Reducing Latency on PixelFlow

Anselmo A. Lastra
University of North Carolina
Chapel Hill, N.C., USA
lastra@cs.unc.edu

ABSTRACT Performance, as measured by the number of primitives rendered per second, has been the most important rendering system design consideration while latency, the amount of time it takes to render an image, has largely been ignored. This is because a moderate amount of latency is not an issue for traditional interactive systems controlled by joysticks. However, latency has emerged as a major consideration when rendering for immersive systems, especially those using head-mounted displays. Maintaining low overall system latency is very important to create an illusion of presence in a virtual environment and very significant contribution to total system latency is the time it takes to generate an image. This paper examines some possible ways to reduce latency in the PixelFlow graphics computer.

We first describe the standard rendering software for PixelFlow, and derive an expression for the time to render an image. We then propose two alternative software systems for the PixelFlow hardware, and derive expressions for the rendering latency. When we compare latencies for some common application scenarios, we find that the two proposed systems render with lower latency than the standard high-throughput system, but find that the benefits are not enough to outweigh the costs.

1 The Latency Problem

In an immersive virtual environment, the time from the beginning of the head-position measurement until the proper image is presented to the user is referred to as the system *latency*. Low latency is critical to maintaining the illusion of presence in a virtual world. High latency is especially disturbing for *augmented reality* systems — systems that superimpose computer-generated images with a view of the real world. Since the user has a frame of reference for the position of virtual objects, it is apparent when these objects do not stand still. In our lab, we have described the latency-induced behavior of objects as "swimming". The latency that we are discussing in this paper, that caused by the image generator, is only part of the total system latency. Other factors include scanout of the image, tracking, and application overhead.

Researchers [1, 6] have used prediction to try to reduce apparent latency by rendering not the user's viewpoint as determined by the tracker, but rather the user's viewpoint as it *will be* when the image is scanned out. However, prediction is not a panacea. As shown by Azuma [2], if the system latency is greater than about 80 ms, he could not accurately predict the user's head position. An explanation for this phenomenon is that the dynamics of a person's head govern the maximum rate of change. During a short period of time, position can not change abruptly and unpredictably. However, beyond that short period, movement is much less predictable. Post-processing of the rendered image [12] by panning or warping is another possible way to reduce apparent latency. We may want to incorporate those techniques into a system, but will not consider it as part of our latency analysis.

This paper evaluates the latency of various proposed software architectures for the PixelFlow [9] graphics computer. We first review the assumptions used in our performance analyses, and briefly present an overview of the hardware architecture. We then detail and evaluate the basic software architecture — one designed to maximize throughput of polygons, not to minimize latency. We then examine two alternative architectures designed to reduce latency. Finally, we make some observations about the applicability of the three rendering strategies.

2 Application Scenario

For this analysis, we are assuming a system using prediction of head position. We would like to be able to render a frame that is correct at some time during the scanout process, say halfway down the display. To do this, we have to begin rendering some time in advance and assume that we will complete the image before vertical retrace. We do not actually have to finish rendering the whole image, of course. Data for scan lines just have to be ready at the time they are to be scanned [11]. However, this is difficult to accomplish properly, so for this analysis we will assume that we completely render the image before vertical retrace. We would like to keep overall system latency below 80 ms, so a target of 30 to 40 ms for the image generation latency is reasonable. For some applications, such as experiments in predictive tracking, we'd like to generate simple images with very low latencies, therefore the performance region well below 30 ms is of secondary interest.

Since the number of primitives that are actually rendered is scene dependent, it is difficult to evaluate the true rendering time. In actual usage, assuming that we do not wish to render the complete model, we may want to use a working rendering time lower than worst case, and be prepared to miss a frame occasionally. However, the worst-case assumption is that the

