GLR,
an OpenGL render server facility

Mark J. Kilgard

Abstract

GLR is a network-extensible *render service* providing OpenGL rendering into rectangular frame buffer regions. GLR virtualizes access to fast, high-quality, but expensive rendering hardware for purposes such as image processing, printing, frame buffer calculations, and off-loaded high-quality rendering.

In particular, off-loaded rendering using GLR permits expensive high-end graphics hardware to be shared and amortized across a group of users on a local area network. Hybrid GLR programs for applications like computer-aided design and scientific visualization can use local OpenGL rendering for dynamic, interactive 3D graphics, then use GLR to render higher-quality static scenes (enabling higher tessellation, high-quality texturing, antialiasing, etc.) with sub-second latency.

GLR's approach is novel because access to the rendering hardware is virtualized using *render intervals* allowing guaranteed access to the frame buffer resources for a client-requested duration. GLR is implemented as a client library and a specialized X window manager that schedules render intervals. Unlike a traditional window manager, the GLR window manager policy controls the mapping and unmapping of windows to effect the render interval policy. GLR demonstrates that the X Window System model is adaptable for uses other than the traditional window system.

1 *Introduction*

It is now commonplace for services that were originally centralized on a single computer to be distributed across a network of computers. For example, disk access, windowing interfaces,

---

Mark graduated with a B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

interactive terminal sessions, database manipulation, and printing are all routinely done via network services.

Distributing computer services makes it possible to amortize expensive hardware resources among multiple, concurrent users. An entire work group, for instance, can share a single laser printer. Every user benefits from access to high-quality printing, while the associated hardware and maintenance costs are spread among all the users. This same principle can be applied to expensive, high-end graphics hardware subsystems.

This paper describes the GLR network service providing access to high-speed, high-quality 3D rendering and imaging that amortizes the cost of expensive, high-end graphics hardware among multiple users. Underlying the system is OpenGL's GLX protocol and a novel use of the X Window System's client-side window manager capability to implement a scheduling policy for frame buffer access.

Section 2 justifies the need for an OpenGL-based network-extensible render service and describes a model for sharing high-end graphics hardware for such a service. Section 3 describes related work. Section 4 describes the GLR programming interface, the means to implement GLR, and scenarios for GLR's use. Section 5 discusses issues for further investigation, including proposed OpenGL extensions to better support GLR. Section 6 concludes.

2 Justification

Today, most workstations have some 3D graphics and image processing support. The speed, functionality, and quality of this support is primarily dependent on the cost of the machine (particularly, the graphics hardware subsystem). In general, the more you pay the more you get. Using the Silicon Graphics product line as an example, an Indy desktop workstation [11] configured with good low-end 3D capabilities costs around $10,000, while an Onyx multi-processor deskside machine with RealityEngine graphics [2] costs $100,000 or more. This order of magnitude difference in system cost represents a substantial difference in interactive 3D graphics quality, capability, and performance demonstrated by these "ballpark" comparisons of the RealityEngine to the Indy's standard graphics subsystem:

- 36 bits of color resolution per pixel versus 12 bits (double buffered).
- Hardware antialiasing through multisampling versus no fast antialiasing support.
- Hardware stencil, depth, alpha, accumulation, and off-screen rendering buffers versus software buffers.
- Hardware texture mapping using high-quality tri-linear mipmapped textures, including 3D hardware texture support for volume rendering, versus texture mapping only through software.
- Almost two orders of magnitude difference in the transform rate for 3D geometry.
- An order of magnitude or more difference in the raw pixel fill rate.
- Well over two orders of magnitude difference in the depth buffered, textured fill rate.

Even while new low-end and mid-range graphics subsystems promise improved performance and functionality, the rate of improvement is likely to be matched or exceeded by improvements in high-end rendering systems. Some high-end features like multisampling are unlikely to be available in low-end hardware in the foreseeable future.

Applications like visual simulation, large-scale computer aided design (CAD), digital film editing, scientific visualization, virtual reality, and medial imaging use high-end graphics systems to meet their demanding requirements for speed, features, and high-quality rendering. In particular, high frame rates, high CPU-to-graphics bandwidth and low update latency are vital to these applications. These demands are summarized by the word *immediacy*. Typically, applications that demand immediacy tend to monopolize the entire machine.

To justify such hardware expense, these applications must have a very high return on investment. The advantage of open heart surgery rehearsal or training commercial and fighter jet pilots via computer simulation justifies the expense, but this model for utilizing high-end graphics hardware limits the applications and markets for such hardware.

The monopolistic model of graphics hardware allocation is too expensive for many application domains. Where immediacy is not a key requirement, a network-extensible *render service* can permit multiple applications to concurrently share a high-end graphics subsystem. Conceptually, such a service lets clients render sophisticated 3D graphics and retrieve the resulting image for display on a low-end workstation or other manipulation. In the same way networked laser printers permit hardware cost to be amortized across multiple users and applications, a networked render service amortizes the cost of high-end graphics hardware and offers higher levels of hardware utilization.

