Realizing OpenGL: two implementations of one architecture

Article · January 1997

DOI: 10.1145/258694.258711

CITATIONS
20

READS
627

1 author:

Mark J. Kligard
NVIDIA
20 PUBLICATIONS **832** CITATIONS

SEE PROFILE

Published in the 1997 SIGGRAPH Eurographics Workshop on Graphics Hardware

Realizing OpenGL: Two Implementations of One Architecture

Mark J. Kilgard
Silicon Graphics, Inc.

Abstract

The OpenGL Graphics System provides a well-specified, widely-accepted dataflow for 3D graphics and imaging. OpenGL is an *architecture*; an OpenGL-capable computer is a hardware manifestation or *implementation* of that architecture. The Onyx2 InfiniteReality and O2 workstations exemplify two very different implementations of OpenGL. The two designs respond to different cost, performance, and capability goals.

Common practice is to describe a graphics hardware implementation based on how the hardware itself operates. However, this paper discusses two OpenGL hardware implementations based on how they embody the OpenGL architecture. An important thread throughout is how OpenGL implementations can be designed not merely based on graphics price-performance considerations, but also with consideration of larger system issues such as memory architecture, compression, and video processing. Just as OpenGL is influenced by wider system concerns, OpenGL itself can provide a clarifying influence on system capabilities not conventionally thought of as graphics-related.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.6 [Computer Graphics]: Methodology and Techniques—Standards

Keywords: OpenGL, Graphics Hardware Architecture, InfiniteReality, O2

1 Introduction

The OpenGL Graphics System provides a well-specified, widely-accepted dataflow for 3D graphics and imaging. While programmers may think of OpenGL as simply a programming interface [7], we take the view that OpenGL defines an *architecture*.

We say a set of implementations manifest an architecture when three conditions are met:

1. The implementations must all have an identical interface and generate functionally equivalent outputs given the same inputs and initial state.

2. The determiner of functional equivalence is something other than a particular implementation.

3. The determiner of functional equivalence does not necessitate that all implementations be operationally identical. (There must be multiple ways to implement the architecture.)

1997 SIGGRAPH Eurographics Workshop

Implementations that are simply "compatible" do not necessarily manifest an architecture. Our definition allows for an implementation to belong to an architecture but have additional capabilities beyond those defined by the architecture.

By our definition, OpenGL is clearly an architecture. While the determiner of functional equivalence is not required to be a codified specification,[1] OpenGL's architecture is indeed defined by its specification [11].

Implementations of an architecture typically accrue significant advantages not available to *ad hoc* implementations or sets of implementations that are compatible yet do not manifest an architecture. Architectures gain an advantage from compatibility, but also tend to be more adaptable and foster innovative implementations through the freedom granted designers in how they realize the architecture. Architectures also tend to be easy to extend because an implementation's behavior is typically not specified for situations not defined by the architecture's functional equivalence.

The intent of this paper is to explore OpenGL's *adaptability* as an architecture. What we refer to as the adaptability of an architecture is not measured by units sold or market share. Instead, we contend that the adaptability of an architecture should be judged by the architecture's ability to codify well-understood functionality, its potential to be cleanly extended to support new capabilities, and its ability to influence positively issues outside the scope of the architecture itself.

Our approach is to consider two manifestations of the OpenGL architecture: the Onyx2 InfiniteReality graphics supercomputer and the O2 desktop workstation. Our examples were chosen because each is the result of quite different cost, performance, and capability goals, but both concretely demonstrate our primary contention that OpenGL is technically successful as an architecture because it is extensible to encompass new capabilities within the scope of interactive graphics *and* because OpenGL can positively influence system issues not directly graphics-related. Our approach is novel because, while we consider concrete implementations, we are fundamentally evaluating OpenGL as a graphics system architecture, not a particular hardware implementation.

Section 2 reviews the OpenGL architecture's scope, philosophy, functionality, and means of extensibility. Section 3 describes how OpenGL is instantiated by the Silicon Graphics Onyx2 InfiniteReality. Section 4 describes how OpenGL is instantiated by the Silicon Graphics O2 workstation. Section 5 contrasts the two implementations based on how they distinctly manifest the OpenGL architecture. Section 6 discusses how the OpenGL architecture influenced and even clarified several non-OpenGL design considerations in both example implementations. Section 7 argues that the OpenGL architecture is "good" because it provides us a framework for building innovative, evolvable, well-integrated graphics systems.

[1] The PC architecture lacks a codified specification but what constitutes a PC has evolved beyond the point that a PC can be described operationally by a single implementation as was originally the case.

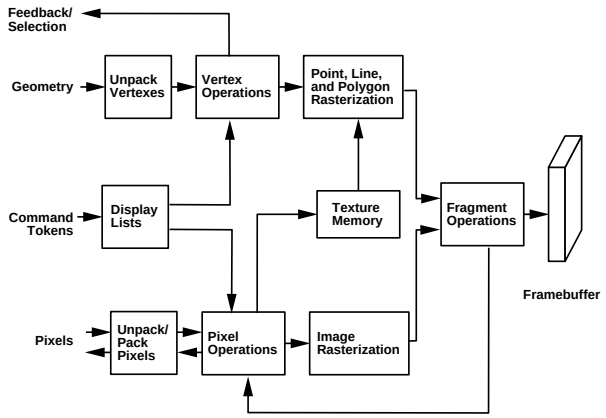

Figure 1: The dataflow within the OpenGL architecture's conceptual state machine.

2 OpenGL is a Visualization Architecture

The OpenGL architecture addresses the task of efficiently converting vertex- and pixel-based data representations into images. While the "GL" in OpenGL stands for Graphics Library, we consider OpenGL's functionality mandate to be larger than that of a traditional 3D graphics library. OpenGL manipulates vertex and pixel data with comparable ease. Moreover, texture mapping provides a "bridge" to effectively combine the rasterization of vertex- and pixel-based data representations.

We consider SGI's early IRIS GL implementation to exemplify the conventional feature set of a 3D graphics library. Over time IRIS GL added texture mapping and image processing operations to its repertoire. These additions served as the motivation for rethinking the purpose of a graphics library during the design of OpenGL. Because OpenGL is well-suited for manipulating both vertex and pixel data, supports texture mapping, and embodies an architecture, we refer to OpenGL as a *visualization architecture*.

2.1 State Machine Philosophy

OpenGL is specified as a state machine. OpenGL commands either set state variables, retrieve state variables, retrieve framebuffer contents, compile or call display lists, or introduce vertex or pixel data into the state machine. Vertex and pixel data introduced into the state machine are processed based on the current OpenGL state settings with the results sent to the framebuffer, texture objects, display lists, or selection/feedback buffer depending on OpenGL's current settings. Figure 1 shows the high-level dataflow within the OpenGL architecture's conceptual state machine.

Beyond OpenGL's state machine model, several philosophical choices help make OpenGL both extensible and adaptable to unexpected situations. In later discussion, we note how these choices are manifested in the two example implementations considered.

OpenGL's state variables are *orthogonal*. In general, the enabling or reconfiguring of OpenGL features does not interfere with other features. For example, lighting calculations can be enabled or disabled independently from the current depth buffering mode. This means programmers can combine features with predictable results. An often unforeseen advantage of feature orthogonality is that multiple independent features can often be combined in useful but unanticipated ways. Much of OpenGL's ease of extensibility is predicated on feature independence. Without orthogonality, multiple architectural extensions lead to confusing interdependencies or even create feature conflicts.

The OpenGL architecture is *client-server* in the abstract sense, not necessarily in a networked sense. Client-server means that the interface between an OpenGL application and an OpenGL implementation is strictly defined and all data passing between the application and implementation is explicit. The client-server separation defines the boundary between OpenGL implementation state and that of the application. This clear boundary makes possible network extensible OpenGL implementations [5] and allows OpenGL to be used as a direct hardware interface.

