# System Support for OpenGL Direct Rendering

Mark J. Kilgard David Blythe Deanna Hohn
Silicon Graphics, Inc.
2011 N. Shoreline
Mountain View, CA 94043-1389
email: mjk@sgi.com blythe@sgi.com hohn@sgi.com

**Abstract**

OpenGL's window system support for the X Window System explicitly allows implementations to support *direct rendering* of OpenGL commands to the graphics hardware. Rendering directly to the hardware avoids the overhead of packing and relaying protocol requests to the X server inherent in indirect rendering.

The OpenGL implementation available for Silicon Graphics workstations supports direct rendering using virtualizable graphics hardware in conjunction with the kernel and the X server. The techniques described provide "maximum performance" rendering for OpenGL. Some of the issues are specific to OpenGL, but most of the techniques described are appropriate for the implementation of any high-performance direct rendering graphics interface.

Keywords: OpenGL, Virtual Graphics, Direct Rendering.

## 1 Introduction

The OpenGL graphics system [14, 11] is a window system independent software interface to graphics hardware for 3D rendering. GLX [8] is the OpenGL extension to the X Window System that specifies how OpenGL integrates with X. The GLX specification explicitly allows (but does not require) implementations to support *direct rendering* of OpenGL commands to the graphics hardware. Direct rendering allows OpenGL commands to bypass the normal X protocol encoding, transport, and X server dispatch. Through sufficient hardware and system software support, OpenGL rendering can achieve the maximum rendering performance from the hardware.

Direct rendering naturally implies that the direct rendering process is running on the local graphics workstation (as opposed to running over the network). Direct rendering is not available if the OpenGL process is connected to a remote X server.

For interactive 3D applications, the maximum possible rendering performance is critical to the success of the application. When available, direct rendering has a substantial performance advantage over rendering indirectly via the X server, i.e., *indirect rendering*. Instead of using the X server as a proxy for rendering, rendering commands are sent directly to the graphics hardware. Direct rendering can be thought of as a means of "cutting out the middle man." Indirect rendering is still useful (and required by GLX) because it allows the same network extensibility and inter-operability of traditional X clients.

This paper discusses how Silicon Graphics, Inc. (SGI) implements direct rendering in its OpenGL implementation through a combination of hardware features, operating system support, X server support, and X and OpenGL library support. SGI implements the described facilities in IRIX 5.3.[1] The next section discusses the goal of direct rendering, SGI's approach for supporting virtualized direct rendering, and support for direct rendering by OpenGL's predecessor and other direct rendering graphics systems. Section 3 describes OpenGL's implementation model and the requirements implied for implementing direct rendering. Section 4 presents how SGI virtualizes access to the graphics hardware to support direct rendering. Section 5 addresses other issues not strictly related to virtualized direct access rendering but still important for supporting direct rendering.

[1] IRIX is the SGI version of the Unix operating system. Most of the facilities described were originally developed for IRIX 5.2.

## 2 Background

Support for direct rendering was purposefully designed into OpenGL's GLX specification. While use of the GLX extension protocol permits interoperability and network-extensibility of OpenGL rendering, forcing the X server as an intermediary for OpenGL rendering imposes inherent limitations on OpenGL rendering performance. Permitting direct rendering avoids the unacceptable situation where expensive, high-performance graphics hardware subsystems designed to support OpenGL [1, 6] have their graphics performance potential starved by the overhead incurred by indirect rendering.

Using OpenGL display lists (non-editable sequences of OpenGL commands that can be downloaded into the X server and later executed) can ease the burden of indirect rendering since it minimizes the GLX protocol needed for rendering. However, use of display lists is often inappropriate for many applications, particularly applications with very dynamic scenes. Such applications favor using immediate mode rendering that requires much higher bandwidth for the OpenGL command stream. Measurements of the IRIX 5.2 OpenGL implementation show indirect immediate mode rendering has inferior performance to direct rendered immediate mode [10].[2] Even programs heavily reliant on display lists are slower when rendering indirectly.

### 2.1 Direct Rendering Benefits

Table 1 breaks down the overhead of indirect rendering relative to direct rendering for the extreme and common cases. The extreme example demonstrates the high-level steps involved in executing the OpenGL `glReadPixels` command used for reading pixels from a window. The command in question requires data to be returned to the application. In the indirect case, this requires a context switch to the X server and back to the OpenGL program. And the pixel data returned is copied three times, as opposed to a single copy in the direct rendering case. While `glReadPixels` requires a round-trip to the X server to return the pixel data, most OpenGL commands return no data.

For most OpenGL commands, the context switch overhead can be amortized over multiple GLX requests by streaming protocol requests. The table's `glVertex3f` example (whereby OpenGL sends a 3D vertex to the hardware; a very common OpenGL operation) is handled in this way. Even with streaming, the indirect case incurs the overhead of encoding, transport, and decoding GLX protocol for requests and replies.

The direct case can eliminate the overhead associated with context switching between the OpenGL program and the X server, protocol encoding, transport, and decoding when performing OpenGL rendering. Direct rendering also improves cache and TLB behavior by avoiding frequent context switches and multiple active contexts [3].

The examples in Table 1 assume the "fast path" of SGI's OpenGL implementation is taken. Being on the "fast path" means the system resources for direct rendering (discussed in detail later) are already made available. Unless resources are in contention or resources are being used for the first time, the "fast path" is the norm.

### 2.2 "Maximum Performance" 3D Rendering

SGI's OpenGL implementation seeks to achieve "maximum performance" OpenGL rendering. For our purposes, "maximum performance" means that when there is no contention for rendering resources, and once utilized resources are made available, graphics rendering performance is limited only by the system's raw graphics performance and the graphics software efficiency. The *maximum* performance potential of the workstation should be achievable.

