Pixel merging for object-parallel rendering: a distributed snooping algorithm

Article · January 1993

DOI: 10.1145/166181.166188

Michael Cox
NVIDIA

Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm

Michael Cox
Pat Hanrahan

Princeton University*

Abstract

In the purely object-parallel approach to multiprocessor rendering, each processor is assigned responsibility to render a subset of the graphics database. When rendering is complete, pixels from the processors must be merged and globally z-buffered. On an arbitrary multiprocessor interconnection network, the straightforward algorithm for pixel merging requires $\bar{d}A$ total network bandwidth per frame, where $\bar{d}$ is the depth complexity of the scene and A is the area of the screen or window. This algorithm is used by the Kubota Pacific Denali and appears to be used by the Evans and Sutherland Freedom series. An alternative algorithm, the PixelFlow algorithm, requires nA network bandwidth per frame, where n is the number of processors. But the merging is pipelined in PixelFlow so that each network link must only support A bandwidth per frame. However, that algorithm requires a separate special-purpose network for pixel merging. In this paper we present and analyze an expected-case $\log(\bar{d})A$ algorithm for pixel merging that uses network broadcast, and we discuss the algorithm's applicability to shared-memroy bus architectures.

CR Categories and Subject Descriptors: B.4.2 [**Input/Output and Data Communications**]: *Input/Output devices – image display*; C.1.2 [**Processor Architectures**]: *Multiprocessors – interconnection Architectures*; F.2.m [**Analysis of Algorithms and Problem Complexity**]: *Miscellaneous*; I.3.1 [**Computer Graphics – Hardware Architecture**]: *Parallel processing*, I.6.4 [**Simulation and Modeling**] *Model validation and analysis*.

General Terms: Algorithms, Architecture, Interconnection Networks, Graphics, Rendering, Systems.

Additional Key Words and Phrases: pixel merging, object-parallel rendering, cache coherency, snoopy cache.

1 Introduction

In addition to the vertical parallelism (a.k.a. operation parallelism) that is exploited by traditional graphics hardware pipelines [1], there is also extensive horizontal parallelism (a.k.a. data parallelism) available. In particular, subsets of the graphical objects to be rendered, *or* regions of the screen, can be divided among the processors of a multiprocessor. We call the first form of parallelism *object parallelism* and note that it partitions the input to the rendering process. We call the second form *image parallelism* and note that it partitions the output of the rendering process. The use of image-parallel rendering has been both explored and implemented by [1], [6], [8], [9], [22], [28], and [30]. Purely object-parallel approaches have been explored by [17], [18], [19] [24], [26], and [29]. Purely object-parallel rendering requires apparently overwhelming additional network bandwidth and pixel processing. Since different processors may produce pixels at the same screen (i.e. x, y) location, global z-buffering is required to combine the pixels from each processor into a complete scene. We refer to this as the *pixel merge problem.*

In this paper we present an algorithm for pixel merging when network broadcast is available, for example on a shared-memory bus multiprocessor. We show that in the expected-case, the algorithm is very network efficient (with respect to bandwidth), and does not require a special-purpose network for pixel merging. We compare this algorithm with two other algorithms for pixel merging, a "straightforward" algorithm used by Kubota Pacific [15] and apparently used by Evans and Sutherland [7] in recently introduced architectures, and the merging algorithm used in the PixelFlow machine [19].

Below, we first digress to review data gathered from instrumentation of several rendering packages, and to discuss the use of these data in trace-driven simulations of object-parallel architectures. Following this we define some terms, and then begin by describing and analyzing the straightforward algorithm, and the PixelFlow algorithm, for pixel merging. We then analyze uniprocessor z-buffering in order to motivate the presentation and analysis of a "distributed snooping" pixel merging algorithm.

2 Rendering traces and trace-driven simulations

We have instrumented several graphics packages, *RenderMan* from Pixar [27], and the graphics library *GL* from Silicon Graphics, Inc. [25]. Pixel traces are generated by placement of library calls into the rendering package at the sites at which pixels are generated, and before they are z-buffered. Thus, we trace every pixel that is generated by the graphics package. The library calls write compressed pixel records to trace files that can be later processed for statistics, and used as input to drive simulations of rendering in object-parallel architectures. Each pixel comprises roughly the following fields: screen-x, screen-y, eye-coordinate-z, red, green, blue, and parent graphics primitive (i.e. the primitive whose scan

*Department of Computer Science, 35 Olden St., Princeton, NJ 08540.

