Distributed Multihead X design

Kevin E. Martin, David H. Dawes, and Rickard E. Faith

29 June 2004 (created 25 July 2001)

Abstract

This document covers the motivation, background, design, and implementation of the distributed multihead X (DMX) system. It is a living document and describes the current design and implementation details of the DMX system. As the project progresses, this document will be continually updated to reflect the changes in the code and/or design. Copyright 2001 by VA Linux Systems, Inc., Fremont, California. Copyright 2001-2004 by Red Hat, Inc., Raleigh, North Carolina

1. Introduction

1.1 The Distributed Multihead X Server

Current Open Source multihead solutions are limited to a single physical machine. A single X server controls multiple display devices, which can be arranged as independent heads or unified into a single desktop (with Xinerama). These solutions are limited to the number of physical devices that can co-exist in a single machine (e.g., due to the number of AGP/PCI slots available for graphics cards). Thus, large tiled displays are not currently possible. The work described in this paper will eliminate the requirement that the display devices reside in the same physical machine. This will be accomplished by developing a front-end proxy X server that will control multiple back-end X servers that make up the large display.

The overall structure of the distributed multihead X (DMX) project is as follows: A single front-end X server will act as a proxy to a set of back-end X servers, which handle all of the visible rendering. X clients will connect to the front-end server just as they normally would to a regular X server. The front-end server will present an abstracted view to the client of a single large display. This will ensure that all standard X clients will continue to operate without modification (limited, as always, by the visuals and extensions provided by the X server). Clients that are DMX-aware will be able to use an extension to obtain information about the back-end servers (e.g., for placement of pop-up windows, window alignments by the window manager, etc.).

The architecture of the DMX server is divided into two main sections: input (e.g., mouse and keyboard events) and output (e.g., rendering and windowing requests). Each of these are describe briefly below, and the rest of this design document will describe them in greater detail.

The DMX server can receive input from three general types of input devices: "local" devices that are physically attached to the machine on which DMX is running, "backend" devices that are physically attached to one or more of the back-end X servers (and that generate events via the X protocol stream from the backend), and "console" devices that can be abstracted from any non-back-end X server. Backend and console devices are treated differently because the pointer device on the back-end X server also controls the location of the hardware X cursor. Full support for XInput extension devices is provided.

Rendering requests will be accepted by the front-end server; however, rendering to visible windows will be broken down as needed and sent to the appropriate back-end server(s) via X11 library calls for actual rendering. The basic framework will follow a Xnest-style approach. GC state will be managed in the front-end server and sent to the appropriate back-end server(s) as required. Pixmap rendering will (at least initially) be handled by the front-end X server. Windowing requests (e.g., ordering, mapping, moving, etc.) will handled in the front-end server. If the request requires a visible change, the windowing operation will be translated into requests for the appropriate back-end server(s). Window state will be mirrored in the back-end server(s) as needed.

1.2 Layout of Paper

The next section describes the general development plan that was actually used for implementation. The final section discusses outstanding issues at the conclusion of development. The first appendix provides low-level technical detail that may be of interest to those intimately familiar with the X server architecture. The final appendix describes the four phases of development that were performed during the first two years of development.

The final year of work was divided into 9 tasks that are not described in specific sections of this document. The major tasks during that time were the enhancement of the reconfiguration ability added in Phase IV, addition of support for a dynamic number of back-end displays (instead of a hard-coded limit), and the support for back-end display and input removal and addition. This work is mentioned in this paper, but is not covered in detail.

2. Development plan

This section describes the development plan from approximately June 2001 through July 2003.

2.1 Bootstrap code

To allow for rapid development of the DMX server by multiple developers during the first development stage, the problem will be broken down into three tasks: the overall DMX framework, back-end rendering services and input device handling services. However, before the work begins on these tasks, a simple framework that each developer could use was implemented to bootstrap the development effort. This framework renders to a single back-end server and provides dummy input devices (i.e., the keyboard and mouse). The simple back-end rendering service was implemented using the shadow framebuffer support currently available in the XFree86 environment.

Using this bootstrapping framework, each developer has been able to work on each of the tasks listed above independently as follows: the framework will be extended to handle arbitrary back-end server configurations; the back-end rendering services will be transitioned to the more efficient Xnest-style implementation; and, an input device framework to handle various input devices via the input extension will be developed.

Status: The boot strap code is complete.

2.2 Input device handling

An X server (including the front-end X server) requires two core input devices -- a keyboard and a pointer (mouse). These core devices are handled and required by the core X11 protocol. Additional types of input devices may be attached and utilized via the XInput extension. These are usually referred to as ''XInput extension devices'',

There are some options as to how the front-end X server gets its core input devices:

1. Local Input. The physical input devices (e.g., keyboard and mouse) can be attached directly to the front-end X server. In this case, the keyboard and mouse on the machine running the front-end X server will be used. The front-end will have drivers to read the raw input from those devices and convert it into the required X input events (e.g., key press/release, pointer button press/release, pointer motion). The front-end keyboard driver will keep track of keyboard properties such as key and modifier mappings, autorepeat state, keyboard sound and led state. Similarly the front-end pointer driver will keep track if pointer properties such as the button mapping and movement acceleration parameters. With this option, input is handled fully in the front-end X server, and the back-end X servers are used in a display-only mode. This option was implemented and works for a limited number of Linux-specific devices. Adding additional local input devices for other architectures is expected to be relatively simple.

   The following options are available for implementing local input devices:

   1. The XFree86 X server has modular input drivers that could be adapted for this purpose. The mouse driver supports a wide range of mouse types and interfaces, as well as a range of Operating System platforms. The keyboard driver in XFree86 is not currently as modular as the mouse driver, but could be made so. The XFree86 X server also has a range of other input drivers for extended input devices such as tablets and touch screens. Unfortunately, the XFree86 drivers are generally complex, often simultaneously providing support for multiple devices across multiple architectures; and rely so heavily on XFree86-specific helper-functions, that this option was not pursued.

   2. The kdrive X server in XFree86 has built-in drivers that support PS/2 mice and keyboard under Linux. The mouse driver can indirectly handle other mouse types if the Linux utility gpm is used as to translate the native mouse protocol into PS/2 mouse format. These drivers could be adapted and built in to the front-end X server if this range of hardware and OS support is sufficient. While much simpler than the XFree86 drivers, the kdrive drivers were not used for the DMX implementation.

   3. Reimplementation of keyboard and mouse drivers from scratch for the DMX framework. Because keyboard and mouse drivers are relatively trivial to implement, this pathway was selected. Other drivers in the X source tree were referenced, and significant contributions from other drivers are noted in the DMX source code.

2. Backend Input. The front-end can make use of the core input devices attached to one or more of the back-end X servers. Core input events from multiple back-ends are merged into a single input event stream. This can work sanely when only a single set of input devices is used at any given time. The keyboard and pointer state will be handled in the front-end, with changes propagated to the back-end servers as needed. This option was implemented and works well. Because the core pointer on a back-end controls the hardware mouse on that back-end, core pointers cannot be treated as XInput extension devices. However, all back-end XInput extensions devices can be mapped to either DMX core or DMX XInput extension devices.

3. Console Input. The front-end server could create a console window that is displayed on an X server independent of the back-end X servers. This console window could display things like the physical screen layout, and the front-end could get its core input events from events delivered to the console window. This option was implemented and works well. To help the human navigate, window outlines are also displayed in the console window. Further, console windows can be used as either core or XInput extension devices.

4. Other options were initially explored, but they were all partial subsets of the options listed above and, hence, are irrelevant.

Although extended input devices are not specifically mentioned in the Distributed X requirements, the options above were all implemented so that XInput extension devices were supported.

The bootstrap code (Xdmx) had dummy input devices, and these are still supported in the final version. These do the necessary initialization to satisfy the X server's requirements for core pointer and keyboard devices, but no input events are ever generated.

Status: The input code is complete. Because of the complexity of the XFree86 input device drivers (and their heavy reliance on XFree86 infrastructure), separate low-level device drivers were implemented for Xdmx. The following kinds of drivers are supported (in general, the devices can be treated arbitrarily as "core" input devices or as XInput "extension" devices; and multiple instances of different kinds of devices can be simultaneously available):

1. A "dummy" device drive that never generates events.

2. "Local" input is from the low-level hardware on which the Xdmx binary is running. This is the only area where using the XFree86 driver infrastructure would have been helpful, and then only partially, since good support for generic USB devices does not yet exist in XFree86 (in any case, XFree86 and kdrive driver code was used where possible). Currently, the following local devices are supported under Linux (porting to other operating systems should be fairly straightforward):

    - Linux keyboard
    - Linux serial mouse (MS)
    - Linux PS/2 mouse
    - USB keyboard
    - USB mouse
    - USB generic device (e.g., joystick, gamepad, etc.)

3. "Backend" input is taken from one or more of the back-end displays. In this case, events are taken from the back-end X server and are converted to Xdmx events. Care must be taken so that the sprite moves properly on the display from which input is being taken.

4. "Console" input is taken from an X window that Xdmx creates on the operator's display (i.e., on the machine running the Xdmx binary). When the operator's mouse is inside the console window, then those events are converted to Xdmx events. Several special features are available: the console can display outlines of windows that are on the Xdmx display (to facilitate navigation), the cursor can be confined to the console, and a "fine" mode can be activated to allow very precise cursor positioning.

2.3 Output device handling

The output of the DMX system displays rendering and windowing requests across multiple screens. The screens are typically arranged in a grid such that together they represent a single large display.

The output section of the DMX code consists of two parts. The first is in the front-end proxy X server (Xdmx), which accepts client connections, manages the windows, and potentially renders primitives but does not actually display any of the drawing primitives. The second part is the back-end X server(s), which accept commands from the front-end server and display the results on their screens.

2.3.1 Initialization

The DMX front-end must first initialize its screens by connecting to each of the back-end X servers and collecting information about each of these screens. However, the information collected from the back-end X servers might be inconsistent. Handling these cases can be difficult and/or inefficient. For example, a two screen system has one back-end X server running at 16bpp while the second is running at 32bpp. Converting rendering requests (e.g., XPutImage() or XGetImage() requests) to the appropriate bit depth can be very time consuming. Analyzing these cases to determine how or even if it is possible to handle them is required. The current Xinerama code handles many of these cases (e.g., in PanoramiXConsolidate()) and will be used as a starting point. In general, the best solution is to use homogeneous X servers and display devices. Using back-end servers with the same depth is a requirement of the final DMX implementation.

Once this screen consolidation is finished, the relative position of each back-end X server's screen in the unified screen is initialized. A full-screen window is opened on each of the back-end X servers, and the cursor on each screen is turned off. The final DMX implementation can also make use of a partial-screen window, or multiple windows per back-end screen.

2.3.2 Handling rendering requests

After initialization, X applications connect to the front-end server. There are two possible implementations of how rendering and windowing requests are handled in the DMX system:

1. A shadow framebuffer is used in the front-end server as the render target. In this option, all protocol requests are completely handled in the front-end server. All state and resources are maintained in the front-end including a shadow copy of the entire framebuffer. The framebuffers attached to the back-end servers are updated by XPutImage() calls with data taken directly from the shadow framebuffer.

   This solution suffers from two main problems. First, it does not take advantage of any accelerated hardware available in the system. Second, the size of the XPutImage() calls can be quite large and thus will be limited by the bandwidth available.

   The initial DMX implementation used a shadow framebuffer by default.

2. Rendering requests are sent to each back-end server for handling (as is done in the Xnest server described above). In this option, certain protocol requests are handled in the front-end server and certain requests are repackaged and then sent to the back-end servers. The framebuffer is distributed across the multiple back-end servers. Rendering to the framebuffer is handled on each back-end and can take advantage of any acceleration available on the back-end servers' graphics display device. State is maintained both in the front and back-end servers.

   This solution suffers from two main drawbacks. First, protocol requests are sent to all back-end servers -- even those that will completely clip the rendering primitive -- which wastes bandwidth and processing time. Second, state is maintained both in the front- and back-end servers. These drawbacks are not as severe as in option 1 (above) and can either be overcome through optimizations or are acceptable. Therefore, this option will be used in the final implementation.

   The final DMX implementation defaults to this mechanism, but also supports the shadow framebuffer mechanism. Several optimizations were implemented to eliminate the drawbacks of the default mechanism. These optimizations are described the section below and in Phase II of the Development Results (see appendix).

Status: Both the shadow framebuffer and Xnest-style code is complete.

2.4 Optimizing DMX

Initially, the Xnest-style solution's performance will be measured and analyzed to determine where the performance bottlenecks exist. There are four main areas that will be addressed.

First, to obtain reasonable interactivity with the first development phase, XSync() was called after each protocol request. The XSync() function flushes any pending protocol requests. It then waits for the back-end to process the request and send a reply that the request has completed. This happens with each back-end server and performance greatly suffers. As a result of the way XSync() is called in the first development phase, the batching that the X11 library performs is effectively defeated. The XSync() call usage will be analyzed and optimized by batching calls and performing them at regular intervals, except where interactivity will suffer (e.g., on cursor movements).

Second, the initial Xnest-style solution described above sends the repackaged protocol requests to all back-end servers regardless of whether or not they would be completely clipped out. The requests that are trivially rejected on the back-end server wastes the limited bandwidth available. By tracking clipping changes in the DMX X server's windowing code (e.g., by opening, closing, moving or resizing windows), we can determine whether or not back-end windows are visible so that trivial tests in the front-end server's GC ops drawing functions can eliminate these unnecessary protocol requests.

Third, each protocol request will be analyzed to determine if it is possible to break the request into smaller pieces at display boundaries. The initial ones to be analyzed are put and get image requests since they will require the greatest bandwidth to transmit data between the front and back-end servers. Other protocol requests will be analyzed and those that will benefit from breaking them into smaller requests will be implemented.

