InfiniteReality: A Real-Time Graphics System

John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal
Silicon Graphics Computer Systems

ABSTRACT

The InfiniteReality™ graphics system is the first general-purpose workstation system specifically designed to deliver 60Hz steady frame rate high-quality rendering of complex scenes. This paper describes the InfiniteReality system architecture and presents novel features designed to handle extremely large texture databases, maintain control over frame rendering time, and allow user customization for diverse video output requirements. Rendering performance expressed using traditional workstation metrics exceeds seven million lighted, textured, antialiased triangles per second, and 710 million textured antialiased pixels filled per second.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation

1 INTRODUCTION

This paper describes the Silicon Graphics InfiniteReality architecture which is the highest performance graphics workstation ever commercially produced. The predecessor to the InfiniteReality system, the RealityEngine™, [Akel93] was the first example of what we term a third-generation graphics system. As a third-generation system, the target capability of the RealityEngine was to render lighted, smooth shaded, depth buffered, texture mapped, antialiased triangles. The level of realism achieved by RealityEngine graphics was well-matched to the application requirements of visual simulation (both flight and ground based simulation), location based entertainment [Paus96], defense imaging, and virtual reality. However, application success depends on two areas: the ability to provide convincing levels of realism and to deliver real-time performance of constant scene update rates of 60Hz or more. High frame rates reduce interaction latency and minimize symptoms of motion sickness in visual simulation and virtual reality applications. If frame rates are not constant, the visual integrity of the simulation is compromised.

InfiniteReality is also an example of a third-generation graphics system in that its target rendering quality is similar to that of RealityEngine. However, where RealityEngine delivered performance in the range of 15-30 Hz for most applications, the fundamental design goal of the InfiniteReality graphics system is to deliver real-time performance to a broad range of applications. Furthermore, the goal is to deliver this performance far more economically than competitive solutions.

Author contacts: {montrym | drb | dignam | migdal}@sgi.com

Most of the features and capabilities of the InfiniteReality architecture are designed to support this real-time performance goal. Minimizing the time required to change graphics modes and state is as important as increasing raw transformation and pixel fill rate. Many of the targeted applications require access to very large textures and/or a great number of distinct textures. Permanently storing such large amounts of texture data within the graphics system itself is not economically viable. Thus methods must be developed for applications to access a “virtual texture memory” without significantly impacting overall performance. Finally, the system must provide capabilities for the application to monitor actual geometry and fill rate performance on a frame by frame basis and make adjustments if necessary to maintain a constant 60Hz frame update rate.

Aside from the primary goal of real-time application performance, two other areas significantly shaped the system architecture. First, this was Silicon Graphics’ first high-end graphics system to be designed from the beginning to provide native support for OpenGL™. To support the inherent flexibility of the OpenGL architecture, we could not take the traditional approach for the real-time market of providing a black-box solution such as a flight simulator [Scha83].

The InfiniteReality system is fundamentally a sort-middle architecture [Moln94]. Although interesting high-performance graphics architectures have been implemented using a sort-last approach [Moln92][Evan92], sort-last is not well-suited to supporting OpenGL framebuffer operations such as blending. Furthermore, sparse sort-last architectures make it difficult to rasterize primitives into the framebuffer in the order received from the application as required by OpenGL.

The second area that shaped the graphics architecture was the need for the InfiniteReality system to integrate well with two generations of host platforms. For the first year of production, the InfiniteReality system shipped with the Onyx host platform. Currently, the InfiniteReality system integrates into the Onyx2 platform. Not only was the host to graphics interface changed between the two systems, but the I/O performance was also significantly improved. Much effort went into designing a graphics system that would adequately support both host platforms.

The remainder of the paper is organized as follows. The next section gives an architectural overview of the system. Where appropriate, we contrast our approach to that of the RealityEngine system. Section 3 elaborates on novel functionality that enables real-time performance and enhanced video capabilities. Section 4 discusses the performance of the system. Finally, concluding remarks are made in Section 5.

2 ARCHITECTURE

It was a goal to be able to easily upgrade Onyx RealityEngine systems to InfiniteReality graphics. Accordingly, the physical partitioning of the InfiniteReality boardset is similar to that of

RealityEngine; there are three distinct board types: the Geometry, Raster Memory, and Display Generator boards (Figure 1).

The Geometry board comprises a host computer interface, command interpretation and geometry distribution logic, and four Geometry Engine processors in a MIMD arrangement. Each Raster Memory board comprises a single fragment generator with a single copy of texture memory, 80 image engines, and enough framebuffer memory to allocate 512 bits per pixel to a 1280x1024 framebuffer. The display generator board contains hardware to drive up to eight display output channels, each with its own video timing generator, video resize hardware, gamma correction, and digital-to-analog conversion hardware.

Systems can be configured with one, two or four raster memory boards, resulting in one, two, or four fragment generators and 80, 160, or 320 image engines.

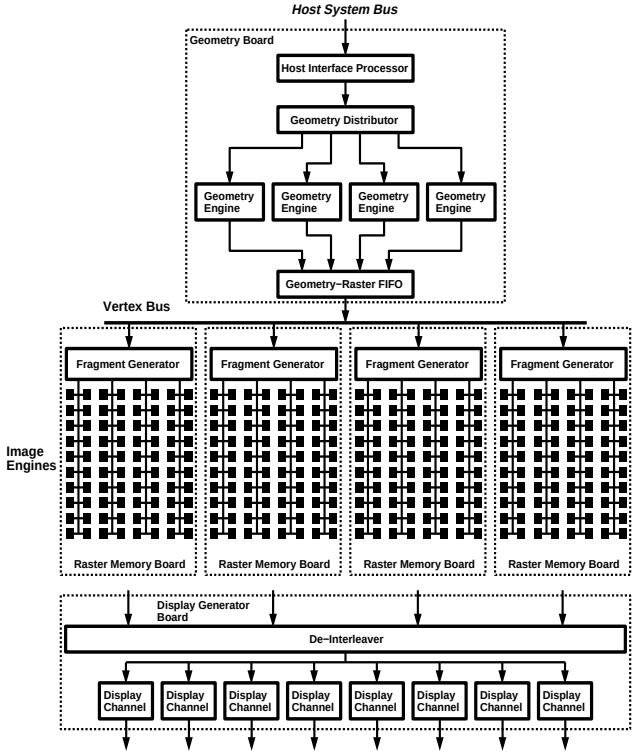

Figure 1: Board-level block diagram of the maximum configuration with 4 Geometry Engines, 4 Raster Memory boards, and a Display Generator board with 8 output channels.

