Graphics Rendering Architecture for a High Performance Desktop Workstation

Chandlee B. Harrell
Farhad Fouladi

Silicon Graphics Computer Systems
2011 North Shoreline Blvd.
Mountain View, CA 94039-7311

Abstract

Hundreds of commercial applications used in mainstream design activities have demonstrated proven demand for 3D graphics rendering products. The demand is for faster and more powerful renderers, thus creating the system design problem of how to achieve maximum rendering performance from the technology available to implement the system. This paper describes a graphics rendering architecture that takes advantage of several novel architectural features: a custom floating point processing core with tailored data stores and bussing structures, the arrangement of these cores into a SIMD processor for low overhead multiprocessing, and the hyper-pipelining of the fixed point scan conversion units for low overhead, high bandwidth pixel generation into an interleaved frame buffer. These features combine to form a solution to the system design problem which distinguishes itself by its overall performance and its ability to maximize performance while minimizing system size. The resulting architecture is capable of over a half million gouraud shaded Z-buffered triangles per second, with a sustained fill rate for gouraud shaded and Z-buffered pixels of 80M pixels per second. The architecture fits in a desktop workstation.

Introduction

A graphics rendering architecture for a high performance desktop workstation is described.

3D graphics workstations are used by a broad range of applications [IRIS92]. Many of the applications fall into the categories traditionally called computer-aided design (CAD), where the designer makes progressive refinements on the shape and dimensioning of a product based on feedback from visual modeling, and computer-aided engineering (CAE), where the designer also wishes to analyze properties of the design such as thermal and stress gradients or structural strength, in addition to shape and appearance. 3D graphics workstations are used in the following applications, among others: car and airplane design, tool design, packaging design, industrial and product design, furniture design, clothing and shoe design, architectural and civil engineering, production floor and plant design, geothermal and atmospheric analysis, molecular modeling, pharmaceutical design, chemical analysis, and film animation and special effects.

Application packages today running on 3D workstations enable design efforts that are compute intensive, limited only by today's renderers. The complexity of models that renderers can effectively handle is far less than the model complexity with which users are attempting to work. This creates tremendous demand for faster and more powerful graphics rendering systems. How to achieve the highest performance rendering system from the technology available is the system design problem that this demand presents to the system designer.

Further clarification of the graphics rendering system design problem is necessary. Most graphics renderers today perform rapid, accelerated rendering of 3-sided polygons and straight line segments. The renderer receives these basic graphics primitives, each primitive with vertex descriptions defined by the application, and performs the calculations to render the primitive as pixel values into the frame buffer [FOLEY90,SEGAL92,VAND87]. The basic graphics primitives allow close approximation to any arbitrary curve or surface by sub-dividing the curve into line segments or the surface into polygons to the point where the rendered image is visually acceptable to the user. For the system designer, the primitives provide a simple and limited set of processing algorithms that must be accelerated, enabling the focus to achieve high performance systems.

A top level flow diagram is presented in Figure 1 illustrating the process for rendering the basic graphics primitives. The graphics renderer receives polygons or lines from the application process and performs the steps shown in the flow diagram to render each polygon or line as color and Z pixel values into the frame buffer. Details of each processing step are carefully discussed in [FOLEY90] and [NEWM79].

Implementation bottlenecks in a graphics rendering system typically appear: 1) in the floating point compute power available for the world coordinate to screen coordinate transformations and for vertex color computations; 2) in the floating or fixed point compute power available for triangle slope and line slope calculations; 3) in the rate of generation of pixel values from the fixed point iterators; 4) and in the achieved pixel bandwidth into the frame buffer.

Commercial architectures have approached these bottlenecks in a variety of ways. [KIRK90] presents an architecture where the per vertex and slope calculations are performed on the host CPU and multiple iteration engines drive an interleaved frame buffer. [APGAR88] also executes the per vertex calculations on the host, but off-loads most of the slope calculations to a fixed point engine, and uses a unique combination of multiple iteration units to drive pixel results into an interleaved system memory. [AKEL88,89] describe an approach utilizing a serial pipeline of floating point processors for the per vertex calculations, fixed point engines for the slope calculations, and multiple iteration units to drive an interleaved frame buffer. The architecture introduced in [TORB87] also uses multiple floating point processors but arranges them into a MIMD parallel processor, uses a fixed point slope engine, and multiple iterators to drive an interleaved frame buffer. [PERS88] uses a single floating point processor to perform both per vertex and slope calculations, and a single iterator to drive an interleaved frame buffer. Note that the interleaved frame buffer is the only feature common to all the approaches, and that most approaches use multiple iteration units.

The goal of the architecture described here is to provide a powerful graphics rendering system, maximizing performance while minimizing size. The architecture utilizes several novel approaches to overcoming rendering bottlenecks. Floating point performance is accomplished through the custom design of a highly efficient floating point processing core, and by employing multiple cores controlled in a low overhead SIMD parallel processor. The floating point core is tailored to accommodate both the per vertex calculations and the triangle and line slope calculations. Fixed point iteration performance is achieved through hyper-pipelining two identical iteration units, allowing each unit to sustain the pixel generation requirements of multiple pixel memory busses. Each iteration unit is pipelined until technology limits of integration are encountered. The multiple memory busses provide the necessary bandwidth into the frame buffer memory.

These features result in a graphics rendering system solution distinguished by overall performance, and by compactness of size. The architecture is implemented in a desktop workstation [INDIG93]. It is capable of over 1.3 million depth-cued lines per second, over half a million gouraud shaded Z-buffered polygons per second, with a sustained fill rate of 80M gouraud shaded Z-buffered pixels per second.

