

A Sorting Classification of Parallel Rendering

Steven Molnar

University of North Carolina at Chapel Hill

Michael Cox

Princeton University

David Ellsworth and Henry Fuchs

University of North Carolina at Chapel Hill

This conceptual model defines three broad sorting classes that illuminate the trade-offs between different parallel rendering approaches.

Graphics rendering is notoriously compute intensive, particularly for realistic images and fast updates. Demanding applications, such as scientific visualization, CAD, vehicle simulation, and virtual reality, can require hundreds of Mflops of performance and gigabytes per second of memory bandwidth—far beyond the capabilities of a single processor. For these reasons, parallelism has become a crucial tool for building high-performance graphics systems, whether special-purpose hardware systems or software systems for general-purpose multicomputers.

We can employ parallelism of various types at many levels. For example, functional parallelism (pipelining) can speed critical calculations, and data parallelism can allow multiple results to be computed at once. Common data-parallel approaches are by object (object parallelism) and by pixel or portion of the screen (pixel or image parallelism).

Several taxonomies of parallel rendering algorithms have been proposed.^{1,4} These taxonomies help in classifying and understanding systems but do not lend themselves easily to comparison or analysis. Some rendering systems have been analyzed in isolation.^{5,7} However, these analyses tend to focus on unique system attributes and make comparisons between systems difficult.

In this article, we describe a classification scheme that we believe provides a more structured framework for reasoning about parallel rendering. The scheme is based on where the sort from object coordinates to screen coordinates occurs, which we believe is fundamental whenever both geometry processing and

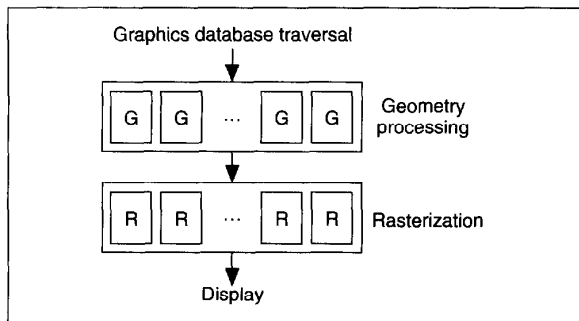
rasterization are performed in parallel. This classification scheme supports the analysis of computational and communication costs, and encompasses the bulk of current and proposed highly parallel renderers—both hardware and software.

We begin by reviewing the standard feed-forward rendering pipeline, showing how different ways of parallelizing it lead to three classes of rendering algorithms. Next, we consider each of these classes in detail, analyzing their aggregate processing and communication costs, possible variations, and constraints they may impose on rendering applications. Finally, we use these analyses to compare the classes and identify when each is likely to be preferable.

About the title photograph

The photograph above shows the simulation results of communications traffic between 36 sort-first processors in a 2D mesh rendering the National Computer Graphics Association's "rotating head" Picture-Level Benchmark (*GPC Quarterly Report*, Vol. 2, No. 4, 1992).

Arrow color indicates the number of primitives transferred between processors for the 4.5 degree rotation of the head model between these two successive frames. Range is 0 (black) to 800 (white) using a heated-object spectrum.



Parallel rendering as a sorting problem

Figure 1 shows a simplified version of the standard, feed-forward rendering pipeline, adapted for parallel rendering. It consists of two principal parts: geometry processing (transformation, clipping, lighting, and so on) and rasterization (scan-conversion, shading, and visibility determination). In this article, we target rendering rates high enough that both geometry processing and rasterization must be performed in parallel. We say such systems are *fully parallel*.

Geometry processing usually is parallelized by assigning each processor a subset of the primitives (objects) in the scene. Rasterization usually is parallelized by assigning each processor a portion of the pixel calculations.

The essence of the rendering task is to calculate the effect of each primitive on each pixel. Due to the arbitrary nature of the modeling and viewing transformations, a primitive can fall anywhere on (or off) the screen. Thus, we can view rendering as a problem of sorting primitives to the screen, as noted by Sutherland, Sproull, and Schumacher in their seminal paper on visible-surface algorithms.⁸ For fully parallel renderers, this sort involves a redistribution of data between processors, because responsibility for primitives and pixels is distributed.

The location of this sort largely determines the structure of the resulting parallel rendering system. Understanding the variety of system structures possible within the constraints of this distributed sort and realizable with available computational resources is the main challenge for designers of fully parallel rendering systems.

The sort can, in general, take place anywhere in the rendering pipeline: during geometry processing (sort-first), between geometry processing and rasterization (sort-middle), or during rasterization (sort-last). Sort-first means redistributing "raw" primitives—before their screen-space parameters are known. Sort-middle means redistributing screen-space primitives. Sort-last means redistributing pixels, samples, or pixel fragments.

Each of these choices leads to a separate class of parallel rendering algorithms with distinct properties. We describe the classes briefly now and examine them in more detail later.

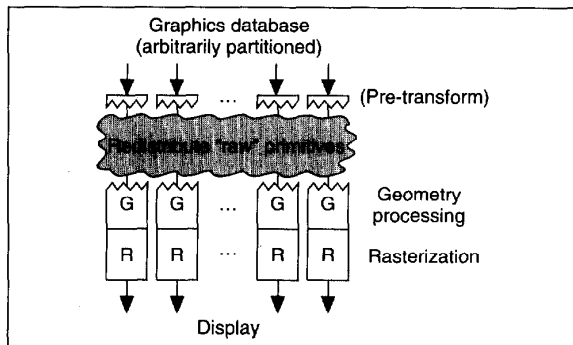
Sort-first

The aim in sort-first is to distribute primitives early in the rendering pipeline—during geometry processing—to processors that can do the remaining rendering calculations (Figure 2). We can do this by dividing the screen into disjoint regions and making processors (called *renderers*) responsible for all rendering calculations that affect their respective screen regions.