In practice, this means no locks need to be acquired and released when rendering. Even so, multiple OpenGL programs should be able to run concurrently. But the overhead from concurrent use of graphics should *only* be introduced when multiple processes are concurrently using the graphics hardware; performance of the single renderer case should not be compromised. Graphics programs should not be burdened with overhead from window clipping; in particular, multi-pass rendering for correct clipping should not be necessary. And the possibility of asynchronous window management operations such as changing window clipping or changing the window origin should not add *any* overhead to the normal case when clips and origins are not changing.

### 2.3 Virtual Graphics

SGI workstations implement *virtual graphics* [17] to achieve the goal of "maximum performance" rendering. Virtual graphics means that every graphics process has the illusion of exclusive access to the graphics rendering engine. Many systems allow direct access to the graphics hardware. Virtual graphics not only allows direct

[2] The presented results showed immediate mode OpenGL graphics performance using indirect rendering ranging from 34% to 68% of the direct rendering performance depending on the model of SGI graphics workstation. The faster the graphics hardware, the higher the relative penalty for using indirect rendering versus direct rendering due to the higher relative overhead of indirect rendering.

| | Indirect `glReadPixels` | | Direct `glReadPixels` | | Indirect `glVertex3f` | | Direct `glVertex3f` |
|---|---|---|---|---|---|---|---|
| 1. | `glReadPixels(...)` | 1. | `glReadPixels(...)` | 1. | `glVertex3f(...)` | 1. | `glVertex3f(...)` |
| 2. | pack GLX protocol request | | | 2. | pack GLX protocol request | 2. | send vertex to HW |
| 3. | write request to kernel | | | 3. | return | 3. | return |
| 4. | block client waiting | | | | | | |
| | to read reply | | | | *additional non-reply* | | |
| 5. | deschedule OpenGL client | | | | *OpenGL requests can be* | | |
| 6. | schedule X server | | | | *batched in protocol buffer* | | |
| 7. | read request from kernel | | | | | | |
| 8. | request decoded and | | | 4. | eventually, flush | | |
| | dispatched to GLX handler | | | | protocol buffer | | |
| 9. | read pixels from | 2. | read pixels from | 5. | deschedule OpenGL client | | |
| | screen to a buffer | | | 6. | schedule X server | | |
| 10. | write reply to kernel | | | 7. | read request from kernel | | |
| 11. | deschedule X server | | | 8. | request decoded and | | |
| 12. | schedule OpenGL client | | | | dispatch to GLX handler | | |
| 13. | decode reply header | | | 9. | send vertex to HW | | |
| 14. | copy reply data from kernel | | | | | | |
| | to final buffer | | | | | | |
| 15. | return | 3. | return | | | | |

Table 1: A comparison of the "fast path" steps involved implementing `glReadPixels` for the indirect and direct fast path cases. The extra steps involve data copying; protocol packing, unpacking, and dispatching; and context switch overhead.

access, but treats graphics as a virtual system resource. This virtual view of graphics allows the system to contend with simultaneous direct access by multiple graphics processes. Virtual graphics also arbitrates the contention for graphics resources such as screen real estate.

There are three classes of contention that virtualized graphics for a window system must arbitrate:

**Concurrent access contention.** When different graphics processes are using a single graphics engine, this hardware state must be context switched. With virtual graphics, this context switching is transparent to the graphics process, much the same way processor context switches are transparent to Unix processes.

**Screen real estate contention.** Window systems arbitrate how windows are arranged on the screen. Virtualized graphics must ensure that rendering is properly clipped to the drawable region of the rendering window. Clipping should be correct even in the face of asynchronous window management operation by the X server. Because window systems like X allow arbitrary overlapping of windows, clipping to arbitrary regions must be possible.

**Non-visible resource contention.** Modern graphics hardware supports features like double buffering, in which a front buffer is displayed while the next animation frame is generated in a non-visible back buffer. When the frame is complete, a buffer swap effectively copies the back buffer contents into the front visible buffer. Graphics hardware can efficiently perform buffer swaps by tagging all the pixels belonging to a swapping window with a single display mode. The displayed buffer for the window's display mode can be instantaneously changed. But the number of display modes is a limited hardware resource so a virtual graphics system must be ready to virtualize display modes.

Each of these classes of contention are dealt with by SGI's virtualized, direct access rendering for OpenGL.

## 2.4 SGI-style Graphics Hardware

Unlike most low-end workstation and PC graphics hardware, SGI graphics hardware does not expose a memory mapped frame buffer. Instead, graphics commands are issued to a graphics engine through the manipulation of memory mapped device registers. This interface to the graphics hardware is often called the *graphics pipe* or simply the *pipe*. Because all rendering operations are done through the graphics pipe, the pipe's virtual memory mapping can be used to control access to graphics rendering, i.e., virtualize graphics.

There is substantial variation in the extent of geometry and rasterization processing implemented within various SGI graphics hardware configurations. The high-end SGI graphics hardware [1] implements almost the entire OpenGL state machine within the graphics hardware. Most pipe commands (called *tokens*) sent to the graphics hardware have a fairly direct mapping to the OpenGL API. The high-end hardware is micro-coded and makes heavy use of pipelining and parallelism in its various stages.

The low-end SGI graphics hardware [15] implements only the back-end of the OpenGL state machine. Most high-level operations such as vertex transformation, polygon rasterization, texturing, and lighting are performed on the host processor inside the OpenGL library. But low-level operations such as line drawing, shading, span rendering, dithering, etc. are implemented in the hardware rendering engine.

Despite the variations in hardware across SGI's product line, a direct rendering OpenGL program will work (though perhaps not find the same frame buffer capabilities or achieve the same performance) across the entire range of SGI OpenGL-capable graphics hardware. All the rendering code for OpenGL that directly accesses the hardware is isolated in the OpenGL library which is implemented as a shared library. The shared library on the system depends on the graphics hardware installed on the workstation, hiding all device dependencies when direct rendering.

