High-Performance Polygon Rendering

Kurt Akeley
Tom Jermoluk

Silicon Graphics, Inc.
2011 North Shoreline Boulevard
Mountain View, CA 94039-7311

ABSTRACT

This paper describes a system architecture for realtime display of shaded polygons. Performance of 100,000 lighted, 4-sided polygons per second is achieved. Vectors and points draw at the rate of 400,000 per second. High-speed pan and zoom, alpha blending, realtime video input, and antialiased lines are supported. The architecture heavily leverages parallelism in several forms: pipeline, vector, and array processing. It is unique in providing efficient and balanced graphics that support interactive design and manipulation of solid models. After an overview of algorithms and computational requirements, we describe the details of the implementation. Finally, the unique features enabled by the architecture are highlighted.

CR Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structures]: Design Styles - Parallel, Pipeline; C.1.2 [Processor Architectures]: Multiprocessors - Parallel processors, Pipeline processors; I.3.1 [Computer Graphics]: Hardware Architecture - Raster Display Devices.

Additional Key Words and Phrases: Graphics Systems.

1. Introduction

Traditional 3D graphics workstations have concentrated on the hardware that transforms points and lines from object-space to screen-space. As users' needs for display of realistic solid objects have increased, demands on graphics architectures have changed significantly. New challenges include increases in transformation rate, incorporation of realtime illumination calculations, and dramatic increases in pixel fill rates.

Contemporary workstations have attempted to satisfy these new demands using a variety of architectures. Swanson and Thayer [11] describe a system which incorporates parallel pixel processors in its raster subsystem. A parallel geometry system is described by Torborg [12]. The lineage of these and other contemporary graphics systems can be traced to works by Clark [3], Clark and Hannah [2], and eventually to Fuchs and Johnson [5].

Our specific goals for the performance and capability of the new architecture were:

- *100,000 polygons per second.* The polygons are RGB, 4-sided, 10x10 pixels, lighted, Gouraud shaded [6], arbitrarily rotated, Z-buffered, clip tested, projected, and rendered into one of multiple, possibly overlapping, windows at an instantaneous rate of 100,000 per second.
- *10 frames per second.* Realtime systems must be able to draw and display at rates exceeding 10 Hz.
- *Window support.* User productivity is optimized by a system that supports multiple fully independent windows.

First we describe our view of the general algorithmic problem of rendering 100,000 polygons per second, paying particular attention to the computational requirements of each step of the process. Then we describe a novel architecture to achieve the goals. Finally we present several unique features that were realized from the implementation.

2. Problem Description

The floating-point performance and pixel fill rates required for interactive display of solids are both exceptional. By examining each component of the problem we show that at least 40 million floating-point operations per second, and fill rates exceeding 10 million pixels per second, are required to render 100,000 polygons per second. The problem components are:

1. Transfer vertex data from memory to the graphics subsystem.
2. Transform vertex coordinates.
3. Transform vertex normals.
4. Light each vertex.
5. Clip each vertex.
6. Project each vertex.
7. Map vertex coordinates to the screen.
8. Fill the resulting screen-space polygons, interpolating color and depth.

9. Clip all drawing against the visible window boundaries.
10. Be prepared to quickly swap drawing contexts at any time.
11. Continually scan the frame buffer at screen refresh rates, interpreting each pixel appropriately as a function of the process to which it belongs.

2.1 Transfer Data

Achieving our goal of 100,000 4-vertex polygons per second requires that 400,000 vertex specifications per second be transferred to the graphics subsystem. A lighted vertex specification includes 6 floating-point values: 3 *(x,y,z)* specify the vertex position, and 3 *(nx,ny,nz)* specify the vertex normal direction. The required data rate is therefore:

$$400{,}000\ (xform/sec) \times 6\ (words/xform) \times 4\ (bytes/word) \approx 10\ Mbytes/sec$$

2.2 Transform Vertexes

Homogeneous 3D vertexes include 4 components, and are transformed by multiplication by a 4x4 matrix. This vector operation requires 16 floating-point multiplications and 12 floating-point additions. The computation rate is therefore:

$$400{,}000\ (xform/sec) \times (16 + 12)\ (flop/xform) \approx 11\ Mflops$$

If it is known that the w component of the vertex to be transformed is 1.0 (a common case with 3D homogeneous data) then 4 multiplications can be eliminated, resulting in a computation rate of:

$$400{,}000 \times (12 + 12) \approx 9.5\ Mflops$$

2.3 Transform Normals

Plane equations are properly transformed by the inverse-transpose of the vertex matrix. This transformation has the same expense as the full vertex transformation. It is unnecessary, however, for lighting calculations.

Vertex normals are plane equations stripped of distance information. Because normals are not position sensitive, translation information in the vertex matrix need not be included in a normal matrix. Also, because vertex normals are normalized to unit length before they are used, uniform scale information need not be applied to a normal matrix. Thus only rotations and non-uniform scales of the vertex matrix are applied, by inverse-transpose, to the normal matrix.

A 3 by 3 normal matrix supports all the required information, and is used to transform the 3-component lighting normals. This transformation requires 9 floating-point multiplications and 6 floating-point additions, for a total computation rate of:

$$400{,}000 \times (9 + 6) \approx 6\ Mflops$$

As was indicated above, normals must be unit length before they are used in lighting calculations. Although it will in some cases be possible to avoid explicit normalization,[1] we compute the expense of this operation. Sum of squares requires 3 floating-point multiplications and 2 floating-point additions. We conservatively estimate the cost of reciprocal square root calculation to be twice that of simple reciprocal calculation: 8 floating-point operations. Finally, each of the three components is multiplied by the newly computed factor. The approximate required computation rate for normalization is:

$$400{,}000 \times (3 + 2 + 16 + 3) \approx 9.5\ Mflops$$

2.4 Light Vertexes

Because the target polygons are Gouraud shaded, lighting calculations are required only at the vertexes of polygons, never in their interiors. Transformed vertex and normal information, current light positions and colors, surface properties, and specific lighting-model properties for each vertex are resolved to a single RGB triple. These triples will be interpolated across the interiors of projected polygons.