2.1 Render Service Applications

A hardware-accelerated render service is appropriate for a wide range of rendering applications including:

Hybrid 3D viewers. Combining the interactivity of inexpensive low-end 3D workstations with a render server permits hybrid applications that trade-off interactivity and rendering quality as appropriate to the application. When the user of such a hybrid application interacts with a 3D model, a simplified representation can be rendered directly by the low-end workstation to achieve fast, interactive display rates (by disabling texture mapping, using coarse tessellation, rendering as wireframe, etc.). However, the render service can provide detailed, textured, antialiased renderings at sub-second latency whenever the model is displayed statically.

For example, a computer aided design program might show the design in wireframe when the user manipulates the design, but when the user stops the manipulation, the design is quickly re-rendered with more detail. Other similar applications include VRML browsing and volume rendering.

3D printing. Printing 3D graphics is complicated because the programming interfaces targeted for the display of interactive 3D graphics rarely have good support for printing.

Demands for performance, fidelity to the on-screen appearance, and the higher resolutions often demanded for printing all make printing 3D scenes difficult. A render service can address these issues by off-loading rendering to fast 3D hardware with large rectangular frame buffer regions.

An application can redirect its on-screen rendering commands to a render service, then read back the rendered image to send to a printer. A print server could automatically convert Inventor or VRML metafiles submitted for printing to a 2D image using a render service and then pass the resulting image to the printer.

Movie sequence rendering. A succession of 3D scenes forming a computer animated movie could be rendered with reasonably high quality by a render service. For example, animators at a computer animation firm could submit movie rendering jobs to the render service, and later retrieve the rendered movie sequences. This lets animators review their work rendered quickly at good quality levels before submission for slower, expensive, higher quality final rendering.

Fast data and image analysis. Many applications such as satellite image analysis, medical imaging, and seismic analysis use image processing and 3D rendering techniques like volume rendering [3], 3D geometry recognition from 2D images, and shadow map generation [9] that can benefit from fast 3D hardware supporting image processing and high-quality texture mapping capabilities but do not require or expect immediacy, making these applications suitable for scheduling through a render service.

While they do not demand the immediacy that justifies an expensive 3D graphics hardware accelerator, all these applications can benefit from access to such hardware. A render service that amortizes the hardware cost across multiple application instances while delivering improved performance can make the use of expensive hardware viable for these applications.

What these various applications share is a common demand: accelerated rendering access to rectangular frame buffer regions. This should be the fundamental service provided by a render service.

2.2 OpenGL as a Basis for a Rendering Service

A render service needs a rendering model and protocol to communicate rendering requests. OpenGL [10] is a widely adopted programming interface for high-performance, interactive 3D graphics. The OpenGL graphics system has features that make it well suited for use as the underlying rendering model and protocol of a render service:

- OpenGL's X Window System extension known as GLX [4] already provides a standard, inter-operable protocol for issuing OpenGL rendering commands.
- OpenGL is window system independent, making it possible to specify a window system independent render service programming interface.
- OpenGL has existing extensions (with associated GLX protocol) for advanced rendering capabilities like multisampling and 3D textures, as well as image processing support for operations like convolutions and histograms.

- OpenGL is widely available and is an accepted standard with existing, well-tuned high-end implementations. Low-end support for OpenGL makes possible hybrid rendering schemes combining local OpenGL rendering results with render service usage.
- OpenGL cleanly separates rendering state from frame buffer state.
- OpenGL is amendable to hardware acceleration [7].
- OpenGL display lists and texture objects[1] allow caching server-side command sequences and textures, reducing transport overhead during rendering.

The GLR render service uses OpenGL's programming interface so access to the graphics hardware can be shared by various clients requesting high-performance rendering. The programming interface and implementation of GLR are discussed in Section 4. Using GLR, multiple clients on a network (including clients running on the same machine that provides the render service) can share a single, expensive graphics subsystem for their sophisticated rendering needs.

Typically, a GLR client renders a scene using OpenGL commands, then retrieves the rendered image from the frame buffer using OpenGL's `glReadPixels`. The retrieved image can then be displayed on a low-end workstation's screen, saved to a file, directed to a printer, or otherwise manipulated.

2.3 Render Service Arbitration and Scheduling Demands

A fundamental problem that a render service must address is how to arbitrate and schedule access to the render service. In practice, multiple render service clients compete for a limited amount of frame buffer real estate and graphics subsystem processing power. Solving this problem demands consideration of the nature of the service provided.

The problem of frame buffer real estate arbitration and scheduling of access to the graphics hardware is not unique to GLR. Window system services like X address this same problem, but the constraints are different. Typically, window systems arbitrate frame buffer access using overlapping windows. By clipping rendering to the visible regions of overlapping windows, a window system only requires that frame buffer state be maintained for exactly the number of pixels visible on the screen (no more, no less). Conveniently, pixel state for obscured window regions need not be maintained. More importantly, all visible window regions are always maintained so they can be displayed on a user's monitor.