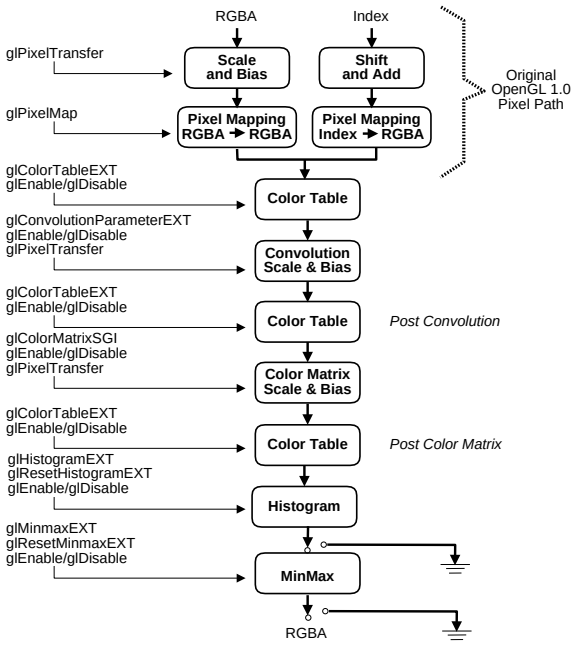

Figure 2: The extended OpenGL pixel path including the convolution, histogram, color matrix, and color table extensions.

The OpenGL architecture is *data format rich*. Immediate mode transfer of pixel and vertex data can be accomplished using OpenGL's wide variety of data sizes and formats. This allows applications to easily transfer their vertex and pixel data to OpenGL by traversing application-dictated data structures. Applications can supply pixel data using various strides, offsets, and component packings. Application performance typically benefits from avoiding data reformatting when transferring data to OpenGL. However, OpenGL implementations must be ready to accept OpenGL's multitude of possible data formats.

The OpenGL architecture is *configurable, but not programmable*. The OpenGL state machine can be thought of as a pipeline with a fixed topology (though various stages may be switched in or out). This mimics the layout of high-performance graphics subsystems where rendering steps are decomposed and instantiated by specialized hardware. The OpenGL architecture clearly encourages this style of implementation. This does create situations where features such as programmable shaders [8] or generalized image processing chains [12] are difficult to express as extensions to the OpenGL architecture.

2.2 Functional Decomposition

Sections 3 and 4 discuss how OpenGL (as specified in version 1.1) is instantiated by our example implementations. Therefore, this section briefly reviews OpenGL's functionality from an architectural standpoint. The operations are explained "bottom up" starting with the lowest level operations that update the framebuffer and moving to the highest level operations that accept commands.

2.2.1 Per-Fragment Processing and Rasterization

A fragment in OpenGL is the bundle of state required to update a specific pixel in the framebuffer. Fragments are generated during rasterization. The per-fragment operations are pixel ownership, scissoring, alpha testing, stencil testing, depth testing, blending, dithering, and logicop. The operations are performed in the order listed though what operations are enabled depends on OpenGL's per-fragment state variables.

Rasterization is the process of breaking a primitive up into fragments that are passed to the per-fragment processing stage. OpenGL supports five types of primitives: points, lines, polygons, pixel rectangles, and bitmaps. The first step in rasterization is determining if a framebuffer pixel is updated by the primitive. Depending on the primitive being rasterized, the current raster position, face culling, point size, line width, line stipple, polygon stipple, and antialiasing state affect which pixels are updated. The next rasterization step determines the fragment depth and color of affected pixels. The alpha color component is altered based on the antialiasing state of geometric primitives. The depth of geometric primitives can be altered depending on the polygon offset state. When enabled, texture mapping and fog modify the color of both geometric and pixel primitives.

2.2.2 Texture Mapping and Mangement

Texturing maps a portion of a specified image onto each primitive for which texturing is enabled. Texture coordinates determine what portion of the image is mapped to the primitive. OpenGL supports both 1D and 2D textures in a wide variety of formats. Texture parameters and the texture environment determine the method of filtering texels and how texels are combined with fragments generated during rasterization.

Texture objects provide the capability to switch between multiple texture images without the overhead of respecifying the texture image each time. Rectangular regions of textures can be incrementally updated using subtexture loads. When a texture image is specified, the constituent pixels are passed through the OpenGL pixel pipeline so the same operations discussed below that apply to drawing, copying, or reading pixel rectangles also transform texture images when they are specified.

2.2.3 Both Vertex and Pixel Processing

OpenGL transforms application-supplied vertex coordinates to window coordinates, clipping the primitives as necessary. Per-vertex lighting is performed if enabled. Texture coordinates are either explicitly supplied by the application or generated based on the vertex coordinates.

OpenGL defines a *pixel path* to process pixels. The pixel path can be configured to perform component scaling, biasing, and remapping via table lookups. Pixels are transformed by the pixel path when pixels are drawn to the framebuffer, read back from the framebuffer, copied within the framebuffer, or downloaded into texture memory. Each pixel transfer case shares the identical pixel processing machinery.

2.2.4 Other Capabilities

Display lists provide a way to cache repeated command sequences for potentially faster execution. Evaluators provide a means to efficiently specify Bézier curves and surfaces. Feedback and selection redirect the results of vertex processing back to the application instead of on to rasterization.

2.3 Extensibility

One key to an architecture's adaptability is its extensibility. OpenGL can be incrementally enhanced through its proven API extension mechanisms. OpenGL's rendering functionality can be extended by adding extensions to OpenGL's core rendering model. Extensions also can be made to OpenGL's window system dependent interface to address issues outside OpenGL's rendering model.

Various OpenGL vendors have already implemented dozens of extensions, and the OpenGL 1.1 update was the result of the OpenGL Architectural Review Board's efforts to fold successful, proven extensions back into the core OpenGL architecture. OpenGL 1.1 added vertex arrays, polygon offset, RGBA logic operations, texture objects, and further texture functionality enabled by texture objects.

The following extensions are important for later discussion.

2.3.1 Imaging Extensions

A key set of OpenGL extensions[2] are the imaging extensions [10]: color table, convolution, color matrix, histogram, and new per-fragment blending modes. Figure 2 shows the extended pixel path.

2.3.2 Hardware Accelerated Off-screen Rendering

Hardware accelerated offscreen rendering is critical for a multitude of techniques that must reliably readback or reuse rendering results. A window system dependent extension for pixel buffers (commonly called *pbuffers*) enables hardware accelerated offscreen rendering.

3 OpenGL as Instantiated by InfiniteReality

Onyx2 InfiniteReality implements the bulk of OpenGL's dataflow within the InfiniteReality graphics subsystem. InfiniteReality is designed to be a "real time" graphics machine meaning that sustained 30 hertz and higher frame rates are achievable even for demanding applications. InfiniteReality's intended application domains are visual simulation, film & video production, real-time image processing, volume rendering, and large-scale CAD.

InfiniteReality is a hardware-intensive design consisting of 13 distinct Application Specific Integrated Circuits (ASICs).[3] InfiniteReality is a multiple-board graphics subsystem with the same board-level architecture as the RealityEngine [1], InfiniteReality's predecessor. A single Transform Manager board connects to 1, 2, or 4 Raster Manager boards and a single Display Generator board. Figure 3 shows an ASIC-level block diagram of InfiniteReality. Figure 4 shows how OpenGL's conceptual state machine (originally shown in Figure 1) roughly maps to InfiniteReality's rendering ASICs. Starting at the host interface and working towards the framebuffer and display back-end, the following discussion shows how the OpenGL architecture is instantiated by InfiniteReality.

[2] Under consideration for inclusion in OpenGL 1.2.

[3] Other sources of information about InfiniteReality are likely to refer to the boards and ASICs that constitute InfiniteReality by "working names" that grew out of historical SGI jargon and tradition. In a few cases, the working names inadequately describe the ASIC or board's true function in the context of OpenGL. For example, the Geometry Engine ASIC handles *both* vertex and pixel data so we refer to it here as a Transform Engine to better suit our purpose of describing how InfiniteReality manifests the OpenGL architecture.