2.1 Host Interface

There were significant system constraints that influenced the architectural design of InfiniteReality. Specifically, the graphics system had to be capable of working on two generations of host platforms. The Onyx2 differs significantly from the shared memory multiprocessor Onyx in that it is a distributed shared memory multiprocessor system with cache-coherent non-uniform memory access. The most significant difference in the graphics system design is that the Onyx2 provides twice the host-to-graphics bandwidth (400MB/sec vs. 200MB/sec) as does Onyx. Our challenge was to design a system that would be matched to the host-to-graphics data rate of the Onyx2, but still provide similar performance with the limited I/O capabilities of Onyx.

We addressed this problem with the design of the display list subsystem. In the RealityEngine system, display list processing had been handled by the host. Compiled display list objects were stored in host memory, and one of the host processors traversed the display list and transferred the data to the graphics pipeline using programmed I/O (PIO).

With the InfiniteReality system, display list processing is handled in two ways. First, compiled display list objects are stored in host memory in such a way that leaf display objects can be “pulled” into the graphics subsystem using DMA transfers set up by the Host Interface Processor (Figure 1). Because DMA transfers are faster and more efficient than PIO, this technique significantly reduces the computational load on the host processor so it can be better utilized for application computations. However, on the original Onyx system, DMA transfers alone were not fast enough to feed the graphics pipe at the rate at which it could consume data. The solution was to incorporate local display list processing into the design.

Attached to the Host Interface Processor is 16MB of synchronous dynamic RAM (SDRAM). Approximately 15MB of this memory is available to cache leaf display list objects. Locally stored display lists are traversed and processed by an embedded RISC core. Based on a priority specified using an OpenGL extension and the size of the display list object, the OpenGL display list manager determines whether or not a display list object should be cached locally on the Geometry board. Locally cached display lists are read at the maximum rate that can be consumed by the remainder of the InfiniteReality pipeline. As a result, the local display list provides a mechanism to mitigate the host to graphics I/O bottleneck of the original Onyx. Note that if the total size of leaf display list objects exceeds the resident 15MB limit, then some number of objects will be pulled from host memory at the reduced rate.

2.2 Geometry Distribution

The Geometry Distributor (Figure 1) passes incoming data and commands from the Host Interface Processor to individual Geometry Engines for further processing. The hardware supports both round-robin and least-busy distribution schemes. Since geometric processing requirements can vary from one vertex to another, a least-busy distribution scheme has a slight performance advantage over round-robin. With each command, an identifier is included which the Geometry-Raster FIFO (Figure 1) uses to recreate the original order of incoming primitives.

2.3 Geometry Engines

When we began the design of the InfiniteReality system, it became apparent that no commercial off-the-shelf floating point processors were being developed which would offer suitable price/performance. As a result, we chose to implement the Geometry Engine Processor as a semicustom application specific integrated circuit (ASIC).

The heart of the Geometry Engine is a single instruction multiple datapath (SIMD) arrangement of three floating point cores, each of which comprises an ALU and a multiplier plus a 32 word register

file with two read and two write ports (Figure 2). A 2560 word on-chip memory holds elements of OpenGL state and provides scratch storage for intermediate calculations. A portion of the working memory is used as a queue for incoming vertex data. Early simulations of microcode fragments confirmed that high bandwidth to and from this memory would be required to get high utilization of the floating point hardware. Accordingly, each of the three cores can perform two reads and one write per instruction to working memory. Note that working memory allows data to be shared easily among cores. A dedicated float-to-fix converter follows each core, through which one floating point result may be written per instruction.

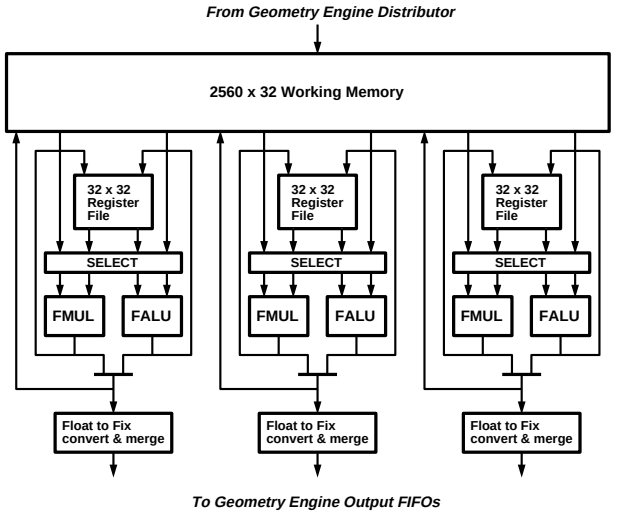

Figure 2: Geometry Engine

We used a very simple scheduler to evaluate the performance effect of design trade-offs on critical microcode fragments. One of the trade-offs considered was the number of pipeline stages in the floating point arithmetic blocks. As we increased the depth of the pipeline from one to four stages, the machine's clock speed and throughput increased. For more than four stages, even though the clock speed improved, total performance did not because our code fragments did not have enough unrelated operations to fill the added computation slots.