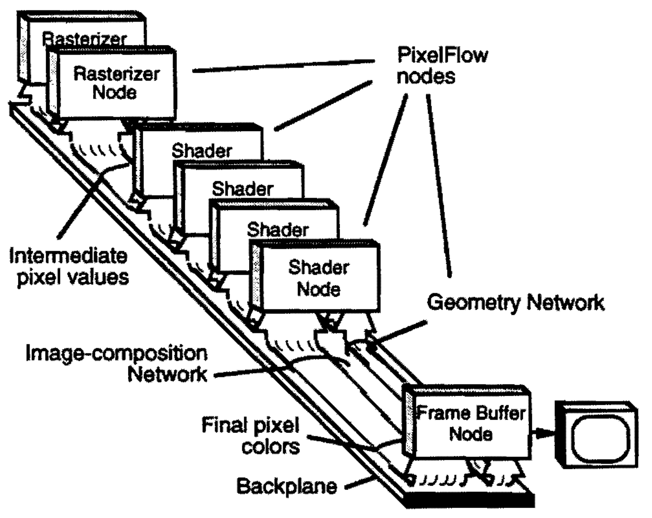

Figure 1. Block diagram of the PixelFlow system.

whole model is being rendered. For this analysis, we will compute the worst-case latency.

We will also assume that the target application uses very little display-list hierarchy. We have found that the use of a deep hierarchy can greatly increase the rendering time because a significant portion of the computational resources of the geometric pipeline stage can be consumed in traversal and matrix manipulation. Users in our laboratory who are concerned with maximum performance, be it rendering or latency, have structured their applications with fairly shallow hierarchies, so this assumption, while not desirable, is representative of a class of applications. We may, for a future experiment, simulate a deeper hierarchy by modeling it directly or assuming that the geometry processing overhead is large.

Although some experimental, high-resolution head-mounted displays are becoming available, the vast majority of current HMDs are driven at NTSC resolutions (the actual display resolution is usually much worse). We will assume a 640 by 512 pixel display, stereoscopic or monoscopic, for our performance models. Ideally, we would like to render anti-aliased images. However, it will cost us some performance. We will examine the latency with and without antialiasing.

3 PixelFlow Hardware Architecture

The PixelFlow architecture [9] is a scaleable architecture for interactive rendering based on the idea of image composition. In an idealized image composition system, complete images are rendered on a set of graphics nodes, each of which is assigned only a portion of the primitives. Depth information is saved for each pixel, along with color. The resulting images are depth composited to yield the final image. As we shall see, there are variations on this simple method. Figure 1 schematically illustrates a PixelFlow system. In this section, we will describe the system briefly and only in enough detail for the discussions to follow. More information is provided by Molnar, et. al. [9].

All PixelFlow nodes are identical, and each is a complete rendering system. Figure 2 is a block diagram of the components on a node. Two Hewlett-Packard PA-RISC 7200 modules (which we refer to as geometry processors or GP), sharing 64 MB of memory, are used for the display list traversal and geometric calculations of the graphics pipeline. This stage of the pipeline is known as the *front-end* [8]. Rasterization and shading (the *back-end* of the pipeline) are performed on a 128 by 64 array of 8-bit pixel processors, each with 256 bytes of local memory, and 128 bytes of communication registers (which may also be used as local memory when not needed for communications). Specialized memory for image-based texturing is accessible from the pixel processors.

The composition network operates at the basic clock speed of 100 MHz, but can send data on both edges of the clock, so the transfer rate is actually 200 MHz. It also operates in both directions so, if the system is configured carefully, we can achieve a maximum of twice the performance. The network is 256 data bits wide. The amount of time to perform a depth composition on one byte of data for every pixel of the processor array is:

$$t_{comp\_byte} = Transfer\_Time\_per\_Byte \cdot Pixels\_per\_Region$$
$$= 1.28\ \mu s$$

In the performance analyses, we will multiply that time by the number of bytes of data used by the algorithm, for example three for 24-bit color. There is also a short, fixed setup time per composition that is dependent on the number of nodes in the system. It is short enough that we can ignore this time in our analysis.

Rendering instructions for the pixel processors are produced by the PA-RISC microprocessors, and placed into GP memory. A DMA controller is used to fetch the instructions from GP memory as needed by the pixel processors. We use GP memory (64MB) rather than a hardware FIFO to store these instructions because we want to be able to buffer a large number, up to two complete frames, of rasterization commands.

