Depth Complexity in Object-Parallel Graphics Architectures

Michael Cox
Pat Hanrahan

ABSTRACT
We consider a multiprocessor graphics architecture object-parallel if graphics primitives are assigned to processors without regard to screen location, and if each processor completely renders the primitives it is assigned. Such an approach leads to the following problem: the images rendered by all processors must be merged, or composited, before they can be displayed. At worst, the number of pixels that must be merged is a frame per processor. Perhaps there is a more parsimonious approach to pixel merging in object-parallel architectures than merging a full frame from each processor.
In this paper we analyze the number of pixels that must be merged in object-parallel architectures. Our analysis is from the perspective that the number of pixels to be merged is a function of the depth complexity of the graphics scene to be rendered, and a function of the depth complexity of each processor's subset of the scene to be rendered. We derive a model of depth complexity of graphics scenes rendered on object-parallel architectures. The model is based strictly on the graphics primitive size distribution, and on number of processors. We validate the model with trace data from a number of graphics applications, and with trace-driven simulations of rendering on object-parallel architectures.
The results of our analysis suggest some directions in design of object-parallel architectures, and suggest that our model can be used in future analysis of design trade-offs in these architectures.

1.1 Introduction

1.1.1 Graphics Parallelism

Let us agree to call the two types of parallelism in computer systems *horizontal parallelism* and *vertical parallelism.* The former has been referred to as data parallelism, multiprocessing, and parallel processing; the latter has been referred to as instruction-level parallelism and pipelining.

Let us further agree to call the two graphics variants of horizontal parallelism *image-parallelism* and *object-parallelism.* The former has also been referred to as screen-space parallelism. In this approach, each processor in a multiprocessor system is assigned a region of the screen, and is responsible for rendering the portions of all primitives that intersect the region. The latter has also been referred to as image-composition [14, 20] In this approach, each processor is assigned some subset of the primitives to be rendered, and may generate pixels in any portion of the screen.

Both vertical parallelism and image parallelism have been successfully employed in many systems, both research [6, 7, 8, 9, 24] and commercial [1, 2, 16]. Object parallelism has received some attention [13, 14, 18, 20, 23], but has neither received the attention nor implementation that image parallelism has received, in spite of apparent advantages. We plan to explore the tradeoffs between object- and image-parallel in future forums, but briefly:

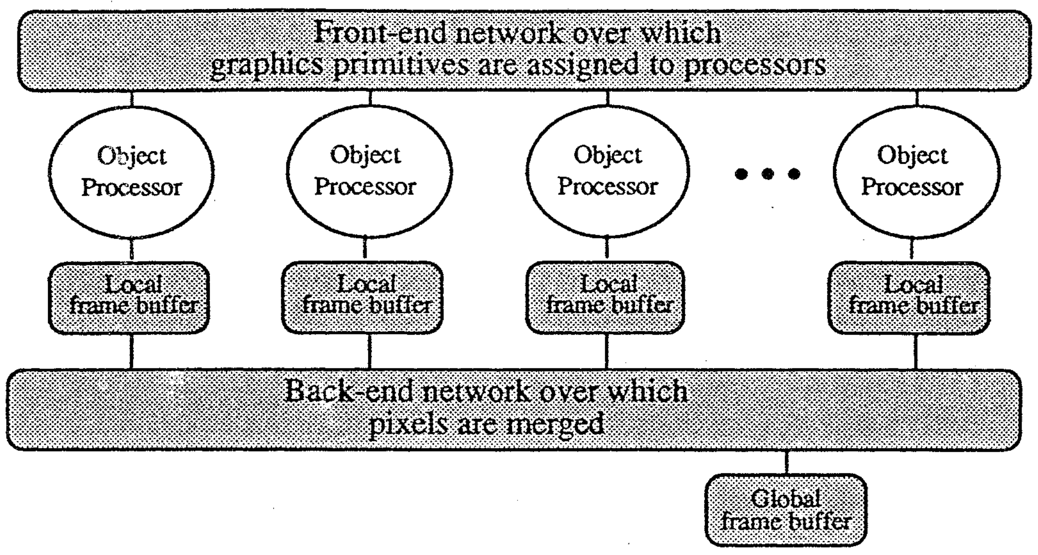


FIGURE 1.1. Generic object-parallel rendering architecture

- As image-parallel machines scale, each processor has fewer and fewer pixels; thus, the ratio of communication overhead for assignment of objects to processors increases.
- The per-primitive setup time also increases, since for each primitive that covers the screen-space of k processors, k processors must perform the setup calculations for rendering (which may include such expensive operations as vertex shading).
- All primitives in image-parallel architectures must be transformed to screen-space before they can be assigned to processors for rendering; this can be expensive for databases of many small primitives, or when high-level object descriptions have high cost of transformation and boundary testing. In any event, this requirement limits potential parallelism by serializing an essentially parallel problem.
- MIMD object-parallel systems will be inherently more flexible, allowing, for example, different processors to implement entirely different graphics rendering pipelines.
- Primitives do not require reassignment between frames in object-parallel systems; thus, such animations as "architectural walk-throughs" can be either faster or cheaper.
- As processors are added to image-parallel architectures, so are wires to frame buffer memory. For many processors, this is a serious routing and engineering problem [3]

However, there is one obvious problem with the object parallel approach to graphics hardware: since any processor's objects almost certainly overlap (when projected into screen coordinates) with objects from other processors, the pixels rendered by all processors must be merged (i.e. z-buffered) before they can be written to and displayed from a global frame buffer.

We refer to this as the *pixel merge problem* and believe that it will be the rate-limiting component in object-parallel architectures.

1.1.2 Object-Parallel Graphics Architectures

A generic object parallel architecture is shown in Figure 1.1. We view rendering on this architecture as comprising three phases: *object assignment to processors*, *rendering*, and *pixel merging*. We begin with a *graphics scene*, which comprises a collection of graphics primitives, or *graphics objects*. These primitives are assigned to the processors via the *front-end network*. Each processor then renders its subset of the graphics scene, performing local hidden surface removal by z-buffering, and producing pixels into a *local frame buffer*. In the final phase, the pixels from all processors' local frame buffers are merged via the *back-end network* into the *global frame buffer*. Of course, in a particular implementation of this object-parallel architecture, the front-end network may also be the back-end network, there may be no local frame buffer, etc.

Now, for each screen position (i.e. frame-buffer coordinate) (x, y), we define the *depth* at (x, y) in a graphics scene to be the number of primitives that render a pixel to (x, y).[1] We define the *depth complexity distribution* of a graphics scene as the distribution of depth over all screen coordinates (x, y); in this paper we will informally refer to the average of this distribution as the *depth complexity*. We can also speak of the depth complexity distribution of any subset of a graphics scene, and in particular of the depth complexity distribution of an arbitrary processor's subset of the graphics scene.