Fourth, an extension is being considered that will allow font glyphs to be transferred from the front-end DMX X server to each back-end server. This extension will permit the front-end to handle all font requests and eliminate the requirement that all back-end X servers share the exact same fonts as the front-end server. We are investigating the feasibility of this extension during this development phase.

Other potential optimizations will be determined from the performance analysis.

Please note that in our initial design, we proposed optimizing BLT operations (e.g., XCopyArea() and window moves) by developing an extension that would allow individual back-end servers to directly copy pixel data to other back-end servers. This potential optimization was in response to the simple image movement implementation that required potentially many calls to GetImage() and PutImage(). However, the current Xinerama implementation handles these BLT operations differently. Instead of copying data to and from screens, they generate expose events -- just as happens in the case when a window is moved from off a screen to on screen. This approach saves the limited bandwidth available between front and back-end servers and is being standardized with Xinerama. It also eliminates the potential setup problems and security issues resulting from having each back-end server open connections to all other back-end servers. Therefore, we suggest accepting Xinerama's expose event solution.

Also note that the approach proposed in the second and third optimizations might cause backing store algorithms in the back-end to be defeated, so a DMX X server configuration flag will be added to disable these optimizations.

Status: The optimizations proposed above are complete. It was determined that the using the xfs font server was sufficient and creating a new mechanism to pass glyphs was redundant; therefore, the fourth optimization proposed above was not included in DMX.

2.5 DMX X extension support

The DMX X server keeps track of all the windowing information on the back-end X servers, but does not currently export this information to any client applications. An extension will be developed to pass the screen information and back-end window IDs to DMX-aware clients. These clients can then use this information to directly connect to and render to the back-end windows. Bypassing the DMX X server allows DMX-aware clients to break up complex rendering requests on their own and send them directly to the windows on the back-end server's screens. An example of a client that can make effective use of this extension is Chromium.

Status: The extension, as implemented, is fully documented in "Client-to-Server DMX Extension to the X Protocol". Future changes might be required based on feedback and other proposed enhancements to DMX. Currently, the following facilities are supported:

1. Screen information (clipping rectangle for each screen relative to the virtual screen)
2. Window information (window IDs and clipping information for each back-end window that corresponds to each DMX window)
3. Input device information (mappings from DMX device IDs to back-end device IDs)
4. Force window creation (so that a client can override the server-side lazy window creation optimization)
5. Reconfiguration (so that a client can request that a screen position be changed)
6. Addition and removal of back-end servers and back-end and console inputs.

2.6 Common X extension support

The XInput, XKeyboard and Shape extensions are commonly used extensions to the base X11 protocol. XInput allows multiple and non-standard input devices to be accessed simultaneously. These input devices can be connected to either the front-end or back-end servers. XKeyboard allows much better keyboard mappings control. Shape adds support for arbitrarily shaped windows and is used by various window managers. Nearly all potential back-end X servers make these extensions available, and support for each one will be added to the DMX system.

In addition to the extensions listed above, support for the X Rendering extension (Render) is being developed. Render adds digital image composition to the rendering model used by the X Window System. While this extension is still under development by Keith Packard of HP, support for the current version will be added to the DMX system.

Support for the XTest extension was added during the first development phase.

Status: The following extensions are supported and are discussed in more detail in Phase IV of the Development Results (see appendix): BIG-REQUESTS, DEC-XTRAP, DMX, DPMS, Extended-Visual-Information, GLX, LBX, RECORD, RENDER, SECURITY, SHAPE, SYNC, X-Resource, XC-APPGROUP, XC-MISC, XFree86-Bigfont, XINERAMA, XInputExtension, XKEYBOARD, and XTEST.

2.7 OpenGL support

OpenGL support using the Mesa code base exists in XFree86 release 4 and later. Currently, the direct rendering infrastructure (DRI) provides accelerated OpenGL support for local clients and unaccelerated OpenGL support (i.e., software rendering) is provided for non-local clients.

The single head OpenGL support in XFree86 4.x will be extended to use the DMX system. When the front and back-end servers are on the same physical hardware, it is possible to use the DRI to directly render to the back-end servers. First, the existing DRI will be extended to support multiple display heads, and then to support the DMX system. OpenGL rendering requests will be direct rendering to each back-end X server. The DRI will request the screen layout (either from the existing Xinerama extension or a DMX-specific extension). Support for synchronized swap buffers will also be added (on hardware that supports it). Note that a single front-end server with a single back-end server on the same physical machine can emulate accelerated indirect rendering.

When the front and back-end servers are on different physical hardware or are using non-XFree86 4.x X servers, a mechanism to render primitives across the back-end servers will be provided. There are several options as to how this can be implemented.

1. The existing OpenGL support in each back-end server can be used by repackaging rendering primitives and sending them to each back-end server. This option is similar to the unoptimized Xnest-style approach mentioned above. Optimization of this solution is beyond the scope of this project and is better suited to other distributed rendering systems.
2. Rendering to a pixmap in the front-end server using the current XFree86 4.x code, and then displaying to the back-ends via calls to XPutImage() is another option. This option is similar to the shadow frame buffer approach mentioned above. It is slower and bandwidth intensive, but has the advantage that the back-end servers are not required to have OpenGL support.

These, and other, options will be investigated in this phase of the work.

Work by others have made Chromium DMX-aware. Chromium will use the DMX X protocol extension to obtain information about the back-end servers and will render directly to those servers, bypassing DMX.

Status: OpenGL support by the glxProxy extension was implemented by SGI and has been integrated into the DMX code base.

3. Current issues

In this sections the current issues are outlined that require further investigation.

3.1 Fonts

The font path and glyphs need to be the same for the front-end and each of the back-end servers. Font glyphs could be sent to the back-end servers as necessary but this would consume a significant amount of available bandwidth during font rendering for clients that use many different fonts (e.g., Netscape). Initially, the font server (xfs) will be used to provide the fonts to both the front-end and back-end servers. Other possibilities will be investigated during development.

3.2 Zero width rendering primitives

To allow pixmap and on-screen rendering to be pixel perfect, all back-end servers must render zero width primitives exactly the same as the front-end renders the primitives to pixmaps. For those back-end servers that do not exactly match, zero width primitives will be automatically converted to one width primitives. This can be handled in the front-end server via the GC state.

3.3 Output scaling

With very large tiled displays, it might be difficult to read the information on the standard X desktop. In particular, the cursor can be easily lost and fonts could be difficult to read. Automatic primitive scaling might prove to be very useful. We will investigate the possibility of scaling the cursor and providing a set of alternate pre-scaled fonts to replace the standard fonts that many applications use (e.g., fixed). Other options for automatic scaling will also be investigated.

3.4 Per-screen colormaps

Each screen's default colormap in the set of back-end X servers should be able to be adjusted via a configuration utility. This support is would allow the back-end screens to be calibrated via custom gamma tables. On 24-bit systems that support a DirectColor visual, this type of correction can be accommodated. One possible implementation would be to advertise to X client of the DMX server a TrueColor visual while using DirectColor visuals on the back-end servers to implement this type of color correction. Other options will be investigated.

D. Background

This section describes the existing Open Source architectures that can be used to handle multiple screens and upon which this development project is based. This section was written before the implementation was finished, and may not reflect actual details of the implementation. It is left for historical interest only.

D.1 Core input device handling

The following is a description of how core input devices are handled by an X server.

D.1.1 InitInput()

InitInput() is a DDX function that is called at the start of each server generation from the X server's main() function. Its purpose is to determine what input devices are connected to the X server, register them with the DIX and MI layers, and initialize the input event queue. InitInput() does not have a return value, but the X server will abort if either a core keyboard device or a core pointer device are not registered. Extended input (XInput) devices can also be registered in InitInput().

InitInput() usually has implementation specific code to determine which input devices are available. For each input device it will be using, it calls AddInputDevice():

> AddInputDevice()This DIX function allocates the device structure, registers a callback function (which handles device init, close, on and off), and returns the input handle, which can be treated as opaque. It is called once for each input device.

Once input handles for core keyboard and core pointer devices have been obtained from AddInputDevice(), they are registered as core devices by calling RegisterPointerDevice() and RegisterKeyboardDevice(). Each of these should be called once. If both core devices are not registered, then the X server will exit with a fatal error when it attempts to start the input devices in InitAndStartDevices(), which is called directly after InitInput() (see below).

> Register{Pointer,Keyboard}Device()These DIX functions take a handle returned from AddInputDevice() and initialize the core input device fields in inputInfo, and initialize the input processing and grab functions for each core input device.

The core pointer device is then registered with the miPointer code (which does the high level cursor handling). While this registration is not necessary for correct miPointer operation in the current XFree86 code, it is still done mostly for compatibility reasons.

> miRegisterPointerDevice()This MI function registers the core pointer's input handle with with the miPointer code.

The final part of InitInput() is the initialization of the input event queue handling. In most cases, the event queue handling provided in the MI layer is used. The primary XFree86 X server uses its own event queue handling to support some special cases related to the XInput extension and the XFree86-specific DGA extension. For our purposes, the MI event queue handling should be suitable. It is initialized by calling mieqInit():

> mieqInit()This MI function initializes the MI event queue for the core devices, and is passed the public component of the input handles for the two core devices.

If a wakeup handler is required to deliver synchronous input events, it can be registered here by calling the DIX function RegisterBlockAndWakeupHandlers(). (See the devReadInput()

description below.)

D.1.2 InitAndStartDevices()

InitAndStartDevices() is a DIX function that is called immediately after InitInput() from the X server's main() function. Its purpose is to initialize each input device that was registered with AddInputDevice(), enable each input device that was successfully initialized, and create the list of enabled input devices. Once each registered device is processed in this way, the list of enabled input devices is checked to make sure that both a core keyboard device and core pointer device were registered and successfully enabled. If not, InitAndStartDevices() returns failure, and results in the the X server exiting with a fatal error.

Each registered device is initialized by calling its callback (dev->deviceProc) with the DEVICE_INIT argument:

> (*dev->deviceProc)(dev, DEVICE_INIT)This function initializes the device structs with core information relevant to the device.
>
> For pointer devices, this means specifying the number of buttons, default button mapping, the function used to get motion events (usually miPointerGetMotionEvents()), the function used to change/control the core pointer motion parameters (acceleration and threshold), and the motion buffer size.
>
> For keyboard devices, this means specifying the keycode range, default keycode to keysym mapping, default modifier mapping, and the functions used to sound the keyboard bell and modify/control the keyboard parameters (LEDs, bell pitch and duration, key click, which keys are auto-repeating, etc).

Each initialized device is enabled by calling EnableDevice():

> EnableDevice()EnableDevice() calls the device callback with DEVICE_ON:
>
> > (*dev->deviceProc)(dev, DEVICE_ON)This typically opens and initializes the relevant physical device, and when appropriate, registers the device's file descriptor (or equivalent) as a valid input source.
>
> EnableDevice() then adds the device handle to the X server's global list of enabled devices.

InitAndStartDevices() then verifies that a valid core keyboard and pointer has been initialized and enabled. It returns failure if either are missing.

D.1.3 devReadInput()

Each device will have some function that gets called to read its physical input. These may be called in a number of different ways. In the case of synchronous I/O, they will be called from a DDX wakeup-handler that gets called after the server detects that new input is available. In the case of asynchronous I/O, they will be called from a (SIGIO) signal handler triggered when new input is available. This function should do at least two things: make sure that input events get enqueued, and make sure that the cursor gets moved for motion events (except if these are handled later by the driver's own event queue processing function, which cannot be done when using the MI event queue handling).

Events are queued by calling mieqEnqueue():

> mieqEnqueue()This MI function is used to add input events to the event queue. It is simply passed the event to be queued.

The cursor position should be updated when motion events are enqueued, by calling either miPointerAbsoluteCursor() or miPointerDeltaCursor():

> miPointerAbsoluteCursor()This MI function is used to move the cursor to the absolute coordinates provided.miPointerDeltaCursor()This MI function is used to move

the cursor relative to its current position.

D.1.4 ProcessInputEvents()

ProcessInputEvents() is a DDX function that is called from the X server's main dispatch loop when new events are available in the input event queue. It typically processes the enqueued events, and updates the cursor/pointer position. It may also do other DDX-specific event processing.

Enqueued events are processed by mieqProcessInputEvents() and passed to the DIX layer for transmission to clients:

> mieqProcessInputEvents()This function processes each event in the event queue, and passes it to the device's input processing function. The DIX layer provides default functions to do this processing, and they handle the task of getting the events passed back to the relevant clients.miPointerUpdate()This function resynchronized the cursor position with the new pointer position. It also takes care of moving the cursor between screens when needed in multi-head configurations.

D.1.5 DisableDevice()

DisableDevice is a DIX function that removes an input device from the list of enabled devices. The result of this is that the device no longer generates input events. The device's data structures are kept in place, and disabling a device like this can be reversed by calling EnableDevice(). DisableDevice() may be called from the DDX when it is desirable to do so (e.g., the XFree86 server does this when VT switching). Except for special cases, this is not normally called for core input devices.

DisableDevice() calls the device's callback function with DEVICE_OFF:

> (*dev->deviceProc)(dev, DEVICE_OFF)This typically closes the relevant physical device, and when appropriate, unregisters the device's file descriptor (or equivalent) as a valid input source.

DisableDevice() then removes the device handle from the X server's global list of enabled devices.

D.1.6 CloseDevice()

CloseDevice is a DIX function that removes an input device from the list of available devices. It disables input from the device and frees all data structures associated with the device. This function is usually called from CloseDownDevices(), which is called from main() at the end of each server generation to close all input devices.