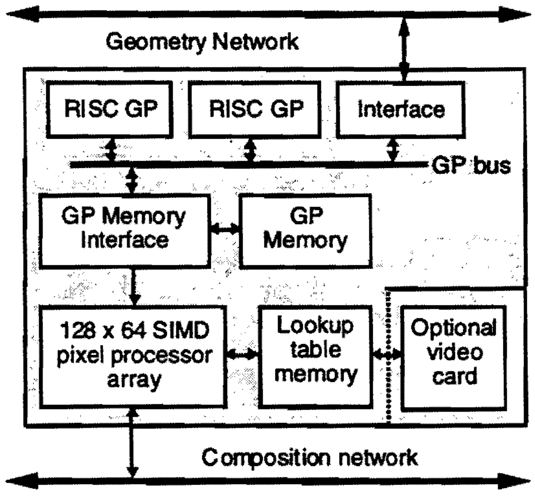

Figure 2. PixelFlow node block diagram.

Instead of providing a separate frame buffer for video scanout, one of the PixelFlow nodes serves as the frame buffer. Any of the PixelFlow nodes can be fitted with a daughter card for scanout. That card has access to texture memory which serves as the actual frame buffer. We expect a variety of video cards for different formats.

For virtual environments, we will often want to render stereo frames. We can do this in one of two ways. We can render them sequentially. However, since we would like to reduce latency, a better way to render two frames on PixelFlow may be to logically divide the machine in half and render the frames concurrently on the halves. This is especially true if we want to reduce latency to below a field time. Note that we typically plan to maintain the display lists in GP memory, so data transfer from the host processor is not an issue.

Performance Estimates. We are currently in the design and construction process for PixelFlow. We can accurately model, on a clock cycle basis, the performance of the pixel array and the composition network, but the front-end, basically consisting of commercial microprocessors, is not modeled to that level of detail. The values for the various execution times used in the following sections were obtained as follows:

Geometric operations - by benchmarking on existing workstations. Since the clock speed is higher and the memory subsystem on PixelFlow is more aggressive than that of the workstations, we expect somewhat better performance on PixelFlow.

Rasterization and shading - obtaining clock-cycle counts of code running on simulators.

Image composition - computed from the composition network design and target clock speeds.

Overhead - estimated from measurements made on Pixel-Planes 5. These are the data most likely to be wrong. However, sensitivity of the final conclusions to errors is low.

4 PixelFlow Software Architecture

This section describes the software that we are developing as the standard rendering system for PixelFlow, and serves as the base system for which we would like to evaluate the latency, and as a point of comparison with alternative methods. We describe the system only in enough detail to allow the reader to understand how we model the latency. More detailed explanations are provided in [7] and [9].

In order to increase performance by reducing the amount of computation, we use the technique of *deferred shading* [4, 13]. This term refers to shading only the final, visible image, not the individual primitives as they are being rasterized. Instead of directly computing color during rasterization, we store *appearance parameters* [3] such as surface normals, intrinsic color, texture coordinates, etc. These appearance parameters are z-buffered and shading is only performed when all of the primitives have been rasterized. The obvious performance advantage, of course, is that pixels that will not be visible in the final image are not shaded.

A problem with deferred shading is that it potentially requires very large amounts of frame-buffer memory in order to hold the appearance parameters. We reduce these memory requirements by using a *virtual buffer* [5]. Instead of completely instancing the frame buffer, we divide it into fixed-size screen regions, and work on the regions one after another. Therefore, we only need to have enough frame-buffer memory to hold the appearance parameters for one screen region. This implies that we must first bucket sort the primitives [8] by screen region. As we shall see, this has an effect on latency because we must traverse all of the frame's primitives before doing any rasterization.

To rasterize, composite the appearance parameters, and shade the resulting visible image, we divide the PixelFlow nodes into three functional types: a set that performs rasterization, a set for shading, and a frame buffer. This functional division is illustrated in figure 1. The rasterization nodes rasterize one screen region at a time, producing z and appearance parameters. These pixel parameters for a screen region, from all of the rasterization nodes, are depth composited, and the parameters for the visible pixels are delivered to one of the nodes responsible for shading. That shading node shades and textures all of the pixels in that region, and the resulting color is sent over the composition network to the frame buffer. Appearance parameters for screen regions are assigned to shading nodes in a round-robin fashion to achieve parallelism in shading as well as in rasterization.