Quite often machine performance is expressed in terms of vertex rates for triangles in long strips whereas application performance is much more likely to be determined by how well a system handles very short strips, with frequent mode changes. The problem of accelerating mode changes and other non-benchmark operations has enormous impact on the microcode architecture, which in turn influences aspects of the instruction set architecture.

To accelerate mode change processing, we divide the work associated with individual OpenGL modes into distinct code modules. For example, one module can be written to calculate lighting when one infinite light source is enabled, another may be tuned for one local point light source, and still another could handle a single spotlight. A general module exists to handle all cases which do not have a corresponding tuned module. Similarly, different microcode modules would be written to support other OpenGL modes such as texture coordinate generation or backface elimination. A table consisting of pointers to the currently active modules is maintained in GE working memory. Each vertex is processed by executing the active modules in the table-specified sequence. When a mode change occurs, the appropriate table entry is changed. Vertex processing time degrades slowly and predictably as additional operations are turned on, unlike microcode architectures which implement hyper-optimized fast paths for selected bundles of mode settings, and a slow general path for all other combinations.

Since microcode modules tend to be relatively short, it is desirable to avoid the overhead of basic-block preamble and postamble code. All fields necessary to launch and retire a given operation, including memory and register file read and write controls, are specified in the launching microinstruction.

2.4 Geometry-Raster FIFO

The output streams from the four Geometry Engines are merged into a single stream by the Geometry-Raster FIFO. A FIFO large enough to hold 65536 vertexes is implemented in SDRAM. The merged geometry engine output is written, through the SDRAM FIFO, to the Vertex Bus. The Geometry-Raster FIFO contains a 256-word shadow RAM which keeps a copy of the latest values of the Fragment Generator and Image Engine control registers. By eliminating the need for the Geometry Engines to retain shadowed raster state in their local RAMs, the shadow RAM permits raster mode changes to be processed by only one of the Geometry Engines. This improves mode change performance and simplifies context switching.

2.5 Vertex Bus

One of our most important goals was to increase transform-limited triangle rates by an order of magnitude over RealityEngine. Given our desire to retain a sort-middle architecture, we were forced to increase the efficiency of the geometry-raster crossbar by a factor of ten. Whereas the RealityEngine system used a *Triangle Bus* to move triangle parameter slope information from its Geometry Engines to its Fragment Generators, the InfiniteReality system employs a *Vertex Bus* to transfer only screen space vertex information. Vertex Bus data is broadcast to all Fragment Generators. The Vertex Bus protocol supports the OpenGL triangle strip and triangle fan constructs, so the Vertex Bus load corresponds closely to the load on the host-to-graphics bus. The Geometry Engine triangle strip workload is reduced by around 60 percent by not calculating triangle setup information. However, hardware to assemble screen space primitives and compute parameter slopes is now incorporated into the Fragment Generators.

2.6 Fragment Generators

In order to provide increased user-accessible physical texture memory capacity at an acceptable cost, it was our goal to have only one copy of texture memory per Raster Memory board. A practical consequence of this is that there is also only one fragment generator per raster board. Figure 3 shows the fragment generator structure.

Connected vertex streams are received and assembled into triangle

primitives. The Scan Converter (SC) and Texel Address Calculator (TA) ASICs perform scan conversion, color and depth interpolation, perspective correct texture coordinate interpolation and level-of-detail computation. Up to four fragments, corresponding to 2x2 pixel regions are produced every clock. Scan conversion is performed by directly evaluating the parameter plane equations at

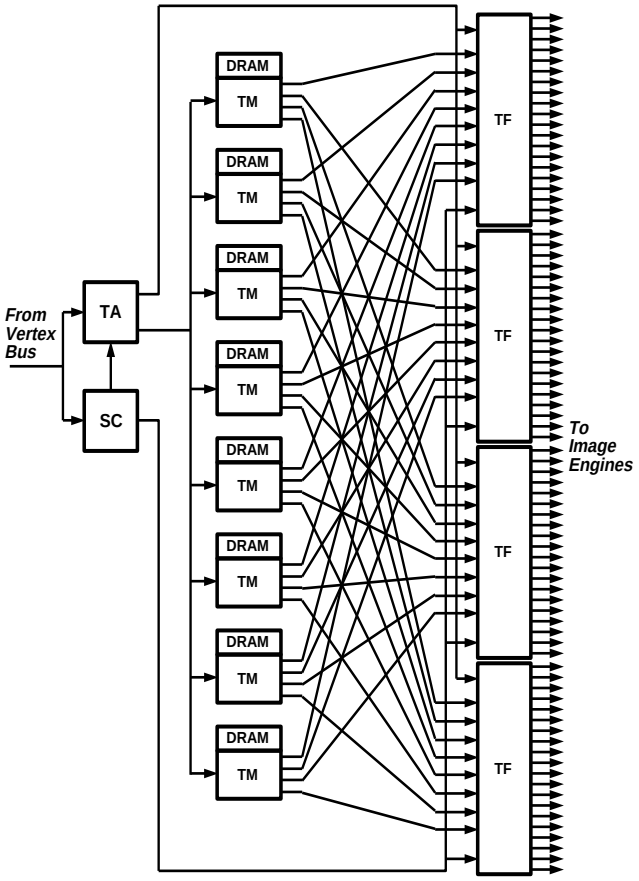

Figure 3: Fragment Generator

each pixel [Fuch85] rather than by using an interpolating DDA as was done in the RealityEngine system. Compared to a DDA, direct evaluation requires less setup time per triangle at the expense of more computation per pixel. Since application trends are towards smaller triangles, direct parameter evaluation is a more efficient solution.

Each texture memory controller (TM) ASIC performs the texel lookup in its four attached SDRAMs, given texel addresses from the TA. The TMs combine redundant texel requests from neighboring fragments to reduce SDRAM access. The TMs forward the resulting texel values to the appropriate TF ASIC for texture filtering, texture environment combination with interpolated color, and fog application. Since there is only one copy of the texture memory distributed across all the texture SDRAMs, there must exist a path from all 32 texture SDRAMs to all Image Engines. The TMs and TFs implement a two-rank omega network [Hwan84] to perform the required 32-to-80 sort.