CloseDevice() calls the device's callback function with DEVICE_CLOSE:

> (*dev->deviceProc)(dev, DEVICE_CLOSE)This typically closes the relevant physical device, and when appropriate, unregisters the device's file descriptor (or equivalent) as a valid input source. If any device specific data structures were allocated when the device was initialized, they are freed here.

CloseDevice() then frees the data structures that were allocated for the device when it was registered/initialized.

D.1.7 LegalModifier()

"

LegalModifier() is a required DDX function that can be used to restrict which keys may be modifier keys. This seems to be present for historical reasons, so this function should simply return TRUE unconditionally.

D.2 Output handling

The following sections describe the main functions required to initialize, use and close the output device(s) for each screen in the X server.

D.2.1 InitOutput()

This DDX function is called near the start of each server generation from the X server's main() function. InitOutput()'s main purpose is to initialize each screen and fill in the global screenInfo structure for each screen. It is passed three arguments: a pointer to the screenInfo struct, which it is to initialize, and argc and argv from main(), which can be used to determine additional configuration information.

The primary tasks for this function are outlined below:

1. Parse configuration info: The first task of InitOutput() is to parses any configuration information from the configuration file. In addition to the XF86Config file, other configuration information can be taken from the command line. The command line options can be gathered either in InitOutput() or earlier in the ddxProcessArgument() function, which is called by ProcessCommandLine(). The configuration information determines the characteristics of the screen(s). For example, in the XFree86 X server, the XF86Config file specifies the monitor information, the screen resolution, the graphics devices and slots in which they are located, and, for Xinerama, the screens' layout.
2. Initialize screen info: The next task is to initialize the screen-dependent internal data structures. For example, part of what the XFree86 X server does is to allocate its screen and pixmap private indices, probe for graphics devices, compare the probed devices to the ones listed in the XF86Config file, and add the ones that match to the internal xf86Screens[] structure.
3. Set pixmap formats: The next task is to initialize the screenInfo's image byte order, bitmap bit order and bitmap scanline unit/pad. The screenInfo's pixmap format's depth, bits per pixel and scanline padding is also initialized at this stage.
4. Unify screen info: An optional task that might be done at this stage is to compare all of the information from the various screens and determines if they are compatible (i.e., if the set of screens can be unified into a single desktop). This task has potential to be useful to the DMX front-end server, if Xinerama's PanoramiXConsolidate() function is not sufficient.

Once these tasks are complete, the valid screens are known and each of these screens can be initialized by calling AddScreen().

D.2.2 AddScreen()

This DIX function is called from InitOutput(), in the DDX layer, to add each new screen to the screenInfo structure. The DDX screen initialization function and command line arguments (i.e., argc and argv) are passed to it as arguments.

This function first allocates a new Screen structure and any privates that are required. It then initializes some of the fields in the Screen struct and sets up the pixmap padding information. Finally, it calls the DDX screen initialization function ScreenInit(), which is described below. It returns the number of the screen that were just added, or -1 if there is insufficient memory to add the screen or if the DDX screen initialization fails.

D.2.3 ScreenInit()

This DDX function initializes the rest of the Screen structure with either generic or screen-specific functions (as necessary). It also fills in various screen attributes (e.g., width and height in millimeters, black and white pixel values).

The screen init function usually calls several functions to perform certain screen initialization functions. They are described below:

{mi,*fb}ScreenInit()The DDX layer's ScreenInit() function usually calls another layer's ScreenInit() function (e.g., miScreenInit() or fbScreenInit()) to initialize the fallbacks that the DDX driver does not specifically handle.

After calling another layer's ScreenInit() function, any screen-specific functions either wrap or replace the other layer's function pointers. If a function is to be wrapped, each of the old function pointers from the other layer are stored in a screen private area. Common functions to wrap are CloseScreen() and SaveScreen().miInitializeBackingStore()This MI function initializes the screen's backing storage functions, which are used to save areas of windows that are currently covered by other windows.miDCInitialize()This MI function initializes the MI cursor display structures and function pointers. If a hardware cursor is used, the DDX layer's ScreenInit() function will wrap additional screen and the MI cursor display function pointers.

Another common task for ScreenInit() function is to initialize the output device state. For example, in the XFree86 X server, the ScreenInit() function saves the original state of the video card and then initializes the video mode of the graphics device.

D.2.4 CloseScreen()

This function restores any wrapped screen functions (and in particular the wrapped CloseScreen() function) and restores the state of the output device to its original state. It should also free any private data it created during the screen initialization.

D.2.5 GC operations

When the X server is requested to render drawing primitives, it does so by calling drawing functions through the graphics context's operation function pointer table (i.e., the GCOps functions). These functions render the basic graphics operations such as drawing rectangles, lines, text or copying pixmaps. Default routines are provided either by the MI layer, which draws indirectly through a simple span interface, or by the framebuffer layers (e.g., CFB, MFB, FB), which draw directly to a linearly mapped frame buffer.

To take advantage of special hardware on the graphics device, specific GCOps functions can be replaced by device specific code. However, many times the graphics devices can handle only a subset of the possible states of the GC, so during graphics context validation, appropriate routines are selected based on the state and capabilities of the hardware. For example, some graphics hardware can accelerate single pixel width lines with certain dash patterns. Thus, for dash patterns that are not supported by hardware or for width 2 or greater lines, the default routine is chosen during GC validation.

Note that some pointers to functions that draw to the screen are stored in the Screen structure. They include GetImage(), GetSpans(), PaintWindowBackground(), PaintWindowBorder(), CopyWindow() and RestoreAreas().

D.2.6 Xnest

The Xnest X server is a special proxy X server that relays the X protocol requests that it receives to a ''real'' X server that then processes the requests and displays the results, if applicable. To the X applications, Xnest appears as if it is a regular X server. However, Xnest is both server to the X application and client of the real X server, which will actually handle the requests.

The Xnest server implements all of the standard input and output initialization steps outlined above.

InitOutput()Xnest takes its configuration information from command line arguments via ddxProcessArguments(). This information includes the real X server display to connect to, its default visual class, the screen depth, the Xnest window's geometry, etc. Xnest then connects to the real X server and gathers visual, colormap, depth and pixmap information about that server's display, creates a window on that server, which will be used as the root window for Xnest.

Next, Xnest initializes its internal data structures and uses the data from the real X server's pixmaps to initialize its own pixmap formats. Finally, it calls AddScreen(xnestOpenScreen, argc, argv) to initialize each of its screens.ScreenInit()Xnest's ScreenInit() function is called xnestOpenScreen(). This function initializes its screen's depth and visual information, and then calls miScreenInit() to set up the default screen functions. It then calls miInitializeBackingStore() and miDCInitialize() to initialize backing store and the software cursor. Finally, it replaces many of the screen functions with its own functions that repackage and send the requests to the real X server to which Xnest is attached.CloseScreen()This function frees its internal data structure allocations. Since it replaces instead of wrapping screen functions, there are no function pointers to unwrap. This can potentially lead to problems during server regeneration.GC operationsThe GC operations in Xnest are very simple since they leave all of the drawing to the real X server to which Xnest is attached. Each of the GCOps takes the request and sends it to the real X server using standard Xlib calls. For example, the X application issues a XDrawLines() call. This function turns into a protocol request to Xnest, which calls the xnestPolylines() function through Xnest's GCOps function pointer table. The xnestPolylines() function is only a single line, which calls XDrawLines() using the same arguments that were passed into it. Other GCOps functions are very similar. Two exceptions to the simple GCOps functions described above are the image functions and the BLT operations.

The image functions, GetImage() and PutImage(), must use a temporary image to hold the image to be put of the image that was just grabbed from the screen while it is in transit to the real X server or the client. When the image has been transmitted, the temporary image is destroyed.

The BLT operations, CopyArea() and CopyPlane(), handle not only the copy function, which is the same as the simple cases described above, but also the graphics exposures that result when the GC's graphics exposure bit is set to True. Graphics exposures are handled in a helper function, xnestBitBlitHelper(). This function collects the exposure events from the real X server and, if any resulting in regions being exposed, then those regions are passed back to the MI layer so that it can generate exposure events for the X application.

The Xnest server takes its input from the X server to which it is connected. When the mouse is in the Xnest server's window, keyboard and mouse events are received by the Xnest server, repackaged and sent back to any client that requests those events.

D.2.7 Shadow framebuffer

The most common type of framebuffer is a linear array memory that maps to the video memory on the graphics device. However, accessing that video memory over an I/O bus (e.g., ISA or PCI) can be slow. The shadow framebuffer layer allows the developer to keep the entire framebuffer in main memory and copy it back to video memory at regular intervals. It also has been extended to handle planar video memory and rotated framebuffers.

There are two main entry points to the shadow framebuffer code:

shadowAlloc(width, height, bpp)This function allocates the in memory copy of the framebuffer of size width*height*bpp. It returns a pointer to that memory, which will be used by the framebuffer ScreenInit() code during the screen's initialization.shadowInit(pScreen, updateProc, windowProc)This function initializes the shadow framebuffer layer. It wraps several screen drawing functions, and registers a block handler that will update the screen. The updateProc is a function that will copy the damaged regions to the screen, and the windowProc is a function that is used when the entire linear video memory range cannot be accessed simultaneously so that only a window into that memory is available (e.g., when using the VGA aperture).

The shadow framebuffer code keeps track of the damaged area of each screen by calculating the bounding box of all drawing operations that have occurred since the last screen update. Then, when the block handler is next called, only the damaged portion of the screen is updated.

Note that since the shadow framebuffer is kept in main memory, all drawing operations are performed by the CPU and, thus, no accelerated hardware drawing operations are possible.

D.3 Xinerama

Xinerama is an X extension that allows multiple physical screens controlled by a single X server to appear as a single screen. Although the extension allows clients to find the physical screen layout via extension requests, it is completely transparent to clients at the core X11 protocol level. The original public implementation of Xinerama came from Digital/Compaq. XFree86 rewrote it, filling in some missing pieces and improving both X11 core protocol compliance and performance. The Xinerama extension will be passing through X.Org's standardization process in the near future, and the sample implementation will be based on this rewritten version.

The current implementation of Xinerama is based primarily in the DIX (device independent) and MI (machine independent) layers of the X server. With few exceptions the DDX layers do not need any changes to support Xinerama. X server extensions often do need modifications to provide full Xinerama functionality.

The following is a code-level description of how Xinerama functions.

Note: Because the Xinerama extension was originally called the PanoramiX extension, many of the Xinerama functions still have the PanoramiX prefix.

PanoramiXExtensionInit()PanoramiXExtensionInit() is a device-independent extension function that is called at the start of each server generation from InitExtensions(), which is called from the X server's main() function after all output devices have been initialized, but before any input devices have been initialized.

PanoramiXNumScreens is set to the number of physical screens. If only one physical screen is present, the extension is disabled, and PanoramiXExtensionInit() returns without doing anything else.

The Xinerama extension is registered by calling AddExtension().

A local per-screen array of data structures (panoramiXdataPtr[]) is allocated for each physical screen, and GC and Screen private indexes are allocated, and both GC and Screen private areas are allocated for each physical screen. These hold Xinerama-specific per-GC and per-Screen data. Each screen's CreateGC and CloseScreen functions are wrapped by XineramaCreateGC() and XineramaCloseScreen() respectively. Some new resource classes are created for Xinerama drawables and GCs, and resource types for Xinerama windows, pixmaps and colormaps.

A region (XineramaScreenRegions[i]) is initialized for each physical screen, and single region (PanoramiXScreenRegion) is initialized to be the union of the screen regions. The panoramiXdataPtr[] array is also initialized with the size and origin of each screen. The relative positioning information for the physical screens is taken from the array dixScreenOrigins[], which the DDX layer must initialize in InitOutput(). The bounds of the combined screen is also calculated (PanoramiXPixWidth and PanoramiXPixHeight).

The DIX layer has a list of function pointers (ProcVector[]) that holds the entry points for the functions that process core protocol requests. The requests that Xinerama must intercept and break up into physical screen-specific requests are wrapped. The original set is copied to SavedProcVector[]. The types of requests intercepted are Window requests, GC requests, colormap requests, drawing requests, and some geometry-related requests. This wrapping allows the bulk of the protocol request processing to be handled transparently to the DIX layer. Some operations cannot be dealt with in this way and are handled with Xinerama-specific code within the DIX layer.PanoramiXConsolidate()PanoramiXConsolidate() is a device-independent extension function that is called directly from the X server's main() function after extensions and input/output devices have been initialized, and before the root windows are defined and initialized.

This function finds the set of depths (PanoramiXDepths[]) and visuals (PanoramiXVisuals[]) common to all of the physical screens. PanoramiXNumDepths is set to the number of common depths, and PanoramiXNumVisuals is set to the number of common visuals. Resources are created for the single root window and the default colormap. Each of these resources has per-physical screen entries.PanoramiXCreateConnectionBlock()PanoramiXConsolidate() is a device-independent extension function that is called directly from the X server's main() function after the per-physical screen root windows are created. It is called instead of the standard DIX CreateConnectionBlock() function. If this function returns FALSE, the X server exits with a fatal error. This function will return FALSE if no common depths were found in PanoramiXConsolidate(). With no common depths, Xinerama mode is not possible.

The connection block holds the information that clients get when they open a connection to the X server. It includes information such as the supported pixmap formats, number of screens and the sizes, depths, visuals, default colormap information, etc, for each of the screens (much of information that xdpyinfo shows). The connection block is initialized with the combined single screen values that were calculated in the above two functions.

The Xinerama extension allows the registration of connection block callback functions. The purpose of these is to allow other extensions to do processing at this point. These callbacks can be registered by calling XineramaRegisterConnectionBlockCallback() from the other extension's ExtensionInit() function. Each registered connection block callback is called at the end of PanoramiXCreateConnectionBlock().

