X Server Multi-rendering for OpenGL and PEX

To be presented at the
8th Annual X Technical Conference,
Boston, Mass., January 25, 1994

Mark J. Kilgard
Simon Hui
Allen A. Leinwand
Dave Spalding

Abstract

To support OpenGL™ and PEX rendering within the Silicon Graphics X server without compromising interactivity, we devised and implemented a scheme named *multi-rendering*. Making minimal changes to the X Consortium sample server's overall structure, the scheme allows independent processes within the X server's address space to perform OpenGL rendering asynchronously to the X server's main thread of execution. The IRIX operating system's process share group facility, user-level and pollable semaphores, and support for virtualized direct access rendering are all leveraged to support multi-rendering. The Silicon Graphics implementation of PEX also uses the multi-rendering facility and works by converting rendering requests into OpenGL commands. Multi-rendering is contrasted with other schemes for improving server interactivity. Unlike co-routines, multi-rendering supports multi-processing; unlike multi-threading, multi-rendering requires minimal locking overhead.

Introduction

The current version[1] of the Silicon Graphics, Inc. (SGI) X server [11] supports both standards for 3D graphics for the X Window System: the OpenGL™ graphics system [16, 15, 14] and PEX 5.1 [20]. OpenGL is an Application Programming Interface (API) for 3D interactive graphics developed by SGI and now administered by the OpenGL Architecture Review Board. OpenGL is the successor to the proprietary IRIS GL graphics interface (the GL stands for graphics library). OpenGL supports the X Window System via the GLX exten-

[1]Shipping with IRIX 5.1; IRIX is the Silicon Graphics version of the Unix operating system.

The authors are Members of the Technical Staff at Silicon Graphics, Inc. Mark, Allen, and Dave work in the X Window System group; Simon works in the OpenGL group. Electronic mail can be addressed to mjk@sgi.com, shui@sgi.com, aal@sgi.com, *and* spalding@sgi.com.

sion [10]. PEX is an X protocol extension developed by the X Consortium to support 3D graphics for X. PEX's rendering functionality and style are largely influenced by the PHIGS and PHIGS PLUS 3D graphics standards [3].

Both extensions create interactivity problems for single-threaded X servers because both OpenGL and PEX requests can take substantial amounts of time to complete. Most X servers based on the X Consortium's sample server dispatch one request at a time and execute each request to completion. This works quite well for the core X protocol since most requests are completed in mere fractions of a second. When requests take a few milliseconds or less to execute, the X server can dispatch requests from all active clients fast enough to create an appearance of simultaneous execution and the user perceives the X server as being both interactive and fair to the clients using it.

But OpenGL and PEX can easily generate requests with arbitrarily long execution times. Even common requests can take several seconds to execute. For an X server that dispatches requests serially and each to completion, the result is an extremely non-interactive X server. Users perceive such X servers as sluggish and difficult to interact with.

The problem is not unique to OpenGL and PEX. Any extension with requests that take considerably more time to execute than the typical core X request can cause interactivity problems for an X server that executes one request at a time to completion.

This paper describes a solution to the interactivity problem named *multi-rendering* used to implement OpenGL and PEX in the Silicon Graphics X server. Multi-rendering involves creating separate threads of execution for rendering inside the X server's address space. These separate threads only interact with the X server's main thread during thread initialization and destruction, when locking a shared memory pool, and to communicate requests to execute and request completion status. Otherwise multi-rendering threads never touch X server data structures such as the window tree and never call core X rendering routines. The main X server thread continues to perform all core X request dispatching and rendering. The rendering threads render directly to the hardware making use of virtualized direct access to avoid needing knowledge of window origins or clip regions.

The IRIX operating system's process share group facility, user-level and pollable semaphores, and support for virtualized direct access rendering provide sufficient machinery to implement multi-rendering. The only performance overhead introduced for core X request execution is very inexpensive memory pool locking. OpenGL and PEX requests are executed without compromising the interactivity of the X server.

The next section describes the goals and requirements for our multi-rendering implementation. The third section considers other approaches to solve the interactivity problem. The fourth section describes operating system facilities used to support multi-rendering. The fifth section details how multi-rendering is implemented in the our X server. The sixth section presents a performance analysis of multi-rendering support.

Requirements and Goals

Multi-rendering is the means, not the end. The concept grew from a set of requirements for the current generation Silicon Graphics X server and the operating system and hardware

capabilities at our disposal. The requirements are:

- Support both indirect and direct OpenGL rendering. Indirect rendering is rendering done by the X server on behalf of the client. Direct rendering is done by the client directly manipulating the hardware. Direct rendering is optionally supported by OpenGL implementations.
- Support the X Consortium's PEX 5.1 extension.
- The same OpenGL rendering library used for direct rendering OpenGL programs "outside" the X server should be used inside the X server.
- The solution must not diminish X server interactivity.