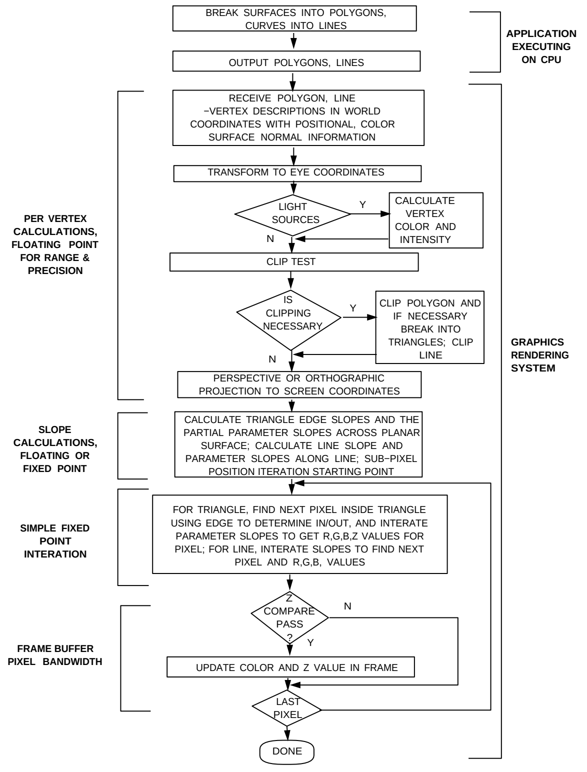
BREAK SURFACES INTO POLYGONS, CURVES INTO LINES
OUTPUT POLYGONS, LINES
APPLICATION EXECUTING ON CPU
RECEIVE POLYGON, LINE –VERTEX DESCRIPTIONS IN WORLD COORDINATES WITH POSITIONAL, COLOR SURFACE NORMAL INFORMATION
TRANSFORM TO EYE COORDINATES
LIGHT SOURCES
Y
N
CALCULATE VERTEX COLOR AND INTENSITY
PER VERTEX CALCULATIONS, FLOATING POINT FOR RANGE & PRECISION
CLIP TEST
IS CLIPPING NECESSARY
CLIP POLYGON AND IF NECESSARY BREAK INTO TRIANGLES; CLIP LINE
GRAPHICS RENDERING SYSTEM
PERSPECTIVE OR ORTHOGRAPHIC PROJECTION TO SCREEN COORDINATES
SLOPE CALCULATIONS, FLOATING OR FIXED POINT
CALCULATE TRIANGLE EDGE SLOPES AND THE PARTIAL PARAMETER SLOPES ACROSS PLANAR SURFACE; CALCULATE LINE SLOPE AND PARAMETER SLOPES ALONG LINE; SUB–PIXEL POSITION ITERATION STARTING POINT
SIMPLE FIXED POINT INTERATION
FOR TRIANGLE, FIND NEXT PIXEL INSIDE TRIANGLE USING EDGE TO DETERMINE IN/OUT, AND INTERATE PARAMETER SLOPES TO GET R,G,B,Z VALUES FOR PIXEL; FOR LINE, INTERATE SLOPES TO FIND NEXT PIXEL AND R,G,B, VALUES
Z COMPARE PASS ?
FRAME BUFFER PIXEL BANDWIDTH
UPDATE COLOR AND Z VALUE IN FRAME
LAST PIXEL
DONE

Figure 1. Process for rendering basic graphics primitives

TOP LEVEL SYSTEM VIEW

This section presents a block diagram of the architecture in Figure 2. The key components are briefly introduced, followed by a description of the overall control structure and the data flow through the system. The subsequent sections discuss each of these key components in detail, describing the critical decisions made to determine their structure, then detailing the internal operation of each component. The final section discusses the technology targeted for the architectural implementation and the implementation results.

The block diagram is shown in Figure 2. The key components are the FIFO interface to the system bus, the Command Processor (CP), the SIMD parallel processor, the dual Raster Engines (RE), and the frame buffer. The SIMD processor is made up of a sequencer, a microcode store, and multiple Geometry Engines (GE). Each GE is a custom floating point processing core. Each Raster Engine is a hyper-pipelined iteration unit.

The SIMD parallel processor executes all the per vertex calculations and the slope calculations shown in Figure 1, the REs perform the fixed point iteration, and the frame buffer pixel bandwidth is determined by the multiple busses into the frame buffer.

Operation is initiated by the CPU sending polygon and line rendering commands into the FIFO across the system bus. The FIFO allows the CPU to generate commands at a rate independent of how fast the rendering occurs. If the FIFO fills up, an interrupt is generated to the CPU for exception handling.

The SIMD parallel processor is fed data from the FIFO by the Command Parser. The CP moves data from the FIFO into the ping-pong input buffers of the Geometry Engines. The GEs read data from the ping-pong buffers, perform necessary floating point computations, and write results to their respective output FIFOs. GE execution is controlled by the common sequencer and control store.

A bus controller resident in the even Raster Engine reads data from the GE output FIFOs and transfers the data into the RE input ping-pong buffers. The REs perform necessary iterations to generate color and Z values and perform the correct pixel updates into the frame buffer. The odd RE generates pixels for the odd numbered scan lines of the frame buffer, and the even RE generates pixels for the even numbered scan lines.

The sections below first discuss the GE custom floating point core solution, followed by a discussion of the control structures required to arrange the GEs into the SIMD parallel processor. This is followed by a description of the hyper-pipelined RE iteration solution.

GEOMETRY ENGINE

The goal for the Geometry Engine design is to achieve the maximum *realized* floating point performance for graphics algorithms, in a single chip solution. The algorithms used for evaluating performance are the per vertex and slope calculations of Figure 1. The decision is made to combine the per vertex and slope calculations into a single floating point solution. Slope calculations are comprised of relatively complex algorithms, difficult to implement in a hardwired fashion, and therefore most effectively implemented in a microcoded processor. Also, the compute cycles required for per vertex calculations is almost evenly balanced with the cycles required for slope calculations. Combining the per vertex and slope calculations into the GE relieves the need to design a second microcoded fixed point processor of similar complexity; and the replication of GEs in the SIMD parallel processor increases both the per vertex and slope processing power together.