2.7 Image Engines

Fragments output by a single Fragment Generator are distributed equally among the 80 Image Engines owned by that generator. Each Image Engine controls a single 256K x 32 SDRAM that comprises its portion of the framebuffer. Framebuffer memory per Image Engine is twice that of RealityEngine, so a single raster board system supports eight sample antialiasing at 1280 x 1024 or four sample antialiasing at 1920 x 1200 resolution.

2.8 Framebuffer Tiling

Three factors contributed to development of the framebuffer tiling scheme: the desire for load balancing of both drawing and video requests; the various restrictions on chip and board level packaging; and the requirement to keep on-chip FIFOs small.

In systems with more than one fragment generator, different fragment generators are each responsible for two-pixel wide vertical strips in framebuffer memory. If horizontal strips had been used instead, the resulting load imbalance due to display requests would have required excessively large FIFOs at the fragment generator inputs. The strip width is as narrow as possible to minimize the load imbalance due to drawing among fragment generators.

The Fragment Generator scan-conversion completes all pixels in a two pixel wide vertical strip before proceeding to the next strip for every primitive. To keep the Image Engines from limiting fill rate on large area primitives, all Image Engines must be responsible for part of every vertical strip owned by their Fragment Generator. Conversely, for best display request load balancing, all Image Engines must occur equally on every horizontal line. For a maximum system, the Image Engine framebuffer tiling repeat pattern is a rectangle 320 pixels wide by 80 pixels tall (320 is the number of Image Engines in the system and 80 is the number of Image Engines on one Raster Memory board).

2.9 Display Hardware

Each of the 80 Image Engines on the Raster Memory boards drives one or two bit serial signals to the Display Generator board. Two wires are driven if there is only one Raster Memory board, and one wire is driven if there are two or more. Unlike RealityEngine, both the number of pixels sent per block and the aggregate video band width of 1200 Mbytes/sec are independent of the number of Raster Memory boards. Four ASICs on the display board (Figure 4) deserialize and de-interleave the 160 bit streams into RGBA10, RGB12, L16, Stereo Field Sequential (FS), or color indexes. The cursor is also injected at this point. A total of 32,768 color index map entries are available.

Color component width is maintained at 12 bits through the gamma table outputs. A connector site exists with a full 12 bit per component bus, which is used to connect video option boards. Option boards support the Digital Video Standard CCIR 601 and a digital pixel output for hardware-in-the-loop applications.

The base display system consists of two channels, expandable to eight. Each display channel is autonomous, with independent

video timing and image resizing capabilities. The final channel output drives eight-bit digital-to-analog converters which can run up to a 220Mhz pixel clock rate. Either RGB or Left/Right Stereo Field Sequential is available from each channel.

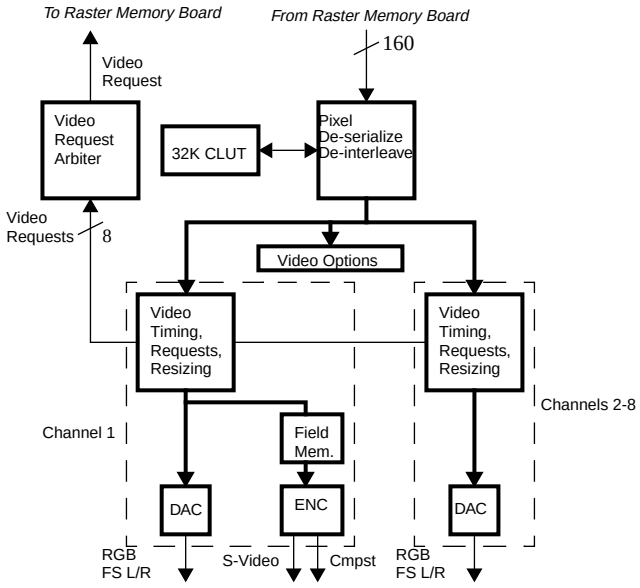

Figure 4: Display System

Video synchronization capabilities were expanded to support independent timing per channel (Figure 5). Swap events are constrained to happen during a common interval. Three different methods are used to synchronize video timing to external video sources. *Framelocking* is the ability to rate lock, using line rate dividers, two different video outputs whose line rates are related by small integer ratios. Line rate division is limited by the programmability of the phase-locked-loop gain and feedback parameters and the jitter spectrum of the input genlock source. The start of a video frame is detected by programmable sync pattern recognition hardware. Disparate source and displayed video formats which exceed the range of framelock are vertically locked by simply performing an asynchronous *frame reset* of the display video timing hardware. In this instance, the pixel clock is created by multiplying an oscillator clock. Identical formats may be *genlocked*. With frame lock or genlock, the frame reset from the pattern recognition hardware will be synchronous, and therefore cause no disturbance of the video signal being sent to the monitor.

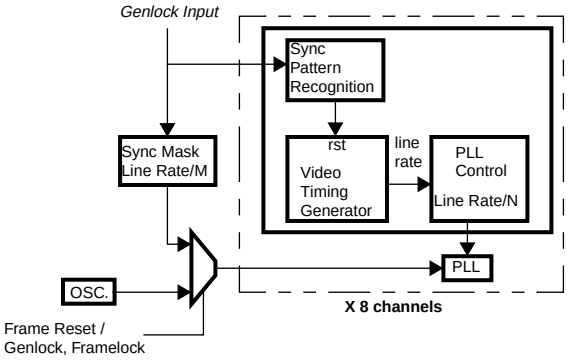

Figure 5: Video Synchronization

Certain situations require the synchronization of drawing between separate graphics systems. This is required in visual simulation installations where multiple displays are each driven by their own graphics system. If one graphics system takes longer than a frame time to draw a scene, the other graphics systems must be kept in lock step with the slowest one. InfiniteReality uses an external *swap ready* wire connecting all the graphics systems together in a wired AND configuration.

