RealityEngine Graphics

Kurt Akeley
Silicon Graphics Computer Systems*

Abstract

The RealityEngine™ graphics system is the first of a new generation of systems designed primarily to render texture mapped, antialiased polygons. This paper describes the architecture of the RealityEngine graphics system, then justifies some of the decisions made during its design. The implementation is near-massively parallel, employing 353 independent processors in its fullest configuration, resulting in a measured fill rate of over 240 million antialiased, texture mapped pixels per second. Rendering performance exceeds 1 million antialiased, texture mapped triangles per second. In addition to supporting the functions required of a general purpose, high-end graphics workstation, the system enables realtime, "out-the-window" image generation and interactive image processing.

CR Categories and Subject Descriptors: I.3.1 **[Computer Graphics]:** Hardware Architecture; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism - *color, shading, shadowing, and texture*

1 Introduction

This paper describes and to a large extent justifies the architecture chosen for the RealityEngine graphics system. The designers think of this system as our first implementation of a third-generation graphics system. To us a generation is characterized not by the scope of capabilities of an architecture, but rather by the capabilities for which the architecture was primarily designed – the target capabilities with maximized performance. Because we designed our first machine in the early eighties, our notion of first generation corresponds to this period. Floating point hardware was just becoming available at reasonable prices, framebuffer memory was still quite expensive, and application-specific integrated circuits (ASICs) were not readily available. The resulting machines had workable transformation capabilities, but very limited framebuffer processing capabilities. In particular, smooth shading and depth buffering, which require substantial framebuffer hardware and memory, were not available. Thus the target capabilities of first-generation machines were the transformation and rendering of flat-shaded points, lines, and polygons. These primitives were not lighted, and hidden surface elimination, if required, was accomplished by algorithms implemented by the application. Examples of such systems are the Silicon Graphics Iris 3000 (1985) and the Apollo DN570 (1985). Toward the end of the first-generation period advances in technology allowed lighting, smooth shading, and depth buffering to be implemented, but only with an order of magnitude less performance than was available to render flat-shaded lines and polygons. Thus the target capability of these machines remained first-generation. The Silicon Graphics 4DG (1986) is an example of such an architecture.

Because first-generation machines could not efficiently eliminate hidden surfaces, and could not efficiently shade surfaces even if the application was able to eliminate them, they were more effective at rendering wireframe images than at rendering solids. Beginning in 1988 a second-generation of graphics systems, primarily workstations rather than terminals, became available. These machines took advantage of reduced memory costs and the increased availability of ASICs to implement deep framebuffers with multiple rendering processors. These framebuffers had the numeric ability to interpolate colors and depths with little or no performance loss, and the memory capacity and bandwidth to support depth buffering with minimal performance loss. They were therefore able to render solids and full-frame scenes efficiently, as well as wireframe images. The Silicon Graphics GT (1988)[11] and the Apollo DN590 (1988) are early examples of second-generation machines. Later second-generation machines, such as the Silicon Graphics VGX[12] the Hewlett Packard VRX, and the Apollo DN10000[4] include texture mapping and antialiasing of points and lines, but not of polygons. Their performances are substantially reduced, however, when texture mapping is enabled, and the texture size (of the VGX) and filtering capabilities (of the VRX and the DN10000) are limited.

The RealityEngine system is our first third-generation design. Its target capability is the rendering of lighted, smooth shaded, depth buffered, texture mapped, antialiased triangles. The initial target performance was 1/2 million such triangles per second, assuming the triangles are in short strips, and 10 percent intersect the viewing frustum boundaries. Textures were to be well filtered (8-sample linear interpolation within and between two mipmap[13] levels) and large enough (1024×1024) to be usable as true images, rather than simply as repeated *textures*. Antialiasing was to result in high-quality images of solids, and was to work in conjunction with depth buffering, meaning that no application sorting was to be required. Pixels were to be filled at a rate sufficient to support 30Hz rendering of full-screen images. Finally, the performance on second-generation primitives (lighted, smooth shaded, depth buffered) was to be no lower than that of the VGX, which renders roughly 800,000 such mesh triangles per second. All of these goals were achieved.

The remainder of this paper is in four parts: a description of the architecture, some specifics of features supported by the architecture, alternatives considered during the design of the architecture, and finally some appendixes that describe performance and implementation details.

*2011 N. Shoreline Blvd., Mountain View, CA 94043 USA, kurt@sgi.com

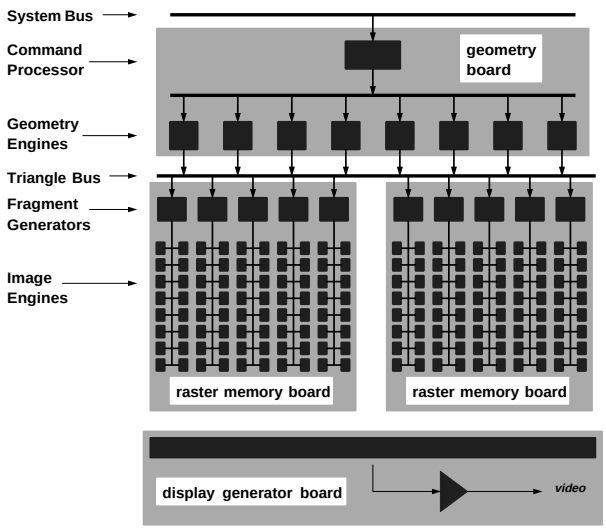

Figure 1. Board-level block diagram of an intermediate configuration with 8 Geometry Engines on the geometry board, 2 raster memory boards, and a display generator board.