In addition to frame buffer real estate arbitration, window systems must schedule graphics hardware access for window system clients. Typically, window systems accomplish this by simply interleaving rendering and other window system requests from multiple clients. To maintain the user's impression of simultaneous operation of various client applications, interleaving must happen at interactive rates.

[1] Texture objects encapsulate a texture map and its parameters; unlike textures embedded in display lists, modifications to the texture map and its parameters are possible. Texture objects are provided in OpenGL version 1.1 (and as an extension to 1.0).

The methods appropriate to a window system service for arbitrating frame buffer real estate and scheduling hardware access are not appropriate for a render service. Window system clients take continuous responsibility for updating newly visible regions of windows, and the overlapping is important to the user's "desktop" impression of access to multiple simultaneous graphical applications. And when scheduling access to the graphics hardware, interactivity is the key concern for window systems.

However, different priorities exist for render services in contrast to window systems. A render service must provide a client access to a rectangular frame buffer region of the requested size. This access should be provided independent of concurrent requests by other clients. The service must provide a guarantee of how frame buffer state is maintained. In practice, this means no overlapping of frame buffer rectangles requested by multiple clients. Also, issuing redraw events is inappropriate. Additionally, it is throughput, not interactivity, that is more important for a render service.

With a window system, the frame buffer is continuously scanned out (displayed) to a computer screen; this is fundamental to the purpose of a window system. In contrast to a window system, a displayed frame buffer is *not* necessary for a render service.[2] This also reflects something fundamental to the nature of a render service. For a render service, rendering to the frame buffer must be read back for the results of the rendering to be useful; displaying of the frame buffer is not required.

Window systems virtualize frame buffer real estate spatially by overlapping windows and virtualize graphics hardware access temporally by fine-grain interleaving. For a render service however, spatial virtualization of screen real estate has limited applicability (tiling is possible, but not generally sufficient), but because a coherent viewable screen image is not necessary to a render service, temporal virtualization of frame buffer real estate (scheduling access to the frame buffer in time) is a viable technique. This scheduling of frame buffer real estate can be coarse-grain since interactivity is not a priority. The scheduling of access to the graphics hardware itself can also be done on the same coarse boundaries.

Conceptually, use of a render service is done as a series of renders, each followed by frame buffer read backs:

> render, read back;
> render, read back;
> render, read back ...

These render and read back pairs provides good boundaries for render service interactions by clients. Because the computer frame buffer state is read back, it is typically not necessary to retain frame buffer data for the client across these boundaries. This assertion is important because otherwise frame buffer state might need to be swapped in and out of the frame buffer (expensive!) when different render service clients share the same physical frame buffer state. For 3D scene rendering, the assertion is reasonable because 3D rendering of a scene typically

[2]In fact, it may be undesirable to the extent displayability adds material cost to the system. The cost of a high-end graphics subsystem used for a render server may be reduced considerably if no display monitor or display generation hardware is required.

does not rely on the previous scene. In the unusual situation when the assertion is not true, a client can restore the read back state from the previous rendering at the beginning of the next rendering.

2.4 GLR Render Intervals

This approach to virtualizing frame buffer real estate and graphics hardware access for a render service is implemented by GLR's *render interval* mechanism.

A render interval is an interval (of real time) during which a GLR client may perform OpenGL rendering to a rectangular frame buffer region for a guaranteed minimum duration. The interval is explicitly begun and ended by a GLR client. The guaranteed minimum duration and rectangle size are specified by the GLR client when the interval is initiated. The guaranteed minimum duration is bounded by an upper-limit determined by the GLR server.

Multiple render intervals may be initiated concurrently by different GLR clients. The GLR server determines which render interval (or possibly multiple intervals) should be active. When a scheduled interval is activated, the state for the interval's rectangular frame buffer region is maintained, and the client can issue OpenGL rendering commands to render to and read back from the frame buffer.

If a GLR client with an active interval exceeds the guaranteed minimum duration of the interval *and* another client's render interval is waiting to be made active, the GLR server expires the active client's interval. When an interval is expired, its associated frame buffer state is no longer retained. No immediate notification of the expiration is sent to the client. A client with an expired interval can continue to render though the frame buffer ceases to be updated.

When a GLR client with an active interval ends its interval, the client is informed if the interval was expired. If the interval was expired, the client can make no assumptions about any frame buffer state read back during the interval.

The purpose of expiring render intervals is to ensure that a misbehaved (or simply unexpectedly slow) client will not prevent access by other clients to the render server for arbitrary periods of time. The server's maximum allowed duration ensures other clients will get fair access to the server. Conceptually, GLR render intervals are a type of *transaction processing*; the rendering "transaction" must be completed within the pre-arranged time or the transaction is aborted (render interval is expired).

GLR clients are expected to re-perform render intervals that are expired. Using a measurement of the duration of the expired interval, the client is assumed to have a better estimate of how long a render interval to request. GLR assumes the client can reproduce the rendering requests again.