The video outputs of all the graphics systems are first locked together. Each pipe monitors the swap ready wire to determine if all the other pipes have finished drawing. A local buffer swap is only allowed to happen if all the graphics systems are ready to swap. In order to cope with slight pipe to pipe variations in video timing, a write exclusion window exists around the swap ready register to guarantee all pipes make the same decision.

Finally an NTSC or PAL output is available with any of the eight channels as the source. Resizing hardware allows for the scaling of any source resolution or windowed subset, to NTSC or PAL resolution.

3 FEATURES

3.1 Virtual Texture

The size of texture databases is rapidly increasing. Texture data that cover the entire world at one meter resolution will be commercially available in 1998. This corresponds to a texture size of 40,000,000 x 20,000,000 texels. Advanced simulation users need to be able to navigate around such large data in real-time. To meet this need, the InfiniteReality system provides hardware and software support for very large *virtual textures*, that is, textures which are too large to reside in physical texture memory.

Previous efforts to support texture databases larger than available texture memory required that the scene database modeler partition the original texture into a number of smaller tiles such that a subset of them fit into physical texture memory. The disadvantage of this approach is that the terrain polygons need to be subdivided so that no polygon maps to more than one texture tile. The InfiniteReality system, by contrast, allows the application to treat the original large texture as a single texture.

We introduce a representation called a *clip-map* which significantly reduces the storage requirements for very large textures. To illustrate the usefulness of the clip-map representation, we observe that the amount of texture data that can be viewed at one time is limited by the resolution of the display monitor. For example, using trilinear mip-map textures on a 1024x1024 monitor, the highest resolution necessary occurs just before a transition to the next coarser level of detail. In this case the maximum amount of resident texture required for any map level is no more than 2048 x 2048 for the finer map, and 1024x1024 for the coarser map, regardless of the size of the original map level. This is the worst case which occurs when the texture is viewed from directly above. In most applications the database is viewed obliquely and in perspective. This greatly reduces the maximum size of a particular level-of-detail that must be in texture memory in order to render a frame.

Recall that a mip-map represents a source image with a pyramidal set of two-dimensional images, each of which covers the full area of the source image at successively coarser resolution [Will83]. A clip-map can be thought of as a subset of the mip-map of the entire texture. It has two parts: a clip-map pyramid which is exactly the same as the coarser levels of the original mip-map, and a clip-map stack which holds a subset of the data in the original mip-map for the finest levels of detail. The clip-map stack levels all have the same size in texture memory, but each successively coarser level covers four times the source image area of the immediately finer level. Figure 6 illustrates the relationships between levels in a clip-map when viewed from above a textured database. The clip-map stack levels are centered on a common point. Each stack level represents larger and larger areas as the resolution of the data they contain becomes coarser and coarser. Figure 7 illustrates a clip-map for a 32K x 32K source image using a 2K x 2K clip-map tile size. Note that the clip-map representation requires about 1/64 the storage of the equivalent 32K x 32K mip-map.

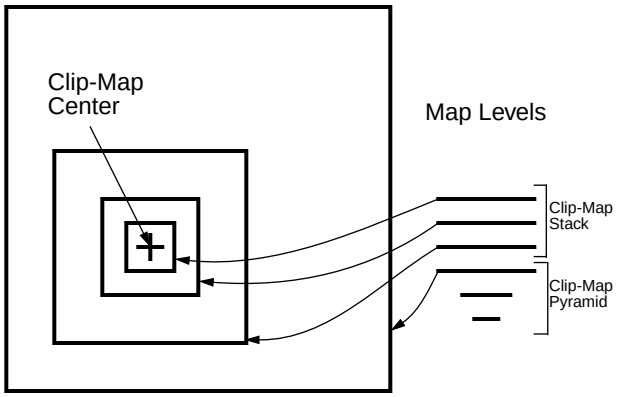

Figure 6: Clip-Map Levels

Because the clip-map stack does not contain the entire texture the position of the clip-map stack needs to be updated to track the viewer's position, or more optimally the center of the viewer's gaze. As the viewer's position or gaze moves, the contents of the clip-map stack should be updated to reflect this movement. New texture data is loaded into the texture memory to replace the texture data that is no longer required. The rate of update of texture data is highest for the finest clip-map stack level and becomes less for coarser stack levels of the clip-map. In the InfiniteReality system, it is not necessary to replace all data in a clip-map level when only a few texels actually need to be updated. The hardware loads new texture data over the old and automatically performs the correct addressing calculations using offset registers. Additionally, the Fragment Generators contain registers that define the clip-map center as it moves through the texture.

If the stack tile size is chosen correctly and the clip-map stack is updated properly as the viewpoint moves through the scene, the InfiniteReality system will produce images identical to those that would have been produced if the entire source mip-map had been resident in texture memory.

It cannot always be guaranteed that the texture data requested during triangle rendering will be available at the requested level of detail. This may occur if the size of the clip-map tile has been chosen to be too small, or the update of the stack center failed to keep pace with the motion of the viewer. The InfiniteReality texture subsystem detects when texture is requested at a higher resolution than is available in texture memory. It substitutes the best available data which is data at the correct spatial position, but at a coarser level-of-detail than requested. As a result, the rendered scene will have regions where the texture will be coarser than if the entire mip-map were resident in texture memory. However, it will otherwise be rendered correctly. This substitution mechanism limits the required clip-map tile size and reduces the required texture update rate.