2 Architecture

The RealityEngine system is a 3, 4, or 6 board graphics accelerator that is installed in a MIPS RISC workstation. The graphics system and one or more MIPS processors are connected by a single system bus. Figure 1 is a board-level block diagram of the RealityEngine graphics accelerator. The geometry board comprises an input FIFO, the Command Processor, and 6, 8, or 12 Geometry Engines. Each raster memory board comprises 5 Fragment Generators (each with its own complete copy of the texture memory), 80 Image Engines, and enough framebuffer memory to allocate 256 bits per pixel to a 1280×1024 framebuffer. The display generator board supports all video functions, including video timing, genlock, color mapping, and digital-to-analog conversion. Systems can be configured with 1, 2, or 4 raster memory boards, resulting in 5, 10, or 20 Fragment Generators and 80, 160, or 320 Image Engines.

To get an initial notion of how the system works, let's follow a single triangle as it is rendered. The position, color, normal, and texture coordinate commands that describe the vertexes of the triangle in object coordinates are queued by the input FIFO, then interpreted by the Command Processor. The Command Processor directs all of this data to one of the Geometry Engines, where the coordinates and normals are transformed to eye coordinates, lighted, transformed to clip coordinates, clipped, and projected to window coordinates. The associated texture coordinates are transformed by a third matrix and associated with the window coordinates and colors. Then window coordinate slope information regarding the red, green, blue, alpha, depth, and texture coordinates is computed.

The projected triangle, ready for rasterization, is then output from the Geometry Engine and broadcast on the Triangle Bus to the 5, 10, or 20 Fragment Generators. (We distinguish between pixels generated by rasterization and pixels in the framebuffer, referring to the former as fragments.) Each Fragment Generator is responsible for the rasterization of 1/5, 1/10, or 1/20 of the pixels in the framebuffer, with the pixel assignments finely interleaved to insure that even small triangles are partially rasterized by each of the Fragment Generators. Each Fragment Generator computes the intersection of the set of pixels that are fully or partially covered by the triangle and the set of pixels in the framebuffer that it is responsible for, generating a fragment for each of these pixels. Color, depth, and texture coordinates are assigned to each fragment based on the initial and slope values computed by the Geometry Engine. A subsample mask is assigned to the fragment based on the portion of each pixel that is covered by the triangle. The local copy of the texture memory is indexed by the texture coordinates, and the 8 resulting samples are reduced by linear interpolation to a single color value, which then modulates the fragment's color.

The resulting fragments, each comprising a pixel coordinate, a color, a depth, and a coverage mask, are then distributed to the Image Engines. Like the Fragment Generators, the Image Engines are each assigned a fixed subset of the pixels in the framebuffer. These subsets are themselves subsets of the Fragment Generator allocations, so that each Fragment Generator communicates only with the 16 Image Engines assigned to it. Each Image Engine manages its own dynamic RAM that implements its subset of the framebuffer. When a fragment is received by an Image Engine, its depth and color sample data are merged with the data already stored at that pixel, and a new aggregate pixel color is immediately computed. Thus the image is complete as soon as the last primitive has been rendered; there is no need for a final framebuffer operation to resolve the multiple color samples at each pixel location to a single displayable color.

Before describing each of the rendering operations in more detail, we make the following observations. First, after it is separated by the Command Processor, the stream of rendering commands merges only at the Triangle Bus. Second, triangles of sufficient size (a function of the number of raster memory boards) are processed by almost all the processors in the system, avoiding only 5, 7, or 11 Geometry Engines. Finally, small to moderate FIFO memories are included at the input and output of each Geometry Engine, at the input of each Fragment Generator, and at the input of each Image Engine. These memories smooth the flow of rendering commands, helping to insure that the processors are utilized efficiently.

2.1 Command Processor

That the Command Processor is required at all is primarily a function of the OpenGL™ [8][7] graphics language. OpenGL is modal, meaning that much of the state that controls rendering is included in the command stream only when it changes, rather than with each graphics primitive. The Command Processor distinguishes between two classes of this modal state. OpenGL commands that are expected infrequently, such as matrix manipulations and lighting model changes, are broadcast to all the Geometry Engines. OpenGL commands that are expected frequently, such as vertex colors, normals, and texture coordinates, are shadowed by the Command Processor, and the current values are bundled with each rendering command that is passed to an individual Geometry Engine. The Command Processor also breaks long connected sequences of line segments or triangles into smaller groups, each group passing to a single Geometry Engine. The size of these groups is a tradeoff between the increased vertex processing efficiency of larger groups (due to shared vertexes within a group) and the improved load balancing that results from smaller groups. Finally, because the Command Processor must interpret each graphics command, it is also able to detect invalid command sequences and protect the

