobody seemed to have a solution for it. An obvious way to deal with it was to create two new themes using black and white cursors respectively. With the new themes, `xcursorsize` would be a working solution for the problem stated in the beginning.

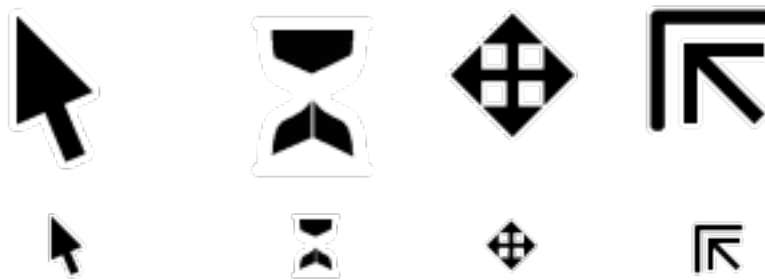


Figure 5: Four shapes of the final "simpleblack80" (top) and "simpleblack32" (bottom) themes

It is a fact though, that having to start a console program with the appropriate parameter each time the cursor size is to be switched is not a good solution from the usability point of view. Creating shortcut icons that invoke the program was better but still meant having two icons. Furthermore, a simple way was needed to select a theme for the large cursor and the small cursor case. In short, the functionality of `xcursorsize` had to be wrapped up in a graphical user interface.

Since I had no experience programming with any of the graphical toolkits that are available under Linux, I read some introductions into the main ones (namely GTK, QT, wxWindows) before I decided to go ahead with GTK. One of the main reasons for using GTK was the possibility to create applets that integrate into the Gnome panel which was available on the projector machine. After reading tutorials and downloading sample applets it seemed appropriate to learn using the GNU Autotools [6] at the same time. They take care of checking for dependent libraries while providing an easy and common way for the end user to build and install the final package. The aim was to provide the program including source code to the community through the VisLab web page which is why I also took a closer look at the details of the GNU General Public License and how to correctly apply the license to my source files.

Instead of providing two separate files I decided that it would be best to include *xcursorsize* in the package of the new program *cursor-applet*. That way users have the possibility to choose between a command line tool and a GUI. Therefore, *xcursorsize* was changed so that the functionality that both programs have in common could be written in a separate file instead of duplicating it. The applet was designed to make switching the cursor size as easy as clicking on its icon in the panel. Since a way was needed to tell the program which folders contain the large and small icons respectively I researched about the best way to keep configuration data in the Gnome environment. I found out that Gnome keeps a database of configuration data ("GConf") that can be used to store application specific settings. After reading through the API documentation which explained how to use the database from C programs, the *cursor-applet* source was changed accordingly. A configuration dialog was programmed that can be accessed by the pop up menu with a right click. With its help, the user can enter a folder name containing the small and large cursors respectively. When switching size the application links the corresponding folder to *~/.icons/cursor-applet* and updates open windows as discussed above. This works because the X server is told to use the 'cursor-applet' theme upon installation of the applet.

With this applet fully functional I now looked into the possibility of automating the switch between the two sizes completely. To recall: whenever the cursor moves out of the control monitor onto the projected screens (which all form one desktop by using DMX) the cursor size had to be switched to large whereas a small cursor had to be set as soon as it re-enters the control screen. If there was a way to track the cursor position this could be done automatically. Studying various articles and tutorials it became clear that there is no way of letting the DMX server report back when the cursor changes screens. Similarly, tracking the movement of the cursor without intercepting mouse input does not work. Therefore, the way of polling the cursor position at a specific interval was chosen although it was a less attractive approach and at first not clear whether it would work flawlessly. A further complication was revealed after reading the DMX documentation. It is not possible to simply retrieve the currently active screen number. Instead, my new program *dmxcursor* was programmed to request the DMX screen set up at start up and store the extents of the control screen in a custom data structure *Frame*. Then, it enters an endless loop polling the mouse cursor position and checking whether it crossed the border between control screen and non-control screen. If necessary, the cursor size is changed and after a delay the loop starts again. After various tests with different intervals I was amazed how effective the program was without increasing the CPU usage notably. I made the interval and control

screen number customizable by a command line parameter and included a help switch explaining those parameters. Similar to `xcursor` no code was duplicated but the same cursor switching code was shared with the other programs. After adjusting the Makefiles and integrating `dmxcursor` into the `cursor-applet` tree I repackaged all files including the two cursor themes in two sizes and a manual page. I completed the project by writing an HTML page with a quick explanation and a download link for the package.

2.3. Accelerated graphics with Chromium

Each projector of the VisLab 3-projector tiled display introduced in section 2.2 is driven by a separate computer equipped with high-quality NVidia graphics cards. During an Access Grid session the powerful 3D capabilities of these adapters are unused as only 2D windows and dialogs are projected. But having such a display cluster in place, it can as well be used as a means of visualizing, inspecting and presenting 3D data with the use of OpenGL based professional visualization software such as VTK and OpenDX. Although DMX makes it possible to use the full extent of the 3 screens for display, it only allows using the hardware acceleration capabilities of the front-end machine which is useless in this case as it is the separate control machine.

Chromium is an open source project based on WireGL that enables accelerated rendering on clusters of workstations. It intercepts calls to OpenGL functions and decomposes them with special algorithms to spread the rendering load between the graphics adapters. Current versions of Chromium support DMX making it ideal to try with the VisLab display cluster. There are several pages of on-line documentation [7] available which I first went through to get a basic understanding of the functionality. After downloading the Chromium source package the configuration had to be adapted to use DMX before building the executables. The time it took to compile and build was used to take a closer look at the way Chromium is set up. As mentioned earlier, Chromium has to be able to intercept OpenGL calls. To make this work, the shared OpenGL library which programs load dynamically is replaced by a fake one that uses Chromium. Under an environment like Linux this is easily accomplished by appending the path to the fake library to the `LD_LIBRARY_PATH` environment variable. Having done that, the first try was to see whether Chromium is set up properly and works on the single machine. Chromium is configured with executable Python scripts that contain everything necessary to run an application. Most importantly, it contains a number of *SPUs* (Stream Processing Units)

that are connected to form a directed acyclic graph which is also called SPU chain. OpenGL function calls pass this chain and are processed by each SPU transparently. The most important SPU is the 'render' SPU which actually renders the OpenGL commands into a window.