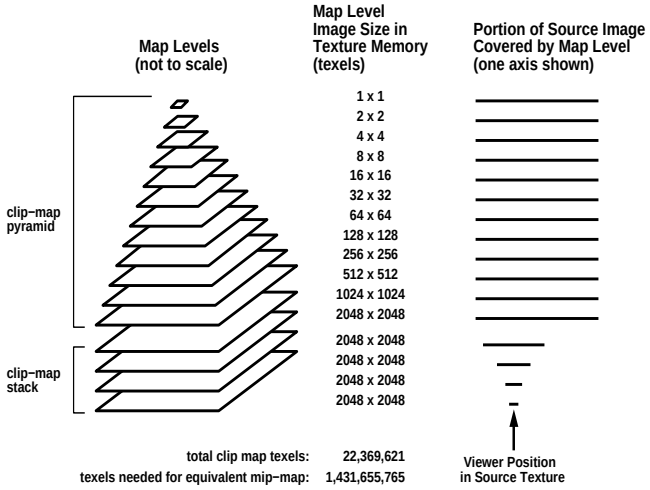

Figure 7: 32Kx32K texture represented as a 2Kx2K clip-map.

The Fragment Generator is limited to addressing a 32K x 32K clip-map. The addressability of clip-maps can be extended to arbitrary texture sizes through software. The software layer needs only to keep track of and implement a transformation from an arbitrarily large texture space into the texture space addressable by the hardware.

3.2 Texture Loading and Paging

We minimize the performance impact of large amounts of texture paging in the design of InfiniteReality system. The graphics subsystem interprets texture images directly as specified by the OpenGL programmer so no host processor translation is required. The front end of the Geometry Subsystem includes pixel unpacking and format conversion hardware; DMA hardware directly implements stride and padding address arithmetic as required by OpenGL. The Fragment Generators accept raster-order texture images at Vertex Bus-limited rates. To eliminate the need for the host computer to make and retain copies of loaded textures for context switching, the hardware supports texture image reads back to the host.

The Geometry-Raster FIFO maintains a separate path through which data bound for texture memory is routed. When the Fragment Generators are busy with fill-limited primitives, pending texture data is transferred over the Vertex Bus and routed to write queues in the TM ASICs. When a moderate amount of texture data is queued for a particular texture DRAM, the TM suspends draw

access and writes the queue contents to that DRAM. Because total bandwidth to and from texture memory is an order of magnitude greater than that of the Vertex Bus, this action only slightly impacts fill rate. For fill-limited scenes, however, this approach utilizes Vertex Bus cycles which would otherwise go unused. Synchronization barrier primitives ensure that no texture is referenced until it has been fully loaded, and conversely, that no texture loading occurs until the data to be overwritten is no longer needed.

3.3 Scene Load Management

3.3.1 Pipeline Performance Statistics

Regardless of the performance levels of a graphics system, there may be times when there are insufficient hardware resources to maintain a real-time frame update rate. These cases occur when the pipeline becomes either geometry or fill rate limited. Rather than extending frame time, it is preferable for the application to detect such a situation and adjust the load on the pipeline appropriately.

The InfiniteReality system provides a mechanism for performing feedback-based load management with application-accessible monitoring instrumentation. Specifically, counters are maintained in the Geometry-Raster FIFO that monitor stall conditions on the Vertex Bus as well as wait conditions upstream in the geometry path. If the counters indicate that there is geometry pending in the Geometry-Raster FIFO, but writes to the Vertex Bus are stalled, then the system is fill rate limited. On the other hand, if the FIFO is empty, then the system is either host or geometry processing limited. By extracting these measurements, the application can take appropriate action whenever a geometry or fill rate bottleneck would have otherwise caused a drop in frame rate.

A common approach to a geometry limited pipeline is for the application to temporarily reduce the complexity of objects being drawn starting with those objects that are most distant from the viewer [Funk93][Rohl94]. This allows the application to reduce the polygon count being sent to the pipeline without severely impacting the visual fidelity of the scene. However, since distant objects do not tend to cover many pixels, this approach is not well-suited to the case where the pipeline is fill limited. To control fill limited situations, the InfiniteReality uses a novel technique termed *dynamic video resizing.*

3.3.2 Dynamic Video Resizing

Every frame, fill requirements are evaluated, and a scene is rendered to the framebuffer at a potentially reduced resolution such that drawing completes in less than one frame time. Prior to display on the monitor, the image is scaled up to the nominal resolution of the display format. Based on the current fill rate requirements of the scene, framebuffer resolution is continuously adjusted so that rendering can be completed within one frame time. A more detailed explanation follows.

Pipeline statistics are gathered each frame and used to determine if the current frame is close to being fill limited. These statistics are then used to estimate the amount by which the drawing time should be reduced or increased on the subsequent frame. Drawing time is altered by changing the resolution at which the image is rendered in the framebuffer. Resolution is reduced if it is estimated that the new image cannot be drawn in less than a frame time. Resolution can be increased if it was reduced in prior scenes, and the current drawing time is less than one frame. The new frame may now be drawn at a different resolution from the previous one. Resolution can be changed in X or Y or both. Magnifying the image back up to the nominal display resolution is done digitally, just prior to display. The video resizing hardware is programmed for the matching magnification ratios, and the video request hardware is programmed to request the appropriate region of the framebuffer.

Finally, to ensure the magnification ratio is matched with the resolution of the frame currently being displayed, loading of the magnification and video request parameters is delayed until the next swap buffer event for that video channel. This ensures that even if scene rendering exceeds one frame time, the resizing parameters are not updated until drawing is finished.

Each channel is assigned a unique display ID, and the swap event is detected for each of these ID's. This swap forces the loading of the new resize parameters for the corresponding video channel, and allows channels with different swap rates to resize.

Note that the effectiveness of this technique is independent of scene content and does not require modifications to the scene data base.

3.4 Video Configurability

One of the goals for the InfiniteReality system was to enable our customers to both create their own video timing formats and to assign formats to each video channel.