GLR clients should always be prepared for a render interval to be expired. Intervals may be expired for events beyond the control of the client (network lapses or the client was a paused by the user in the middle of a render interval).

A few important points about GLR render intervals:

- It is only frame buffer state that is lost when a render interval is expired; OpenGL rendering state (including display lists and texture objects) is retained across multiple render intervals.
- GLR provides a means to set up rendering state (including display lists and texture objects) outside of a render interval.
- When multiple GLR clients are concurrently requesting render intervals, the server schedules render intervals not only based on the order the intervals were scheduled, but also on the requested duration of each interval. This scheduling policy encourages clients to accurately estimate their expected rendering time.
- The time spent within a render interval is returned whether the interval succeeds or was expired. This information not only lets the client better estimate expired intervals, but also better bound the time required for successful intervals to the client's scheduling advantage. Generally, clients are well served to add a "bit of slop" to requested durations to avoid unexpected slight timing anomalies.
- Even when there is contention due to multiple clients for the scheduling of render intervals, clients are essentially unaware of this contention (except that contention may cause an expiration that might not otherwise have occurred, something all clients should prepare for).
- Nothing forces a client to read back frame buffer state or specifies what frame buffer state should be returned. The client is free to read back subregions of the allocated region, read back only the green pixel components, read back ancillary buffer state like depth buffer values, etc.

Further details about GLR render intervals are left for Section 4.

3 Related Work

3.1 Adobe's Display PostScript NX

Adobe's Display PostScript NX facility [1] is a render service based on the PostScript rendering model. The NX facility provides transparent support for the Display PostScript Extension to X.

The primary purpose of Adobe's render server is to make up for X servers that lack direct support for the Display PostScript extension. This is in contrast to the primary purpose of GLR which is to provide access to high-performance 3D graphics hardware in a way that amortizes the cost of such hardware across multiple users. GLR can be used to obtain both better functionality (high-end features not available on low-end workstations) and improved performance.

Because GLR uses OpenGL (a pixel-oriented, interactive 3D oriented rendering model) and the NX facility uses PostScript (a resolution-independent 2D oriented rendering model), these render services address different needs.

Frame buffer mechanism	Volatility	Non obscured	Hardware accelerated	Frame buffer state allocation
X windows	yes	no	yes	windowed
GLX pixmaps	no	yes	no	fixed
volatile pbuffers	yes	yes	yes	volatile
non-volatile pbuffers	no	yes	yes	tiled/swapped
GLR render intervals	timed	yes	yes	tiled/timed

Table 1: Comparison of OpenGL frame buffer mechanisms.

3.2 Distributed Rendering

Because rendering has many opportunities for exploiting parallelism, considerable work has been done distributing rendering across a network of computers for better performance. Typically the rendering algorithms used are based on radiosity or ray tracing [8] and suited to general purpose processors instead of accelerated 3D graphics hardware. OpenGL was the rendering interface used by DECS/OpenGL [5], but the prototype was to demonstrate generic distributed computing with OpenGL rendering as a vehicle, not to build a production OpenGL implementation suitable for use as a render service.

In contrast to GLR's approach of *exporting* a network-extensible service for accessing and amortizing the cost of high-performance 3D graphics hardware, distributed rendering *creates* such a rendering resource from a network of general purpose processors. While distributed rendering may be suited for relatively slow-to-render and computationally expensive scenes generated using raytracing and radiosity techniques, GLR will likely offer superior performance when interactive rendering techniques suffice.

3.3 Render Service Transport Compression

Levoy [6] has considered techniques for using a high-end graphics workstation as a rendering server for a low-end workstation in a manner consistent with GLR's approach. Levoy specifically addresses the network transport cost of transporting raw images across a network connecting the workstations. Levoy proposes a polygon-assisted compression scheme to minimize the required network bandwidth.

The approach relies on sending compressed image differences from the render server to the client. Potentially, this technique is applicable to GLR.

3.4 Other OpenGL Frame Buffer Access Mechanisms

Conceptually, GLR render intervals are yet another mechanism for OpenGL rendering to a frame buffer. Table 1 compares key features of currently available OpenGL frame buffer mechanisms. OpenGL pixel buffers (or pbuffers) [12] exist as an experimental GLX extension implemented by Silicon Graphics for rendering into hardware accelerated, off-screen frame buffer memory.

The frame buffer state of a *volatile pbuffer* may be revoked at any time (inconvenient for applications). The frame buffer state of a *non-volatile pbuffer* is maintained always (expensive to implement and undermines sharing by multiple applications). A non-volatile pbuffer is like a render interval with an infinite timeout; a volatile pbuffer is like a render interval with a zero timeout. Render intervals represent a compromise between complete volatility of frame buffer state and total non-volatility.