A straightforward implementation of this system would have the rasterization stage of the pipeline idle while geometric operations are being performed, and the geometric stage of the pipeline idle while rasterization and shading are performed. To increase throughput (by utilizing the hardware as much as possible), we pipeline complete frames. One frame is being rasterized while the next frame is being transformed. Two buffers in GP memory hold rendering instructions. One is being filled by the geometry processing, while the other is being emptied for the rasterization.

Since geometric and pixel processing are completely separate parts of the pipeline, total rendering system latency is the sum of the front-end and the back-end processing times

$$Latency = t_{front} + t_{back}$$

We can express the front-end processing time as

$$t_{front} = t_{overhead} + \frac{t_{trans} \cdot n}{n_{rast}}$$

where n is the number of primitives, n_{rast} is the number of nodes devoted to rasterization, t_{trans} is the time perform all of the geometric operations for a primitive (the actual transformation, plus trivial reject, clipping, etc.), and the $t_{overhead}$ term covers the frame startup costs. Note that t_{trans} includes the bucket sorting of the primitive (in our code it is a tightly coupled part of the geometry processing routine). We are treating the two PA-RISC processors as one geometry engine, the mode in which we will use them. The rasterization time for all primitives is divided by the number of rasterization nodes to account for the effect of parallelism (we are assuming perfect load balancing). We measured t_{trans} as 1.85 µs, and estimate $t_{overhead}$ to be 0.1 ms.

Analysis of the back-end computation is more complex. Several rendering tasks are being pipelined: rasterization, composition, shading, transfer of the resulting image (color) to the frame-buffer node, and copy of the image to the frame-buffer memory. The back-end latency of the system is going to be primarily whichever of these tasks takes the longest to complete. A smaller component is going to be the pipeline filling and emptying which is negligible because we are processing many screen regions. Therefore

$$t_{back} = max(t_{rast}, t_{comp}, t_{shade}, t_{load})$$

where t_{rast} is the time to rasterize all of the primitives, t_{comp} is the time to composite the image, t_{shade} is the time to shade the complete image, and t_{load} is the time to write the image to the frame buffer.

To derive an expression for rasterization time, we must consider the fact that primitives can span more than one screen region. An expression for computing the number of screen regions that a primitive is expected to cover as a function of primitive and region size was derived by John Eyles to analyze performance of Pixel-Planes 5. The derivation is detailed in [10]. We call this the *bin-replication factor* or *overlap*:

$$Overlap = \frac{(w+W)}{W} \cdot \frac{(h+H)}{H}$$

where w and h are the width and height of an average primitive, and W and H are the width and height of a screen region. For primitives with 10 by 10 bounding boxes, and our screen region of 128 by 64 pixels, the overlap is 1.25. For eight-sample anti-aliased rendering, the overlap is 1.52.

The rasterization time is

$$t_{rast} = \frac{t_{rast\_per\_prim} \cdot n \cdot Overlap}{n_{rast}}$$

where $t_{rast\_per\_prim}$, which we compute to be 0.83 µs, is the time to rasterize a single primitive, and n is the number of primitives.

Composition time is just the time we derived in the previous section to composite a single byte of data for a complete region, t_{comp_byte}, times the number of bytes used for appearance parameters, multiplied by the number of regions. We will need a total of 21 bytes of shading parameters for our examples (4 for z, 6 for surface normals, 3 for color, 4 for u and v, 2 for texture ID, and 2 for texture scale), therefore

$$t_{comp} = n_{regions} \cdot t_{comp\_byte} \cdot Bytes\_of\_App\_Parameters$$

or

$$t_{comp} = n_{regions} \cdot 26.88\ \mu s$$

The main components of the shading time are the texture memory address computation and the lookup of the mip-map texture from memory. Address computation takes 15.4 µs. The time to read a texture depends on the number of pixels that are to be textured in the current region. We will use the worst-case time, a lookup for all of the pixels, which is 95 µs. Lighting computations are pipelined with the texture lookup. We need to consider the time to shade all of the regions, divided by the number of PixelFlow nodes devoted to shading, or

$$t_{shade} = \frac{n_{regions} \cdot 110.4\mu s}{n_{shaders}}$$

Loading color to the frame buffer is

$$t_{load} = n_{regions} \cdot 52\ \mu s = 2.08\ ms$$

which is short enough that it is not a factor when rendering any significant number of primitives.

