Accommodating Memory Latency In A Low-cost Rasterizer

Bruce Anderson,
David Sarnoff Research Center[1]

Rob MacAulay
Ben Cheese Electronic Design[2]

Andy Stewart
Phoenix VLSI Design[3]

Turner Whitted
The University of North Carolina[4]

Abstract

This paper describes design tradeoffs in a very low cost rasterizer circuit targeted for use in a video game console. The greatest single factor affecting such a design is the character of memory to which the image generator is connected. Low costs generally constrain the memory dedicated to image generation to be a single package with a single set of address and data lines. While overall memory bandwidth determines the upper limit of performance in such a small image generator, memory latency has a far greater effect on the design.

The use of Rambus memory provides more than enough aggregate bandwidth for a frame buffer as long as blocks of pixels are moved in each transfer, but its high latency can stall any processor not matched to the memory. The design described here utilizes a long pixel pipeline to match its internal processing latency to the external frame buffer memory latency.

1. Small Scale Image Generator

High performance graphics which was once limited to hugely expensive flight simulators, and which eventually made its way into small laboratories has rapidly become a standard fixture on the desktop and in video game consoles. However, designs of graphics processors for the low end bear little resemblance to their larger predecessors. As with any mass-market product, the overriding constraint in a low-end graphics system is cost. Given even modest performance targets, the need to minimize costs, and the consequent need to interface to commodity memory parts, devising an appropriate architecture is something of a trick.

[1]Princeton, New Jersey, USA banderson@sarnoff.com
[2]Royston, Cambridgeshire, UK rob@benchees.demon.co.uk
[3]Towcester, Northamptonshire, UK andys@phoenixvlsi.co.uk
[4]Chapel Hill, North Carolina, USA jtw@cs.unc.edu

In this paper we start from a design decision to use high bandwidth, high latency, physically compact, commodity Rambus memory. Given that choice, we demonstrate that a long pixel pipeline makes most effective use of the available memory bandwidth, and we show the results of simulating a specific implementation of the architecture. That such a mechanism is effective is not surprising since several current designs use it. We make no claim of originality; we merely wish to examine the performance of this type of design.

In the following sections, we review some of the more common rasterizer design approaches, including two new low-cost ones, delve into a more detailed discussion of matching rasterizer processing latency to off-chip memory latency, pursue this design approach with an extremely low-cost rasterizer example, and finish with simulation results which illustrate the effectiveness of the technique. (To be clear, our definition of extremely low cost means a combination of graphics processor and memory costing in the low tens of dollars.)

While the performance benefits of matched latencies for best case geometry may be obvious on paper, we have found it useful to simulate the example rasterizer to see how badly performance degrades for less than optimal geometry.

2. The Memory Interface Bottleneck

DRAM access is a bottleneck in graphics system design. The effects of limited memory bandwidth and ways to overcome it have been discussed previously [10]. However, the limitations imposed by memory latency have not been widely described.

A fundamental limitation of current designs stems from choice of algorithm. Almost all polygon image generators support the z-buffer visibility algorithm. This requires that data representing a single pixel be read from memory, processed, and then conditionally written. Because of latency in memory accesses, data read from the z-buffer is not immediately available to the display processor. Contention for memory further aggravates the latency problem.

While the alternative of back-to-front pre-sorting of polygons can avoid the overhead of memory read latency suffered by the z-buffer visibility test, pre-sorting has its own overhead. Furthermore, it is hard to resist employing translucency effects which are enabled by the drawing in back-to-front order. Like the z-buffer algorithm, translucency effects require a read before write and share the latency problem with the z-buffer.

For a number of years, the trend in the design of image generators has been to gain greater performance through the use parallel processors and interleaved memory. This has been especially true for that portion of the hardware which writes individual pixels. Here, the common practice has been to couple each of a large number of processors directly to an individual memory chip [5,3,1]. Both the high degree of parallelism and the autonomy of each pixel processor reduce the memory access problem even when using commodity DRAM. In an extreme case the processor is embedded directly in memory [7] but this is no longer an inexpensive memory part.