Notice that scenes with larger depth complexity require more processing for z-buffering when rendered on a uniprocessor. In an object-parallel architecture, scenes with larger depth complexity potentially generate more back-end pixel traffic than those with lower depth complexity. This is because each primitive in the initial graphics scene may overlap arbitrarily many other primitives (when projected onto the frame buffer). If all such primitives are rendered by the same processor (as they are in image-parallel architectures), then z-buffering is performed entirely at the local frame buffer; if on the other hand such primitives are rendered by different processors, z-buffering must also be performed on the back-end network or at the global frame buffer. Thus, higher depth complexity potentially means greater traffic on the back-end network, and if all depth complexity is removed by local (per-processor) z-buffering, the back-end traffic is minimized. *Thus, we can analyze both local frame buffer requirements, and back-end traffic by analyzing the depth complexity distribution.*

In particular, note in Figure 1.1 that a graphics scene's depth complexity is potentially removed at three sites: the local frame buffers, the back-end network, and the global frame buffer. The current paper investigates the first of these, local depth complexity removal. Since the back-end network is left undefined, the conclusions of the current paper are applicable to object-parallel architectures in general. In particular, our conclusions bear upon local frame buffer design in object-parallel architectures, and upon the pixel traffic that can be expected on the back-end network of an object-parallel architecture after local depth complexity removal.

In this paper, we derive an analytic model of the depth complexity distribution and apply it to some "typical" graphics scenes, and to the partitions of those scenes as they might be rendered on an object-parallel multiprocessor. We validate the model against trace data taken from a number of graphics applications, and against trace-driven simulations of rendering on object-parallel architectures. The traces we employ have been gathered from instrumentation of the *RenderMan* graphics package from Pixar [22], and of the *Graphics Library (GL)* interface from Silicon Graphics Inc. [19]. Finally, we draw conclusions both from the analytic model and from the trace data and trace-driven simulations.

[1]We assume in this paper that each primitive's projection to screen coordinates is not self-overlapping.

1.2 Analytic Model of Depth Complexity

Let

$A =$ the area of the screen, in pixels.

$R =$ the set of graphics primitives to be rendered.

$N = |R|$.

For each $r \in R$, $|r| =$ the number of pixels to which graphics primitive r renders. That is, $|r|$ is the *projected size* of primitive r.

$R_k = \{r| \ |r| = k\}$; that is, R_k is the subset of R comprising primitives of size k.

$N_k = |R_k|$.

$n =$ the number of processors on which the scene is rendered

$p_d =$ the probability that (after rendering) the depth at an arbitrary pixel in an arbitrary processor's local frame buffer is exactly d. When $n = 1$, this is the probability that the depth at an arbitrary pixel from a rendering of the initial graphics scene is exactly d.

In addition, we will informally refer to the "underlying size distribution" of particular graphics scenes (or simply to the "size distribution"). By this we mean the probability density function defined over k by $P[r \in R, |r| = k] = R_k/N$.

In this paper, we derive a generating function of the probability density of depth complexity in graphics scenes. The model we derive depends upon measured or assumed values for N, N_k $(1 \leq k \leq A)$, and n. In the current paper we measure the N_k and N of various graphics scenes, and explore the dependence on n by sensitivity analysis.

The model depends on the following assumptions:

(1) We assume that screen coverage by $r \in R$ is uniformly and randomly distributed. More precisely, we assume that for arbitrary pixels (x, y) and (w, z), and randomly selected primitive $r \in R$, that the probability that r renders to (x, y) is equal to the probability that r renders to (w, z).

(2) We assume that graphics primitives are uniformly and randomly assigned to processors to be rendered.

We model rendering as follows. A primitive r is randomly selected (without replacement) from the graphics scene R, is randomly assigned to a processor, and is rendered by that processor. The depth at each pixel (in the processor's local frame buffer) to which r renders increases by one. Primitives are selected, assigned to processors, and rendered until no primitives remain.[2]

Consider this process from a single pixel location (x, y) in an arbitrary processor's local frame buffer. A single primitive $r, |r| = k$ renders to (x, y) with probability $k/(An)$ (by assumptions 1 and 2). The generating function of this probability is

$$H_k(v) = (1 - \tfrac{k}{An}) + \tfrac{k}{An}v$$

[2]Of course, primitives may be assigned in one phase and rendered in another, or rendering may occur concurrently, etc.

Generating functions are useful tools in exploring probability density functions (cf. [12, 25]). In $H_k(v)$, for example, the coefficient of v^0 expresses the probability that r does not render to (x,y), the coefficient of v^1 expresses the probability that it does. Generating functions can be multiplied to obtain the convolution of the probability density functions they represent, provided that probabilities are independent. Thus, for example, the coefficient of v^2 in $H_k(v)^2$ expresses the probability that given two primitives of size k, both render to pixel (x,y).

Now, consider all primitives $r \in R_k$. The probability that exactly d of these render to (x,y) is binomially distributed, and is expressed (by assumptions 1 and 2) by the coefficient of v^d in the generating function

$$H_k(v)^{N_k} = \left[(1-\tfrac{k}{An}) + \tfrac{k}{An}v\right]^{N_k}$$

Finally, the generating function G of the probability density of depth complexity is the product of $H_k(v)^{N_k}$ over all possible graphics primitive sizes, that is

$$G(v) = \textstyle\prod_{k=1}^{A} \left[(1-\tfrac{k}{An}) + \tfrac{k}{An}v\right]^{N_k} \tag{1}$$

Recall probability p_d that an arbitrary pixel in an arbitrary processor's local frame buffer has depth exactly d; this is the coefficient of v^d of $G(v)$. It remains to extract these coefficients; we show in Appendix .1 that the probabilities obey the following recurrence:

$$p_d = \textstyle\sum_{j=0}^{d-1} \tfrac{(-1)^j}{d} \left[\sum_{k=1}^{A} N_k \left(\tfrac{k}{An-k}\right)^{j+1}\right] p_{d-1-j} \qquad when \;\; d > 0 \tag{2}$$

and

$$p_0 = \textstyle\prod_{k=1}^{A} \left(1-\tfrac{k}{An}\right)^{N_k} \tag{3}$$

Thus, given the graphics primitive size distribution for a given scene, and the number of processors in the object-parallel architecture, we can predict the depth complexity probability density function. We can also therefore calculate the expected depth complexity, and variance, and the expected number of active pixels per local frame buffer, which we define to be the number of pixels to which at least one primitive renders. Active pixels are important because they are those that must be transmitted from the local frame buffer over the back-end network to the global frame buffer in an object-parallel architecture.

Note that p_0 is the probability that for an arbitrary pixel (x,y), *no* primitive $r \in R$ renders to (x,y), and therefore that the probability that a pixel is active is simply

$$Pr[arbitrary\ pixel\ (x,y)\ is\ active\ ] = (1-p_0) \tag{4}$$

Finally, though we do not focus on these statistics in the current paper, the *expected depth complexity*, EG, and *variance of depth complexity*, VG are, respectively

$$EG = G'(1) = \textstyle\sum_{k=1}^{A} \tfrac{k}{An} N_k$$

and

$$VG = G''(1) + G'(1) - G'(1)^2 = \tfrac{1}{n} \textstyle\sum_{k=1}^{A} \tfrac{k}{An} N_k (1-\tfrac{k}{An})$$

1.3 Traces and Trace-Driven Simulations

We have instrumented several graphics packages to generate traces of the pixels generated during rendering of graphics scenes. To date, RenderMan [22] and a software implementation of SGI's GL [19] have been instrumented. In addition, we have developed or borrowed several translators from other formats into either RenderMan or GL. Of course, RenderMan renders by oversampling, and thus we trace "samples" rather than pixels; however, the treatment of samples is made virtually identical by considering each a pixel of a larger screen, or by considering a pixel a sample set of size one. We will not distinguish between pixels and samples in this paper, and will use the terms interchangeably.