Some sample configuration files are provided and I ran the Atlantis demo on the local machine to test the functionality. To do so, I had to execute the configuration using Python which itself runs the master controller or *mothership*. Then a *crserver* had to be started which waits for commands from the mothership. In this simple case that was all to do and I could see the Atlantis demo running in a window. In a multi-computer set up though each of the display machines have to have its own *crserver* running to receive commands from the mothership. Since it would be rather complicated to manually start a *crserver* on each machine before being able to start an OpenGL application, there is a way to automate that given that the machine running the mothership can connect to the display machines without a password in a secure manner. In the VisLab case this is possible because the 3 display machines are connected to the control machine via a separate private network. If everything is set up correctly starting an application that requires OpenGL in the conventional way triggers the execution of a Chromium configuration file by Python. This file creates a mothership before connecting to each of the cluster machines using a secure shell client. After starting up a *crserver* on each of those machines the SPU chain is loaded which now processes the OpenGL commands from the client application.



Figure 6: Atlantis accelerated by Chromium running over all 3 projectors at VisLab

Setting this up was relatively easy because a sample configuration (*autodmx.conf*) is provided with Chromium. I had to change this file to reflect the correct installation path and to use *ssh* instead of *rsh* to connect to the back-end machines. Then, a *~/.crconfigs* file was created that contains the path to this configuration file. The fake GL library uses this file to find the correct Python program. Finally, the necessary Chromium files (particularly *crserver*) were transferred to each cluster node. Figure 6 shows a successful test run of Atlantis using all three displays.

Now that Chromium worked with a demo program it was interesting to find out about the performance compared to a DMX-only system. *glxgears* gave a first idea with its fps (frames per seconds) output which jumped from 224fps to 285fps enabling Chromium. But it is known to depend on too many factors that can falsify the results. Therefore, I decided to download and try one of the visualization toolkits to see if there is a noticeable performance boost when using Chromium. Since I had experience with OpenDX, it was my choice and I compiled the package and tried it out. Not knowing that DX does not use OpenGL by default I wondered why Chromium did not start up and the graphics were slow as before. After trying different things I found the switch to enable graphics acceleration and was able to manipulate large 3D models in real-time over the large wall. Although this showed the positive effect of Chromium I was interested in a more precise speed-up factor. Therefore, I started an Internet search for applicable benchmarking programs that could be run. SPEC (Standard Performance Evaluation Corporation) offers a comprehensive OpenGL graphics test suite called Viewperf [8] which was found to be a suitable way of measuring the performance difference. The free 430MB package was downloaded, extracted and compiled while reading the documentation. Isolating the graphics subsystem from other system components while testing all kinds of graphics capabilities, Viewperf promised to give a good comparison of system performance with and without Chromium. First I ran the hour-long test suite with a window size of 1280x1024 in a plain X session just using the control screen. After that two different window sizes of 1280x1024 and 3840x1024 were used running it on the DMX cluster with and without Chromium enabled. Table 1 shows that while there is a significant drop of performance enabling DMX, Chromium increases the average number of frames drawn per second noticeably which for some cases is close to the results from the smaller window size using plain X.

	<i>Plain X</i>	<i>X+DMX</i>	<i>X+DMX+Cr</i>
light @ 1280x1024	1.775	0.5181	1.0760
light @ 3840x1024	N/A	0.5155	0.8111
maya @ 1280x1024	2.287	0.6809	1.7440
maya @ 3840x1024	N/A	0.6847	2.0340

Table 1: Mean of Frames per Second of 2 SPECViewperf tests (light and maya) with different window sizes and modes (Hardware: 3x P4 Celeron 2.80GHz, Nvidia Quadro4 380 128MB)

2.4. X2X-mod

Experienced Access Grid operators are often asked by new operators to help at meetings or sessions in other Access Grid nodes. This involves moving video windows to sensible places on the desktop, audio adjustments and many other things that are not directly visible by the audience. Often, only little interfering is necessary from the helping operator and it would be convenient if he could assist using his own laptop and peripherals from inside the room, without having to walk over to the control machine.

The client-server architecture of the X window system makes it generally possible to connect and interfere with a remote X server. But a program was needed that allows for a quick and reliable way of interacting with a remote display. Although VNC (Virtual Network Computing) [9] is a powerful and well-established application that enables users to see the desktop of a remote machine and interact with it using the own peripherals it is inappropriate for the aimed purpose. Not only do both parties need to run the VNC software (client and server respectively) but the actual use generates significant network traffic while making it hard to perform the work when the system to be controlled has a large resolution (as is the case here). Since both parties sit in the same room and see the display only mouse and keyboard input have to be controllable.

After an Internet search and tests with several applications, x2x-mp [10] was found. It was developed at Princeton University using DECs x2x as a basis. Once the custom window manager is started on a remote computer users can connect and interact with it using the x2x-mp program and their own mouse. As a result, each user has his own mouse cursor that he can move independent from the others. To make those cursors distinguishable different colors are used and since there may be several focused windows now (one for each user) their border color reflect the cursor color to make clear who has the focus.

