Published in the
January/February 1993 issue of *The X Journal*.

Going Beyond the MIT Sample Server: The Silicon Graphics X11 Server

Mark J. Kilgard
Silicon Graphics Inc.
Revision : 1.20

April 21, 1993

Abstract

The MIT X11 Sample Server is the starting point for nearly all X11 server implementations. Most server vendors add value beyond the sample server. Silicon Graphics has done extensive work to enhance the performance and functionality of its X server implementation. The server supports X across Silicon Graphics' entire line of high-performance graphics hardware. This article describes six important areas of enhancement made to the Silicon Graphics server: integration with the IRIS GL graphics library, a high performance input subsystem, the non-frame buffer porting layer, support for specific hardware features, the dynamic linking of hardware support, and the Display PostScript extension.

1 Introduction

Nearly all X11 server implementations use the MIT X11 Sample Server as a starting point. The sample server is a mature, reasonably portable implementation of an X11 server written in C and extensively tuned for efficient and fast performance.

While many X users do in fact run the MIT Sample Server as their actual X server, most vendors do distribute their own modified version of the sample server. Indeed, this is a primary purpose for the MIT Sample Server. Vendors modify the sample server for numerous reasons including:

- The vendor's particular hardware and operating system are not supported by the MIT Sample Server.[1]
- The vendor wishes to support an optimized version of the server for its proprietary graphics hardware.
- The vendor wishes to integrate X with other non-X based graphics systems.
- The vendor desires to add enhancements to the server through X extensions or other server modifications.

In the case of Silicon Graphics, all these reasons apply.

The purpose of this article is to describe some innovative ways in which the Silicon Graphics X server goes beyond the MIT Sample Server. The hope is such a description will raise the expectations users have for the quality, performance, and functionality of production X11 servers. This article serves as a "tour" of various additions Silicon Graphics has made to the core MIT Sample Server. While many features and modifications have been added to the Silicon Graphics X server, certain improvements stand out as important features worth examining. These features are:

- The X Window System is integrated with the IRIS GL graphics library. The X server plays a critical role in allocating and arbitrating access to display resources. This is what makes possible the seamless interleaving of X windows and GL windows performing high-performance 3D rendering.
- The X server manages an advanced input subsystem including a shared memory input queue and kernel supported hardware cursor tracking.
- The Non-Frame Buffer porting layer provides a machine-independent way to take advantage of high performance features of non-frame buffer graphics hardware.
- The dynamic linking of DDXs (the device dependent part of the server) allows smaller server executables and the ability to support new graphics hardware by only supplying a new DDX.

[1] It should be noted that MIT's X11 Release 5 distribution *does* have client and library support for Silicon Graphics workstations.

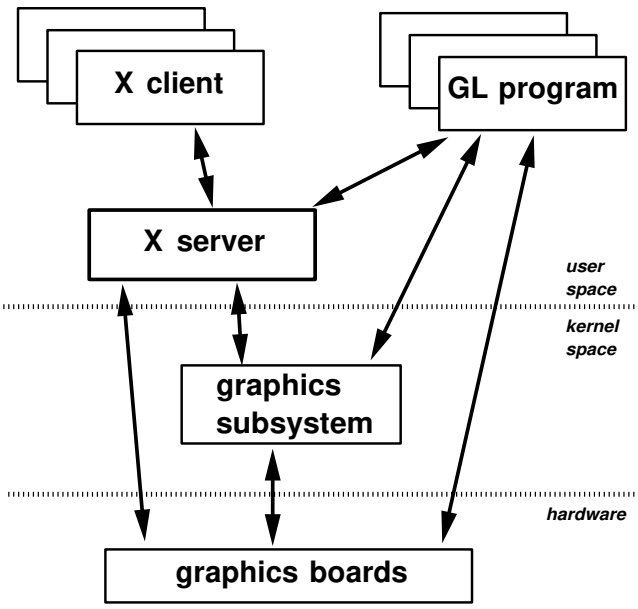


Figure 1: How the X server and GL and X programs "talk" to graphics resources and each other.

- The server is well tuned to graphics capabilities of Silicon Graphics hardware. Cross hair cursors, multiple hardware colormaps, and overlay plane support are some examples.
- Adobe's Display PostScript extension is available.

Together these enhancements provide a feature-rich production-quality X server which greatly extends the capabilities of the MIT Sample Server upon which it is based.

Certainly, many other vendors offer features with the similar capabilities. The intent of this article is neither to hype the Silicon Graphics X server nor be an extensive review of the capabilities of production X servers. Instead the intent is to make X users aware of server features available today which will in all likelihood become common in the future. Vaporware and theoretical features are left to marketing brochures and academic papers.