1. *Normals specified with unit length, no non-uniform scales.*

The lighting model that we chose for our performance requirement estimates is:

- A single light - infinitely distant
- A viewer - infinitely distant
- Ambient, diffuse, and specular reflection components

The equation

$$C_{object} = C_{ambient} + C_{diffuse}C_{light}(N{\cdot}L) + C_{specular}C_{light}(N{\cdot}H)^n$$

is executed 3 times, once each for red, green, and blue (all C's are RGB vectors). Dot products are evaluated only once. Including tests for overflow, each optimized lighting computation requires 12 floating-point multiplications, 10 floating-point additions, 5 floating-point comparisons, and a table lookup. The total compute power required is:

$$400{,}000 \times (12 + 10 + 5 + 1) \approx 11\ Mflops$$

2.5 Clip

Polygons are correctly clipped in all cases using the Sutherland-Hodgman Algorithm [10]. This algorithm requires one floating-point compare per clipping plane, 6 compares per vertex in a 3D system. Additional floating-point operations are required only in the event of actual clipping, i.e. infrequently. In either case the algorithm execution time is dominated by data movement and branching, not by floating-point operations.

Using proper optimization to avoid unnecessary clipping, the floating-point demands for clipping are:

$$400{,}000 \times 6 \approx 2.5\ Mflops$$

2.6 Project

Each vertex is projected in homogeneous space by division of its x, y, and z components by its w component. This is accomplished most efficiently by first computing $1/w$, then multiplying each of x, y, and z by this factor. We asserted previously that computing a reciprocal requires 8 floating-point operations. Thus perspective division requires a total of 11 floating-point operations, at an aggregate rate of:

$$400{,}000 \times (8 + 3) \approx 4.5\ Mflops$$

2.7 Viewport and Fix

Projected vertexes are mapped to screen coordinates with a simple affine calculation. Each of the three vertex components is scaled by an independent scale factor, offset by an independent offset, and converted to integer screen coordinates. The total floating-point calculation rate is:

$$400{,}000 \times (3 + 3 + 3) \approx 3.5\ Mflops$$

The total floating-point performance required to convert object-space coordinate vertexes and normals to screen-space coordinates and colors is:

Operation	*Mflops*
Vertex Transformation	9.5
Normal Transformation	6
Normal Normalization	9.5
Lighting	11
Clipping	2.5
Projection	4.5
Viewport	3.5
Total	46.5

2.8 Fill and Smooth Shade Screen Space Polygons

Screen-space vertexes, now interpreted as polygon vertexes, are used to specify the boundaries of frame buffer regions to be smooth shaded. Although the details of this shade/fill operation are implementation dependent, we can safely describe some of the problems that will be encountered. They are:

- *Test for convexity.* A substantially more complex algorithm is required to fill concave polygons.
- *Decompose to trapezoids.* It will almost always be desirable to reduce the full polygons to collections of screen-aligned trapezoids, whose

parallel edges are in the direction of preferred fill in the frame buffer.

- *Calculate slopes.* Slopes for all parameters to be interpolated must be computed. Substantial integer precision must be maintained if interpolation errors are to be avoided.
- *Write to the frame buffer.* A huge memory bandwidth is required to fill polygons at the rate of 100,000 per second. Our benchmark 10 pixel by 10 pixel polygons fill at 10 Million pixels per second. A substantially higher fill rate is desirable for larger polygons. Screen clear, really just filling a large, screen-aligned polygon, demands the highest fill rate.

2.9 Clip all Drawing Against Visible Window Boundaries

The clipping and viewport operations described above will limit polygon filling to a screen-aligned rectangular region, or *window.* They are not sufficient, however, if the window is non-rectangular, either because it is obscured by another window, or because it wasn't rectangular to begin with. In some systems the problem of obscured windows is deferred by always rendering into rectangular regions, then assembling the final screen image on the fly. While attractive in many respects, this solution does nothing to solve the fundamental problem of non-rectangular windows, and is expensive in terms of memory consumption.

2.10 Change Contexts

The entire graphics system must be prepared, at any moment, to save the drawing state of the current context and restore the state of a previously interrupted context. This operation must happen quickly in most cases, but must also be able to support a huge number, perhaps hundreds, of independent contexts. Support for a *working set* of processes is therefore desirable.

2.11 Scan Frame Buffer

Although the bandwidth required to scan a high-resolution frame buffer at 60 Hertz is enormous (pixel rates of 110 to 125 MHz are standard today) the availability of inexpensive Video RAMs allows this bandwidth to be supported with little engineering effort. We don't emphasize this problem here. Rather, we concentrate on the issue of multiple windows as it pertains to frame buffer output.

It is desirable that imaging to separate windows be as independent as possible. Most important, perhaps, is that windows be able to select and alter their buffer modes independently. Single and double buffer images must coexist, and double buffer images must swap buffers independently. Further, if pixels can be interpreted in different ways, it is important that the interpretation in each window also be independent.

An important exception is color mapping, the process of interpreting pixel values as indexes into a table of RGB triples. While it would seem that each process should have its own table, it is sometimes desirable to share table entries between processes. Thus a single table, large enough to supply separate areas to processes that desire independence, yet shared by all processes, is an appropriate solution.