A different approach to overcoming the memory access bottleneck is to cache the frame buffer within the pixel processor [6]. FBRAM also uses a pixel cache, but moves both the cache and portions of the pixel computation into the memory itself [4]. This is in keeping with a trend in memory design that either explicitly includes a cache inside the DRAM package or uses the latches on the sense amplifiers as a row buffer, which can be considered a cache. Rambus memory, as one example, achieves peak transfer rates out of its row buffers of nearly 500 megabytes per second over an 8-bit wide data path by moving data out of a row buffer on both edges of a 250 MHz clock. Even so, latency in Rambus memory chips is still high, and its effects are made worse when spatial coherence exists only for small runs.

However, because cost is such an overwhelming concern in a consumer product, the use of such physically compact high-bandwidth commodity memory chips is attractive in spite of their high latency. This dictates that the low-cost designer must learn to live with memory latency. Physical compactness carries another burden since the lack of parallel datapaths to multiple memory chips takes away some of the design flexibility that one would want in a rasterizer. With this constraint, the most popular way to attack latency is to stretch the processor pipeline so that its processing latency matches the external memory latency. (The reasons why this provides better processing efficiency will become clearer in the following section.)

Among the low cost systems which have been implemented with high latency memory are SGI's graphics system for the Nintendo 64 and Microsoft's Talisman [8]. Like the workstation processors that preceded them, these designs must cope with memory contention caused by texture fetches as well as the frame buffer read-modify-write cycles mentioned above, albeit in very different ways.

The Nintendo 64 reportedly employs a long pixel pipeline that matches processing latency with memory latency for frame buffer reads and writes. Texture is cached on-chip, and presumably moved to the cache in large enough blocks that raw bandwidth makes up for any re-use inefficiency.

Microsoft's Talisman rasterizer chip avoids off-chip read-modify-write frame buffer operations altogether by caching both textures and the frame buffer [8]. Because entire tiles of the output image are generated in a single pass, the off-chip frame buffer becomes write only and latency is not an issue for frame buffer accesses. As an important side benefit, using an on-chip frame buffer cache makes the use of an A-buffer algorithm [2] practical. Latency for texture reads is an issue, and it is further aggravated by using compressed textures which must be de-compressed on-the-fly in circuitry which adds even more delay to texture fetches. A complex combination of predictive texture caching and two levels of cache alleviates this problem. One would assume that the Talisman circuitry would be more expensive than either the Ninetendo 64 or the simple rasterizer used as the example in this paper.

It is worth noting that the effectiveness of texture caching is a complex, poorly understood topic on its own and is beyond the scope of this paper. However, one should not extend assumptions about texture caching performance for flight simulators and workstations to game engines. The amount of content tuning to match the characteristics of the game platform has traditionally been extreme for games. To a certain extent, this means that a knowledgeable game designer can force texture cache effectiveness to be whatever is needed by tuning the content [9].

3. Living With Latency

As noted above, the nature of both z-buffering and transparency is that frame buffer data is first read, then modified, and finally written. In many rasterizers, including the one described in this paper, pixel data is accessed in consecutive memory locations which represent consecutive pixels along a horizontal row. We refer to a row of pixels between the left and right edges of a polygon as a *span*.

Because of its reasonable cost, small physical size, and low pin count, Rambus memory has been a popular choice for several low cost graphics systems. Internally, the memory uses conventional DRAM cells and caches each row, so out-of-page accesses are a significant performance hit. Furthermore, with the particular memory controller used in this project, even best case read latency is approximately 62 nanoseconds.

One way to boost the performance of the rasterizer is to heavily pipeline its operations. This introduces an internal processing latency. In the worst possible case the memory latency would be cascaded with the processing latency to reduce pixel throughput to a level even lower than from memory latency alone. The goal of our design is to achieve the best case in which the processing latency and the memory latency are concurrent, and match each other.

In our circuit, pixels along a span are read and written in blocks which are a multiple of 8 bytes. As indicated Figure 1, the crucial delay is that between initiating a read request and data ready. Pipelining the per-pixel computation along a span introduces a series of intermediate processing steps which can be arranged to overlap the period between read requests and actual data reads. As we show in the following sections, ordering the pixel calculations so that only a small calculation is required to compute final pixel color gives the quickest possible turn-around between data reads and pixel write requests. Data writes also incur latency which spills over into the following read request. For the sake of convenience, we lump all of these delays together. However, at low resolution, we have been able to optimize by arranging data in the frame buffer so that writes following a read are highly likely to fall into the same page as the read.