0-8186-4920-8/93 $3.00 © 1993 IEEE

Trace	Primitives			Total pixels	Screen area	Average depth, $\bar{d}$
	(pre-clipping)	(post-clipping)	average size			
Bike	16144	9618	311.66	2997538	1258408	2.38
Cube	48000	47459	13.61	6459105	806520	8.01
Zinnia	N/A	11458	36.68	420246	786432	0.53
Roses	N/A	123820	7.78	963872	196608	4.90
Wash.ht	441800	49999	56.45	2822472	894916	3.15
Brooks	9995	9345	1137.87	4944040	1449616	3.41
Capitol	7153	7153	184.95	1322980	1572864	0.84
Rad	7096	7096	227.67	1615499	1449616	1.14

Table 1: Graphics Scenes Traced

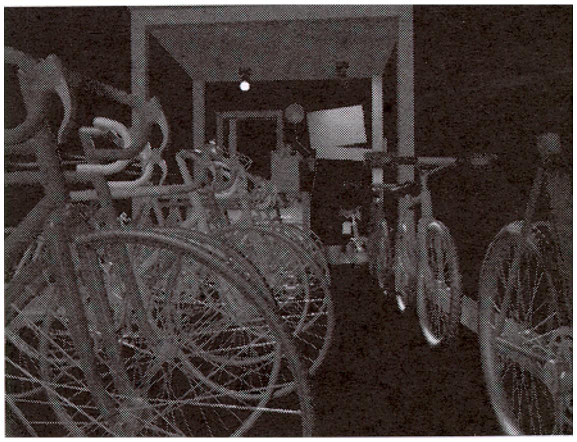

Figure 1: The *Bike* scene traced

conversion generated the pixel).

A summary of the scene traces used in the current paper is shown in Table 1. The name of the trace appears in the first column, with the number of primitives (before and after clipping) in the second and third columns. Following this are the average graphics primitive size (in pixels or samples, depending on the rendering package), the total number of pixels (or samples) generated during rendering of the scene, and the resolution A of the window. The last column specifies the average depth $\bar{d}$ of each scene.[1] A more complete discussion of these traces and their statistics may be found in [4]. Examples are shown in Figures 1 through 4. These are all copyrighted and are reproduced here by permission. The *Bike* was produced by E. Ostby and B. Reeves, and originally appeared on the cover of *SIGGRAPH '87* [21]. *Brooks* is a model of Fred Brooks' house project [3]. *Roses* is copyright 1990 by D. R. Fowler, J. Hanan, and P. Prusinkiewicz. *Zinnia* is copyright 1990 by D. R. Fowler, P. Prusinkiewicz, J. Hanan, and N. Fuller. Both appear in [23].

Now, since each pixel in a graphics trace contains a pointer to the primitive that generated it, the set of primitives can be culled from the trace file. This set can be assigned by simulation to n processors, and each processor can be considered to have rendered exactly those pixels whose "parent graphics primitive" record field matches a primitive assigned to that processor. For all trace-driven simulations reported in the current paper, assignment of primitives has been by round-robin. That is, primitives have been assigned sequentially (mod n) in the order in which they were encountered in the original graphics model. This strategy is similar to strictly random assignment, but can still be expected to lead to good computational load balancing in an object-parallel architecture (e.g. it is similar to the primitive "scattering" of [18]).

3 Terms and concepts

We will require some terms. Let

$d_{x,y}$ = the *depth* at pixel location (x, y).

$\bar{d}$ = the *depth complexity* of a given scene; that is the average depth over the screen.

p_d = the probability that at arbitrary pixel location (x, y), $d_{x,y} = d$.

A = the resolution of the screen or window, in pixels.

n = the number of processors in the multiprocessor.

Consider one pixel location (x, y). When a graphics scene is rendered, there may be multiple graphics primitives that generate a pixel for (x, y). These multiple pixels are the motivation behind z-buffering. For any given scene, we refer to the number of primitives that render to a given (x, y) as the *depth* $d_{x,y}$ at (x, y). We refer to the average depth $\bar{d}$ over the screen as the *depth complexity* of the scene. If depth is not evenly distributed over (x, y) then we also refer to the probability p_d that at arbitrary (x, y), the depth $d_{x,y} = d$. We distinguish *active* from *inactive* pixels at each processor. As each processor renders its subset of the scene, it produces pixels for *some* of the A pixel locations. We say that a pixel location is *active* if at least one pixel has been rendered to that (x, y), and that it is *inactive* otherwise.

4 A straightforward pixel merging algorithm

In perhaps the most straightforward solution to pixel merging on a general-purpose multiprocessor, each node is as-

[1] The *Bike's* average depth in this table is a correction from [4]

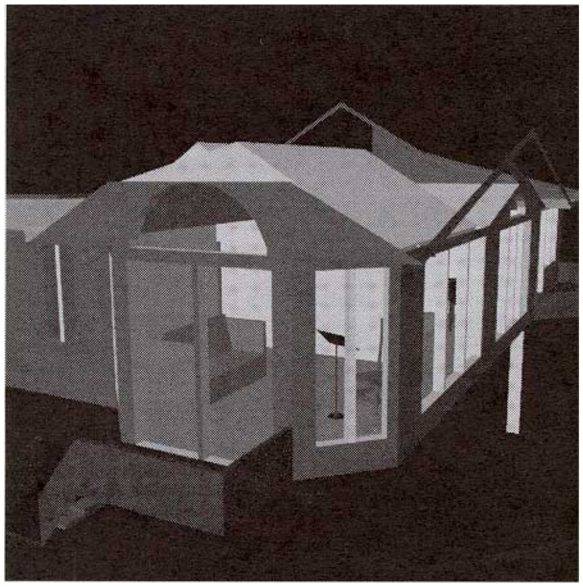

Figure 2: The *Brooks* scene traced

signed responsibility for merging pixels from some subset of the A pixel locations. That is, after (or as) each node generates pixels for its subset of the graphics database, it sends each pixel to the destination node that is responsible for merging for that pixel's (x, y). Each destination node z-buffers the pixels it receives. This is the algorithm employed by Kubota Pacific in the Denali architecture [15], and appears to be the algorithm used by Evans and Sutherland in their Freedom series [7].