5 Reducing Latency by Eliminating Frame Pipelining

Much of the latency in the previous system is caused by the large granularity of the pipelining — two full frames. Can we reduce the latency by not pipelining frames, while still maintaining adequate performance? The coarse granularity of the pipelining was due to our need to sort the primitives based on their screen-space position. Can we improve this to reduce the amount of pipelining?

One technique that we have identified is to make a separate pass over all of the primitives to sort them into screen regions, but doing as little work as possible. The minimum amount of work we have to do is to transform the vertices and perform a bounding box test. After the sorting pass, a second pass is made to generate the rasterization instructions. These are sent to the pixel-processor array one region at a time rather than one frame at a time. We will have to write the screen-space coordinates of the vertices into memory during the first pass because the second pass will not be in traversal order, but in screen-space order. A throughput penalty is that the pixel processors may have to be kept idle during the sorting pass, so there is lower resource utilization.

However, the biggest disadvantage of pre-sorting is that we are visiting the display list data more than once. As we are all aware, a bottleneck of modern microprocessors is the bandwidth to main memory. Part of the improvement in processor speed that we have seen is due to the extensive use of chaching, thus reducing the access to main memory. This fact shows us that we are likely to pay a penalty for this extra memory traffic.

The latency of this system is the time it takes to make the separate sorting pass, t_{first_pass}, plus the time to complete the rest of the front-end processing, and the back-end processing, t_{second_pass}.

$$Latency = t_{first\_pass} + t_{second\_pass}$$

The first pass consists of some frame startup overhead, plus some processing for each primitive.

$$t_{first\_pass} = t_{overhead} + \frac{t_{sort} \cdot n}{n_{rast}}$$

We benchmarked t_{sort} to equal 0.74 µs. The second pass latency is the maximum of the time it takes to complete the front-end computations, the rasterization, composition, shading and copy to the frame buffer. We will call the second-pass geometry work, generating the rendering commands, t_{gen}.

$$t_{second\_pass} = max(t_{gen}, t_{rast}, t_{comp}, t_{shade}, t_{load})$$

The times t_{rast}, t_{comp}, t_{shade}, and t_{load} are the same as in the previous system. The time to generate rendering commands is

$$t_{gen} = \frac{t_{gen\_each} \cdot n}{n_{rast}}$$

We benchmarked t_{gen_each} as 1.45 µs.

6 Completely Instanced Frame Buffer

The deferred shading and virtual buffer techniques used in the previous designs serve to increase throughput and enable high-quality shading, but result in a fairly large minimum latency. Even if we are rendering only one primitive, we still incur the complete shading time and the sorting step increases latency. What if we want a very low minimum latency design, especially important for predictive tracking experiments, and for augmented reality applications? What we would like to do is to eliminate the deferred shading, and pipeline the front-end computations with the back-end at a fine granularity. This section presents the design for a simple, non-texturing system with a completely instanced frame buffer similar to the Slats system [11].

This system eliminates bucket sorting. Primitives are rasterized in the order in which they are traversed. As with most rendering systems, we can light the vertices on the front-end and interpolate color. We could potentially texture each primitive as it is being rasterized. However, we do not have any experience with this process and expect it to be expensive, so will not consider texturing in our analysis of this system. The final steps in the rendering of the frame are to composite the colors, deliver the result to the frame-buffer node, and write the image to memory for scanout.

The determining factor in our ability to realize this design is the desired size of the display since it determines frame-buffer needs. Do we have enough memory on the pixel processors? We need about 4 bytes for *z* and 3 bytes for color — a total of 7 bytes per display pixel. For our low-resolution example, we need 640 by 512 pixels, or 40 screen regions, for a total of 280 bytes. Recall that each pixel processor has 256 bytes of local memory plus 128 bytes of communications registers. Half of the communications register space is used to access texture memory. The other 64 bytes consists of two sets of composition-network transfer buffers, one for each direction. Since we are not compositing while rendering, we can use those 64 bytes as general purpose memory. That leaves us with a total of 384 bytes available, plenty for our purposes.

Note that this system differs from the previous ones in two fundamental ways, thus making direct performance comparisons difficult. First, it does not defer shading. Second, it does not texture. Unfortunately, we can not separate the two cases because PixelFlow was not designed for texturing during scan conversion. However, analysis of this design is valuable to see what performance we can achieve when we sacrifice texturing.