Compilation of the window manager and the x2x-mp application were straight forward and a test using two computers worked. However, similar to the VNC case there are drawbacks with this approach for the purpose intended. Having to install a custom window manager on the destination machine is inconvenient and not user friendly due to the fact that it is based on *wm2* - an unconfigurable very basic window manager with little features. My aim was to gain knowledge from the code of x2x-mp and its base x2x to create my own application that can use multiple cursors (as x2x-mp can) without depending on a special window manager (as x2x). Since x2x works for Linux and Windows (using Cygwin) the source code (one file) is long and hard to read containing preprocessor conditions all over the file to disable the Cygwin code when compiled for Linux and vice versa. I decided to clean up the code first and leave only the part for Linux to be able to go through and understand how it works. Having done that, I went through the functions and data structures trying to follow what happens when the program is run. I referred to the Xlib manual [11] quite often as I had not done much X programming before. Changing little details and adding custom code helped me to understand the purpose of every function eventually. There are two ways of triggering a connection to the destination. When run with a direction parameter (-north, -south, -east or -west) the program runs invisibly and a connection is established whenever the mouse moves to the edge given by the direction. For example if -east was specified then moving the mouse to the right edge of the desktop connects to the destination computer and all further mouse and keyboard input affect only the destination until the cursor moves back across the border. This is achieved by creating an invisible window on the source machine at the right edge whose motion callback function initiates the connection. In the second mode x2x displays a little window containing only a string and the connection is established by left clicking into the window while a disconnection is achieved by clicking both buttons at once. Having understood the way x2x works I did the same with x2x-mod to see how it manages to display multiple cursors. Fortunately, the code is very similar and most of the differences are additions for the multi pointer case. Reading the code revealed that the additional cursors are actually *windows* which look like cursors using the X11 shape extension. That was no surprise as I was dealing with a desktop application and the underlying X server currently supports only one cursor. The main reason x2x-mp relied on a modified window manager was to make it possible to color window borders as mentioned above. This feature was not important in my case and I was confident that I could remove the dependency for a special window manager while still allowing multiple cursors on the screen.

Taking x2x as the starting point for my new project *x2x-mod* I removed features that were not needed and simplified a major part of the rest of the code by creating new functions and making adaptations. The implementation of the second cursor was a greater challenge. I started by trying to display a simple window on the destination desktop once the connection was established. After being connected both, keyboard and mouse input are intercepted by the program using the *XGrabPointer* and *XGrabKeyboard* functions. Every mouse and keyboard event can therefore be processed by callback functions which already exist from x2x. To make the new cursor window movable the mouse movement callback was changed accordingly. I wanted to leave the option to take control of the cursor itself instead of adding a second cursor which is why several additions were made to the code to act accordingly. Now that moving the new cursor window with the remote mouse worked the task was to make the buttons work. This was a bit trickier but the x2x-mp source gave me some hints on how to do it. Because the mouse does not move the *real* cursor but only a window, simply forwarding mouse clicks would not work as this would result in a click at the position of the real cursor. Instead, whenever the user presses a mouse button the real pointer has to be moved to the location of the cursor window before sending the button event to the X server. The *XTest* extension is of great help containing functions such as *XTestFakeMotionEvent* and *XTestFakeButtonEvent*. Problems occurred at first because with the real cursor being located at the same coordinates as the fake cursor, the cursor window received the click event. As a result I made the cursor window move some pixels to the side before sending the button press and return to the original spot after. Similarly the real cursor had to be moved back to where it was to avoid confusion of the user. Double-clicks, right-clicks and drag events worked as expected. Unfortunately, no solution was found to make highlighting of menu items work using the fake mouse pointer without help from the window manager.

What was left to do was making the application more user-friendly. Similar to x2x and x2x-mp all settings (such as the destination server) were transient and had to be appended to command line options every time the program was started. Unlike cursor-applet it was not clear whether x2x-mod would be run in a Gnome environment or not and I did not want to add unnecessary library dependencies to the project. Therefore, GConf was out of question for a place to store the configuration data. Instead, I created a regular text file to hold the configuration similar to many other Linux programs. To prevent further growth of the x2x-mod C file a second file was introduced for all functions that are related to the configuration. This included saving and loading at first with new data structures and helper functions such as a parser which takes care of converting string values from the file to the

correct type (for example the string 'false' had to be translated to be a boolean etc.). Since the user would now have to edit the file manually to make changes a GUI was necessary that provides widgets to set the 13 possible options in an easy way. As with GConf it was not appropriate to use GTK so I looked for more general graphical toolkits. With the X server being a necessary dependency the most widely available toolkit is Xt (the X Toolkit Intrinsic) which is the basis for the Motif widget set [12].

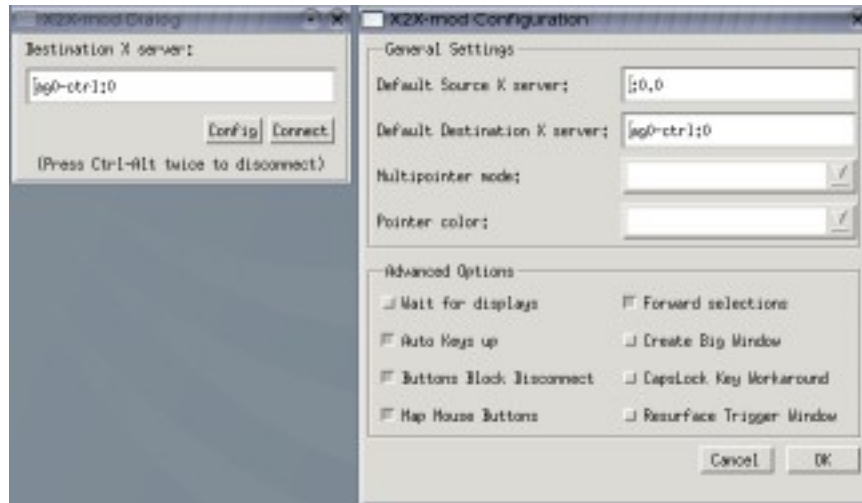


Figure 7: Main window (left) and configuration dialog (right) of x2x-mod

Being new to programming with Motif, I downloaded and read the *Motif Programmer's Guide* as well as the 2 volumes of the *Motif Programmer's reference*. After some coding I found that it is not hard to learn but care has to be taken that the hierarchy of widgets is created as intended. I managed to create a configuration dialog with Xt using several different widget types and added buttons on the x2x-mod main window to access this dialog or to connect (see Figure 7). Packaging the files after successful tests with remote computers, the task was complete.