The attraction of this algorithm is that it works on any multiprocessor for which there is network connectivity between all nodes. On a shared-memory machine, mailbox communication can be used to emulate messages.[2] Another advantage of the algorithm is that merging can begin before rendering is complete, since each node can forward pixels to their destinations before it completes rendering the scene. On the other hand, the interconnection network must carry every pixel generated.

Consider the algorithm in greater detail. As each node renders pixels, it may or may not z-buffer them locally (that is, with respect to the other pixels it generates for its subset of the graphics database). If each node *does not* z-buffer its pixels locally, then no pixels are deleted from consideration until they have been received by the destination processors. Thus, $\bar{d}A$ pixels must be sent, received, and compared, and aggregate network traffic is $\bar{d}A$. If each node *does* locally z-buffer pixels, then potentially some pixels are deleted before they must be sent. However, analysis and trace-driven simulations have shown that on multiprocessors larger than even 8 processors, few pixels are deleted by local z-buffering [4]. Thus, in this case, network traffic is still about $\bar{d}A$ even if there is per-processor local z-buffering.

To summarize, the straightforward pixel merging algorithm requires aggregate network bandwidth of $\bar{d}A$ times the number of bits to represent a pixel, and requires $O(\bar{d}A)$ processor cycles to route, send, receive, and compare pixels.

[2] Of course, direct z-buffering is possible, but the algorithm above avoids contention if the granularity of memory access is larger than a pixel.

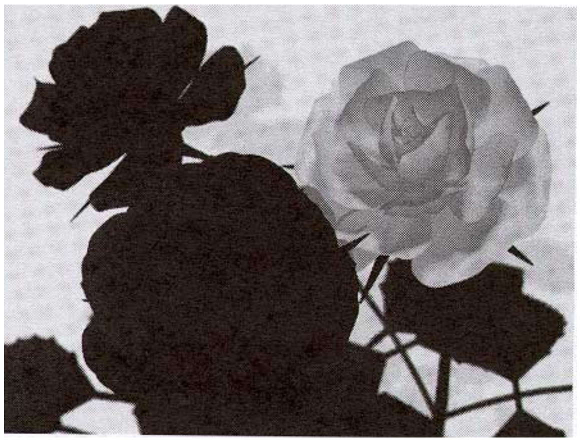

Figure 3: The *Roses* scene traced

5 The PixelFlow algorithm

An object-parallel implementation that employs an alternative merging algorithm is currently under construction at the University of North Carolina, Chapel Hill [19]. We will call this algorithm the PixelFlow algorithm, even though it has also been proposed by [18], [24], and [29]. In the PixelFlow algorithm, processors are connected by a pipeline network for pixel merging. Each processor has an associated full frame buffer, to which it renders. Upon completion, the first node streams its full frame to the second node; the second node, while receiving the frame, reads from its own frame buffer, and z-buffers each of its own pixels with each of the pixels it receives from the previous node. The second processor forwards a single pixel for each location to the next processor. And so on.

The advantage of this strategy is that it is simple, and statically partitions load across the links of the pixel merging network. Thus, the network is deterministic and performance can be predicted. On the other hand, the network load is greater than it need be; the PixelFlow strategy does not take advantage of local frame buffer sparsity - each node transmits A pixels regardless of whether or not they are all active. Analysis and trace-driven simulations have suggested that on the scenes discussed in the current paper, even on an 8-node machine only 10% of each local frame buffer comprises active pixels [4], so the overhead is significant. Furthermore, the PixelFlow strategy does require a special-purpose network for pixel merging, and since a full frame from each node is transmitted, each link and merging circuit must be quite fast, especially if oversampling is supported.

Now, consider the algorithm in greater detail. Since each processor sends A pixels to the next, and there are n links in the pipeline (including the link to the frame buffer itself), the total bandwidth required on the network is nA. Of course, since merging is pipelined, the bandwidth required on each link is only A. With respect to computation bandwidth, each node must read A pixels from local storage and also from the previous node, compare these two streams, and write A pixels to the next node. Thus, in aggregate, $O(nA)$ processing is required to read, compare, send and receive pixels.

