A Sorting Classification of Parallel Rendering

Steven Molnar*, Michael Cox†,
David Ellsworth*, Henry Fuchs*

*University of North Carolina at Chapel Hill and †Princeton University

Front-page photo: Simulation of communication traffic between sort-first processors rendering NCGA "head" Picture-Level benchmark [1]. Arrow color indicates the number of primitives transferred between processors between these two successive frames. Range is 0 (black) to 800 (white) using a heated-object spectrum.

Abstract

We describe three broad classes of parallel rendering methods, based on where the sort from object-space to screen space occurs. These classes encompass most feed-forward parallel software and hardware rendering architectures that have been described to date. After introducing the classes, we perform a coarse analysis of the aggregate processing and communication costs of each and identify constraints they impose on the rendering application. The aim is to provide a conceptual model of the tradeoffs between the approaches as an aid to designers and implementers of high-performance, parallel rendering systems.

Introduction

Graphics rendering is notoriously compute intensive, particularly when realistic images and fast updates are required. Demanding applications, such as scientific visualization, CAD, vehicle simulation, and virtual reality can require hundreds of MFLOPS of floating-point 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 to building high-performance graphics systems, whether these be special-purpose hardware systems, or software systems for general-purpose multicomputers.

Parallelism of various types may be employed at many levels: for example, *functional parallelism* (pipelining) can speed critical calculations and *data parallelism* can be used to compute multiple results 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 [2, 3, 4]. These taxonomies are useful for classifying and understanding systems, but do not lend themselves easily to comparison or analysis. Some rendering systems have been analyzed in isolation [5, 6, 7]. However, these analyses tend to focus on unique attributes of each system and make comparison between systems difficult.

In this paper we describe a classification scheme, which we hope will provide 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 to be fundamental whenever both geometry processing and rasterization are performed in parallel. This classification scheme allows computational and communication costs to be analyzed and encompasses the bulk of current and proposed highly parallel renderers — both hardware and software.

The paper is organized as follows: First we review 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.

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, etc.), and rasterization (scan-conversion, shading, and visibility determination). In this paper we target rendering rates that are sufficiently high that both geometry processing and rasterization must be performed in parallel. We say such systems are *fully parallel*.

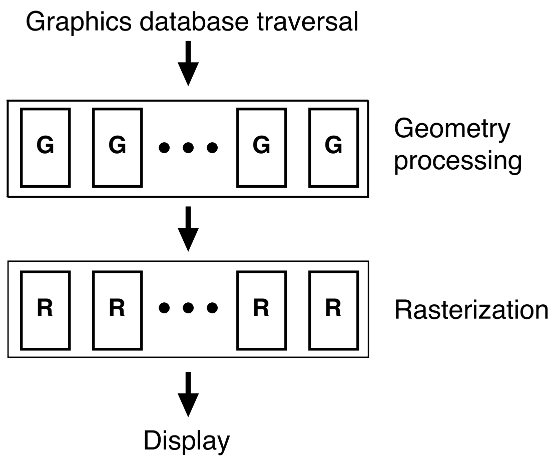


Figure 1. Graphics pipeline in a fully parallel rendering system. Processors G perform geometry processing. Processors R perform rasterization.

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 rendering can be viewed 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 in later sections.

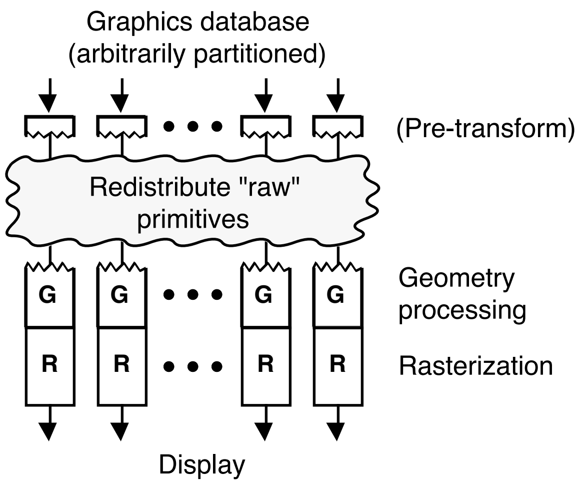


Figure 2. Sort-first. Redistributes raw primitives during geometry processing.

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). This generally is done by dividing the screen into disjoint regions and making processors (called *renderers*) responsible for all rendering calculations that affect their respective screen regions.