3. Architectural Solution

Our graphics subsystem is a part of a complete workstation whose host processor comprises multiple RISC-based CPUs. The CPUs and the graphics interface share a high-speed 64-bit synchronous bus of proprietary design. While the primary design goal of the bus was multiprocessor support, it includes special support for data transfer to the graphics subsystem.

The graphics system itself is partitioned into four pipelined subsystems. In order of data flow these are:

1. *Geometry subsystem.* Supports all floating-point operations. Transforms data from object-space coordinates to screen-space coordinates.
2. *Scan conversion subsystem.* Interprets screen-space coordinate vertexes as points, lines, and polygons, and generates appropriate fill instructions. Interpolates color and depth data across lines and polygons.
3. *Raster subsystem.* Maintains a 1280 by 1024 frame buffer of 96-bit pixels. Executes pixel algorithms such as replace, depth conditional replace (Z-buffer), and blend.
4. *Display subsystem.* Continually scans the frame buffer to supply video data to the monitor. Each pixel is interpreted individually as a function of the window to which it belongs.

Each of these four subsystems, as well as the host interface, is described in detail below.

3.1 Data Transfer

As we have seen, the graphics subsystem consumes data at roughly 10 Mbytes per second. The 64 Mbyte per second synchronous bus that connects the host processors to the graphics subsystem is able to handle this load. Thus large blocks of geometric data can be transferred across this bus to the graphics subsystem. Such transfers, however, are not consistent with the programming model desired for the machine.

The target graphics library includes commands such as *vertex(x,y,z)* and *normal(nx,ny,nz).* Each command is implemented as a subroutine in the language of the calling application program. Each specifies data from an arbitrary structure. Graphics programs using commands such as these can operate directly from application data bases. Programs need not be 'compiled' into display lists, and therefore can traverse data under complete application control. Finally, because all traversal code and data reside in main memory, there is essentially no limit to the size of either.

We seek a solution that retains the desired properties of *immediate-mode* graphics, and supports extremely high-performance graphics. Our answer is to create new commands that operate on 2, 3, and 4-component vectors, rather than on scalar values. Thus *vertex(x,y,z)* becomes *vertex(xyz)*, where *xyz* is the address of three adjacent floating-point values. Hardware support allows each host processor to make a single memory reference when dealing with such a vector. The memory and synchronous bus deliver the data to the graphics subsystem in a single burst, making optimum use of the available bus bandwidth. Vectors that straddle page boundaries are detected and handled automatically.

By providing support for program controlled data transfer to the graphics subsystem at rates far in excess of 10 Mbytes per second, we allow for both future increases in graphics performance, and for desirable inefficiencies in program execution. One such inefficiency is shared graphics libraries. While such libraries exact a performance penalty at each call, they greatly reduce code size, both on disk and in memory, and also support object code compatibility between machines with different graphics subsystems.

3.2 Geometry Subsystem

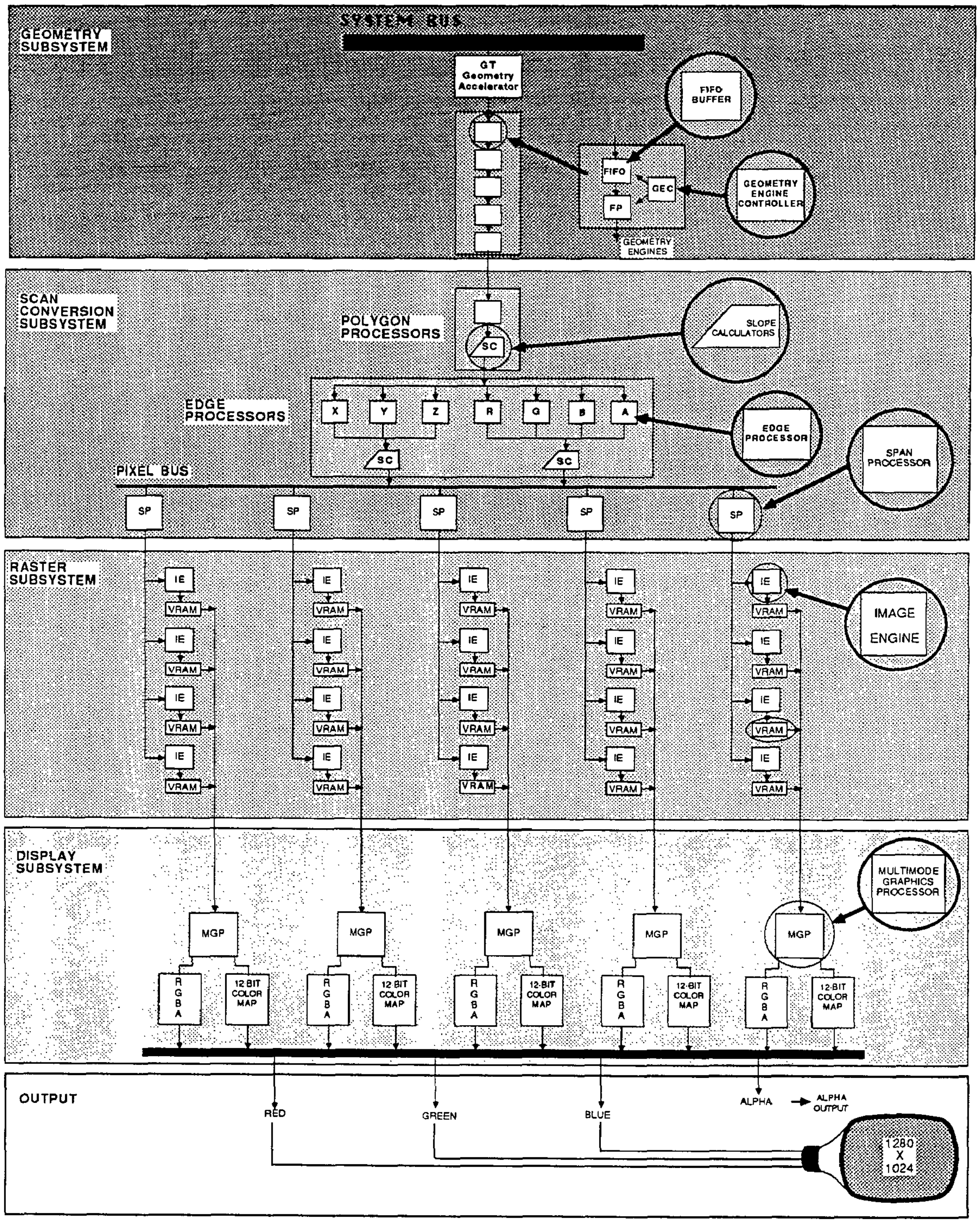
The geometry subsystem comprises a single conversion and FIFO module, followed by 5 identical floating-point processors (Geometry Engines®). These 6 processors are organized as a single pipeline. Each executes a specific subset of the rendering algorithm, minimizing microcode space requirements.