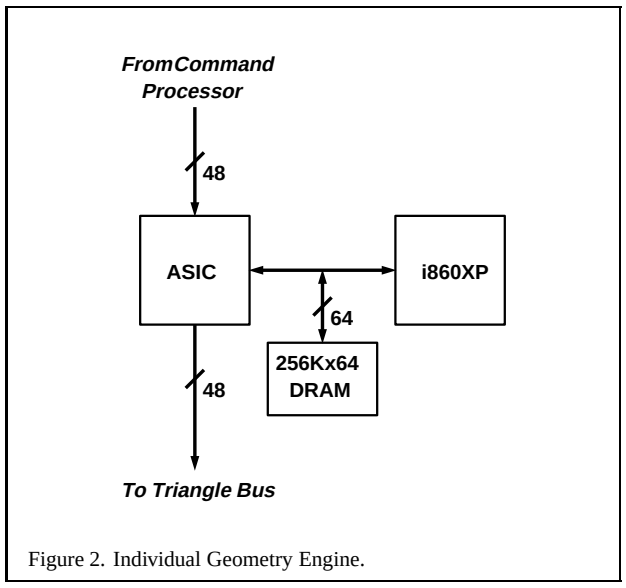

Figure 2. Individual Geometry Engine.

subsequent processors from their effects.

Non-broadcast rendering commands are distributed to the Geometry Engines in pure round-robin sequence, taking no account of Geometry Engine loading. This approach was chosen for its simplicity, and is efficient because the processing requirements of primitives are usually very similar, and because the input and output FIFOs of each Geometry Engine smooth the imbalances due to data-dependent processing such as clipping.

2.2 Geometry Engines

The core of each Geometry Engine is an Intel i860XP processor. Operating at 50MHz, the combined floating point multiplier and ALU can achieve a peak performance of 100 MFLOPS. Each Intel processor is provided 2 Mbytes of combined code/data dynamic memory, and is supported by a single ASIC that implements the input and output FIFOs, a small register space from which the i860XP accesses incoming commands, and specialized data conversion facilities that pack computed slope data into a format accepted by the Fragment Generators. (Figure 2.)

All Geometry Engine code is first developed in C, which is cross compiled for the i860XP on MIPS RISC development systems. Code that is executed frequently is then re-coded in i860XP assembly code, showing the greatest improvement in performance where scheduling of the vector floating point unit is hand optimized. The assembly code is written to conform to the compiler's link conventions, so that hand-coded and compiled modules are interchangeable for development and documentation purposes.

Most floating point arithmetic is done in single precision, but much of the texture arithmetic, and all depth arithmetic after projection transformation, must be done in double precision to maintain the required accuracy. After transformation, lighting, and clipping, the rasterization setup code treats each parameter as a plane equation, computing its signed slope in the positive X and Y screen directions. Because the parameters of polygons with more than 3 vertexes may be non-planar, the Geometry Engine decomposes all polygons to triangles.

2.3 Triangle Bus

The Triangle Bus acts as a crossbar, connecting the output of each Geometry Engine to the inputs of all the Fragment Generators. Because all Geometry Engine output converges at this bus, it is a potential bottleneck. To avoid performance loss, the Triangle Bus was designed with bandwidth to handle over one million shaded, depth buffered, texture mapped, antialiased triangles per second, more than twice the number of primitives per second that were anticipated from an 8 Geometry Engine system. This performance cushion allows the later-conceived 12 Geometry Engine system to render at full performance, in spite of the greater than expected performance of the individual engines.

In addition to broadcasting the rasterization data for triangles to the Fragment Generators, the Triangle Bus broadcasts point and line segment descriptions, texture images, and rasterization mode changes such as blending functions.

2.4 Fragment Generators

Although each Fragment Generator may be thought of as a single processor, the data path of each unit is actually a deep pipeline. This pipeline sequentially performs the initial generation of fragments, generation of the coverage mask, texture address generation, texture lookup, texture sample filtering, texture modulation of the fragment color, and fog computation and blending. These tasks are distributed among the four ASICs and eight dynamic RAMs that comprise each Fragment Generator. (Figure 3.)

Fragments are generated using Pineda arithmetic[9], with the algorithm modified to traverse only pixels that are in the domain of the Fragment Generator. A coverage mask is generated for 4, 8, or 16 sample locations, chosen on a regular 8×8 subsample grid within the square boundaries of the pixel. The hardware imposes no constraints on which subset of the 64 subsample locations is chosen, except that the same subset is chosen for each pixel. The subset may be changed by the application between frames.

Depth and texture coordinate sample values are always computed at the center-most sample location, regardless of the fragment coverage mask. The single depth sample is later used by the Image Engines to derive accurate depth samples at each subpixel location, using the X and Y depth slopes. Taking the texture sample at a consistent location insures that discontinuities are avoided at pixels that span multiple triangles. Color sample values are computed at the center-most sample location only if it is within the perimeter of the triangle. Otherwise the color sample is taken at a sample location within the triangle perimeter that is near the centroid of the covered region. Thus color samples are always taken within the triangle perimeter, and therefore never wrap to inappropriate values.