Along with these requirements, we needed a mechanism that would let us meet a number of important goals:

- Do not compromise X server reliability.
- Limit changes to the X Consortium's sample server organization to keep manageable the task of integrating future X Consortium releases and bug fixes.
- Do not require changes to extensions not using multi-rendering.
- Support dynamic loading of the OpenGL and PEX extensions to reduce overhead when not using these extensions.
- Minimize synchronization and locking overhead when using multi-rendering.
- Add no measurable overhead when not using multi-rendering.
- Render PEX using OpenGL to require only a single 3D device-dependent component.
- Provide a single mechanism to work across SGI's full line of graphics hardware.
- Minimize use of expensive operating systems resources. In particular, processes and rendering nodes should be allocated judiciously.
- Meet the schedule for release.

The goals reflect the desire to keep the X server maintainable, to minimize the overall engineering effort required, and to maintain high overall performance. We feel the multi-rendering scheme we have implemented meets all of the above stated requirements and goals.

Other sets of requirements, goals, and system capabilities are likely to lead to other approaches. For example, IBM's OpenGL implementation [12] has a similar set of requirements but does not adopt multi-rendering.

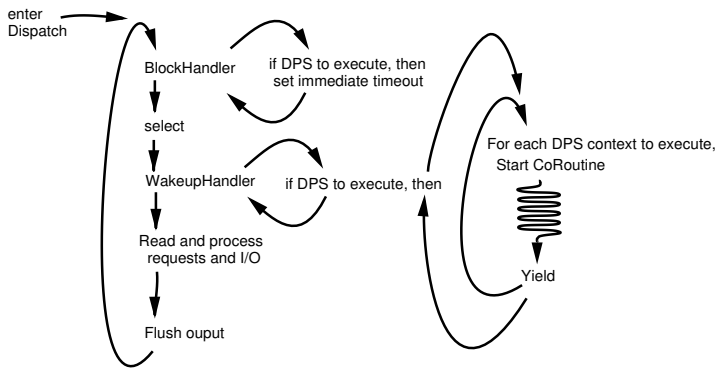

Figure 1: Adobe's Display PostScript extension uses co-routines run from an X server wakeup handler whenever PostScript commands are ready to be executed. Frequent co-routine yielding maintains server interactivity.

Other Approaches

Previous work has been done to solve the interactivity problems that arise when combining lengthy rendering commands with the X Window System. This section surveys three mechanisms. SGI's multi-rendering approach is novel because it supports X extensions that would otherwise cause interactivity problems with minimal X server changes and supports true concurrency with coarse-grain synchronization.

Coroutines in Display PostScript

Adobe's Display PostScript (DPS) extension for X [1] uses a co-routine mechanism for the interpretation of PostScript. A client can create a `DPSContext` and send arbitrary PostScript commands to be interpreted. Unlike core X requests which execute immediately, DPS's `PSGiveInput` request only places PostScript commands in some `DPSContext`'s buffer. Then the X server invokes the PostScript interpreter on the contents of each `DPSContext` with input to process.

Figure 1 shows how DPS's co-routine execution mechanism is integrated into the X server's dispatch loop using the `BlockHandler` and `WakeupHandler` facility. The co-routine mechanism has two advantages: *no* locking is necessary, and operating system process context switching is minimized. The mechanism has the disadvantage that true parallelism is impossible.

The DPS extension is supported by the Silicon Graphics X server using co-routines as described. Multi-rendering and DPS co-exist. Neither mechanism interferes with the other.

Multi-threading the Server

Data General and Omron in association with the X Consortium have designed and implemented a multi-threaded version of the X server called MTX [18]. The project has two major goals:

- Improved interactivity for long duration protocol requests, and
- Better utilization of multi-processor platforms.

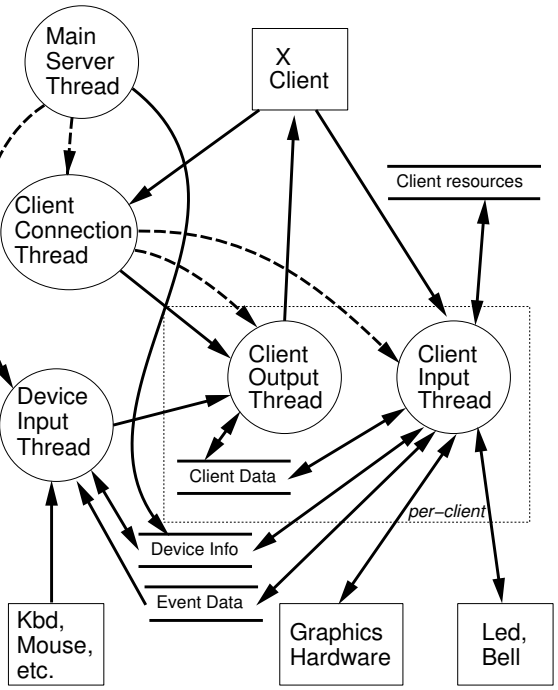

Figure 2: MTX server threads organization. Circles represent threads; squares represent I/O sources; parallel lines represent monitors.