D.3.1 Xinerama-specific changes to the DIX code

There are a few types of Xinerama-specific changes within the DIX code. The main ones are described here.

Functions that deal with colormap or GC -related operations outside of the intercepted protocol requests have a test added to only do the processing for screen numbers > 0. This is because they are handled for the single Xinerama screen and the processing is done once for screen 0.

The handling of motion events does some coordinate translation between the physical screen's origin and screen zero's origin. Also, motion events must be reported relative to the composite screen origin rather than the physical screen origins.

There is some special handling for cursor, window and event processing that cannot (either not at all or not conveniently) be done via the intercepted protocol requests. A particular case is the handling of pointers moving between physical screens.

D.3.2 Xinerama-specific changes to the MI code

The only Xinerama-specific change to the MI code is in miSendExposures() to handle the coordinate (and window ID) translation for expose events.

D.3.3 Intercepted DIX core requests

Xinerama breaks up drawing requests for dispatch to each physical screen. It also breaks up windows into pieces for each physical screen. GCs are translated into per-screen GCs. Colormaps are replicated on each physical screen. The functions handling the intercepted requests take care of breaking the requests and repackaging them so that they can be passed to the standard request handling functions for each screen in turn. In addition, and to aid the repackaging, the information from many of the intercepted requests is used to keep up to date the necessary state information for the single composite screen. Requests (usually those with replies) that can be satisfied completely from this stored state information do not call the standard request handling functions.

E. Development Results

In this section the results of each phase of development are discussed. This development took place between approximately June 2001 and July 2003.

E.1 Phase I

The initial development phase dealt with the basic implementation including the bootstrap code, which used the shadow framebuffer, and the unoptimized implementation, based on an Xnest-style implementation.

E.1.1 Scope

The goal of Phase I is to provide fundamental functionality that can act as a foundation for ongoing work:

1. Develop the proxy X server
    - The proxy X server will operate on the X11 protocol and relay requests as necessary to correctly perform the request.
    - Work will be based on the existing work for Xinerama and Xnest.
    - Input events and windowing operations are handled in the proxy server and rendering requests are repackaged and sent to each of the back-end servers for display.
    - The multiple screen layout (including support for overlapping screens) will be user configurable via a configuration file or through the configuration tool.
2. Develop graphical configuration tool
    - There will be potentially a large number of X servers to configure into a single display. The tool will allow the user to specify which servers are involved in the configuration and how they should be laid out.
3. Pass the X Test Suite
    - The X Test Suite covers the basic X11 operations. All tests known to succeed must correctly operate in the distributed X environment.

For this phase, the back-end X servers are assumed to be unmodified X servers that do not support any DMX-related protocol extensions; future optimization pathways are considered, but are not implemented; and the configuration tool is assumed to rely only on libraries in the X source tree (e.g., Xt).

E.1.2 Results

The proxy X server, Xdmx, was developed to distribute X11 protocol requests to the set of back-end X servers. It opens a window on each back-end server, which represents the part of the front-end's root window that is visible on that screen. It mirrors window, pixmap and other state in each back-end server. Drawing requests are sent to either windows or pixmaps on each back-end server. This code is based on Xnest and uses the existing Xinerama extension.

Input events can be taken from (1) devices attached to the back-end server, (2) core devices attached directly to the Xdmx server, or (3) from a "console" window on another X server. Events for these devices are gathered, processed and delivered to clients attached to the Xdmx server.

An intuitive configuration format was developed to help the user easily configure the multiple back-end X servers. It was defined (see grammar in Xdmx man page) and a parser was implemented that is used by the Xdmx server and by a standalone xdmxconfig utility. The parsing support was implemented such that it can be easily factored out of the X source tree for use with other tools (e.g., vdl). Support for converting legacy vdl-format configuration files to the DMX format is provided by the vdltodmx utility.

Originally, the configuration file was going to be a subsection of XFree86's XF86Config file, but that was not possible since Xdmx is a completely separate X server. Thus, a separate config file format was developed. In addition, a graphical configuration tool, xdmxconfig, was developed to allow the user to create and arrange the screens in the configuration file. The -configfile and -config command-line options can be used to start Xdmx using a configuration file.

An extension that enables remote input testing is required for the X Test Suite to function. During this phase, this extension (XTEST) was implemented in the Xdmx server. The results from running the X Test Suite are described in detail below.

E.1.3 X Test Suite

"

E.1.3.1 Introduction

"

The X Test Suite contains tests that verify Xlib functions operate correctly. The test suite is designed to run on a single X server; however, since X applications will not be able to tell the difference between the DMX server and a standard X server, the X Test Suite should also run on the DMX server.

The Xdmx server was tested with the X Test Suite, and the existing failures are noted in this section. To put these results in perspective, we first discuss expected X Test failures and how errors in underlying systems can impact Xdmx test results.

E.1.3.2 Expected Failures for a Single Head

"

A correctly implemented X server with a single screen is expected to fail certain X Test tests. The following well-known errors occur because of rounding error in the X server code: XDrawArc: Tests 42, 63, 66, 73 XDrawArcs: Tests 45, 66, 69, 76

The following failures occur because of the high-level X server implementation: XLoadQueryFont: Test 1 XListFontsWithInfo: Tests 3, 4 XQueryFont: Tests 1, 2

The following test fails when running the X server as root under Linux because of the way directory modes are interpreted: XWriteBitmapFile: Test 3

Depending on the video card used for the back-end, other failures may also occur because of bugs in the low-level driver implementation. Over time, failures of this kind are usually fixed by XFree86, but will show up in Xdmx testing until then.

E.1.3.3 Expected Failures for Xinerama

"

Xinerama fails several X Test Suite tests because of design decisions made for the current implementation of Xinerama. Over time, many of these errors will be corrected by XFree86 and the group working on a new Xinerama implementation. Therefore, Xdmx will also share X Suite Test failures with Xinerama.

We may be able to fix or work-around some of these failures at the Xdmx level, but this will require additional exploration that was not part of Phase I.

Xinerama is constantly improving, and the list of Xinerama-related failures depends on XFree86 version and the underlying graphics hardware. We tested with a variety of hardware, including nVidia, S3, ATI Radeon, and Matrox G400 (in dual-head mode). The list below includes only those failures that appear to be from the Xinerama layer, and does not include failures listed in the previous section, or failures that appear to be from the low-level graphics driver itself:

These failures were noted with multiple Xinerama configurations: XCopyPlane: Tests 13, 22, 31 (well-known Xinerama implementation issue) XSetFontPath: Test 4 XGetDefault: Test 5 XMatchVisualInfo: Test 1

These failures were noted only when using one dual-head video card with a 4.2.99.x XFree86 server: XListPixmapFormats: Test 1 XDrawRectangles: Test 45

These failures were noted only when using two video cards from different vendors with a 4.1.99.x XFree86 server: XChangeWindowAttributes: Test 32 XCreateWindow: Test 30 XDrawLine: Test 22 XFillArc: Test 22 XChangeKeyboardControl: Tests 9, 10 XRebindKeysym: Test 1

E.1.3.4 Additional Failures from Xdmx

"

When running Xdmx, no unexpected failures were noted. Since the Xdmx server is based on Xinerama, we expect to have most of the Xinerama failures present in the Xdmx server. Similarly, since the Xdmx server must rely on the low-level device drivers on each back-end server, we also expect that Xdmx will exhibit most of the back-end failures. Here is a summary: XListPixmapFormats: Test 1 (configuration dependent) XChangeWindowAttributes: Test 32 XCreateWindow: Test 30 XCopyPlane: Test 13, 22, 31 XSetFontPath: Test 4 XGetDefault: Test 5 (configuration dependent) XMatchVisualInfo: Test 1 XRebindKeysym: Test 1 (configuration dependent)

Note that this list is shorter than the combined list for Xinerama because Xdmx uses different code paths to perform some Xinerama operations. Further, some Xinerama failures have been fixed in the XFree86 4.2.99.x CVS repository.

E.1.3.5 Summary and Future Work

"

Running the X Test Suite on Xdmx does not produce any failures that cannot be accounted for by the underlying Xinerama subsystem used by the front-end or by the low-level device-driver code running on the back-end X servers. The Xdmx server therefore is as ''correct'' as possible with respect to the standard set of X Test Suite tests.

During the following phases, we will continue to verify Xdmx correctness using the X Test Suite. We may also use other tests suites or write additional tests that run under the X Test Suite that specifically verify the expected behavior of DMX.

E.1.4 Fonts

In Phase I, fonts are handled directly by both the front-end and the back-end servers, which is required since we must treat each back-end server during this phase as a ''black box''. What this requires is that the front- and back-end servers must share the exact same font path. There are two ways to help make sure that all servers share the same font path:

1. First, each server can be configured to use the same font server. The font server, xfs, can be configured to serve fonts to multiple X servers via TCP.
2. Second, each server can be configured to use the same font path and either those font paths can be copied to each back-end machine or they can be mounted (e.g., via NFS) on each back-end machine.

One additional concern is that a client program can set its own font path, and if it does so, then that font path must be available on each back-end machine.

The -fontpath command line option was added to allow users to initialize the font path of the front end server. This font path is propagated to each back-end server when the default font is loaded. If there are any problems, an error message is printed, which will describe the problem and list the current font path. For more information about setting the font path, see the -fontpath option description in the man page.

E.1.5 Performance

Phase I of development was not intended to optimize performance. Its focus was on completely and correctly handling the base X11 protocol in the Xdmx server. However, several insights were gained during Phase I, which are listed here for reference during the next phase of development.

1. Calls to XSync() can slow down rendering since it requires a complete round trip to and from a back-end server. This is especially problematic when communicating over long haul networks.
2. Sending drawing requests to only the screens that they overlap should improve performance.

E.1.6 Pixmaps

Pixmaps were originally expected to be handled entirely in the front-end X server; however, it was found that this overly complicated the rendering code and would have required sending potentially large images to each back server that required them when copying from pixmap to screen. Thus, pixmap state is mirrored in the back-end server just as it is with regular window state. With this implementation, the same rendering code that draws to windows can be used to draw to pixmaps on the back-end server, and no large image transfers are required to copy from pixmap to window.

E.2 Phase II

The second phase of development concentrates on performance optimizations. These optimizations are documented here, with x11perf data to show how the optimizations improve performance.

All benchmarks were performed by running Xdmx on a dual processor 1.4GHz AMD Athlon machine with 1GB of RAM connecting over 100baseT to two single-processor 1GHz Pentium III machines with 256MB of RAM and ATI Rage 128 (RF) video cards. The front end was running Linux 2.4.20-pre1-ac1 and the back ends were running Linux 2.4.7-10 and version 4.2.99.1 of XFree86 pulled from the XFree86 CVS repository on August 7, 2002. All systems were running

Red Hat Linux 7.2.

E.2.1 Moving from XFree86 4.1.99.1 to 4.2.0.0

For phase II, the working source tree was moved to the branch tagged with dmx-1-0-branch and was updated from version 4.1.99.1 (20 August 2001) of the XFree86 sources to version 4.2.0.0 (18 January 2002). After this update, the following tests were noted to be more than 10% faster: 1.13 Fill 300x300 opaque stippled trapezoid (161x145 stipple) 1.16 Fill 1x1 tiled trapezoid (161x145 tile) 1.13 Fill 10x10 tiled trapezoid (161x145 tile) 1.17 Fill 100x100 tiled trapezoid (161x145 tile) 1.16 Fill 1x1 tiled trapezoid (216x208 tile) 1.20 Fill 10x10 tiled trapezoid (216x208 tile) 1.15 Fill 100x100 tiled trapezoid (216x208 tile) 1.37 Circulate Unmapped window (200 kids)

And the following tests were noted to be more than 10% slower: 0.88 Unmap window via parent (25 kids) 0.75 Circulate Unmapped window (4 kids) 0.79 Circulate Unmapped window (16 kids) 0.80 Circulate Unmapped window (25 kids) 0.82 Circulate Unmapped window (50 kids) 0.85 Circulate Unmapped window (75 kids)

These changes were not caused by any changes in the DMX system, and may point to changes in the XFree86 tree or to tests that have more "jitter" than most other x11perf tests.

E.2.2 Global changes

During the development of the Phase II DMX server, several global changes were made. These changes were also compared with the Phase I server. The following tests were noted to be more than 10% faster: 1.13 Fill 300x300 opaque stippled trapezoid (161x145 stipple) 1.15 Fill 1x1 tiled trapezoid (161x145 tile) 1.13 Fill 10x10 tiled trapezoid (161x145 tile) 1.17 Fill 100x100 tiled trapezoid (161x145 tile) 1.16 Fill 1x1 tiled trapezoid (216x208 tile) 1.19 Fill 10x10 tiled trapezoid (216x208 tile) 1.15 Fill 100x100 tiled trapezoid (216x208 tile) 1.15 Circulate Unmapped window (4 kids)

The following tests were noted to be more than 10% slower: 0.69 Scroll 10x10 pixels 0.68 Scroll 100x100 pixels 0.68 Copy 10x10 from window to window 0.68 Copy 100x100 from window to window 0.76 Circulate Unmapped window (75 kids) 0.83 Circulate Unmapped window (100 kids)

For the remainder of this analysis, the baseline of comparison will be the Phase II deliverable with all optimizations disabled (unless otherwise noted). This will highlight how the optimizations in isolation impact performance.

E.2.3 XSync() Batching

During the Phase I implementation, XSync() was called after every protocol request made by the DMX server. This provided the DMX server with an interactive feel, but defeated X11's protocol buffering system and introduced round-trip wire latency into every operation. During Phase II, DMX was changed so that protocol requests are no longer followed by calls to XSync(). Instead, the need for an XSync() is noted, and XSync() calls are only made every 100mS or when the DMX server specifically needs to make a call to guarantee interactivity. With this new system, X11 buffers protocol as much as possible during a 100mS interval, and many unnecessary XSync() calls are avoided.