### 2.4.1 Virtualized, Direct Rendering Needs

While the degree to which the OpenGL state machine is supported in hardware varies across the product line, virtualized direct access rendering requires functionality across all SGI graphics hardware configurations. The requirements are:

**A context-switchable graphics engine.** The state of the graphics hardware must be context-switchable and the context switch must be able to be performed preemptively, including in the middle of a command.

**Window relative rendering.** A process using virtual graphics renders using window relative coordinates, so that the window location need not be tracked if the window is moved.

**Arbitrary window clipping.** As mentioned earlier, a window system supporting arbitrarily overlapping windows, can result in arbitrary window clipping regions, so the hardware must support arbitrary clipping. And the window's clip and location can change asynchronously to the direct rendering process so the window clip must be able to change without the renderer's knowledge.

**Double buffering.** Support must exist for per-window double buffering. Also, OpenGL's front/back relative naming of buffers must be supported. A direct renderer should be unaware of the absolute hardware buffer it is rendering to.

### 2.4.2 Window Clipping Hardware

Most windows have rather simple clip regions, consisting of a small number of rectangles. For this common case, the hardware can support a set of *clip rectangles* to quickly clip rendered pixels not belonging to the window [12]. The number of clip rectangles varies with graphics hardware but is typically less than eight. These clip rectangles are generally part of the graphics hardware context. The context for each direct renderer can use the same rectangular clipping hardware.

Arbitrary window clips will require far more clip rectangles than is reasonable to support in hardware. A second method of clipping is used for such windows. SGI hardware also supports *clipping planes*. Clipping planes are non-visible frame buffer planes used to encode per-pixel clipping IDs (often called CIDs). If the pixels of a window all have the same CID (and only pixels belonging to the window have that CID), the graphics hardware can clip to an arbitrary window by enabling CID testing. A pixel rendered into the window is drawn only if the CID of the pixel being modified matches the CID that is set in the graphic hardware context.

The number of planes set aside for maintaining CIDs can be rather small. On low-end SGI hardware, the clipping planes are only 2 bits deep. This means three CIDs are possible (2 bit planes provide 4 CID values but one CID value must be used for screen real estate *not* belonging to the CID assigned windows). As will be explained later, CIDs may need to be virtualized because they are a limited resource.

### 2.4.3 Display Mode Support

Display mode IDs (often called DIDs) are like CIDs, but instead of providing clipping information, DIDs determine on a per-pixel basis how each pixel value on the screen should be displayed. The DID for a pixel is looked up in the hardware *display mode table* to determine how the pixel should be displayed. The DID mode indicates whether the pixel is RGB or color index (i.e., the color is determined by a colormap); if color index, what hardware colormap to use if there is more than one; the depth of the pixel (how many bits of the pixel value are significant); if the pixel is double buffered, and if so, what buffer (A or B) should be displayed. Figure 1 gives an example of how a DID determines how a pixel should be displayed.

Logically, DIDs are stored in a set of non-visible frame buffer planes. Usually 16 or 32 DIDs are available. In low-end hardware, devoting 4 to 5 bitplanes per pixel to store the DID is too expensive. In this case, the DID values are run-length encoded. This encoding is convenient becauses the frame buffer is scanned out in horizontal lines. But logically, there is still a DID per pixel. In principle, a complex arrangement of display modes on the screen might be too complex to represent with a run-

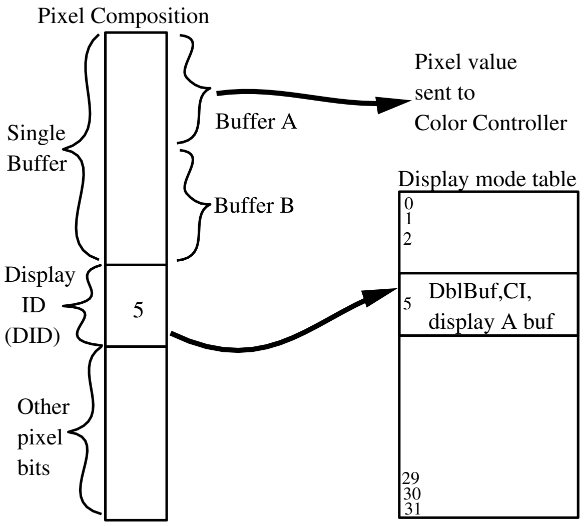



Figure 1: Example of display ID hardware for supporting double buffering. The pixel shown is assigned display ID 5 meaning the pixel should be treated as a double buffered, color index pixel with buffer A being the displayed buffer.

length encoded table. In practice, run-length encoded DID tables work extremely well.

Note that many windows can all *share* the same DID if their pixels all use the same display mode. For example, all non-double buffered 8-bit color index windows using the same colormap can share the same DID.

When a process requests a buffer swap for a window (in OpenGL, `glXSwapBuffers` would be called), a double buffer window must have an *unshared* or *swappable* DID. This is because the buffer swap is accomplished by toggling the displayed buffer for the window. If the pixels for the window (and *only* the pixels for the window) are all in the same DID, the buffer swap happens cleanly. What this means is a double buffered window that needs to swap must be placed on an unshared DID before the swap can happen. Like CIDs, the number of DIDs is limited by hardware so DIDs may also need to be virtualized.

While DIDs and CIDs are discussed here as distinct entities, it is possible to combine display mode and clipping information into a single set of bitplanes. This may be useful because often the clipping ID planes and display mode ID planes both have the same regions each assigned a CID and DID. Combining CIDs and DIDs can make better use of frame buffer memory. If 32 DIDs were supported and 4 CIDs were supported, a combined scheme would support 128 combined DID/CID values! But combining DIDs and CIDs means DID information cannot be run-length encoded. This is a graphics hardware design trade off.