Initially, primitives are assigned to renderers in some arbitrary 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 for some primitives, a portion of geometry processing is done on the wrong renderer. The results of these calculations must either be sent or they must be recomputed on the new renderer(s).

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

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 geometry processing and rasterization are performed on separate processors in many systems, 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 may be separate sets of processors, or they may time-share the same physical processors.

During each frame, geometry processors transform, light, etc. their portion of the primitives and classify them with respect to screen region boundaries. They then transmit all of these screen-space primitives to the appropriate rasterizer or rasterizers, as shown in Figure 3.

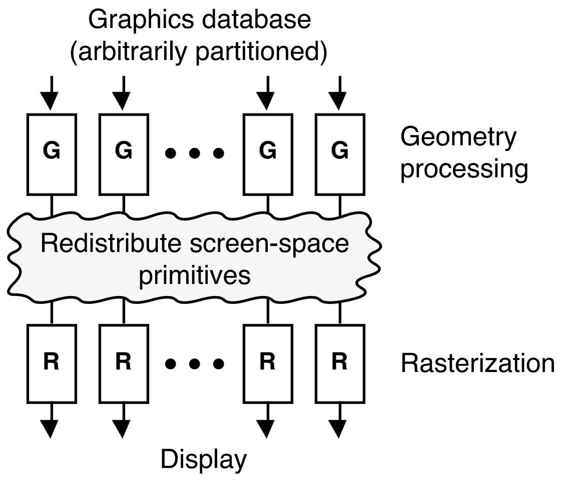

Graphics database
(arbitrarily partitioned)
G
Geometry processing
Redistribute screen-space primitives
R
Rasterization
Display

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

Sort-middle is general and straightforward, and has been the most common approach to date for both hardware [9, 10, 11] and software [6, 7, 12] parallel rendering systems. We will examine the advantages and disadvantages of sort-middle 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. Processors in sort-last (called *renderers*) each are assigned arbitrary subsets of the primitives. Each 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 of the pixel data generated on all of the renderers. For interactive or real-time applications rendering high-quality images, this can result in very high data rates.

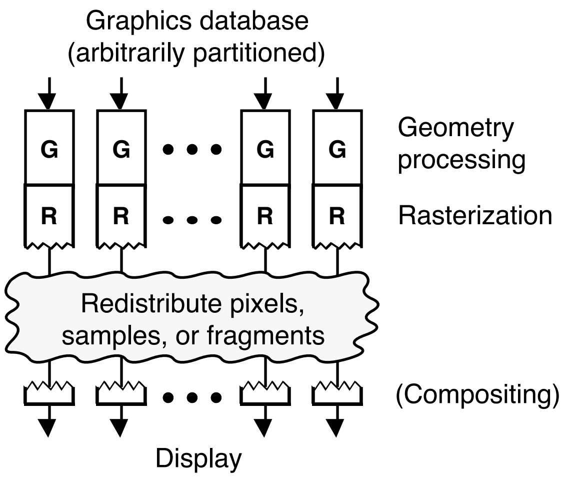


Figure 4. Sort-last. Redistributes pixels, samples, or pixel fragments during rasterization.

Sort-last can be done in at least two ways. One approach, which we call *SL-sparse*, minimizes communication by only distributing 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* in which a processor was assigned 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 *SL-sparse* [17, 18, 19]. We will examine sort-last in more detail shortly.

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 will first consider the processing required to render on a uniprocessor, and will then evaluate the additional processing and communication requirements of the parallel methods. We will focus on the inherent overhead in the methods, and will not specifically model factors such as load balancing, buffering, transport delay, etc., which affect performance significantly, but depend on the detailed implementation of a system and on the graphics scene itself. We will, however, discuss these factors (particularly load balancing) where appropriate.

Uniprocessor pipeline

For the analysis that follows, we refine the rendering pipeline as shown in Figure 5. First, some rendering systems *tessellate* primitives in order to generate higher quality images (RenderMan [20] is one widely used example). Tessellation is the process of decomposing larger primitives into smaller ones, typically into polygons or polygonal meshes. Not all rendering packages tessellate. We include it in the pipeline because it can greatly expand the number of primitives that need to be displayed. 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.

Pipeline stage	Processing cost
Geometry:	
pre-tessellation	n_r «*geom-pre-tess*» +
post-tessellation	n_d «*geom-post-tess*»
Pixel Rendering	n_d «*rend-setup*» + $n_d a_d S$ «*rend*»
Visibility	$n_d a_d S$ «*comp*»

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

We assume that we are rendering a dataset 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 $n_d = Tn_r$ of these, 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 (*i.e.* 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 «*geom-pre-tess*» ("«...»"