The GE design goal is met with a custom floating point processing core. Analysis shows that a custom unit with tailored data stores,

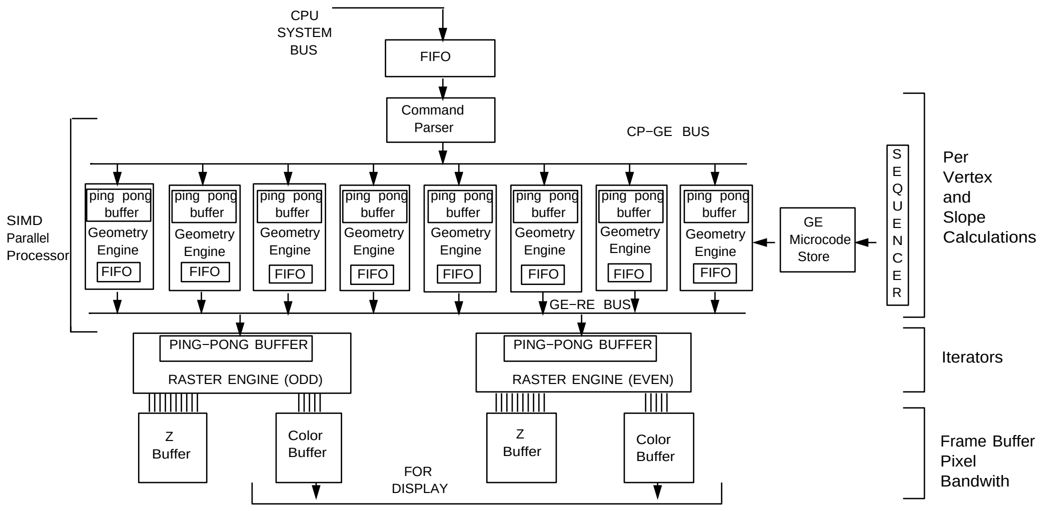

Figure 2. Block Diagram of the architecture

bussing structures, and sequencing control achieves higher realized performance and a more compact solution than available commercial alternatives. Therefore a custom approach is chosen.

Analysis of the per vertex calculations and the slope calculations shows an even balance between multiplies and adds, therefore one multiplier and one adder are chosen for the GE core. The GE design approach follows the fundamental principle of maximizing the utilization of the most expensive resource: the floating point multiplier (FMPY) and the floating point adder (FALU). The following observations for maximizing utilization are taken into account in the GE design: high data bandwidth to the correct operands is needed into the FMPY and FALU; multiple threads of the same algorithm must be active simultaneously. Enough bandwidth to appropriate data storage and data sources is needed to avoid lost cycles waiting on an operand that is slow to retrieve. A single thread of execution may have several additions followed by several multiply operations, thus wasting the FMPY or the FALU until a result is available from the other unit. Multiple threads of execution is the solution.

The Geometry Engine block diagram is shown in Figure 3. Six different busses and four ports from the register file drive the four inputs to the FMPY and the FALU. Two of the busses provide immediate wrap-around of FMPY and FALU results back to their inputs. One bus gives access to the ping-pong buffer loaded by the Command Parser, while two more busses give access to a pair of special data stores. The sixth bus accesses off-chip memory that is used for expansion, and typically holds the global variables for the GE.

A multi-port register file is included for scratch storage of intermediate results. The register file is critical to allowing multiple simultaneous threads of calculation. Feedback paths from FMPY and FALU result outputs are provided for single-threaded operation, but when two threads conflict by needing the same unit for their next computation, then one thread must be stalled by storing the intermediate result in the register file until the appropriate unit becomes free.

On the other hand, a multi-port register file is an expensive commodity and its size is limited. Reviewing the per vertex calculations concludes that the ping-pong buffer and the register file are sufficient to perform the per vertex calculations with maximum FMPY and FALU utilization. On reviewing the slope calculations, however, it is noted that frequently data from each vertex of a triangle, or both vertices of a line, are needed simultaneously during multi-threaded computation. The register file cannot be made big enough to hold the data structures for each vertex. The GE is designed to have three separate data stores, one for each vertex of a triangle or for the two vertices of a line, used during the slope calculation process. The ping-pong buffer is used to hold the data structure for one vertex, while the two special data stores hold the data structures for up to two more vertices.

This extensive memory and bussing structure is wasted without flexible independent addressing and flexible control of data movement. This is accomplished through a very wide instruction word which allows control of the breadth of resources.

The result of the described structure is that simultaneous access can be made to the ping-pong buffer, the two special data stores, the global variables memory, the result outputs, and the register file by any of the four FMPY and FALU inputs. Multiple threads of execution supported by this accessible bandwidth into the FMPY and FALU inputs maximizes FMPY and FALU utilization.

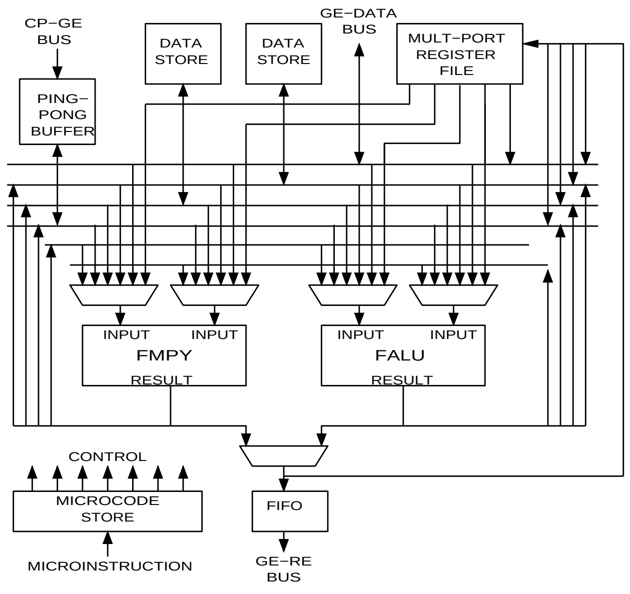

Figure 3. Geometry Engine block diagram

GE operation occurs as follows. The Command Parser loads data into the ping-pong buffer. The ping-pong buffer allows CP loading of data into one side of the buffer while the GE is executing and accessing the other side of the buffer. The CP initiates GE execution by informing the GE sequencer that data is fully loaded. The sequencer looks up instructions in the GE microcode store, and these instructions control the execution functions of the GE. For lines and triangles, the GE performs per vertex calculations, accessing data from the ping-pong buffer, then constructs vertex data structures based on screen space coordinates and puts one vertex data structure back in the ping-pong buffer and up to two more vertex data structures into each of the special data stores. Slope calculations are then performed, drawing operands from the ping-pong buffer and the two special data stores. Calculated iteration coefficients and initial values are passed to the Raster Engines by storing them to the output FIFO.