## 2.5 Previous IRIS GL Support

The predecessor to OpenGL is SGI's proprietary IRIS GL. While OpenGL leaves window system operations to the native window system (for example, the X Window System or Windows NT), IRIS GL provides its own window management routines. OpenGL has a similar 3D rendering philosophy to IRIS GL, but OpenGL is a distinctly new interface. The OpenGL state machine is well defined and the OpenGL API has a cleaner design and regular name space. One of the most important changes in OpenGL from IRIS GL is the clean separation of renderer state from window state. In IRIS GL, the renderer and window state were coupled.

Locally running IRIS GL programs use virtualized, direct access rendering. In fact, most of the experience in supporting virtualized, direct access rendering for OpenGL was a result of experience with IRIS GL. The current IRIS GL direct rendering support actually uses the same support OpenGL uses.

## 2.6 Other Approaches

Direct rendering in the manner SGI describes in this paper is not the only option for implementing direct rendering. And direct rendering is not a necessity for production OpenGL implementations. The IBM OpenGL implementation described in [7] does not utilize direct rendering. Most currently available OpenGL implementations do not support direct rendering.

Previous to IBM's support for the OpenGL standard, IBM licensed IRIS GL from SGI for 3D hardware in the original RS/6000. Their IRIS GL implementation uses virtualized, direct rendering much like SGI's IRIS GL implementation [16, 5].

Hewlett-Packard provides direct rendering support for their Starbase Graphics Library [2] with an approach that is different from SGI's direct rendering mechanism. Hewlett-Packard's approach acquires a fast lock to be held during rendering to the graphics engine. This locking allows clipping to be coordinated in software via shared memory window clip serial numbers and proprietary X requests to query the current clip of a window. While such a system avoids the complex hardware and operating system support involved in SGI's virtualized, direct access rendering mechanism, it forces explicit, fine-grain locking to arbitrate access to the graphics hardware.

## 3 OpenGL Requirements

The OpenGL GLX specification provides the model used to integrate OpenGL with the X Window System. Understanding the GLX model motivates how SGI implements virtualized, direct access rendering specifically for OpenGL and X.

### 3.1 Context Model

An OpenGL rendering context (or `GLXContext`) is logically an instance of an OpenGL state machine. When a context is created using `glXCreateContext`, the creator has the option of requesting a direct rendering context. If the program is running locally and the OpenGL implementation supports direct rendering, a direct rendering context will be created. Everything that can be done with a direct context can be done with an indirect context (the reverse is not true) so requesting a direct context but being returned an indirect context is acceptable.

Once a context is created, that context can be bound or "made current" to a drawable (either a window or pixmap) supporting OpenGL rendering by calling `glXMakeCurrent`. Not only is the context bound to the drawable, but also to the thread calling `glXMakeCurrent`. Once bound, any OpenGL calls issued by the thread are issued using the current context and affect the current drawable. Only one thread can be bound to a given context at a time; but multiple contexts (bound to different threads) can be bound to a single drawable. Subsequent calls to `glXMakeCurrent` rebind the thread to the newly specified drawable and context.

### 3.2 Sharing of Window State

GLX explicitly allows the sharing of window state. For example, all OpenGL renderers bound to a double buffered window share the same notion of front and back buffer state. This means if one client calls `glXSwapBuffers` on a window bound to by other OpenGL renderers, the other renderers maintain the same view of which buffer is front and which is back.

One requirement of GLX that proves difficult to meet is the sharing and management of ancillary buffer contents for multiple renderers bound to the same window. Ancillary buffers are non-visible buffers used by rendering operations. Examples are stencil, depth, and accumulation buffers. Sharing ancillary buffers is straightforward if they are supported in hardware, but sharing of buffers implemented via software is more difficult to correctly support.

### 3.3 Per Window Double Buffering

OpenGL supports per-window double buffering using the `glXSwapBuffers` call. A side effect of calling `glXSwapBuffers` on a window that the calling thread is currently bound to is that further rendering to the window will not execute until the buffer swap completes. Double buffer hardware usually times the buffer swap to occur during vertical retrace. The `glXSwapBuffers` may return before the buffer swap completes, but the OpenGL implementation is then responsible for delaying any further OpenGL rendering to the window until the buffer swap actually occurs.

## 4 Virtualizing SGI Graphics

When an OpenGL process creates a direct OpenGL rendering context, the process opens the graphics device. The process allocates an IRIX kernel resource known as a *rendering node*. A rendering node is a virtual graphics hardware context and permits the graphics pipe to be mapped into or "attached to" the process's address space so a process can directly access the graphics hardware. Every direct rendering OpenGL context has an associated rendering node. Note that rendering nodes are completely hidden from OpenGL programs. The allocation and use of rendering nodes is purposefully not made available for use by applications. They exist only to support implementing the OpenGL and IRIS GL APIs.

The SGI X server [9] also uses a rendering node to access the graphics pipe. But the X server's rendering node is marked as being the *board manager* rendering node. The board manager rendering node is allowed to call a number of special board manager `ioctl`s used for validating and invalidating resources associated with other rendering nodes. By acting as the board manager, the X server must process messages sent by the kernel indicating the needs of rendering nodes to have their virtual graphics resources validated. The X server receives the messages through a shared memory input queue (or *shmiq*) also used by the kernel to efficiently pass input device events to the X server. The purpose of the kernel messages and how the X server responds to them are discussed shortly. The X server also uses its rendering node for standard X server rendering.

### 4.1 Making Current to a Window

Before a direct rendering process can begin referencing the graphics pipe through the rendering node's pipe memory mapping, the rendering node must be bound to a window. For OpenGL, this happens at `glXMakeCurrent` time via a graphics driver `ioctl`.