Terms used in the analysis

Term	Definition
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 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.

denotes "cost", most naturally in units of time). The n_d display primitives that result are then processed by the geometry pipeline stages following tessellation, at a cost of n_d «*geom-post-tess*». Rasterization follows. Pixel rendering is the first stage and consists of two parts: set-up and per-sample rendering operations, whose costs are, respectively, n_d «*rend-setup*» and $n_d a_d S$ «*rend*». Finally, the cost of compositing is $n_d a_d S$ «*comp*», where «*comp*» is the cost of compositing one sample (generally a z comparison and conditional write). In the uniprocessor model, there is no data redistribution or "sort".

Sort-first analysis

We begin our analysis with sort-first. Figure 6 shows its rendering pipeline and what it costs beyond uniprocessor rendering. We assume in this analysis 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 redistribute later in the pipeline and send the transformed data. Although we will not focus on this alternative, its analysis is similar to the case we will 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 processor, and a primitive may fall in several buckets when 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 «*pre-xform*» and the cost to put the primitive in each of its buckets is «$bucket_r$», then the overhead for these stages is n_r «*pre-xform*» and $n_r O_r$ «$bucket_r$».

Pipeline stage	SF overhead cost
Pre-transform	n_r «*pre-xform*»
Bucketization	$n_r O_r$ «$bucket_r$»
Redistribution	$\boxed{c n_r O_r \text{ «}prim_r\text{»}}$
Geometry:	
pre-tessellation	$n_r(O_r\text{-}1) f_r$ «*geom-pre-tess*»+
post-tessellation	$n_d(O_d\text{-}1) f_d$ «*geom-post-tess*»
Pixel Rendering	$n_d(O_d\text{-}1)$ «*rend-setup*»
Visibility	—

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

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 will need to be sent. However, if we render multiple frames in an animation or real-time application and the scene does not change much between frames, only a small fraction of the primitives may need to be sent.[1] 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 $c n_r O_r$ «$prim_r$», where «$prim_r$» is the size of the data structure to represent a raw primitive. We will not explicitly tally the processing cost of communication, but it should be noted that it is proportional to this term.

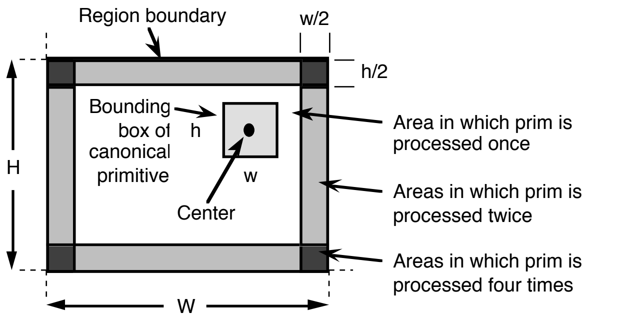
After redistribution, sort-first algorithms may accrue other overhead that a uniprocessor pipeline would not. If a raw primitive falls in exactly one bucket, then it undergoes geometry processing exactly once, and the costs are the same as they would be on a uniprocessor. If the primitive overlaps more 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

[1]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.

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 following geometric construction:


Region boundary
w/2
h/2
Bounding box of canonical primitive
h
w
Center
H
W
Area in which prim is processed once
Areas in which prim is processed twice
Areas in which prim is processed four times

If we assume screen regions have size W x H and a typical primitive has a bounding box of size w□x□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.

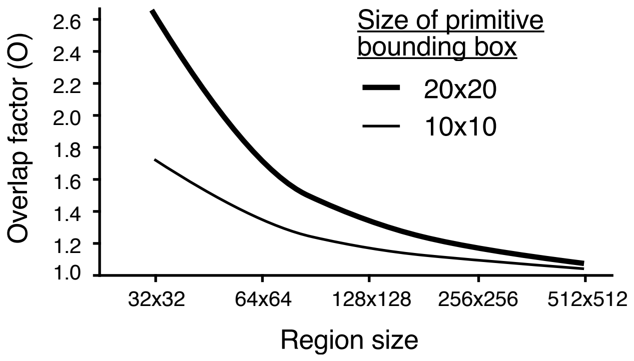
If we assume that primitives have an equal probability of falling anywhere within a region, the probability of landing in a corner, edge, or center region is, respectively:

$$\frac{4(w/2)(h/2)}{WH}, \frac{2(w/2)(H-h)+2(h/2)(W-w)}{WH}, \frac{(W-w)(H-h)}{WH}.$$

Weighting each of these by the number of regions affected (4, 2, and 1, 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 UNC, is valid even if $w > W$ and $h > H$. The following graph plots O for various region and bounding-box sizes. We have found that these values correlate well with data obtained from actual renderings [21].