The Silicon Graphics X server is named `Xsgi` (after the command used to invoke it) and will be referred to as such through the remainder of this article. SGI is an abbreviation for Silicon Graphics Incorporated. The specific server discussed is currently available in IRIX 4.0.x. IRIX is the SGI version of Unix.

2 Integration with the IRIS GL

Any user of Silicon Graphics workstations is familiar with the 3D capabilities of the machines. All SGI machines support a high-performance 3D rendering library known as the IRIS GL (which stands for Graphics Library). The GL Application Programmer Interface (API) is not dependent on a particular windowing system. In fact, before SGI standardized on the X Window System, SGI implemented the GL for the NeWS window system. When SGI made the transition to X, the GL needed to be reimplemented for use with X.

The current IRIS GL 4.0[2] is implemented through a system called *virtualized direct access rendering* meaning that each GL program sees and communicates directly with the graphics hardware of the machine. The GL library hides the device dependencies of talking directly to the hardware.

Using direct access rendering is in contrast to the philosophy of X where programs talk to the graphics hardware only indirectly via X protocol requests to the X server. But direct access does make possible the extremely fast 3D rendering available on SGI machines. Figure 1 shows how X clients and GL programs access the graphics resources they need. Notice that GL programs do talk directly to the X server via an X protocol connection. The GL library communicates to the X server but this is totally hidden from the GL programmer. The X server is used by GL programs for non-rendering tasks such as window creation and input delivery. So in a strict sense, GL programs *are* X clients but are distinguished for the purpose of this discussion.[3]

Since the X server wants to control the real estate on the screen, GL programs are forced to coordinate how they use graphics resources with the X server. This is the major challenge of seamlessly integrating GL with X. The current `Xsgi` achieves this quite admirably. Excepting the difference in quality of rendering for GL windows and the speed of some window management operations, GL windows are otherwise indistinguishable to the naive user.

So how is this accomplished? The `Xsgi` had to be substantially enhanced to support GL. Also the operating system kernel and GL libraries required major modifications to achieve seamless GL integration.

2.1 SGI Style Graphics Hardware

Understanding how SGI supports the GL requires a more detailed explanation of the type of graphics hardware SGI develops. The major way SGI graphics hardware differs from its PC and low-end workstation counterparts is that the *frame buffer* is not directly available for manipulation.

SGI hardware *does* contain a frame buffer but it is manipulated by sending rendering commands through a memory bank of registers to the graphics pipeline. The hardware then performs the specified requests. This ap-

[2]The new OpenGL will also be available as an X extension but this article focuses on the currently available GL. See Karlton [6] for information about the OpenGL.

[3]SGI does support a mixed-model which allows a programmer to use both X and GL in the same program but programmers must deal with added programming complexity.

proach is well-suited to the type of highly-pipelined, high-performance graphics subsystems developed by SGI.[4]

In virtualized direct access rendering, each program desiring direct graphics hardware access requests from the operating system a *rendering node* which consists of a virtualized bank of the graphics registers mapped into the program's address space and the necessary kernel maintained state. Each program manipulates the registers as if it were the only one using the graphics subsystem. Before using its rendering node, the program binds the node to an X window. The operating system via virtual memory takes the responsibility of virtualizing access to the graphics registers. This is done by only *actually* mapping the registers to one program at a time. Other programs using the graphics registers have invalid virtual memory mappings to the registers.

When a second program tries to access the registers, the registers are not actually mapped into its memory. The operating system detects this memory access exception, saves the graphics context of the currently executing graphics program, restores the graphics context of the program now wishing to use the graphics hardware, and restarts the second program. This whole process is transparent to the programs accessing the graphics hardware. The idea is very similar to techniques used by demand paged virtual memory to allow the appearance of more physical memory than a system actually has.

SGI graphics systems are also capable of hardware double buffering. Double buffering allows images to be generated in a frame by frame style very conducive to computer animation. Double buffering can hide image generation from user sight and avoid other visual artifacts such as flicker. The GL supports rendering into double buffered windows as well as normal single buffered windows.

Double buffer hardware works by splitting the frame buffer's bitplanes in half. The video controller selects pixels to be displayed from whichever buffer is considered the *front* one. The hardware's sense of front and back can instantaneously be reversed. This operation is called a *buffer swap*.

A CRT display performs a *vertical retrace* when the electron guns must be reset to the top corner of the screen after each screen refresh (typically around 60 times per second). If the buffer swap can be synchronized to happen during the vertical retrace, a flicker-free buffer swap can be achieved. SGI hardware interrupts the CPU during a vertical retrace for this reason.

