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.)

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.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee

1997 SIGGRAPH/Eurographics Workshop
Copyright 1997 ACM 0-89791-961-0/97/8..$3.50

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

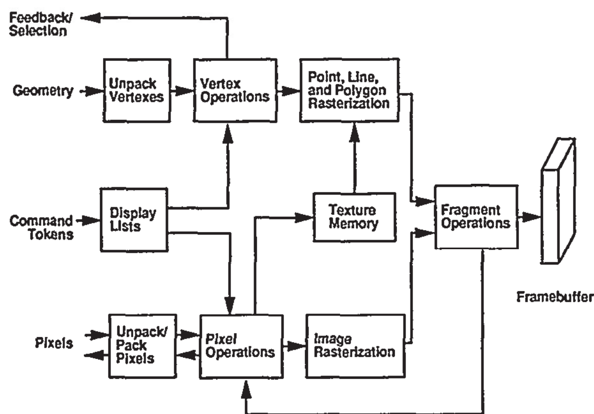

Figure 1: The dataflow within the OpenGL architecture's conceptual state machine.

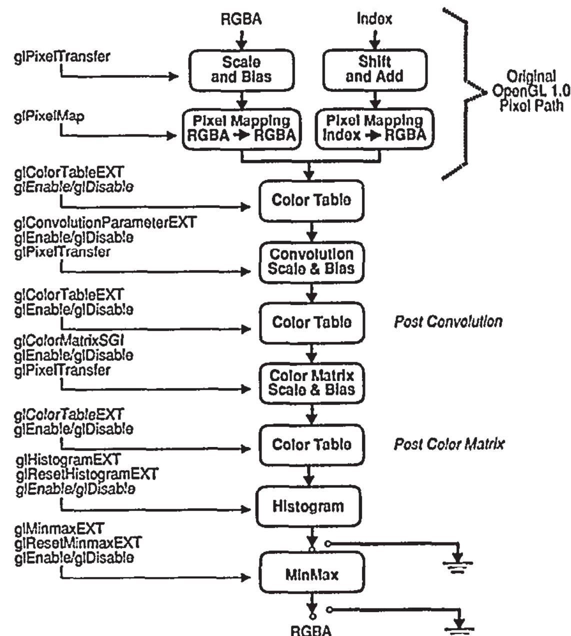

Figure 2: The extended OpenGL pixel path including the convolution, histogram, color matrix, and color table extensions.

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.

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

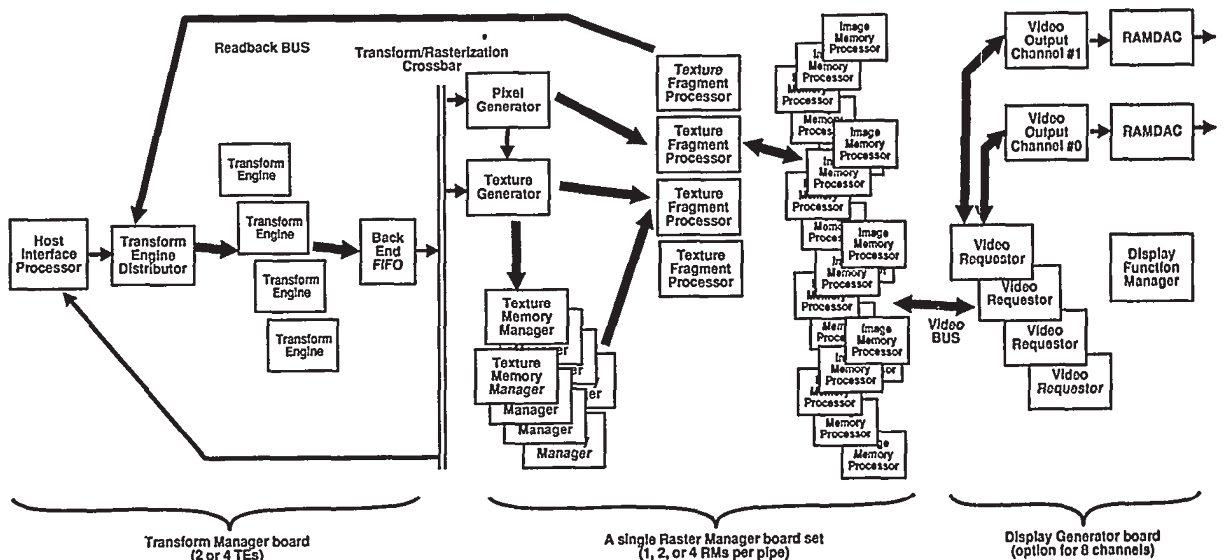

Figure 3: ASIC-level diagram showing the InfiniteReality graphics subsystem architecture.

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

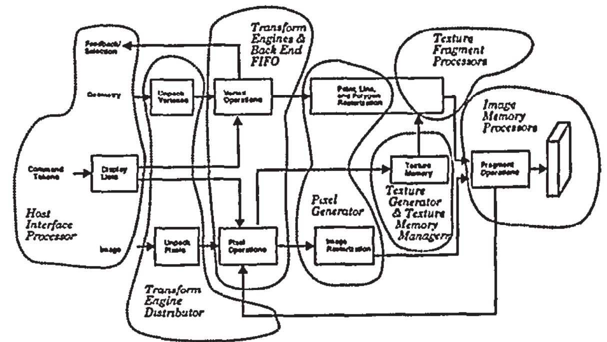

Figure 4: How the conceptual OpenGL state machine roughly maps to InfiniteReality's rendering ASICs.

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.

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-

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.