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

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 n_r raw primitives is $n_r(O_r\text{-}1)f_r$ «*geom-pre-tess*», and for the Tn_r display primitives is $n_d(O_d\text{-}1)f_d$ «*geom-post-tess*».

Finally, each display primitive that overlaps several screen regions requires pixel rendering setup for each region. This adds processing overhead of $n_d(O_d\text{-}1)$ «*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 keep track of 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 Benchmark [1], and an architectural walkthrough of a building interior seen through a head-mounted display. In both cases, we assumed a display size of 1280x1024 pixels and a region size of 64x64 pixels. In the Picture-Level Benchmark, the head contains 60,000 triangles and rotates 4.5° 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_dS samples. This means that for each raw primitive that sort-first redistributes, sort-middle must redistribute T display primitives, and sort-last must redistribute Ta_dS 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 (and the other two approaches) are susceptible to several types of load imbalance. 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.

A second type of load imbalance arises from the distribution of primitives over the screen. In sort-first, it is very likely that some regions of the screen will 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 may not achieve an optimal load distribution for any given frame, or it may be done dynamically, which increases algorithm complexity, but may achieve better results. When using frame-to-frame coherence, dynamic load balancing must be constrained so that a region can be 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, the advantages of sort-first are:

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

It has the following disadvantages, however:

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

Sort-middle analysis

Sort-middle algorithms are in some sense the most "natural". In contrast to sort-first, primitives are redistributed 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_dO_d «*bucket*$_d$». The bandwidth required to send these buckets is n_dO_d «*prim*$_d$», 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.

Pipeline stage	***SM* overhead**
Geometry	—
Bucketization	n_dO_d «*bucket*$_d$»
Redistribution	$\boxed{n_dO_d \text{ «}\mathit{prim}_d\text{»}}$
Pixel Rendering	$n_d(O_d$-1) «*rend-setup*»
Visibility	—

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

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 will transfer a large number of display primitives. If there is no tessellation or T is small, sort-middle will transfer roughly the same number of display primitives as there are raw primitives. Sort-first transfers raw primitives, so for sort-middle to compare favorably, T must be small.

Load balancing

Sort-middle can suffer load imbalances from object assignment and the clumping of primitives into regions in the same manner as sort-first. Load balancing the assignment of objects in hierarchical display structures has been explored [22]. The other problem, primitive clumping, has been the focus of much of the research in hardware [23] and software [4, 7, 12] sort-middle renderers. The main techniques are to make regions smaller (and more numerous) and to assign regions dynamically to processors. This tends to increase the overlap factor O_d, and hence should be applied with care.

Pipeline stage	***SL-sparse***	***SL-full***
Geometry	—	—
Pixel Rendering	—	—
Redistribution	$n_r a_r S$ «*sample*»	*NAS* «*sample*»
Visibility	—	*NAS* «*comp*»

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

Advantages and disadvantages

In summary, sort-middle has the following advantages:

- General and straightforward; redistribution occurs at a natural place in the pipeline.

It has the following disadvantages, however:

- High communication costs if tessellation ratio is high.
- Susceptible 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 [24]. 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 may be improved in some cases. The simplest sparse algorithm merges every pixel rendered by every processor. Under some circumstances (*e.g.* when broadcast is available [25]), it is possible to merge only a fraction of the pixels rendered at each pixel location. 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 $n_r a_r S$ «*sample*» 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* «*sample*» network bandwidth and communication processing. The renderers in *SL-full* perform the same *dAS* «*comp*» 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* «*comp*» 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 $n_r a_r$ (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 contents of the scene. (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 size of the scene).

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 [21].

Oversampling

Many systems perform anti-aliasing by oversampling: calculating the color for some number of samples that lie within each pixel and filtering these samples down to one color. In sort-last systems that oversample, samples are treated as pixels and merged and processed 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 not uncommon. For these systems, bandwidth must be provided 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 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. This can be addressed by assigning compositors interleaved arrays of pixels so a primitive from one renderer is likely to 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:

- Renderers implement the full rendering pipeline and are independent until pixel merging.