The latency of this system is the time it takes to completely render all of the primitives for the first antialiasing subsample, the times it takes just to rasterize the subsequent subsamples (we will store the rendering instructions in memory and reuse them for the other subsamples), the time it takes to composite all but the final subsamples, plus, for the last sample, the maximum of the composition and frame-buffer loading times (since the latter steps are pipelined).

$$\begin{aligned} Latency = {} & max(t_{front}, t_{rast}) + (n_{samples} - 1) \cdot (t_{rast} + t_{comp}) \\ & + max(t_{comp}, t_{load}) \end{aligned}$$

Note that antialiasing is very different for this design. The previous two systems performed supersampling in one pass, whereas this one takes multiple rasterization and composition passes. As we shall see, anti-aliased performance suffers.

The expressions for the front-end processing time and for rasterization are the same as for the first system (although the values of the constants change):

$$t_{front} = t_{overhead} + \frac{t_{trans} \cdot n}{n_{rast}}$$

$$t_{rast} = \frac{t_{rast\_per\_prim} \cdot n \cdot Overlap}{n_{rast}}$$

We benchmarked t_{trans} as 0.87 μs and measured $t_{rast\_per\_prim}$ as 0.62 μs. The composition time

$$t_{comp} = n_{regions} \cdot t_{comp\_byte} \cdot Bytes\_of\_Color = 360\ \mu s$$

is very short, since we are only sending color.

7 Conclusions

Using the expressions that we have derived, we evaluated the performance of the three systems, the base system, pre-sorting system, and non-texturing system, using a medium-sized configuration of 20 PixelFlow nodes (one of which is a frame-buffer node). We examined both single-sample, and sub-sampled antialiasing modes.

Figure 3 shows the latency of the three systems with antialiasing enabled. Two machine configurations are used for the systems that are texturing. One configuration devotes four of the nodes to shading (leaving fifteen for rasterization), while the other configuration uses only one node for shading. The non-texturing system is a poor performer, both on minimum latency with only a few polygons, and at our latency target region of 30 to 40 ms. This is due to the fact that the necessary technique for antialiasing, multi-passes of the rasterization, is very inefficient when compared to the single-pass antialiasing possible with more available pixel-processor memory on the other two designs.

For the texturing systems, using more nodes as shaders reduces the minimum latency when very few polygons are being rendered. This is because, initially, all of the frame time is due to the fixed cost of shading. Once enough polygons are being rasterized, the rasterization costs dominate the latency. For our target latency range of 30 to 40 ms, we get much better performance using only one node for shading and the others for rasterization. The crossover point will shift if we require higher-quality, more time-consuming shading, as we expect for some applications [7].

There is also a latency performance advantage to pre-sorting, at the cost of some throughput of course. However, it was surprising to us how little the performance increases for any give latency. For example, in the region of interest between 30 and 40 ms, the difference amounts to only 10 to 15 percent. It is unlikely that this small increase in performance would be worth the effort of creating and maintaining a pre-sorting system. Given the slopes of the curves, the difference will increase and make the pre-sorting system look better, but at latency ranges that are not of interest to us.

Figure 4 shows the same configurations, but without antialiasing. When not antialiasing, pre-sorting does not show a performance advantage because the extra cost of the first pass offsets the elimination of the pipelining. If texturing is not required, the simple non-texturing system has a large latency performance advantage. Not only is it rendering very simple triangles, but the geometric and rasterization stages are completely pipelined. The only other cause of latency is the copy to the frame buffer. At a latency of 30 ms, the system should be able to render over 600,000 triangles per frame.

We could reduce the latency even further. If using an interlaced display, such as NTSC, one can render only fields at a time instead of full frames [11]. This reduces the minimum latency but mandates the generation of an image every field (60 Hz for NTSC). If the rendering of a frame takes longer than the planned time, you have to adopt some contingency strategy, as is done on some flight simulators. Rendering fields is a useful technique if we need to reduce latency below our target of 30 to 40 ms. It's likely that it would be a good addition to the simple, non-texturing system.