A prototype of MTX will be made available in the X Consortium's X11R6. MTX is an object-based design which assumes inexpensive threads and the use of monitors for synchronization. Figure 2 shows how MTX uses threads.

MTX allows X requests from multiple connections to be dispatched concurrently. On a multi-processor machine, overall throughput could be increased. The server overhead of multiplexing multiple connections is reduced because two threads are dedicated to each client connection; all client I/O does not have to be dispatched by a single thread using `select`.

But MTX does have disadvantages. It uses relatively fine-grain locking (particularly compared to the locking required for multi-rendering) on nearly all X server data structures. While minimizing the costs of locking and synchronization was a primary performance concern, locks add overhead to all operations requiring locking. Compare this with no locks needed for a single-threaded X server.

The most recent publicly available performance characterization of an MTX server indicates 75% of the tested `x11perf` tests were within plus or minus 10% of a comparable X11R5 single-threaded server.[2] It should be noted that `x11perf` involves only a single client so no lock conflicts were likely to be encountered. In real use, lock conflicts would add extra overhead which is hard to characterize based on `x11perf` results. It is claimed that "in general, clients do not seem to generate many resource conflicts," though window managers

[2]This information is based on a performance characterization from a `README.MTX` file available to X Consortium members. The file is time stamped May 3, 1993. Presumably the MTX tests were run on a multi-processor machine.

do tend to create conflicts. A subjective characterization of the performance indicates that using MTX, interactivity is improved compared to a single-threaded server.

MTX does introduce substantial changes to the general organization of the X server. Locking and other concurrency issues diverge the MTX server source code from the X Consortium's current single-threaded sample implementation and complicate X server maintenance. The design specification for MTX [8] is comprehensive and does include guidelines for writing extensions; but code added to the MTX server must be re-entrant, thread-safe, and abide by MTX locking rules. The burden for X server maintenance complexity should be included in the appraisal of the MTX approach.

IRIS GL Direct Rendering

SGI's proprietary IRIS GL, while not an X server extension like the first two examples, does handle the broader problem of maintaining interactivity for simultaneous 3D rendering from multiple programs. IBM also supports a version of IRIS GL [6].

In IRIX 4.0.x as well as IRIX 5.x, IRIS GL programs cooperate with the SGI X server for validation of rendering resources but render directly to the hardware. Operating system support, window system support, and hardware support all combine to provide virtualized direct access rendering. This system support is discussed in the next section.

Our multi-rendering scheme uses much of the same system support originally implemented for IRIS GL, though a good deal of the support needed to be enhanced to support new capabilities allowed by OpenGL such as binding multiple renderers to a single window.

Even though IRIS GL rendering is primarily designed for local use where the graphics program directly accesses the graphics hardware, a network extensible component known as DGL allows IRIS GL programs to run over the network connecting to an SGI workstation supporting IRIS GL. In this case, a program known as `dgld` runs as a proxy for the remote program. The `dgld` is connected to the remote program via a byte-stream network connection. DGL protocol is sent across the connection to the `dgld`. IRIS GL commands which are normally sent directly to the hardware are encoded using the DGL protocol and sent to the `dgld` proxy to be executed. A separate `dgld` runs for every remote IRIS GL program using DGL.

In some respects, this scheme is very similar to multi-rendering except that the proxy executes outside of the X server's address space and a totally separate protocol is used for IRIS GL requests. The DGL proxy scheme has a potential reliability advantage over multi-rendering since bugs or crashes of the `dgld` do not affect the integrity of the X server. A multi-rendering approach requires the rendering processes to maintain higher standards of reliability than a `dgld` proxy requires.

The proxy scheme is inappropriate for OpenGL and PEX because both OpenGL's GLX protocol and the PEX protocol are X extension protocols. Like all X extensions, the GLX and PEX protocols are embedded in the X11 protocol stream. Since the X server must read the OpenGL and PEX requests, it is most efficient for the rendering processes to execute in the same address space as the X server so that requests can be communicated to the rendering threads via shared memory.

System Support for Multi-rendering

SGI's multi-rendering scheme leverages a number of system facilities supported by IRIX and SGI's graphics hardware. Process share groups are the basis for the concurrency needed to support multi-rendering. User-level and pollable semaphores are the basis for the efficient synchronization. And virtualized direct access rendering provides the support for context switching and virtualizing the graphics hardware.

Process Share Groups

Process share groups [4] provide a way for multiple related processes to share an address space. These processes are not very different from normal Unix processes except for the sharing that they support. Unlike light-weight, user-level threads, processes in a share group are scheduled by the kernel. For multi-processor machines, processes in a share group can execute concurrently.