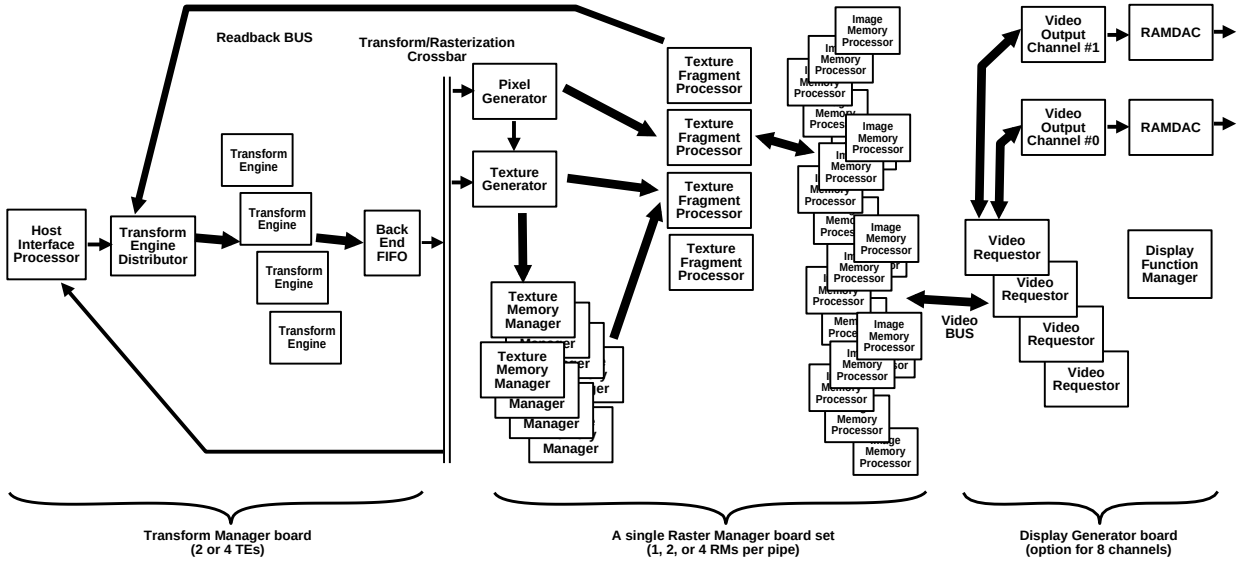

Figure 3: ASIC-level diagram showing the InfiniteReality graphics subsystem architecture.

3.1 Host Interface

The client-server structure of OpenGL makes it possible for essentially the entire OpenGL feature set to be implemented within the InfiniteReality graphics subsystem. The host-based OpenGL library is largely used to setup efficient data transfers to and from the graphics subsystem. For example, an immediate mode `glVertex3f` call returns in 7 instructions. This consists of jumping through a redirection table, writing the `Vertex3f` token followed by the three floating point coordinates to the graphics FIFO address, and returning.

OpenGL commands and data enter InfiniteReality via a high-bandwidth proprietary IO bus where they are received by the Host Interface Processor (HIP) that decodes and dispatches OpenGL command streams. Commands can be sent either by programmed IO or via Direct Memory Access (DMA).

The HIP's Input Control and Mapping (ICU) logic arbitrates the OpenGL command stream from one of three sources: the host-filled graphics FIFO, the host-activated input DMA stream, or a local DMA stream used for calling locally cached display lists. The ICU performs basic OpenGL command stream error checking and directs commands for subsequent processing. Pixel and vertex commands and some mode changes are simply passed along for further processing. To process OpenGL command streams with data rates over 300 MBs/second, the ICU must be very fast. More complex OpenGL commands involving display lists, more complicated state management, DMA setup, or non-rendering tasks can be redirected to a microcoded 32-bit RISC core. Most of the RISC core's microcode is written in C.

Display lists are cached in 15 of the 16 megabytes of external memory managed by the RISC core (one megabyte is used for state and microcode). The HIP's local DMA facility allows cached display lists to be passed through the ICU just as if the command sequence was generated by the host. Most immediate mode OpenGL calls result in IO writes to the hardware's graphics FIFO address. The graphics FIFO is mapped into the address space of direct rendering OpenGL applications [6]. OpenGL command streams can also be "pulled into" the HIP via input DMA. Large textures, pixel arrays, vertex arrays, and host-resident display lists can all be transferred this way. Because DMA transfers involve fixing host physical memory mappings, DMA is initiated with operating system support.

The HIP is also responsible for returning OpenGL data back to the host. The results of `glGet*`, feedback, selection, and `glReadPixels` are all returned via DMA. The HIP is responsible for any data reassembly required before returning the data to the host.

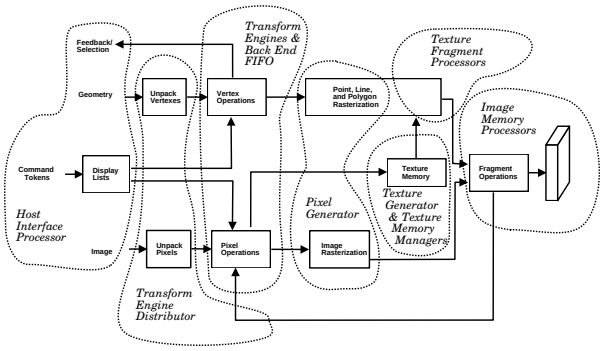

Figure 4: How the conceptual OpenGL state machine roughly maps to InfiniteReality's rendering ASICs.

3.2 Vertex and Pixel Transform Subsystem

The HIP sends the partially decoded OpenGL command stream to the Transform Engine Distributor (TED). The TED front end is responsible for converting OpenGL's data format rich command stream into a canonical format in preparation for handing the data to the Transform Engines (TEs) for processing. For example, double precision floating point or integer coordinates are forced to single precision floating point. Pixel data is also reformatted as necessary. Commands to change OpenGL state are mostly passed through unaltered. Given the high data bandwidths involved and the flexibility that OpenGL allows, the TED front end must be very fast.

The TED backend distributes bundles of work to 2 or 4 TEs that perform the actual vertex and pixel transformations required. Managing OpenGL's `glBegin`/`glEnd` and per-vertex state is done through a microcoded state machine. The TED also must ensure that OpenGL transformation state is synchronized among the multiple TEs to guarantee proper OpenGL command serialization se-

mantics despite multiple active TEs. The TED performs a mapping of OpenGL command tokens to TE microcode addresses so that the TE can immediately begin command execution. Work is typically assigned to the least busy TE.

The TE ASIC is a custom microcoded floating point processor. Each TE has a peak performance of 540 megaflops achieved using three SIMD floating point cores. The TEs use custom support logic to accelerate graphics-specific operations such as clipping. A carefully tuned memory system is essential to keep the floating point units continually busy. To minimize the amount of microcode required given the variety of geometry and pixel transformations potentially enabled, microcode modules are "stitched" together based on the current OpenGL geometry or pixel transformation state. For example, the lighting microcode module would only be added to the TE's geometry microcode sequence if lighting is currently enabled.

The TEs implement the pixel path functionality including the extended pixel path functionality described in Section 2.3.1. Special care is taken in the TED and TEs to manage pixel distribution when pixel convolution is enabled. Another pixel path challenge is memory management for the various lookup tables, convolution kernels, histogram bins, and other pixel path state that must be maintained within each TE. Both pixel rectangles and texture downloads flow through the TEs and so the identical microcode transforms both types of pixel data identically as required by OpenGL.

The complete Transform Manager subsystem can sustain geometry transformation rates of over 11 million polygons/second.

3.3 Transformation to Rasterization Crossbar

The transformed vertices and pixels from the TEs flow out in packets that must be reordered by the Back End FIFO (BEF). The BEF is a 4 megabyte FIFO intended to minimize stalling the TEs during framebuffer clears or the rasterization of very large polygons or pixel rectangles.