Pipeline Stage	*Sort-first*	*Sort-middle*	*SL-sparse*	*SL-full*
Geometry	n_r «*pre-geom*»	—	—	—
(Bucketization)	n_r «*bucket*$_r$»	$T\ n_r$ «*bucket*$_d$»	—	—
Visibility	—	—	—	*NAS* «*comp*»
Redistribution	$\boxed{c n_r \text{ «}prim_r\text{»}}$	$\boxed{T\ n_r \text{ «}prim_d\text{»}}$	$\boxed{n_r a_r S \text{ «}sample\text{»}}$	$\boxed{NAS \text{ «}sample\text{»}}$

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

- Less prone to load imbalance.
- *SL-full* merging can be embedded in a linear network, making it linearly scalable.

However, it has the following disadvantage:

- Pixel traffic may be extremely high, particularly when oversampling.

Comparison of approaches

In this final section we compare the three approaches and provide some guidance in determining when each is preferable. The analytic models just developed provide a good starting point, but as mentioned above, other factors, such as characteristics of the application and hardware on which the algorithm will be implemented, also come into play. 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 due to primitive overlap is likely to be small for most applications (see sidebar on p. 6). If we ignore primitive overlap, the processing and communication overhead for the different approaches are as shown in Figure 9.

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 remaining geometry and rasterization tasks. This will depend on the complexity of the implementation: this overhead may be substantial when compared with the rendering costs of simple rendering algorithms, but may 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 of the approaches require extra processing to handle the redistribution of primitives or pixels, but this depends on communication bandwidth requirements, which we will consider next.

Communication cost

Figure 9 shows the communication costs for the different approaches (in boxes). We will 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, the entire dataset of *raw* primitives is transferred every frame. This is similar to sort-middle, but sort-middle transfers *display* primitives; there are T times as many of them. 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 that are 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 tradeoffs between sort-first/sort-middle and *SL-sparse* depend on the relative sizes of primitives and primitives' and pixels' data structures. In particular, sort-first (no coherence) is favored if «*prim*$_r$» $< a_r S$ «*sample*» and sort-middle is favored if «*prim*$_d$» $< a_d S$ «*sample*». 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 $NA < n_r$, or in other words, unless the depth complexity of the entire image is greater than the number of processors. On sample datasets analyzed by the authors, depth complexity ranged from 0.53 to 12.9 with a median of 3.2 [21, 24]. From these results we conclude that *SL-sparse* requires less communication bandwidth than *SL-full* under most conditions.

Hardware vs. software

So far we have ignored a point that is critical to this discussion: we have considered processing and communication costs to be abstract quantities, such as floating-point operations or bits. This ignores the fact that their *real* costs (in time, dollars, watts, etc.) depend on the ease in which the architecture or machine can perform these operations. For example, communicating a bit of data 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 sort-last is accelerated in hardware, as it is in several current commercial machines [17, 18, 19], it may 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 application and set of rendering algorithms that may be employed. 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. Some rendering systems allow rendering order to determine visibility as well as depth value (for effects like stencils and transparency). This may be difficult to implement efficiently on a sort-last system (it can present problems on any of these fully-parallel renderers).

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 of the approaches are sensitive to load imbalances arising from the initial object assignment.

Toward the future

Systems of the future will have to perform at much higher levels than systems of today. Performance will need 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 of 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 the per-processor region 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 tradeoffs between the approaches, are possible. For example, a sort-first or sort-middle algorithm might choose to render rather than redistribut small primitives, and 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 leads to a simple way of classifying parallel rendering algorithms. Based on this observation, we have proposed three classes: sort-first, sort-middle, and sort-last. We have analyzed some of the fundamental cost tradeoffs and described some of the qualitative tradeoffs between these approaches to give the reader a framework for choosing between them.

Although it would be reassuring to state categorically that one approach is always preferred, it would be inaccurate: tradeoffs must be considered which are dependent on implementation and application. We have attempted in this paper 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 this work will motivate further investigations into the tradeoffs between alternative parallel rendering strategies.

Acknowledgements

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 paper. We thank Carl Mueller for providing values of *c* and help with the front illustration. We also wish to thank Pat Hanrahan and Craig Kolb for valuable comments on an earlier version of this paper.

We are grateful to our sponsors: ARPA (grant DABT63-92-C-0053 and ISTO order 7510), NSF (grant MIP-9000894), and Apple Computer.