3.2.1 Conversion and FIFO Module This module accepts coordinate data in 4 formats: 16-bit integer, 24-bit integer, 32-bit IEEE floating-point, and 64-bit IEEE double precision floating-point. Color data are accepted as packed integers as well as in the coordinate formats. All data are converted to 32-bit IEEE floating-point format for consumption by the Geometry Engines. Hardware format conversion supports direct transfer of data from user structures to the graphics subsystem without performance penalty.

A 512-word FIFO precedes the conversion module. The hardware interrupts when a *high water* mark is passed, allowing user programs to transfer data to the graphics subsystem without concern for flow control. On interrupt, the operating system blocks the user program until the FIFO empties past a *low water* mark. Thus transfer rate is adversely affected by flow control protocol only when it has already exceeded the ability of the graphics system to accept data.

3.2.2 Geometry Engine Each of the five Geometry Engines is an identical module capable of 20 million single-precision floating-point operations per second (Mflops). Each includes separate high-speed microcode and data memories. The engines accept and output commands accompanied by up to 4 data words. Command interpretation is accomplished by program jump, followed by normal program counter operations. Hardware support in each engine includes:

FIGURE 1


SYSTEM BUS
GEOMETRY SUBSYSTEM
GT Geometry Accelerator
FIFO BUFFER
FIFO
GEC
FP
GEOMETRY ENGINE CONTROLLER
GEOMETRY ENGINES
SCAN CONVERSION SUBSYSTEM
POLYGON PROCESSORS
SC
SLOPE CALCULATORS
EDGE PROCESSORS
X
Y
Z
R
G
B
A
EDGE PROCESSOR
PIXEL BUS
SP
SPAN PROCESSOR
RASTER SUBSYSTEM
IE
VRAM
IMAGE ENGINE
DISPLAY SUBSYSTEM
MGP
MULTIMODE GRAPHICS PROCESSOR
R G B A
12 BIT COLOR MAP
OUTPUT
RED
GREEN
BLUE
ALPHA
ALPHA OUTPUT
1280 X 1024

- *Flow control.* Microcode ignores the issues of command availability, and of the subsequent engine's ability to accept commands. Hardware blocking is provided for both cases.
- *FIFO input buffer.* Each engine includes 4 command buffers at its input. Buffers store up to 4 data words and a command address.
- *Pipeline support.* Although the Geometry Engine has 6 internal pipeline stages, individual microcode instructions specify complete operations, including data sources, operations to be performed, and data destinations.
- *Concurrent command execution.* The flow control and branch hardware support concurrent command (as well as instruction) execution.

The above, taken together, allow each engine to operate efficiently. The pipeline sustains the required 40-50 Mflops floating-point rate with 5 modules whose aggregate peak rate is 100 Mflops, an efficiency approaching 50 percent. This efficiency is not achieved in more general purpose vector processors, especially when the vectors are of 3 to 4 element length.

Graphics tasks are distributed among the 5 Geometry Engines as follows:

1. Matrix and normal transformation. Matrix and normal stacks. Normal normalization.
2. Lighting calculations.
3. Clip testing.
4. Perspective division. Clipping (when required).
5. Viewport transformation. Color clamping to a maximum value. Depthcue calculations.

3.3 Scan Conversion Subsystem

Screen coordinates sent from the geometry subsystem to the Scan Conversion Subsystem specify the vertexes of points, lines, and polygons. The Scan Conversion Subsystem performs the calculations required to reduce vertex data to individual pixels. Each pixel is assigned an x, y, and z coordinate and an r, g, b, and α color value. Color and z data are always interpolated linearly between vertexes and between the edges of polygons.

The task of scan-converting polygons is partitioned into three separate processors within the Scan Conversion Subsystem. The first two of these processors, the Polygon and Edge processors, use a pseudo floating-point representation to maintain coordinate integrity when calculating slopes. In addition, the y coordinates of polygon edges are computed to 1/8 pixel tolerance. All depth and color component iterations are first corrected to the nearest pixel center, then iterated in full-pixel steps. As a result, iterated color and depth values remain planar across polygonal surfaces, and subsequent Z-buffer calculations result in clean intersections.