The BEF broadcasts the contents of its FIFO across the Transform/Rasterization Crossbar connecting the BEF to 1, 2, or 4 Raster Manager boards. Two main types of requests are sent over the crossbar: texture (or *load*) requests and rendering (or *draw*) requests. The crossbar also feeds back to the HIP to implement selection/feedback, state retrieval, and context switching.

The BEF actually maintains two distinct FIFOs: the draw FIFO for rendering and the load FIFO for texture download. The draw FIFO takes priority over the load FIFO, but the load FIFO drains whenever the draw path is stalled. The draw path can stall because it has gotten backed up with rasterization work or because it is waiting on a texture to download. Waiting for a texture to fully download provides an interlock that ensures textures are always properly loaded before use. The advantage of this scheme is that textures can be downloaded concurrently with rendering to increase overall throughput.

3.4 Primitive Rasterization

Geometric and image primitives, texture data, and mode changes are all broadcast over the Transform/Rasterization Crossbar to the Raster Manager boards. The crossbar can sustain a maximum bandwidth of 400 MBs/second. The Pixel Generator (PG) and Texel Generator (TG) ASICs on each Raster Manager listen for the data flowing from the BEF. Both the PG and TG rasterize image and geometry primitives sent over the crossbar. The PG almost completely rasterizes primitives. Depending upon the current OpenGL rasterization state, the highly pipelined PG scan converts geometric primitives, pixel zooms images, scissors, interpolates color and depth between vertices, calculates coverage alpha values for antialiasing, and applies the polygon stipple. The only rasterization steps not done in the PG are texture and fog application. The PG can sustain the rasterization of over 12 million polygons a second.

3.5 Texturing

InfiniteReality is balanced to render just as fast with its highest quality (linear mipmap linear) texturing enabled as when rendering with texturing disabled. This requires a very fast and sophisticated texture subsystem.

Using data received over the Transform/Rasterization Crossbar and rasterization results passed to it from the PG, the TG needs to initiate texel fetches for textured primitives in parallel with the rasterization work done by the PG. The TG needs to rasterize only textured primitives to the point that the TG can generate the necessary per-fragment texture coordinates interpolated across the primitive.

Texture coordinate information is broadcast to 8 Texture Memory (TM) ASICs. Each Raster Manager board is configured with either 16 or 64 megabytes of texture memory split evenly among the TMs. Texture accesses tend be highly redundant as nearby texels are often needed multiple times in the course of filtering the texels for a given textured primitive. The TMs act as specialized memory controllers that are optimized for texel access patterns.

InfiniteReality includes numerous texture extensions introduced by RealityEngine including sharpen texture, detail texture, 3D texture for volume rendering, and post-filtering texture lookup tables. InfiniteReality also includes new texture features such as clipmapping for rendering continuous terrain and various modes for better video texture mapping.

3.6 Fragment Processing

Texels from the TMs and texture coordinate information from the TG are combined in one of 4 Texture Fragment (TF) ASICs. The TFs also receive the actual fragments generated by the PG. The information from the TMs and TG are used to perform OpenGL's texture filtering modes such as linear mipmap linear filtering. A post-filtering stage can optionally scale, bias, and perform a table look up on the filtered texels. These extra steps are OpenGL extensions that are useful for image processing and volume rendering effects. Fully filtered texels are then combined with the fragments from the PG based on the current OpenGL texture environment. If enabled, fog is applied. The last operation done by the TF is the per-fragment alpha test.

Each TF is connected to 5 Image Memory Processor (IMP) ASICs. Each IMP ASIC contains 4 instances of the IMP core. Each IMP core manages 1 megabyte of external memory containing the framebuffer. The IMPs manage 80 megabytes total per Raster Manager. Each IMP core manages a scattered distribution of pixels and receives fragments from its TF. The IMP core performs all OpenGL per-fragment operations except alpha testing which is done in the TF and scissoring which is done in the PG.

The IMPs maintain multiple depth and color samples per pixel to realize order-independent antialiasing. The IMPs also perform OpenGL's accumulation buffer [4] operations.

A single Raster Manager board can sustain textured pixel fill rates of 200 megapixels per second. The combined textured fill rate with four Raster Managers is therefore 800 megapixels per second.

3.7 Display Generator Subsystem

The Display Generator board is responsible for generating analog video streams based on the current contents of the framebuffer maintained by the IMPs in the Raster Manager. InfiniteReality supports 2 or 8 analog video output channels . Each Video Output Channel (VOC) ASIC generates video requests sent over a serial interface to the IMPs. The IMPs respond with the requested framebuffer color

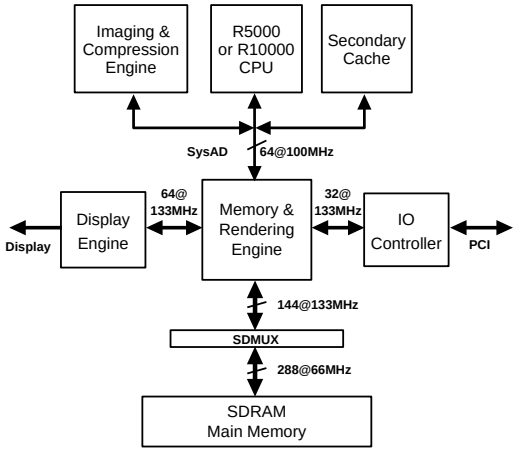

Figure 5: Block diagram showing the O2 system-level architecture.

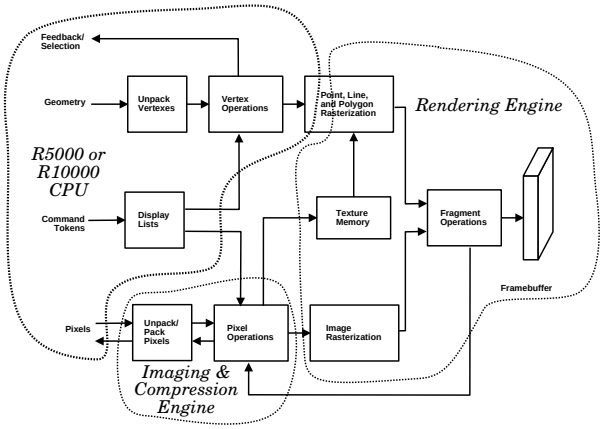

Figure 6: How the conceptual OpenGL state machine roughly maps to O2's various ASICs.

information on the Video Bus. The core OpenGL architecture does not directly concern itself with video issues so further details about the Display Generator are put off until Section 6.

3.8 Reading and Copying Pixel Data

The OpenGL pixel path draws pixels and downloads textures, but must also transform pixels that are copied (`glCopyPixels`) or read back to the host (`glReadPixels`). When a framebuffer read or copy is initiated, the IMPs send framebuffer pixels to the TFs that transfer on the data over the Readback Bus to the TED. The TED feeds the framebuffer pixel data through the TEs much as if it were pixel data originated from the host.

The fetched pixel data is transformed by the TEs and then is either rendered back into the framebuffer in the case of `glCopyPixels` (just like the `glDrawPixels` case) or is transferred back to the host in the case of `glReadPixels`. When reading pixels, the BEF directs the pixels across the Transform/Rasterization Crossbar where the HIP reassembles the pixel data before DMAing the pixels back to the host.

OpenGL's requirement that texture memory must be retrievable necessitates a pathway for texels to be returned to the host. The TMs can pass texture contents to the TF where data passes over the Readback Bus and eventually back to the host. Unlike `glReadPixels`, retrieved texture contents are not transformed by the pixel path.

3.9 Offscreen Rendering

Excess framebuffer memory can be allocated to pbuffers for offscreen rendering as described in Section 2.3.2. The amount of renderable offscreen memory is limited and depends on the resolution of the framebuffer. While pbuffers allow full speed offscreen rendering, because pbuffers are carved from "excess" framebuffer space, pbuffers on InfiniteReality can suffer from thrashing or volatility when pbuffer resources come into contention with other pbuffers or the "deep" ancillary buffers belonging to windows. Window framebuffer state always takes priority over pbuffers.