The kernel manages a cache of bound and recently bound windows. The cache, known as the *pane cache*, allows OpenGL threads to quickly bind and rebind to windows. If the window being bound to is not found in the pane cache, a currently unbound pane cache entry is selected and reused for the window, and a message is sent to the X server notifying it that a direct renderer is interested in rendering to the specified X window ID. This message allows the X server to initialize data structures for the window to keep track of direct rendering.

A new entry in the pane cache will have two important resources marked invalid. The first resource is the clip resource. When valid, this means the X server has properly informed the kernel of the proper window origin and clip rectangles or CID to be used for clipping rendering to the window. The second resource indicates if the window is assigned a swappable DID.

### 4.2 Context Switching

Access to the graphics pipe is mediated using virtual memory techniques. The graphics pipe is physically mapped to only one process at a time. Other processes using the graphics pipe will have invalid virtual memory mappings for their pages corresponding to the pipe. If a process without valid mappings accesses a pipe address, a page fault is generated. The kernel graphics driver handles this page fault. If the graphics pipe pages can be physically mapped immediately (there are reasons access to the pipe might temporarily be denied that are discussed later), the kernel will save the graphics context of the current process and mark that process's graphics pipe pages invalid. Then the kernel restores the graphics hardware context for the faulting process and validates that process's memory mapping for the pipe.

The context switching sequence is transparent to the processes involved. If a single process is using the pipe, the pipe does not need to be context switched. Graphics pipe context switching only occurs when there is contention for the pipe. Preemptive scheduling by the kernel ensures graphics processes get a fair allocation of time to access the graphics pipe. On multi-processor machines, the scheduler needs special modifications to prevent two simultaneously scheduled graphics processes from continually stealing the graphics pipe from each other.

### 4.3 Clip Validation

As hinted above, the graphics pipe mapping is not always immediately validated when a process faults on a pipe access.

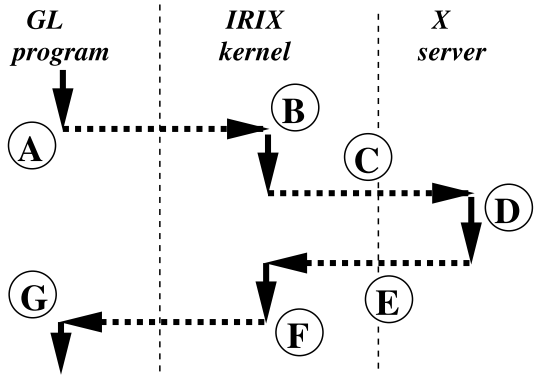
Each rendering node has a clip resource which is either valid or invalid. When valid, the graphics kernel driver has up-to-date window clipping parameters in the pane cache for the window that can be loaded into the rendering node's graphics hardware context (either a set of clip rectangles or a CID value to use). When the clip resource is valid and there is no other reason for the pipe to be invalid, the kernel graphics page fault handler validates the graphics pipe pages after loading the correct clipping information into the graphics hardware context.

If the rendering node's clip resource is invalid, the faulting process is suspended and a clip validate message is sent to the X server. The X server receives the message through the shmiq and issues a board manager reserved "clip validate" `ioctl` to the graphics driver informing the kernel of the correct clipping parameters for the window. When the pane cache entry for the window is updated with the new clip information, then any processes waiting for the clip resource of the window to be validated are awakened. The result is that a graphics process can asynchronously have its clip updated without any knowledge on the part of the process. An example of the clip validation process is described in Figure 2.

Clip resources are possibly invalidated by the X server whenever the window tree is manipulated by X window management operations. The X server tracks which windows are being used for direct rendering, and if the clips of any of these windows change the X server uses a board manager reserved "clip invalidate" `ioctl` to tell the kernel to invalidate the clip resource of any rendering nodes bound to the windows. The invalidation of a clip happens asynchronously to the direct rendering program and can happen at any time the X server needs it to. The next time a direct rendering program attempts to render to the window, the clip validation process ensures a new, correct clip is loaded into the direct renderer's graphics hardware context.

The other reason the clip resource of a window might be invalidated is if too many windows that need CIDs assigned to them are being directly rendered to. Remember there are a limited number of CIDs in the hardware. The X server can virtualize clips by invalidating the clip of one window assigned a CID, reusing the CID by repainting another window with the newly freed CID, and then validating the clip of a window and assigning it the newly reassigned CID. The X server makes no attempt to handle thrashing or starvation due to repeated CID invalidations and validations, but in practice, because most windows use clip rectangles, CID thrashing is not a problem.

Notice that clip validation happens lazily. Not until a direct renderer actually touches the pipe does a clip




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.
G) GL program continues running with no knowledge of the interruption.

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

validation begin. Often when clips on the screen change, they change repeatedly (a window manager using opaque move is a good example of this). So it makes sense to validate clips only on demand.

The pane cache minimizes clip validation costs when an OpenGL process repeatedly binds and rebinds to different windows (an operation expected to be common for OpenGL programs). A rendering node can be bound to a previously bound window, and if the window is still in the pane cache and the clip is still valid, immediately have a valid clip resource.

## 4.4 Fast Buffer Swaps

A continuously animated application swaps buffers frequently enough that the operation should be optimized. As explained earlier, SGI implements a buffer swap by toggling the display mode table entry for the unshared DID assigned to the window during vertical retrace.

If the `glXSwapBuffers` command is called on the currently bound window, the buffer swap is considered to be in the OpenGL command stream. This allows a direct renderer to provide a buffer swap without contacting the X server. The graphics driver provides a "buffer swap" `ioctl` which can be issued by direct renderers. The result is to schedule a buffer swap at the next vertical retrace for the thread's currently bound window. If the `glXSwapBuffers` command is not for the currently bound window, the OpenGL library generates a GLX protocol request to swap the buffers and lets the X server perform the buffer swap.