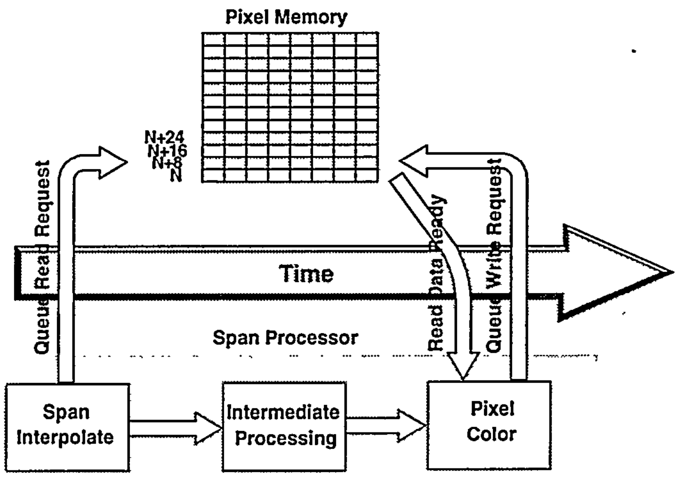


Figure 1. Memory latency and pixel pipeline.

4. Design

As stated above, the architecture of this graphics system is derived from constraints. Over and above all other considerations, it is designed to be inexpensive. Its features include z-buffering, translucency, and perspective corrected texture mapping with bi-linear filtering. No attempt is made to support z-buffering and transparency simultaneously. Either textures or surface colors can be transparent, but not both simultaneously. Each of these feature compromises is made in the name of reduced chip complexity.

At the front end, our target was to be competitive with the current generation of 32-bit game consoles at a peak geometric throughput of between 300,000 and 400,000 polygons per second. The fill rate of 25 million pixels per second is high enough to display simple 3D games at 640x480 resolution and more than adequate to drive a single channel HMD at 320x200 resolution. Aiming for a 0.5 micron fabrication process, timing estimates from placement and routing of portions of the ASICs designed for the game console showed us that a system clock of 50 MHz would be the upper limit. Verilog simulations of just the rasterizer indicate that it will run at speeds as high as 80 MHz.

The design process is not merely an exercise in balancing performance versus cost. Once the desired features are established, a top level design can be reached by varying the free design parameters. This top level design was refined in the following steps:

1. A single data path was established for pixel calculations, i.e. only one pixel comes out of the pipe at a time. This then forces the pixel fill rate to be a sub-multiple of the clock frequency.

2. The designers felt that one pixel every clock period was overkill for the target application and that half that rate would save chip area while providing adequate performance. With a 50 MHz clock, the fill rate for this design is 25 million pixels per second. Since some operating modes require fewer operations it is quite practical to run faster for simple operations, e.g. Gouraud or flat shading, and slower for more complex modes, e.g. transparent bi-linearly interpolated textures. Because the schedule for this project was aggressive, we settled on fixed timing for all modes as a way to simplify debugging of the design.

3. With these parameters fixed, most of the remaining design decisions follow from the timing.

The high level block diagram of the rasterizer is given in Figure 2 below. In the simplest possible terms, the design attempts to match its throughput to the expected latency of the memory, and stalls when that latency is exceeded.

Of all the paths in the circuit, texture calculation is the longest. After interpolating the homogeneous texture coordinates, *u*, *v*, and *w*, perspective corrected *u*, *v* pairs are fed into a FIFO. This is the narrowest data path in the circuit and therefor the least expensive to buffer. The overall length of the pixel pipeline can be effectively changed by growing or shrinking the *u*, *v* FIFO. Counting two stages of the *u*, *v* FIFO, the total number of pipeline stages is 18.

While stretching out the texture mapping data path lowers its cost, it complicates the design elsewhere. Normally, color (r,g,b, and *alpha*) and depth (z) would be interpolated at the same time as the frame buffer address and the texture coordinates. One would naively expect to buffer the interpolated color and depth values in a FIFO that matches the length of the texture pipeline. However we find it much less expensive to defer interpolation until just-in-time. In this way the circuit can buffer only the *r*, *g*, *b*, *alpha*, and *z* starting points and differences in a compact buffer.