SIMD PARALLEL PROCESSOR

A single floating point processor cannot achieve the desired performance. Therefore multiple floating point processors are used in the design. The following goals for multiprocessing led to the SIMD parallel processor solution: 1) a linear performance increase must be achieved with the addition of Geometry Engines; 2) the multiprocessing solution must have the lowest possible impact over and above a uniprocessor solution.

Three approaches are considered for the multiprocessing solution. The first is a pipeline of floating point processors [AKEL88, 89]. Each pipeline stage performs a subset of the per vertex and slope computations, passing intermediate results to the next processor in the pipeline. Each pipeline processor is executing a different set of code to implement its separate subset of the algorithm. This approach has several disadvantages. The throughput of a pipeline is the speed of the slowest processing step. Overall performance is determined by the processor with the biggest subset of the algorithm to process. Since the algorithm cannot be divided into perfectly equal subsets, a less-than-linear performance gain is achieved. Also note, that to add processors, a new subdivision of the algorithm must take place and new code must be written and tuned. The final disadvantage of this approach is in the burden of overhead the approach requires. Although having the advantage of not requiring the distribution mechanism at the head of the pipe needed by the next two approaches considered, each processor does require its own sequencer, microcode store, globals data store, in addition to control logic to interface each of the pipeline stages.

The second approach considered is a parallel MIMD (Multiple Instruction Multiple Data) array of processors [TORB87]. Each processor performs independent execution of the per vertex and slope calculations for its own polygon or line primitive. Linear performance gains are attained when the same kind of primitive is distributed to each processor, thus satisfying the first multiprocessing goal. Processors may be added without requiring changes to processor code. The disadvantage of the MIMD parallel processor lies in the overhead required to implement such an approach. A parallel processor requires a distribution function that takes primitives in the FIFO (received from the CPU) and disburses a primitive to each of the processors present. A MIMD parallel processor also requires that each processor has its own sequencer, microcode store, and globals data store.

The third approach considered is a parallel SIMD (Single Instruction Multiple Data) array of processors. Each processor executes the same instruction in lockstep, but is computing results for its own polygon or line primitive. Like the MIMD processor already examined, the SIMD parallel processor achieves linear performance gains with the addition of processors when the same kind of primitive is distributed to each processor. The advantage of the SIMD approach is in the low overhead required to implement a multiprocessor. All processors share the same sequencer, the same microcode store, and the same globals data memory. The only implementation overhead required over a uniprocessing solution is the addition of the distribution function. It is worth noting that this is a simple function and therefore a small overhead to tolerate. The SIMD parallel processor is chosen as it optimally achieves the multiprocessing goals.

Note that a key assumption to accomplishing linear performance gain from a parallel processor (SIMD or MIMD) is that the same kind of primitive is distributed to each of the processors (all lines or all polygons). This requires that the primitives coming through the FIFO from the CPU arrive in significant groupings of lines together and polygons together, rather than a fully random distribution of lines and polygons. For a MIMD processor, if the FIFO holds alternating lines and polygons, the throughput slows down to the rate of the slower primitive - the polygon. For a SIMD processor, alternating lines and polygons is a worst case scenario. Performance will reduce to that of a uniprocessor. Extensive analysis of model data sets used on 3D workstations shows polygons typically clump in large bunches and lines do the same. This is particularly true of CAD/CAE applications. The result is linear performance gain for parallel processor arrangements.

The unique system features required for SIMD parallel processing will now be discussed. Please refer to Figure 2. The features included for SIMD processing are the distribution function performed by the CP, sequencing functions to allow SIMD branching, common bus for the microinstruction, common bus for the globals data store, and indirect addressing requirements into GE memories. The GE input ping-pong buffer and output FIFO are also crucial to performance.

COMMAND PARSER

To describe the operation of the Command Parser, we must first explain the needs of the distribution function. The purpose of the CP is to analyze the command and data stream coming through the FIFO, distribute data accordingly to the GEs, and subsequently initiate GE execution. To perform this function, the CP must detect boundaries between primitives, detect whether subsequent primitives are of the same or different kind, and maintain the correct order of primitive disbursement to the GEs.

Please refer to Figure 4 for a diagram of the Command Parser. The CP is microcoded for flexibility. This allows different routines for primitives comprised of vertices with different kinds of attributes, and the exception handling of polygons with greater than three sides.

CP operation begins with the arrival of a command token in the FIFO. The command token causes the CP sequencer to branch to a routine appropriate for the kind of primitive arriving in the FIFO. This branch mechanism inherently defines primitive boundaries. The command token is read from the FIFO and stored in the Current Command register. A compare function allows branching based on whether the current command token just arrived is identical or different from the last command token received. If the token is identical, then the arriving primitive can be distributed to the next GE in the parallel processor. If the token is different, then the GEs that have already been loaded with data must swap their input ping-pong buffer and begin executing before the arriving primitive can be distributed to the next GE. The token compare mechanism allows the CP to branch to different routines to handle these two cases.