The `ioctl` returns immediately. This is good because waiting for the vertical retrace could cause a delay equal to the vertical retrace interval (typically at the rate of 60 times per second). But further drawing to the window must be held off until the buffer swap completes. To do this, the graphics driver invalidates the "allow rendering" resource and invalidates the virtual memory mapping to the graphics pipe. Any further access to the pipe by the rendering node will stall the process until the buffer swap completes. When the buffer swap completes, the "allow rendering" resource will be revalidated so rendering can continue.

Because which buffer is front and which is back is window state shared by all OpenGL direct renderers bound to the window, the kernel will also update the absolute sense of what buffer is front and back for any other rendering nodes bound to the window being swapped. The graphics hardware provides a means to switch what buffer is the front and back buffer without the knowledge of the direct renderer.

The advantage of not immediately stalling the process until the buffer swap completes is that most animation applications have a certain amount of computation to do before the next image (typically called a frame) can be rendered. By not immediately stalling the process, this computation can be overlapped with waiting for the buffer to swap.

## 4.5 Window Display Mode Validation

In the discussion of buffer swapping so far, it was assumed that the window to be swapped was indeed on an unshared DID. Since DIDs are a limited hardware resource, this may not necessarily be true. In this case, DIDs must be virtualized.

Similar to a rendering node's clip resource, rendering nodes also have a "swappable window" resource. The resource is valid if the X server has placed the window on an unshared DID. It is invalid if the window's DID is shared by other windows or the X server has revoked the ability to swap (this need is made clear when mixing

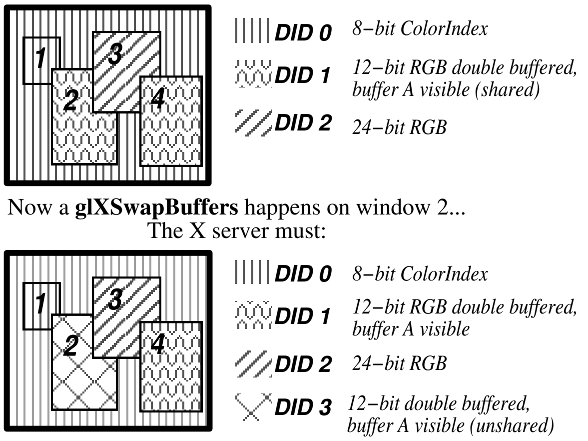



Figure 3: Example of a `glXSwapBuffers` being performed on window 2 which shares the same display ID (DID 1)with window 4. Before the buffer swap can be performed, the X server must rewrite the display IDs such that window 2 is on an unshared display ID (DID 3).

OpenGL and X rendering is discussed).

If a process issues the "swap buffer" `ioctl` and the process's current rendering node does not have its "swappable window" resource valid, a message is sent to the X server requesting the X server place the window on an unshared DID and validate the rendering node's "swappable window" resource. The X server complies with the request and validates the "swappable window" for the window via a board manager reserved `ioctl`. Once validated, the buffer swap can then be scheduled.

As in the virtualization of CIDs, unshared DIDs may be stolen from windows already on an unshared DID to validate the resources of rendering nodes attempting buffer swaps. When a DID is stolen, the window previously on an unshared DID will have to find another window to share a DID with that has the identical display mode (because a window should *never* be assigned a DID with a display mode not matching the correct display mode for the window). Forcing a window to share a DID with another window may force that other window to have its "swap buffer" resource invalidated since it might have previously had an unshared DID allocated to it. Figure 3 is an example of a window needing to be assigned an unshared DID.

### 4.6 Shared Software Buffers

OpenGL's GLX specification requires implementations to support various types of ancillary buffers. When there is no hardware support for these various types of buffers, OpenGL implementations are expected to support these buffers in software by allocating host memory.

GLX requires the contents of ancillary buffers to be shared between renderers binding to a window and the contents of these buffers to be retained even when no renderers are bound to the window. For hardware buffers, these requirements are typically straightforward to meet since the ancillary buffers exist in the hardware frame buffer.

OpenGL indirect rendering could easily allow ancillary buffers to be shared between renderers since all the buffers would exist in the X server's address space and the X server has immediate knowledge of the changing state of windows.

Combining direct rendering with retained, shared software ancillary buffers is difficult to achieve without compromising performance. The SGI direct rendering OpenGL implementation does not currently support the correct sharing of ancillary buffers between renderers in different address spaces. Each OpenGL library instance allocates software ancillary buffers for its own address space. These buffers can be shared between renderers in the same address space. Also, the contents of these buffers are retained only for the lifetime of the address space.

Incorrectly supporting software buffers is strictly speaking a violation of what OpenGL requires. But few programs rely on sharing buffers across address spaces. Sharing software buffers is an area where SGI's OpenGL implementation does not properly isolate window state from rendering state. Further work needs to be done to support ancillary buffer sharing.

One problem that must be solved is the de-allocation of software ancillary buffers when windows are destroyed. The OpenGL library has no obvious way to find out when an X window it is maintaining software ancillary buffers for is destroyed so it can know to deallocate those buffers. Since the buffers tend to be quite large, leaking ancillary buffers is extremely expensive.

SGI solves the problem by adding a private extension to the X server that requests the X server to generate a `SpecialDestroyNotify` when a specified X window is destroyed. The first time an OpenGL rendering context is bound to a window, this request is made for the new window. Hooks in the X extension library (`libXext`) allow `SpecialDestroyNotify` events to trigger a callback into OpenGL to deallocate the associated software buffers. The event is never seen by an X program. Other mechanisms such as using the standard X `DestroyNotify` event proved unreliable since the X client might not be selecting for that event.