Out of more than 300 x11perf tests, 8 tests became more than 100 times faster, with 68 more than 50X faster, 114 more than 10X faster, and 181 more than 2X faster. See table below for summary.

The following tests were noted to be more than 10% slower with XSync() batching on: 0.88 500x500 tiled rectangle (161x145 tile) 0.89 Copy 500x500 from window to window

E.2.4 Offscreen Optimization

Windows span one or more of the back-end servers' screens; however, during Phase I development, windows were created on every back-end server and every rendering request was sent to every window regardless of whether or not that window was visible. With the offscreen optimization, the DMX server tracks when a window is completely off of a back-end server's screen and, in that case, it does not send rendering requests to those back-end windows. This optimization saves bandwidth between the front and back-end servers, and it reduces the number of XSync() calls. The performance tests were run on a DMX system with only two back-end servers. Greater performance gains will be had as the number of back-end servers increases.

Out of more than 300 x11perf tests, 3 tests were at least twice as fast, and 146 tests were at least 10% faster. Two tests were more than 10% slower with the offscreen optimization: 0.88 Hide/expose window via popup (4 kids) 0.89 Resize unmapped window (75 kids)

E.2.5 Lazy Window Creation Optimization

As mentioned above, during Phase I, windows were created on every back-end server even if they were not visible on that back-end. With the lazy window creation optimization, the DMX server does not create windows on a back-end server until they are either visible or they become the parents of a visible window. This optimization builds on the offscreen optimization (described above) and requires it to be enabled.

The lazy window creation optimization works by creating the window data structures in the front-end server when a client creates a window, but delays creation of the window on the back-end server(s). A private window structure in the DMX server saves the relevant window data and tracks changes to the window's attributes and stacking order for later use. The only times a window is created on a back-end server are (1) when it is mapped and is at least partially overlapping the back-end server's screen (tracked by the offscreen optimization), or (2) when the window becomes the parent of a previously visible window. The first case occurs when a window is mapped or when a visible window is copied, moved or resized and now overlaps the back-end server's screen. The second case occurs when starting a window manager after having created windows to which the window manager needs to add decorations.

When either case occurs, a window on the back-end server is created using the data saved in the DMX server's window private data structure. The stacking order is then adjusted to correctly place the window on the back-end and lastly the window is mapped. From this time forward, the window is handled exactly as if the window had been created at the time of the client's request.

Note that when a window is no longer visible on a back-end server's screen (e.g., it is moved offscreen), the window is not destroyed; rather, it is kept and reused later if the window once again becomes visible on the back-end server's screen. Originally with this optimization, destroying windows was implemented but was later rejected because it increased bandwidth when windows were opaquely moved or resized, which is common in many window managers.

The performance tests were run on a DMX system with only two back-end servers. Greater performance gains will be had as the number of back-end servers increases.

This optimization improved the following x11perf tests by more than 10%: 1.10 500x500 rectangle outline 1.12 Fill 100x100 stippled trapezoid (161x145 stipple) 1.20 Circulate Unmapped window (50 kids) 1.19 Circulate Unmapped window (75 kids)

E.2.6 Subdividing Rendering Primitives

X11 imaging requests transfer significant data between the client and the X server. During Phase I, the DMX server would then transfer the image data to each back-end server. Even with the offscreen optimization (above), these requests still required transferring significant data to each back-end server that contained a visible portion of the window. For example, if the client uses XPutImage() to copy an image to a window that overlaps the entire DMX screen, then the entire image is copied by the DMX server to every back-end server.

To reduce the amount of data transferred between the DMX server and the back-end servers when XPutImage() is called, the image data is subdivided and only the data that will be visible on a back-end server's screen is sent to that back-end server. Xinerama already implements a subdivision algorithm for XGetImage() and no further optimization was needed.

Other rendering primitives were analyzed, but the time required to subdivide these primitives was a significant proportion of the time required to send the entire rendering request to the back-end server, so this optimization was rejected for the other rendering primitives.

Again, the performance tests were run on a DMX system with only two back-end servers. Greater performance gains will be had as the number of back-end servers increases.

This optimization improved the following x11perf tests by more than 10%: 1.12 Fill 100x100 stippled trapezoid (161x145 stipple) 1.26 PutImage 10x10 square 1.83 PutImage 100x100 square 1.91 PutImage 500x500 square 1.40 PutImage XY 10x10 square 1.48 PutImage XY 100x100 square 1.50 PutImage XY 500x500 square 1.45 Circulate Unmapped window (75 kids) 1.74 Circulate Unmapped window (100 kids)

The following test was noted to be more than 10% slower with this optimization: 0.88 10-pixel fill chord partial circle

E.2.7 Summary of x11perf Data

With all of the optimizations on, 53 x11perf tests are more than 100X faster than the unoptimized Phase II deliverable, with 69 more than 50X faster, 73 more than 10X faster, and 199 more than twice as fast. No tests were more than 10% slower than the unoptimized Phase II deliverable. (Compared with the Phase I deliverable, only Circulate Unmapped window (100 kids) was more than 10% slower than the Phase II deliverable. As noted above, this test seems to have wider variability than other x11perf tests.)

The following table summarizes relative x11perf test changes for all optimizations individually and collectively. Note that some of the optimizations have a synergistic effect when used together.

1: XSync() batching only 2: Off screen optimizations only 3: Window optimizations only 4: Subdivprims only 5: All optimizations

1	2	3	4	5	Operation
2.14	1.85	1.00	1.00	4.13	Dot
1.67	1.80	1.00	1.00	3.31	1x1 rectangle
2.38	1.43	1.00	1.00	2.44	10x10 rectangle
1.00	1.00	0.92	0.98	1.00	100x100 rectangle
1.00	1.00	1.00	1.00	1.00	500x500 rectangle
1.83	1.85	1.05	1.06	3.54	1x1 stippled rectangle (8x8 stipple)
2.43	1.43	1.00	1.00	2.41	10x10 stippled rectangle (8x8 stipple)
0.98	1.00	1.00	1.00	1.00	100x100 stippled rectangle (8x8 stipple)
1.00	1.00	1.00	1.00	0.98	500x500 stippled rectangle (8x8 stipple)
1.75	1.75	1.00	1.00	3.40	1x1 opaque stippled rectangle (8x8 stipple)
2.38	1.42	1.00	1.00	2.34	10x10 opaque stippled rectangle (8x8 stipple)
1.00	1.00	0.97	0.97	1.00	100x100 opaque stippled rectangle (8x8 stipple)
1.00	1.00	1.00	1.00	0.99	500x500 opaque stippled rectangle (8x8 stipple)

1.82	1.82	1.04	1.04	3.56	1x1 tiled rectangle (4x4 tile)
2.33	1.42	1.00	1.00	2.37	10x10 tiled rectangle (4x4 tile)
1.00	0.92	1.00	1.00	1.00	100x100 tiled rectangle (4x4 tile)
1.00	1.00	1.00	1.00	1.00	500x500 tiled rectangle (4x4 tile)
1.94	1.62	1.00	1.00	3.66	1x1 stippled rectangle (17x15 stipple)
1.74	1.28	1.00	1.00	1.73	10x10 stippled rectangle (17x15 stipple)
1.00	1.00	1.00	0.89	0.98	100x100 stippled rectangle (17x15 stipple)
1.00	1.00	1.00	1.00	0.98	500x500 stippled rectangle (17x15 stipple)
1.94	1.62	1.00	1.00	3.67	1x1 opaque stippled rectangle (17x15 stipple)
1.69	1.26	1.00	1.00	1.66	10x10 opaque stippled rectangle (17x15 stipple)
1.00	0.95	1.00	1.00	1.00	100x100 opaque stippled rectangle (17x15 stipple)
1.00	1.00	1.00	1.00	0.97	500x500 opaque stippled rectangle (17x15 stipple)
1.93	1.61	0.99	0.99	3.69	1x1 tiled rectangle (17x15 tile)
1.73	1.27	1.00	1.00	1.72	10x10 tiled rectangle (17x15 tile)
1.00	1.00	1.00	1.00	0.98	100x100 tiled rectangle (17x15 tile)
1.00	1.00	0.97	0.97	1.00	500x500 tiled rectangle (17x15 tile)
1.95	1.63	1.00	1.00	3.83	1x1 stippled rectangle (161x145 stipple)
1.80	1.30	1.00	1.00	1.83	10x10 stippled rectangle (161x145 stipple)
0.97	1.00	1.00	1.00	1.01	100x100 stippled rectangle (161x145 stipple)
1.00	1.00	1.00	1.00	0.98	500x500 stippled rectangle (161x145 stipple)
1.95	1.63	1.00	1.00	3.56	1x1 opaque stippled rectangle (161x145 stipple)
1.65	1.25	1.00	1.00	1.68	10x10 opaque stippled rectangle (161x145 stipple)
1.00	1.00	1.00	1.00	1.01	100x100 opaque stippled rectangle (161x145...
1.00	1.00	1.00	1.00	0.97	500x500 opaque stippled rectangle (161x145...
1.95	1.63	0.98	0.99	3.80	1x1 tiled rectangle (161x145 tile)
1.67	1.26	1.00	1.00	1.67	10x10 tiled rectangle (161x145 tile)
1.13	1.14	1.14	1.14	1.14	100x100 tiled rectangle (161x145 tile)
0.88	1.00	1.00	1.00	0.99	500x500 tiled rectangle (161x145 tile)
1.93	1.63	1.00	1.00	3.53	1x1 tiled rectangle (216x208 tile)
1.69	1.26	1.00	1.00	1.66	10x10 tiled rectangle (216x208 tile)
1.00	1.00	1.00	1.00	1.00	100x100 tiled rectangle (216x208 tile)
1.00	1.00	1.00	1.00	1.00	500x500 tiled rectangle (216x208 tile)
1.82	1.70	1.00	1.00	3.38	1-pixel line segment
2.07	1.56	0.90	1.00	3.31	10-pixel line segment
1.29	1.10	1.00	1.00	1.27	100-pixel line segment
1.05	1.06	1.03	1.03	1.09	500-pixel line segment
1.30	1.13	1.00	1.00	1.29	100-pixel line segment (1 kid)
1.32	1.15	1.00	1.00	1.32	100-pixel line segment (2 kids)
1.33	1.16	1.00	1.00	1.33	100-pixel line segment (3 kids)
1.92	1.64	1.00	1.00	3.73	10-pixel dashed segment
1.34	1.16	1.00	1.00	1.34	100-pixel dashed segment
1.24	1.11	0.99	0.97	1.23	100-pixel double-dashed segment
1.72	1.77	1.00	1.00	3.25	10-pixel horizontal line segment
1.83	1.66	1.01	1.00	3.54	100-pixel horizontal line segment
1.86	1.30	1.00	1.00	1.84	500-pixel horizontal line segment
2.11	1.52	1.00	0.99	3.02	10-pixel vertical line segment
1.21	1.10	1.00	1.00	1.20	100-pixel vertical line segment
1.03	1.03	1.00	1.00	1.02	500-pixel vertical line segment
4.42	1.68	1.00	1.01	4.64	10x1 wide horizontal line segment
1.83	1.31	1.00	1.00	1.83	100x10 wide horizontal line segment
1.07	1.00	0.96	1.00	1.07	500x50 wide horizontal line segment
4.10	1.67	1.00	1.00	4.62	10x1 wide vertical line segment
1.50	1.24	1.06	1.06	1.48	100x10 wide vertical line segment
1.06	1.03	1.00	1.00	1.05	500x50 wide vertical line segment

2.54	1.61	1.00	1.00	3.61	1-pixel line
2.71	1.48	1.00	1.00	2.67	10-pixel line
1.19	1.09	1.00	1.00	1.19	100-pixel line
1.04	1.02	1.00	1.00	1.03	500-pixel line
2.68	1.51	0.98	1.00	3.17	10-pixel dashed line
1.23	1.11	0.99	0.99	1.23	100-pixel dashed line
1.15	1.08	1.00	1.00	1.15	100-pixel double-dashed line
2.27	1.39	1.00	1.00	2.23	10x1 wide line
1.20	1.09	1.00	1.00	1.20	100x10 wide line
1.04	1.02	1.00	1.00	1.04	500x50 wide line
1.52	1.45	1.00	1.00	1.52	100x10 wide dashed line
1.54	1.47	1.00	1.00	1.54	100x10 wide double-dashed line
1.97	1.30	0.96	0.95	1.95	10x10 rectangle outline
1.44	1.27	1.00	1.00	1.43	100x100 rectangle outline
3.22	2.16	1.10	1.09	3.61	500x500 rectangle outline
1.95	1.34	1.00	1.00	1.90	10x10 wide rectangle outline
1.14	1.14	1.00	1.00	1.13	100x100 wide rectangle outline
1.00	1.00	1.00	1.00	1.00	500x500 wide rectangle outline
1.57	1.72	1.00	1.00	3.03	1-pixel circle
1.96	1.35	1.00	1.00	1.92	10-pixel circle
1.21	1.07	0.86	0.97	1.20	100-pixel circle
1.08	1.04	1.00	1.00	1.08	500-pixel circle
1.39	1.19	1.03	1.03	1.38	100-pixel dashed circle
1.21	1.11	1.00	1.00	1.23	100-pixel double-dashed circle
1.59	1.28	1.00	1.00	1.58	10-pixel wide circle
1.22	1.12	0.99	1.00	1.22	100-pixel wide circle
1.06	1.04	1.00	1.00	1.05	500-pixel wide circle
1.87	1.84	1.00	1.00	1.85	100-pixel wide dashed circle
1.90	1.93	1.01	1.01	1.90	100-pixel wide double-dashed circle
2.13	1.43	1.00	1.00	2.32	10-pixel partial circle
1.42	1.18	1.00	1.00	1.42	100-pixel partial circle
1.92	1.85	1.01	1.01	1.89	10-pixel wide partial circle
1.73	1.67	1.00	1.00	1.73	100-pixel wide partial circle
1.36	1.95	1.00	1.00	2.64	1-pixel solid circle
2.02	1.37	1.00	1.00	2.03	10-pixel solid circle
1.19	1.09	1.00	1.00	1.19	100-pixel solid circle
1.02	0.99	1.00	1.00	1.01	500-pixel solid circle
1.74	1.28	1.00	0.88	1.73	10-pixel fill chord partial circle
1.31	1.13	1.00	1.00	1.31	100-pixel fill chord partial circle
1.67	1.31	1.03	1.03	1.72	10-pixel fill slice partial circle
1.30	1.13	1.00	1.00	1.28	100-pixel fill slice partial circle
2.45	1.49	1.01	1.00	2.71	10-pixel ellipse
1.22	1.10	1.00	1.00	1.22	100-pixel ellipse
1.09	1.04	1.00	1.00	1.09	500-pixel ellipse
1.90	1.28	1.00	1.00	1.89	100-pixel dashed ellipse
1.62	1.24	0.96	0.97	1.61	100-pixel double-dashed ellipse
2.43	1.50	1.00	1.00	2.42	10-pixel wide ellipse
1.61	1.28	1.03	1.03	1.60	100-pixel wide ellipse
1.08	1.05	1.00	1.00	1.08	500-pixel wide ellipse
1.93	1.88	1.00	1.00	1.88	100-pixel wide dashed ellipse
1.94	1.89	1.01	1.00	1.94	100-pixel wide double-dashed ellipse
2.31	1.48	1.00	1.00	2.67	10-pixel partial ellipse
1.38	1.17	1.00	1.00	1.38	100-pixel partial ellipse
2.00	1.85	0.98	0.97	1.98	10-pixel wide partial ellipse