User-level and Pollable Semaphores

User-level semaphores are provided by IRIX and accessible to programmers via the *shared arena* facility [17]. A shared arena allows related or unrelated processes to allocate and share semaphores [5], locks, and memory. SGI's X server multi-rendering facility uses the shared arena's semaphore mechanism. The semaphores are termed *user-level* because only when a process must block on a semaphore does the process need to enter the kernel. In the common case of no contention on the semaphore, the semaphore can be acquired (P) and released (V) with no kernel intervention. This means user-level semaphores are considerably more efficient than a semaphore mechanism that requires a system call per semaphore access.

A variation on user-level semaphores (requiring more kernel support than simple user-level semaphores) is the pollable semaphore. When an acquire operation is attempted by a process and the semaphore is not immediately available, instead of blocking, the process is queued to receive the semaphore and the process continues to execute. A pollable semaphore has an accompanying file descriptor. This file descriptor can be specified in the `select` system call read mask. When the semaphore is acquirable, the next `select` call on the accompanying file descriptor will fall through with the semaphore acquired by the selecting process.

In the case of multi-rendering, a pollable semaphore is used to coordinate request completion with the X server's main thread. The X server can select on the pollable semaphore's accompanying file descriptor just like the file descriptors for client and device input.

The user-level and pollable semaphores are designed to be used with process share groups. In conjunction, true multi-processor concurrency with minimal synchronization overhead can be achieved. The pollable semaphores are particularly well suited for a program like the X server which multiplexes several input sources and sinks via the `select` system call.

Virtualized Direct Access Rendering

SGI designs graphics hardware to permit both direct *and* virtualized rendering. Direct rendering refers to the ability for normal programs to manipulate the graphics hardware directly. Users are hidden from the actual details of the hardware by either the OpenGL or IRIS GL graphics interfaces (implemented as shared libraries). Virtualized rendering refers

to the ability to interleave rendering with other rendering processes and properly constrain rendering to any designated window.

Direct rendering is common on personal computers; on a PC, a program may directly manipulate the frame buffer. Direct rendering allows maximum graphics performance. Virtualized rendering is a must for window systems since rendering must be clipped to the window's drawable region. The X Window System is a good example of virtualized rendering. For common graphics architectures, direct access and virtualized access are traded against each other. Either a single application has direct access to the entire graphics subsystem for its own purposes, or a process, such as the X server, or perhaps the operating system kernel arbitrates access to the graphics resources.

SGI (along with other vendors of sophisticated graphics hardware) [19] provides *both* direct and virtualized rendering using hardware and operating system support. SGI graphics hardware does not expose a frame buffer for direct manipulation. Instead a bank of registers is mapped into the address space of a process wishing to utilize the graphics hardware. Graphics commands are generated by manipulating the banks of graphics registers. In conjunction with operating system support for virtual memory, the graphics hardware interface can be context switched between multiple processes, all apparently using the graphics hardware simultaneously.

Direct access to the graphics hardware is also virtualized. Rendering is window relative; arbitrary window clipping is performed by the hardware. A window's origin and clip can be updated by the window system without the knowledge or consent of a process rendering into the updated window.

The operating system resource which implements a virtual graphics pipeline is known in IRIX as a *rendering node*. Programs desiring to perform virtualized direct access allocate a rendering node and bind it to the window they are interested in rendering to. The rendering node may be rebound to other windows. Allocation and manipulation of rendering nodes is hidden by the graphics library implementation. Rendering nodes are a virtual abstraction of a particular graphics board's graphics context, so a rendering node is allocated for use with a single graphics board.

Like any virtualized resource, there is always the potential for thrashing. In SGI's architecture, there are a limited number of physical hardware contexts (sometimes only one); there are a limited number of clipping planes (necessary for arbitrary clipping); and there are a limited number of unshared display IDs (used for instantaneous buffer swaps when double buffering). The graphics systems are designed to handle average to heavy loads, but the potential for thrashing does exist. Even under extremely heavy loads, the operating system is designed to context switch the available graphics resources transparently.

The SGI X server plays a special role in virtualizing the hardware. The X server and the operating system kernel coordinate through a facility known as the Rendering Resource Manger (RRM). Only the X server has complete knowledge of every window's drawable region, origin, and display mode. Even without multi-rendering for PEX and OpenGL support, IRIS GL support requires that the X server must correctly maintain hardware clips and display IDs. The X server and the operating system kernel coordinate via special

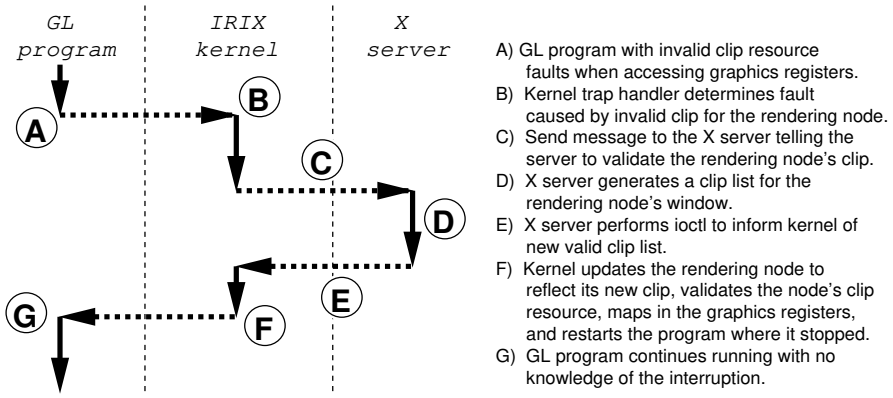

Figure 3: RRM clip validation assisted by the X server.

messages.