2.5. Unifying the display & capture machines

Capturing video from at least one camera source and displaying video streams from remote nodes are both essential tasks that an Access Grid system has to accomplish throughout a session. Typically, each of those tasks is dealt with on a separate computer - a display machine and a capture machine. Not only does this increase the overall performance but also the usability. The latter is due to the fact that each capture device is driven by a service which itself needs an instance of the video tool *vic*. Additionally, an instance of *vic*

is running on the display side which displays both, local and remote streams. As a result a one-machine set-up is usually cluttered by several vic windows of which only one is really needed. Nevertheless, the cost benefits of using one machine makes that option still attractive. Modern computers are powerful enough to manage capturing and displaying at the same time. But it is important that both tasks get their share of memory and CPU time no matter what the other one is currently doing. Also, a solution is needed for the usability issue. I was asked to look into virtualization techniques and whether they could solve this problem.

Virtualization allows running multiple virtual machines at the same time that share hardware resources of one physical computer. The *host* system must provide a robust and transparent way to make the *guests* function as if they are running on separate machines. Until now this was a challenging task as there was no hardware support to ease controlling of resources. This is changing with the new processor generation having built-in support for virtualization. But there are several Virtual Machine Monitors on the market that work quite well using current processors, with Xen [13] being very interesting as it is a mature open source implementation that offers very high performance.

Reading the on line documentation I found out what was involved in setting up a Xen system and how the components work together. Such a system has an additional software layer called the *hypervisor* that is loaded directly after booting and only then starts the operating system. To achieve high performance without too much overhead despite not having hardware support, the operating system (or its kernel) has to be modified. This type of virtualization is called *paravirtualization*. Closed-source systems like Microsoft Windows do not work with Xen as they cannot be modified. In contrast, the Xen package contains the necessary patches to make FreeBSD, Linux, NetBSD and Plan9 work as virtual machines. Since the current systems at VisLab are Linux based this was good news. Besides the hypervisor, a Xen system consists of several guests (also called *domains*) with one of them being a "Privileged Guest" (called *dom0*). This *dom0* is loaded by the hypervisor and is the main system on the machine as it has full hardware access (in particular the graphics card). Also, all other guests (the *domU*'s) are started and shutdown from this system.

I was given a clean system to explore the capabilities of Xen and see what is involved in setting it up. After extracting the package I issued 'make world' to build the hypervisor and the necessary tools. The command also downloads the source package of the Linux kernel version it is compatible with (2.6.12 at that time) before extracting and patching it to work with Xen. In the default configuration two kernel trees are generated, one for the *dom0* and

another for an unprivileged guest system. As a result, not only did I have to wait for the hypervisor to be built but additionally for two complete kernel builds. Because I had to make sure that the necessary drivers had been compiled into the kernels I started the configuration for dom0 and domU. I noticed that while some categories were not available compared to the default Linux kernel there was a new Xen-specific category. The two main options inside this category are used to switch between a privileged and unprivileged kernel and allow physical device access respectively. While domU was already configured with sensible defaults (unprivileged kernel, no hardware drivers etc.) changes were necessary for dom0 to reflect the current kernel configuration. It was not as easy as I first thought because the kernel versions differed so I had to go through the options manually and compare them. After rebuilding and installing the kernels the grub boot menu was updated to contain an entry for the Xen system. Besides the kernel name, the amount of memory to be used by dom0 was given as an additional option. Since all Xen domains share the real memory that is available this option is needed to let the hypervisor know how much memory to reserve for dom0.

I booted with the new kernel and with the exception of the initial hypervisor output no difference was noticeable. Part of this output was a warning that the TLS libraries should be made inaccessible (by renaming the folder `/lib/tls`) due to their incompatibility with Xen which causes a major decrease in performance. I renamed the folder as suggested before rebooting. The "communication" with the hypervisor is done using a daemon (*xend*) and Python based tools. After starting the daemon I was able to use the tools to see a list of running domains (currently only dom0) as well as information about system resources and also retrieve the initial hypervisor output for inspection. Being an independent virtual machine each guest domain needs to be assigned virtual hardware resources like memory, disk space, network adapters and also a name to distinguish between the guests. Therefore, for each domU a separate configuration file in Python format needs to be created containing this information. Having a compiled kernel, the domU only lacked disk space with an installed Linux system to run. I knew from reading the documentation that I could either use a (real) separate partition or create a file with a file system (a *virtual disk* similar to a VMWare disk file). Although creating this file is easy with Linux (using the *dd* and *mkfs* commands) I soon discovered that installing a Linux distribution after mounting it is not that straight forward. The Xen homepage contains links to bootstrap scripts for several distributions that make it possible to install into a folder other than the root folder. However, with Slackware not included in the list the only way was to recursively copy the contents of the current system into the virtual file system or create and use a separate

partition. Although the latter was a more reasonable solution I decided to go the way of copying to see what was involved in using a file. The file was mounted with the *loop* option similar to a CD drive and a copy command duplicated the current system inside the virtual file system. According to the Xen documentation the file can now be used simply by specifying its path in the domain configuration file. Attempting to do that, I did not succeed and was confronted with an error. Digging into the Python script that is responsible for loading the file I followed the steps manually that it was supposed to do and managed to run the guest using the file. Since Linux can have several bound loop devices the script is basically probing all loop devices (*/dev/loop0*, */dev/loop1*, etc.) binding the first free one it finds to the file specified in the configuration using *losetup*. Before changing the guest to use a real partition I used the working system to solve some issues I encountered. First of all, several init-scripts put error messages on the screen which I simply disabled because they were related to hardware that was not available or not accessible in the virtual environment. One exception to that was the network script. Currently set to use DHCP and the same host name as dom0 I had to change both and give it a static IP until the DHCP server was told about the new MAC address (which is a custom address specified in the configuration file). Next, I deleted the contents of the */boot* and */lib/modules* folders or in other words the kernel and its modules. This is possible because the kernel for the guest system is loaded by the Xen scripts and is therefore located on the dom0 drive. The corresponding modules had to be transferred over to be usable by the guest though. While the system mainly worked now the guest's X server was not able to start up. Instead, a connection to the console can be requested using the Xen tools. As the intention was to use domU as the capture machine, I realized that a non functioning X server would be a problem because it has to run instances of vic which require X. Browsing through the Xen forums revealed that there is no support for framebuffer for guest domains at the moment and consequently the X server will not work. Reporting this major drawback to my supervisor he mentioned an alternative to vic that does not require X. Searching for such a tool led to Open Mash [14] a streaming media toolkit that apparently included a tool that I could use as a replacement for vic. It took me a while to understand the folder structure of this Tcl/Tk suite not to mention its compilation, partly because the latest version is over 2 years old and partly because it is not well-documented. I found two folders (*hvic* and *rvic*) with which I experimented. None of the tools worked straight away and I spent hours until I found a way to make *rvic* run. It is invoked with several command line options and also needs a configuration file. Again, no explanation was found about the many options and while trying different methods a