Initially, we assign primitives to renderers in some arbitrary

Figure 1. Graphics pipeline in a fully parallel rendering system. Processors *G* perform geometry processing. Processors *R* perform rasterization.

Figure 2. Sort-first redistributes raw primitives during geometry processing.



fashion. When rendering begins, each renderer does enough transformation to determine into which region(s) each primitive falls, generally computing the screen-space bounding box of the primitive. We call this *pre-transformation*, and it may or may not involve actually transforming the primitive. In some cases, primitives will fall into the screen regions of renderers other than the one on which they reside. These primitives must then be redistributed over an interconnect network to the appropriate renderer (or renderers), which then perform the remainder of the geometry-processing and rasterization calculations for these primitives.

This redistribution of primitives at the beginning of the rendering process is the distinguishing feature of sort-first. It clearly involves overhead, since it performs some geometry processing for some primitives on the wrong renderer. The results of these calculations must either be sent to or recomputed on the new renderer(s).

Sort-first is the least explored of the three classes. To our knowledge, no one has built a sort-first system. Although sort-first may seem impractical at first, we will see later that it can require much less communication bandwidth than the other approaches, if primitives are tessellated or if frame-to-frame coherence can be exploited. We will discuss its potential advantages and disadvantages in more detail later.

Sort-middle

In sort-middle, primitives are redistributed in the middle of the rendering pipeline—between geometry processing and rasterization. Primitives at this point have been transformed into screen coordinates and are ready for rasterization. Since many systems perform geometry processing and rasterization on separate processors, this is a natural place to break the pipeline.

In a sort-middle system, geometry processors are assigned arbitrary subsets of the primitives to be displayed; rasterizers are assigned a portion of the display screen (generally a contiguous region of pixels, as in sort-first). The two processor groups can be separate sets of processors, or they can time-share the same physical processors.

During each frame, geometry processors transform, light,

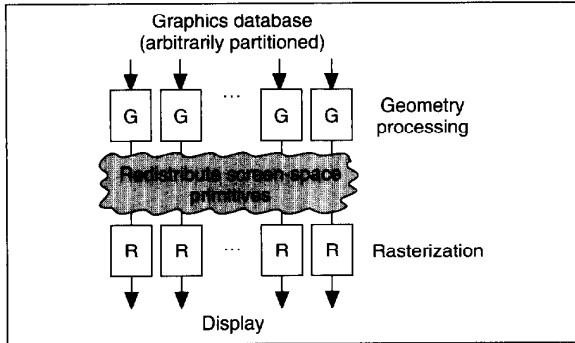


Figure 3. Sort-middle redistributes screen-space primitives between geometry processing and rasterization.

and perform other geometric operations on their portion of the primitives, and classify them with respect to screen region boundaries. Then they transmit all these screen-space primitives to the appropriate rasterizer or rasterizers, as shown in Figure 3.

Sort-middle is general and straightforward. It has been the most common approach to date for both hardware⁹⁻¹¹ and software^{6,7,12} parallel rendering systems. We will examine its advantages and disadvantages in more detail later.

Sort-last

Sort-last defers sorting until the end of the rendering pipeline—after primitives have been rasterized into pixels, samples, or pixel fragments. Sort-last systems assign each processor (renderer) arbitrary subsets of the primitives. Each renderer computes pixel values for its subset, no matter where they fall in the screen. Renderers then transmit these pixels over an interconnect network to compositing processors, which resolve the visibility of pixels from each renderer (Figure 4).

In sort-last, renderers operate independently until the visibility stage—the very end of the rendering pipeline. The interconnect network, however, must handle all the pixel data generated on all the renderers. For interactive or real-time applications rendering high-quality images, this can result in very high data rates.

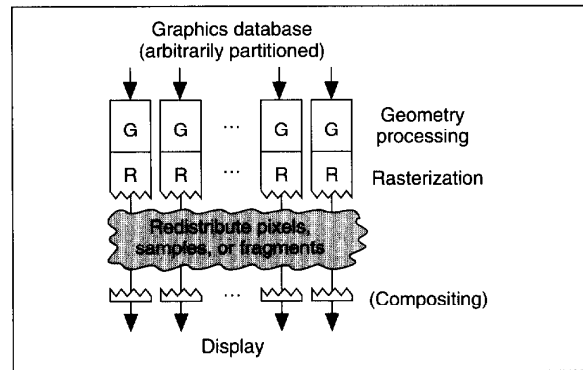
Sort-last can be done in at least two ways. One approach, which we call *SL-sparse*, minimizes communication by distributing only those pixels actually produced by rasterization. The second approach, called *SL-full*, stores and transfers a full image from each renderer. Both methods have advantages, as we will see later.

Sort-last systems have existed in various forms for more than 20 years. The 1967 GE NASA II flight simulator used a simple version of *SL-full* that assigned a processor to each primitive.¹³ Since then, several primitive-per-processor^{3,14} and multiple-primitive-per-processor^{15,16} *SL-full* systems have been proposed. Several recent commercial systems have used the *SL-sparse* approach.¹⁷⁻¹⁹

Processing and communication model

We now analyze each of the three rendering methods in more detail. The aim is to build a quantitative model of their processing and communication costs to use as a basis for comparing them. We first consider the processing required to render on a uniprocessor. Then we evaluate the additional processing and communication requirements of the parallel methods. We focus

Figure 4. Sort-last redistributes pixels, samples, or pixel fragments during rasterization.