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 conversion generated the pixel).

Some statistics can be gathered directly from the trace files (e.g. the depth complexity distribution of the original scene, the number of primitives, and the size distribution). Some statistics require simulated assignment of the primitives to the processors of an object-parallel architecture. Since each pixel 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 assignment strategy can be expected to lead to good computational load balancing in an object-parallel architecture (the primitive "scattering" of [14]), and has the additional advantages that 1) it requires very little computation by the processor that assigns objects, 2) it does not require a processor that explicitly assigns objects: if all processors have shared access to the graphics database, each processor i can scan the database and select for rendering every ith primitive $(mod\ n)$.

Some of the scenes below were traced by applying an instrumented version of RenderMan to an input file in RenderMan RIB file format [22]; subsequent trace-driven simulation of their rendering has been done by separately assigning each primitive encountered in the RIB file to a processor. Some of the scenes below were traced by rendering them through an instrumented software implementation SGI's GL [19], and storing a trace of the GL primitive calls; subsequent trace-driven simulation of their rendering has been done by separately assigning each GL primitive to a processor.

We report on results from the following traces in the current paper. Basic statistics of these scenes are summarized in Table 1.1.

- *Bike* – This scene was produced by E. Ostby and B. Reeves using RenderMan, and appeared on the cover of *SIGGRAPH '87* [15]. The scene is in the form of a RenderMan RIB file. The *Bike* is under copyright [15].

- *Cube* – This scene is the RenderMan RIB file produced from Exercise 2.6 of [22]. The scene comprises thousands of tiny cubes packed together to form a large cube.

- *Zinnia* – This scene was produced by Deborah Fowler while at the University of Regina, and appears in [17]. The scene comprises a number of plant stems in a vase. The scene is in Rayshade format [11], which we convert to GL with the Rayshade utility *rayview*. This scene is under copyright [17].

- *Roses* – This scene was also produced by Deborah Fowler, and appears in [17]. The scene comprises three quite photorealistic roses, with stems and ornamental greenery, and was traced by the same procedure as was Zinnia. This scene is under copyright [17].

- *Wash.ht* – This scene is the result of several levels of translation. From U.S. Geologic Survey *DEM* format data [21], we convert to Rayshade heightfield format (which is simply a two-dimensional array of z-values) [11]. We then convert from heightfield format to RenderMan RIB format by creating a mesh of triangles. This scene is a craggy mountain. We simulate the scene's assignment in an object-parallel architecture by assuming that all processors receive the heightfield data, and themselves render every nth triangle created when the mesh is created.

- *Brooks* – This scene is Fred Brooks' model of an entire house, from the University of North Carolina [4]. We convert the UNC format into RenderMan RIB format. This scene is under copyright [4].

- *Capitol* – This scene is AutoCAD output of a model of the U.S. Capitol building. The output is in RenderMan RIB format.

- *Rad* – This scene is output from the radiosity algorithm of [10]; the scene is the room model that appears in that publication. The algorithm partitions a scene into a mesh chosen specifically to maximize photorealism and minimize the number of polygons produced. One can imagine a "walk-through" with output from just such an algorithm on an object-parallel architecture: the radioisty calculations are done off-line, and the walk-through is performed in real-time by real-time object-parallel rendering.

TABLE 1.1. Graphics Scenes Traced

Trace name	Primitives (pre-clipping)	Primitives (post-clipping)	Total pixels in trace	Scene screen area	Average depth
Bike	16144	9618	2032965	1258408	1.62
Cube	48000	47459	6459105	806520	8.01
Zinnia	N/A	11458	420246	786432	0.53
Roses	N/A	123820	963872	196608	4.90
Wash.ht	441800	49999	2822472	894916	3.15
Brooks	9995	9345	4944040	1449616	3.41
Capitol	7153	7153	1322980	1572864	0.84
Rad	7096	7096	1615499	1449616	1.14

Due to space limitations, we do not include photos of these scenes in the current paper. The interested reader may find some of them in their original publications; most of them appear with full trace and simulation statistics in an extended version of this paper [5]. However, the distributions of graphics primitive sizes appears in Figure 1.2. Recall from