colleague suggested looking for a virtual X server he had heard about instead. It would act as a replacement for the real X server without requiring hardware access or a framebuffer device and therefore make vic work. I quickly discovered that the tool he mentioned is Xvfb (X virtual framebuffer) which conveniently ships with the "real" X server. After reading the manual page I started Xvfb with appropriate options. Setting the *DISPLAY* environment variable made vic run on the guest system.

Obviously the GUI cannot be seen in this way but it is not needed for the capture machine whose task is merely to stream the video data captured by the cameras through the network. This led to the next problem to be solved, namely the unprivileged guest had to be able to access the video device. Since a device can only be used exclusively by one domain, the hypervisor can be configured to hide specific PCI devices from dom0 through the kernel command line. I determined the PCI bus and device number of the capture hardware and modified the grub menu file accordingly before rebooting. After confirming that the device was no longer available from dom0 I changed the configuration file of the guest to include the aforementioned device. Then, enabling the 'physical device access' option and the appropriate drivers in the kernel configuration the guest kernel was rebuilt. However, with these changes the guest did not start up anymore. I tracked the error down to the physical device access kernel option. Whenever it was enabled the guest would not boot. A search in the forums did not help and although it appeared illogical, additionally enabling the 'Privileged Guest (domain 0)' option in the kernel configuration solved the problem. This fact made unclear why there were two separate options if both had to be either enabled or disabled at the same time but I did not find an answer to this question. With the changes applied and Xvfb running, vic was started in the guest domain and the video streams were successfully received by dom0. Next I tried to achieve the same using a real partition instead of a file and it all turned out to be straight forward. A final issue was left to find out before writing a step-by-step guide for other Access Grid users. In the current configuration the guest system has one virtual network interface with its own IP address to connect to the network of dom0. This works because the Xen network scripts create a bridge and bind the appropriate interfaces to it whenever domU is started. Similarly, the bridge is destroyed as soon as the guest is shut down. The question was raised whether Xen can be set up to use NAT (Network Address Translation) instead which would allow the guest to use the same IP address as dom0. The network scripts mentioned earlier include an alternative file for a network setup using routing and Xen's main configuration file contains the name of the script to execute. However, I had to write a modified file to achieve the intended NAT setup using the routing script as a template.

The bridge is still needed in this case but it is configured with a local IP address (192.168.0.1). A subsequent *iptables* reconfiguration made the script work as intended. I included this script in the step-by-step guide for users who prefer using NAT. My supervisor used the guide to change another Access Grid computer to a Xen system and after some modifications it was put on the VisLab website which concluded my work on Xen.

2.6. Echo canceler setup under Linux

Unlike personal video-conferencing setups where it is sufficient to have a headset to talk and listen an Access Grid node necessarily has speakers and several microphones that can be located anywhere in the room. Therefore, it is necessary (and even a minimum requirement for a node [18]) to have an echo canceler device. VisLab as well as many other nodes use the powerful ClearOne XAP400 device for this purpose. Other well-known models from the same company include the XAP800 and the older AP400. They all come with a RS232 interface and are completely configured through this interface by a set of serial commands. ClearOne offers a free Windows software called G-Ware (or AP-Ware for the older models) that comes with a mouse-driven GUI and eases the configuration and maintenance of the device.

However, with the Access Grid software being available for many platforms this meant that non-Windows users had to have a Windows system (and license) just to be able to configure their echo canceler. The VisLab node is completely controlled by Linux machines and the problem was "solved" by using the virtualization software VMWare Workstation [15]. This was a usable short-term solution but had several drawbacks. Besides the license fees for both, Windows and VMWare, it involved setting up a Windows installation as a virtual machine and each time the tool was needed this machine had to be started up which caused delays of several minutes. More importantly, changes are usually necessary while the Access Grid software is running. But both programs being very resource intensive this was not possible without making the system unresponsive. I was asked to look into the possibility of running G-Ware under Linux with the help of Wine [16]. Wine is the product of many developers around the world and aims at offering a free open-source reimplementation of the Windows API that is fully compatible with Microsoft's commercial counterpart. The project is over 12 years old and just made the transition from an alpha to a beta stage when I started working on it. Although this is a

surprise to most people at first, it is mostly due to the rapidly changing and evolving original API of Windows and additionally due to the poor or wrong documentation of the same. There are many advantages of using Wine over VMWare or a separate Windows installation. Besides the cost benefits (no Windows license is needed) a Windows program running under Linux using Wine looks and behaves like any other Linux program and is not slower than under Windows because no significant overhead is generated.