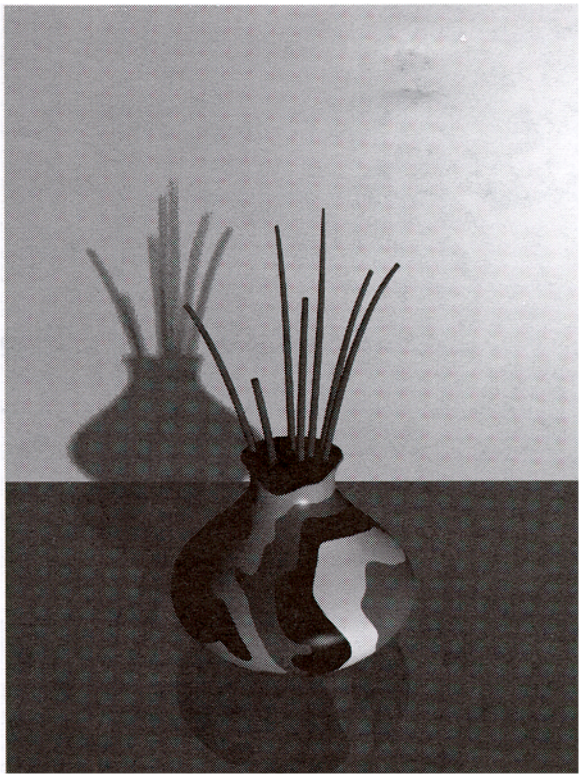

Figure 4: The *Zinnia* scene traced

6 Uniprocessor z-buffering

We have seen that the straightforward pixel merging algorithm requires in aggregate $O(\bar{d}A)$ processing time, and requires network bandwidth of $\bar{d}A$ pixels per frame. The PixelFlow algorithm requires $O(nA)$ processing time, and aggregate network bandwidth of nA pixels.

Consider now the processing and memory bandwidth required by local z-buffering. For any pixel location (x, y), $d_{x,y}$ pixels are rendered. For each of these pixels, the z-value in the z-buffer must be read from location (x, y), z-values must be compared, and if the new pixel is "frontmost" it must be written to the frame buffer and its z-value must be written to the z-buffer. What is the probability that a particular pixel will be accepted as "frontmost"? Well, this depends on how many, and on which pixels have previously been z-buffered. The first pixel at (x, y) that is z-buffered is certainly accepted as frontmost. The second pixel at (x, y), in general, is accepted as frontmost with probability $1/2$, the third with probability $1/3$, *etc.*

Somewhat more formally, consider all of the pixels that render to location (x, y). First assume that the order in which they are z-buffered is randomly distributed. Then the expected number $EF_{x,y}$ of pixels accepted as frontmost at (x, y) during z-buffering is

$$EF_{x,y} = \sum_{1 \leq d \leq d_{x,y}} 1/d$$

Second, assume for simplicity that the depth $d_{x,y}$ at each (x, y) is $\bar{d}$, and to avoid unenlightening complexity, that $\bar{d}$ is an integer.[3] Third, assume that the order in which pixels are z-buffered at location (x, y) is independent of the order in which pixels are z-buffered at location (x', y'). Then the expected number EF of pixels accepted as frontmost over the entire screen during z-buffering is

$$\begin{aligned} EF &= A \sum_{1 \leq d \leq \bar{d}} 1/d \\ &= AH_{\bar{d}} \\ &\approx A \log(\bar{d}) \end{aligned}$$

where $H_{\bar{d}}$ is the $\bar{d}^{th}$ harmonic number, for which $\log(\bar{d})$ is an approximation.

For each pixel z-buffered, the previous value in the z-buffer must be read and a comparison be made. Thus $\bar{d}A$ pixels must be read and compared. However, only about $\log(\bar{d})A$ pixels must be written, since only this many are expected to be frontmost. Thus, the expected cost of z-buffering on a uniprocessor is $O(\bar{d}A + \log(\bar{d})A)$.

When the uniprocessor z-buffering algorithm is distributed on a multiprocessor, we would like to write onto the interconnection network only those pixels "that must be written." The straightforward pixel merging algorithm distributes z-buffering by dividing the process between pixel read and pixel compare, and thus requires $\bar{d}A$ network bandwidth. The PixelFlow algorithm distributes z-buffering essentially by reading, comparing, and writing every pixel location n times, and thus requires nA network bandwidth. Can we do better, and under what circumstances?

7 A distributed snooping merge algorithm

We propose the following algorithm for pixel merging when network broadcast is available, for example as it is on a shared bus. Pixel merging begins after all processors have completed rendering, and proceeds as follows. Consider the n processors in a multiprocessor ordered from $1 \leq i \leq n$, and suppose there is a global frame buffer to which can be broadcast pixel values. Processor 1 first broadcasts to the global frame buffer each pixel it has rendered (that is, it broadcasts all *active* pixels from its local frame buffer). While processor 1 broadcasts, *each of the other processors listens*, and compares each broadcast pixel with the corresponding pixel in its local memory. If the broadcast pixel "hides" the local pixel, the listening processor deletes the hidden pixel from local memory; if the local pixel hides the broadcast pixel, the snooping processor *does nothing yet*. In turn, then, each processor k broadcasts its active pixels from local memory *that have not already been deleted*, and each processor j, $k < j \leq n$ listens and potentially deletes pixels from its local memory. When all processors have broadcast to the global frame buffer all their non-deleted local pixels, pixel merging is complete, and the global frame buffer contains a correctly z-buffered image.