Pixels are not flushed between spans, but are flushed between polygons to insure that the texture map remains valid for pixels still progressing through the pipeline. We note, however, the a

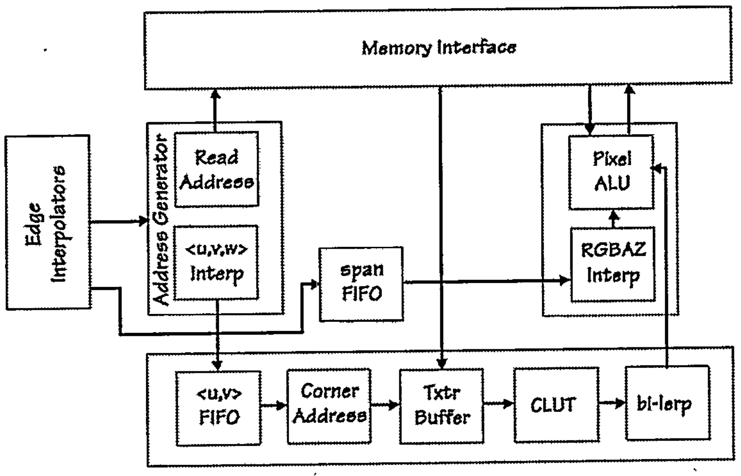


Figure 2. Rasterizer block diagram.

pixel is considered flushed when its write is queued, not when it is actually written into the frame buffer.

5. Simulation

While a stretched pipeline sounds good in principle, a simulator is the best verification of the design. There have been two simulators built for this project. One is a Verilog model of the actual circuit and the other is a functional simulator written in C. The simulations reported here come from the functional simulator.

For the functional simulator, memory and rasterizer timing is limited to horizontal spans. The C program which serves as a wrapper for the simulator makes no attempt to mimic the timing of the edge interpolators. We find that this is sufficient because the design includes enough edge interpolators to insure that no part of the span processor will stall except on the shortest spans and this stall is explicitly mimicked. The memory model keeps track of its own state autonomously and propagates the appropriate time penalty for data reads and writes.

In this particular design the least expensive memory configuration has more peak bandwidth than the rasterizer chip and its application require. While we are primarily concerned with insuring that the rasterizer circuit does not stall, we also attempt to make the largest transfers possible to and from memory.

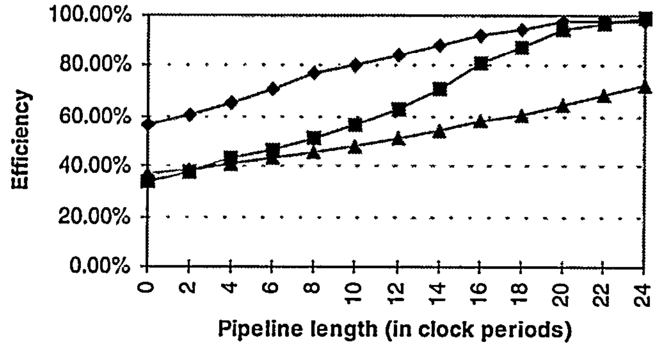
Our efficiency figure is the ratio of actual pixel fill rate to peak fill rate. As the plots in Figures 3 and 4 show, stretching the pipeline to match memory latency allows the circuit to reach its peak rate. The top line in each plot, denoted by diamond shaped data points, indicates the efficiency when the maximum memory transfer size is 64 bytes. The bottom line, denoted by triangle shaped data points, demonstrates how badly performance can suffer if the z buffer and color fall into separate memory pages. The plot in the middle, denoted by square shapes, shows how performance is degraded when transfer lengths are limited to 8 bytes. Not surprisingly, this limitation is less of a problem for the short spans of small polygons than the longer spans of large polygons.

The scenes in Figures 3 and 4 are obviously contrived with the first containing 4 triangles which cover approximately 12,000 pixels each. Figure 4 contains 2016 triangles, each of which covers approximately 18 pixels. Real game applications contain a mix of large and small polygons, but these examples are intended to show the extremes.

These plots are not entirely accurate since the periodic interruption of video refresh is disabled to make the result more understandable. Furthermore, the examples reuse a single texture map for all polygons, so the overhead of paging texture into SRAM is not accounted for. Nevertheless we can see several trends. As expected, stretching the pixel pipeline to match latency yields a dramatic improvement in efficiency in all circumstances. Arranging data in memory so that page misses are reduced is also a major performance win. Lastly, increasing the size of each transfer yields a marginal improvement, at least when rendering large polygons.