Vertex data are not passed directly from the geometry subsystem to the Scan Conversion Subsystem, but rather are accumulated in one of two 256-vertex buffers. Vertex representations in this buffer are always the same, regardless of the operating mode of the system. Hardware on both the Geometry and Scan sides of the buffer is optimized to operate on these vertexes. Thus, the Polygon Processor receives entire polygons, rather than individual vertexes. It operates on vertexes directly from this buffer, avoiding unnecessary copying and interpretation.

The Polygon Processor both sorts vertexes from left to right and checks for convexity in one simple, pipelined operation. The sorted vertexes are decomposed into trapezoids. Slopes of y, z, r, g, b, and α are computed relative to delta x. Coordinates and slopes for each edge are passed to the Edge Processor. Trapezoid edges are handles at the rate of 1 per microsecond.

The Edge Processor iterates along the top and bottom edges of the trapezoid, generating at each iteration the top and bottom coordinates of a single *span*.[2] Spans are always iterated bottom to top. Therefore hardware

SCAN CONVERSION SUBSYSTEM

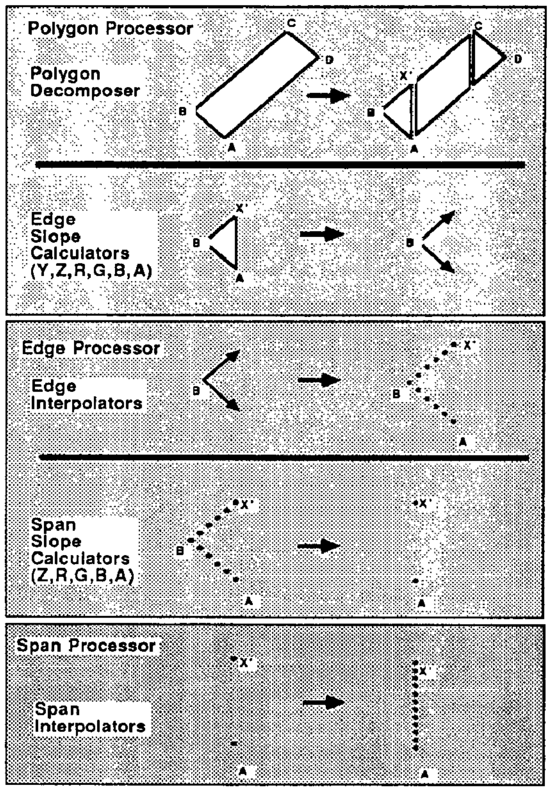


FIGURE 2

is provided to swap span ends as necessary, both avoiding a complex test at trapezoid decomposition time, and correctly handling *bow-tie* polygons, which occur frequently at surface silhouettes.

The color, y, and z edge components comprise 2 vectors which are iterated in parallel by multiple, proprietary engines. Spans are generated at the rate of 2 per microsecond.

The y components of span endpoints are computed to 1/8 pixel accuracy. Color and depth slopes are computed using delta y to this accuracy. This slope is then used to iterate to the nearest pixel center. The final span definition comprises the corrected initial color and depth values, the color and depth slopes, the integer x and y values of the bottom pixel, and the span length.

The Edge Processor delivers each span to one of five Span Processors. Each Span Processor manages every fifth column of pixels in the frame buffer. Since spans generated from a single polygon are always adjacent, the span processing load is evenly distributed across the five Span Processors. Each Span Processor iterates through its span using the initial and slope values provided, treating color and z span components as a vector. Pixel specifications are generated at the rate of 8 per microsecond. Thus the aggregate fill rate of the 5 Span Processors is 40 million pixels per second (Z-buffered).

3.4 Raster Subsystem

The Raster Subsystem contains 20 Image Engines TM, each of which is an independent state machine that controls 1/20th of the frame buffer memory. Groups of 4 Image Engines are driven by each Span Processor. The array of Image Engines tiles the frame buffer in a 5-wide, 4-high, pattern.

Bitplane memory is organized into 5 banks, comprising a total of 96 bits per pixel. The banks are arranged as follows:

- *Image banks.* Two banks of 32 bits each, organized as 8 bits each of red, green, blue, and alpha data.

2. *We refer to vertical lines of pixels as* spans, *horizontal lines as* scans.