4 The GLR Interface and Implementation

4.1 The Programming Interface

The GLR programming interface has a C language binding. GLR routines begin with the `glr` prefix; GLR types begin with the `GLr` prefix; GLR constants begins with the `GLR_` prefix. To include prototypes for GLR routines:

```
#include <GL/glr.h>
```

4.1.1 GLR Setup

To use GLR, a client application first establishes a GLR *session*:

```
GLrSession session;
session = glrOpenSession(NULL);
```

When `NULL` is passed to `glrOpenSession`, the GLR server hostname is determined by the `GLR_SERVER` environment variable. Otherwise, the string names the GLR server to connect to; the empty string connects to a GLR server on the local machine. The `GLrSession` returned is passed to subsequent GLR calls; however, a `NULL` return value indicates failure. `glrCloseSession` is passed an open `GLrSession` to be closed down.

Given a GLR session handle, the client can allocate a GLR *canvas type*. An attribute list is supplied that describes the minimum set of frame buffer capabilities requested. For example:

```
GLrCanvasType canvasType;
int atts[] = { GLR_RGBA, GLR_DEPTH_SIZE, 16,
  GLR_RED_SIZE, 1, GLR_GREEN_SIZE, 1, GLR_BLUE_SIZE, 1,
  GLR_SAMPLE_SGIS, 4 /* multisampling */, 0 };

canvasType = glrGetCanvasType(session, atts);
```

A GLR canvas type roughly corresponds to an OpenGL-capable X visual. `glrGetCanvasType` operates much like GLX's `glXChooseVisual` routine. If `glrGetCanvasType` returns `NULL`, a canvas type meeting the attribute list requirements was not available.

Once a canvas type is gotten, a GLR *canvas* can be created by calling `glrCreateCanvas`:

```
GLrCanvas canvas;
canvas = glrCreateCanvas(canvasType, NULL);
```

The second parameter can be used to name an existing GLR canvas with which display lists are shared. Unlike GLX in which a GLX drawable and context are two distinct entities, GLR ties together rendering context state with frame buffer state. The set of frame buffer capabilities supported by a canvas is determined by the canvas's canvas type.

Calling `glrEstablishRenderState` lets a client make OpenGL calls to establish OpenGL render state for a GLR canvas (without retaining frame buffer state). For example:

```
int success;
success = glrEstablishRenderState(canvas);
if (!success) {
  /* failed */
}
```

Subsequent OpenGL calls (until another OpenGL context is selected) affect the canvas's OpenGL state machine. During this period, OpenGL state can be updated and retrieved (including creating display lists and use of selection and feedback). Because the canvas's frame buffer state is retained only within a render interval, any frame buffer state returned by an OpenGL command (for example, `glReadPixels`) is undefined.

4.1.2 Using GLR Render Intervals

Frame buffer state is retained during a render interval (or until the interval is expired). The interval begins with `glrBeginRenderInterval` and ends with `glrEndRenderInterval`:

```
int width = 400, height = 400;
int timeout = 500;               /* milliseconds */
int wait = -1;                   /* negative means forever */
int duration;                    /* returned as milliseconds */

success = glrBeginRenderInterval(canvas, width, height, timeout, wait);
/*
 * Perform OpenGL operations.
 */
success = glrEndRenderInterval(canvas, &duration);
```

In the above example (assuming the interval is successfully begun), a 400 by 400 pixel unclipped frame buffer rectangle would be guaranteed retained for at least 500 milliseconds (of real time) once the render interval is scheduled. The `wait` parameter determines how long to wait for the render interval to be scheduled. Zero means return immediately if the interval can not begin immediately; a negative value means wait forever; otherwise, wait for the specified number of milliseconds.

The value returned by `glrEndRenderInterval` indicates if the render interval was expired. If the interval was expired, frame buffer state retrieved within the interval cannot be assumed correct. The duration returned by `glrEndRenderInterval` indicates the duration of the interval from its initiation (not including time waiting for the interval to be scheduled) and `glrEndRenderInterval` being called.

GLR routine	Xlib, GLX, and C library routines used to implement it
glrOpenSession	XOpenDisplay, XGetWindowProperty
glrGetCanvasType	glXChooseVisual
glrCreateCanvas	XCreateWindow, glXCreateContext
glrBeginRenderInterval	XChangeProperty, glXMakeCurrent, XMapWindow, XPending, select, gettimeofday
glrEndRenderInterval	XPending, XUnmapWindow, gettimeofday
glrEstablishRenderState	glXMakeCurrent
glrGetMaxCanvasWidth	DisplayWidth
glrFreeCanvasType	free
glrDestroyCanvas	XDestroyWindow, glXDestroyContext, free
glrCloseSession	XCloseDisplay, free

Table 2: Primary Xlib, GLX, and C library routines used to implement selected GLR client library routines.

This interval duration can be used to determine by how many milliseconds the render interval timeout was exceeded or over-estimated. This time may be of use when resubmitting an expired render interval with a longer render interval duration or for better estimating similar, future render intervals.

A more involved example of using render intervals is presented later in Section 4.3.