Given the technology and timing used in the example circuit, a pressing question is how well does this design principle fare for faster clock rates. Running the same tests used in figures 3 and 4 through a simulator set for 100 MHz, with a pipeline stretched out twice as long, yields 94% and 72% utilization of the rasterizer circuit for the large polygon case and small polygon case respectively. The argument against excessive extension for this design is that the pipeline is flushed at the end of each polygon. Small polygon performance will suffer, as the simulation indicates.

The goal and, hopefully, the result of this simulation is to gain some guidance for future designs which make use of high latency memory. Whether effective systems can be made depend on factors beyond just the circuitry of the rasterizer, e.g. memory contention, content tuning, etc., but we feel safe in concluding that high memory latency is not an insurmountable impediment to the design of fast, low-cost rasterizers.



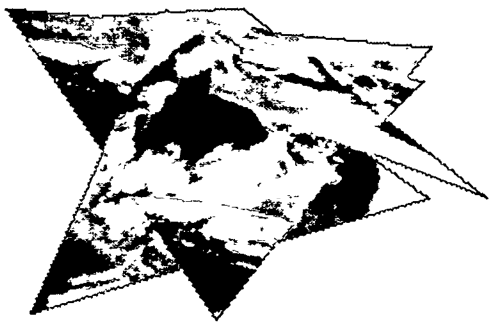
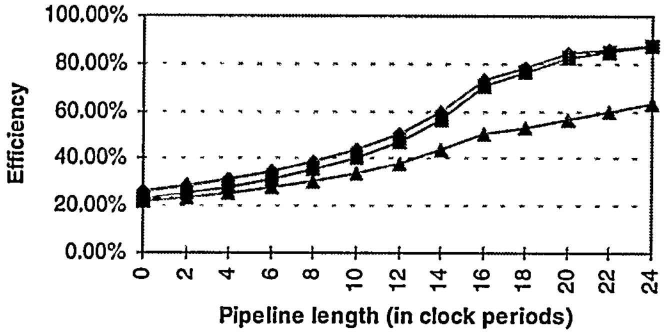


Figure 3. Rasterizer efficiency as a function of pipeline length for random large polygons.



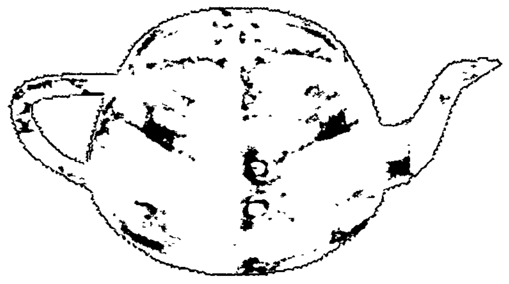


Figure 4. Rasterizer efficiency as a function of pipeline length for meshes of small triangles.

6. References

[1] Akeley, K., and T. Jermoluk. "High-Performance Polygon Rendering," Computer Graphics, 22 4, August 1988, pp. 239-246.

[2] Carpenter, Loren C., et al, "The A-buffer, an Antialiased Hidden Surface Method," in Proceedings of SIGGRAPH 84, pp. 103-108.

[3] Clark, J. H., and M. Hannah. "Distributed Processing in a High-Performance Smart Image Memory, Lambda 1,3, 4th Quarter 1980, pp. 40-45.

[4] Deering, M. F., S. A. Schlapp, and M. G. Lavelle, "FBRAM: A new Form of Memory Optimized for 3D Graphics," *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM Siggraph, pp. 167-174.

[5] Fuchs, H., and B. Johnson. "An Expandable Multiprocessor Architecture for Video Graphics," Proceedings of the 6th ACM-IEEE Symposium on Computer Architecture, April 1979, pp. 58-67.

[6] Goris, A, B. Fredrickson, and H. Baeverstad, "A Configurable Pixel Cache for Fast Image Generation," IEEE *CG&A* 7, 3 (March 1987), pp. 24-32.

[7] Molnar, S., J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics,* 26, 2 (July 1992), ACM Siggraph, pp. 231-240.

[8] Torborg, Jay, and James T. Kajiya, "Talisman: Commodity Realtime 3D Graphics for the PC," in Proceedings of SIGGRAPH 96, pp. 353-363.

[9] *Red Planet,* FASA/Virtual Worlds Entertainment.

[10] Whitton, M., "Memory Design for Raster Graphics Displays," IEEE *CG&A* 4, 3 (March 1984), pp. 48-65.