3.10 Context Switching

OpenGL permits multiple concurrent contexts. InfiniteReality context switches as necessary to support multiple processes each using OpenGL. Context switches can be synchronous such as when a process changes to a different rendering context with `glXMakeCurrent` or completely asynchronous due to the operating system's scheduling of multiple concurrently rendering processes [13]. Both cases are handled basically the same way from the hardware's point of view.

A special context switch token is generated by the operating system when a context switch is required. The token "pushes" HIP, TED, TE, and BEF state out over the Transform/Rasterization Crossbar where it is DMAed back to the host. Commands preceding the context switch token simply execute to completion. No OpenGL context state is read back from the Raster Manager. Instead, the BEF "snoops" all Raster Manager ASIC register writes and thereby shadows the current Raster Manager state. After the context switch token pushes out the current state, the operating system initiates input DMA to load the next context's state. The output DMA to save the previous context executes simultaneously with the input DMA to load the new context. The old context streams out while the new context streams in. Raster Manager register values that had been shadowed in the BEF are replayed to restore the complete Raster Manager state. Special care must be taken to context switch texture objects in the TMs' texture memory and display lists maintained in the HIP's external memory. Since there is often enough memory to keep all the current texture objects and display lists resident at once, swapping of texture objects and display lists occurs only in overextended situations.

4 OpenGL as Instantiated by O2

O2 delivers integrated 3D graphics, image processing, audio, compression and video processing capabilities in a cost-effective, small form factor. Figure 5 shows a chip-level block diagram of the O2 system architecture. Notice that the Memory & Rendering Engine (MRE) is implemented by a single ASIC. Graphics is inseparable from the system.

Traditional PC and workstation designs treat graphics as a distinct resource with its own dedicated framebuffer and rendering hardware. O2 has no dedicated framebuffer memory; any memory in the system can be scanned out by the O2's Display Engine ASIC. Moreover, any memory in the system can be used as texture memory or as a destination for rendering. O2 also contains a specialized Imaging & Compression Engine (ICE) ASIC with access to the O2's main memory through DMA transfers. The shared system bus has a peak bandwidth of 2.1 GBs/second in order to satisfy the combined demands of the Display Engine, the Imaging & Compression Engine, the MIPS CPU, and the IO Engine for memory bandwidth.

The MIPS CPU, Imaging & Compression Engine, and Mem-

ory & Rendering Engine are all used in combination to implement OpenGL rendering. The simplified decomposition is that: the CPU does vertex processing, rasterization setup, and state management; the ICE performs most pixel path transformations; and the MRE performs rasterization, texture mapping, and pixel update. Figure 6 shows how OpenGL's conceptual state machine (originally shown in Figure 1) roughly maps to O2's hardware.

4.1 Host-based Vertex Processing

The O2 CPU is either an R5000 or R10000 MIPS processor. The CPU performs all vertex processing including lighting, transformation, face culling, and clipping. The CPU also calculates primitive plane equation parameters for the MRE's Rendering Engine. The CPU sends commands to the MRE using direct register writes. The CPU also is responsible for OpenGL state management. This involves selecting the appropriate code pathway for OpenGL's current mode settings. Unlike InfiniteReality where OpenGL API calls largely serve to transfer OpenGL commands directly to the hardware, O2's OpenGL library has a substantial amount of CPU-based code. While most of the O2 OpenGL library is written in C, the rendering paths for geometric primitives with common rendering modes enabled are completely written in highly tuned MIPS assembly code.

The R5000 is a low-cost 64-bit dual-issue superscalar microprocessor well-optimized for single-precision floating point operations. The R10000 is a higher performance 64-bit 5-way superscalar microprocessor featuring out-of-order instruction execution. Either processor is well-suited for OpenGL's vertex transformations that are largely single precision floating point short vector operations.

4.2 Imaging Engine for Pixel Operations

The ICE is a specialized processor for imaging, video processing, and compression tasks. The ICE has two programmable execution units relevant for implementing OpenGL's pixel path. A pipelined RISC processor based on a subset of the MIPS R3000 microprocessor serves as a scalar unit. A vector unit acts as a coprocessor to the scalar unit. The vector unit performs the same computation in parallel on 8 or 16 element vectors in a single cycle where each element is either a 16-bit or 8-bit data item respectively. The two execution units are tightly coupled and the vector unit executes instructions in parallel with the scalar unit. The ICE supplies itself with data using a DMA unit that can send and receive rectangles of image data to and from main memory. The processor has only a very limited amount of internal memory. It has 6 kilobytes of internal data memory and 4 kilobytes of internal instruction memory.

Device dependent libraries such as OpenGL set up image processing tasks for the ICE. The operating system schedules access to the ICE. SGI-supplied device-dependent libraries seeking to execute algorithms using the ICE must first setup an ICE execution template. The template contains ICE microcode, physical memory mappings for the ICE DMA unit, and initial data parameters. Once an ICE execution template is set up, the library requests the operating system to instantiate the template. This means that the operating system DMAs the requested ICE microcode and initial data into the ICE internal memories, sets up the DMA mappings for the ICE, sets the ICE program counter, and starts the ICE executing (normally the ICE idles if it has nothing assigned to it). A typical ICE task is to convolve or otherwise transform a block of pixels. The actual pixel data to transform is read into the ICE via DMA, quickly transformed, then written back out again with DMA. The microcode assumes that its execution template has been properly initialized. This means that its data memory is appropriately set up and that the ICE DMA mapping is likewise appropriately set up so that pixel data will be DMAed from and to the right places. When the task completes, the ICE notifies the CPU with an interrupt. The operating system then notifies the process initiating the ICE operation of the operation's completion. The ICE cannot be preemptively context switched. Use of the ICE relies on ICE requests completing in some small amount of time.

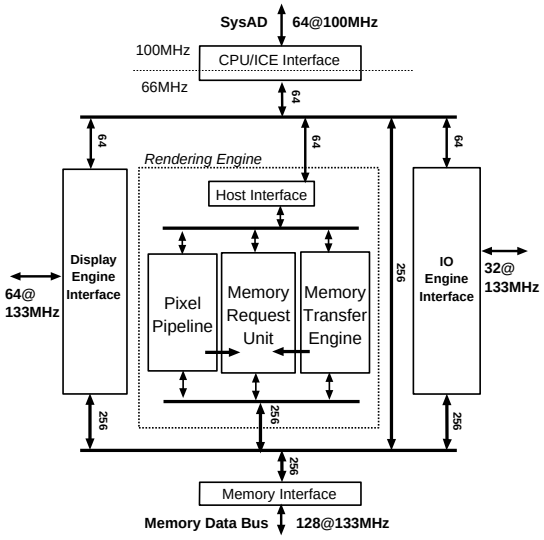

Figure 7: Block diagram of O2's Memory & Rendering Engine.

The O2 OpenGL library uses the ICE whenever possible to speed pixel path operations. All the OpenGL imaging extensions named in Section 2.3.1 are acceleratable through ICE microcode. Numerous pieces of ICE microcode are available to implement various configurations of the OpenGL pixel path. The CPU must carefully decide if a particular pixel path configuration can be successfully accelerated by the ICE. This determination depends not simply on whether a given ICE microcode module exists for the current pixel path configuration, but also whether the pixel path has a small enough set of associated pixel path parameters that the current pixel path configuration can be implemented within the limited data space available to the ICE. This means some extremely complex pixel path configurations might not be executable using the ICE. In practice however, the ICE accelerates most typical pixel path configurations. The ICE also accelerates OpenGL's accumulation buffer functionality.