This algorithm is a derivative of the well-known "snooping" cache coherency protocols upon which a number of shared-memory multiprocessors have been based (cf. [13]) In these architectures, each node maintains the consistency of its

[3] It is shown in [5] for essentially the same equation in the next section that the first assumption is a worst-case and results in an upper bound for the expected value of pixels accepted as frontmost.

cache with other nodes' memory accesses by snooping on a shared bus over which memory updates must be written. In these protocols for cache coherency, snooping is employed in order to maintain consistent globally shared virtual memory. In our scheme, snooping is employed in order to remove pixels, wherever possible, from consideration by pixel merging, and thus to reduce the bandwidth required on a shared broadcast medium such as a bus.

Consider the network bandwidth required by this algorithm. In the worst-case, a sequence of planes ordered back-to-front is assigned to processors such that the first processor renders the backmost plane, the last processor renders the frontmost frame. This results in network traffic of nA pixels per frame. But this scene is uninteresting from the perspective of animation or visualization. We turn now to an analysis of the expected-case performance of the algorithm.

7.1 Expected-case network traffic of distributed snooping

The analysis of the distributed snooping algorithm is similar to the analysis of z-buffering on a uniprocessor. Here however, we first assume that after rendering, there is no pixel location in any *local frame buffer* at which $d_{x,y} > 1$. In essence, we assume that no two primitives that render to (x, y) are assigned to the same processor. This means that no pixels are deleted by local z-buffering, and represents a worst-case assumption for the expected traffic of the snooping algorithm [5]. Also, as discussed in section 4, previous work has suggested that for machines larger than about 8 processors, few pixels are deleted in object-parallel architectures by local z-buffering. Second, as we did in analyzing uniprocessor rendering, we assume that over all A pixel locations, the z-values of the $d_{x,y}$ pixels that render to (x, y) are randomly and independently distributed. Third, we will assume essentially as we did for uniprocessor rendering that the pixel z-values to be broadcast are independent of the order of the processors that must broadcast them. From these assumptions it follows that the probability that the k^{th} broadcast pixel (at (x, y)) is frontmost is $1/k$.

Now, consider a single pixel location (x, y). The first processor with an active pixel at (x, y) must broadcast its pixel, and must do so with probability 1. The second processor that has (x, y) active must on average broadcast its pixel with probability 1/2, since it has the frontmost of two pixels with probability 1/2. The third processor that has (x, y) active broadcasts a pixel with probability 1/3, and so on. Since no pixels have been previously deleted at (x, y) by local z-buffering, the pixel traffic expected from location (x, y) is

$$ET_{x,y} = \sum_{1 \le d \le d_{x,y}} 1/d \qquad (1)$$

$$= H_{d_{x,y}} \qquad (2)$$

where $H_{d_{x,y}}$ is the harmonic number for $d_{x,y}$.

If we know the distribution of depth over A, the expected traffic from all pixel locations is

$$ET = A \sum_{1 \le d} p_d H_d \qquad (3)$$

Some work characterizing distributions of depth in graphics scenes has been done (cf. [4]). However, we can safely bound expected traffic. We will say that depth is "evenly" distributed over A if for all (x, y), $\lfloor \bar{d} \rfloor \le d_{x,y} \le \lceil \bar{d} \rceil$. We show in [5] that this represents a worst-case assumption for the expected traffic of the snooping algorithm; that is, when graphics scene depth is evenly distributed, the expected-case traffic of the algorithm is worse than it is for any other distribution of depth. Thus, given $\alpha = (\bar{d} - \lfloor \bar{d} \rfloor)$, we bound ET by

$$ET \le A\left[(1-\alpha)H_{\lfloor \bar{d} \rfloor} + \alpha H_{\lceil \bar{d} \rceil}\right] \qquad (4)$$

where $H_{\lfloor \bar{d} \rfloor}$ and $H_{\lceil \bar{d} \rceil}$ are the harmonic numbers for $\lfloor \bar{d} \rfloor$ and $\lceil \bar{d} \rceil$. Now, harmonic numbers grow very slowly; as already noted in the analysis of uniprocessor z-buffering, harmonic numbers grow about as $\log(\bar{d})$. Thus, as the depth at (x, y) increases, expected traffic grows only about as the log() of the average depth. We now turn to trace-driven simulation results to corroborate this prediction of bandwidth requirements of the distributed snooping algorithm for pixel merging.

7.2 Trace-driven simulation results for distributed snooping

The results of trace-driven simulations are shown in Figures 5 and 6. In these figures, expected, simulated, and maximum traffic are plotted. The y-axis is the ratio of the respective traffic to the minimum possible traffic. Minimum traffic is simply the number of active pixels observed when the scene is rendered on a uniprocessor; that is, from every active pixel location must be written at least one pixel to the global frame buffer. Maximum traffic is the traffic generated by the straightforward pixel merging algorithm using broadcast (the global frame buffer is assigned responsibility to z-buffer all pixels), which requires bandwidth $\bar{d}A$. Expected traffic has been calculated from equation 4.

The trace-driven simulations are labeled in the figures as "observed" since these simulations are in some sense exact. Since the traces have been gathered from actual rendering, then if the primitives were distributed round-robin on a real multiprocessor as they were in simulation, exactly the traffic observed in simulation would be observed in real object-parallel rendering.