Based on a level-of-detail (LOD) calculation and the texture coordinate values at the fragment center, the addresses of the eight texels nearest the sample location in the mipmap of texture images are produced. Eight separate banks of texture memory are then accessed in parallel at these locations. The 8 16-bit values that result are merged with a trilinear blend, based on the subtexel coordinates and the LOD fraction, resulting in a single texture color that varies smoothly from frame to frame in an animation. The entire bandwidth of the 8-bank texture memory is consumed by a single Fragment Engine, so each Fragment Engine includes its own complete copy of all texture images in its texture memory, allowing all Fragment Generators to operate in parallel. Separate FIFO memories on the address and data ports of each texture memory bank

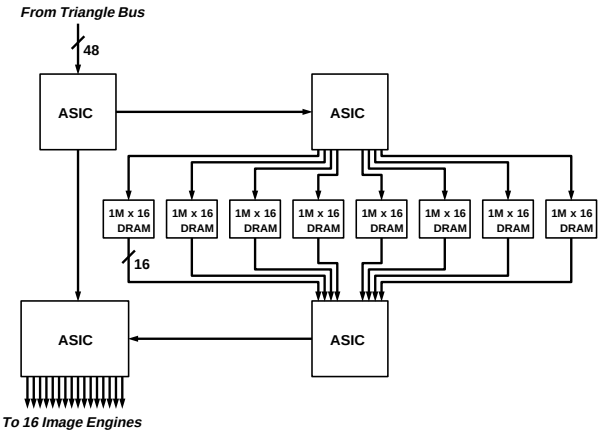

Figure 3. Individual Fragment Generator.

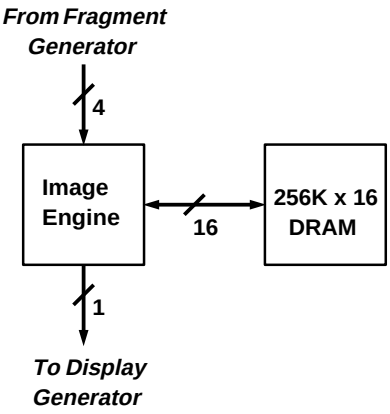

Figure 4. Individual Image Engine.

insure that random page boundary crossings do not significantly degrade the bandwidth available from the dynamic RAMs.

The last ASIC in the Fragment Generator applies the texture color to the fragment's smooth shaded color, typically by modulation. It then indexes its internal fog table with the fragment's depth value and uses the resulting fog blend factor (computed by linear interpolation between the two nearest table entries) to blend the fragment color with the application-defined fog color.

2.5 Image Engines

Fragments output by a single Fragment Generator are distributed equally among the 16 Image Engines connected to that generator. When the triangle was first accepted by the Fragment Generator for processing, its depth slopes in the X and Y screen directions were broadcast to each Image Engine, which stored them for later use. When an Image Engine accepts a fragment, it first uses these two slope values and the fragment's depth sample value to reconstruct the depth values at each subpixel sample location. The arithmetic required for this operation is simplified because the subpixel sample locations are fixed to a regular 8×8 grid. The calculations are linear because depth values have been projected to window coordinates just like the X and Y pixel coordinates. At each sample location corresponding to a '1' in the fragment's coverage mask, the computed depth value is compared to the depth value stored in the framebuffer. If the comparison succeeds, the framebuffer color at that subsample location is replaced by the fragment color, and the framebuffer depth is replaced by the derived fragment depth. If any change is made to the pixel's contents, the aggregate pixel color is recomputed by averaging the subpixel sample colors, and is immediately written to the displayable color buffer that will contain the final image.

Each Image Engine controls a single 256K $\times$ 16 dynamic RAM that comprises its portion of the framebuffer. (Figure 4.) When the framebuffer is initialized, this memory is partitioned equally among 4K, 8K, or 16K pixels, resulting in pixels with 1024, 512, or 256 bits. All subsample depth and color samples, as well as the one, two, or four displayable color buffers and other auxiliary buffers, are stored in this memory. By default, colors are stored with 12 bits per red, green, blue, and alpha component in both the displayable buffers and the subpixel samples. Depth values are 32 bits each, and are normally required only for each subpixel sample, not for the displayable color buffer or buffers. Color and depth sample resolutions can be reduced to 8,8,8 and 24 bits to allow more samples to be stored per pixel. The 4K partition stores 8 high-resolution samples per pixel, or 16 low-resolution samples per pixel, in addition to two displayable color buffers of the same resolution. The 8K partition stores 4 high-resolution samples per pixel, or 8 low-resolution samples per pixel, again with two displayable color buffers of the same resolution. The 16K partition cannot be used to support multisample antialiasing.

Because the number of raster memory boards (1, 2, or 4) and the number of pixels per Image Engine (4K, 8K, or 16K) are independent, the RealityEngine system supports a wide variety of framebuffer dimensions, color and depth resolutions, and subpixel samples. For example, a single raster board system supports 16-sample antialiasing at 640×512 resolution or aliased rendering at 1280×1024 resolution, and a 4-board system supports 8-sample antialiasing at true HDTV (1920×1035) resolution or 16-sample antialiasing at 1280×1024 resolution.