Buffer selection needs to be supported on a per-pixel basis so windows of arbitrary shape can be buffer swapped. The buffering status of a given pixel depends on its *display identifier* (DID). By properly *painting* the DIDs to reflect window layout, buffer swaps can be performed on a per-window basis.

[4] Readers interested advanced graphics subsystems are referred to the "Advanced Raster Graphics Architecture" chapter of Foley [3] which includes a discussion of SGI's GTX hardware.

2.2 The Rendering Resource Manager

Virtualized direct access rendering is managed by a piece of the IRIX operating system known as the Rendering Resource Manager (RRM). The RRM is also responsible for maintaining window clipping and ensuring properly timed buffer swaps for double buffered windows.

Window clipping can not be done wholely in the kernel because only the X server has the centralized knowledge about the window hierarchy and window layout. For this reason, the RRM treats the X server process in a special way. The X server registers itself with RRM as the *board manager*. The board manager is informed about how rendering nodes are associated with X windows.

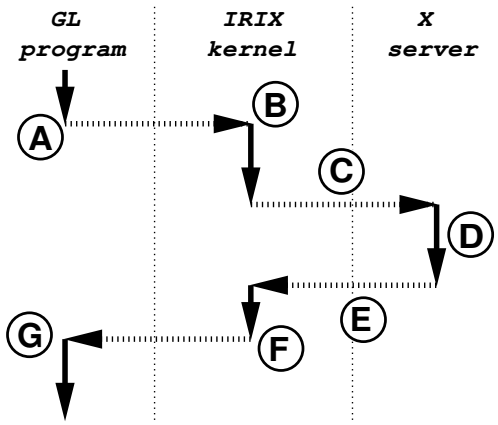
When the window layout changes (for example by a window resize), the X server can inform the kernel that the clip of a rendering node is invalid if the clip of its associated window changes. A rendering node with an invalid clip has its bank of graphics registers mapped as invalid and the node is marked as having an invalid clip. The next time the node is used, the kernel will trap the access and inform the X server that the rendering node's clip should be validated. The X server determines the rendering node's proper clip and performs a special `ioctl` reserved for the board manager to revalidate the node's clip. Once revalidated, the process using the node is restarted. The clip validation is transparent to the process using the rendering node. Figure 2 outlines how a clip validation would take place.

To make the clipping scheme work, rendering nodes must be able to have their window clip updated without any program knowledge of the change. This is possible because arbitrary hardware clipping is available and all GL rendering is window relative.

The X server must also become involved in performing some buffer swaps. The number of DIDs a piece of hardware supports is limited. This means DIDs sometimes need to be shared between windows. But when a double buffered window needs to perform a buffer swap, it needs to have an unshared DID. The X server manages DID allocation. Reallocation of DIDs requires the server to *repaint* the DIDs for the screen.

Normally when a GL program wants to swap the buffer of its window, it does a special buffer swap `ioctl`. This requests the kernel to schedule a buffer swap operation for the window's DID during the next vertical retrace interrupt. But if the window's DID is currently shared, the kernel must request the window be reallocated to use an unshared DID. The kernel communicates with `Xsgi` as in the clip validation case. Once a rendering node is using an unshared DID, the buffer swap can be scheduled.

This discussion has been simplified by limiting it to how RRM interacts with the X server. It is important to note



A) GL program with invalid clip resource faults when accessing graphics registers.
B) Kernel trap handler determines fault caused by invalid clip for the rendering node.
C) Message put in shmiq telling X server to validate the rendering node's clip.
D) X server generates a clip list for the rendering node's window.
E) X server performs ioctl to inform kernel of new valid clip list.
F) Kernel updates the rendering node to reflect its new clip, validates the node's clip resource, maps in the graphics registers, and restarts the program where it stopped.

Figure 2: RRM clip validation assisted by the X server.

that the GL completely hides all knowledge of RRM related issues from the GL programmer.

Despite the complexity of the RRM, the system achieves quite amazing performance. In part, this is because resources such as window clipping are only validated when they must be and the need for such validations is relatively rare when compared to the amount of graphics rendering commands being generated.

SGI is not the only company to have integrated the GL into the X Window System. IBM also supports GL in certain RISC System/6000 platforms. Their approach is described in Haletky [4] and Tucker [9].