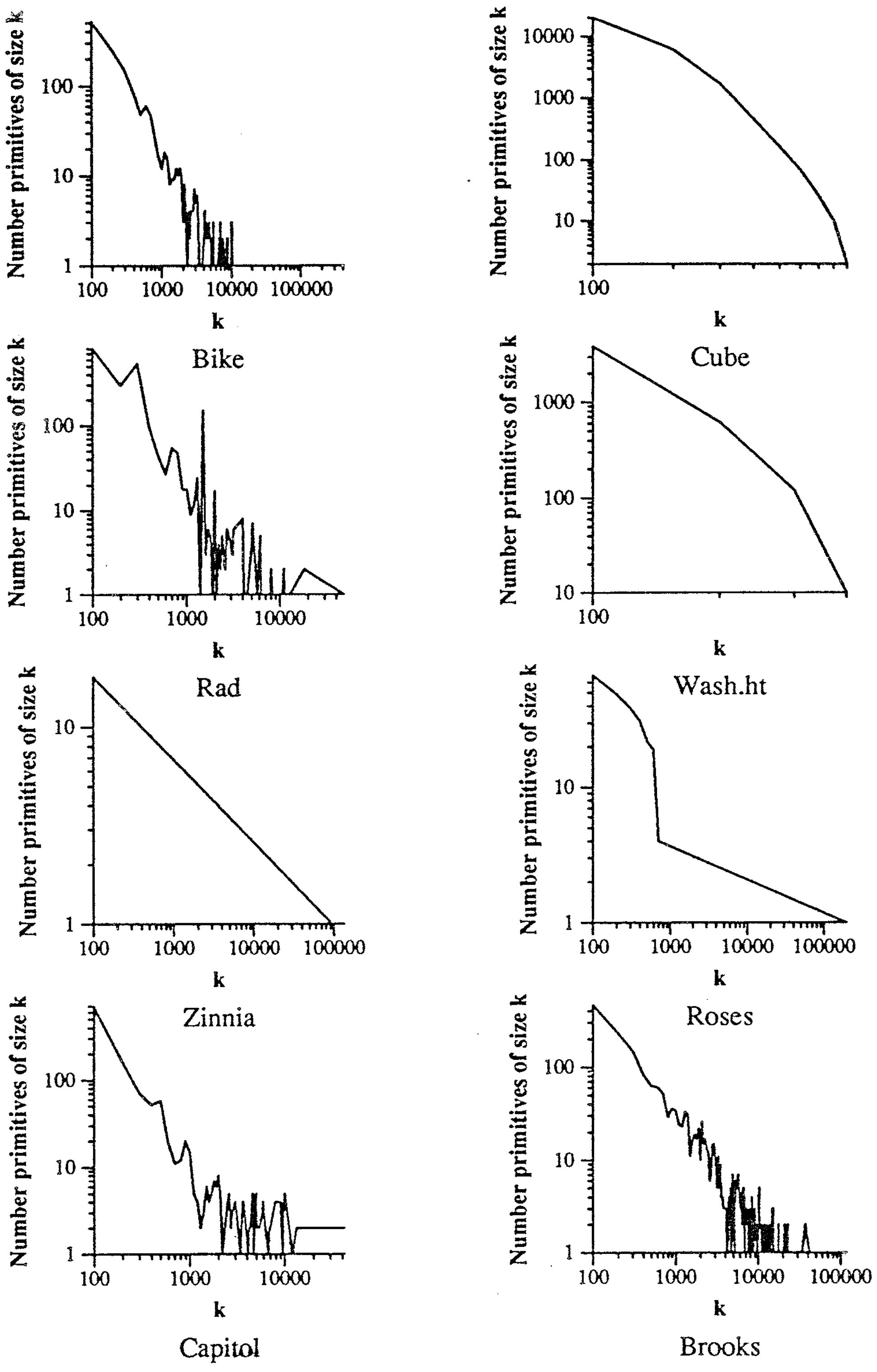


FIGURE 1.2. Size distributions of models analyzed

Section 1.2 that these distributions serve essentially as the independent variable in our model of depth complexity.

Notice in Table 1.1 and Figure 1.2 the wide range of model complexity (number of primitives, number of samples generated, screen coverage) and depth complexity, and the large variability in the underlying primitive size distributions of the models (in particular the large differences in scale of x- and y-axes).

1.4 Results

We have analyzed both predicted and observed statistics of the scenes described in Section 1.3. Predicted statistics have been computed using equations 2 and 4, with the parameter N_k taken directly from the underlying size distributions shown in Figure 1.2. Observed statistics have been computed by trace-driven simulation as discussed in Section 1.3.

1.4.1 Fit of Model

Figure 1.3 shows a trend in predicted vs. observed depth complexity that occurs in all traces evaluated in the current paper. The fit of the model of Section 1.2 may (or may not) be a good predictor of observed depth complexity in graphics scenes rendered on a uniprocessor ($n = 1$); however, the fit becomes better as n increases. In Figure 1.3 it can be seen that by $n = 8$ the fit is very close.

Recall in the model of Section 1.2 that an assumption was made that graphics primitives are uniformly and randomly distributed with respect to their placement on the screen. Clearly, not all graphics models possess this property (in particular, note the *Cube* for $n = 1$ in Figure 1.4); there may be strong spatial coherence in the organization of the scene's graphics primitives.

However, recall from Section 1.3 that in our trace-driven simulations of depth complexity, primitives have been assigned to processors in round-robin fashion in the order in which they occurred in the original graphics model. Such object assignment can be expected to destroy any spatial coherence present in an initial model; and, in fact, it appears to do so. Note the *Cube* in Figure 1.4. Even though for $n = 1$, the model predicts depth complexity far from the observed values, by the time the scene is split across 16 processors, the predicted and observed values of depth complexity are close. For many of the scenes traced, the model provides a reasonable fit even for $n = 1$ [5], and for all scenes traced provides a very good fit for very small n (further examples are shown in Figure 1.5). Furthermore, for the scenes evaluated, once the model of Section 1.3 converges with observed depth complexity, its fit remains good for larger values of n [5].

1.4.2 Average Active Samples

As each processor in an object-parallel architecture renders its subset of the graphics scene, it does not necessarily write to each pixel in its local frame buffer. In fact, while most of the graphics scenes evaluated cover most of the screen (when $n = 1$), their screen coverage steeply declines when they are split between multiple processors.

The model of Section 1.2 predicts well the observed screen coverage, especially as n increases, and closely matches the observed decline in screen coverage. This can be seen in Figures 1.6 and 1.7.

It can be seen that on average, the fraction of active samples in each processor's local frame buffer is small even for very few processors. This observation may affect two design choices in object-parallel architectures:

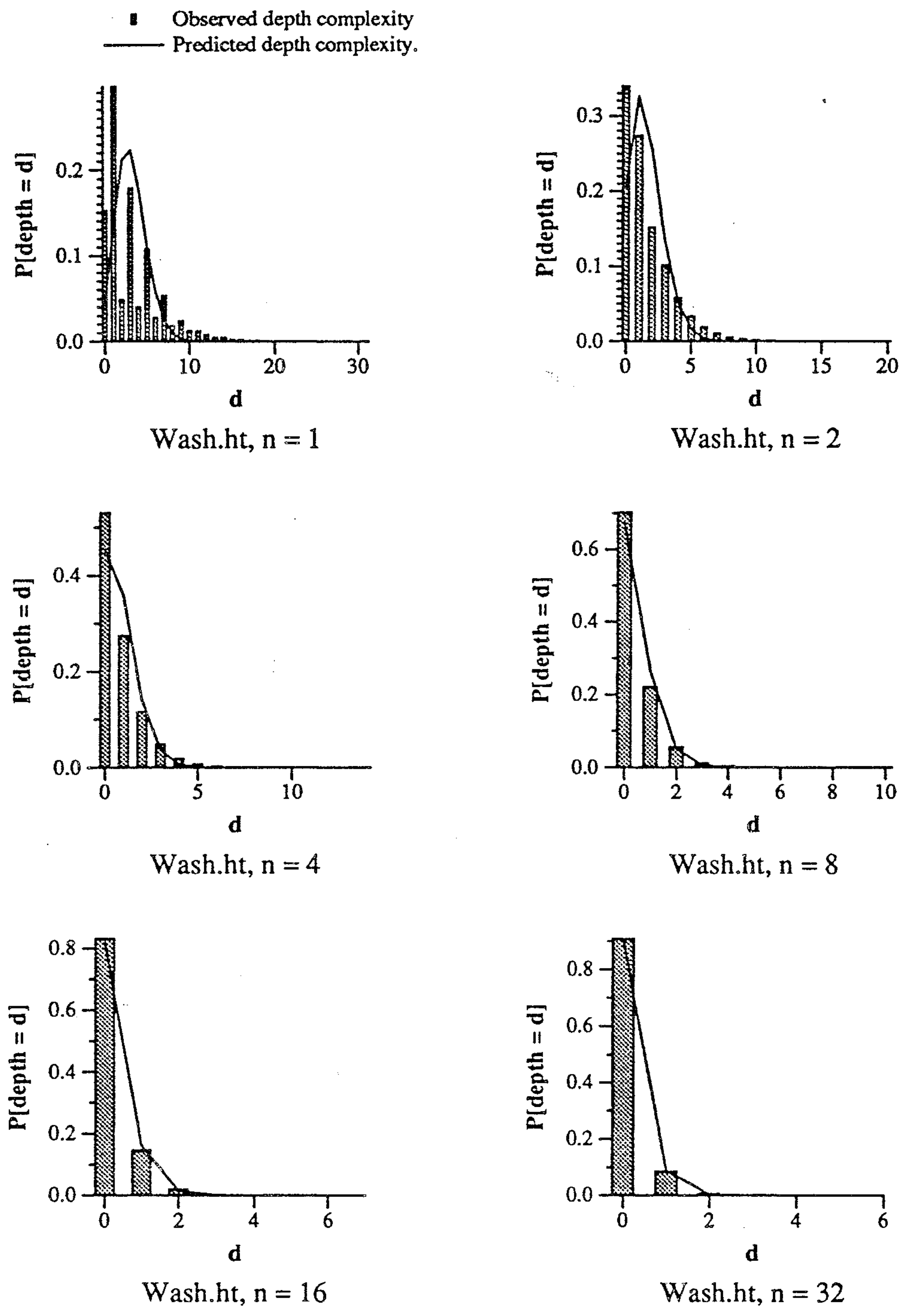


FIGURE 1.3. Depth complexity distribution in *Wash.ht*

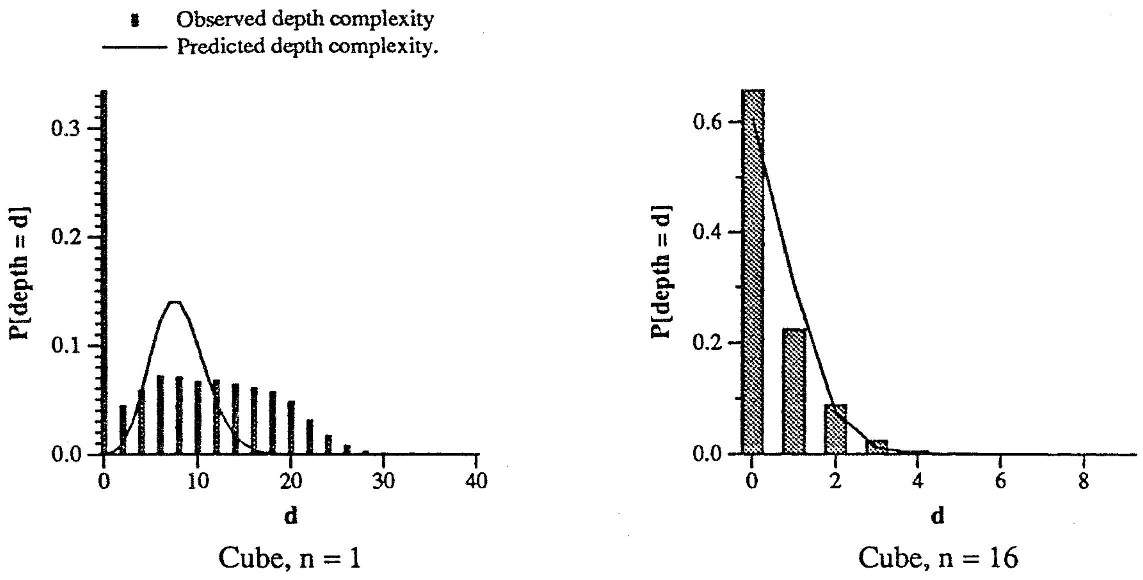


FIGURE 1.4. Depth complexity in *Cube*

- As discussed in Section 1.1.2, in object-parallel architectures, pixels produced by the multiple processors must be merged from local frame buffers across some back-end network. One strategy is to merge all pixels from all local frame buffers [14]. Alternatively, our results suggest that strategies that merge only those pixels rendered have the potential of greater speed (since the back-end network must support less traffic).

- Since only a fraction of each local frame buffer is utilized each frame, most of the local frame buffers represent unused hardware. Our results suggest that object-parallel architectures that employ sparse local frame buffers have the potential of reduced cost. Of course, such frame buffers must be cheaper themselves than standard frame buffers. Also, since some processors will render more pixels than average, local frame buffers must be sized appropriately. This is addressed briefly in the following section.

1.4.3 Active Sample Distribution and Maximum

Any object-parallel architecture that employs sparse local frame buffers in order to take advantage of the observations of the previous section must handle the worst-case.

As discussed in [5], in general, the fraction of active samples in all local frame buffers is close to the average value. However, if the fraction of active samples is large in any frame buffer, then all sparse local frame buffers must be sufficiently sized to handle this case. Clearly, there will be some frame buffer that contains as many active samples as the largest primitive, and if many primitives are large, we might expect that many local frame buffers will be substantially full.

On the other hand, if almost all local frame buffers contain very few samples, then most local frame buffers will be substantially unused. As discussed in [5], as n increases, almost all processors render about the expected number of active samples (which, as shown in Figures 1.6 and 1.7, is small). Two examples of this trend are shown in Figure 1.8. Note that for $n = 64$, only one (*Roses*) or two (*Bike*) processors need more than a fifth of their local frame buffers to render their assigned primitives.

Thus, by far the the most common case is that each processor renders very few pixels. Although each processor must be able to handle the worst case, it seems wasteful that