2.6 Display Hardware

Each of the 80 Image Engines on the raster memory board drives a single-bit, 50 MHz path to the display board, delivering video data at 500 MBytes per second. All 160 single-bit paths of a two raster memory board configuration are active, doubling the peak video data rate. The paths are time multiplexed by pairs of raster memory boards in the four board configuration. Ten crossbar ASICs on the display board assemble the 80 or 160 single-bit streams into individual color components or color indexes. Color components are then dithered from 12 bits to 10 bits and gamma corrected using 1024×8 lookup tables. The resulting 8-bit color components drive digital-to-analog converters and are output to the monitor. Color indexes are dereferenced in a 32K-location lookup table, supporting separate color lookup tables for each of up to 40 windows on the screen. Per-pixel display modes, such as the color index offset, are supported by a combination of Image Engine and display board hardware, driven by window ID bits stored in the framebuffer [1].

3 Features

This section provides additional information regarding the architecture's antialiasing, texture mapping, stereo, and clipping capabilities.

3.1 Antialiasing

The architecture supports two fundamentally different antialiasing techniques: alpha and multisample. Alpha antialiasing of points and lines is common to second generation architectures. Alpha antialiasing is implemented using subpixel and line-slope indexed tables to generate appropriate coverage values for points and lines, compensating for the subpixel position of line endpoints. Polygon coverage values are computed by counting the '1's in the full precision 8×8 coverage mask. The fragment alpha value is scaled by the fractional coverage value, which varies from 0.0, indicating no coverage, to 1.0, indicating complete coverage. If pixel blending is enabled, fragments are blended directly into the color buffer – no subpixel sample locations are accessed or required. Alpha antialiasing results in higher quality points and lines than does multisample antialiasing, because the resolution of the filter tables is greater than the 4 bit equivalent of the 16-sample mask. While alpha antialiased primitives should be rendered back-to-front or front-to-back (depending on the blend function being used) to generate a correct image, it is often possible to get an acceptable point or line image without such sorting. Alpha antialiased polygons, however, must be sorted near to far to get an acceptable image. Thus this technique is efficiently applied to polygons only in 2D scenes, such as instrument panels, where primitive ordering is fixed and a slight increase in quality is desired.

Multisample antialiasing has already been described. Its principal advantage over alpha antialiasing is its order invariance - points, lines, and polygons can be drawn into a multisample buffer in any order to produce the same final image. Two different mask generation techniques are supported in multisample mode, each with its own advantages and disadvantages. The default mask generation mode is called point sampled; the alternate mode is area sampled. A point sampled mask is geometrically accurate, meaning that each mask bit is set if and only if its subpixel location is within the perimeter of the point, line, or polygon outline. (Samples on the primitive's edge are included in exactly one of the two adjacent primitives.) Such masks insure the correctness of the final image, at the expense of its filtered quality. The final image is correct because all the samples that comprise it are geometrically valid - none having been taken outside their corresponding primitives. It is poorly sampled because the number of bits set in the mask may not closely correspond to the actual area of the pixel that is covered by the primitive, and the final filtering quality depends on this correspondence. Area sampling attempts to insure that the number of '1's in the sample mask is correct plus or minus 1/2 a sample, based on the actual coverage of pixel area by the primitive. (Figure 5.) In order to accomplish this, area sampled masks necessarily include samples that are outside the primitive outline, resulting in image artifacts such as polygon protrusions at silhouettes and T-junctions. Area sampled masks are implemented with a technique that is related to the one described by Andreas Schilling[10]. Point and area sampling can be selected by the application program on a per-primitive basis.

The desirable multisample property of order invariance is lost if alpha transparency and pixel blending are used. Alpha does sometimes carry significant information, usually as a result of the alpha channel in the texture application. For example, trees are often drawn as single polygons, using an alpha matte to express their shape. In order to handle alpha transparency without requiring pixel blending, the Image Engines have the ability to convert fragment alpha values to pseudo-random masks, which are then logically ANDed with the fragment's coverage mask. This method, while not geometrically accurate, provides usable antialiasing of texture mattes, and is order invariant.

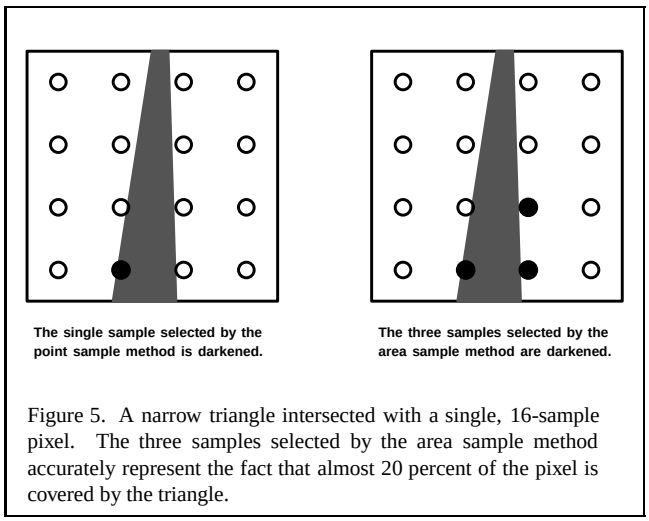

Figure 5. A narrow triangle intersected with a single, 16-sample pixel. The three samples selected by the area sample method accurately represent the fact that almost 20 percent of the pixel is covered by the triangle.