3 The Input Subsystem

While X mandates that every X server support a keyboard and mouse, there is no standard system interface for accessing such devices on Unix systems. This means each vendor has its own input subsystem for its X server. SGI's input subsystem not only meets the basic requirement to support a keyboard and mouse but also has the following features:

- A shared memory input queue is supported for high-performance.
- A wide variety of input devices are supported, including 3D devices such as the Spaceball. The X Input extension is used to support such devices.
- Input devices are supported abstractly; knowledge of specific input devices is isolated to modular kernel device drivers.
- Hardware cursor tracking is supported in the kernel.

As will be seen, these features provide a more functional, more responsive input subsystem than that available in the MIT Sample Server.

3.1 Shared Memory Input Queue Support

A shared memory input queue (called a shmiq by SGI and pronounced *shmick*) is a fast way of receiving input device events by eliminating the operating system overhead to read input devices. Instead of reading the input devices through Unix file descriptors, the operating system deposits input events directly into a region of the server's address space. The region is organized as a ring buffer. A head and tail variable are used to tell what is available in the buffer. If the head and tail are equal the buffer is empty. The operating system updates the head as events are deposited into the ring buffer while the server updates the tail as events are read out.

A hook for supporting the shmiq is provided by MIT's device independent X code (DIX). It allows a vendor to supply a pointer to two words of memory by calling a routine named `SetInputCheck`. These words can be used as the head and tail of a shmiq. This interface is described in Angebranndt [2]. A test exists in the `WaitForSomething` routine in the server's operating system code (OS). It compares the values pointed to by the passed parameters to `SetInputCheck`; if they are not equal, input is assumed to be available and `ProcessInputEvents` is called to read the input out of the shmiq.

By default, the DIX code registers pointers to unequal variables. In this way, unless a server explicitly calls `SetInputCheck`, the support for a shmiq has no effect.

The ability of an X server to use a shmiq is dependent on whether the operating system implements the appropriate user/kernel interface. The IRIX shmiq device driver is implemented as a STREAMS multiplexor. STREAMS is a flexible device driver interface developed by AT&T [1]. The shmiq driver is implemented this way becauses it allows an arbitrary number of input sources to be linked to it so all input sources are *funneled* through the shmiq. Figure 3 how the shmiq interacts with other system components.

3.2 The IDEV Interface

The input devices can be supported by implementing STREAMS modules which translate the raw device input into abstract events which will be sent to the shmiq driver (and on to the server). This interface is known as IDEV. The shmiq driver expects messages from the input devices

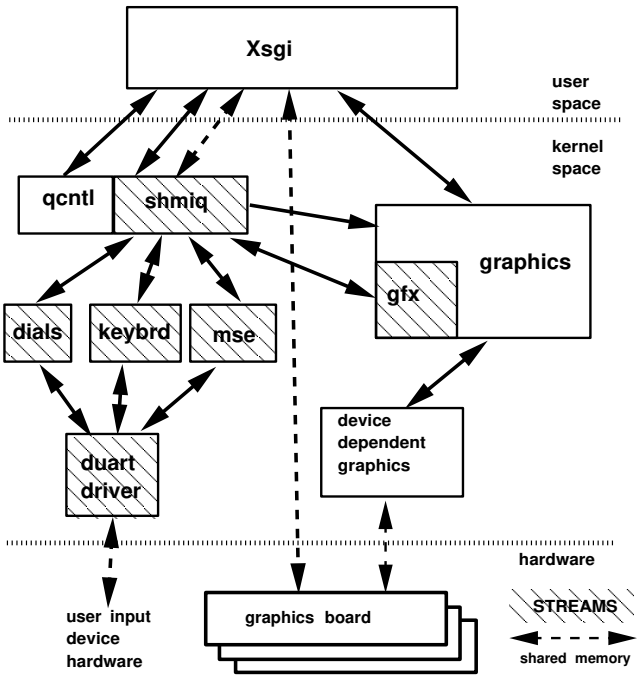


Figure 3: How the shmiq interacts with input devices, graphics, and `Xsgi`. (The qcntl driver is an implementation wart which allows `Xsgi` to wait via the `select` system call on shmiq input.)

to be in the form of IDEV events. IDEV device events appear as valuator, button, and pointer state changes. Along with a uniform set of events, `Xsgi` can send down a standard set of abstract commands to control the various input devices.

This allows the server to see input devices as abstract input sources and does not require special server code to be written every time a new input device is supported. Instead, device specific knowledge of input devices is encapsulated in an IDEV-based STREAMS module linked into the kernel.

IDEV modules currently exist for the SGI mouse and keyboard, a dial box, a Spaceball, and graphics tablets. Prototypes have even been implemented to support MIDI based keyboards and musical instruments!