The Wine homepage includes a compatibility page which lists many Windows programs with comments on how well they work with Wine. A search on the page brought no hits for G-Ware (which was less surprising) so I started downloading the latest Wine source code via CVS to see for myself. Having had no experience with Wine yet I used the time it took to compile to read the documentation that is available on the Wine homepage. It took over 30 minutes to complete building and installing and the last step was to download the G-Ware installation files from ClearOne. Running the Windows program is as simple as typing "wine setup.exe" and worked quite well besides some font issues. But starting the application itself resulted in not more than a page fault exception at first. I looked for configuration options that might help using *winecfg*. Since Wine tries to be compatible with every Windows version, it is possible to select a specific version it should act as in the configuration. But no matter which setting I chose the result was the same exception. The next step was to analyze the debug output to get some more information about the cause of the fault. It seemed that the crash involved the OLE (Object Linking and Embedding) libraries which form (together with DCOM, the Distributed Component Object Model) a fundamental framework used by Windows to be able to embed an object created by one application into another. An example is to embed an Excel sheet in a Word document. After an Internet search it became clear that one of the two most important technologies that Wine developers were currently working on was OLE with the other being MSI (the Microsoft Installer). To date the implementation was not mature enough to be usable by many programs and a temporary solution was to use native Windows OLE libraries which are available from the Microsoft homepage. More specifically, Wine had to be configured to simulate Windows 98 behavior and use native ole32, oleaut32 and rpcrt4 libraries. This demonstrates a crucial capability of Wine: built-in libraries can be easily exchanged by their native counterparts with the exception of the ones that interact directly with the hardware or device drivers of course.

Using the native libraries I succeeded in getting the G-Ware GUI to start up (see Figure 8) and all dialogs seemed to behave as expected. Only the text was misaligned which was a font issue I would be looking into later. More importantly, establishing a connection to the

echo canceler device (connected through the serial port) had to be tested. Initiating the connection brought up several "unit errors" with an error number in a dialog.

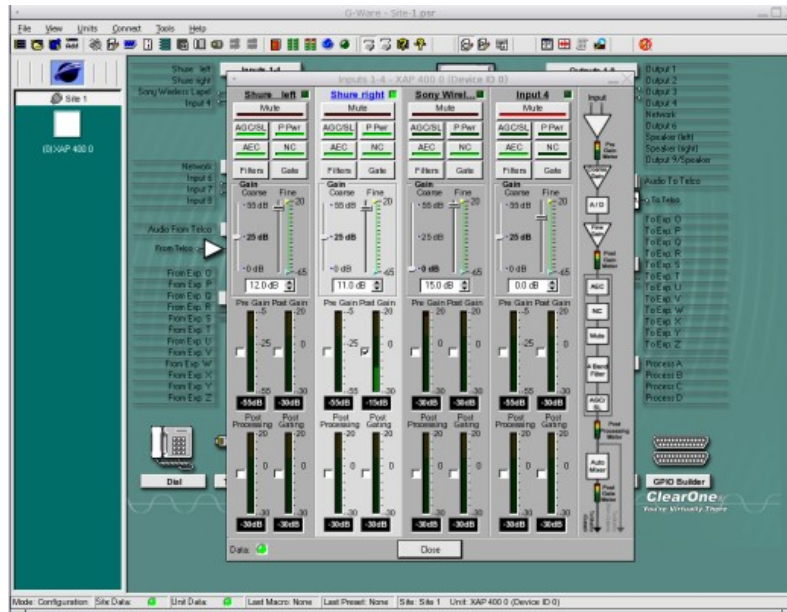


Figure 8: The G-Ware GUI running with Wine under Linux

At the same time Wine filled the console with "fixme" messages until the whole application crashed with a final error that read "pipe: too many open files". With the help of an Internet search I created the following little shell script "fdcount" to investigate the error message:

```
ps augwx | grep wine | grep -v grep | perl -e 'while (<STDIN>)
{ @x=split(" "); system("ls /proc/$x[1]/fd") if !$seen{$x[4]}++;}' | wc
-l
```

When being run it outputs the number of used file descriptors by all processes that belong to Wine. In a separate console I ran "watch -n 1 fdcount" to see how the number of open files evolved while starting G-Ware and connecting to the device. And the number clearly rose until the limit of the system was reached and the program crashed. At this point I started to dig into the Wine source code and after some time found the bug that was responsible. A file handle leaked in the serial communication code which only happened in one branch of an if-statement. I inserted the code to release the handle, built the corresponding library (kernel32.dll) and started G-Ware again. The unit errors still occurred as did the Wine messages but the program was stable and did not crash anymore. Having the XAP400 manual, I looked for an explanation of unit error codes. One error was described as a "bad checksum" whereas the other occurring error meant "command too big". I read through the complete chapter about the serial protocol that is used by the device to get an understanding of what data is transferred between program and unit.

Enabling the "comm" debug channel of Wine revealed that upon connection the program requested all information that is stored in the device. Inspecting the results showed that most data had been transferred correctly but some information was missing. This was consistent between different runs which was an important finding. If there was a way to monitor what data is actually sent and received, a comparison could be made between Windows and Linux. For the Windows side the company Sysinternals [19] offers a free monitoring tool named "PortMon" which was used with the still existing VMWare installation to get a log output of the traffic during connection. I knew from studying the manual that the echo canceler supports two communication modes, text and binary. When in text mode both, commands and responses are legible ASCII strings and a command has a documented structure. Unfortunately, the PortMon output made clear that after handshaking with the device G-Ware switched to binary mode and all following commands are binary encoded strings which are not documented. Nevertheless it would help to see whether a log obtained on the Linux side would be different. After trying different debug channels it seemed there was no built-in way of logging data sent through the serial port. To be able to get advice from other Wine developers and submit patches I subscribed to the wine-devel and wine-cvs mailing lists. In a first post I reported about the leaking file handle and asked for a way to see the data that is transferred. Uwe Bonnes, a Wine developer and the author of large parts of the serial communication code verified that my findings were correct and the patch was committed to the Wine tree. He also gave me instructions on how to make Wine output data written to files (the serial port is a file under Linux).