Terms used in the analysis

A	The resolution of the screen, in pixels.
N	The number of processors in the multiprocessor.
n_r, n_d	The number of raw and display primitives, respectively.
a_r, a_d	The average size, in pixels, of a raw and display primitive.
O_r, O_d	The average number of screen regions that raw and display primitives overlap.
f_r, f_d	The fraction of pre-tessellation and post-tessellation geometry processing that is duplicated when a raw primitive overlaps multiple screen regions.
T	The tessellation ratio, n_d/n_r .
S	The number of samples per pixel.
c	The fraction of raw primitives that must be redistributed each frame when sort-first takes advantage of frame-to-frame coherence.

on the inherent overhead in the methods and do not specifically model factors such as load balancing, buffering, and transport delay. Such factors affect performance significantly, but depend on the detailed system implementation and the graphics scene itself. We will, however, discuss these factors (particularly load balancing) where appropriate.

See the sidebar for definitions of terms used in the analyses.

Uniprocessor pipeline

For the analysis that follows, we refine the rendering pipeline as shown in Figure 5 on the next page. First, some rendering systems tessellate primitives to generate higher quality images (RenderMan is one widely used example). Tessellation decomposes larger primitives into smaller ones, typically into polygons or polygonal meshes. Not all rendering packages perform tessellation, but we include it in the pipeline because it can greatly expand the number of primitives that need to be displayed.

Pipeline stage	Processing cost
Geometry:	
Pre-tessellation	$n_r \llcorner \text{geom-pre-tess} \gg +$
Post-tessellation	$n_d \llcorner \text{geom-post-tess} \gg$
Pixel rendering	
	$n_d \llcorner \text{rend-setup} \gg +$
	$n_d a_d S \llcorner \text{rend} \gg$
Visibility	
	$n_d a_d S \llcorner \text{comp} \gg$

Second, we break rasterization into two stages called pixel rendering (computing pixel values) and visibility (determining which pixels are visible), since some algorithms perform these on separate processors. Finally, we do not explicitly mention shading, which can be a major consumer of processing cycles, but can occur almost anywhere in the pipeline. Shading should be considered a part of geometry processing or of pixel rendering, as appropriate.

We assume that we are rendering a data set containing n_r raw primitives with average size a_r . We will call primitives that result from tessellation *display primitives*. If T is the tessellation ratio, there are $n_d = T n_r$ display primitives, with average size $a_d = a_r / T$. If there is no tessellation, $T = 1$, $n_d = n_r$, and $a_d = a_r$. We assume that we are rendering an image containing A pixels and that we are to compute S samples per pixel. For simplicity, we assume that all primitives are potentially visible (that is, lie within the viewing frustum).

Processing costs

Figure 5 lists the processing costs for each stage of the uniprocessor rendering pipeline. First, the n_r raw primitives are processed by the stages of the geometry pipeline up to tessellation. The cost for this is $n_r \llcorner \text{geom-pre-tess} \gg$ (“ $\llcorner \dots \gg$ ” denotes cost, most naturally in units of time). The resulting n_d display primitives are then processed by the geometry pipeline stages following tessellation, at a cost of $n_d \llcorner \text{geom-post-tess} \gg$. Rasterization follows. Pixel rendering is the first stage and consists of two parts: setup and per-sample rendering operations, whose costs are $n_d \llcorner \text{rend-setup} \gg$ and $n_d a_d S \llcorner \text{rend} \gg$, respectively. Finally, the cost of compositing is $n_d a_d S \llcorner \text{comp} \gg$, where “ $\llcorner \text{comp} \gg$ ” is the cost of compositing one sample (generally a z comparison and conditional write). There is no data redistribution or “sort” in the uniprocessor model.

Sort-first analysis

We begin our analysis with sort-first. Figure 6 shows its rendering pipeline and what it costs beyond uniprocessor rendering. Our analysis assumes that primitives are redistributed as early in the rendering pipeline as possible and that renderers discard transformed data when they send a primitive to a new renderer. (An alternative is to send the transformed data. Although we will not focus on this alternative, its analysis is similar to the case we present here.)

Processing and communication costs

The first steps in sort-first are to “pre-transform” the raw primitives so that their screen extents are known and to classify them with respect to screen-region buckets. Each bucket belongs to a region, and a primitive falls in several buckets when

Figure 5. Rendering pipeline and processing costs in a uniprocessor implementation.

Figure 6. Sort-first processing and communication overhead (communication costs indicated by box).

Pipeline stage	SF overhead
Pre-transform	$n_r \llcorner \text{pre-xform} \gg$
Bucketization	$n_r O_r \llcorner \text{bucket} \gg$
Redistribution	$c n_r O_r \llcorner \text{prim} \gg$
Geometry:	
Pre-tessellation	$n_d (O_r - 1) f_r \llcorner \text{geom-pre-tess} \gg +$
Post-tessellation	$n_d (O_d - 1) f_d \llcorner \text{geom-post-tess} \gg$
Pixel rendering	$n_d (O_d - 1) \llcorner \text{rend-setup} \gg +$
Visibility	—

it overlaps several regions. We define an *overlap factor* O_r , which is the average number of regions a raw primitive overlaps. If the cost to precompute a primitive’s screen coordinates is “ $\llcorner \text{pre-xform} \gg$ ” and the cost to put the primitive in each of its buckets is “ $\llcorner \text{bucket} \gg$,” then the overhead for these stages is $n_r \llcorner \text{pre-xform} \gg$ and $n_r O_r \llcorner \text{bucket} \gg$.

Next, primitives on the wrong renderer must be distributed to the correct renderer(s). The number of primitives redistributed depends on the application and whether frame-to-frame coherence is employed. For example, if we are rendering a single frame, almost all the primitives must be sent. However, if we render multiple frames in an animation or real-time application and the scene does not change much between frames, we may need to send only a small fraction of the primitives. (Note that this type of coherence is only available in sort-first, as it is the only technique that distributes raw primitives. The other methods distribute data that has undergone view-dependent processing.)