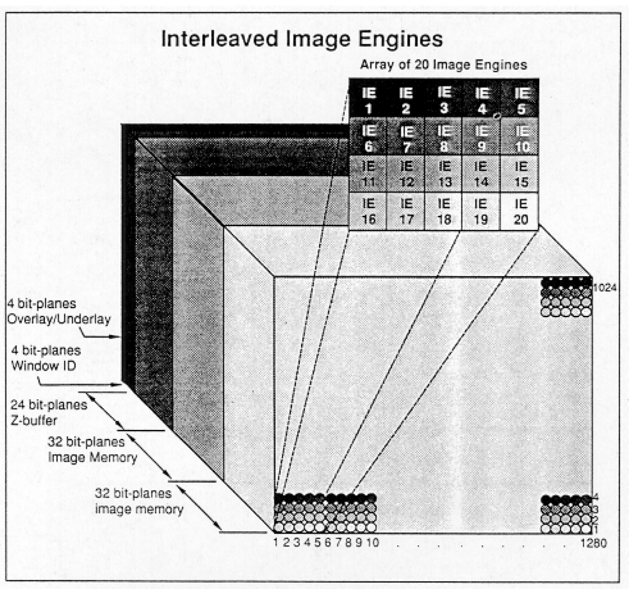


FIGURE 3

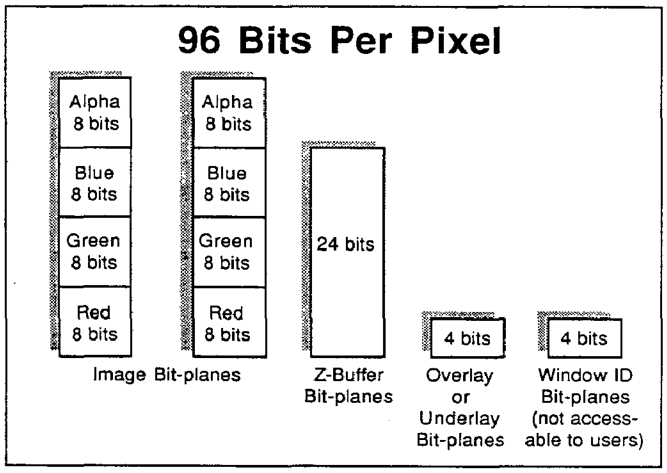


FIGURE 4

- *Depth bank.* One bank of 24 bits. Stores 24-bit integer depth information when used in conjunction with the Image Engine Z-buffer pixel algorithm. It is also available for image data.
- *Overlay bank.* One bank of 4 bits. Two bits are reserved for the window manager.
- *Window ID bank.* One bank of 4 bits, used by the window manager to tag pixels based on the drawing process to which they belong.

Image Engines operate as specialized memory controllers, supporting video RAM refresh, display refresh, and a handful of pixel access algorithms. These algorithms include:

- *Replace.* Replace the destination color with the source color.
- *Z-buffer.* Compare the source and destination *z* values. If the test passes[3] replace the destination color and *z* with the source color and *z*,
- *Z-buffer blend.* Like Z-buffer, but replace the destination color with a linear combination of the source and destination colors.
- *High-speed clear.* Simple replace available only for large, screen-aligned rectangles.

Although the Image Engines are simple machines, their parallel operation and multiple algorithms result in extremely powerful pixel fill operation. Their aggregate performance for the various pixel algorithms is:

Pixel Algorithm	Fill Rate Mpixel/sec
Replace	80
Z-buffer	40
Z-buffer with blending	10
High-speed clear	160

3.5 Display Subsystem

The Display Subsystem receives pixel data from the frame buffer, interprets it, and routes the resulting red, green, blue, and alpha data to the Digital-to-Analog converters for display. Five Multimode Graphics Processors (MGPs) operate in parallel, one assigned to the pixels controlled by each Span Processor. These MGPs receive all 64 image bank bits, the 4 auxiliary bank bits, and the 4 window ID bits for each pixel. They interpret the image and auxiliary bits as a function of the window ID bits, using an internal 16-entry table.

3.6 Context Switching

The graphics subsystem is designed to support context switching with minimal overhead. Because significant quantities of state are accumulated in each of the 5 Geometry Engines, each maintains complete context for 16 independent processes in its local data memory. The Geometry Engines are also able to dump and restore context to and from a host processor, allowing more than 16 processes to share the hardware. Thus a *working set* of up to 16 processes is supported, with essentially no limit to the total number of processes.

Because the Edge, Span, and Image Processors are unable to return state information, the few states stored in these processors are shadowed by the Polygon Processor. The Polygon Processor state, including shadow state, is minimal, and is therefore maintained by a host CPU.

4. Performance

We achieved our polygon performance and quality objectives, including operation in a window environment. The subsystems used to achieve this goal are carefully balanced in performance, resulting in a cost effective solution. Vertexes are transformed and lighted at the rate of 400,000 per second, just matching the desired rate of 100,000 4-vertex polygons per second. The Polygon Processor sorts the vertexes of 4-vertex polygons, and computes slopes for 4 edges, in just under 10 microseconds, again just meeting the required performance. The edge processor iterates along edges and generates spans at the rate of 2 spans per microsecond, slightly faster than required to generate 14 spans in 10 microseconds. Spans are iterated and pixels generated at the aggregate rate of 40 million per second, four times the rate needed to meet the performance objective, but invaluable for smoothing the performance transition between small and large polygons.

Some performance notes for various operations follow:

- *Polygons.* Because the system is balanced to render lighted, smooth shaded polygons, there is no performance benefit for not using these features. Thus flat shaded, smooth shaded but unlighted, and lighted 4-side polygons all render at the target rate of 100,000 per second. Small triangles render at 120,000 per second, again regardless of mode. Large polygons render at rates limited by the 40 million pixel per second fill rate.
- *Vectors.* The draw rate of short vectors is transformation limited, resulting in 400,000 connected vectors per second, or 200,000 unconnected vectors per second. Long vector rates are limited by the 8 million pixel per second fill rate (16 million with the Z-buffer disabled).
- *Window clear.* The performance of even moderately complex animations can be limited by the time required to clear the window. The special 160 million pixel per second fill rate, available only for window clear, allows a screen-size window to be cleared in 8.2

3. *Any combination of greater than, equal to, and less than can be specified.*

milliseconds. Thus full screen animations running at 10 Hz lose only 8% of their draw time to screen clear. Even 30 Hz animations lose only 25% of their draw time.

- *Pixel access.* An important ancillary function of 3D graphics is high-speed host access to the frame buffer. This is useful for image display and storage, image convolution, paint programs, and many other applications. The new raster architecture supports host read and write rates of 5 million pixels per second.

5. Special Features

Our design goal achieved, let us now consider some other features of the new graphics architecture.

5.1 Pan and Zoom

Typical raster systems handle pan and zoom as a display process by altering the fetching of data for the monitor. Frame buffer scan lines are output multiple times to achieve vertical zoom, and are output at reduced rates to achieve horizontal zoom. Initial pixel addresses are altered to achieve horizontal and vertical pan. In all cases the video data rate is either reduced or unaffected. Thus, while the implementation is complex, it makes no performance demands on the hardware.