When the window layout changes, any window used for virtualized direct access rendering and whose drawable region was affected by the change must have the window's clip invalidated by the X server. A special RRM `ioctl` is used by the X server to communicate clip invalidations to the kernel. The kernel will mark any rendering nodes bound to the window with an invalid clip resource and unmap the virtual pages corresponding to the graphics hardware banks. The next time the process attempts to use the rendering node, the process will generate a page fault. The kernel will determine that the fault corresponds to the address of the unmapped graphics hardware. An RRM message will be sent to the X server to validate the clip of the window. The X server will receive the message, determine the correct clip for the window, and revalidate the window's clip and origin. The kernel will then remap the graphics hardware pages and resume the execution of the process. This entire sequence is transparent to the faulting graphics process. Figure 3 shows an example of how clip validation works.

Multi-rendering Implementation

When the X server first starts, no multi-rendering threads exist. Neither the PEX nor the OpenGL extension have been initialized. The main X server process creates a rendering thread for a client when the client first uses either the OpenGL or PEX extensions. Once created, the rendering thread exists for the lifetime of the client. A per-thread structure associated with the client holds the semaphores, data, and pointers that the main thread and the rendering thread need to communicate with each other. There is at most one rendering thread per client.

Each newly created rendering thread performs some basic initialization, then waits on a blocking user-level semaphore until the main thread commands it to perform some item of work. When the main thread decodes a protocol request that requires access to rendering hardware, it creates a command structure and fills it with information about the protocol request to be executed. The main thread then "hands off" the command structure to the rendering thread by releasing the semaphore that the rendering thread was waiting on. The main thread continues servicing client connections and device input. Request dequeueing

and dispatch for the multi-rendering client are suspended until the main thread is informed that the rendering thread has completed its task.

The mechanism that reports completion back to the main thread must be non-blocking. The main thread must be free to service other clients while multi-rendering is in progress. So when the rendering thread has completed, it alerts the main thread via a pollable semaphore. If the main thread is blocked in `select` when the completion occurs, it is awakened immediately, and resumes request processing for the multi-rendering client. If instead it is busy processing a request for some other client when the completion occurs, the main thread will be informed of the multi-rendering completion the next time it calls `select`.

When a request is handed off to the client's rendering thread, the main server thread "pushes back" the current request on the client's input connection while waiting for completion. This allows completion status to be reported synchronously back to the client.

Because any PEX request can fail and report an error condition, a push-back mechanism is always used for PEX rendering requests. A push-back means that when the client processing is resumed, it is still at the same request. In contrast, most OpenGL rendering does not define protocol replies or errors so the OpenGL extension can avoid push-back for most OpenGL commands.

Pre-existing capabilities in the X server are used to implement the scheme described above [13]. The `IgnoreClient` function is used to suspend processing on the client until the multi-rendering task is completed; `AttendClient` then allows that processing to continue. Block and wakeup handlers are used to modify the X server's `select` mask to add the file descriptors for the pollable semaphores used to signal multi-renderer command completion. The `ResetCurrentRequest` server function performs the push-back operation.

Example of Execution

Figure 4 shows the synchronization that takes place between a client, the X server, and a rendering thread when an OpenGL rendering request is executed.

1. An OpenGL client sends a GLX extension request, which contains a batch of OpenGL commands.
2. The X server receives the request and dispatches it to the GLX extension.
3. The request is handed over to the client's rendering thread. Request input from this client is suspended by calling `IgnoreClient`.
4. The rendering thread wakes up to receive the request.
5. The rendering thread decodes the batch of commands in the request, issuing a call to the OpenGL rendering library for each command. The rendering library attempts to access the graphics hardware pages.
6. Assuming an invalid RRM resource for the rendering node, the page access causes a fault, and the kernel sends an RRM message to the X server for resource validation.
7. The X server receives and dispatches the RRM message.