The standard, highly pipelined implementation suffers from somewhat higher latency than the pre-sorting system, but also provides maximum throughput. At the region of interest, the pre-sorting system reduces the latency, but not enough to make it very interesting. The simple, Gouraud-shading system provides the maximum performance when antialiasing and texturing are not required.

Acknowledgments

I am grateful to the reviewers for their excellent suggestions. Thanks to the PixelFlow development team for many valuable discussions, and especially to Steve Molnar for his input, and to Jon McAllister, Lawrence Kesteloot and Lee Westover for some performance timings. We thank Hewlett-Packard for their generous donations of equipment.

This research is supported in part by the Advanced Research Projects Agency, ARPA ISTO Order No. A410 and the National Science Foundation, Grant No. MIP-9306208.

References

[1] Azuma, R. and G. Bishop: Improving Static and Dynamic Registration in an Optical See-through HMD, *Proc. Siggraph 94*, 197-204.

[2] Azuma, R.: Predictive Tracking for Augmented Reality, *Ph.D. Dissertation*, University of North Carolina, Chapel Hill, N.C., 1995 (also available as UNC CS Technical Report TR95-007).

[3] Cook, R. L.: Shade Trees, *Proc. Siggraph 84*, pp. 223-231.

[4] Deering, M., S. Winner, B. Schediwy, C. Duffy, and N. Hunt: The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics, *Proc. Siggraph 88*, 21-30.

[5] Gharachorloo N., S. Gupta, R. F. Sproull and I. E. Sutherland: A Characterization of Ten Rasterization Techniques, *Proc. Siggraph 89*, 355-368.

[6] Liang, J., C. Shaw, and M. Green, On Temporal-Spatial Realism in a Virtual Reality Environment, *Proc. ACM Symp. on User Interface Software and Technology*, 1991, 19-25.

[7] Lastra, A., S. Molnar, M. Olano, and Y. Wang: Real-Time Programmable Shading, *Proc. 1995 Symp. on Interactive 3D Graphics*, 59-66.

[8] Molnar S., and H. Fuchs: Advanced Raster Graphics Architecture, in Foley, J., van Dam, A., Feiner, S., and J. Hughes, *Computer Graphics: Principles and Practice, 2nd ed.*, Addison-Wesley, 1990, 855-922.

[9] Molnar S., J. Eyles, and J. Poulton: PixelFlow: High-Speed Rendering Using Image Composition, *Proc. Siggraph 92*, 231-240.

[10] Molnar, S., M. Cox, D. Ellsworth, and H. Fuchs: A Sorting Classification of Parallel Rendering, *IEEE CG & A*, **14**(4), July 1994, 23-32.

[11] Olano, M., J. Cohen, M. Mine, and G. Bishop: Combatting Rendering Latency, *Proc. 1995 Symp. on Interactive 3D Graphics*, 19-24.

[12] Regan, M., and R. Pose: Priority Rendering with a Virtual Reality Address Recalculation Pipeline, *Proc. Siggraph 94*, 155-162.

[13] Tebbs, B., U. Neumann, J. Eyles, G. Turk, and D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading", UNC CS Technical Report TR92-034.