Several observations on these results deserve note:

- In all but one of the simulations, expected traffic is close to observed traffic.
- When depth complexity is high, distributed snooping quite effectively reduces traffic; this is particularly true for the scenes with depth complexity greater than three, *Brooks*, *Cube*, *Roses*, and *Wash.ht.* Traffic is reduced by factors of about 3.3, 4.0, 2.3 and 2.0 respectively.
- When depth complexity is low, distributed snoooping eliminates traffic, but since there is less depth, traffic reduction is less dramatic. In *Bike*, *Capitol*, and *Rad*, reduction is by less than a factor of two.
- In one simulation (*Zinnia*), observed exceeds expected traffic. Recall from section 2 that in simulation, assignment of graphics primitives has been by round-robin. In this scene, the largest primitive (the background) appears first in the database, violating the assumption of

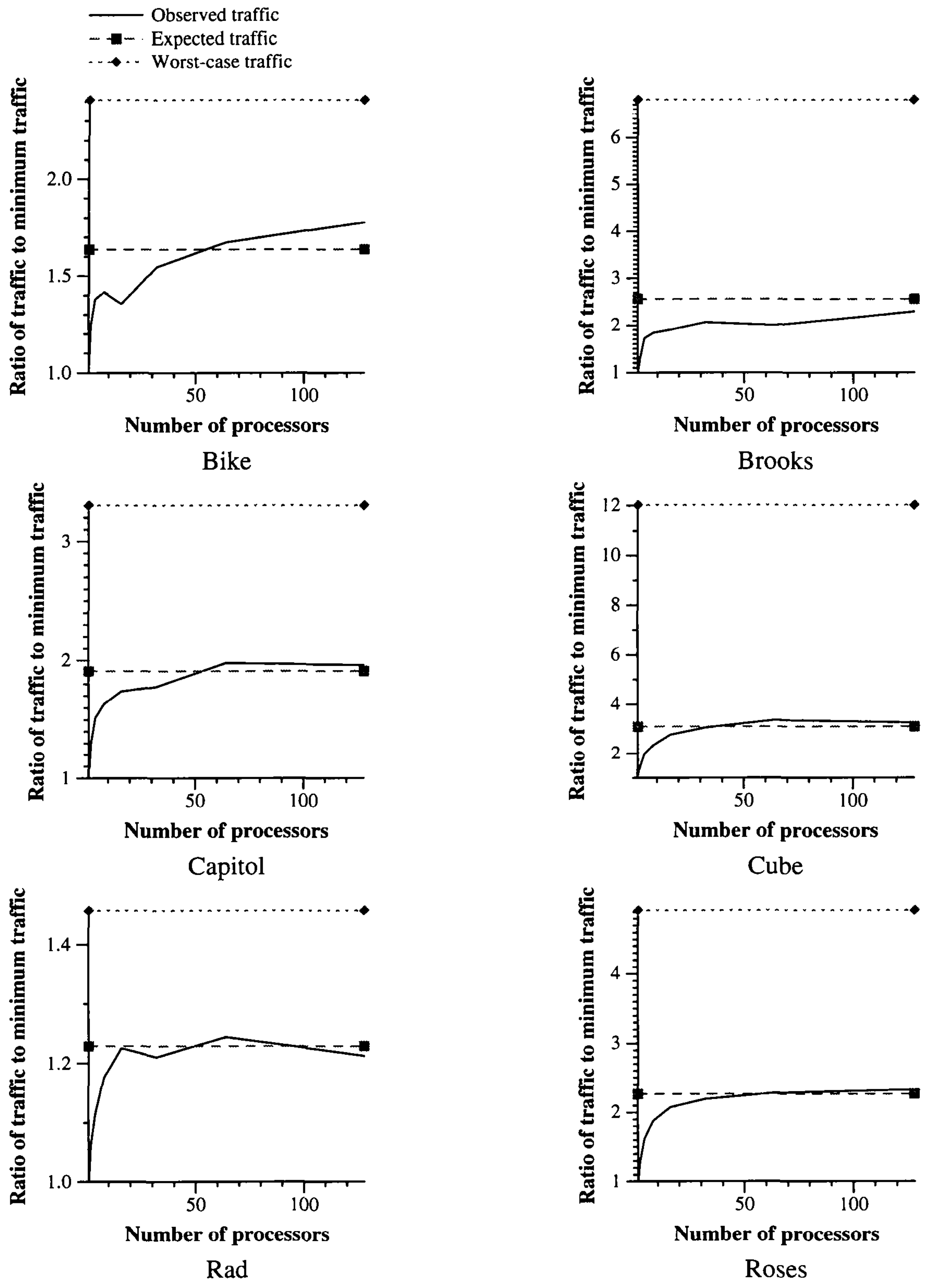

Figure 5: Distributed snooping traffic: observed and expected

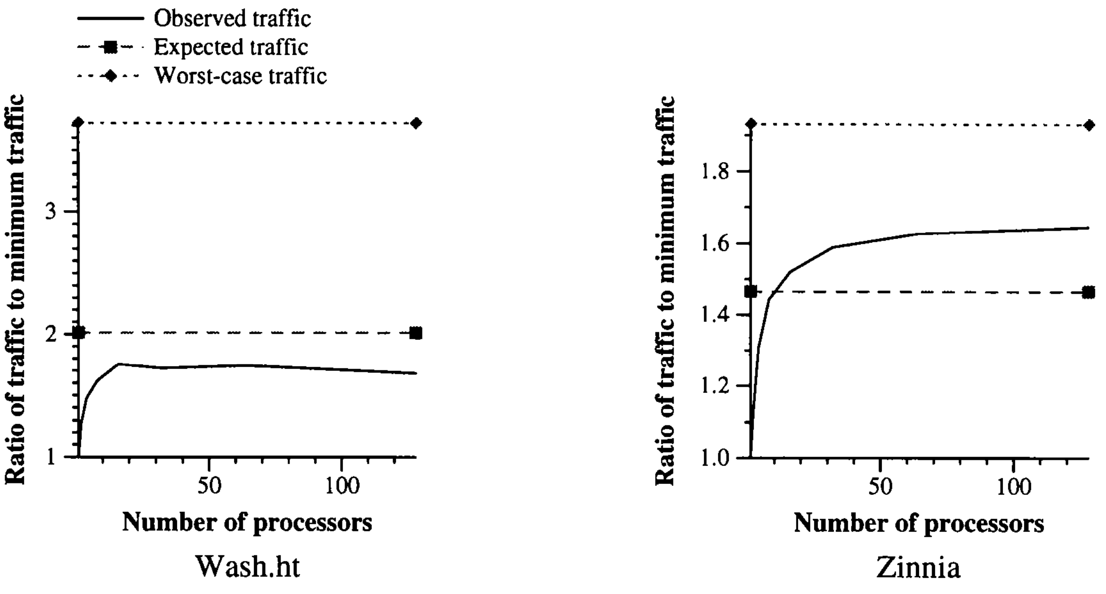

Figure 6: Distributed snooping traffic: observed and expected

independence between z-value order and processor order. Not only is the assumption violated, but a pathologic case is created. This points to the need in object assignment algorithms for use with distributed snooping to consider background primitives separately, and even to force their assignment to processors later in the broadcast order. Of course, randomizing object assignment can be expected to solve this problem as well.

- In general, observed traffic grows initially as a function of number of processors before reaching a plateau, while expected traffic is flat. This is because local z-buffering has been ignored. An analysis of depth complexity in object-parallel architectures suggests that local z-buffering can be ignored *except* on small machines. However, as can be seen, the assumption that local z-buffering can be ignored is conservative.

The combination of expected-case analysis and trace-driven simulations show that the distributed snooping algorithm is effective at reducing the network traffic inherent in object-parallel rendering. Below we discuss the performance and an application of distributed snooping in more detail.

8 Future directions: a distributed snooping z-buffer