3.3 Kernel Cursor Tracking

Figure 3 also shows the shmiq driver associated with the kernel graphics drivers. Two points should be made. First, the shmiq driver is not only used for user input devices but is also the mechanism used to receive RRM events when GL programs need resources validated. The second reason is because the on-screen cursor is actually positioned by the shmiq driver. This allows very smooth cursor movement since the cursor is updated independently of how busy the X server might be. X servers without this feature require the server to reposition the cursor each time the pointer is moved.

Without a hardware cursor, cursor positioning is particularly inefficient because it requires the cursor to be undrawn, moved, and redrawn each time the cursor moves. It also means the cursor must be removed when graphics near or under the cursor are drawn. Then the cursor is redrawn. This makes for a very jerky cursor with a lot of flicker. `Xsgi` avoids this problem.

When the shmiq driver receives a new cursor location from the mouse (or other pointer device), it calls into the graphics driver to position the cursor. This tracking happens without any intervention from `Xsgi`.

Kernel cursor tracking has less overhead and also has an important psychological effect. When the server gets busy and the cursor has jerky movement, the user perceives a less interactive workstation. Kernel cursor tracking allows the cursor to move smoothly independent of how busy the server is, so the user perceives a more interactive system overall.

3.4 The X Input Extension

The core X protocol only specifies a mouse and keyboard device. Additional devices can be supported through the X Consortium's standard X Input extension specified in Patrick [8]. While a sample implementation of the X Input extension is supplied in MIT's X11R5 distribution, no versions of the MIT Sample Server use the code.

The X Input extension is supported by `Xsgi`. SGI's implementation understands IDEV events read from the shmiq. In this way, the X Input extension support avoids having to understand specific input devices. All the input devices mentioned earlier are accessible through the X Input extension.

4 The Non-Frame Buffer Layer

In order to support the large, growing number of graphics subsystems that SGI develops, SGI X server engineers developed a porting layer well-suited for porting X to advanced hardware with acceleration features and without exposed frame buffers. The Non-Frame Buffer (NFB) layer resulted. The goals of NFB are the following:

- Allow quick server bring up for new hardware with reasonable initial performance.
- Allow incremental software tuning to improve performance.
- Provide machine independent support for advanced hardware features such multiple hardware colormaps and overlays.
- Provide machine independent rendering optimizations.

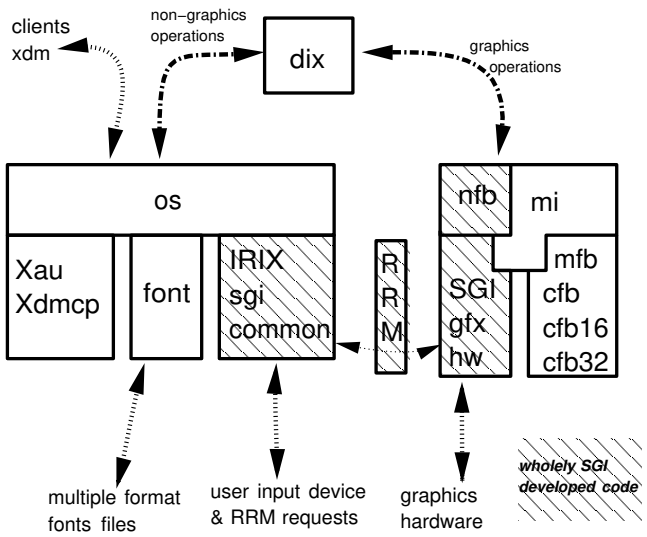


Figure 4: `Xsgi` internal organization.

- Ability to easily track MIT changes.

Experience has shown these goals are well met by the NFB. Currently, SGI supports eleven distinct pieces of hardware using NFB. With respect to tracking MIT changes, only very minor changes are needed to put the NFB in an X11R5-based server.

It is important to note that the NFB layer does not obsolete MIT supplied code such as the color frame buffer (CFB), monochrome frame buffer (MFB), and machine independent (MI) pieces of code. All these chunks of software are still in `Xsgi`. The MFB and CFB code is used for managing pixmaps. Most routines in the MI code are used, particularly the complex rendering algorithms (wide lines, dashed arcs, etc.) that could not easily be better supported by NFB. But some MI code like the software cursor routines are not needed by `Xsgi`.

To see how all the various layers and pieces of X server internal code fit together, see Figure 4. A more technical discussion of the details of the NFB can be found in Weinstein [10].