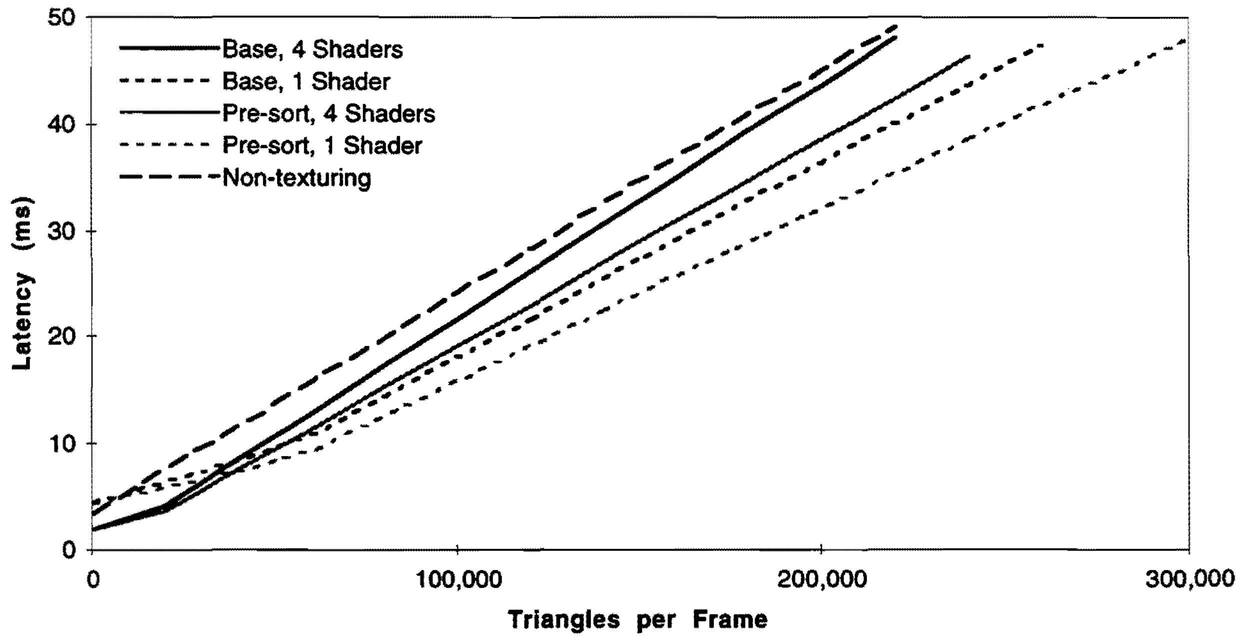

Figure 3. Supersampling performance of the three systems. The two systems which use deferred shading are shown in two configurations, with one and four shading nodes.

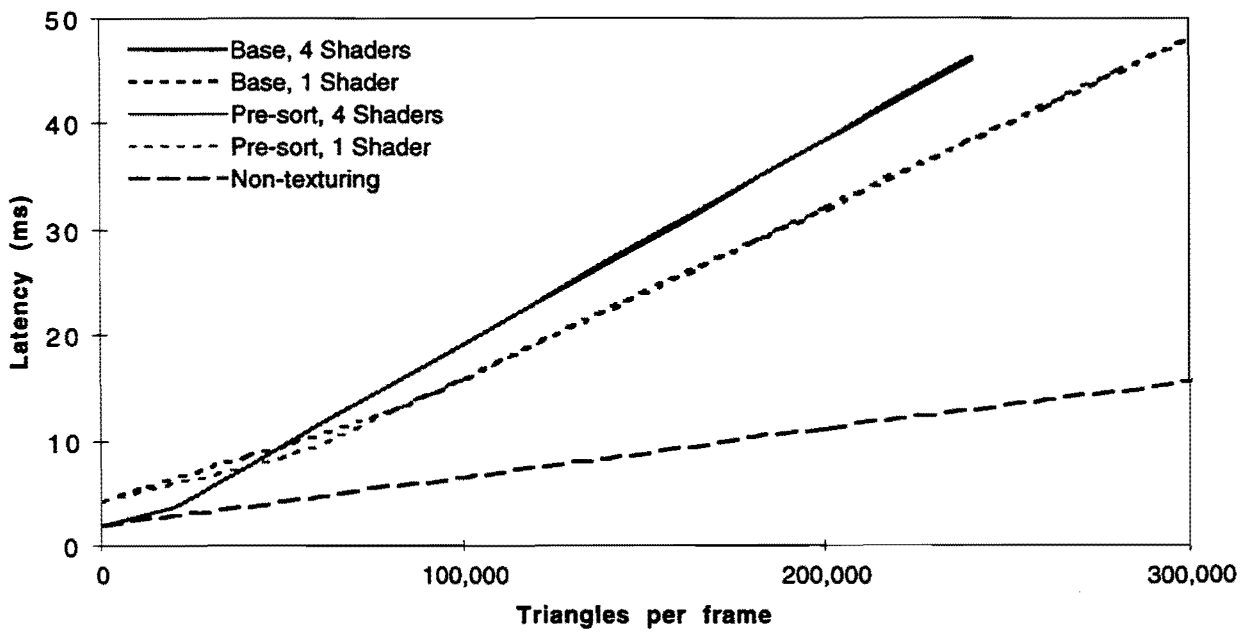

Figure 4. Single sample performance of the three systems. The two systems which use deferred shading are shown in two configurations, with one and four shading nodes.