To take this application-dependent behavior into account, we define the parameter c , the fraction of primitives that must be redistributed. The total network bandwidth required is then $c n_r O_r \llcorner \text{prim} \gg$, where “ $\llcorner \text{prim} \gg$ ” is the size of the data structure representing a raw primitive. We will not explicitly tally the processing cost of communication, but note that it is proportional to this term.

After redistribution, sort-first algorithms may accrue other overhead that a uniprocessor pipeline would not. If a raw primitive falls in exactly one bucket, then it undergoes geometry processing exactly once, and the costs are the same as they would be on a uniprocessor. If the primitive overlaps more than one region, however, there will be duplication of effort. Each additional processor responsible for a given raw primitive must duplicate some fraction f_r of pre-tessellation geometry processing and some fraction f_d of post-tessellation geometry processing. These fractions will depend on the algorithms chosen for geometry processing. For example, explicit clipping to region boundaries requires duplication of effort, while implementations that avoid clipping avoid this extra work. From these considerations, the additional cost of geometry processing for the

Pipeline stage	SM overhead
Geometry	—
Bucketization	$n_d O_d \llbracket \text{bucket}_d \rrbracket$
Redistribution	$n_d O_d \llbracket \text{prim}_d \rrbracket$
Pixel rendering	$n_d (O_d - 1) \llbracket \text{rend-setup} \rrbracket$
Visibility	—

n_r raw primitives is $n_r(O_r - 1)f_r$ «geom-pre-tess». For the Tn_d display primitives, it is $n_d(O_d - 1)f_d$ «geom-post-tess».

Finally, each display primitive that overlaps several screen regions requires pixel rendering setup for each region. This adds processing overhead of $n_d(O_d - 1)$ «rend-setup».

Frame-to-frame coherence

If we are rendering a single frame, most primitives will be on the wrong renderer and will require redistribution—unless we are very lucky. Hence, c will be close to 1. However, if we render several related frames in sequence, we may be able to take advantage of frame-to-frame coherence to reduce overhead. To do this, processors that render a primitive during one frame must retain that primitive for the next frame. If there is significant spatial coherence between frames, fewer primitives will require redistribution during the next frame and c will be close to 0. Of course, the application must support retained-mode (display-list) rendering, and bookkeeping is needed to track primitive ownership.

To get an estimate of c , we have analyzed traces from two rendering sequences: the rotating head model in the NCGA Graphics Performance Committee's Picture-Level Benchmark²⁰ (see the title-page photo) and an architectural walkthrough of a building interior seen through a head-mounted display. In both cases, we assumed a display size of $1,280 \times 1,024$ pixels and a region size of 64×64 pixels. In the Picture-Level Benchmark, the head contains 60,000 triangles and rotates 4.5 degrees each frame. Here, c varied between 0.13 and 0.16. The architectural model contained 64,796 triangles. During a 60-second exploration of a room within the model, c varied between 0.0 and 0.64 with a mean value of 0.15. Thus, we see that c can be quite small in practice, making sort-first appealing for applications with substantial coherence.

Tessellation and oversampling

If a system employs tessellation and oversampling, then each raw primitive generates T display primitives, and each of these in turn generates $a_d S$ samples. This means that for each raw primitive redistributed by sort-first, sort-middle must redistribute T display primitives and sort-last must redistribute $T a_d S$ samples. If T and S are large, it may make the most sense to redistribute raw primitives, since there are far fewer of them. We will see after considering sort-middle and sort-last that sort-first has the lowest cost under these circumstances.

Load balancing

Sort-first is susceptible to several types of load imbalance (as are the other two approaches). The most obvious is the initial distribution of primitives across processors. Even if equal numbers of primitives are assigned to each processor, primitives may require different amounts of work, so load imbalances can result.

Figure 7. Sort-middle processing and communication overhead (communication costs indicated by box).

A second type of load imbalance arises from the distribution of primitives over the screen. In sort-first, some regions of the screen will very likely receive many more primitives than others. Also, some primitives may take longer to process than others. A way to combat this is to make regions smaller and make each processor responsible for more than one region. This can be done statically, which is simple, but might not achieve an optimal load distribution for any given frame. It can also be done dynamically, which increases algorithm complexity but may achieve better results. When using frame-to-frame coherence, we must constrain dynamic load balancing so that each region is assigned to the same processor in successive frames. Also, dividing the screen more finely to improve the load balance tends to increase c .

Advantages and disadvantages

In summary, sort-first has two advantages:

- Communication requirements are low when the tessellation ratio and the degree of oversampling are high, or when frame-to-frame coherence can be exploited.
- Processing nodes implement the entire rendering pipeline for a portion of the screen.

Its disadvantages, however, are

- Susceptibility to load imbalance. Primitives may clump into regions, concentrating the work on a few renderers.
- Necessity of retained mode and complex data handling code to take advantage of frame-to-frame coherence.

Sort-middle analysis

Sort-middle algorithms are in some sense the most natural. In contrast to sort-first, they redistribute primitives after geometry processing, that is, after screen coordinates have already been computed. Figure 7 shows the sort-middle rendering pipeline and its additional costs.

Processing and communication costs

In sort-middle, the cost of geometry processing is the same as on a uniprocessor. Sort-middle algorithms first accrue overhead when they redistribute display primitives. Each of the n_d primitives must be placed into O_d buckets at a total cost of $n_d O_d \llbracket \text{bucket}_d \rrbracket$. The bandwidth required to send these buckets is $n_d O_d \llbracket \text{prim}_d \rrbracket$, and the processing cost of communication is proportional. After redistribution, display primitives that overlap more than one region require extra pixel rendering setup. The cost for this is $(O_d - 1)$ times the cost on a uniprocessor. Visibility calculations cost the same in sort-middle as they would on a uniprocessor.

Figure 8. Sort-last processing and communication overhead (communication costs indicated by box).