Even when the ICE is suitable for accelerated processing, the OpenGL library makes a dynamic check before attempting to use the ICE to make sure it is not already in use. If the ICE is found in use or if the pixel path configuration is too complex or otherwise not supported by the ICE, OpenGL falls back to a general CPU-based pixel path implementation.

4.3 Integrated Rendering and Texturing

The MRE ASIC serves as both O2's memory controller *and* OpenGL rendering processor. Because the rendering unit is so tightly coupled with the memory subsystem, the complete memory read/write requirements for OpenGL texture fetching, ancillary buffer operations, clip ID based window ownership testing, and color buffer update are all serviced via main memory accesses; O2 has no graphics-specific memories. The rendering unit implements almost all of OpenGL's texture, per-fragment, and rasterization functionality.

The MRE plays the central role of arbitrating memory accesses by the CPU, the ICE, the Display Engine, the IO Engine, and the MRE's internal Rendering Engine. Figure 7 shows the internal structure of the MRE. The MRE's Rendering Engine contains three

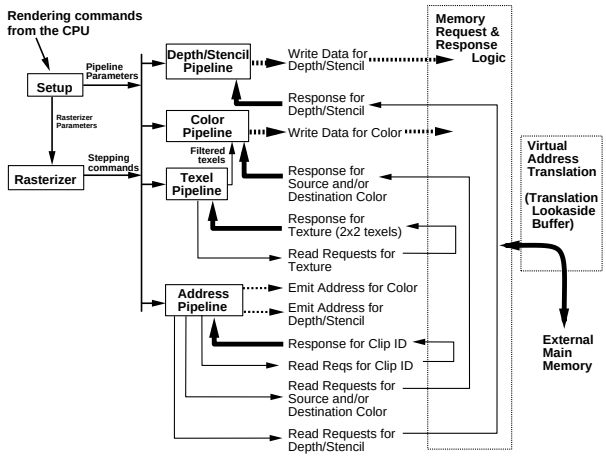

Figure 8: Block diagram of the rasterization pipeline and memory request/response logic for the MRE's rendering engine.

high-level functional blocks: the Pixel Pipeline that performs all of OpenGL's rasterization, texturing, and per-fragment operations; the Memory Request Unit that queues color, depth/stencil, clip ID, and texture memory fetches; and the Memory Transfer Engine that performs fast clears and copies. The Rendering Engine's performance relies heavily on hiding memory latency by prefetching framebuffer and texel data in advance of the various pipeline stages that require the data.

Figure 8 shows the dataflow within the Rendering Engine's Pixel Engine block. Commands to rasterize OpenGL primitives arrive from the host interface. Stepping commands from the Rasterizer generate fragment information fed into four parallel pipelines: the Color Pipeline, the Texture Pipeline, the Stencil/Depth Pipeline, and the Address Pipeline. The Color Pipeline calculates the color of fragments. The Texture Pipeline fetches texels from main memory and filters them into a single texel that is passed on to the Color Pipeline if texturing is enabled. The Stencil/Depth Pipeline discards fragments based on depth and/or stencil testing if enabled. The Address Pipeline clips fragments as necessary and determines the address for writing the final fragment color into the framebuffer.

The various pipeline stages have the option to "fail" the fragment at various points. For example, if a fragment fails the depth or stencil tests, the Address and Color pipelines will not update the actual pixel in memory. Stippling, alpha testing, window clipping, and scissoring can also fail a fragment.

For reasonable performance, the Rendering Engine is heavily pipelined so that multiple fragments are processed in different pipeline stages at the same time to increase throughput. The Pixel Engine's most critical task in pipelining OpenGL's rasterization, texturing, and per-fragment processing sequences is hiding the memory latency introduced by operations requiring main memory accesses. These operations are texturing, stencil/depth testing and update, window clipping, retrieving the destination color for blending, and final pixel update.

The straightforward way to hide memory access latency in a pipeline is to add prefetch delay stages, but a more effective approach is to reorder the pipeline stages to move work not dependent on a particular memory access into stages that would otherwise serve only as delay stages. By carefully decomposing the sequence of operations defined by OpenGL, O2's Pixel Engine fills idle pipeline stages with work that is logically "after" an OpenGL operation requiring a completed memory read when the work does not depend on the read result.

For example, the OpenGL pipeline dictates that fog must be applied after texture. Texture application is done in the Color Pipeline but cannot proceed until the Texture Pipeline generates the fragment's filtered texel. Producing a filtered texel involves fetching texels from main memory and introduces a delay due to the latency of reading data from main memory. Applying fog requires no main memory fetches but does involve generating the fog blend factor. During the memory access delay required by the Texture Pipeline to generate a filtered texel, the Color Pipeline calculates the fog blend factor. This means that the parameters to apply fog are all available immediately after the texture application pipeline stage completes. There are several other places in OpenGL's ordering of rasterization, texturing, and per-fragment operations where computations performed logically "after" an OpenGL operation requiring a memory read can be partially computed during pipeline delays needed to wait for memory reads to complete. O2's Pixel Engine takes advantage of these opportunities to control overall fragment latency even while using pipelining to increase throughput.

O2 can render at over 60 megapixels per second and can render using linear mipmap linear texturing at over 30 megapixels per second. A 180 Mhz R5000 can render over 450,000 smooth shaded, depth tested, non-textured 50-pixel triangles per second.

4.4 Flexible Framebuffer Management

As mentioned earlier, O2 does not have a dedicated framebuffer; the Rendering Engine is fully capable of rendering into *any* area of the O2's main memory. In addition, any area of memory can be used to store texture, depth/stencil, overlay, and window clipping information. All Rendering Engine memory accesses use a special Rendering Translation Lookaside Buffer (RTLB) in the Rendering Engine's Memory Request Unit. RTLB entries must point to actual physical memory; there is no allowance for CPU-like page faults during rendering. The RTLB is for rendering only and is completely distinct from the TLB used by the CPU; RTLB entries have no relationship to user process address spaces.

The extra cost of RTLB lookups when reading or writing memory during rendering motivated the decision to locate the Rendering Engine within what would otherwise simply be the memory controller in a more traditional system architecture. Marginally higher memory latencies to framebuffer and texture memory due to using the RTLB justify the Rendering Engine's clever pipelining of OpenGL rendering to hide as much of the memory latency as possible.

Each RTLB entry points to a 64 kilobyte tile of physically contiguous memory. Tiles may be scattered throughout physical memory. A standard color buffer tile can represent a 128x128 rectangle of 32-bit RGBA pixels. Other types of tiles contain overlay, stencil/depth, texture, or window clipping state.

In normal system operation, the operating system allocates a set of tiles for use as the displayable framebuffer and configures the Display Engine to continuously convert the framebuffer region into an analog video stream. The Display Engine fetches framebuffer color data directly from main memory. Since video scan-out is a continuous real-time demand, the MRE assigns the Display Engine the highest memory access priority. The bandwidth required for video scan-out uses up about one half of the available 2.1 GBs/second of total system memory bandwidth.

Tiling helps reduce the amount of system memory that must be dedicated to graphics. For example, O2 supports 32-bit RGBA double buffered windows, but the tiles to maintain the second 32-bit color buffer are only allocated where 32-bit RGBA double buffered windows are renderable. Likewise, the framebuffer tiles for depth and stencil buffers are only allocated where windows with depth and stencil buffers are renderable. System memory not used for depth, stencil, or 32-bit back buffers can be used by the system as general purpose memory. Figure 9 shows an example of how O2's tiling scheme for allocating framebuffer memory can use less total mem-

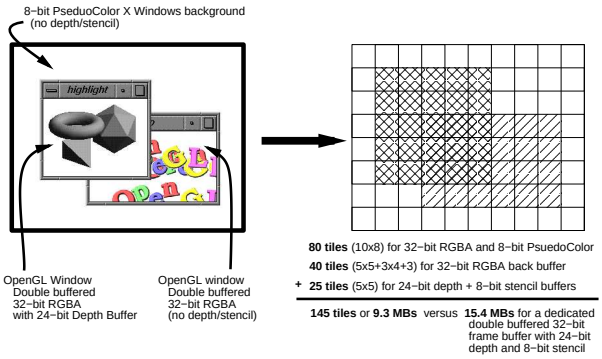

Figure 9: Example demonstrating how O2's framebuffer tiling scheme can reduce the memory requirements for graphics compared to a dedicated framebuffer scheme.

ory than a dedicated framebuffer capable of comparable resolution.