### 4.7 Cursors

Logically, a window system cursor "floats" above the windows on the screen. Standard dumb frame buffer graphics hardware for window systems requires special software support for managing the window system cursor.

The standard "software cursor" technique [13] used to manage the window system cursor is to save the pixels under the cursor when the cursor is rendered. When the cursor's image might interfere with rendering or frame buffer read back, the cursor must be undrawn (restoring the saved pixels) and redrawn on completion of the rendering or read back.

The undraw/redraw technique described above is reasonable if the window system server is the only renderer *and* the window system server manages the cursor. But using direct rendering, asynchronous direct renderers need to render into windows containing the cursor but do not have immediate knowledge of the cursor to utilize the undraw/redraw technique. Techniques for integrating software cursors with direct rendering have been implemented [2], but they require a graphics hardware locking strategy that is incompatible with SGI's goal of "maximum performance" rendering.

The alternative to a software cursor is hardware support for a cursor. Normally, this consists of support in the video back end that merges in the cursor image into the video output. Using a hardware cursor eliminates both the rendering overhead and flicker of the software technique. All SGI graphics hardware supports hardware cursors, thereby decoupling direct renderering from window system cursor management.

## 5 Other Issues

There are other issues that do not relate directly to supporting virtualized, direct access rendering, but that still are important to the implementation of SGI's direct rendering support.

### 5.1 Overlays and Underlays

OpenGL supports overlay and underlay planes. Overlays are frame buffer image planes that are displayed preferentially to the normal frame buffer image planes. A special transparent pixel value can be used to "show through" to the contents of the normal planes. Underlay planes are like overlays but are displayed deferentially to the normal planes. Overlays and underlays are useful for text annotation, rubber banding, transient menus, and animation effects. While a simple frame buffer has a single layer, graphics hardware supporting overlays and/or underlays can be thought of as having multiple, stacked frame buffer layers.

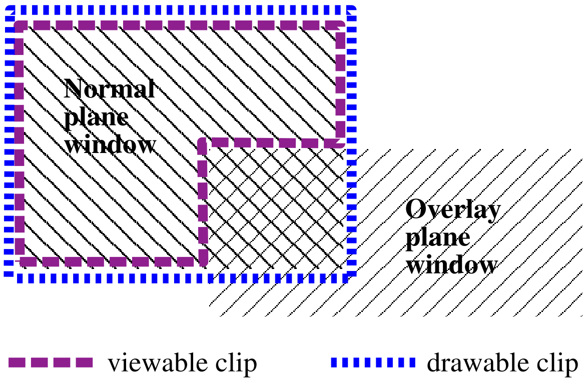



Figure 4: The difference between the drawable clip and visible clip of a window occluded by a window in the overlay planes.

As mentioned previously, OpenGL treats windows in the overlay and underlay planes as first class windows in the X window hierarchy which is the convention for handling overlays in X [4]. IRIS GL had a simpler notion where frame buffer layers all existed in a single window spanning each frame buffer layer. Windows in non-normal layers are just like other windows excepting transparency effects and potentially fewer expose events being generated.

The most important insight into the support for frame buffer layers in a window system is that the drawable clip and the visible region of a window are no longer always identical. Figure 4 demonstrates this point.

SGI found that its older hardware which supported a single layer of integrated CIDs and DIDs made it impossible to support direct OpenGL rendering into the overlay planes. This old hardware is sufficient to implement IRIS GL's simpler model for layered frame buffers, but a single layer of DIDs combined with CIDs makes it impossible to perform CID clipping for overlay plane windows while keeping the display modes correct for the normal plane windows.

The current high-end SGI hardware supports separate DID information per frame layer to solve this problem (to support multiple display modes for the overlays). With separate CIDs and DIDs, the CID planes can generally be used for both overlay and normal planes clipping. Using CIDs for clipping does not change how the display modes are arranged. But using CIDs for both overlay and normal planes clipping could contribute to CID thrashing since the drawable region for an overlay window might overlap the drawable region of a normal plane window. Separate clipping planes for each frame buffer layer could remedy the problem, but CID thrashing due to sharing clipping planes between layers has not proven to be a problem in practice.

### 5.2 Synchronization Issues

GLX treats the OpenGL command stream and the X request stream as two independent sequences of commands. These streams may execute at different rates. GLX supports `glXWaitX` and `glXWaitGL` that allow the X and OpenGL command streams to be explicitly synchronized. `glXWaitGL` prevents subsequent X requests from executing until any outstanding OpenGL commands have completed. `glXWaitX` prevents subsequent OpenGL commands from executing until any outstanding X requests have completed. When using indirect rendering, these calls force their appropriate sequentiality *without* the cost of a round-trip. These calls can be thought of as synchronization tokens actually embedded in the X protocol stream. When direct rendering, a `glXWaitX` does require a `XSync` to ensure an X requests have completed.

### 5.3 Correct Front Buffer Rendering for X

Support for double buffering introduces a new complication to mixing non-OpenGL X rendering with OpenGL rendering for SGI. When X rendering is done to an X window, the render operations should affect the front buffer of the window. So X rendering must be directed into the correct relative buffer, i.e., the front buffer.

As discussed earlier, rendering nodes virtualize the relative access to front and back buffers when using double buffering. But the X server renders all non-OpenGL rendering to all windows through a single rendering node. The X server does not rely on its rendering node for correct window clipping or to determine correctly the front or back buffer. In part this is because the X server does not bind to particular windows the way OpenGL does.

Because the X server has full knowledge of the window tree, it does its clipping in software (and sometimes with hardware assist). The use of virtualized clipping within the X server is difficult for two reasons:

- The X server renders to many more windows than OpenGL programs do. Using virtualized clipping would quickly exhaust the hardware clipping resources.
- Because the same thread in the X server does core rendering as does the validation of virtual clip resources, use of virtualized clipping would deadlock the X server.