3.2 Texture Mapping

In addition to the 2-dimension texture maps described in the architecture section, 1- and 3-dimension maps are also supported. The eight million texel memory associated with each Fragment Generator stores 2D mipmapped images up to 1024×1024, and 3D nonmipmapped images up to $256 \times 256 \times 64$. Thus 3D textures can be used to render volumetric images of substantial resolution, at rates up to 30 frames per second. The S, T, and R texture coordinates of each fragment are computed by interpolating S/W, T/W, R/W, and 1/W, then doing the correct divisions at each pixel, resulting in perspective-corrected mapping. Level-of-detail is also computed for each pixel, based on the worst-case of the four pixel-to-texel X and Y ratios.

Linear filtering of the nearest texels and mipmap levels is supported for 1D, 2D, and 3D textures, blending a total of 16 texel colors in the 3D mode. In the 2D case such linear filtering is commonly known as trilinear. Bicubic interpolation is supported for 2D, nonmipmapped textures, again blending 16 texels. There is no support for cubic filtering of 1D or 3D textures, or of any mipmapped textures. The default 16-bit texel size supports RGBA texels at 4-bits per component, RGB texels at 5-bits per component (6 bits for green), intensity-alpha texels at 8-bits per component, and intensity texels at 12-bits per component. 32-bit and 48-bit texels can be specified by the application with proportional loss of performance. The maximum RBGA texel resolution is 12-bits per component, equal to the maximum framebuffer color resolution.

Texture magnification can be done by extrapolation of mipmap levels, resulting in a sharpening of the highest resolution mipmap image, or the highest resolution image can be blended with a replicated 256×256 detail image, greatly increasing the apparent resolution of the texture without requiring excessive texture storage. Filter functions for RGB and for alpha can be specified separately

to improve the quality of texture mattes. Finally, texture memory can be loaded from the application processor's memory at the rate of 80 million 16-bit texels per second, allowing the application to treat texture memory as a managed cache of images.

3.3 Stereo in a Window

Image Engine memory can be configured with separate left and right color buffers for both the visible and nonvisible displayable color buffers, resulting in a total of four 48-bit color buffers per pixel. The display hardware alternately displays the left and right buffer contents of the visible buffers of all windows so configured, and drives a sync signal that can be used to control screen or head-mounted shutters. This stereo-in-a-window capability is both formally and practically compatible with the X protocol: formally because neither framebuffer dimensions nor pixel aspect ratio are changed when it is enabled or disabled, and practically because it allows monoscopic windows such as menus to be rendered and displayed correctly. To reduce eye fatigue, it is advisable to select a reduced-dimension framebuffer when the window system is initialized, allowing the frame display rate to be increased to 90+ Hz within the 140 MHz pixel limit of the display board.

3.4 Fast Clipping

RealityEngine polygon clipping is faster than that of our earlier designs for two fundamental reasons: it is implemented more efficiently, and it is required less often. Higher efficiency results from the MIMD Geometry Engine architecture. Because each of the engines executes an independent code sequence, and because each has significant input and output FIFOs, random clipping delays affect only a single engine and are averaged statistically across all the engines. Also, because each Geometry Engine comprises only a single processor, all of that engine's processing power can be devoted to the clipping process. SIMD architectures are less efficient because all processors are slowed when a single processor must clip a polygon. Pipelines of processors, and even MIMD arrangements of short pipelines, are less efficient because only a fraction of available processing power is available to the clipping process.

The requirement for clipping is reduced through a technique we call scissoring. Near and far plane clipping are done as usual, but the left, right, bottom, and top frustum edges are moved well away from the specified frustum, and all triangles that fall within the expanded frustum are projected to extended window coordinates. If culling is done by the application, almost no triangles will actually intersect the sides of the expanded frustum. Projected triangles that are not fully within the viewport are then scissored to match the edges of the viewport, eliminating the portions that are not within the viewport. The Pineda rasterization algorithm that is employed easily and efficiently handles the additional rectilinear edges that result, and no fragment generation performance is lost on scissored regions.

4 Design Alternatives

We think that the most interesting part of design is the alternatives considered, and the reasons for choices, rather than the details of the result. This section highlights some of these alternatives, in roughly decreasing order of significance.

4.1 Single-pass Antialiasing

Multi-pass accumulation buffer antialiasing using an accumulation buffer [3] is order invariant, and produces high-quality images in 10 to 20 passes. Further, a system that was fast enough to render 10 to 20 full scene images per frame would be a fantastic generator of aliased images. So why design a complex, multisample framebuffer to accomplish the same thing in one pass? The answer is that significantly more hardware would be required to implement a multi-pass machine with equivalent performance. This is true not only because the multi-pass machine must traverse and transform the object coordinates each pass, but in particular because texture mapping would also be performed for each pass. The component costs for traversal, transformation, parameter interpolation, and texture mapping constitute well over half of the multisample machine cost, and they are not replicated in the multisample architecture. A competing multi-pass architecture would have to replicate this hardware in some manner to achieve the required performance. Even the PixelFlow architecture[6], which avoids repeated traversal and transformation by buffering intermediate results, must still rasterize and texture map repeatedly.