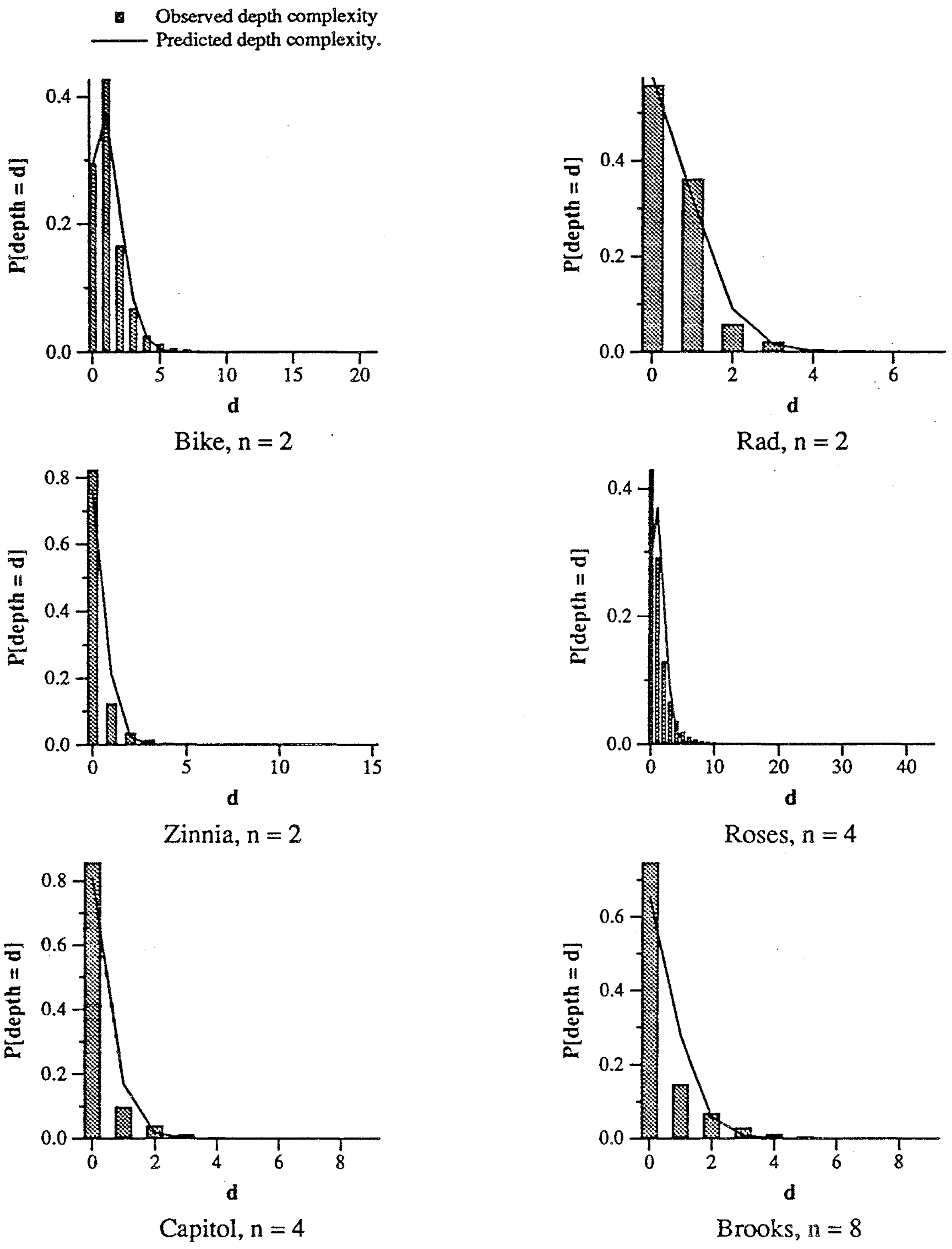
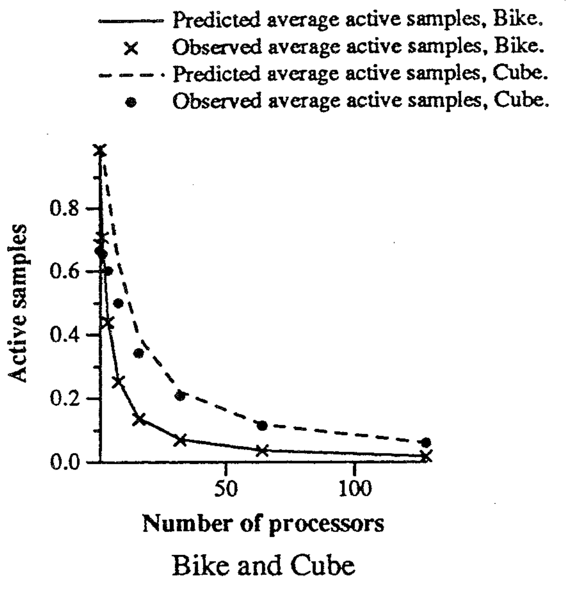


FIGURE 1.5. Depth complexity in *Bike*, *Rad*, *Zinnia*, *Roses*, *Capitol*, and *Brooks*



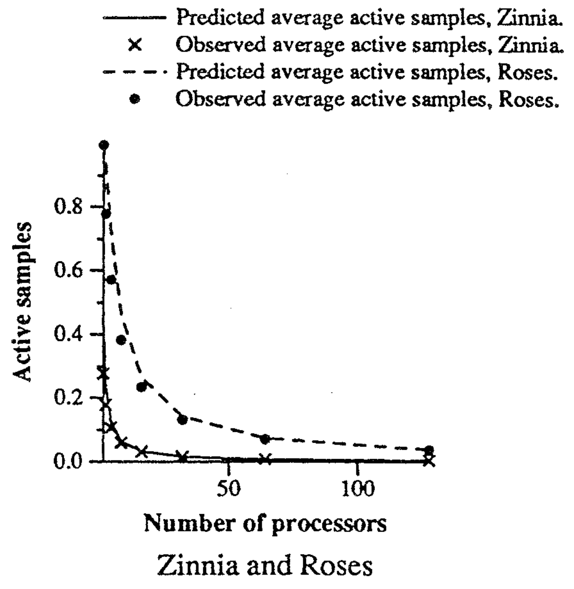
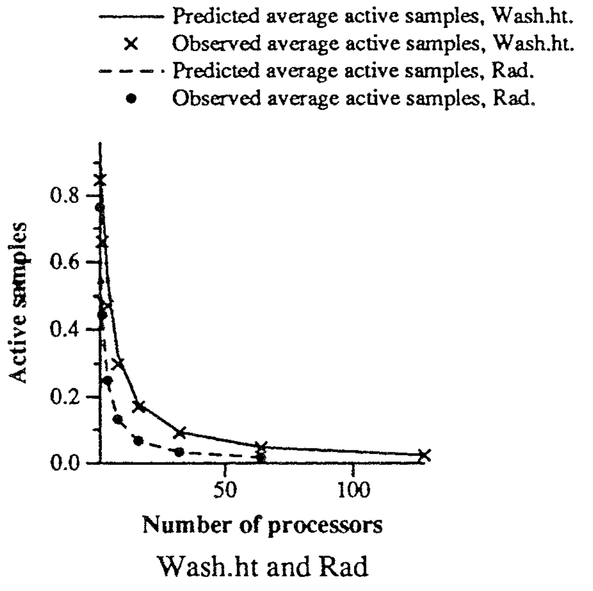