The CP must determine to which GE the arriving primitive should be written. A round robin scheme of distribution is chosen,

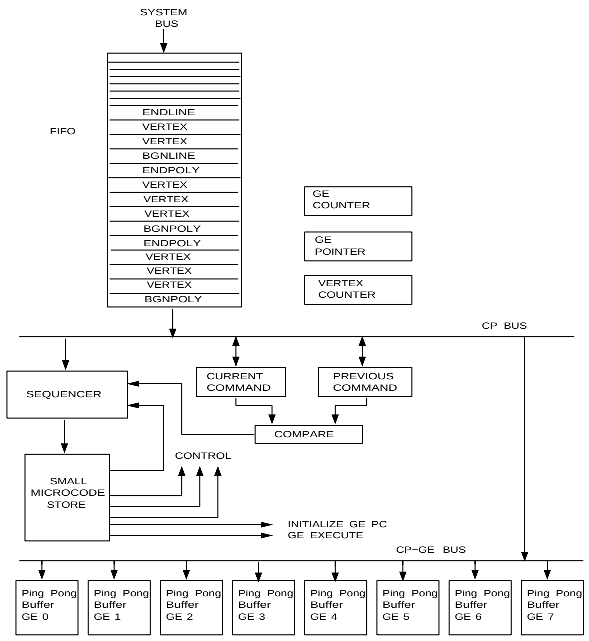

Figure 4. Command Parser block diagram

primitives being loaded in a continuous sequence from GE #0 through to GE #7, and back around. Referring back to Figure 2, primitive coefficients calculated by the GEs are pulled from the GE output FIFOs in the same round robin order. A pointer to the GE that is currently being loaded, and a counter which maintains the number of GEs that have been loaded since the last execute command provide the tools to determine for which GE the arriving primitive is destined. The incrementing and clearing of these counters is under microcode control. After choosing the appropriate GE, the CP pulls vertex data from the FIFO and writes it across the CP-GE Bus and into the GE's ping-pong buffer.

Once all 8 GEs have been loaded, or when the current primitive is different from the previous primitive, the CP must initiate GE execution. The CP first tells the GE sequencer which GEs are loaded, passes the GE sequencer the appropriate address to begin execution, and then issues the GE sequencer an execute command. An interlock mechanism will stall the CP if the GE is currently executing at the time of the CP execute command, and will initiate GE execution only when the previous execution is complete. Once the interlock mechanism clears, it is an indication that the GE ping-pong buffers have been swapped, and the CP resumes distribution of primitives from the FIFO.

GE SEQUENCER

The GE sequencer is shown in Figure 2. The sequencer is based on a standard uniprocessor design. Flexible branch functions are supported for jumps and subroutine calls. Branching is controlled within separate fields of the GE's wide instruction word. This allows concurrent branching with the GE datapath control, thus not affecting datapath performance thru branches.

To this uniprocessor design base are added functions which allow control of multiple SIMD processors. The GE sequencer has control to stall each of the GEs independently. This control is used in two different ways. The first is on receipt of an execute command from the CP once the GEs are idle. The GE sequencer will decode which GEs the CP has loaded from information passed by the CP. Those GEs not loaded will be stalled by the GE sequencer for the duration of the primitive execution. The second fashion the stall control is used is for implementing conditional subroutine calls across SIMD processors. If a subset of the processors does not pass the condition, that subset is stalled by the GE sequencer for the duration of the subroutine call, while the remaining processors execute the subroutine. As an example, conditional subroutine calls are used for implementing the lighting and clipping branches shown in Figure 1.

MICROCODE STORE AND GLOBALS MEMORY STORE

The GE sequencer accesses the next microinstruction from the GE microcode store (Figure 2). The microinstruction word controls all the GE internal functions, as well as the GE sequencer. The piece of the microinstruction word controlling the GEs is bussed to all the GEs for simultaneous execution.

Additional memory (not depicted) can be added external to the GEs as an expansion memory to store global variables required in execution. The GE Data Bus (Figure 3) of each GE is bussed together and connected to a globals memory store.

INDIRECT ADDRESSING

As explained in the section above on the Geometry Engine (Figure 3), data is read from the ping-pong buffer and the two special data stores to perform the slope calculations for a line or triangle. Depending upon orientation of the primitive on the screen, these data stores may need to be accessed differently by different processors. In order to do this effectively in a SIMD processing environment, indirect addressing is provided into these data stores. This minimizes cycles spent out of SIMD lockstep execution and is crucial to SIMD performance.

INPUT PING-PONG BUFFER AND OUTPUT FIFO

The GE input ping-pong buffer and the GE output FIFO are also crucial to SIMD performance. Without a ping-pong buffer at the input to the GE, the CP would have to load 8 GEs *after* GE execution of the previous primitive completes, eliminating significant parallelism. The FIFO at the GE output allows all GEs to write their results in lockstep execution. Without the FIFO, a SIMD implementation would not be feasible.

RASTER ENGINE

The goal for the Raster Engine is to obtain the fastest gouraud shaded Z-buffered fill rate in a single chip. It is also desired to be able to use multiple copies of the same chip to obtain further increases in rendering performance.

There are two major bottlenecks in rasterization: pixel generation, and memory bandwidth. Pixel generation, the first bottleneck, can be increased in two different ways. Contemporary architectures have traditionally increased the rate of pixel generation by replicating in parallel the number of fixed point iterators, utilizing enough iterators to achieve the desired pixel rate. Hyper-pipelining a single iteration unit is the approach taken in this architecture. Hyper-pipelining adds pipeline stages to a single iterator until the desired rate of pixel generation is achieved. The pipeline stages added to the iterator require significantly fewer gates than would be required to replicate iterators. Therefore, hyper-pipelining is chosen as the minimum solution for performance. Memory bandwidth, the second bottleneck, is increased by using an interleaved frame buffer across multiple memory banks.