4.2 Multisample Antialiasing

Multisample antialiasing is a rather brute-force technique for achieving order invariant single-pass antialiasing. We investigated alternative sorting buffer techniques derived from the A-buffer algorithm[2], hoping for higher filter quality and correct, single-pass transparency. These techniques were rejected for several reasons. First, sort buffers are inherently more complex than the multisample buffer and, with finite storage allocations per pixel, they may fail in undesirable ways. Second, any solution that is less exact than multisampling with point sampled mask generation will admit rendering errors such as polygon protrusions at silhouettes and T-junctions. Finally, the multisample algorithm matches the single-sample algorithm closely, allowing OpenGL pixel techniques such as stencil, alpha test, and depth test to work identically in single or multisample mode.

4.3 Immediate Resolution of Multisample Color

Our initial expectation was that rendering would update only the multisample color and depth values, requiring a subsequent resolution pass to reduce these values to the single color values for display. The computational expense of visiting all the pixels in the framebuffer is high, however, and the resolution pass damaged the software model, because OpenGL has no explicit scene demarcations. Immediate resolution became much more desirable when we realized that the single most common resolution case, where the fragment completely replaces the pixel's contents (i.e. the fragment mask is all ones and all depth comparisons pass) could be implemented by simply writing the fragment color to the color buffer, making no change to the 4, 8, or 16 subsample colors, and specially tagging the pixel. Only if the pixel is subsequently partially covered by a fragment is the color in the color buffer copied to the appropriate subsample color locations. This technique increases the performance in the typical rendering case and eliminates the need for a resolution pass.

4.4 Triangle Bus

All graphics architectures that implement parallel primitive processing and parallel fragment/pixel processing must also implement a crossbar somewhere between the geometry processors and the framebuffer[5]. While many of the issues concerning the placement of this crossbar are beyond the scope of this paper, we will mention some of the considerations that resulted in our Triangle Bus architecture. The RealityEngine Triangle Bus is a crossbar between the Geometry Engines and the Fragment Generators. Described in RealityEngine terms, architectures such as the Evans & Sutherland Freedom Series™ implement Geometry Engines and Fragment Generators in pairs, then switch the resulting fragments to the appropriate Image Engines using a fragment crossbar network. Such architectures have an advantage in fragment generation efficiency, due both to the improved locality of the fragments and to only one Fragment Generator being initialized per primitive. They suffer in comparison, however, for several reasons. First, transformation and fragment generation rates are linked, eliminating the possibility of tuning a machine for unbalanced rendering requirements by adding transformation or rasterization processors. Second, ultimate fill rate is limited by the fragment bandwidth, rather than the primitive bandwidth. For all but the smallest triangles the quantity of data generated by rasterization is much greater than that required for geometric specification, so this is a significant bottleneck. (See Appendix 2.) Finally, if primitives must be rendered in the order that they are specified, load balancing is almost impossible, because the number of fragments generated by a primitive varies by many orders of magnitude, and cannot be predicted prior to processor assignment. Both OpenGL and the core X renderer require such ordered rendering.

The PixelFlow[6] architecture also pairs Geometry Engines and Fragment Generators, but the equivalent of Image Engines and memory for a 128×128 pixel tile are also bundled with each Geometry/Fragment pair. The crossbar in this architecture is the compositing tree that funnels the contents of rasterized tiles to a final display buffer. Because the framebuffer associated with each processor is smaller than the final display buffer, the final image is assembled as a sequence of 128×128 logical tiles. Efficient operation is achieved only when each logical tile is rasterized once in its entirety, rather than being revisited when additional primitives are transformed. To insure that all primitives that correspond to a logical tile are known, all primitives must be transformed and sorted before rasterization can begin. This substantially increases the system's latency, and requires that the rendering software support the notion of frame demarcation. Neither the core X renderer nor OpenGL support this notion.

4.5 12-bit Color

Color component resolution was increased from the usual 8 bits to 12 bits for two reasons. First, the RealityEngine framebuffer stores color components in linear, rather than gamma-corrected, format. When 8-bit linear intensities are gamma corrected, single bit changes at low intensities are discernible, resulting in visible banding. The combination of 12-to-10 bit dithering and 10-bit gamma lookup tables used at display time eliminates visible banding. Second, it is intended that images be computed, rather than just stored, in the RealityEngine framebuffer. Volume rendering using 3D textures, for example, requires back-to-front composition of multiple slices through the data set. If the framebuffer resolution is just sufficient to display an acceptable image, repeated compositions will degrade the resolution visibly. The 12-bit components allow substantial framebuffer composition to take place before artifacts become visible.

Figure 6. A scene from a driving simulation running full-screen at 30 Hz.

Figure 7. A 12x magnified subregion of the scene in figure 6. The sky texture is properly sampled and the silhouettes of the ground and buildings against the sky are antialiased.