After obtaining and cleaning up the log file it was possible to look at both versions side by side. Unfortunately, the format of both was quite different and the output from PortMon was truncated so a simple diff did not work. A quick look through the outputs showed differences in that apparently more bytes were accumulated under Linux before they were sent out compared to the Windows case. Furthermore, it became clear that the binary commands had a length of multiples of 8. Where strings of 8-96 bytes were transmitted at once under Windows, Linux seemed to wait until at least 32 bytes are accumulated and after a while the length did not drop below 200 bytes. This looked suspicious and looking a bit further I found another difference. The even numbers suddenly turned into odd numbers like 193 or 33. It took several hours analyzing the hundreds of lines without losing track of which byte on the Linux side is found where on the Windows side. But eventually the reason for the odd numbers was found. Every occurrence of the byte 0xFF in the Windows stream seemed to be replaced by 0xFFFF in the Linux stream.

Coincidentally, I rebooted the computer and noticed that this was no more the case and many (but not all) of the errors I received before disappeared. My guess was that the serial port was in a different state before the reboot that caused the byte doubling. While investigating I started VMWare to get information about the port settings under Windows whereupon the problem reappeared. Obviously VMWare changed the port settings without resetting them after quitting. Nevertheless Wine was probably to blame for not setting the state of the serial port properly. I used the command *setserial* to see if it would give a hint on the cause. Unfortunately, the output was not detailed enough to contain the port's flags. Not knowing of other tools to get the information I decided to download the source code of the terminal emulator *minicom* which I could use to communicate with the echo canceler device manually and modify to show the flags of the port. After compiling and configuring *minicom* I was able to send commands to the device and receive a response. Next, I ran G-Ware and confirmed that *minicom* did set the port parameters back to a state where the 0xFF bytes are not duplicated. Taking a closer look at the initialization code I found the lines that are responsible for setting the port flags. After adding a debug output that displays the flags before and after initialization I was able to determine the flag that caused the duplication. It was the PARMRK flag and the description revealed that its purpose is to mark parity errors in the data stream by the hex sequence "FF 00 xx" where xx is the character which was received in error. But for this to work valid 0xFF characters have to be specially marked which is the reason it is duplicated.

With this finding I prepared a patch for Wine that disables this flag when initializing the port and sent it together with an explanation to the mailing list. As noted earlier this did not solve all problems and the remaining errors cost me several days of changing code in the serial communication functions in Wine to find the reason. I learned that while many Windows serial port functions had a more or less similar counterpart under Linux there is especially one function that needed to be emulated in a complicated way, namely *WaitCommEvent*. Its purpose is to inform a client application when a specific serial port event has occurred. Such an event is for example the arrival of data which waits to be read by the application. Not having a Linux counterpart, Wine creates a thread when this function is called and regularly polls the state of the port for events only returning if the requested event was detected. From the debug output of a G-Ware run I could see that after each output to the serial port the program called *WaitCommEvent* to wait for response by the device. But interestingly that function is also called once in the beginning which led to my assumption that since there are several threads running that poll the port status at the same time, they might interfere and cause the problems. To find out if that was true I

changed the code to poll the port only with one thread at a time making subsequent threads wait for previous ones to finish. After some G-Ware runs and modifications to my code the accumulation of data noted earlier seemed to be worse and the number of errors was increased. That led me to the function `PurgeComm` which is used to flush the buffers and delete characters pending to be sent or received. It was programmed to use the Linux `tcflush` function which does the same thing according to the documentation of both. Nevertheless I exchanged the calls to `tcflush` by `tcdrain` whose only difference is that pending characters are not discarded but sent/received and the function returns only then. A subsequent G-Ware run worked perfectly. I received no errors and all data was correctly transmitted. After reverting the changes I made to the threading part and confirming that G-Ware still runs without error, the Wine developers were informed by the mailing list and asked for comments. The decision was not to apply the change upstream before confirming with other programs that `tcflush` is wrong. Therefore, I created a patch with the intention to provide it to other interested users including a guide on how to set up Wine and G-Ware under Linux.

I finished off by copying the missing fonts into the right folder to fix the text misalignment problem. All steps were written down in detail in a step-by-step guide which was put on the VisLab website. Since Wine is developing rapidly I kept following the mailing list until the end of my time at VisLab to see if any amendments were necessary to the guide (for example because the native libraries are no longer needed).

2.7. “Shared PowerPoint” under Linux

Having success with Wine and G-Ware (described in section 2.6) my final task was to try and repeat the success with Microsoft PowerPoint. More specifically, the "Shared Presentation" extension included with the Access Grid software and explained in section 1.2 uses Open Office's *Impress* under Linux¹ as PowerPoint is not available for this system. Although Impress has a high compatibility with files generated in PowerPoint, it is inevitable that some files are not rendered exactly as in PowerPoint which may be disturbing in a shared presentation session where some participants are using Windows and others Linux. For this reason it would be a helpful feature to give the user the option of running PowerPoint (with the help of Wine) under Linux as the back-end of the Shared Presentation.

¹ In this context, Linux is used as the term for Linux-like systems (such as FreeBSD)

Being the latest Office suite from Microsoft, Office 2003 was the first version I tried to run with Wine. The simple way of executing "wine setuppro.exe" did not work but I knew that these executables are mainly wrappers that start the Microsoft Installer (MSI) with correct parameters. The directory listing revealed that the installer script is called "PRO11.MSI" and the alternative way of installing it is therefore by issuing "wine msiexec PRO11.MSI". *msiexec* is the executable of the Microsoft Installer and is still under major development in the Wine project. Nevertheless, the invocation succeeded and the installer dialog appeared on the screen. After clicking through and choosing to install PowerPoint the installation went ahead and copied files but failed before being finished. After analyzing the log output and trying to understand the relevant Wine code it seemed that Wine's version of the installer lacked features that were needed by the install script. Returning an error code made the script halt prematurely without finishing the installation. I changed the return code of the appropriate function to always return "success" to see if that would install all necessary files. The result was that the installer went ahead and apparently "finished successfully". Invoking PowerPoint worked to my surprise but only for a few seconds before I was confronted with a dialog telling me that "This application is not installed for the current user". Although it was clear that the error was related to the installation being forced to finish I was not sure where to look for a solution. My only clue was that obviously all files had been installed properly so that the reason for the error must be related to missing registry entries. Inspecting Wine's trace outputs and comparing the relevant registry entries with those under Windows (installed in a VMWare virtual machine) eventually led me to the missing entries which I added to the Wine registry to solve the problem. PowerPoint seemed to work flawlessly until I discovered a problem with certain files I was trying to open. After loading them PowerPoint deadlocked but I could not find what the non-working files had in common at first and decided to try my luck with Office 2000 before spending too much time on this issue. It was surprisingly easy to install this version without encountering problems. Two native libraries had to be downloaded and copied into the Wine folder though. Unfortunately, I soon discovered the same problem that I encountered with PowerPoint 2003. Some files caused a deadlock and this time I spent several days to find the reason and a solution. Previews that are saved by default with a PowerPoint file were causing the issue and traces led me to Wine's *bitblt* (bit-block transfer) functions for Metafiles. Making Wine use an alternative function solved the problem and I had a fully functional PowerPoint 2000 application running under Linux.