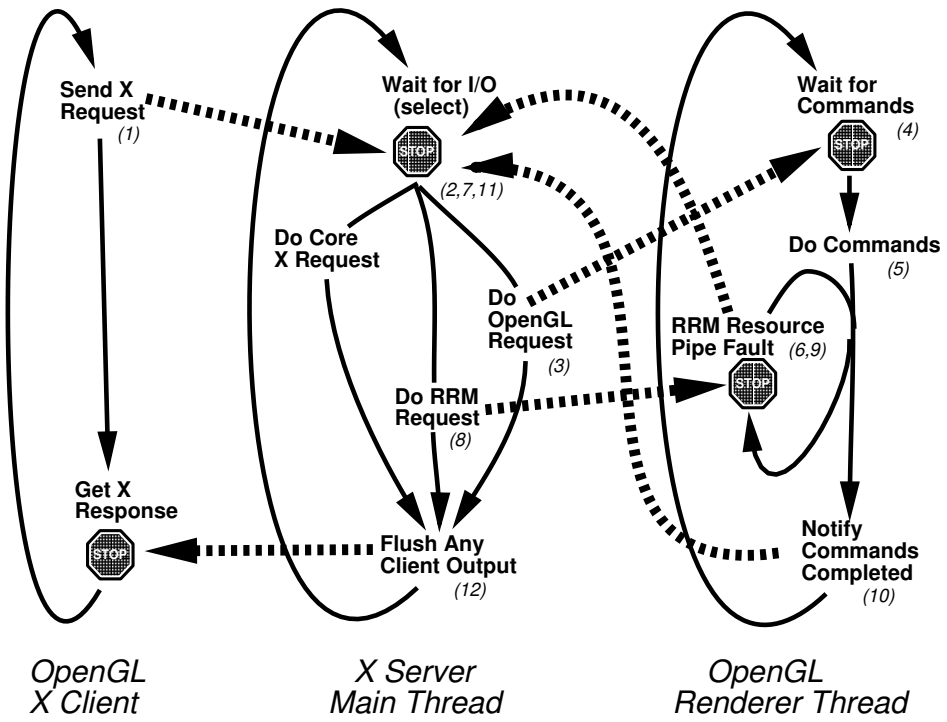

Figure 4: Simplified diagram showing the synchronization points for an X client rendering OpenGL through the X server using multi-rendering. Solid lines represent flow of control; dashed lines indicate where one thread resumes another.

8. Resources are validated and a response is sent to the kernel.

9. The kernel remaps the graphics hardware pages and resumes execution of the rendering thread.

10. The rendering thread announces that it has completed the rendering request by releasing the polling semaphore.

11. The semaphore release causes the X server to wake up from `select`. A wakeup handler notices that the OpenGL thread has finished rendering, and resumes request input from the client by calling `AttendClient`.

12. Returned data, if any, is sent to the client.

Steps 6 through 9 demonstrate the handling of an invalid RRM resource. In this case, the operating system has unmapped the renderer's graphics pipe due to some invalid RRM resource. The X server's main thread and kernel conspire to validate the invalid resource transparently. This is not the common case but does happen when the window's drawable clip changes or the rendering node is requesting a buffer swap and the window does not have an unshared display ID. It is possible for multiple RRM resource validations to occur during a rendering thread's execution.

Note that in Figure 4, there is only a single point where the main X server thread blocks. This allows the main X server thread to avoid deadlock when it is both waiting for a render-

ing thread request to complete *and* handling RRM resource validation requests for blocked rendering threads.

Reliability Assurances

A few basic rules keep the main server thread and the multi-rendering threads from deadlocking or colliding with each other.

- Semaphored memory allocation functions - these keep collisions from occurring in dynamic memory allocation and deallocation routines.
- One thread per client - no matter how many rendering contexts (OpenGL contexts, or PEX renderers) a client has, this ensures sequentiality of requests by only processing one protocol request at a time for each client. Therefore only one rendering thread is needed per client.
- Specialization - in order to minimize the number of locks (and opportunities for deadlock), some tasks are restricted to the main server thread. These include anything that manipulates the server's data structures. For example, any creating, destroying, and looking up X resources is *only* done by the main X server thread. OpenGL and PEX rendering requests are likewise *only* executed by the client's rendering thread.

  This creates certain implementation requirements for extension request processing functions. For example, in the OpenGL extension the main X server thread decodes protocol requests, looks up X resource IDs, and either increments reference counts or copies data from X internal structures so that a structure will never disappear while it is being referenced in a rendering thread. Even when a client connection closes, some data structures cannot be freed immediately. A special interface is provided so that the rendering thread can perform clean-up operations after the client connection is closed; only after the rendering thread clean-up action is done can the main X server thread be allowed to free the client's resources.
- Hand-off and rendezvous - the mechanisms described in the previous section allow the main thread to relinquish control of the client's shared data until it is informed that the rendering thread has completed its task. This cooperative processing eliminates the need for more locks.
- Pure procedures - rendering state data is unique per client. The only static variables allowed are those containing constant data. Any exception to this rule would require a lock, thus defeating potential concurrency among rendering threads.

The GLX extension was implemented with these rules in mind. The GLX code neatly segregates the parts of request processing functions that are performed in the main X server thread in separate source files from the processing done by the rendering thread.