This typical pan and zoom implementation, however, has some undesirable properties:

- It either operates on the entire screen, which is unacceptable in a window environment, or it becomes unmanageably complex.
- The effort and cost expended solving pan and zoom in this manner do not contribute to the machine in any other way.

The second point is of particular interest. We prefer solutions that have a synergistic effect on the performance of the entire machine.

Recall the bus that connects the Edge Processor to the five Span Processors. This *pixelbus* transmits span definitions during polygon fill, but is also designed to support pixel transfers during line fill. (The Edge Processor fills lines as though each was a single trapezoid edge, generating pixels at the rate of 8 (Z-buffered) or 16 million per second.)

The addition of a small pixel cache on the *pixelbus* allows pixels to be read and written in blocks large enough to achieve performance roughly equal to the peak *pixelbus* rates:

Operation	Mpixel/sec
read	5.3
write	16.0

Because write cycles greatly outnumber read cycles when the zoom factor is large, fill rates approach the higher write rate as the zoom factor is increased. The fill rates for a variety of zoom factors are:

Zoom Factor	Mpixel/sec
1	4.0
2	9.1
3	12.0
4	13.5
8	15.3

With this performance it is possible to zoom 1/4 of the screen by a factor of 2 at the rate of 7 frames per second. Smaller areas, common in window-capable systems, easily zoom at 30 frames per second. Because the effects of pan and zoom are limited to a single window, or to multiple windows with independent factors if desired, the full screen with all its windows remains a useful resource.

High-speed pixel copy leverages *pixelbus* throughput, which was also required for line drawing. By emphasizing high-speed pixel read and write, we improve the performance of transfers between host memory and the frame buffer, and also support real-time video input.

5.2 Window ID Masking

Each pixel in the frame buffer includes a 4-bit ID field that is unique to the process that controls that pixel. Previous architectures [9] have used this per-pixel window ID field to control interpretation of pixel contents at display time.[4] Such an ID, read out and interpreted as a part of the display process, easily supports independent buffer mode specification on a per-pixel basis. Windows can independently select single or double buffer operation, and double buffer windows can swap buffers independently. Colorindex or RGB operation is also selected independently on a per-window basis. Thus, while the notion of a pixel ID is not new, its use as a drawing mask is.

The new graphics hardware includes pipelined hardware that tests the ID of each pixel against the ID of the current drawing process. If the test fails, the draw operation is aborted with no change to the frame buffer contents. Otherwise, the drawing operation is completed in the currently specified manner. Because the compare operation is truly pipelined, there are no drawing order requirements imposed by the test. All drawing operations to the frame buffer, including lines, are ID masked with no performance penalty.[5]

ID masking supports both partially obscured windows and non-rectangular windows (such as round clocks or templates) in a simple and consistent manner. It imposes no constraints on window size or shape, and never results in loss of performance.

5.3 Realtime Video

The new graphics architecture is capable of capturing both NTSC and PAL images in real time. These images are transferred to an arbitrary window on the screen via the *pixelbus* at the rate of 16 million pixels per second. Once in the frame buffer, they can be operated on just like images from any other source. Frame grab rate is controlled by the drawing program, allowing the simple program loop:

```
while (TRUE)
        grab a frame
        modify the image
        swap buffers
```

to operate as expected. Multiple buffers within the grabbing hardware insure that no frames are missed as long as the sum of the *grab a frame* period and the *modify the image* period does not exceed 1/30 of a second. The resulting NTSC or PAL image can be output in the same video format, allowing the hardware to act as a realtime video filter. Genlock and the alpha channel output allow additional video sources to be merged in a useful manner.

5.4 Alpha Blending

Each of the twenty Image Engines includes both ALU and microcode support for an alpha blending algorithm. This blending algorithm, used while operating in RGB mode, causes the destination pixel values to be a linear combination of the previous destination values and the new source values.

$$C_{dst} = F_{dst}C_{dst} + F_{src}C_{src}$$

$$F \in 0,\ 1,\ alpha,\ 1{-}alpha,\ C_{src},\ 1{-}C_{src},\ C_{dst},\ 1{-}C_{dst}$$

The algorithm operates identically on red, green, blue, and alpha color components, each of which is stored as an 8-bit value in the frame buffer. Algorithm options are specified in table format. All of the operations described by Porter and Duff [8], as well as others, are available.

The frame buffer provides complete support for image compositing, including output of the alpha channel for external image merging. In addition, such a blending function at the tail end of a geometric graphics system provides capabilities well beyond traditional image compositing. Specifically, because blending is supported in conjunction with Z-buffer operation, geometrically specified *solids* can be blended to simulate the effects of transparency. With some attention to the order in which image components are specified, useful engineering images can be created.

5.5 Antialiased Lines

While the problem of realtime antialiasing of geometric images (as discussed by Crow [4]) has yet to be solved by a workstation, it has become possible to solve limited subsets of this problem. Our specific

4. *Silicon Graphics has applied for patent protection for this technology.*
5. *Silicon Graphics has applied for patent protection for this technology.*

implementation solves the problem of realtime rendering of antialiased lines against a constant color background. It is related to the algorithm described by Gupta and Sproull [7].

We require subpixel position information to properly antialias a line. This information is unavailable in graphics systems that rely on the Bresenham [1] algorithm for line iteration. It is available in a system that iterates using a digital differential analyzer (DDA). The DDA approach has been avoided in the past because of the division required. Since hardware has been provided to compute both color and depth slopes, the cost of computing line slopes, and thus of DDA iteration, has become insignificant.