References

1. National Computer Graphics Association, *GPC Quarterly Report*, Vol. 2, No. 4, 4th, Fairfax, VA, 1992.
2. N. Gharachorloo, S. Gupta, R.F. Sproull, I.E. Sutherland, "A Characterization of Ten Rasterization Techniques," *Computer Graphics* (Siggraph '89 Proceedings), Vol. 23, No. 3, July 1989, pp. 355–368.
3. S. Molnar, H. Fuchs, "Advanced Raster Graphics Architecture," Chapter 18 in J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.
4. S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Ltd., Wellesley, MA, 1992.
5. K. Akeley, T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics* (Siggraph '88 Proceedings), Vol. 22, No. 4, Aug. 1988, pp. 239–246.
6. T. W. Crockett, T. Orloff, "A Parallel Rendering Algorithm for MIMD Architectures," *Proceedings of the 1993 Parallel Rendering Symposium*, ACM, New York, 1993, pp. 35–42.
7. S. Whitman, "A Task Adaptive Parallel Graphics Renderer," to appear in *IEEE CG&A*, July 1994.
8. I.E. Sutherland, R.F. Sproull, R.A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, Vol. 6, No. 1, March 1974, pp. 1–55.
9. H. Fuchs, J. Poulton, *et. al*, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics* (Siggraph '89 Proceedings), Vol. 23, No. 3, July 1989, pp. 79–88.
10. K. Akeley, "RealityEngine Graphics", *Computer Graphics* (Siggraph '93 Proceedings), Aug. 93, pp. 109–116.
11. M. Deering, S.R. Nelson, "Leo: A System for Cost Effective 3D Shaded Graphics," *Computer Graphics* (Siggraph '93 Proceedings), Aug. 93., pp. 101–108.
12. D. Ellsworth, "A Multicomputer Polygon Rendering Algorithm for Interactive Applications," to appear in *IEEE CG&A*, July 1994.
13. R. Bunker, R. Economy, "Evolution of GE CIG Systems," *SCSD Document*, General Electric Company, Daytona Beach, FL 32015, 1989.
14. G. C. Roman, T. Kimura, "A VLSI Architecture for Real-Time Color Display of Three-Dimensional Objects," *Proceedings of IEEE Micro-Delcon*, March 20, 1979, pp. 113–118.
15. 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, pp. 183–199.
16. S. Molnar, J. Eyles, J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics* (Siggraph '92 Proceedings), Vol. 26, No. 2, July 1992, pp. 231–240.
17. Evans and Sutherland Computer Corporation, *Freedom Series Technical Report*, October 1992.
18. Fujitsu Limited, "AG Series Graphics Technical Overview", Fujitsu Open Systems Solutions, Inc., San Jose, CA 95134-2022, 1993.
19. Kubota Pacific Computer, *Denali Technical Overview*, version 1.0, March 1993.
20. S. Upstill, *The RenderMan Companion*, Addison-Wesley, Reading MA, 1989.
21. S. Molnar, *Image-Composition Architectures for Real-Time Image Generation*, Ph.D. Thesis, TR 91-046, University of North Carolina at Chapel Hill, October 1991.
22. D. Ellsworth, H. Good, B. Tebbs, "Distributing Display Lists on a Multicomputer," *Computer Graphics* (Proceedings 1990 Symposium on Interactive 3D Graphics), Vol. 24, No. 2, March 1990, pp. 147-155.
23. F. Parke, "Simulation and Expected Performance Analysis of Multiple Processor Z-buffer Systems," *Computer Graphics* (Siggraph '80 Proceedings), Vol. 14, No. 3, July 1980, pp. 48–56.
24. M. Cox, P. Hanrahan, "Depth Complexity in Object-Parallel Graphics Architectures," *Proceedings of the Seventh Workshop on Graphics Hardware,* Eurographics Technical Report Series, ISSN 1017-4656, 1992, pp. 204–222.
25. M. Cox, P. Hanrahan, "Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm," *Proceedings of the 1993 Parallel Rendering Symposium* , ACM, New York, 1993, pp. 49–56.

Author Biographies

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 Caltech and an MS and PhD in computer science from UNC-Chapel Hill.

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

David Ellsworth has recently joined Division, Inc. as a software engineer. He received a BS in electrical engineering and computer science from the University of California at Berkeley and a MS in computer science from UNC-Chapel Hill. He expects to have completed a PhD in computer science from UNC-Chapel Hill by the time of this publication.

Henry Fuchs is Federico Gil professor of computer Science and adjunct professor of radiation oncology at the University of North Carolina of 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.

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.