FIGURE 1.6. Active Samples, *Bike*, *Cube*, *Zinnia*, and *Roses*



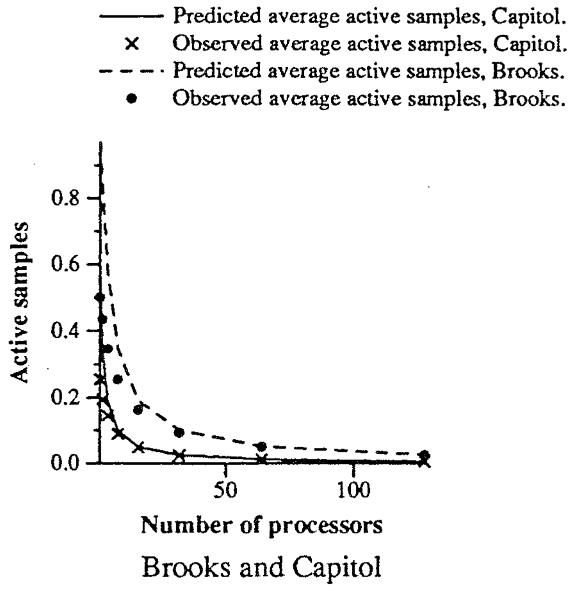


FIGURE 1.7. Active Samples, *Wash.ht*, *Rad*, *Brooks*, and *Capitol*

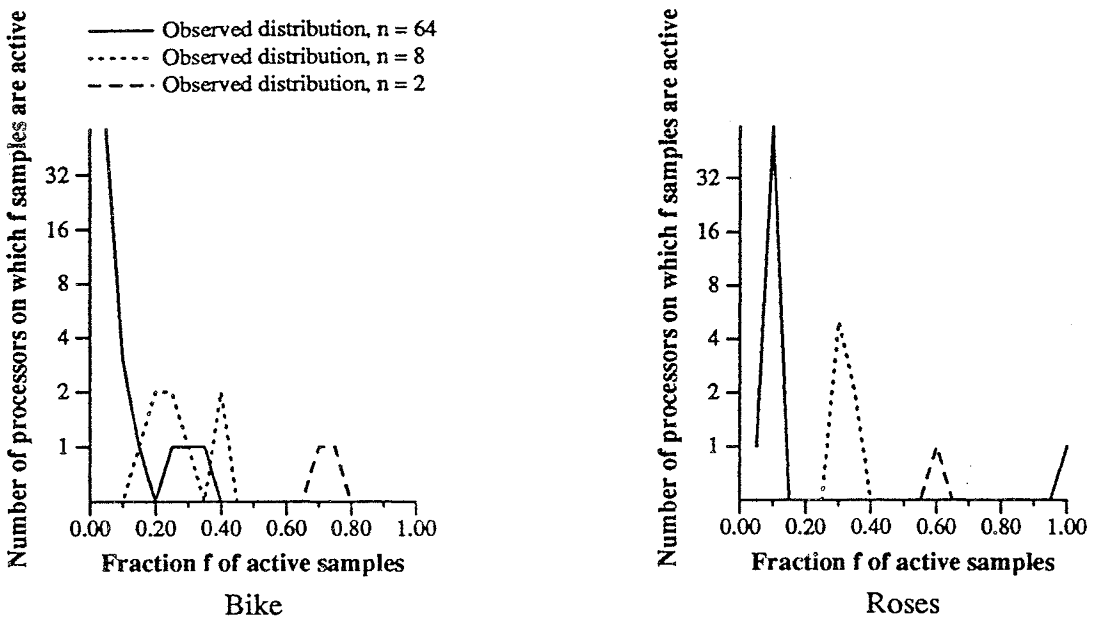


FIGURE 1.8. Per-processor Distribution of Active Samples, *Bike* and *Roses*

almost always the worst case will not occur. Perhaps the appropriate design will employ a sparse local frame buffer, with appropriate mechanism to handle spill-over.

1.4.4 Local Depth Complexity Removal

Our model and trace results show that for smaller n, especially for scenes with many primitives, local depth complexity removal can continue to play a significant role in object-parallel architectures (some scenes for $n = 8$ are shown in Figure 1.9 – note that $d = 0$ has been removed from these graphs and that the y-axis scale has changed).

On the other hand, as expected, as n grows large, very few pixels have depth greater than one [5]. For these larger machines, depth complexity removal plays essentially no role in reduction of back-end network traffic. Perhaps an object-parallel design of many processors, and no local frame buffer, can exploit this.

1.5 Conclusions

We find several results from analysis of the predicted vs. observed depth complexity distribution for the scenes discussed in this paper. These are, in summary:

- The fit of the model to observed depth complexity distributions on *uniprocessors* is sometimes good; the fit to observed depth complexity on *multiprocessors* is very good even for small n, and becomes better with increasing n.
- The model predicts well the fraction of "active samples" in local frame buffers of object-parallel multiprocessors; this predictive validity is weaker for very small n but becomes remarkably good as n increases.

  Furthermore, the model predicts well a behavior observed in trace-driven simulations: the fraction of active samples in local frame buffers declines quite quickly as n increases. This observation has consequences for back-end object-parallel merging strategies, and may have consequences for local frame buffer design.

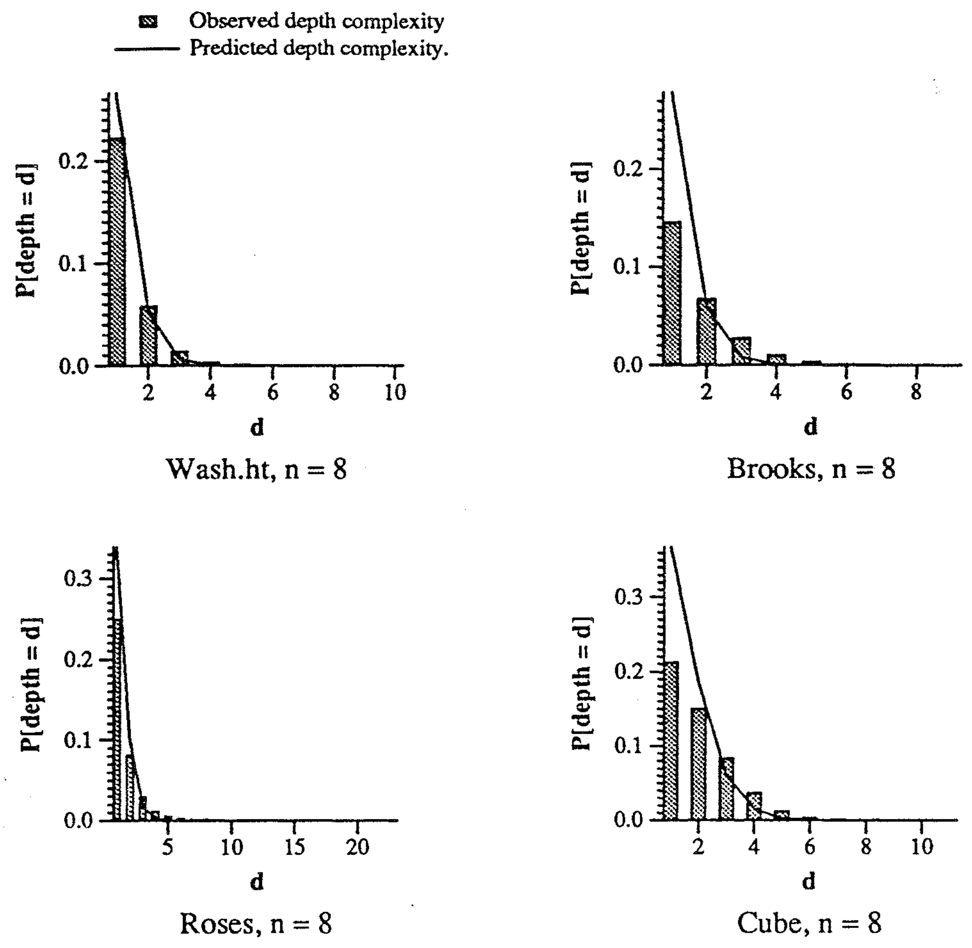


FIGURE 1.9. Depth Complexity on Small Machines

- Graphics scenes may contain one or a few very large primitives (e.g. a floor); the smallest local frame buffer in an object-parallel architecture can in some sense be no smaller than the largest such primitive.

  However, the observed distribution of depth complexity across n processors in the scenes traced suggests that most local frame buffers in an object-parallel architecture need be only a small fraction of a full frame buffer. This observation suggests that local frame buffers may be most cost-efficient when they are small, with an additional mechanism for spill-over of (infrequent) large primitives.

- On small machines, local depth complexity removal can be significant in reducing what would otherwise be back-end traffic in an object-parallel architecture. On the other hand, while local depth complexity removal in larger machines is insignificant in back-end traffic removal, vanishingly few pixels in these large machines' local frame buffers have depth greater than one; this observation suggests that large machines with no local frame buffers warrant investigation.