This required that the underlying video timing hardware had to be more flexible than in the RealityEngine. Capabilities were expanded in the video timing and request hardware's ability to handle color field sequential, interlace, and large numbers of fields. The biggest change needed was an expanded capability to detect unique vertical sync signatures when genlocking to an external video signal. Since our customers could define vertical sync signatures whose structure could not be anticipated, the standard approach of simply hard-wiring the detection of known sync patterns would have been inadequate. Therefore, each video channel contains programmable pattern recognition hardware, which analyzes incoming external sync and generates resets to the video timing hardware as required.

In previous graphics systems, multi-channel support was designed as an afterthought to the basic single channel display system. This produced an implementation that was lacking in flexibility and was not as well integrated as it could have been. In the RealityEngine system, support for multiple channels was achieved by pushing video data to an external display board. The software that created multi-channel combinations was required to emulate the system hardware in order to precisely calculate how to order the video data. Ordering had to be maintained so each channel's local FIFO would not overflow or underflow. This approach was not very robust and made it impossible for our customers to define their own format combinations.

In the InfiniteReality system, every video channel was designed to be fully autonomous in that each has its own programmable pixel clock and video timing. Each video channel contains a FIFO, sized to account for latencies in requesting frame buffer memory. Video data is requested based on each channel's FIFO levels. A round robin arbiter is sufficient to guarantee adequate response time for multiple video requests.

Format combinations are limited to video formats with the same swap rate. Thus, the combination of 1280x1024@60Hz + 640x480@180Hz field sequential + 1024x768@120Hz stereo + NTSC is allowed but combining 1920x1080@72Hz and 50Hz PAL is not.

In order to achieve our design goal of moving more control of video into the hands of our customers, two software programs were developed. The first program is the Video Format Compiler or **vfc**. This program generates a file containing the microcode used to configure the video timing hardware. The source files for the compiler use a language whose syntax is consistent with standard video terminology. Source files can be generated automatically using templates. Generating simple block sync formats can be accomplished without any specific video knowledge other than knowing the width, height and frame rate of the desired video display format. More complex video formats can be written by modifying an existing source file or by starting from scratch. The Video Format Compiler generates an object file which can be loaded into the display subsystem at any time. Both the video timing hardware and the sync pattern recognition hardware are specified by the **vfc** for each unique video timing format.

The second program is the InfiniteReality combiner or **ircombine**. Its primary uses are to define combinations of existing video formats, verify that they operate within system limitations, and to specify various video parameters. Both a GUI and a command line version of this software are provided. Once a combination of video formats has been defined, it can be saved out to a file which can be loaded at a later time. The following is a partial list of **ircombine** capabilities:

- Attach a video format to a specific video channel
- Verify that the format combination can exist within system limits
- Define the rectangular area in framebuffer memory to be displayed by each channel
- Define how data is requested for interlace formats
- Set video parameters (gain, sync on RGB, setup etc.)
- Define genlock parameters (internal/external, genlock source format, horizontal phase, vertical phase)
- Control the NTSC/PAL encoder (source channel, input window size, filter size)
- Control pixel depth and size

4 PERFORMANCE

The InfiniteReality system incorporates 12 unique ASIC designs implemented using a combination of 0.5 and 0.35 micron, three-layer metal semiconductor fabrication technology.

Benchmark performance numbers for several key operations are summarized in Tables 1, 2, and 3. In general, geometry processing rates are seven to eight times that of the RealityEngine system and pixel fill rates are increased by over a factor of three. Note that the depth buffered fill rate assumes that every Z value passes the Z comparison and must be replaced which is the worst case. In practice, not every pixel will require replacement so the actual depth buffered fill rates will fall between the stated depth buffered and non depth buffered rate.

Although the benchmark numbers are impressive, our design goals focused on achieving real-time application performance rather than the highest possible benchmark numbers. Predicting application performance is a complex subject for which there are no standard accepted metrics. Some of the reasons that applications do not achieve peak benchmark rates include the frequent execution of mode changes (e.g. assigning a different texture, changing a surface material, etc.), the use of short triangle meshes, and host processing limitations. We include execution times for commonly performed mode changes (Table 4) as well as performance data for shorter triangle meshes (Table 5). Practical experience with a variety of applications has shown that the InfiniteReality system is successful in achieving our real-time performance goals.

We were pleasantly surprised by the utility of video resizing as a fill rate conservation tool. Preliminary simulations indicated that we could expect to dynamically reduce framebuffer resolution up to ten percent in each dimension without substantially degrading image quality. In practice, we find that we can frequently reduce framebuffer resolution up to 25% in each dimension which results in close to a 50% reduction in fill rate requirements.

unlit, untextured tstrips	11.3 Mtris / sec
unlit, textured tstrips	9.5 Mtris / sec
lit, textured tstrips	7.1 Mtris / sec

Table 1: Non Fill-Limited Geometry Rates

non-depth buffered, textured, antialiased	830 Mpix / sec
depth buffered, textured, antialiased	710 Mpix / sec

Table 2: Non Geometry-Limited Fill Rates (4 Raster Memory boards)

RGBA8	83.1 Mpix / sec (332 Mb / sec)

Table 3: Peak Pixel Download Rate

glMaterial	240,941 / sec
glColorMaterial	337,814 / sec
glBindTexture	244,537 / sec
glMultMatrixf	1,110,779 / sec
glPushMatrix / glPopMatrix	1,489,454 / sec

Table 4: Mode Change Rates

Length 2 triangle strips	4.7 Mtris / sec

Table 5: Geometry Rates for Short Triangle Strips

Length 4 triangle strips	7.7 Mtris / sec
Length 6 triangle strips	8.6 Mtris / sec
Length 8 triangle strips	9.0 Mtris / sec
Length 10 triangle strips	11.3 Mtris / sec

Table 5: Geometry Rates for Short Triangle Strips

The above numbers are for unlit, untextured triangle strips. Other types of triangle strips scale similarly.