The next step was to write a Windows program that is able to remotely control

PowerPoint. Basically, I had to duplicate the functionality that the Windows version of the Shared Presentation application includes. It uses OLE (Object Linking and Embedding, described in section 2.6) to instantiate and control PowerPoint programmatically. But since there is no direct way of using OLE from a Linux application that was the tricky part and after some thinking the only possible way turned out to be the transfer of messages through a network socket. A (Wine driven) Windows program would start PowerPoint using OLE and listen for commands which arrive from the Linux part of the Shared Presentation program. These commands could then be translated to OLE methods and sent to PowerPoint.

Despite having VMWare on a separate machine I installed QEMU [17] to gain experience using this free alternative with similar capabilities. Creating a virtual machine was straight forward and although it took half a day, I had a working Windows environment including Microsoft Visual Studio and Office 2000 in the end. Next, I had to learn about OLE and socket programming to achieve my task. After initial tests where I successfully started PowerPoint and invoked methods using the OLE interface I created a new C++ project (PPControl) and implemented the main loop that opens a socket and waits for commands to arrive. The Shared Presentation provided me with the commands that needed to be implemented. Table 2 lists the command strings that PPControl understands and their description. For convenience I chose strings with a fixed length of 4. This made it easy to extract the argument that some commands require and which is simply appended to the command string. For example loading slide 5 would be achieved by sending the string "goto5" to PPControl via the socket.

<i>Command</i>	<i>Description</i>
load	Load a PowerPoint file (argument: file name)
next	Advance to the next slide
prev	Go to previous slide
goto	Go to a specific slide number (argument: requested number)
slds	Return the number of slides in the current file
step	Return the animation step number of the current slide
curr	Return the number of the currently active slide
lerr	Return the last error code (or zero if no error occurred)
stop	Stop the current slide show without exiting PowerPoint
quit	Stop the slide show and exit PowerPoint (exits PPControl as well)

Table 2: Valid commands for PPControl and their description

Knowing that Microsoft offers a free PowerPoint Viewer that allows viewing of PowerPoint slide shows without owning the Office suite I also examined whether it is a usable alternative to PowerPoint 2000 for my purpose. Unlike the newer PowerPoint Viewer 2003, its predecessor offers OLE support and works well with Wine. While designing the C++ classes I took this into account and created two separate classes (one for PowerPoint, one for the Viewer) and made them derive from the same abstract base class. That way PPControl can probe whether PowerPoint is installed and use the PowerPoint Viewer if that is not the case (using Polymorphism).

Now that the Windows part was working I had to write the Linux counterpart by changing the Python code of the Shared Presentation and adding another module. Since there was no GUI element to choose between different presentation back-ends (Impress or PowerPoint) the code was changed to check for a working PPControl application in the path and only use Impress as a fall back. That way, installing Wine and PPControl would automatically activate its use by the Shared Presentation. The Python documentation revealed that it was relatively easy to use sockets and send or receive data compared to the C++ case under Windows. After implementing the counterpart to all commands I started a test run and successfully controlled PowerPoint from the Shared Presentation dialog. I repackaged the Shared Application and documented the steps necessary to set it up (including Wine) on a web page which concluded this task.

3. Conclusion

During my 6 months at VisLab I gained significant knowledge in various areas of computer science. At the same time I was able to use techniques I had learned from my studies to solve the tasks I was given. Having had mainly Windows programming skills I acquired invaluable experience in all aspects of Linux use and programming. Xt, Motif, GTK and GConf are only some of the Application Programming Interfaces that I learned to master from scratch at VisLab. Similarly, I started with no Python and Tcl/Tk knowledge yet I am able to understand and write applications using these languages now. For the first time I was able to contribute to open source communities (Wine, Access Grid) and due to many interesting tasks that are left to solve in Wine and other open source software I will continue to help.

Finally, I would like to acknowledge the help of my supervisors Dr. Nicole Bordes, Prof.

Bernard Pailthorpe and Chris Willing who assisted me with questions and difficulties I encountered.

4. References

- [1] <http://www.vislab.com.au>
- [2] <http://www.vislab.uq.edu.au>
- [3] <http://www.accessgrid.org>
- [4] <http://www.vislab.uq.edu.au/research/accessgrid/equipment.html>
- [5] <http://dmx.sourceforge.net/projects/dmx>
- [6] <http://www.gnu.org/software/autoconf>
- [7] <http://chromium.sourceforge.net/doc/index.html>
- [8] <http://www.spec.org/gpc/opc.static/vp81info.html>
- [9] <http://www.realvnc.com>
- [10] <http://www.cs.princeton.edu/~gwallace/multicursor/multicursor.html>
- [11] <http://tronche.com/gui/x/xlib>
- [12] <http://www.opengroup.org/openmotif>
- [13] <http://www.cl.cam.ac.uk/Research/SRG/netos/xen>
- [14] <http://www.openmash.org>
- [15] <http://www.vmware.com>
- [16] <http://www.winehq.org>
- [17] <http://fabrice.bellard.free.fr/qemu>
- [18] <http://www.accessgrid.org/agdp/guide/min-req/1.1.2/c44.html>
- [19] <http://www.sysinternals.com>