Sometimes, a client may want to determine if the current render interval has expired without ending the render interval itself. This is particularly useful if a long render interval has been scheduled. The expiration status of the current interval can be checked like this:

```
stillGood = glrQueryRenderIntervalStatus(canvas);
if (stillGood == 0) {
  /* Interval has expired.  glrEndRenderInterval guaranteed to fail. */
} else {
  /* State read back before glrQueryRenderIntervalStatus is valid. */
}
```

4.1.3 Remaining Routines

In case another OpenGL context is bound during a render interval (for example, calling `glXMakeCurrent` for non-GLR OpenGL rendering during a render interval), a process can resume OpenGL command dispatching for the current render interval by calling `glrResumeRenderInterval`.

The GLR server may establish limits on the maximum guaranteed interval duration and the maximum canvas width and height (in pixels). These limits by be queried by calling `glrGetTimeoutLimit`, `glrGetMaxCanvasWidth`, and `glrGetMaxCanvasHeight` respectively.

To destroy a canvas, call `glrDestroyCanvas`. A canvas type may be freed by calling

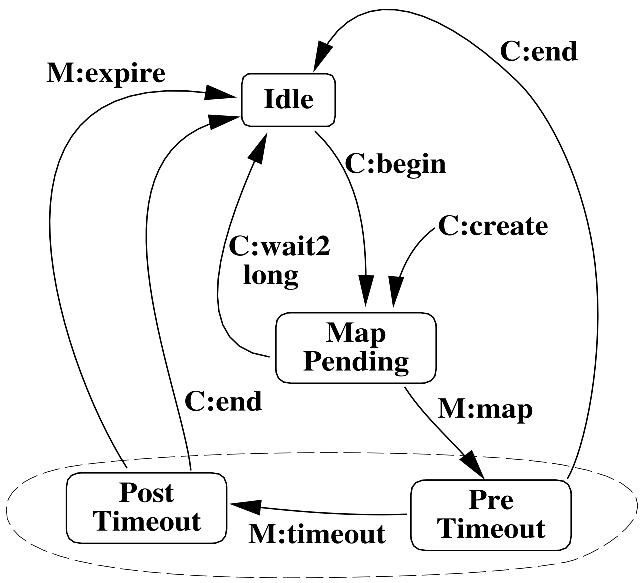


Figure 1: GLR window state transitions. Dashed-lasso indicates states when windows are mapped. "C" prefixed transitions indicate GLR client initiated transitions. "M" prefixed transitions indicate GLR manager initiated transitions. Destroy transitions can happen from any state.

`glrFreeCanvasType`. A GLR session may be closed by calling `glrCloseSession`.

4.2 GLR Client Library and Server Implementation

The GLR programming interface does not expose its implementation through its programming interface. This enables future GLR implementations to be implemented differently, but the discussion below describes GLR's current implementation.

The client interface is implemented as a shared library that utilizes the Xlib and GLX interfaces. The GLR server is implemented as the combination of an OpenGL/GLX-capable X server and a special X window manager client called `glrmanager` that enforces the GLR arbitration and scheduling policy. The client library and `glrmanager` have an agreed protocol for requesting, scheduling, and terminating render intervals. The X server needs no special modifications to support GLR. The client library and `glrmanager` implementations are quite portable and use no vendor-specific features. Table 2 shows the important underlying routines used to implement various GLR routines.

`glrmanager` selects substructure notify events on the root window like all X window managers, but unlike standard window managers, `glrmanager` enforces GLR's policy for render intervals. Figure 1 shows the states that `glrmanager` will move a client's X window through when a client schedules a render interval.

An interval is scheduled when a client attempts to map a GLR canvas's X window. When `glrmanager` actually maps the window, the client's render interval begins, indicated to the client by the delivery of a `MapNotify` event from the X server. To expire a window (i.e., invalidate its frame buffer state), `glrmanager` simply unmaps the window. When

```
if (sceneChanged) {
  do {
    glrBeginRenderInterval(canvas, width, height,  timeout, waitForever);
    renderVolume();
    glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, image);
    success = glrEndRenderInterval(canvas, &timeout);
    timeout = timeout * 1.4;  /* over-estimate by adding slop */
  } while(!success);
}
glXMakeCurrent(indy_display, window, context);
glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image);
glXSwapBuffers(indy_display, window);
```

Figure 2: Transformed code fragment for a hypothetical volume visualization program transformed to off-load expensive volume visualization rendering to a GLR server, then displaying the resulting image on a low-end workstation. Note that full RGB image data is read back, instead of only luminance.

`glrEndRenderInterval` is called, a GLR client determines if its render interval succeeded based on whether the X window for its canvas received any `UnmapNotify` events.