The performance of the InfiniteReality system makes practical the use of multipass rendering techniques to enhance image realism. Multipass rendering can be used to implement effects such as reflections, Phong shading, shadows, and spotlights [Sega92]. Figure 8 shows a frame from a multipass rendering demonstration running at 60Hz on the InfiniteReality system. This application uses up to five passes per frame and renders approximately 40,000 triangles each frame.

5 CONCLUSION

The InfiniteReality system achieves real-time rendering through a combination of raw graphics performance and capabilities designed to enable applications to achieve guaranteed frame rates. The flexible video architecture of the InfiniteReality system is a general solution to the image generation needs of multichannel visual simulation applications. A true OpenGL implementation, the InfiniteReality brings unprecedented performance to traditional graphics-intensive applications. This underlying performance, together with new rendering functionality like virtual texturing, paves the way for entirely new classes of applications.

Acknowledgments

Many of the key insights in the area of visual simulation came from Michael Jones and our colleagues on the Performer team. Gregory Eitzmann helped architect the video subsystem and associated software tools. The multipass rendering demo in Figure 8 was produced by Luis Barcena Martin, Ignacio Sanz-Pastor Revorio, and Javier Castellar. Finally, the authors would like to thank the talented and dedicated InfiniteReality and Onyx2 teams for their tremendous efforts. Every individual made significant contributions to take the system from concept to product.

REFERENCES

[Akel93] K. Akeley, “RealityEngine Graphics”, *SIGGRAPH 93 Proceedings*, pp. 109-116.

[Evan92] Evans and Sutherland Computer Corporation, “*Freedom Series Technical Report*”, Salt Lake City, Utah, Oct. 92.

[Funk93] T. Funkhouser, C. Sequin, “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments”, *SIGGRAPH 93 Proceedings*, pp. 247-254.

[Fuch85] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, J. Eyles, and J. Poulton, “Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes”, *SIGGRAPH 85 Proceedings*, pp. 111-120.

[Hwan84] K. Hwang, F. Briggs, “*Computer Archetecture and Parallel Processing*”, McGraw-Hill, New York, pp. 350-354, 1984.

[Moln92] S. Molnar, J. Eyles, J. Poulton, “Pixelflow: High-Speed Rendering Using Image Composition”, *SIGGRAPH 92 Proceedings*, pp. 231-240.

[Moln94] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, “A Sorting Classification of Parallel Rendering”, *IEEE Computer Graphics and Applications, July 94, pp. 23-32.*

[Paus96] R. Paush, J. Snoddy, R. Taylor, E. Haseltine, “Disney’s Aladdin: First Steps Toward Storytelling in Virtual Reality”, *SIGGRAPH 96 Proceedings*, pp. 193-203.

[Scha83] B. Schachter, “Computer Image Generation”, John Wiley & Sons, New York, 1983.

[Sega92] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. Haeberli, “Fast Shadows and Lighting Effects Using Texture Mapping”, *SIGGRAPH 92 Proceedings*, pp. 249-252.

[Rohl94] J. Rohlf, J. Helman, “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3DGraphics, *SIGGRAPH 94 Proceedings*, pp. 381-394.

[Will83] L. Williams, “Pyramidal Parametrics”, *SIGGRAPH 83 Proceedings*, pp. 1-11.

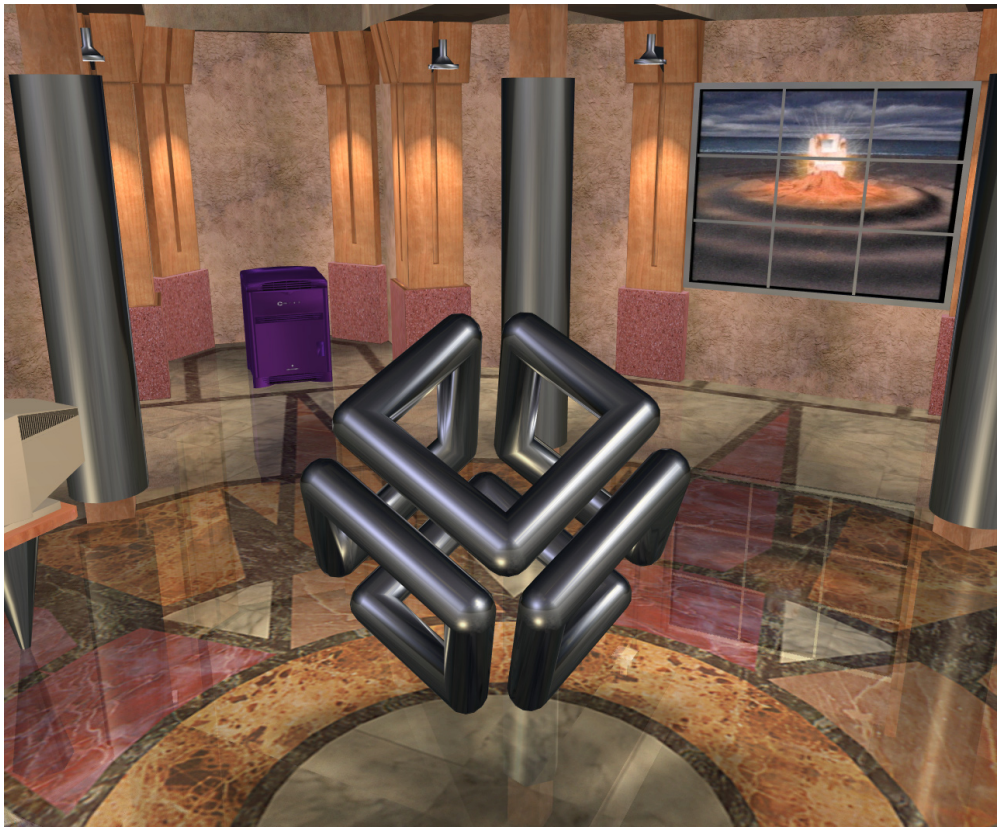

Figure 8: An example of a high quality image generated at 60 Hz using multipass rendering techniques