Conclusion

The RealityEngine system was designed as a high-end workstation graphics accelerator with special abilities in image generation and image processing. This paper has described its architecture and capabilities in the realm of image generation: 20 to 60 Hz animations of full-screen, fully-textured, antialiased scenes. (Figures 6 and 7.) The image processing capabilities of the architecture have not been described at all; they include convolution, color space conversion, table lookup, histogramming, and a variety of warping and mapping operations using the texture mapping hardware. Future developments will investigate additional advanced rendering features, while continually reducing the cost of high-performance, high-quality graphics.

Acknowledgments

It was a privilege to be a part of the team that created RealityEngine. While many team members made important contributions to the design, I especially acknowledge Mark Leather for developing the multisample antialiasing technique that was eventually adopted, and for designing a remarkable integrated circuit (the Image Engine) that implemented his design. Also, special thanks to Doug Voorhies, who read and carefully marked up several drafts of this paper, Finally, thanks to John Montrym, Dan Baum, Rolf van Widenfelt, and the anonymous reviewers for their clarifications and insights.

Appendix 1: Measured Performance

The two most significant performance categories are transform rate: the number of primitives per second that can be processed by the Geometry Engines, and fill rate: the number of fragments per second that can be generated and merged into the framebuffer. Running in third-generation mode (lighting, smooth shading, depth buffering, texturing and multisample antialiasing) a 12 Geometry Engine system can process 1.5 million points, 0.7 million connected lines, and 1.0 million connected triangles per second. In second-generation mode (lighting, smooth shading, and depth buffering) the same system can process 2.0 million points, 1.3 million connected lines, and 1.2 million connected triangles per second. Measured third-generation fill rates for 2 and 4 raster board systems are 120 and 240 million fragments per second. Measured second-generation fill rates for 1, 2, and 4 raster board systems are 85, 180, and 360 million fragments per second. The third-generation fill rate numbers are somewhat dependent on rendering order, and are therefore chosen as averages over a range of actual performances.

Appendix 2: Bandwidth and other Statistics

Triangle Bus, fragment transfer path, and Image Engine to framebuffer memory bandwidths are in roughly the ratios of 1:10:20. Specific numbers for the typical two raster board configuration are 240 Mbyte/sec on the Triangle Bus, 3,200 Mbyte/sec aggregate on the 160 Fragment Generator to Image Engine busses, and 6,400 Mbyte/sec aggregate on the 160 Image Engine to framebuffer connections.

Because the 6,400 Mbyte/sec framebuffer bandwidth is so much larger than the bandwidth required to refresh a monitor (roughly 800 Mbyte/sec at $1280 \times 1024 \times 76$Hz) we implement the framebuffer memory with dynamic RAM rather than video RAM, accepting the 12 percent fill rate degradation in favor of the lower cost of commodity memory. Geometry Engine memory and texture memory are also implemented with commodity, 16-bit data path dynamic RAM. Total dynamic memory in the maximally configured system is just over 1/2 Gigabyte.

References

[1] AKELEY, KURT AND TOM JERMOLUK. High-Performance Polygon Rendering. In *Proceedings of SIGGRAPH '88* (August 1988), pp. 239–246.

[2] CARPENTER, LOREN. The A-buffer, An Antialiased Hidden Surface Method. In *Proceedings of SIGGRAPH '84* (July 1984), pp. 103–108.

[3] HAEBERLI, PAUL AND KURT AKELEY. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Proceedings of SIGGRAPH '90* (August 1990), pp. 309–318.

[4] KIRK, DAVID AND DOUGLAS VOORHIES. The Rendering Architecture of the DN10000VS. In *Proceedings of SIGGRAPH '90* (August 1990), pp. 299–308.

[5] MOLNAR, STEVEN. *Image-Composition Architectures for Real-Time Image Generation*. University of North Carolina at Chapel Hill, Chapel Hill, NC, 1991.

[6] MOLNAR, STEVEN, JOHN EYLES AND JOHN POULTON. PixelFlow: High-Speed Rendering Using Image Composition. In *Proceedings of SIGGRAPH '92* (July 1992), pp. 231–240.

[7] NEIDER, JACQUELINE, MASON WOO AND TOM DAVIS. *OpenGL Programming Guide*. Addison Wesley, 1993.

[8] OPENGL ARCHITECTURE REVIEW BOARD. *OpenGL Reference Manual*. Addison Wesley, 1992.

[9] PINEDA, JUAN. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of SIGGRAPH '88* (August 1988), pp. 17–20.

[10] SCHILLING, ANDREAS. A New Simple and Efficient Antialiasing with Subpixel Masks. In *Proceedings of SIGGRAPH '91* (July 1991), pp. 133–141.

[11] SILICON GRAPHICS, INC. *Iris 4DGT Technical Report*. Silicon Graphics, Inc., Mountain View, CA, 1988.

[12] SILICON GRAPHICS, INC. *Technical Report - Power Series*. Silicon Graphics, Inc., Mountain View, CA, 1990.

[13] WILLIAMS, LANCE. Pyramidal Parametrics. In *Proceedings of SIGGRAPH '83* (July 1983), pp. 1–11.