4.3 Usage Scenarios and Applications

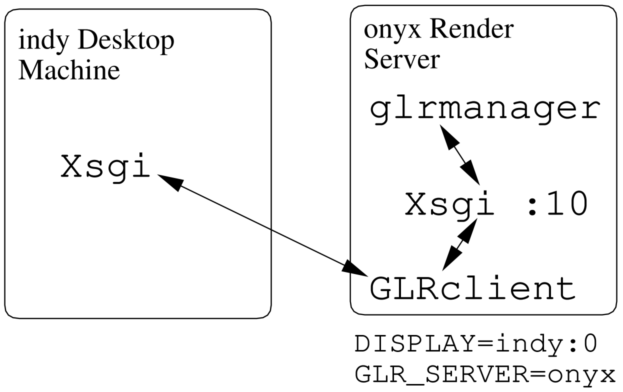
To test GLR's implementation, several sample GLR clients were implemented. The most elaborate is a GLR version of Brian Cabral's `volren` real-time volume renderer[3] using 3D texture mapping to view a human skull data-set. In its original form, the application must run locally and requires RealityEngine graphics. After tuning the GLR version, including minimizing the retrieved region and only reading back luminance data instead of RGB (the skull volume is not rendered in color), the application displayed the skull data-set to an Indy connected by ethernet at 3-4 frames per second versus 10-15 frames per second for the original `volren` displaying locally on the RealityEngine. This result is remarkable because high quality volume visualization is not otherwise available on an Indy workstation. The GLR version is limited by network transport overhead so increased network bandwidth (like FDDI) would improve the display rate.

What are the essential modifications to `volren` to make it utilize GLR? The original `volren` has a redraw routine functionally equivalent to:

```
glXMakeCurrent(onyx_display, window, context);
renderVolume();
glXSwapBuffers(onyx_display, window);
```

The `renderVolume` routine executes the expensive volume visualization rendering algorithm

[3] Cabral's original `volren` source code is available at `ftp://sgigate.sgi.com/pub/demos/`



Scenario 1: GLR client runs on capable server (multi-processor, fast disk I/O) and renders direct GLR images; X and GLR images (via GLX) sent to desktop X server. Examples: volume rendering; data- or CPU-intensive scientific applications.

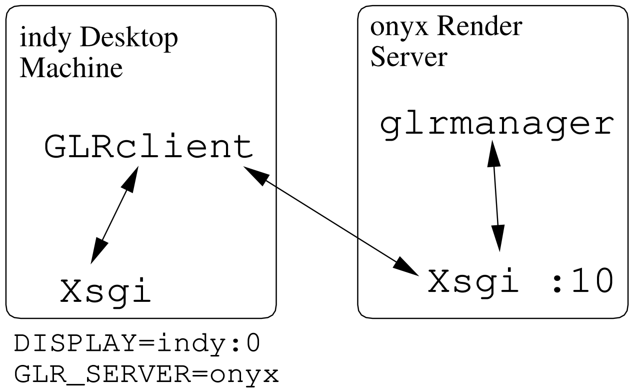
using sophisticated OpenGL extension routines for 3D texture mapping. To off-load this rendering algorithm to a GLR server, the code would be transformed as shown in Figure 2.

The code assumes a GLR session has been established to a high-end GLR server capable of executing the `renderVolume` via accelerated hardware. The transformed code uses GLR render intervals to execute `renderVolume` and read back the resulting frame buffer image with `glReadPixels`. In the case of an expired render interval, the render is retried (with a hopefully better estimated duration interval) until it succeeds. The successfully retrieved image is then displayed on the low-end workstation by calling `glDrawPixels`. As an optimization, the image is only re-rendered using a GLR render interval if the scene has changed since the last image rendered.

Because of the disk bandwidth and host-to-RealityEngine bandwidth required, and the multiprocessing requirements of `volren`, the GLR version runs on the Onyx, but displays to the Indy as shown in Scenario 1.

GLR is flexible as to where the GLR application can run. GLR lets the application run where most efficient (on the high-end workstation in the case of `volren`) while still displaying to a low-end workstation. Scenario 2 displays the reverse configuration where the GLR application runs on the low-end workstation.

Using Scenario 2, a second program called `glrduckpond` uses Open Inventor (which renders via OpenGL) to draw a duck swimming in a pond. The user can interactively rotate the duck around the pond with the mouse. This application demonstrates a *hybrid* GLR application. When the duck is moved interactively, the duck is rendered on the local Indy workstation but at a coarse tessellation. When the user stops rotating the duck, the duck is re-rendered using GLR on a RealityEngine with very fine tessellation and multisampling enabled, typically in less than half a second for a 300 by 300 pixel RGB window. The resulting antialiased and highly tessellated scene could not have otherwise been rendered as quickly on an Indy. The program demonstrates how GLR off-loaded rendering can be effectively combined with local rendering to trade-off high-quality rendering with good interactivity.



Scenario 2: Desktop GLR client connecting to a remote GLR server. GLR client can interactively spin/manipulate model locally, but fall back to GLR for high quality static renders. Examples: CAD applications, 3D modelers, molecule viewers, VRML viewers.