Pipeline stage	SL-sparse overhead	SL-full overhead
Geometry	—	—
Pixel rendering	—	—
Redistribution	$n, a, S \llcorner \text{sample} \gg$	$NAS \llcorner \text{sample} \gg$
Visibility	—	$NAS \llcorner \text{comp} \gg$

From Figure 7 we can see that the overhead for sort-middle depends on the number of display primitives n_d and on the display-primitive overlap factor O_d . These in turn depend on the degree of tessellation.

Tessellation

Since $n_d = Tn_r$, the overhead of bucketization and redistribution in sort-middle depends critically on the tessellation ratio T . If T is large, sort-middle transfers a large number of display primitives. If there is no tessellation or T is small, sort-middle transfers roughly the same number of display primitives as there are raw primitives. Sort-first transfers raw primitives, so T must be small for sort-middle to compare favorably.

Load balancing

Like sort-first, sort-middle can suffer load imbalances from object assignment and from primitives clumping into regions. A static load-balancing scheme for hierarchical display lists has been shown to be effective on systems with up to 40 processors.²¹ The other problem, primitive clumping, has been the focus of much research in hardware²² and software^{4,7,12} sort-middle renderers. The main techniques are to make regions smaller and to assign regions dynamically to processors. The former tends to increase the overlap factor O_d and should therefore be applied with care.

Advantages and disadvantages

In summary, sort-middle has the advantage of being general and straightforward. Redistribution occurs at a natural place in the pipeline.

Its disadvantages are

- High communication costs if the tessellation ratio is high.
- Susceptibility to load imbalance between rasterizers when primitives are distributed unevenly over the screen.

Sort-last analysis

Sort-last algorithms are perhaps the most variable of the three classes. First, there is the difference between sparse merging and full-frame merging. Sparse merging takes advantage of the observation that renderers in a sort-last system may generate pixels for only a fraction of the screen, and only these pixels need be merged.²³ Full-frame merging takes advantage of the fact that merging a full frame from each processor is very regular and can be done using simple hardware.¹⁶ Second, even sparse merging algorithms can be improved in some cases. The simplest sparse algorithm merges every pixel rendered by every processor. Under some circumstances (for example, when broadcast is available), it is possible to merge only a fraction of the pixels rendered at each pixel location.²⁴

Here, we will analyze only two cases: simple sparse merging (SL-sparse) and full-frame merging (SL-full). Figure 8 shows the sort-last rendering pipeline and the additional costs of SL-sparse and SL-full not found in uniprocessor rendering.

Processing and communication costs

Sort-last geometry processing and pixel rendering cost the same as they would on a uniprocessor. After pixel samples are rendered, however, they must be redistributed for compositing. Since SL-sparse sends only the samples generated, it requires $n, a, S \llcorner \text{sample} \gg$ network bandwidth, where «sample» is the size of the sample data structure. SL-sparse also requires communication processing proportional to bandwidth. After redistribution, SL-sparse performs the same visibility calculations that would be performed on a uniprocessor.

SL-full merges a full frame from each of the N processors and therefore requires $NAS \llcorner \text{sample} \gg$ network bandwidth and communication processing. The renderers in SL-full perform the same $dAS \llcorner \text{comp} \gg$ visibility calculations (in the nodes' local z -buffers) as would a uniprocessor. In addition, they must merge N full frames of pixel samples, so they perform $NAS \llcorner \text{comp} \gg$ more visibility calculations than would a uniprocessor.

Comparing the two approaches, we see that the per-frame overhead for SL-sparse depends on the total number of pixels generated, n, a, r , (and, therefore, the size of the scene), but is independent of the number of processors. The overhead for SL-full, on the other hand, depends on the number of processors N and the screen resolution A , but not on the scene contents. (If the frame rate is to remain constant, however, N must increase with the number of primitives, so the cost of SL-full depends indirectly on the scene size).

If the communication network in SL-full is implemented as a pipeline, increasing N increases the available communication bandwidth by the same factor, thereby stenciling the network to fit the algorithm. This gives it an unusual property of linear scalability.²⁵

Oversampling

Many systems perform antialiasing by oversampling, that is, they calculate the color for some number of samples that lie within each pixel and filter these samples down to one color. Sort-last systems that oversample treat samples as pixels, and merge and process them similarly. Thus, the degree of oversampling S linearly affects the cost of sort-last algorithms, as shown in Figure 8. Oversampling rates of up to 16 are common. These systems must provide bandwidth accordingly.

Load balancing

Sort-last renderers can suffer load imbalances from an uneven distribution of rendering work in the same manner as sort-first and sort-middle. Sort-last is less prone to load imbalances from

Estimating primitive overlap

Both sort-first and sort-middle have a common source of inefficiency: Primitives that cross region boundaries must be processed in multiple regions. We express this inefficiency in terms of the overlap factor (O), the number of regions covered by a typical primitive.

We can estimate O using the geometric construction shown in Figure A. If we assume that screen regions have size $W \times H$ and a typical primitive has a bounding box of size $w \times h$, a primitive will contribute to four regions if the center of its bounding box falls into one of the four corner areas, two regions if the center of its bounding box falls into one of the four edge areas, and one region otherwise.

If we assume that primitives have an equal probability of falling anywhere within a region, the probability of landing in a corner, edge, or center region is, respectively,

$$\frac{4(w/2)(h/2)}{WH},$$

$$\frac{2(w/2)(H-h) + 2(h/2)(W-w)}{WH},$$

$$\frac{(W-w)(H-h)}{WH}.$$

Weighting these by the number of regions affected (four, two, and one, respectively), we can sum them to get O , the expected number of regions affected by a primitive:

$$O = \left(\frac{W+w}{W} \right) \left(\frac{H+h}{H} \right)$$

This equation, first derived by John Eyles of the University of North Carolina, is valid even if $w > W$ and $h > H$. The graph in Figure B plots O for various region and bounding-box sizes. We have found these values to correlate well with data obtained from actual renderings.²⁵

Thus we see that if primitives are small compared to region size, primitive overlap contributes only a small amount of the overall rendering cost.