O2's ability to render into arbitrary system memory means that the pbuffer capability described in Section 2.3.2 is straightforward to support with O2. When an OpenGL application switches to a pbuffer, the pbuffer's physical tile addresses are loaded into the RTLB.

The X server and operating system coordinate with the OpenGL library to setup the Rendering Engine, ICE, and Display Engine appropriately. User applications have no means to manipulate directly O2's rendering, imaging, or display hardware. OpenGL, X window system requests, and SGI-supplied device-dependent libraries for digital media are the only mechanisms available for controlling O2's graphics and display hardware.

5 Contrasting the Implementations

InfiniteReality and O2 are quite distinct implementations of the same architecture; contrasting the two implementations of OpenGL offers a number of insights into the adaptability of OpenGL as an architecture.

5.1 Hardware Specialization and Replication

Comparing Figures 4 and 6 shows the degree to which InfiniteReality and O2 distribute OpenGL's functionality across hardware.

InfiniteReality clearly dedicates considerable hardware resources to the implementation of OpenGL's functionality. InfiniteReality's approach to implementing OpenGL almost gets to the point of devoting a specialized ASIC to each functional block in the OpenGL state machine. While O2 does utilize specialized hardware to implement OpenGL, O2's approach is considerably less hardware intensive. O2 does not multiply instance any of its ASIC-level components, but InfiniteReality relies heavily on replicating ASICs to boost parallelism, particularly in the texturing, rasterization, and per-fragment subsystems. This difference in approach reflects very different cost and performance goals. Replicating ASICs to the extent done in InfiniteReality is also probably only viable for systems driven more by performance than cost.

What InfiniteReality demonstrates is that the OpenGL architecture can scale its performance considerably through hardware specialization and replication. We observe that there are certain areas in InfiniteReality's design where ASIC replication is not exploited. The HIP, TED, TG, PG, and Transform/Rasterization Crossbar are all areas of the design where OpenGL's dataflow is effectively serialized. These serialized points in InfiniteReality's design are addressed by heavily pipelining the ASICs and by building a very high-bandwidth, but expensive interconnect in the case of the Transform/Rasterization Crossbar. These serializations are likely to result in future scalability issues when designing even higher performance, hardware-intensive OpenGL implementations.

5.2 Delivering the Necessary Memory Bandwidth

Both O2 and InfiniteReality demonstrate that OpenGL implementations must find ways to sustain the memory bandwidths necessary for fast graphics. Again, InfiniteReality's approach is driven by performance more than cost. InfiniteReality uses dedicated memories extensively to deliver the memory bandwidth required by InfiniteReality's performance goals. For example, the TMs and IMPs can be thought of as very specialized memory controllers optimized for the particular access patterns of texture memory and per-fragment operations respectively. The chief disadvantage of adding specialized, dedicated memories is that the total memory in the system is not generally available. For example, InfiniteReality precludes excess texture memory bandwidth or capacity in the TMs from being made available to the IMPs as framebuffer memory bandwidth or capacity. The HIP's external RAM, per-TE RAMs, and the BEF external FIFO RAM are other examples of dedicated memories in InfiniteReality that cannot be made generally available. Moreover, all memory in the InfiniteReality graphics subsystem is totally unavailable for use by the system except via graphics operations.

O2 has *no* dedicated memory. Not only is texture and framebuffer memory interchangeable, but unused texture or framebuffer memory can be used by the CPU as general system memory. Indeed, there is not any such thing as "texture memory" or "framebuffer memory" per se in O2. This flexibility comes at a cost; O2 has nowhere near the memory bandwidth available in InfiniteReality. It also means that the MRE's Rendering Engine has to be particularly clever about hiding memory latencies that result from the RTLB's ability to access graphics data anywhere in main memory. There are definitely cost and expandability advantages to O2's approach. Not dedicating graphics memory lets the O2 workstation generally run with less total main memory. This reduces overall system costs because memory costs are an important factor in total system cost. Also, if graphics intensive use of an O2 demands more memory and is forcing the system's general memory to page to disk, the user has the option to expand the O2's memory. In addition to the RTLB, the other key feature that makes O2's unified memory approach viable is having 2.1 GBs/second of system bus bandwidth to sustain video scan-out *and* the other system memory bandwidth demands.

Another approach to improving the memory bandwidth for OpenGL is to replicate texture memory. Replicating texture memory is attractive because fast texturing demands high texture memory bandwidths. InfiniteReality does not replicate texture memory within a Raster Manager board, but texture memory is replicated among multiple Raster Managers. To avoid replication within a Raster Manager, the 8 TMs must be fully connected with the 4 TFs so that any TF can fetch texels from any TM. The cost of this interconnect had to be weighed against the cost of replicating texture memory within the Raster Manager while still being able to achieve InfiniteReality's texture fill rate design goal.

To reduce costs, O2 does not replicate texture memory at all.

This draws out two points about the OpenGL architecture. First, fast implementations of the OpenGL architecture demand large amounts of memory bandwidth. Specialized, dedicated memories can meet this demand. In some cases, memory replication may even be needed. But second, O2 demonstrates that OpenGL can be implemented in such a way that memory can be considered a unified resource. The OpenGL architecture does not force on the implementor a particular approach towards managing graphics memory.

5.3 Pipelining for Throughput

An implementation technique extensively used by both InfiniteReality and O2 is pipelining. Every chip in both InfiniteReality and O2 benefit to some extent from pipelining. The very explicit description of the OpenGL dataflow and its static topology make it straightforward to apply hardware pipelining techniques when implementing OpenGL. If the OpenGL architecture allowed its operations to be reordered, the architecture would likely be less amenable to hardware pipelining.

5.4 OpenGL as a Direct Hardware Interface

InfiniteReality demonstrates that OpenGL can serve as an actual "hardware interface" instead of being simply a Hardware Abstraction Layer (HAL). InfiniteReality's OpenGL library largely serves to transfer OpenGL commands and their parameters to the InfiniteReality graphics subsystem. This allows for extremely high-performance OpenGL implementations because the entire burden of executing OpenGL commands can be off-loaded onto specialized hardware.

The ability to use OpenGL as *the* hardware interface is possible because of OpenGL's client-server model, its immediate mode interface, and its lack of features that are not readily amenable to hardware acceleration.

O2 implements OpenGL as a HAL, not as a true hardware interface. O2 performs a substantial amount of OpenGL's functionality on its CPU. Treating OpenGL as a true hardware interface is involved and expensive because it requires the hardware to implement the entire OpenGL state machine because OpenGL implementations must be complete. Being able to implement OpenGL as either a HAL or as the true hardware interface makes OpenGL adaptable to a wide variety of hardware/software divisions when implementing OpenGL up to and including the hardware almost fully manifesting OpenGL's client-server interface as undertaken by InfiniteReality. In the other extreme, OpenGL is also implementable entirely in software [9].

5.5 Distinct or Reused Data Paths

Substantial cost savings can be realized by reusing hardware data paths; likewise, implementing distinct data paths may offer worthwhile performance gains due to hardware specialization. OpenGL has a number of abstract data paths, and OpenGL implementors can decide whether combinations of OpenGL's various data paths should be mapped to a single hardware data path or if the data paths are better implemented as distinct specialized hardware data paths.

Notice in Figure 1 how the geometry and pixel data paths are largely distinct except that they share the same set of per-fragment operations. O2 uses the CPU to transform geometric primitives, but uses the ICE to transform pixel data. Using distinct hardware makes sense because O2's CPU is good at the floating point calculations required for transforming geometry and the ICE's integer vector processing capabilities are well suited to implementing OpenGL's extended pixel path. The O2's distinct hardware data paths for geometry and pixel transformation are in contrast to InfiniteReality's approach of using the same TEs for both geometry and pixel transformation.