5 Issues for Further Investigation

Improving the performance of GLR based applications depends primarily on reducing network bandwidth. Also, more investigation is needed of how multiple concurrent GLR applications interact and what GLR server and client policies can best reduce contention and the time spent waiting for a rendering interval to be scheduled.

The rendering network bandwidth (the OpenGL commands sent to render the scene) can be managed through careful use of display lists, texture objects, and state set-up outside of render intervals using `glrEstablishRenderState`. This implies that GLR's performance is largely limited by the network bandwidth to read back images.

5.1 Sharing and Context Switching

One issue that arose during GLR's development was that considerable overhead could be introduced by graphics hardware context switching. An early version of `glrmanager` attempted to tile frame buffer rectangles within the physical frame buffer to ensure no overlap. This permits multiple render intervals to be simultaneously active.

The result was a substantial decrease in overall performance when there was contention from multiple GLR clients due to the overhead of context switching the graphics hardware state. High-end graphics subsystems tend to be very stateful and context switching is therefore relatively expensive. Multiple scheduled GLR render intervals meant context switching occurred. If `glrmanager` instead only schedules a single GLR render interval at a time, context switching overhead is minimized and overall throughput can be improved.

In addition to context switching of basic hardware rendering state, swapping of texture memory can be particularly expensive. Multiple instances of `volren` (both the original *and* the GLR capable version) will oversubscribe a RealityEngine's total texture memory. The resulting texture memory swapping greatly slows rendering performance.

For traditional high-end graphics applications that tend to monopolize the graphics hardware, context switching and texture swapping overhead is not an issue, but for systems to be used as render servers, sharing the hardware via GLR is likely to create more context switching and texture swapping overhead. Future high-end graphics subsystem designs should place more importance on minimizing context switching overheads.

5.2 GLR Support via OpenGL Extensions

OpenGL is extensible so it is appropriate to consider what extensions to OpenGL would improve the perceived and actual performance of GLR. Possibilities include:

- Compressed image formats to compress pixels read from and uncompress pixels displayed to the frame buffer.

- An extension to add a *scan-line skip count* (0 by default) indicating how many scan-lines to skip between every scan-line drawn to or read from the frame buffer. A scan-line skip count of 1 would permit even and odd scan-line interleaving. A GLR client could read back all the odd scan-lines (or every third scan line, etc.) and immediately display them; afterword, read and display the even scan-lines. This incremental display might give the GLR application user faster perceived feedback.

- Using OpenGL's existing *copy texture* extension, perform staged read backs by reading back the frame buffer state into a texture object, end the render interval, call `glrEstablishRenderState`, and then (outside the render interval) read the texture object's contents (possible because texture objects are render state, not frame buffer state) back to the GLR client. This allows another render interval to be scheduled while the previous client is still reading back its frame buffer state.

- The described GLR implementation based on `glrmanager` dedicates the high-end graphics subsystem to GLR. This is a serious disadvantage because a machine set up to work as a GLR render server will not be available for normal window system sessions. If GLR render intervals can be implemented using OpenGL pixel buffers instead of windows so that render intervals would use off-screen memory for their rendering, the graphics subsystem might be able to support both GLR and a window system simultaneously.

5 Conclusions

GLR demonstrates a novel use of the X Window System and OpenGL to construct a high-performance render service. GLR's render interval approach virtualizes access to the rendering performance and frame buffer state of a high-end graphics subsystem so the hardware can be amortized by a network of users.

References

[1] Adobe Systems, *Display PostScript NX Software Concepts and Facilities*, Release 1.0, June 1, 1993.

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

[3] Brian Cabral, Nancy Cam, Jim Foran, "Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware," *Symposium on Volume Visualization*, 1994.

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

[5] Joseph Kiniry, *DECS: A Distributed Enterprising Computing System, DECS/OpenGL: A Generic Distributed Graphics Library*, M.S. Thesis, Department of Computer Science, University of Massachusetts, November 1994.

[6] Mark Levoy, "Polygon-Assisted JPEG and MPEG Compression of Synthetic Images," SIGGRAPH Conference Proceedings, August 1995.

[7] Mark J. Kilgard, "Hardware for Accelerating OpenGL," *The X Journal*, SIGS Publications, September/October 1995.

[8] Kwan-liu Ma, James Painter, Charles Hansen, Michael Krogh, "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," *Parallel Rendering Symposium Proceedings*, October 1993.

[9] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, Paul Haeberli, "Fast shadows and Lighting Effects Using Texture Mapping," *SIGGRAPH Conference Proceedings*, 1992.

[10] Mark Segal, Kurt Akeley, *The OpenGL™ Graphics System: A Specification*, Ver. 1.0, Silicon Graphics, June 21, 1995.

[11] Silicon Graphics, "Indy Graphics," *Indy™ Technical Report*, Ver. 1.0, 1993.

[12] Paula Womack, Kurt Akeley, "OpenGL pixel buffer extension specification (experimental)," Silicon Graphics internal specification, November 10, 1995.