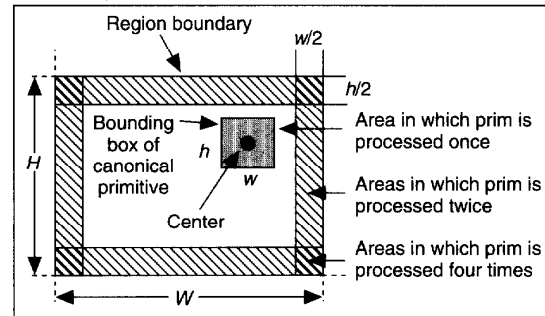
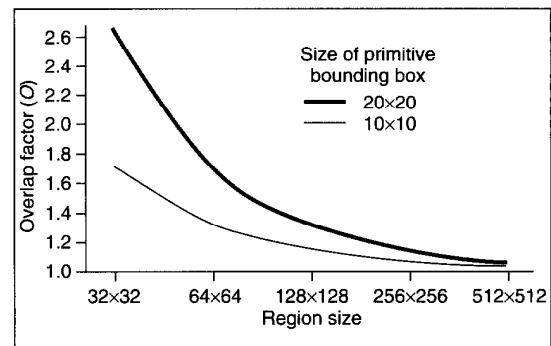


Figure A. Geometric construction for determining primitive overlap.

Figure B. Overlap factor as a function of region size.



primitive clumping, however, since renderers handle the entire screen. In SL-sparse, network traffic and compositing can be unbalanced if more pixels are sent to one compositor than another. Assigning compositors interleaved arrays of pixels addresses this problem by making it likely that a primitive from one renderer will send equal numbers of pixels to every compositor. SL-full does not suffer from this latter type of load imbalance.

Advantages and disadvantages

In summary, sort-last has the following advantages:

- Processing nodes implement the entire rendering pipeline for a portion of the primitives.
- It is less prone to load imbalance.
- SL-full merging can be embedded in a linear network, making it linearly scalable.

However, pixel traffic in sort-last can be extremely high, particularly when oversampling. This is its chief disadvantage.

Comparison of approaches

The analytic models just developed provide a good starting point, but as mentioned earlier, other factors come into play, such as characteristics of the application and hardware on which the algorithm will be implemented. We will see that none of the approaches is a clear winner or loser under all conditions. Rather, each is potentially useful for some set of applications and implementation constraints.

Processing cost

Figures 6 through 8 list the processing overhead for the three rendering approaches. We can simplify these by observing that overhead caused by primitive overlap is likely to be small for most applications (see the sidebar “Estimating primitive overlap”). If we ignore this overlap, the processing and communication overhead for the different approaches are as shown in Figure 9 on the next page.

How important is each of these factors? Pre-transformation overhead (sort-first) and bucketization overhead (sort-first and sort-middle) may or may not be significant relative to the re-

Figure 9. Processing and communication overhead, assuming primitive overlap is negligible.

Pipeline stage	Overhead			
	Sort-first	Sort-middle	SL-sparse	SL-full
Geometry	$n_r \ll \text{pre-geom} \gg$	—	—	—
(Bucketization)	$n_r \ll \text{bucket} \gg$	$T n_r \ll \text{bucket} \gg$	—	—
Visibility	—	—	—	$NAS \ll \text{comp} \gg$
Redistribution	$cn_r \ll \text{prim} \gg$	$T n_r \ll \text{prim} \gg$	$n_r a_s \ll \text{sample} \gg$	$NAS \ll \text{sample} \gg$

maining geometry and rasterization tasks. This overhead can be substantial when compared with the rendering costs of simple rendering algorithms, but it can be insignificant when compared with the costs of high-quality rendering algorithms.

SL-sparse has little processing overhead beyond the cost of redistributing pixels. SL-full requires much more processing for compositing, suggesting hardware support.

All the approaches require extra processing to handle the redistribution of primitives or pixels, but the amount depends on communication bandwidth requirements, which we consider next.

Communication cost

Figure 9 shows the communication costs for the different approaches (in boxes). We consider these in turn.

First, note that the communication requirements for sort-first depend on c . When c is small, the communication requirements for sort-first can be very small. However, if sort-first is used without coherence or if c is close to 1, it transfers the entire data set of raw primitives every frame. This is similar to sort-middle, but sort-middle transfers display primitives, and there are T times as many display primitives as raw primitives. Hence, if the tessellation ratio T is high, sort-first requires less bandwidth. On the other hand, if $c = 1$ and there is little or no tessellation, sort-middle is preferred over sort-first.

We can further understand this trend by considering average primitive size. Systems that tessellate often generate display primitives about a pixel or so in size. The smaller the display primitive, the more accurate the rendering of curved surfaces. Under these circumstances, $T \approx a_r$, and there are about a_r times as many display primitives as raw primitives.

The trade-offs between sort-first/sort-middle and SL-sparse depend on the relative sizes of the data structures that implement primitives and pixels. In particular, sort-first (no coherence) is favored if $\ll \text{prim} \gg < a_r S \ll \text{sample} \gg$, and sort-middle is favored if $\ll \text{prim} \gg < a_r S \ll \text{sample} \gg$. So if primitives tend to be simple but cover a large area of the screen, or if oversampling is employed, sort-first or sort-middle are favored. If primitives are complex but cover a small screen area, SL-sparse is favored.

Comparing SL-sparse and SL-full, we see that SL-sparse is favored unless $NA < n_r a_r$, that is, unless the depth complexity of the entire image is greater than the number of processors. Sample data sets that we have analyzed have had depth complexity ranging from 0.53 to 12.9 with a median of 3.2.^{23,25} This argues that SL-sparse requires less communication bandwidth than SL-full under most conditions.