5 Graphics Hardware Support

The NFB is the framework in which all of SGI's graphics hardware is supported for X. But each piece of hardware still requires its own device dependent bottom layer to talk directly to the hardware.

As mentioned earlier, no SGI graphics hardware has an exposed frame buffer. Instead of manipulating a frame buffer, graphics commands are sent down a graphics pipeline. The sophistication of the pipeline varies. On low end machine such as the Indigo with Starter graphics, the pipeline is rather short and the commands are rather primitive (most complex GL operations are implemented in software). On high end machines, very complex, highly pipelined hardware is available for the highest performance.

5.1 Performance

SGI hardware and system software has always been tuned for the highest possible GL performance. As mentioned earlier, NeWS was SGI's previous window system. For this reason, some early SGI graphics hardware is not well suited for X.

An example of this is the lack of support of hardware support for logical pixel operations make some early systems slow at non-source logical pixel operations. (NeWS is based on PostScript which does not support logical operations on pixels). Another example is the lack of two color hardware cursors on old hardware (some early Personal Iris models).

Since X has become SGI's standard windowing system in IRIX 4.0, hardware is now designed to suit X's capabilities as well as GL's needs. For example, the Indigo Starter graphics has special commands tuned to X rendering.

End users of X servers may be surprised by the amount of tuning that goes into a production X server (not just by SGI but nearly all vendors of production X servers). MIT can supply fast machine independent algorithms and good frame buffer manipulation layers (CFB and MFB) but only vendors are in a position to best utilize the capabilities of their proprietary graphics hardware features. Tools such as `x11perf` and profilers are used to locate performance problems and squeeze out all the performance available.

Using NFB lets SGI split optimizations into two categories: those generally available to non-frame buffer hardware and those optimizations for a particular graphics board.

An example of a general non-frame buffer optimization is the use of screen-to-screen copies (via bitblit hardware) to quickly replicate a pattern (such as the root window stipple) across the screen. The pattern can be drawn once, then successive fast screen-to-screen copies are used to draw larger and larger chunks of the pattern in avalanche fashion. This optimization rightly belongs in NFB since the algorithm is generally useful to lots of non-frame buffer hardware. An example of an optimization suited for a particular piece of hardware is the use of Starter Indigo's Direct Memory Access (DMA) to drive a memory-to-screen copy at the *maximum possible* system throughput.

5.2 A Cross Hair Cursor

SGI also tries to support specific hardware features beyond what MIT's Sample Server generally allows. A good example of a hardware feature supported by `Xsgi` outside the normal capabilities of X is the ability to display a hard-

ware cross hair cursor. A cross hair cursor consists of a horizontal and vertical line which intersect at the cursor screen position. All SGI hardware supports such a cursor and the GL provides software support. When the GL was ported to X, support for a cross hair cursor needed to be available via the X server.

Cross hair cursor support could have been provided via a proprietary X extension. But an extension is a heavyweight solution and would be overkill. In the case of a cross hair cursor, it can be supported by allocating an XID for the cross hair cursor and placing a property on the root window of each screen named `_SGI_CROSSHAIR_CURSOR` which contains the XID of the cross hair cursor for that screen. Then clients use the cross hair cursor by querying the property and associating the cross hair cursor XID with the window desiring a cross hair cursor. `Xsgi` treats the cross hair cursor XID specially and requests the hardware to generate a cross hair cursor when installed for a window.

By not requiring an extension, cross hair cursor support requires a minimum of code support and clients do not need a special library to use the feature. An example of a program which makes the root cursor a cross hair cursor is presented in Appendix A.

5.3 Hardware Colormaps

Another very important hardware feature SGI supports in `Xsgi` where it is available is the use of multiple hardware colormaps. All recent SGI hardware has this feature. This can eliminate much of the ugly and distracting effects of *colormap flashing* which occurs when clients try to allocate their own colormaps. The vast majority of graphics hardware supports only a single colormap. This makes colormap flashing inevitable when X clients allocate multiple colormaps on such servers.

X11 has always supported multiple hardware colormaps. Part of the information passed to a client when the server starts is the minimum and maximum number of installed colormaps that can be supported per screen. For screens without multiple hardware colormaps, both values are one.

The minimum number of colormaps is the minimum number guaranteed to be installed simultaneously. The maximum number of colormaps is the maximum number of colormaps that "might possibly be installed simultaneously." On SGI equipment which supports multiple hardware colormaps, the maximum number will be greater than one. Generally, the minimum number will still be one for `Xsgi`. This is because a simultaneous installation is difficult to *guarantee* on a server like `Xsgi` where visuals such as those for overlay planes may not be capable of supporting multiple hardware colormaps.