Acknowledgements:

We are grateful to Joel Friedman of Princeton, whose suggestion to consider multinomials led to a nice improvement of equation 1 over a previous version; our thanks also go to Michael Hirsch of Princeton, whose suggestion to take subsequent derivatives and evaluate at zero is implemented in Appendix .1

We are grateful to Tony Apodaca of Pixar for the source code required to trace RenderMan, and for the AutoCad output that is the *Capitol* of Section 1.3. Thanks to Craig Kolb of Princeton, whose help with Rayshade allowed *Zinnia*, *Roses*, and *Wash.ht* to be traced, and many thanks to Deborah Fowler and Przemyslaw Prusinkiewicz of the University of Calgary who provided the elegant *Zinnia* and *Roses*.

.1 Coefficients of $G(v)$

Let

$[v^d]G(v) =$ the coefficient of v^d of $G(v)$.

$d^{\underline{m}} = d$ to the "falling power of m", that is $\prod_{i=0}^{m-1}(d-i)$

As discussed in Section 1.2, $p_d = [v^d]G(v)$. That is

$$G(v) = \sum_{d=0}^{N} p_d \; v^d$$

We will approach the extraction of $[v^d]G(v)$ as follows. It can be verified that the m^{th} derivative of G, $G^{(m)}$ is of the form

$$G^{(m)}(v) = \sum_{m=0}^{d} d^{\underline{m}} \; p_d \; v^{d-m}$$

Thus in general

$$p_d = [v^{d-m}]G^{(m)}(v)/d^{\underline{m}}$$

and in particular

$$p_d = G^{(d)}(0)/d!$$

Thus, we can extract the coefficients of $G(v)$ by repeated differentiation and evaluation at zero.

Logarithmic differentiation of G yields a form for G' in terms of G and some F. Subsequent derivatives of G' yield a form in terms of (again) G and in terms of some function of higher derivatives of F. We will proceed by finding a closed form for the derivatives of F, which therefore provides us with a form for the derivatives of G.

Differentiating the logarithm of equation 1 yields $G'(v)/G(v)$, and

$$G'(v) = G(v)F(v)$$

where

$$F(v) = \sum_{k=1}^{A} N_k \frac{k}{An-k+kv}$$

Lemma 1:

$$F^{(j)}(v) = \sum_{k=1}^{A} (-1)^j \; j! \; N_k \left[\frac{k}{An-k+kv}\right]^{j+1} \qquad \textit{when } j \geq 0 \tag{5}$$

Proof: As basis, note that

$$F'(v) = \sum_{k=1}^{A} -N_k \left[\frac{k}{An-k+kv}\right]^2$$

The inductive step is simply

$$\begin{aligned} F^{(j+1)}(v) &= \textstyle\sum_{k=1}^{A} -(j+1)(-1)^j\, j!\, N_k\, k\, k^{j+1} \left[\frac{1}{An-k+kv}\right]^{j+2} \\ &= \textstyle\sum_{k=1}^{A} (j+1)!(-1)^{j+1} N_k \left[\frac{k}{An-k+kv}\right]^{j+2} \end{aligned}$$

Now, for readability, let $G = G(v)$, $G^{(m)} = G^{(m)}(v)$, $F = F(v)$, and $F^{(m)} = F^{(m)}(v)$.

Lemma 2:

$$G^{(m)} = \textstyle\sum_{j=0}^{m-1} \begin{pmatrix} m-1 \\ j \end{pmatrix} F^{(j)} G^{(m-1-j)} \tag{6}$$

Proof: As basis, note that

$$\begin{aligned} G^{(1)} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} F^{(0)} G^{(0)} \\ &= F^{(0)} G^{(0)} \end{aligned}$$

Differentiating $G^{(m)}$, we find that

$$\begin{aligned} G^{(m+1)} &= \textstyle\sum_{j=0}^{m-1} \begin{pmatrix} m-1 \\ j \end{pmatrix} F^{(j+1)} G^{(m-1-j)} \\ &+ \begin{pmatrix} m-1 \\ j \end{pmatrix} F^{(j)} G^{(m-j)} \end{aligned}$$

Separating these two terms into separate summations, substituting $(j-1)$ for j in the first summation, recombining terms, and discarding terms that are 0, we have that

$$\begin{aligned} G^{(m+1)} &= \textstyle\sum_{j=0}^{m} \begin{pmatrix} m-1 \\ j-1 \end{pmatrix} F^{(j)} G^{(m-j)} + \begin{pmatrix} m-1 \\ j \end{pmatrix} F^{(j)} G^{(m-j)} \\ &= \textstyle\sum_{j=0}^{m} \begin{pmatrix} m \\ j \end{pmatrix} F^{(j)} G^{(m-j)} \end{aligned}$$

Finally, substituting equation 5 in equation 6, and setting $v = 0$, we have equations 2 and 3.

.2 References

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

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

[3] F. Baskett, Silicon Graphics Inc., personal communication, 1991.

[4] F. Brooks, "Fred Brooks' House", Dataset from Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

[5] M. Cox, P. Hanrahan, "Depth Complexity in Object-Parallel Graphics Architectures," Tech. Report No. 382-92, Department of Computer Science, Princeton University, Princeton NJ, September 1992.

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

[7] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Computer Graphics*, v. 23, no. 3, July 1989.

[8] N. Gharachorloo, S. Gupta, E. Hokenek, P. Balasubramanian, B. Bogholtz, C. Mathieu, C. Zoulas, "Subnanosecond Pixel Rendering with Million Transistor Chips", *Computer Graphics*, v. 22, no. 4, August 1988.

[9] N. Gharachorloo, S. Gupta, R. F. Sproull, I. E. Sutherland, "A Characterization of Ten Rasterization Techniques", *Computer Graphics*, v. 23, no. 3, July 1989.

[10] P. Hanrahan, D. Salzman, and L. Aupperle, "A Rapid Hierarchical Radiosity Algorithm", *Computer Graphics*, v. 25, no. 4, July 1991.

[11] C. E. Kolb, *Rayshade User's Guide and Reference Manual*, Draft 0.4, Department of Computer Science, Princeton University, Princeton NJ, 1992.

[12] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Math*, Addison-Wesley, Reading MA, 1989.

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

[14] S. Molnar, *Image-Composition Architectures for Real-Time Image Generation*, TR91-046, University of North Carolina at Chapel Hill, October 1991.

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

[16] M. Potmesil, E.M. Hofert, "The Pixel Machine: A Parallel Image Computer", *Computer Graphics*, v. 23, no. 3, July 1989.

[17] P. Prusinkiewicz and A. Lindenmayer, with J. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer, *The Algorithmic Beauty of Plants*, Springer-Verlag, New York NY, 1990.

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

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

[20] C. Shaw, M. Green, J. Schaeffer, "A VLSI Architecture for Image Composition,", in *Advances in Computer Graphics Hardware III*, Springer-Verlag, New York NY, 1988.

[21] U.S. Geological Survey, "Digital Elevation Model (DEM)" datasets, contact Earth Science Information Center, USGS, 507 National Center, Reston VA 22092.

[22] S. Upstill, *The RenderMan Companion*, Addison-Wesley, Reading MA, 1989.

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

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

[25] H.S. Wilf, *Generatingfunctionology*, Academic Press Inc., San Diego CA, 1990.