Hardware versus software

So far we have considered processing and communication costs to be abstract quantities, such as floating-point operations or bits. This ignores the critical fact that their real costs (in time, dollars, watts, and so forth) depend on how easily the architec-

ture or machine can perform these operations. For example, communicating a data bit over an Ethernet network can be orders of magnitude more expensive than sending the same bit over a dedicated hardware communication channel. Similarly, if communication in a sort-last system is accelerated in hardware, as it is in several current commercial machines,¹⁷⁻¹⁹ it can be less expensive according to some measures than communication in a sort-middle system that sends much less data. We can view hardware acceleration, then, as a way of reducing the real costs of critical or "bottlenecked" operations.

Other factors

Some of the approaches place constraints on the applications and rendering algorithms that can employ them. For example, only applications that use retained mode can use sort-first with frame-to-frame coherence, since the graphics database must reside on the renderers. Sort-last constrains the choice of rendering algorithms because visibility is determined strictly by compositing. In all three approaches, redistributed primitives or samples may arrive at destination renderers or compositors "out of order." Some rendering systems allow rendering order to determine visibility as well as depth value for effects like stencils and transparency. In such systems, all three approaches require additional computation and possibly bandwidth to enforce ordering of the redistributed primitives or samples.

In sort-first and sort-last, each processor implements an entire rendering pipeline. Renderers in these systems may be able to take advantage of techniques and/or designs used in "single-stream" renderers. Sort-middle breaks the rendering pipeline between geometry processing and rasterization, so a sort-middle design must make screen-space data accessible to the communication network.

Finally, the approaches differ with respect to load balancing. Sort-first and sort-middle both are susceptible to primitives clustering in regions. SL-sparse can suffer from contention at compositors. All the approaches are sensitive to load imbalances arising from the initial object assignment.

Toward the future

Future systems will have to perform at much higher levels than systems of today. Performance needs to scale in at least two ways: increased resolution (due to increased screen sizes and greater demand for antialiasing) and increased primitive performance.

Increasing screen resolution while leaving the number of primitives constant increases the communication costs of both versions of sort-last relative to sort-first and sort-middle. For SL-sparse, it increases the number of pixels covered by each primitive; for SL-full, it increases the effective screen area AS .

Increasing the number of primitives while leaving the screen resolution unchanged directly increases the communication

costs of all the approaches except SL-full—and indirectly increases the cost of SL-full, since N would have to increase. However, SL-full differs from the other approaches in that its communication is fixed and local, allowing it to scale with the number of processors.

The overlap factor O , though small today, is a function of N . As machine size increases to a thousand processors or more, the size of per-processor regions can get quite small. When this happens, O increases, eventually driving the overhead of sort-first and sort-middle to unacceptable levels.

Finally, we have presented the simplest view—that redistribution occurs at only one point in the rendering pipeline. Hybrid architectures, which exploit trade-offs between the approaches, are possible. For example, a sort-first or sort-middle algorithm might choose to render rather than redistribute small primitives and to redistribute pixels instead. Other hybrids are possible. We conjecture that hybrids will be a fertile area for future work.

Conclusion

The intrinsic sorting problem in rendering is the basis for the simple way of classifying parallel rendering algorithms described and analyzed here. Although it might be reassuring to state categorically that one approach is always preferred, it would be inaccurate. Trade-offs depend on implementation and application. We have tried to provide a framework and appropriate tools for the reader to select the right approach given his or her own application and machine requirements. We also hope our work will motivate further investigations into the trade-offs between alternative parallel rendering strategies. □

Acknowledgments

We wish to thank John Poulton, one of the original inventors of the sort-first/middle/last classification scheme, and the entire Pixel-Planes group for valuable discussions on the ideas in this article. We thank Carl Mueller for providing values of c and help with the title-page illustration. We also thank Pat Hanrahan and Craig Kolb for valuable comments on an earlier version of this article.

We are grateful to our sponsors: ARPA (grant no. DABT63-92-C-0053 and ISTO order no. 7510), NSF (grant MIP-9000894), and Apple Computer.