Determining the total number of pipeline stages and the number of memory busses for the RE is a recursive process, and depends on the integration limits of technology. To achieve the maximum fill rates, the iteration pipeline must support a pixel generation rate of

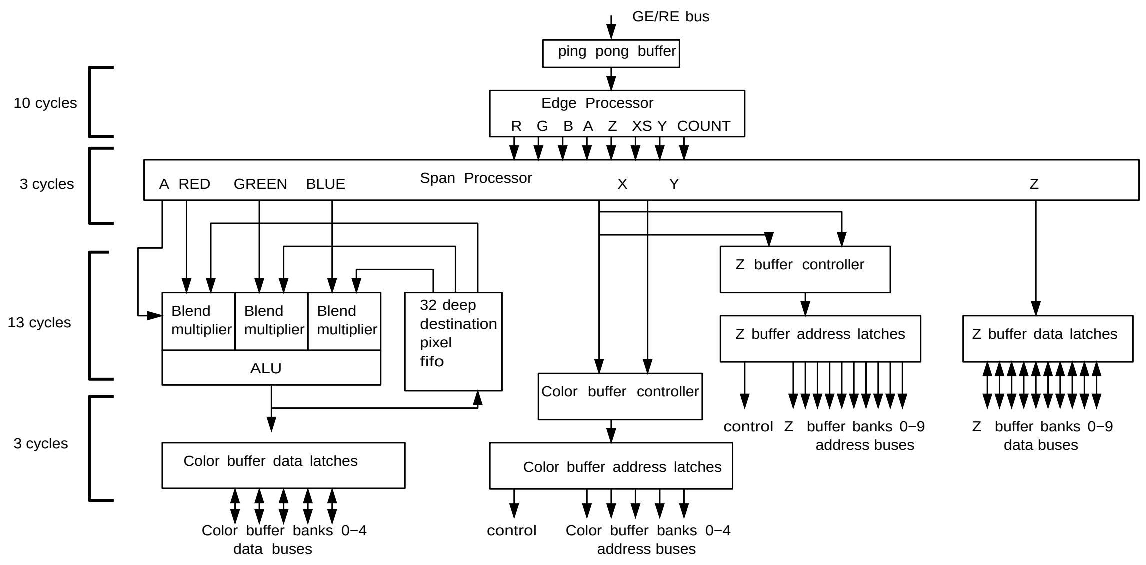

Figure 5. Raster Engine block diagram

N times the page mode bandwidth of a frame buffer DRAM, where N is the number of memory busses used. A sample pipeline depth is analyzed and the die size computed. The conclusion of this recursive process led to the resultant architecture with a single hyper-pipelined Raster Engine driving a five-way interleaved color buffer.

Given a five-way interleave on the color buffer, the pipeline clock rate is set at five times the DRAM page mode bandwidth, under the assumption a pixel is generated every clock. The slowest element of the RE pipeline is the key to ensuring the clock rate can be met, and is what was checked during the recursive analysis. This element is the DDA unit of the iterators. A DDA unit consists of a two input adder with a 2:1 multiplexer on one of its inputs. The output of the adder is fed into a register which is then fed back to the second input of the adder. The resultant clock rate for a five-way interleave color buffer drives the number of pipeline stages in the Raster Engine. The hyper-pipelined Raster Engine has 26 pipeline stages from the input ping-pong registers which hold the line and triangle iteration parameters to the point where pixels are written into the color buffer.

For the system architecture implemented, it is decided to incorporate two raster engines to obtain the desired performance on the desktop.

The RE implementation is now discussed in detail. A diagram of the Raster Engine is shown in Figure 5. The RE is capable of drawing rectangle, triangle and line primitives. Each primitive requires a set of iteration coefficients which are downloaded from the GE FIFOs into the RE ping-pong buffers. Once the ping-pong buffers are loaded, the RE initiates rendering of the primitive.

The execution units of the Raster Engine consist of four major sections:

- > edge processor;
- > span processor;
- > per-pixel operators;
- > memory controllers.

The edge processor combines with the span processor to perform the task of converting a primitive into pixels. The edge processor decomposes triangles into horizontal spans, and decomposes lines into pixels. It has two iterators for computing the beginning and end X location of the span, and six iterators to computer R,G,B,A,Z,Y for the first pixel on the span. Next some terms must be defined. The major edge of a triangle connects the vertex with maximum Y coordinate value to the vertex with minimum Y. The edge connecting the vertex with maximum Y to the vertex with the middle Y value is called the first minor edge. The edge that runs between the vertex with the middle Y and the vertex with the minimum Y value is termed the second minor edge. The edge processor begins by iterating down the major edge and the first minor edge. When the processor detects the middle Y has been crossed, it swaps the first minor edge with the second minor edge and continues down the triangle until the minimum Y coordinate is reached. For each span, the edge processor computes the initial R,G,B,A,X,Y,Z values for the first pixel on the span as well as the number of pixels that have to be rendered for that span. This information is passed to the span processor. When drawing lines, only one of the two edge iterators is used to generate the X coordinate. The edge processor has 10 pipe stages and can generate a new span every other clock.

The span processor has 6 iterators. These iterators walk through the pixels on a span and generate the R,G,B,A,X,Z parameters for each pixel on the span. The processor can generate one or four pixels per clock. When gouraud shading and/or Z-buffering, the span processor will generate one pixel per clock in the X direction. When a span is flat shaded and not Z-buffered, the span processor generates 4 pixels per clock. The block write feature of the VRAMs used in the color buffer is utilized to write all 4 pixels generated in one memory cycle, thus quadrupling the fill performance for screen clears and for rendering flat shaded 2D surfaces. For lines, parameters from the edge processor get passed through. The span processor has a pipeline latency of 3 clocks.

The Raster Engine supports a rich set of pixel operators required by commonly used graphics libraries [SEGAL92, VAND87]. Pixels operators fall into two categories. The first category of operators modify the color of the pixel, such as logicop and blend. Blend and logicop are operations performed between the generated source color and the destination color that is already stored in the color buffer. They require readback from the color buffer which is described below. There are three sets of multipliers to perform the blend function for the R,G,B components. These multipliers are followed by an ALU which performs the logic operations. These two sections together contain 10 pipeline stages.