1.89	1.86	1.00	1.00	1.89	100-pixel wide partial ellipse
3.49	1.60	1.00	1.00	3.65	10-pixel filled ellipse
1.67	1.26	1.00	1.00	1.67	100-pixel filled ellipse
1.06	1.04	1.00	1.00	1.06	500-pixel filled ellipse
2.38	1.43	1.01	1.00	2.32	10-pixel fill chord partial ellipse
2.06	1.30	1.00	1.00	2.05	100-pixel fill chord partial ellipse
2.27	1.41	1.00	1.00	2.27	10-pixel fill slice partial ellipse
1.98	1.33	1.00	0.97	1.97	100-pixel fill slice partial ellipse
57.46	1.99	1.01	1.00	114.92	Fill 1x1 equivalent triangle
56.94	1.98	1.01	1.00	73.89	Fill 10x10 equivalent triangle
6.07	1.75	1.00	1.00	6.07	Fill 100x100 equivalent triangle
51.12	1.98	1.00	1.00	102.81	Fill 1x1 trapezoid
51.42	1.82	1.01	1.00	94.89	Fill 10x10 trapezoid
6.47	1.80	1.00	1.00	6.44	Fill 100x100 trapezoid
1.56	1.28	1.00	0.99	1.56	Fill 300x300 trapezoid
51.27	1.97	0.96	0.97	102.54	Fill 1x1 stippled trapezoid (8x8 stipple)
51.73	2.00	1.02	1.02	67.92	Fill 10x10 stippled trapezoid (8x8 stipple)
5.36	1.72	1.00	1.00	5.36	Fill 100x100 stippled trapezoid (8x8 stipple)
1.54	1.26	1.00	1.00	1.59	Fill 300x300 stippled trapezoid (8x8 stipple)
51.41	1.94	1.01	1.00	102.82	Fill 1x1 opaque stippled trapezoid (8x8 stipple)
50.71	1.95	0.99	1.00	65.44	Fill 10x10 opaque stippled trapezoid (8x8...
5.33	1.73	1.00	1.00	5.36	Fill 100x100 opaque stippled trapezoid (8x8...
1.58	1.25	1.00	1.00	1.58	Fill 300x300 opaque stippled trapezoid (8x8...
51.56	1.96	0.99	0.90	103.68	Fill 1x1 tiled trapezoid (4x4 tile)
51.59	1.99	1.01	1.01	62.25	Fill 10x10 tiled trapezoid (4x4 tile)
5.38	1.72	1.00	1.00	5.38	Fill 100x100 tiled trapezoid (4x4 tile)
1.54	1.25	1.00	0.99	1.58	Fill 300x300 tiled trapezoid (4x4 tile)
51.70	1.98	1.01	1.01	103.98	Fill 1x1 stippled trapezoid (17x15 stipple)
44.86	1.97	1.00	1.00	44.86	Fill 10x10 stippled trapezoid (17x15 stipple)
2.74	1.56	1.00	1.00	2.73	Fill 100x100 stippled trapezoid (17x15 stipple)
1.29	1.14	1.00	1.00	1.27	Fill 300x300 stippled trapezoid (17x15 stipple)
51.41	1.96	0.96	0.95	103.39	Fill 1x1 opaque stippled trapezoid (17x15...
45.14	1.96	1.01	1.00	45.14	Fill 10x10 opaque stippled trapezoid (17x15...
2.68	1.56	1.00	1.00	2.68	Fill 100x100 opaque stippled trapezoid (17x15...
1.26	1.10	1.00	1.00	1.28	Fill 300x300 opaque stippled trapezoid (17x15...
51.13	1.97	1.00	0.99	103.39	Fill 1x1 tiled trapezoid (17x15 tile)
47.58	1.96	1.00	1.00	47.86	Fill 10x10 tiled trapezoid (17x15 tile)
2.74	1.56	1.00	1.00	2.74	Fill 100x100 tiled trapezoid (17x15 tile)
1.29	1.14	1.00	1.00	1.28	Fill 300x300 tiled trapezoid (17x15 tile)
51.13	1.97	0.99	0.97	103.39	Fill 1x1 stippled trapezoid (161x145 stipple)
45.14	1.97	1.00	1.00	44.29	Fill 10x10 stippled trapezoid (161x145 stipple)
3.02	1.77	1.12	1.12	3.38	Fill 100x100 stippled trapezoid (161x145 stipple)
1.31	1.13	1.00	1.00	1.30	Fill 300x300 stippled trapezoid (161x145 stipple)
51.27	1.97	1.00	1.00	103.10	Fill 1x1 opaque stippled trapezoid (161x145...
45.01	1.97	1.00	1.00	45.01	Fill 10x10 opaque stippled trapezoid (161x145...
2.67	1.56	1.00	1.00	2.69	Fill 100x100 opaque stippled trapezoid (161x145..
1.29	1.13	1.00	1.01	1.27	Fill 300x300 opaque stippled trapezoid (161x145..
51.41	1.96	1.00	0.99	103.39	Fill 1x1 tiled trapezoid (161x145 tile)
45.01	1.96	0.98	1.00	45.01	Fill 10x10 tiled trapezoid (161x145 tile)
2.62	1.36	1.00	1.00	2.69	Fill 100x100 tiled trapezoid (161x145 tile)
1.27	1.13	1.00	1.00	1.22	Fill 300x300 tiled trapezoid (161x145 tile)
51.13	1.98	1.00	1.00	103.39	Fill 1x1 tiled trapezoid (216x208 tile)
45.14	1.97	1.01	0.99	45.14	Fill 10x10 tiled trapezoid (216x208 tile)
2.62	1.55	1.00	1.00	2.71	Fill 100x100 tiled trapezoid (216x208 tile)

1.28	1.13	1.00	1.00	1.20	Fill 300x300 tiled trapezoid (216x208 tile)
50.71	1.95	1.00	1.00	54.70	Fill 10x10 equivalent complex polygon
5.51	1.71	0.96	0.98	5.47	Fill 100x100 equivalent complex polygons
8.39	1.97	1.00	1.00	16.75	Fill 10x10 64-gon (Convex)
8.38	1.83	1.00	1.00	8.43	Fill 100x100 64-gon (Convex)
8.50	1.96	1.00	1.00	16.64	Fill 10x10 64-gon (Complex)
8.26	1.83	1.00	1.00	8.35	Fill 100x100 64-gon (Complex)
14.09	1.87	1.00	1.00	14.05	Char in 80-char line (6x13)
11.91	1.87	1.00	1.00	11.95	Char in 70-char line (8x13)
11.16	1.85	1.01	1.00	11.10	Char in 60-char line (9x15)
10.09	1.78	1.00	1.00	10.09	Char16 in 40-char line (k14)
6.15	1.75	1.00	1.00	6.31	Char16 in 23-char line (k24)
11.92	1.90	1.03	1.03	11.88	Char in 80-char line (TR 10)
8.18	1.78	1.00	0.99	8.17	Char in 30-char line (TR 24)
42.83	1.44	1.01	1.00	42.11	Char in 20/40/20 line (6x13, TR 10)
27.45	1.43	1.01	1.01	27.45	Char16 in 7/14/7 line (k14, k24)
12.13	1.85	1.00	1.00	12.05	Char in 80-char image line (6x13)
10.00	1.84	1.00	1.00	10.00	Char in 70-char image line (8x13)
9.18	1.83	1.00	1.00	9.12	Char in 60-char image line (9x15)
9.66	1.82	0.98	0.95	9.66	Char16 in 40-char image line (k14)
5.82	1.72	1.00	1.00	5.99	Char16 in 23-char image line (k24)
8.70	1.80	1.00	1.00	8.65	Char in 80-char image line (TR 10)
4.67	1.66	1.00	1.00	4.67	Char in 30-char image line (TR 24)
84.43	1.47	1.00	1.00	124.18	Scroll 10x10 pixels
3.73	1.50	1.00	0.98	3.73	Scroll 100x100 pixels
1.00	1.00	1.00	1.00	1.00	Scroll 500x500 pixels
84.43	1.51	1.00	1.00	134.02	Copy 10x10 from window to window
3.62	1.51	0.98	0.98	3.62	Copy 100x100 from window to window
0.89	1.00	1.00	1.00	1.00	Copy 500x500 from window to window
57.06	1.99	1.00	1.00	88.64	Copy 10x10 from pixmap to window
2.49	2.00	1.00	1.00	2.48	Copy 100x100 from pixmap to window
1.00	0.91	1.00	1.00	0.98	Copy 500x500 from pixmap to window
2.04	1.01	1.00	1.00	2.03	Copy 10x10 from window to pixmap
1.05	1.00	1.00	1.00	1.05	Copy 100x100 from window to pixmap
1.00	1.00	0.93	1.00	1.04	Copy 500x500 from window to pixmap
58.52	1.03	1.03	1.02	57.95	Copy 10x10 from pixmap to pixmap
2.40	1.00	1.00	1.00	2.45	Copy 100x100 from pixmap to pixmap
1.00	1.00	1.00	1.00	1.00	Copy 500x500 from pixmap to pixmap
51.57	1.92	1.00	1.00	85.75	Copy 10x10 1-bit deep plane
6.37	1.75	1.01	1.01	6.37	Copy 100x100 1-bit deep plane
1.26	1.11	1.00	1.00	1.24	Copy 500x500 1-bit deep plane
4.23	1.63	0.98	0.97	4.38	Copy 10x10 n-bit deep plane
1.04	1.02	1.00	1.00	1.04	Copy 100x100 n-bit deep plane
1.00	1.00	1.00	1.00	1.00	Copy 500x500 n-bit deep plane
6.45	1.98	1.00	1.26	12.80	PutImage 10x10 square
1.10	1.87	1.00	1.83	2.11	PutImage 100x100 square
1.02	1.93	1.00	1.91	1.91	PutImage 500x500 square
4.17	1.78	1.00	1.40	7.18	PutImage XY 10x10 square
1.27	1.49	0.97	1.48	2.10	PutImage XY 100x100 square
1.00	1.50	1.00	1.50	1.52	PutImage XY 500x500 square
1.07	1.01	1.00	1.00	1.06	GetImage 10x10 square
1.01	1.00	1.00	1.00	1.01	GetImage 100x100 square
1.00	1.00	1.00	1.00	1.00	GetImage 500x500 square
1.56	1.00	0.99	0.97	1.56	GetImage XY 10x10 square