References

1. F. Crow, "Parallelism in Rendering Algorithms," *Proc. Graphics Interface 88*, Morgan Kaufman, San Mateo, Calif., 1988, pp. 87-96.
2. N. Gharachorloo et al., "A Characterization of Ten Rasterization Techniques," *Computer Graphics* (Proc. Siggraph), Vol. 23, No. 3, July 1989, pp. 355-368.
3. S. Molnar and H. Fuchs, "Advanced Raster Graphics Architecture," in *Computer Graphics: Principles and Practice*, 2nd ed., J. D. Foley et al., eds., Addison-Wesley, Reading, Mass., 1990.
4. S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Wellesley, Mass., 1992.
5. K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics* (Proc. Siggraph), Vol. 22, No. 4, Aug. 1988, pp. 239-246.
6. T. W. Crockett and T. Orloff, "A Parallel Rendering Algorithm for MIMD Architectures," *Proc. Parallel Rendering Symposium*, ACM Press, New York, 1993, pp. 35-42.
7. S. Whitman, "A Task Adaptive Parallel Graphics Renderer," *IEEE CG&A*, Vol. 14, No. 4, July 1994, pp. 41-48.
8. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, Vol. 6, No. 1, Mar. 1974, pp. 1-55.
9. H. Fuchs et al., "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics* (Proc. Siggraph), Vol. 23, No. 3, July 1989, pp. 79-88.
10. K. Akeley, "RealityEngine Graphics," *Computer Graphics* (Proc. Siggraph), Aug. 1993, pp. 109-116.
11. M. Deering and S.R. Nelson, "Leo: A System for Cost Effective 3D Shaded Graphics," *Computer Graphics* (Proc. Siggraph), Aug. 1993, pp. 101-108.
12. D. Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers," *IEEE CG&A*, Vol. 14, No. 4, July 1994, pp. 33-40.
13. R. Bunker and R. Economy, "Evolution of GE CIG Systems," tech. report, Simulation and Control System Dept., General Electric, Daytona Beach, Fla., 1989.
14. G.C. Roman and T. Kimura, "A VLSI Architecture for Real-Time Color Display of Three-Dimensional Objects," *Proc. IEEE Micro-Delcon*, IEEE Press, Piscataway, N.J., 1979, pp. 113-118.
15. C. Shaw, M. Green, and J. Schaeffer, "A VLSI Architecture for Image Composition," in *Advances in Computer Graphics Hardware III*, Springer-Verlag, New York, 1988, pp. 183-199.
16. S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics* (Proc. Siggraph), Vol. 26, No. 2, July 1992, pp. 231-240.
17. Evans and Sutherland Computer Corporation, *Freedom Series Technical Report*, Salt Lake City, Utah, Oct. 1992.
18. Fujitsu Limited, "AG Series Graphics Technical Overview," Fujitsu Open Systems Solutions, San Jose, Calif., 1993.
19. Kubota Pacific Computer, *Denali Technical Overview*, version 1.0, Santa Clara, Calif., Mar. 1993.
20. National Computer Graphics Association, *GPC Quarterly Report*, Vol. 2, No. 4, Fairfax, Va., 1992.
21. D. Ellsworth, H. Good, and B. Tebbs, "Distributing Display Lists on a Multicomputer," *Computer Graphics* (Proc. Symp. Interactive 3D Graphics), Vol. 24, No. 2, Mar. 1990, pp. 147-155.
22. F. Parke, "Simulation and Expected Performance Analysis of Multiple Processor Z-buffer Systems," *Computer Graphics* (Proc. Siggraph), Vol. 14, No. 3, July 1980, pp. 48-56.
23. M. Cox and P. Hanrahan, "Depth Complexity in Object-Parallel Graphics Architectures," *Proc. Seventh Workshop on Graphics Hardware*, Eurographics Technical Report Series, 1992, pp. 204-222.
24. M. Cox and P. Hanrahan, "Pixel Merging for Object-Parallel Rendering: A Distributed Snooping Algorithm," *Proc. Parallel Rendering Symp.*, ACM Press, New York, 1993, pp. 49-56.
25. S. Molnar, *Image-Composition Architectures for Real-Time Image Generation*, doctoral dissertation, TR 91-046, University of North Carolina at Chapel Hill, Oct. 1991.



Steven Molnar is a research assistant professor of computer science at the University of North Carolina at Chapel Hill. His research interests include architectures and algorithms for real-time, realistic image generation and VLSI-based system design. He received a BS in electrical engineering from the California Institute of Technology and an MS and PhD in computer science from UNC-Chapel Hill.



Michael Cox is completing his dissertation in computer science at Princeton University, where he works on algorithms for parallel rendering. He holds a BA in biology from the University of California at Santa Cruz and an MA in computer science from Princeton University.



David A. Ellsworth is a doctoral candidate in the Computer Science Department at the University of North Carolina at Chapel Hill. His research interests include parallel rendering, performance modeling, and parallel algorithms. Ellsworth received a BS in electrical engineering and computer science from the University of California at Berkeley and an MS in computer science from University of North Carolina at Chapel Hill.



Henry Fuchs is Federico Gil professor of computer science and adjunct professor of radiation oncology at the University of North Carolina at Chapel Hill. His research interests include high-performance graphics hardware (he founded the Pixel-Planes project at UNC), 3D medical imaging, and head-mounted displays and virtual environments. He received a BA in information and computer science from the University of California at Santa Cruz and a PhD in computer science from the University of Utah.

Readers may contact Molnar, Ellsworth, and Fuchs at the Dept. of Computer Science, Sitterson Hall, UNC-Chapel Hill, Chapel Hill, NC 27599-3175. Contact Cox at the Dept. of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08540.



VISION CHIPS: Implementing Vision Algorithms with Analog VLSI Circuits

edited by Christof Koch and Hua Li

Examines the significant progress made in the last few years in better understanding the operations underlying visual processing. This has resulted in a large number of well-studied algorithms for locating edges, computing disparities, estimating motion fields, and finding discontinuities in depth, color, motion, and texture. The book covers the acquisition of images, spatio-temporal filtering to remove noise and to emphasize features, and circuits and theoretical concepts of estimation. In addition, it introduces three circuits that output a single variable associated with a contour or an entire object in an image, and discuss the employment of charge-coupled devices (CCDs) for different imaging applications.

Sections: Background and Overview, Filtering and Edge Detection, Motion and Stereo Processing, Towards Object-Oriented Attributes, CCD Techniques for Image Processing, Related Issues.

c.510 pages. October 1994. Cloth.
ISBN 0-8186-6492-4.
Catalog # 6492-01
\$55.00 Members \$42.00

CVPR '94 IEEE Computer Society Conference on Computer Vision and Pattern Recognition

June 21-23, 1994 — Seattle, WA

Sections: Surfaces, Object Recognition, Pattern Recognition/Systems, Curves and Contours, 3D Vision, Motion, Filtering, Systems and Applications, Active Vision, Physics-Based Vision, Document Processing, Projective Geometry and Invariance, 3D Models, Feature Extraction.

1,032 pages. June 1994. Paper.
ISBN 0-8186-5825-8.
Catalog # 5825-02
\$150.00 Members \$75.00

IEEE COMPUTER SOCIETY PRESS

10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1264

- ▼ Call toll-free: 1-800-CS-BOOKS ▼
- ▼ Fax: (714) 821-4641 ▼
- ▼ E-Mail: cs.books@computer.org ▼