The second category of pixel operators perform tests on pixel parameters to allow conditional updating of color pixel values. Examples in this category are the Z-compare test and stencil test. The Z-compare test is used to determine pixel visibility in the third dimension. The stencil test is used to provide more general conditional test operations. The Z-comparison is done in parallel with blend and logicop in the same number of pipeline stages.

There are memory controllers for two separate memory ports on the Raster Engine: the color buffer port and the Z-buffer port. The color buffer is a five-way interleaved memory port, and the Z-buffer is a 10-way interleaved memory port. The Z-buffer operation consists of reading back the old Z value stored in the Z-buffer, comparing that Z value with the newly generated Z value and, if the comparison passes indicating the new pixel is visible, the new Z value and color value are written into the Z-buffer and color buffer respectively. Since the Z-buffer requires two accesses (a read and a write) for every write access to the color buffer, the Z-buffer port is designed with twice the interleaving of the color buffer to accommodate Z-buffered fill at the color gouraud shaded update rates.

As we noted above, a write access to the color buffer takes 5 clocks. Similarly, the pipelined read-modify-write access to the Z-buffer takes 10 clocks. Adjacent pixels along a span are allocated to adjacent banks of the Z-buffer interleave. Since it takes 10 clocks to perform a read-modify-write, and we have a 10-way interleave, bank contention does not occur along a span and a one pixel per clock comparison rate is achieved.

The 10 banks of the Z-buffer interleave share the same page address to reduce memory controller complexity. There is a single block of logic for page fault detection. Each bank can access a different column address within the page. A score boarding technique is used to keep track of the state of each bank. When a pixel is dispatched to a bank, a bit in the score board is set to specify that the bank is busy. Thus, any pixel accesses to the same bank will be blocked and a bank contention stall generated to stop pixel flow until the bank is again idle.

The color buffer has a five-way interleave. As explained above, the pipeline depth is chosen such that five pixels are generated in a single VRAM page mode cycle time, allowing contentionless color fills along a span. Read-modify-write operations to the color frame buffer (for blend and logicop) are supported at half the fill performance of straight color write operations. Values in the color buffer are first read into a FIFO in the RE to await the “modify” step of the operation. When the FIFO fills, the contents of the FIFO are then merged with the newly generated incoming pixel stream and the result is written back into the color buffer. This two-pass operation is continued until rendering is complete. The color buffer memory controller has a 3 clock latency.

The operation of two REs together will be briefly discussed. The two Raster Engines work on the same primitive together. The rendering task is split based on span number. All even spans of a primitive (when the Y coordinate is even) are rendered by the “even” Raster Engine; all odd spans are rendered by the “odd” Raster Engine. This results in a doubling of fill performance. The edge processor in each RE iterates through all spans, but each RE rejects the spans that do not belong to it, and the edge processor continues iteration to the next span.

TECHNOLOGY

This section briefly discusses the technology used in the implementation. The technology targeted for the custom logic design is a 1.0 micron double metal CMOS gate array and standard cell process. The process can achieve the equivalent of 100K gates on a single die. The 1M-bit DRAM family is the targeted memory technology. The design consists primarily of custom parts and memory components. The design contains over a million gates of custom logic, and is implemented across three 5” x 13” PC boards.

CONCLUSION

A graphics rendering architecture has been described which is distinguished by its overall performance, and by its ability to maximize performance while minimizing system size. The architecture is shipping as a product in the IRIS Indigo Extreme. A scaled version of the architecture was introduced in IRIS Indigo^2 Elan. The architecture provides state-of-the-art rendering performance in a desktop 3D workstation.

ACKNOWLEDGEMENTS

Sincere thanks to Marc Hannah and Dave Galbi for their major efforts on the architecture. Thanks to Vimal Parikh for his advice on the floating point solution. Finally, overwhelming appreciation must go to the whole design team, every one of whom made significant contributions, and who made it possible.

REFERENCES

[AKEL88] K. Akeley, T. Jermoluk, “High-Performance Polygon Rendering”, Computer Graphics (Proc. SIGGRAPH), Vol. 22, No. 4, August 1988, pp. 239-246.

[[AKEL89] K. Akeley, “The Silicon Graphics 4D/240GTX Superworkstation”, IEEE Computer Graphics and Applications, Vol. 9, No. 4, July 1989, pp. 71-83.

[APGAR88] B. Apgar, B. Bersack, A. Mammen, “A display system for the Stellar Graphics Supercomputer Model GS1000”, Computer Graphics (Proc. SIGGRAPH), Vol. 22, No. 4, August 1988, pp. 255-262.

[FOLEY90] J. Foley, A. van Dam, S. Feiner, J. Hughes, “Computer Graphics, Principles and Practice”, 2nd edition,, Addison-Wesley Publishing, 1990.

[INDIG93] “The Indigo2 Technical Report”, Silicon Graphics Computer Systems, 1993.

[IRIS92] “Iris Partner Catalogue”, Silicon Graphics Computer Systems, 1992.

[KIRK90] D. Kirk, D. Voorhies, “The Rendering Architecture of the DN10000VS”, Computer Graphics (Proc. SIGGRAPH), Vol. 24, No. 4, August 1990, pp. 299-308.

[NEWM79] W. Newman, R. Sproull, “Principles of Interactive Computer Graphics”, McGraw-Hill Book Company, Second Edition, 1979.

[PERS88] “The Personnal Iris™: A Technical Report”, Silicon Graphics Computer Systems, 1988.

[SEGAL92] M. Segal, K. Akeley, “The OpenGL™ Graphics System: A Specification (version 1.0)”, Silicon Graphics Computer Systems, 30 June 1992.

[TORB87] J. Torborg, “A Parrallel Processor Architecture for Graphics Arithmetic Operations”, Computer Graphics (Proc. SIGGRAPH), Vol. 21, No. 4, July 1987, pp 197-204.

[VAND87] A. van Dam, et. al., “PHIGS+ Functional Description Rev. 2”, Jointly developed PHIGS+ specification, 1987.