1.02 1.00 1.00 1.00 1.02 GetImage XY 100x100 square
1.00 1.00 1.00 1.00 1.00 GetImage XY 500x500 square
1.00 1.00 1.01 0.98 0.95 X protocol NoOperation
1.02 1.03 1.04 1.03 1.00 QueryPointer
1.03 1.02 1.04 1.03 1.00 GetProperty 100.41 1.51 1.00 1.00 198.76 Change graphics context
45.81 1.00 0.99 0.97 57.10 Create and map subwindows (4 kids)
78.45 1.01 1.02 1.02 63.07 Create and map subwindows (16 kids)
73.91 1.01 1.00 1.00 56.37 Create and map subwindows (25 kids)
73.22 1.00 1.00 1.00 49.07 Create and map subwindows (50 kids)
72.36 1.01 0.99 1.00 32.14 Create and map subwindows (75 kids)
70.34 1.00 1.00 1.00 30.12 Create and map subwindows (100 kids)
55.00 1.00 1.00 0.99 23.75 Create and map subwindows (200 kids)
55.30 1.01 1.00 1.00 141.03 Create unmapped window (4 kids)
55.38 1.01 1.01 1.00 163.25 Create unmapped window (16 kids)
54.75 0.96 1.00 0.99 166.95 Create unmapped window (25 kids)
54.83 1.00 1.00 0.99 178.81 Create unmapped window (50 kids)
55.38 1.01 1.01 1.00 181.20 Create unmapped window (75 kids)
55.38 1.01 1.01 1.00 181.20 Create unmapped window (100 kids)
54.87 1.01 1.01 1.00 182.05 Create unmapped window (200 kids)
28.13 1.00 1.00 1.00 30.75 Map window via parent (4 kids)
36.14 1.01 1.01 1.01 32.58 Map window via parent (16 kids)
26.13 1.00 0.98 0.95 29.85 Map window via parent (25 kids)
40.07 1.00 1.01 1.00 27.57 Map window via parent (50 kids)
23.26 0.99 1.00 1.00 18.23 Map window via parent (75 kids)
22.91 0.99 1.00 0.99 16.52 Map window via parent (100 kids)
27.79 1.00 1.00 0.99 12.50 Map window via parent (200 kids)
22.35 1.00 1.00 1.00 56.19 Unmap window via parent (4 kids)
9.57 1.00 0.99 1.00 89.78 Unmap window via parent (16 kids)
80.77 1.01 1.00 1.00 103.85 Unmap window via parent (25 kids)
96.34 1.00 1.00 1.00 116.06 Unmap window via parent (50 kids)
99.72 1.00 1.00 1.00 124.93 Unmap window via parent (75 kids) 112.36 1.00 1.00 1.00 125.27 Unmap window via parent (100 kids) 105.41 1.00 1.00 0.99 120.00 Unmap window via parent (200 kids)
51.29 1.03 1.02 1.02 74.19 Destroy window via parent (4 kids)
86.75 0.99 0.99 0.99 116.87 Destroy window via parent (16 kids) 106.43 1.01 1.01 1.01 127.49 Destroy window via parent (25 kids) 120.34 1.01 1.01 1.00 140.11 Destroy window via parent (50 kids) 126.67 1.00 0.99 0.99 145.00 Destroy window via parent (75 kids) 126.11 1.01 1.01 1.00 140.56 Destroy window via parent (100 kids) 128.57 1.01 1.00 1.00 137.91 Destroy window via parent (200 kids)
16.04 0.88 1.00 1.00 20.36 Hide/expose window via popup (4 kids)
19.04 1.01 1.00 1.00 23.48 Hide/expose window via popup (16 kids)
19.22 1.00 1.00 1.00 20.44 Hide/expose window via popup (25 kids)
17.41 1.00 0.91 0.97 17.68 Hide/expose window via popup (50 kids)
17.29 1.01 1.00 1.01 17.07 Hide/expose window via popup (75 kids)
16.74 1.00 1.00 1.00 16.17 Hide/expose window via popup (100 kids)
10.30 1.00 1.00 1.00 10.51 Hide/expose window via popup (200 kids)
16.48 1.01 1.00 1.00 26.05 Move window (4 kids)
17.01 0.95 1.00 1.00 23.97 Move window (16 kids)
16.95 1.00 1.00 1.00 22.90 Move window (25 kids)
16.05 1.01 1.00 1.00 21.32 Move window (50 kids)
15.58 1.00 0.98 0.98 19.44 Move window (75 kids)
14.98 1.02 1.03 1.03 18.17 Move window (100 kids)
10.90 1.01 1.01 1.00 12.68 Move window (200 kids)
49.42 1.00 1.00 1.00 198.27 Moved unmapped window (4 kids)

50.72	0.97	1.00	1.00	193.66	Moved unmapped window (16 kids)
50.87	1.00	0.99	1.00	195.09	Moved unmapped window (25 kids)
50.72	1.00	1.00	1.00	189.34	Moved unmapped window (50 kids)
50.87	1.00	1.00	1.00	191.33	Moved unmapped window (75 kids)
50.87	1.00	1.00	0.90	186.71	Moved unmapped window (100 kids)
50.87	1.00	1.00	1.00	179.19	Moved unmapped window (200 kids)
41.04	1.00	1.00	1.00	56.61	Move window via parent (4 kids)
69.81	1.00	1.00	1.00	130.82	Move window via parent (16 kids)
95.81	1.00	1.00	1.00	141.92	Move window via parent (25 kids)
95.98	1.00	1.00	1.00	149.43	Move window via parent (50 kids)
96.59	1.01	1.01	1.00	153.98	Move window via parent (75 kids)
97.19	1.00	1.00	1.00	157.30	Move window via parent (100 kids)
96.67	1.00	0.99	0.96	159.44	Move window via parent (200 kids)
17.75	1.01	1.00	1.00	27.61	Resize window (4 kids)
17.94	1.00	1.00	0.99	25.42	Resize window (16 kids)
17.92	1.01	1.00	1.00	24.47	Resize window (25 kids)
17.24	0.97	1.00	1.00	24.14	Resize window (50 kids)
16.81	1.00	1.00	0.99	22.75	Resize window (75 kids)
16.08	1.00	1.00	1.00	21.20	Resize window (100 kids)
12.92	1.00	0.99	1.00	16.26	Resize window (200 kids)
52.94	1.01	1.00	1.00	327.12	Resize unmapped window (4 kids)
53.60	1.01	1.01	1.01	333.71	Resize unmapped window (16 kids)
52.99	1.00	1.00	1.00	337.29	Resize unmapped window (25 kids)
51.98	1.00	1.00	1.00	329.38	Resize unmapped window (50 kids)
53.05	0.89	1.00	1.00	322.60	Resize unmapped window (75 kids)
53.05	1.00	1.00	1.00	318.08	Resize unmapped window (100 kids)
53.11	1.00	1.00	0.99	306.21	Resize unmapped window (200 kids)
16.76	1.00	0.96	1.00	19.46	Circulate window (4 kids)
17.24	1.00	1.00	0.97	16.24	Circulate window (16 kids)
16.30	1.03	1.03	1.03	15.85	Circulate window (25 kids)
13.45	1.00	1.00	1.00	14.90	Circulate window (50 kids)
12.91	1.00	1.00	1.00	13.06	Circulate window (75 kids)
11.30	0.98	1.00	1.00	11.03	Circulate window (100 kids)
7.58	1.01	1.01	0.99	7.47	Circulate window (200 kids)
1.01	1.01	0.98	1.00	0.95	Circulate Unmapped window (4 kids)
1.07	1.07	1.01	1.07	1.02	Circulate Unmapped window (16 kids)
1.04	1.09	1.06	1.05	0.97	Circulate Unmapped window (25 kids)
1.04	1.23	1.20	1.18	1.05	Circulate Unmapped window (50 kids)
1.18	1.53	1.19	1.45	1.24	Circulate Unmapped window (75 kids)
1.08	1.02	1.01	1.74	1.01	Circulate Unmapped window (100 kids)
1.01	1.12	0.98	0.91	0.97	Circulate Unmapped window (200 kids)

E.2.8 Profiling with OProfile

OProfile (available from http://oprofile.sourceforge.net/) is a system-wide profiler for Linux systems that uses processor-level counters to collect sampling data. OProfile can provide information that is similar to that provided by gprof, but without the necessity of recompiling the program with special instrumentation (i.e., OProfile can collect statistical profiling information about optimized programs). A test harness was developed to collect OProfile data for each x11perf test individually.

Test runs were performed using the RETIRED_INSNS counter on the AMD Athlon and the CPU_CLK_HALTED counter on the Intel Pentium III (with a test configuration different from the one described above). We have examined OProfile output and have compared it with gprof output. This investigation has not produced results that yield performance increases in x11perf numbers.

E.2.9 X Test Suite

The X Test Suite was run on the fully optimized DMX server using the configuration described above. The following failures were noted: XListPixmapFormats: Test 1 [1] XChangeWindowAttributes: Test 32 [1] XCreateWindow: Test 30 [1] XFreeColors: Test 4 [3] XCopyArea: Test 13, 17, 21, 25, 30 [2] XCopyPlane: Test 11, 15, 27, 31 [2] XSetFontPath: Test 4 [1] XChangeKeyboardControl: Test 9, 10 [1]

[1] Previously documented errors expected from the Xinerama
implementation (see Phase I discussion). [2] Newly noted errors that have been verified as expected
behavior of the Xinerama implementation. [3] Newly noted error that has been verified as a Xinerama
implementation bug.

E.3 Phase III

During the third phase of development, support was provided for the following extensions: SHAPE, RENDER, XKEYBOARD, XInput.

E.3.1 SHAPE

The SHAPE extension is supported. Test applications (e.g., xeyes and oclock) and window managers that make use of the SHAPE extension will work as expected.

E.3.2 RENDER

The RENDER extension is supported. The version included in the DMX CVS tree is version 0.2, and this version is fully supported by Xdmx. Applications using only version 0.2 functions will work correctly; however, some apps that make use of functions from later versions do not properly check the extension's major/minor version numbers. These apps will fail with a Bad Implementation error when using post-version 0.2 functions. This is expected behavior. When the DMX CVS tree is updated to include newer versions of RENDER, support for these newer functions will be added to the DMX X server.

E.3.3 XKEYBOARD

The XKEYBOARD extension is supported. If present on the back-end X servers, the XKEYBOARD extension will be used to obtain information about the type of the keyboard for initialization. Otherwise, the keyboard will be initialized using defaults. Note that this departs from older behavior: when Xdmx is compiled without XKEYBOARD support, the map from the back-end X server will be preserved. With XKEYBOARD support, the map is not preserved because better information and control of the keyboard is available.

E.3.4 XInput

The XInput extension is supported. Any device can be used as a core device and be used as an XInput extension device, with the exception of core devices on the back-end servers. This limitation is present because cursor handling on the back-end requires that the back-end cursor sometimes track the Xdmx core cursor -- behavior that is incompatible with using the back-end pointer as a non-core device.

Currently, back-end extension devices are not available as Xdmx extension devices, but this limitation should be removed in the future.

To demonstrate the XInput extension, and to provide more examples for low-level input device driver writers, USB device drivers have been written for mice (usb-mou), keyboards (usb-kbd), and non-mouse/non-keyboard USB devices (usb-oth). Please see the man page for information on Linux kernel drivers that are required for using these Xdmx drivers.

E.3.5 DPMS

The DPMS extension is exported but does not do anything at this time.

E.3.6 Other Extensions

The LBX, SECURITY, XC-APPGROUP, and XFree86-Bigfont extensions do not require any special Xdmx support and have been exported.

The BIG-REQUESTS, DEC-XTRAP, DOUBLE-BUFFER, Extended-Visual-Information, FontCache, GLX, MIT-SCREEN-SAVER, MIT-SHM, MIT-SUNDRY-NONSTANDARD, RECORD, SECURITY, SGI-GLX, SYNC, TOG-CUP, X-Resource, XC-MISC, XFree86-DGA, XFree86-DRI, XFree86-Misc, XFree86-VidModeExtension, and XVideo extensions are not supported at this time, but will be evaluated for inclusion in future DMX releases. See below for additional work on extensions after Phase III.

E.4 Phase IV

E.4.1 Moving to XFree86 4.3.0

For Phase IV, the recent release of XFree86 4.3.0 (27 February 2003) was merged onto the dmx.sourceforge.net CVS trunk and all work is proceeding using this tree.

E.4.2 Extensions

E.4.2.1 XC-MISC (supported)

XC-MISC is used internally by the X library to recycle XIDs from the X server. This is important for long-running X server sessions. Xdmx supports this extension. The X Test Suite passed and failed the exact same tests before and after this extension was enabled.

E.4.2.2 Extended-Visual-Information (supported)

The Extended-Visual-Information extension provides a method for an X client to obtain detailed visual information. Xdmx supports this extension. It was tested using the hw/dmx/examples/evi example program. Note that this extension is not Xinerama-aware -- it will return visual information for each screen even though Xinerama is causing the X server to export a single logical screen.

E.4.2.3 RES (supported)

The X-Resource extension provides a mechanism for a client to obtain detailed information about the resources used by other clients. This extension was tested with the hw/dmx/examples/res program. The X Test Suite passed and failed the exact same tests before and after this extension was enabled.

E.4.2.4 BIG-REQUESTS (supported)

This extension enables the X11 protocol to handle requests longer than 262140 bytes. The X Test Suite passed and failed the exact same tests before and after this extension was enabled.

E.4.2.5 XSYNC (supported)

This extension provides facilities for two different X clients to synchronize their requests. This extension was minimally tested with xdpyinfo and the X Test Suite passed and failed the exact same tests before and after this extension was enabled.

E.4.2.6 XTEST, RECORD, DEC-XTRAP (supported) and XTestExtension1 (not supported)

The XTEST and RECORD extension were developed by the X Consortium for use in the X Test Suite and are supported as a standard in the X11R6 tree. They are also supported in Xdmx. When X Test Suite tests that make use of the XTEST extension are run, Xdmx passes and fails exactly the same tests as does a standard XFree86 X server. When the rcrdtest test (a part of the X Test Suite that verifies the RECORD extension) is run, Xdmx passes and fails exactly the same tests as does a standard XFree86 X server.

There are two older XTEST-like extensions: DEC-XTRAP and XTestExtension1. The XTestExtension1 extension was developed for use by the X Testing Consortium for use with a test suite that eventually became (part of?) the X Test Suite. Unlike XTEST, which only allows events to be sent to the server, the XTestExtension1 extension also allowed events to be recorded (similar to the RECORD extension). The second is the DEC-XTRAP extension that was developed by the Digital Equipment Corporation.

The DEC-XTRAP extension is available from Xdmx and has been tested with the xtrap* tools which are distributed as standard X11R6 clients.

The XTestExtension1 is not supported because it does not appear to be used by any modern X clients (the few that support it also support XTEST) and because there are no good methods available for testing that it functions correctly (unlike XTEST and DEC-XTRAP, the code for XTestExtension1 is not part of the standard X server source tree, so additional testing is important).