Ensuring X rendering always goes into the front buffer cannot be done using the relative access allowed through rendering nodes. Instead, the X server makes sure it renders into the correct absolute buffer (buffer A or B, as opposed to front or back).

But as discussed before, buffer swaps can be scheduled by direct renderers without the attention or knowledge of the X server. If the X server has validated the swappable DID resource of a window, it can no longer ensure rendering goes into the front buffer. The X server could query the hardware to determine state of the buffer. Querying the current front buffer would require a query per-rendering operation. Worse yet, the query's information is only valid for the instant of the query.

If the X server wants to render into the window of a double buffered window that has its swappable DID resource validated, the X server invalidates the unshared DID resource for the window. A graphics driver `ioctl` is used to perform the invalidation. This invalidation accomplishes two things:

- Revokes the permission of direct renderers to swap the buffer. Further attempts to swap the window will be held off and result in a message to the X server to revalidate the unshared DID resource. The swap will be delayed until the X server revalidates the resource.
- And, returns the current state of the buffer.

While this operation is heavy-handed, it lets the X server render correctly into the front buffer. With stable knowledge of which absolute buffer is being displayed, the X server can render correctly.

The overhead of the scheme is minimized because the X server only checks if it needs to revoke a double buffered window's shared DID resource during the validation of a graphics context (GC) and window. Serial numbers determine when the window and a given GC are validated with respect to each other. When the shared DID resource is validated, the serial number of the window is updated with a unique value, forcing any previously valid GCs to become invalid. Any X rendering will then force a GC validation, at which time the shared DID resource can be revoked.

### 5.4 X Server Multi-rendering for Indirect Rendering

Indirect rendering is still required by the GLX specification so even the best direct rendering support still requires that indirect rendering be supported. SGI's OpenGL implementation treats indirect rendering as a special case of direct rendering.

Independently scheduled threads within the X server execute indirectly rendered OpenGL commands. A distinct thread is created for each X connection using OpenGL indirect rendering. This thread within the X server's address space executes the same OpenGL rendering code that direct renderers use and utilizes the same system support for direct rendering. One can think of the OpenGL rendering threads within the X server as proxies that execute OpenGL commands on behalf of an X client using indirect OpenGL rendering. The OpenGL rendering threads within the server only coordinate with the main X server thread to hand off commands to execute and return results. Otherwise, these threads do not manipulate *any* X server data structures. This technique is called *multi-rendering* and is discussed in greater detail in [10].

## Acknowledgments

Much of the understanding for supporting direct rendering of OpenGL came from the implementation of its predecessor, the IRIS GL. The work of Jeff Doughty, John Giannandrea, Luther Kitahata, Paul Mielke, Jeff Weinstein, and others on IRIS GL's resource management laid the basis for the facilities described here. We would like to acknowledge all those at Silicon Graphics who helped ensure OpenGL's successful implementation, notably Kurt Akeley, Peter Daifuku, Simon Hui, Phil Karlton, Mark Stadler, Paula Womack, and David Yu.

## References

[1] Kurt Akeley, "RealityEngine Graphics," *SIGGRAPH 93 Conference Proceedings*, August 1993.

[2] Jeff Boyton, et.al., "Sharing Access to Display Resources in the Starbase/X11 Merge System," *Hewlett-Packard Journal*, December 1989.

[3] Bradley Chen, "Memory Behavior of an X11 Window System," *USENIX Conference Proceedings*, January 1994.

[4] Peter Daifuku, "A Fully Functional Implementation of Layered Windows," *The X Resource: Proceedings of the 7th Annual X Technical Conference*, O'Reilly & Associates, Issue 5, January 1993.

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

[6] Chandlee Harrell, Farhad Fouladi, "Graphics Rendering Architecture for a High Performance Desktop Workstation," *SIGGRAPH 93 Conference Proceedings*, August 1993.

[7] Chandrasekhar Narayanawami, et.al., "Software OpenGL: Architecture and Implementation," *IBM RISC System/6000 Technology: Vol. II*, 1993.

[8] Phil Karlton, *OpenGL Graphics with the X Window System*, Ver. 1.0, Silicon Graphics, April 30, 1993.

[9] Mark J. Kilgard, "Going Beyond the MIT Sample Server: The Silicon Graphics X11 Server," *The X Journal*, SIGS Publications, January 1993.

[10] Mark J. Kilgard, Simon Hui, Allen A. Leinwand, Dave Spalding, "X Server Multi-rendering for OpenGL and PEX," *The X Resource: Proceedings of the 8th X Technical Conference*, O'Reilly & Associates, Issue 9, January 1994.

[11] OpenGL Architecture Review Board, *OpenGL Reference Manual: The official reference document for OpenGL, Release 1*, Addison Wesley, 1992.

[12] Desi Rhoden, Chris Wilcox, "Hardware Acceleration for Window Systems," *Computer Graphics*, Association for Computing Machinery, Vol. 23, Num. 3, July 1989.

[13] David Rosenthal, Adam de Boor, Bob Scheifler, "Godzilla's Guide to Porting the X V11 Sample Server," *X11R5 Documentation*, Massachusetts Institute of Technology, April 12, 1990.

[14] Mark Segal, Kurt Akeley, *The OpenGL$^{TM}$ Graphics System: A Specification*, Ver. 1.0, Silicon Graphics, April 30, 1993.

[15] Silicon Graphics, "Indy Graphics," *Indy$^{TM}$ Technical Report*, Ver. 1.0, 1993.

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

[17] Doug Voorhies, David Kirk, Olin Lathrop, "Virtual Graphics," *Computer Graphics*, Vol. 22, Num. 4, August 1988.