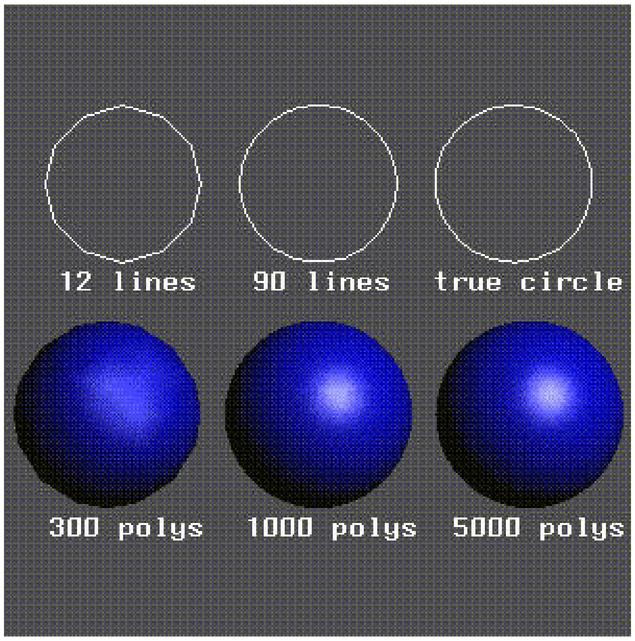

Figure 6. Demonstration of curve and surface approximation using graphics primitives. Note effect of increasing tessellation depth on image quality, and on the number of primitives to render.

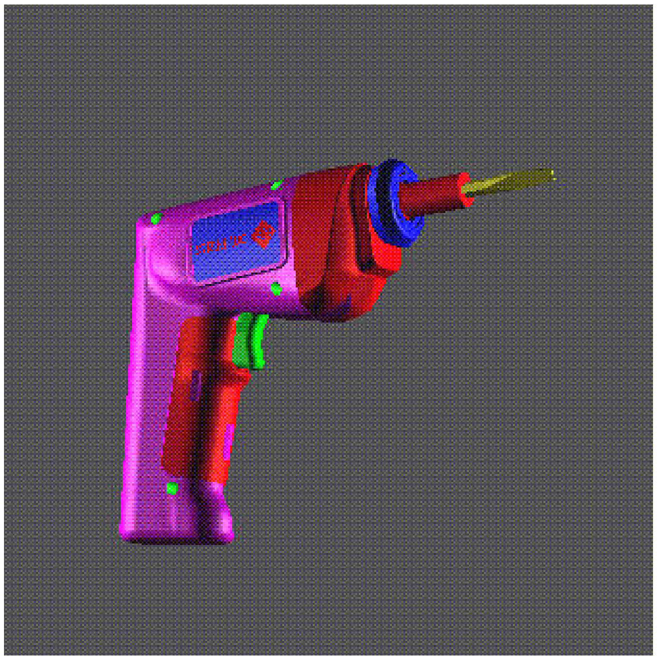

Figure 7. Shaded-lighted image (2 directional lights) (Data Courtesy of Cisigraph Corporation) has 31774 triangles, 827961 pixels and was rendered in 0.13 seconds.

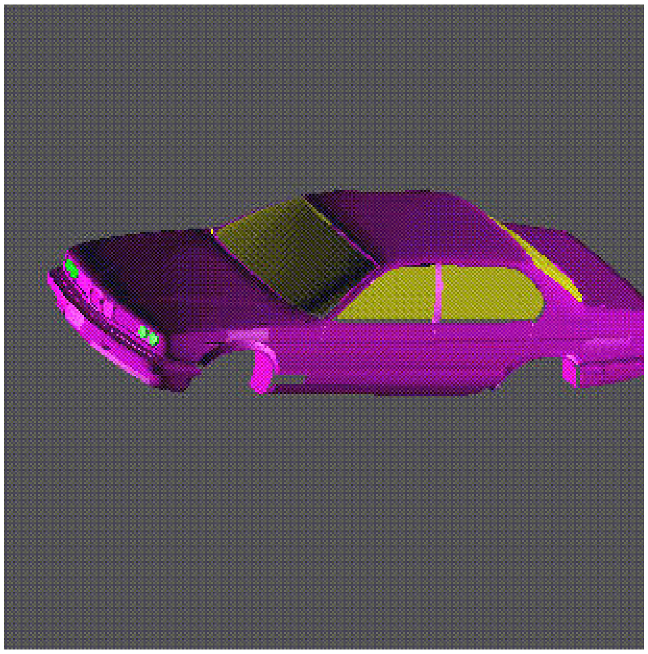

Figure 8. Shaded-lighted image (2 directional lights) (data courtesy of Cisigraph Corporation) has 77420 triangles, 526235 pixels, and was rendered in 0.29 seconds.

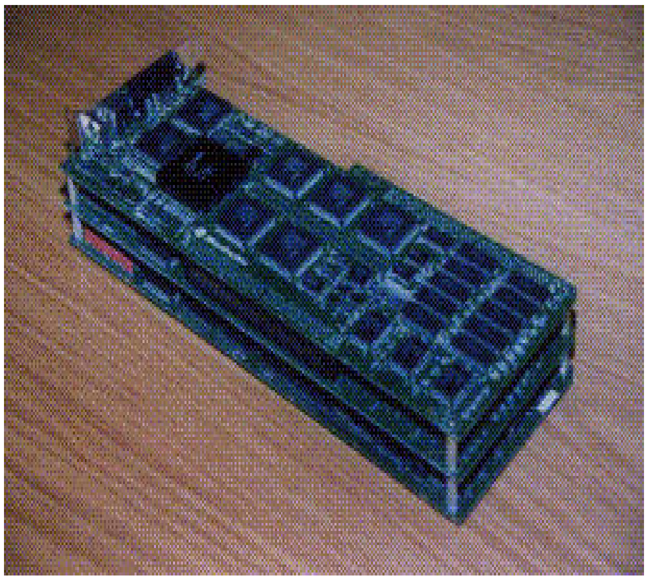

Figure 9. Indigo^2 Extreme graphics render board set..