Most of these extensions are documented in the X11R6 source tree. Further, several original papers exist that this author was unable to locate -- for completeness and historical interest, citations are provide:

XRECORDMartha Zimet. Extending X For Recording. 8th Annual X Technical Conference Boston, MA January 24-26, 1994.DEC-XTRAPDick Annicchiarico, Robert Chesler, Alan Jamison. XTrap Architecture. Digital Equipment Corporation, July 1991.XTestExtension1Larry Woestman. X11 Input Synthesis Extension Proposal. Hewlett Packard, November 1991.

E.4.2.7 MIT-MISC (not supported)

The MIT-MISC extension is used to control a bug-compatibility flag that provides compatibility with xterm programs from X11R1 and X11R2. There does not appear to be a single client available that makes use of this extension and there is not way to verify that it works correctly. The Xdmx server does not support MIT-MISC.

E.4.2.8 SCREENSAVER (not supported)

This extension provides special support for the X screen saver. It was tested with beforelight, which appears to be the only client that works with it. When Xinerama was not active, beforelight behaved as expected. However, when Xinerama was active, beforelight did not behave as expected. Further, when this extension is not active, xscreensaver (a widely-used X screen saver program) did not behave as expected. Since this extension is not Xinerama-aware and is not commonly used with expected results by clients, we have left this extension disabled at this time.

E.4.2.9 GLX (supported)

The GLX extension provides OpenGL and GLX windowing support. In Xdmx, the extension is called glxProxy, and it is Xinerama aware. It works by either feeding requests forward through Xdmx to each of the back-end servers or handling them locally. All rendering requests are handled on the back-end X servers. This code was donated to the DMX project by SGI. For the X Test Suite results comparison, see below.

E.4.2.10 RENDER (supported)

The X Rendering Extension (RENDER) provides support for digital image composition. Geometric and text rendering are supported. RENDER is partially Xinerama-aware, with text and the most basic compositing operator; however, its higher level primitives (triangles, triangle strips, and triangle fans) are not yet Xinerama-aware. The RENDER extension is still under development, and is currently at version 0.8. Additional support will be required in DMX as more primitives and/or requests are added to the extension.

There is currently no test suite for the X Rendering Extension; however, there has been discussion of developing a test suite as the extension matures. When that test suite becomes available, additional testing can be performed with Xdmx. The X Test Suite passed and failed the exact same tests before and after this extension was enabled.

E.4.2.11 Summary

To summarize, the following extensions are currently supported: BIG-REQUESTS, DEC-XTRAP, DMX, DPMS, Extended-Visual-Information, GLX, LBX, RECORD, RENDER, SECURITY, SHAPE, SYNC, X-Resource, XC-APPGROUP, XC-MISC, XFree86-Bigfont, XINERAMA, XInputExtension, XKEYBOARD, and XTEST.

The following extensions are not supported at this time: DOUBLE-BUFFER, FontCache, MIT-SCREEN-SAVER, MIT-SHM, MIT-SUNDRY-NONSTANDARD, TOG-CUP, XFree86-DGA, XFree86-Misc, XFree86-VidModeExtension, XTestExtensionExt1, and XVideo.

E.4.3 Additional Testing with the X Test Suite

E.4.3.1 XFree86 without XTEST

After the release of XFree86 4.3.0, we retested the XFree86 X server with and without using the XTEST extension. When the XTEST extension was not used for testing, the XFree86 4.3.0 server running on our usual test system with a Radeon VE card reported unexpected failures in the following tests: XListPixmapFormats: Test 1 XChangeKeyboardControl: Tests 9, 10 XGetDefault: Test 5 XRebindKeysym: Test 1

E.4.3.2 XFree86 with XTEST

When using the XTEST extension, the XFree86 4.3.0 server reported the following errors: XListPixmapFormats: Test 1 XChangeKeyboardControl: Tests 9, 10 XGetDefault: Test 5 XRebindKeysym: Test 1

XAllowEvents: Tests 20, 21, 24 XGrabButton: Tests 5, 9-12, 14, 16, 19, 21-25 XGrabKey: Test 8 XSetPointerMapping: Test 3 XUngrabButton: Test 4

While these errors may be important, they will probably be fixed eventually in the XFree86 source tree. We are particularly interested in demonstrating that the Xdmx server does not introduce additional failures that are not known Xinerama failures.

E.4.3.3 Xdmx with XTEST, without Xinerama, without GLX

Without Xinerama, but using the XTEST extension, the following errors were reported from Xdmx (note that these are the same as for the XFree86 4.3.0, except that XGetDefault no longer fails): XListPixmapFormats: Test 1 XChangeKeyboardControl: Tests 9, 10 XRebindKeysym: Test 1

XAllowEvents: Tests 20, 21, 24 XGrabButton: Tests 5, 9-12, 14, 16, 19, 21-25 XGrabKey: Test 8 XSetPointerMapping: Test 3 XUngrabButton: Test 4

E.4.3.4 Xdmx with XTEST, with Xinerama, without GLX

With Xinerama, using the XTEST extension, the following errors were reported from Xdmx: XListPixmapFormats: Test 1 XChangeKeyboardControl: Tests 9, 10 XRebindKeysym: Test 1

XAllowEvents: Tests 20, 21, 24 XGrabButton: Tests 5, 9-12, 14, 16, 19, 21-25 XGrabKey: Test 8 XSetPointerMapping: Test 3 XUngrabButton: Test 4

XCopyPlane: Tests 13, 22, 31 (well-known XTEST/Xinerama interaction issue) XDrawLine: Test 67 XDrawLines: Test 91 XDrawSegments: Test 68 Note that the first two sets of errors are the same as for the XFree86 4.3.0 server, and that the XCopyPlane error is a well-known error resulting from an XTEST/Xinerama interaction when the request crosses a screen boundary. The XDraw* errors are resolved when the tests are run individually and they do not cross a screen boundary. We will investigate these errors further to determine their cause.

E.4.3.5 Xdmx with XTEST, with Xinerama, with GLX

With GLX enabled, using the XTEST extension, the following errors were reported from Xdmx (these results are from early during the Phase IV development, but were confirmed with a late Phase IV snapshot): XListPixmapFormats: Test 1 XChangeKeyboardControl: Tests 9, 10 XRebindKeysym: Test 1

XAllowEvents: Tests 20, 21, 24 XGrabButton: Tests 5, 9-12, 14, 16, 19, 21-25 XGrabKey: Test 8 XSetPointerMapping: Test 3 XUngrabButton: Test 4

XClearArea: Test 8 XCopyArea: Tests 4, 5, 11, 14, 17, 23, 25, 27, 30 XCopyPlane: Tests 6, 7, 10, 19, 22, 31 XDrawArcs: Tests 89, 100, 102 XDrawLine: Test 67 XDrawSegments: Test 68 Note that the first two sets of errors are the same as for the XFree86 4.3.0 server, and that the third set has different failures than when Xdmx does not include GLX support. Since the GLX extension adds new visuals to support GLX's visual configs and the X Test Suite runs tests over the entire set of visuals, additional rendering tests were run and presumably more of them crossed a screen boundary. This conclusion is supported by the fact that nearly all of the rendering errors reported are resolved when the tests are run individually and they do no cross a screen boundary.

Further, when hardware rendering is disabled on the back-end displays, many of the errors in the third set are eliminated, leaving only: XClearArea: Test 8 XCopyArea: Test 4, 5, 11, 14, 17, 23, 25, 27, 30 XCopyPlane: Test 6, 7, 10, 19, 22, 31

E.4.3.6 Conclusion

We conclude that all of the X Test Suite errors reported for Xdmx are the result of errors in the back-end X server or the Xinerama implementation. Further, all of these errors that can be reasonably fixed at the Xdmx layer have been. (Where appropriate, we have submitted patches to the XFree86 and Xinerama upstream maintainers.)

E.4.4 Dynamic Reconfiguration

During this development phase, dynamic reconfiguration support was added to DMX. This support allows an application to change the position and offset of a back-end server's screen. For example, if the application would like to shift a screen slightly to the left, it could query Xdmx for the screen's <x,y> position and then dynamically reconfigure that screen to be at position <x+10,y>. When a screen is dynamically reconfigured, input handling and a screen's root window dimensions are adjusted as needed. These adjustments are transparent to the user.

E.4.4.1 Dynamic reconfiguration extension

The application interface to DMX's dynamic reconfiguration is through a function in the DMX extension library: Bool DMXReconfigureScreen(Display *dpy, int screen, int x, int y) where dpy is DMX server's display, screen is the number of the screen to be reconfigured, and x and y are the new upper, left-hand coordinates of the screen to be reconfigured.

The coordinates are not limited other than as required by the X protocol, which limits all coordinates to a signed 16 bit number. In addition, all coordinates within a screen must also be legal values. Therefore, setting a screen's upper, left-hand coordinates such that the right or bottom edges of the screen is greater than 32,767 is illegal.

E.4.4.2 Bounding box

When the Xdmx server is started, a bounding box is calculated from the screens' layout given either on the command line or in the configuration file. This bounding box is currently fixed for the lifetime of the Xdmx server.

While it is possible to move a screen outside of the bounding box, it is currently not possible to change the dimensions of the bounding box. For example, it is possible to specify coordinates of <-100,-100> for the upper, left-hand corner of the bounding box, which was previously at coordinates <0,0>. As expected, the screen is moved down and to the right; however, since the bounding box is fixed, the left side and upper portions of the screen exposed by the reconfiguration are no longer accessible on that screen. Those inaccessible regions are filled with black.

This fixed bounding box limitation will be addressed in a future development phase.

E.4.4.3 Sample applications

An example of where this extension is useful is in setting up a video wall. It is not always possible to get everything perfectly aligned, and sometimes the positions are changed (e.g., someone might bump into a projector). Instead of physically moving projectors or monitors, it is now possible to adjust the positions of the back-end server's screens using the dynamic reconfiguration support in DMX.

Other applications, such as automatic setup and calibration tools, can make use of dynamic reconfiguration to correct for projector alignment problems, as long as the projectors are still arranged rectilinearly. Horizontal and vertical keystone correction could be applied to projectors to correct for non-rectilinear alignment problems; however, this must be done external to Xdmx.

A sample test program is included in the DMX server's examples directory to demonstrate the interface and how an application might use dynamic reconfiguration. See dmxreconfig.c for details.

E.4.4.4 Additional notes

In the original development plan, Phase IV was primarily devoted to adding OpenGL support to DMX; however, SGI became interested in the DMX project and developed code to support OpenGL/GLX. This code was later donated to the DMX project and integrated into the DMX code base, which freed the DMX developers to concentrate on dynamic reconfiguration (as described above).

E.4.5 Doxygen documentation

Doxygen is an open-source (GPL) documentation system for generating browseable documentation from stylized comments in the source code. We have placed all of the Xdmx server and DMX protocol source code files under Doxygen so that comprehensive documentation for the Xdmx source code is available in an easily browseable format.

E.4.6 Valgrind

Valgrind, an open-source (GPL) memory debugger for Linux, was used to search for memory management errors. Several memory leaks were detected and repaired. The following errors were not addressed:

1. When the X11 transport layer sends a reply to the client, only those fields that are required by the protocol are filled in -- unused fields are left as uninitialized memory and are therefore noted by valgrind. These instances are not errors and were not repaired.

2. At each server generation, glxInitVisuals allocates memory that is never freed. The amount of memory lost each generation approximately equal to 128 bytes for each back-end visual. Because the code involved is automatically generated, this bug has not been

fixed and will be referred to SGI.

3. At each server generation, dmxRealizeFont calls XLoadQueryFont, which allocates a font structure that is not freed. dmxUnrealizeFont can free the font structure for the first screen, but cannot free it for the other screens since they are already closed by the time dmxUnrealizeFont could free them. The amount of memory lost each generation is approximately equal to 80 bytes per font per back-end. When this bug is fixed in the the X server's device-independent (dix) code, DMX will be able to properly free the memory allocated by XLoadQueryFont.

E.4.7 RATS

RATS (Rough Auditing Tool for Security) is an open-source (GPL) security analysis tool that scans source code for common security-related programming errors (e.g., buffer overflows and TOCTOU races). RATS was used to audit all of the code in the hw/dmx directory and all "High" notations were checked manually. The code was either re-written to eliminate the warning, or a comment containing "RATS" was inserted on the line to indicate that a human had checked the code. Unrepaired warnings are as follows:

1. Fixed-size buffers are used in many areas, but code has been added to protect against buffer overflows (e.g., XmuSnprint). The only instances that have not yet been fixed are in config/xdmxconfig.c (which is not part of the Xdmx server) and input/usb-common.c.

2. vprintf and vfprintf are used in the logging routines. In general, all uses of these functions (e.g., dmxLog) provide a constant format string from a trusted source, so the use is relatively benign.

3. glxProxy/glxscreens.c uses getenv and strcat. The use of these functions is safe and will remain safe as long as ExtensionsString is longer then GLXServerExtensions (ensuring this may not be ovious to the casual programmer, but this is in automatically generated code, so we hope that the generator enforces this constraint).

CONTENTS

1. Introduction 1
   1.1 The Distributed Multihead X Server 1
   1.2 Layout of Paper 2

2. Development plan 2
   2.1 Bootstrap code 2
   2.2 Input device handling 3
   2.3 Output device handling 4
   2.4 Optimizing DMX 6
   2.5 DMX X extension support 7
   2.6 Common X extension support 7
   2.7 OpenGL support 7

3. Current issues 8
   3.1 Fonts 8
   3.2 Zero width rendering primitives 8
   3.3 Output scaling 8
   3.4 Per-screen colormaps 8

D. Background 9
   D.1 Core input device handling 9
   D.2 Output handling 12
   D.3 Xinerama 15

E. Development Results 17
   E.1 Phase I 17
   E.2 Phase II 20
   E.3 Phase III 30
   E.4 Phase IV 31