PEX Specific Integration Issues

The PEX extension was not written to adhere to these rules; rather, it was adapted from an external body of code,[3] so the distinction between code executed by the main thread and that executed in the rendering thread was retrofitted into the imported PEX code.

We did this in a way that does not change the directory structure and file names of the imported code, so that future releases of the original PEX code can be easily integrated into our PEX implementation. For each PEX protocol request, we asked the question: does the code to process this request need access to rendering hardware, or need access to data structures owned by the rendering thread? If the answer is no, then that code was left unchanged; it will be executed only from the main X server thread. Code in the other category was divided into the part to be executed in the X server's main thread and the part to be executed in the rendering thread. Fortunately we were able to fit all of the PEX code into this simple model. Where the code originally was something like this template:

```
PEXProcessFoobarRequest(...original args...)
{
    <check original args>
    <look up resources>
    if (<any errors>)
        return (<some error>);

    <process request>
    return (<result of processing request>);
}
```

We changed it to:

```
PEXProcessFoobarRequest(...original args...)
{
    <check original args>
    <look up resources>
    if (<any errors>)
        return (<some error>);

    sgiAsyncExec(ASYNC_PEXProcessFoobarRequest, arg1, ... arg4);
    return (Success);
}

ASYNC_PEXProcessFoobarRequest(arg1, ... arg4)
{
    <process request>
    return (<result of processing request>);
}
```

In this scheme the function `sgiAsyncExec` performs all of the processing necessary to package up the request arguments into a command packet and hand that packet off to the rendering

[3]We chose to base our PEX implementation on Digital Equipment Corporation's PEX product, but that fact is irrelevant to multi-threading. The same techniques that we describe could be applied to the X Consortium's PEX implementation.

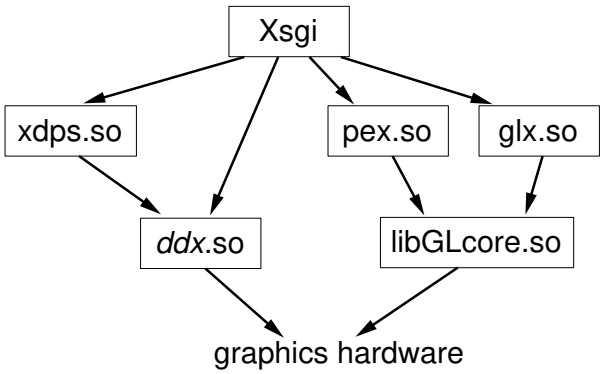

Figure 5: How rendering requests flow from the device-independent main X server module through the dynamically loaded shared objects.

thread, using the push-back mechanism described above. A modification to the PEX dispatch function detects when a pushed-back request has completed, and passes the request completion status back to the client instead of dispatching the request a second time.

This approach works well when the original PEX request processing code looks like the first template above. Where this was not the case, the code was massaged to fit this model. Note that the parameters passed to `sgiAsyncExec` must be simple variables (but not X resource IDs) or pointers to structures which are guaranteed not to change or disappear until the rendering thread has completed its processing.

Dynamic Extension Linking

One of our goals was to support dynamic extension loading. This allows the OpenGL and PEX extensions to be loaded and initialized only when a client first uses either extension. We already supported dynamic loading of the Device-Dependent X (DDX) server code as well as the Display PostScript extension.

To support dynamic loading of an X extension, "stub" code must exist in the X server executable to register the extension. Then the first time an extension request is executed, the stub code uses `dlopen` and `dlsym` to load the extension dynamically. The extension's initialization is performed, the stub extension entry points are replaced with the actual entry points, and the client's request is dispatched.

The OpenGL GLX extension code is contained in a `glx.so` shared object module [9]. This module has all of the code for dispatching GLX protocol and multi-rendering. The actual OpenGL device-dependent code is in the `libGLcore.so` shared object module. This module contains the *exact* same code used by direct rendering OpenGL programs. In fact, SGI OpenGL programs that link with the `libGL.so` OpenGL shared library, implicitly pull in the same `libGLcore.so` object module used inside the X server. The `libGL.so` only contains GLX routines; the OpenGL "proper" routines are in `libGLcore.so`. Once `glx.so` decodes an OpenGL command, it calls the corresponding OpenGL routine in `libGLcore.so` just as a direct rendering program would.

Since our PEX implementation renders via OpenGL, using PEX not only loads the `pex.so` shared object module; it also forces the GLX extension to be loaded. PEX calls OpenGL

System Configuration	Using display lists			Using immediate mode		
	Direct	Indirect		Direct	Indirect	
Onyx, 4 processors	758.0	745.5	(98%)	758.0	258.0	(34%)
Indigo^2 Extreme	143.2	125.4	(87%)	143.1	82.1	(57%)
R4000 Indigo Entry	34.4	29.5	(86%)	35.2	24.0	(68%)
Indy SC	36.4	31.5	(86%)	36.8	23.5	(64%)

Table 1: FlyntStones of OpenGL rendering performance, comparing direct and indirect for both display list and immediate mode rendering.

routines in `libGLcore.so` but also needs multi-rendering support routines in `glx.so`.