Our antialias line algorithm forces line end points to pixel-centered positions, then uses sub-pixel information to smooth interior pixels. Each line is drawn twice, the second time offset one pixel position in the direction opposite the major line direction. During each pass the 3 most significant fraction bits of y, if the line is x major, or x, if the line is y major, as well as the pass (first or second) are used to drive a table lookup of pixel coverage information (see Picture 1). The table output is a 4-bit colorindex, which is concatenated with the 8 most significant bits of the current drawing colorindex to form the new pixel value. Thus constant color lines access 1 of 16 colormap locations as a function of pixel coverage. When appropriate values are loaded into the colormap, attractive antialiased lines result.

Of course the current colorindex, and thus the upper 8 bits of that index, can be iterated while the antialiased line is being drawn. When this iteration is controlled as a function of depth, and appropriately scaled ramps of 16 entries are created in the colormap, depthcued antialiased lines are drawn. Antialiased lines of different colors can be drawn by simply changing the current colorindex between lines, again with appropriate ramp specifications.

Line intersections are handled in one of three ways:

- *Depth Buffered.* Z-buffer conditional pixel fill can be used to force the nearest (or farthest) line's color to dominate pixels where lines intersect.
- *Color Buffered.* The same Z-buffer hardware can be retargeted to branch on colorindex, rather than depth, information. This insures that the intensity of a pixel is never diminished. This algorithm works well with single-color images that include many intersections.
- *Painter's algorithm.* Each pixel takes the last value that is written to it.

6. *Summary*

We have presented a parallel architecture for high speed polygon rendering. The system achieved its goal of 100,000 polygons per second through an efficient and balanced implementation of a novel architecture. In addition, several features new to workstation graphics were introduced. The implementation of the graphics subsystem consists of a 5-board set utilizing 50 copies of 7 proprietary chips and 7 additional commercial microprocessors.

Benchmark testing of a completed system immediately prior to publication yielded the following results:

- *101,000 quadrilaterals per second.* 100 pixel, arbitrarily rotated, lighted, Z-buffered.
- *137,000 triangles per second.* 50 pixel, arbitrary strip direction, lighted, Z-buffered.
- *394,000 lines per second.* 10 pixel, arbitrarily directed, depthcued, Z-buffered.
- *210,000 antialiased lines per second.* 10 pixel, arbitrarily directed, Z-buffered.
- *8.3 millisecond full-screen clear.* Both color and Z-buffer banks cleared.

7. *Acknowledgements*

We appreciate and thank the entire Silicon Graphics team.

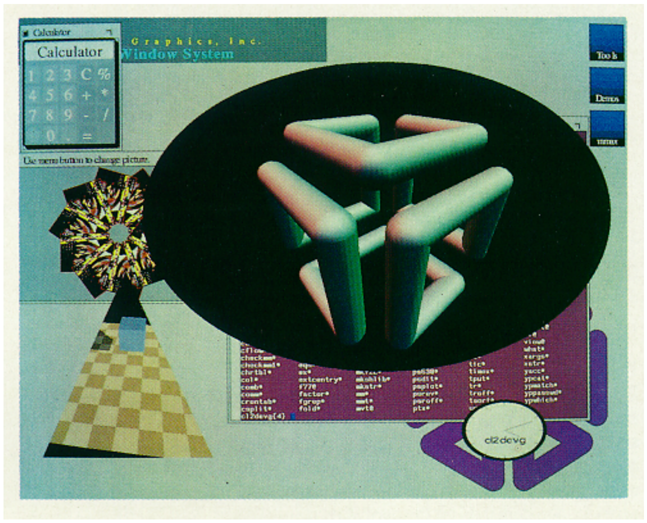


Figure 5

Silicon Graphics Superworkstation windowing system simultaneously exhibits high-performance 3D graphics, multi-mode graphics, and arbitrarily shaped windows.

8. *References*

1. Bresenham, J. Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal 4*, 1 (1965), 25-30.
2. Clark, Jim and Hannah, Marc. Distributed Processing in a High-Performance Smart Image Memory. *Lambda 1*, 3 (4th Quarter 1980), 40-45.
3. Clark, Jim. The Geometry Engine, A VLSI Geometry System for Graphics. *Computer Graphics (ACM) 16*, 3 (1982), 127.
4. Crow, Frank. The Aliasing Problem in Computer-Generated Shaded Images. *Communications of the ACM 20*, November 1977, 799-805.
5. Fuchs, Henry and Johnson, B. An Expandable Multiprocessor Architecture for Video Graphics. Proceedings of the 6th ACM-IEEE Symposium on Computer Architecture (April 1979), 58-67.
6. Gouraud, H. Continuous Shading of Curved Surfaces. *IEEE Transactions on Computers C-20*, 6 (June, 1971), 623-629.
7. Gupta, Satish and Sproull, Robert. Filtering Edges for Gray-Scale Displays. Technical Report, Carnegie-Mellon University, Computer Science Department, 1981.
8. Porter, Thomas and Duff, Tom. Compositing Digital Images. Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27, 1984). In *Computer Graphics 18*, 3 (July 1984), 253-259.
9. Silicon Graphics. IRIS 4D/70 Superworkstation Technical Report. Silicon Graphics, Mountain View, CA 1987.
10. Sutherland, Ivan and Hodgman, Gary. Reentrant Polygon Clipping. *Communications of the ACM 17*, 1 (January 1974), 32.
11. Swanson, Roger and Thayer, Larry. A Fast Shaded-Polygon Renderer. Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics 20*, 4 (August 1986), 95-101.
12. Torborg, John. A Parallel Processor Architecture for Graphics Arithmetic Operations. Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics 21*, 4 (July 1987), 197-204.