But for common uses of common visuals, the multiple hardware colormaps are extremely useful. Instead of sharing the default colormap, programmers with applications which demand a large number of colors find the being able to allocate their own entire colormap is very useful.

As with any hardware resource, you can eventually run out of hardware colormaps and colormap flashing will still result on SGI machines in this case, but it is a much less likely occurrence. Users who deal with applications that generously use color resources will definitely appreciate the multiple hardware colormaps of `Xsgi`.

Because many pieces of hardware support hardware colormaps, the code for managing multiple colormaps is in the NFB layer so only the hardware dependent code for updating the colormap resources needs to be in the hardware dependent section.

5.4 Overlay Plane Support

Overlay planes are another set of graphics planes which *overlap* the normal set of graphics planes. It is much like having a front and back frame buffer. A *transparent* value for a pixel in the front frame buffer allows the corresponding pixel in the back frame buffer to show through. Overlay planes are useful when temporary graphics (text annotations for example) are placed atop complex images. The text can be removed or redrawn without disturbing the underlying image. Pop up menus are another use for overlay planes.

Overlays planes are in some sense a performance hack. The same visual effect could be achieved without overlays but with *much more* work. They attempt to provide better performance by minimizing the amount of redrawing necessary for programs that use them.

All SGI graphics subsystems support overlay planes. While traditional X does not support overlay planes, the GL does. MIT's Sample Server was not designed with overlays in mind and so X support for overlay planes needed to be added by SGI.

The current implementation of `Xsgi` supports X clients and GL clients using overlay planes. By default the SGI window manager `4Dwm` places its menus in the overlay planes. The basic idea is to treat overlay planes as another type of X visual. Overlay planes do greatly complicate many areas of the server. When overlay planes are in use, the region of a window to be rendered and the region of a window that is visible are not always identical. Colormaps, event distribution, exposure generation, and backing store are all complicated by introducing overlay planes. A technical survey of all these concerns and SGI's solution can be found in Newman [7].

Despite these hurdles, overlay planes can be used effectively to achieve graphics effects very difficult to achieve without the overlays. For this reason, vendors will continue to supply and users will continue to use overlay planes in applications. As sophisticated graphics hardware becomes more common, more X users and vendors will be dealing with the capabilities of overlay planes.

6 Dynamic Linking of DDXs

Supporting a large number of graphics devices has repercussions beyond simply writing the necessary graphics code to make X work. It has a tendency to create very large server executables necessary to support all available hardware. Not only does the server become bulky but new versions of the server must be released to support each new piece of hardware.

To remedy this situation, `Xsgi` supports dynamically linkable DDXs which encapsulate the device dependent portion of the server. A DDX is X server jargon for Device Dependent X. When the server starts it looks in the `/usr/lib/X11/dyDDX` directory to find the installed DDXs which are available. Each DDX in the directory is dynamically linked into the server while the server is running. A probe routine is run. If the probe routine detects the hardware the DDX is intended for, the DDX will install itself into the server. If the hardware is not found, the server will unload the DDX and continue looking for a DDX that matches the available hardware.

It is possible if multiple pieces of different graphics hardware is found, multiple DDXs will be linked and installed. Normally, only a single DDX for a machine's available graphics hardware needs to be installed. This conserves disk space.

In the case of the MIT Sample Server, the server is entirely linked as one monolithic server which only supports the graphics hardware it was compiled to support.

Currently Silicon Graphics supports eleven different DDXs for each of its available pieces of graphics hardware.

Other advantages exist to this approach. The hardware independent portion of the X server can be upgraded without changing any of the DDXs. And a given DDX can be upgraded to fix a bug or use a better tuned version of the DDX. It is possible a major change in `Xsgi` will cause the interface between the core server and the DDXs to change (as was the case in moving from R4 to R5). To handle this eventuality, version numbers are encoded in each server and DDX to avoid mismatches.

Silicon Graphics licensed this same dynamic DDX linking technology (along with the NFB) to Santa Cruz Operation for use in their Xsight X server because SCO had to support the tremendous number of advanced graphics adapters available to PCs. The technology is a proven, flexible way to support multiple pieces of graphics hardware without compromising server size or upgradability.

7 Display PostScript

Adobe's Display PostScript is a valuable X extension which adds PostScript's powerful rendering model to `Xsgi`. It is the enabling software that allows programs such as Frame Technology's FrameMaker to include encapsulated PostScript in documents and the Silicon Graphics bundled program `xpsview` to preview PostScript documents. More on Display PostScript can be found in Holzgang [5].