Figure 5 shows how the various shared object modules dynamically loaded into the X server interact. A rendering request for core X, Display PostScript, PEX, or OpenGL would logically travel from the main `Xsgi` executable to the graphics hardware by passing through the appropriate request dispatching code to be rendered by either the X device-dependent rendering code or the OpenGL rendering code.

Performance

Multi-rendering has three primary performance objectives. First, as stated in the goals, no measurable overhead should be added when not using multi-rendering. X server benchmarking of core X rendering using `x11perf` shows no measurable performance differences between our X server with multi-rendering support and our X server compiled without multi-rendering support.

The second performance objective is that indirect OpenGL rendering should add minimal overhead compared with direct rendering. To measure the overhead added by indirect rendering, we implemented a benchmark that renders an animated, lighted, shaded, depth buffered hierarchical dinosaur model composed of 261 medium-sized polygons per frame. The 300 by 300 pixel rendering window is cleared every frame. We measured the number of dinosaurs that could be rendered per second (a.k.a. *FlyntStones*).

The dinosaur can be rendered using display lists or in immediate mode. Also the benchmark can be executed using either direct or indirect rendering. When performing indirect rendering using display lists, the client down loads the display lists required to render the dinosaur into the X server; each frame consists of rotating the model-view matrix and executing the dinosaur display list. The time to download the display list is not included in the measurements. The X protocol overhead per frame should be marginal in this case.

When performing indirect, immediate mode rendering, all of the OpenGL commands needed to render the dinosaur must be sent via the X protocol to the X server for execution. The X protocol overhead per frame will be higher than the overhead using display lists.

Table 1 presents performance results spanning the breadth of SGI's current product line [7, 2]. The overhead of multi-rendering when using display lists appears to be under 15%,

except in the case of the multi-processor system where the overhead is a negligible 2% - demonstrating the ability of multi-rendering to utilize multiple processors. As expected, the immediate mode results show a higher overhead due to the added transport and dispatching overhead inherent in immediate mode OpenGL command execution. The penalty for indirect, immediate mode rendering is higher for faster and relatively less host-limited systems like the Onyx.

The final performance objective is an assurance that X server interactivity has indeed been maintained using multi-rendering. Tests running six or more continuously animated, indirect rendering, immediate mode OpenGL programs still allow interactive movement of windows using the window manager, typing into shells proceeds at normal rates, and pop-up menus continue to activate in under a second. The same applies to PEX programs.

Acknowledgments

The authors would like to thank the following Silicon Graphics engineers and managers for their assistance in implementing X server multi-rendering: David Blythe, Peter Daifuku, Kipp Hickman, Deanna Hohn, Phil Karlton, Robert Keller, Todd Newman, Kevin Smith, Mark Stadler, and David Yu.

References

[1] Adobe Systems, Inc., *The Display PostScript System*, Version 1.0, 1991.

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

[3] American National Standards Institute, "Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) Part 4 - Plus Luumiere und Surfaces (PHIGS PLUS)," Draft Proposed Standard X3H31-89-05, July 25, 1989.

[4] Jim Barton, Chris Wagner, "Enhanced Resource Sharing in Unix," *Computing Systems*, Volume 1, Number 2, Spring, 1988.

[5] E.W. Dijkstra, "Cooperating Sequential Processes," *Technical Report EWD-123*, Technological University, Eindhoven, Netherlands, 1965.

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

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

[8] Mike Haynes, et.al., *Component Design Specification for the MIT Multi-Threaded X Window Sample Server*, Version 5.0, The X Consortium, April 15, 1993.

[9] Mark Himelstein, "An Implementation of Dynamic Shared Objects," MIPS Computer Systems, Inc., unpublished, September 17, 1992.

[10] Phil Karlton, *OpenGL™ Graphics with the X Window System*, Version 1.0, Silicon Graphics, April 30, 1993.

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

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

[13] Elias Israel and Erik Fortune, *The X Window System Server*, Digital Press, 1992.

[14] Jackie Neider, Tom Davis, Mason Woo, *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*, Addison Wesley, 1993.

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

[16] Mark Segal, Kurt Akeley, *The OpenGL™ Graphics System: A Specification*, Version 1.0, Silicon Graphics, April 30, 1993.

[17] Silicon Graphics, *Parallel Programming on Silicon Graphics Computer Systems*, Version 1.0, Document Number 007-0770-010, December 1990.

[18] John Allen Smith, "The Multi-Threaded X Server," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Issue 1, Winter 1992.

[19] Doug Voorhies, David Kirk, Olin Lathrop, "Virtual Graphics," *Computer Graphics*, Volume 22, Number 4, August 1988.

[20] Paula Womack, et.al., "PEX Protocol Specification, Version 5.1," The X Consortium, August 31, 1992.