The previous section has shown that distributed snooping can effectively reduce network traffic when network broadcast is available. However, distributed snooping is expensive when processing bandwidth is also considered. On average, $\log(\bar{d})A$ pixels must be broadcast, and $n\log(\bar{d})A$ pixels must be read from the network and compared to local pixels, and $[\bar{d}-\log(\bar{d})]A$ pixels must be deleted from local z-buffers. The required processing bandwidth is thus $O(n\log(\bar{d})A + \bar{d}A)$, even though the required network bandwidth is only $\log(\bar{d})A$. This suggests that although distributed snooping is attractive when network bandwidth is limited, hardware support is desirable. We call such support a *distributed snooping z-buffer*. We are in the process of design for such hardware for shared-memory bus multiprocessors. Our design requires local buffering, and merges pixels by memory-mapping the global frame buffer onto the shared-memory bus. We plan to continue this work and present the results in another forum.

9 Conclusions

In this paper we have explored the processing and network bandwidth requirements of three algorithms for pixel merging in object-parallel rendering algorithms. The first algorithm, which we have referred to as the straightforward algorithm, requires network bandwidth of $\bar{d}A$, where $\bar{d}$ is the depth complexity of the scene, and A is the resolution of the screen or window. This algorithm is appealing in that it works on any multiprocessor with connectivity between all nodes, but may be naive in its use of network bandwidth.

The second merging algorithm, which we have referred to as the PixelFlow algorithm, requires aggregate network bandwidth of nA, where n is the number of processors in the multiprocessor. Although this bandwidth is greater than that required by the naive algorithm, the PixelFlow algorithm statically balances network load across n links so that each link must support deterministic traffic of A per frame. However, this algorithm does not take advantage of sparsity in local frame buffers, and requires a fast special-purpose merging network.

Finally, we have presented and analyzed an algorithm for pixel merging when network broadcast is available. We refer to the algorithm as the distributed snooping algorithm, and we have shown in this paper that in the expected case, it requires network bandwidth $\log(\bar{d})A$, and does not require a special-purpose network for pixel merging, although hardware support or snooping hardware modification may be desirable.