Because Adobe's Display PostScript extension is proprietary, it is not available in the MIT Sample Server. Many vendors have licensed Adobe's Display PostScript extension and make it available in their production server.

`Xsgi` allows the Display PostScript to be dynamically linked in just like dynamically linked DDXs. `Xsgi` operates in such a way that the extension is only loaded if Display PostScript extension requests are actually made. This reduces the size of the server when not using Display PostScript in your session.

Additionally, just like graphics hardware dynamic DDXs, the Display PostScript DDX allows `Xsgi` executable and the Display PostScript subsystem to be upgraded independently.

8 Conclusions

Overall, `Xsgi` makes significant improvements in the state of X server technology. While many features and concerns `Xsgi` addresses are not common to today's X servers, almost certainly X graphics hardware and X servers will increase in sophistication and production servers with `Xsgi`'s features will be commonplace. Many vendors today are supplying similar features and many more will follow.

It is hoped users will understand and appreciate the type of features `Xsgi` showcases and demand servers which not only implement the X protocol but do so to the best that technology will allow.

9 Acknowledgments

The SGI X server is the result of many engineers most notably Peter Daifuku, Erik Fortune, Robert Keller, Spencer Murray, Todd Newman, Tom Paquin, Paul Shupak, Dave Spalding, and Jeff Weinstein.

The words IRIS and IRIX are trademarks of Silicon Graphics, Inc. RISC System/6000 is a trademark of IBM. Spaceball is a trademark of Spatial Systems Incorporated. X Window System is a trademark of the Massachusetts Institute of Technology.

A Cross Hair Cursor Example

```
 1  /* compile: cc -o crosshair crosshair.c -lX11_s -lsun */

 2  #include <X11/Xlib.h>
 3  #include <X11/Xatom.h>
 4  #include <stdio.h>

 5  main(argc, argv)
 6  int argc;
 7  char *argv[];
 8  {
 9     Display *display;
10     Window root;
11     Atom crosshair_prop, atom_type;
12     int rc, format;
13     unsigned long nitems, remaining;
14     XID *value;

15     display = XOpenDisplay(NULL);
16     if(display == NULL) {
17        fprintf(stderr, "crosshair: cannot open display %s\n",
18           XDisplayName(NULL));
19        exit(1);
20     }
21     crosshair_prop = XInternAtom(display, "_SGI_CROSSHAIR_CURSOR", True);
22     if(crosshair_prop == None) {
23        fprintf(stderr, "crosshair: could not intern _SGI_CROSSHAIR_CURSOR\n");
24        exit(1);
25     }
26     root = DefaultRootWindow(display);
27     rc = XGetWindowProperty(display, root, crosshair_prop, 0L, 1L, False,
28        XA_CURSOR, &atom_type, &format, &nitems, &remaining,
29        (unsigned char *) &value);
30     if(rc != Success) {
31        fprintf(stderr, "crosshair: XGetWindowProperty failed\n");
32        exit(1);
33     }
34     XDefineCursor(display, root, value[0]);
35     XCloseDisplay(display);
36  }
```

References

[1] AT&T, *UNIX System V Release 4, Programmer's Guide: STREAMS*, Prentice Hall, 1990.

[2] Susan Angebranndt, Raymond Drewry, Phil Karlton, and Todd Newman, Bob Scheifler, Keith Packard, "Definition of the Porting Layer for the X v11 Sample Server," *X11 Release 5 documentation*, April 22, 1991.

[3] James Foley, Andries van Dam, Steven Feiner, and John Hughes, *Computer Graphics: Principles and Practice*, 2nd edition, Addison-Wesley Publishing, 1990.

[4] Edward Haletky and Linas Vepstas, "Integration of GL with the X Window System," *Xhibition 91 Proceedings*, 1991.

[5] David A. Holzgang, *Display PostScript Programming*, Addison-Wesley, 1990.

[6] Phil Karlton, "Integrating the GL into the X Environment: A High Performance Rendering Extension Working with and Not Against X," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Issue 1, Winter 1992.

[7] Todd Newman, "How Not to Implement Overlays in X," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Issue 1, Winter 1992.

[8] Mark Patrick and George Sachs, "X11 Input Extension Library Specification" and "X11 Input Extension Protocol Specification," *X11 Release 5 documentation*, 1991.

[9] C. H. Tucker and C. J. Nelson, "Extending X for High Performance 3D Graphics," *Xhibition 91 Proceedings*, 1991.

[10] Jeff Weinstein, "NFB, an X Server Porting Layer," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Issue 1, Winter 1992.