6 Wider System Influences of OpenGL

The available technology and graphics price-performance goals cited in the previous section certainly drive the focus of an OpenGL implementation, but the OpenGL architecture can also contribute to important design decisions based on wider system considerations. We believe this to be a very important contribution of the OpenGL architecture. In our view, OpenGL provides an abstract model for graphics hardware design, but *also* supplies an architectural framework that can be adapted to serve system-wide design goals. Consider the various ways that the InfiniteReality and O2 OpenGL implementations influence and cleanly co-exist with other system design considerations.

6.1 Memory Organization

Sections 4.4 and 5.2 have already discussed O2's lack of dedicated framebuffer and texture memory. The ability of the MRE's Rendering Engine to render directly into system memory is a radical departure from traditional workstation and PC architectures.

The Stellar GS1000 [2] and GS2000 had a similar capability to render into system virtual memory with a specialized Rendering Processor, but the O2 approach is substantially different and more sophisticated. The Stellar implementation did not have a Display Engine like what O2 has so an image rendered into main memory still had to be block copied into a dedicated framebuffer to be displayed. The Stellar machine lacked the technology to incorporate a bus sufficiently fast to scan-out video from main system memory. Also, the Stellar approach uses the same TLB as the main processor instead of a special RTLB as used by O2's Rendering Engine.

The VIEWS [3] design has a distinct TLB for graphics and directly scans out its tiled framebuffer as video, but the framebuffer memory is dedicated graphics memory unlike O2. Also, VIEWS manages only the main color buffer in a tiled fashion. This is in contrast to O2 that manages the main color buffer, overlays, textures, depth/stencil buffers, and pixel buffers all via tiles of unified system memory. The VIEWS design tries to keep each window on unique tiles both to retain the window's contents to avoid the expense of window damage repair and to avoid the expense of arbitrary window clipping. O2 does not attempt to retain window contents and has multiple clip rectangles and clip ID testing for clipping to arbitrary windows. Retaining complete windows was considered for O2, but dropped because of the unbounded memory requirements, the relative speed of window repair on a fast machine such as O2, the ease of arbitrary window clipping with inexpensive hardware, and the increased software complexity.

While the ability to render into main memory is not completely novel nor is it something specifically enabled by OpenGL, the ability to easily adapt the OpenGL architecture to this radically different framebuffer arrangement is a testament to the adaptability of the OpenGL architecture.

6.2 Scalable Graphics Multiprocessing

Onyx2's system architecture is that of a Scalable Shared-memory MultiProcessor (S2MP). This means R10000 processor nodes are connected to each other by a scalable network, not via a single fixed-bandwidth shared bus. Combining multiple R10000 CPUs and multiple InfiniteReality graphics subsystems within a single system would easily consume all of the available bandwidth of a system designed around a single shared bus. However, Onyx2's scalable system architecture supports a "RealityMonster" configuration where 16 CPUs can drive 8 InfiniteReality pipes in a single system. In theory, the architecture could scale even further.

OpenGL's architecture presents no barriers to multiple independent processors simultaneously making OpenGL calls to distinct OpenGL graphics subsystems. By combining rendering results from different InfiniteReality pipes, multiprocessor multi-pipe OpenGL applications can achieve rendering rates that surpass the limits of a single InfiniteReality subsystem. Effectively combining rendering results is straightforward for tasks where the rendering work is easily divided in screen space, but such easy divisions are not always the case. While the complexity depends on the nature of

the rendering task, balancing the work done in distinct pipes and effectively compositing the results for display generally requires careful programming.

6.3 Video Display Capabilities

InfiniteReality has a very sophisticated video display subsystem. It allows for multiple video channels to each show a potentially different subrectangle of the framebuffer. Each video channel also can be separately resized on a per-frame basis. This can be used to dynamically adjust the fill rate requirements to maintain real-time frame rates. Because of the low latency required to update the video resizing hardware, dynamic video resizing is made available through a window system dependent OpenGL extension.

Other video related issues such as the synchronization of buffer swaps are likewise handled through window system dependent OpenGL extensions. Window system dependent OpenGL extensions are providing effective ways to expose real-time video display capabilities. Another video capability clarified by OpenGL is stereo. Basic left/right buffer rendering support for stereoscopic displays is built into the core of OpenGL. Stereo rendering and display is built into both InfiniteReality and O2.

6.4 Video Input and Video Texturing

O2 supports a special data type called a Digital Media buffer (commonly called a *DMbuffer*) that permits sharing and exchanging time sensitive visual data between compression devices and algorithms, video input/output, graphics rendering and texturing, and the CPU.

O2's unique digital media architecture allows a DMbuffer to be associated with a pbuffer. Among the possible applications, a pbuffer associated with a DMbuffer can be used to texture with video images. Through the combination of DMbuffers and OpenGL, O2 can accomplish low-overhead video texture mapping from O2's standard digital video camera. This is an example of how the 3D rendering and imaging functionality available through OpenGL can be combined with digital media capabilities such as live video.

6.5 Compression

The O2 Imaging & Compression Engine that is used to implement the OpenGL imaging pipeline is also designed to quickly perform common compression algorithms and a variety of color space conversions. With the inclusion of a bitstream entropy decoder, O2's ICE also enables JPEG and MPEG compression and decompression since the vector processor and scalar core can be reprogrammed to calculate the discrete cosine transformation required for JPEG and MPEG compression.

This is an excellent example of how computational elements for implementing OpenGL functionality can be reused for purposes not specifically pertaining to OpenGL.

7 Conclusions

OpenGL is a visualization architecture. By contrasting how the OpenGL architecture is manifested in the Onyx2 InfiniteReality and O2 computers, we observe that OpenGL is adaptable to very different cost, performance, and capability goals. Moreover, we can see in areas such as memory architecture, video processing, and compression how OpenGL can play an important role in clarifying the way in which these capabilities fit into the system as a whole.

We believe that the adaptability of the OpenGL architecture represents an important development in graphics hardware architecture because different OpenGL implementations can benefit from compatibility but still have substantial opportunities to adapt to specific system-wide requirements.

References

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

[2] Brian Apgard, Bret Bersack, Abraham Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000," *SIGGRAPH 88 Proceedings*, August 1988.

[3] Anthony Barkans, "Virtual Memory System Organization for Bit-Mapped Graphics Displays," *Advances in Computer Graphics Hardware IV*, Springer-Verlag, 1989.

[4] Paul Haeberli, Kurt Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *SIGGRAPH 90 Proceedings*, August 1990.

[5] Phil Karlton, Paula Womack, *OpenGL Graphics with the X Window System* (the GLX specification), Version 1.2, 1996.

[6] Mark Kilgard, David Blythe, Deanna Hohn, "System Support for OpenGL Direct Rendering," *Graphics Interface 95*, 1995.

[7] Renate Kempf, Chris Frazier, *OpenGL Reference Manual*, 2nd edition, Addison-Wesley, 1992.

[8] Anselmo Lastra, Steven Molnar, Marc Olano, Yulan Wang, "Real-Time Programmable Shading," *ACM 1995 Symposium on Interactive 3D Graphics*, April 1995.

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

[10] Randi Rost, "Using OpenGL for Imaging," *SPIE Medical Imaging '96 Image Display Conference*, February 1996.

[11] Mark Segal, Kurt Akeley, *The OpenGL™ Graphics System: A Specification*, Ver. 1.1, Silicon Graphics, April 30, 1993.

[12] Silicon Graphics, *ImageVision Library Programmer's Guide*, Document number 007-1387-040, 1996.

[13] Doug Voorhies, David Kirk, Olin Lathrop, "Virtual Graphics," *SIGGRAPH 88 Proceedings*, August 1988.