10 Acknowledgements

We thank Frank Crow and Apple Computer, Kai Li and Matsushita Information Technology Laboratory, and DARPA (grant DABT63-92-C-0053) for funding. We also thank Mark Greenstreet, University of British Columbia, for valuable comments on a previous version of this paper, and Channing Verbec of IBM for valuable discussion on IBM's Power Visualization System.

References

[1] Akeley, Kurt, and Tom Jermoluk, "High-Performance Polygon Rendering", *Computer Graphics*, v. 22, no. 4, August 1988.

[2] Apgar, Brian, Bret Bersack, Abraham Mammen, "A Display System for the Stellar Graphics Supercomputer Model GS1000", *Computer Graphics*, v. 22, no. 4, August 1988.

[3] Brooks, Fred, "Fred Brooks' House Project", Architects: J. Knox Tate IV and F. P. Brooks Jr., Dataset from Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

[4] Cox, Michael, and Pat Hanrahan, "Depth Complexity in Object-Parallel Graphics Architectures," *Proceedings of the Seventh Workshop on Graphics Hardware, Eurographics '92*, Cambridge, England, September 1992.

[5] Cox, Michael, and Pat Hanrahan, "Evenly Distributed Depth is the Worst for Distributed Snooping", Research Report CS-TR-427-93, Department of Computer Science, Princeton University, June 1993.

[6] Diede, Tom, *et. al*, "The Titan Graphics Supercomputer Architecture", *IEEE Computer*, v. 21, no. 9, September 1988.

[7] Evans and Sutherland Computer Corporation, *Freedom Series Technical Report*, October 1992.

[8] Fuchs, Henry, and John Poulton, "Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design*, Vol. 2, No. 3, Q3 1981.

[9] Fuchs, Henry, *et. al*, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Computer Graphics*, v. 23, no. 3, July 1989.

[10] Fussel, Donald, and Bharat Deep Rathi, "A VLSI-Oriented Architecture for Real-time Raster Display of Shaded Polygons", *Graphics Interface '82*, 1982.

[11] Gharachorloo, Nader, *et. al*, "Subnanosecond Pixel Rendering with Million Transistor Chips", *Computer Graphics*, v. 22, no. 4, August 1988.

[12] Gharachorloo, Nader, Satish Gupta, Robert Sproull, Ivan Sutherland, "A Characterization of Ten Rasterization Techniques", *Computer Graphics*, v. 23, no. 3, July 1989.

[13] Hennessy, John, and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Mateo CA, 1990.

[14] International Business Machines Corporation, *Power Visualization System*, Product Marketing, Dept. 590, 8 Skyline Drive, Hawthorne NY 10532, 1992.

[15] Kubota Pacific Computer, *Denali Technical Overview*, version 1.0, March 1993.

[16] Lovett, Tom, and Shreekant Thakkar, "The Symmetry Multiprocessor System", *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[17] Molnar, Steven, "Combining Z-buffer Engines for Higher-Speed Rendering", in *Advances in Computer Graphics Hardware III*, Springer-Verlag, New York NY, 1988.

[18] Molnar, Steve, *Image-Composition Architectures for Real-Time Image Generation*, Ph.D Thesis, University of North Carolina at Chapel Hill, October 1991.

[19] Molnar, Steve, John Eyles, and John Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Computer Graphics*, v. 26, no. 2, July 1992.

[20] Northcutt, J. Duane, *et. al*, "A High Resolution Video Workstation", Sun Microsystems Laboratories Inc. SMLI-92-0056, August 1991.

[21] Ostby, Ellen, and William Reeves, "A Night in the Bike Store", cover of *Computer Graphics*, v. 21, n. 7, July 1987.

[22] Potmesil, Michael, and Eric Hofert, "The Pixel Machine: A Parallel Image Computer", *Computer Graphics*, v. 23, no. 3, July 1989.

[23] Prusinkiewicz, Przemyslaw and Aristid Lindenmayer, with James Hanan, F. David Fracchia, Deborah Fowler, Martin de Boer, and Lynn Mercer, *The Algorithmic Beauty of Plants*, Springer-Verlag, New York NY, 1990.

[24] Schneider, Bengt-Olaf, and Ute Claussen, "PROOF: An Architecture for Rendering in Object Space", in *Advances in Computer Graphics Hardware III*, Springer-Verlag, New York NY, 1988.

[25] Silicon Graphics, *Graphics Library Programming Guide*, Document Number 007-1210-040, 1991.

[26] Shaw, Chris, Mark Green, and Jonathan Schaeffer, "A VLSI Architecture for Image Composition,", in *Advances in Computer Graphics Hardware III*, Springer-Verlag, New York NY, 1988.

[27] Upstill, Steve, *The RenderMan Companion*, Addison-Wesley, Reading MA, 1989.

[28] Verbec, Channing, personal communication, October 1992, on the hardware and software architecture of the IBM Power Visualization System.

[29] Weinberg, Richard, "Parallel Processing Image Synthesis and Anti-Aliasing", *Computer Graphics*, v. 15 no. 3, August 1981.

[30] Whitman, Scott, *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers International, Boston MA, 1992.