CHAPTER 1
INTRODUCTION

1.1 REALISTIC IMAGES IN REAL TIME

A central goal of computer graphics is to produce images by computer that are so realistic that they cannot be distinguished from reality—so-called *photorealism.* Real-time computer graphics has an analogous goal: to produce a sequence of moving images so rapidly and with enough fidelity that they are indistinguishable from a moving scene in the real world. We call this goal *cinerealism*.

High quality rendering methods, such as ray tracing and radiosity, have made major strides toward photorealism, though much work still remains. Cinerealism is a more elusive goal, since it adds the constraint of time to the image-generation process. Generally, algorithms that produce the most realistic images require the most time to execute, thus the two goals of real-time computer graphics oppose each other.

During the 25-30 years that real-time graphics has been developing, great progress has been made toward cinerealism on many fronts:

- Display technologies have improved. Raster-graphics displays have largely replaced line-drawing displays.
- Lighting and shading models have improved. Gouraud and Phong shading have largely replaced flat shading. Texturing is now becoming available.
- Visibility algorithms have improved. General methods, such as the *z*-buffer algorithm, have replaced restrictive methods, such as fixed-priority or depth-sort algorithms.
- Image fidelity has improved. Early systems used monochrome displays and were plagued by aliasing (stairsteps and "jaggies"). Current systems display continuous-tone colors and avoid most aliasing artifacts.
- More complex datasets can now be displayed. The earliest systems displayed only 10 to 100 vectors in real time (30 frames per second or higher). Early raster systems displayed only a few hundred polygons in real time. Current systems display tens of thousands of polygons in real time.

Progress in this field has been propelled both by technology and by advances in algorithms. Real-time graphics has profited from the same advances in semiconductor technology that have dramatically reduced the size and cost, and increased the performance of general-purpose computers. Algorithms have been developed to compute more realistic images. Many of these, such as Gouraud and Phong shading, antialiasing, and texturing are now implemented in hardware.

Current workstation and experimental graphics systems display almost 100,000 Gouraud- or Phong-shaded unantialiased polygons in real-time and significantly fewer than this with antialiasing and/or limited texturing [SGI90, FUCH89]. Flight-simulator graphics systems support texturing comprehensively, but have much lower polygon performance [STAR89, EVAN91]. Despite this impressive progress, current systems fall far short of cinerealism. To accurately model real-world scenes, more primitives must be displayed and antialiasing, texturing, and other features, such as shadows and transparency, are needed. All of these enhancements require extra processing. To implement them while generating images at real-time rates requires exceedingly high computation rates.

1.2 MULTIPROCESSOR GRAPHICS

The architects of general-purpose computers have dealt with the need for higher performance by increasing clock speeds and by performing operations concurrently. *Concurrent processing* has become a major area of research in computer architecture. It has two principal forms: pipelining and parallelism. These are also the major building blocks for high-speed graphics architectures. Virtually all interactive or real-time graphics systems exploit concurrency in some fashion [MOLN90].

There are two fundamental types of calculations involved in computing an image: *geometry calculations*, which convert primitives from their native coordinate system to screen coordinates, and *rasterization*, which converts geometry information into shaded pixels. Both must be performed rapidly and in balance with each other. Geometry calculations can be parallelized by processing multiple primitives and/or multiple primitive components (frequently vertices of triangles) in parallel. Rasterization can be parallelized by computing multiple pixels or multiple primitives in parallel.

Early graphics systems performed most operations serially, parallelizing only the calculations in the inner loops of the geometry and rasterization stages. These systems were comparatively simple and had straightforward programming models. Later systems, such as Silicon Graphics' VGX and Pixel-Planes 5, have applied multiple layers of parallelism to achieve higher performance. Such systems have very high peak rendering speeds, but are so complex to program that these speeds are seldom attained in actual applications.

1.3 IMAGE COMPOSITION

This thesis explores a larger-grained form of parallelism than has been traditionally been applied to the image-generation problem—entire images are computed in parallel. The central idea is to partition the display dataset statically over a number of independent renderers. Each renderer then computes an image of the primitives in its partition. The images from each renderer are then merged or composited into a final image of the entire dataset.

Architectures of this type are called *image-composition architectures*. The approach has many possible variations. For example, the dataset could be partitioned over the renderers so that images from each renderer can be overlaid in a predetermined order. Alternatively, each pixel of each image can have its own priority, so that composition is done on a pixel-by-pixel basis.

Image composition offers two intriguing advantages over existing architectural techniques:

- **Arbitrary scalability.** System performance can be scaled arbitrarily by increasing the number of renderers.
- **Simple programming model.** Renderers compute images of their portions of the database independently and, therefore, can be programmed as simple graphics systems.

These potential advantages make image composition an appealing method for building very high performance graphics systems, since they directly address two of the most troublesome problems of existing architectures.

The approach is not without its problems, however. Compositing images from each renderer requires very high data bandwidths throughout the system—higher even than video rates, since priority information, as well as color, are generally needed for compositing. Additional features such as antialiasing and complex shading models increase the required bandwidth further. Renderers cannot be built on a small scale either, since they must be able to generate full-resolution images at high frame rates.

1.4 THESIS SUMMARY AND CONTRIBUTION OF THIS WORK

This dissertation is not the first attempt to apply image-composition to generating images. Image composition has been defined, explored, and implemented in many different contexts (see Chapter 3). It is the basis, for example, of chroma-key, a common technique for superimposing video signals. It has been used in computer graphics to compute animations. It has also been used as a basis for several graphics architectures, such as Cohen and Demetrescu's polygon processor pipeline [DEME80] and General Electric's NASA II flight simulator [BUNK89].

This dissertation differs from previous work in that it uses image composition with large-scale (multiple-primitive) renderers to create very high performance systems. It also addresses the problem of antialiasing comprehensively, which has not been done before. The overall aim is to establish the following thesis:

> "Image composition forms the basis of a family of flexible, scalable image-generation architectures that can achieve higher performance than existing architectures."

We will do this in two parts: First we will explore the architectural space of image-composition architectures, examining variations on the approach and their advantages and disadvantages. Second, we will present the design of a prototype image-composition system to substantiate the claims in the thesis statement.

The prototype system design uses Pixel-Planes 5 technology and antialiases by supersampling. It is targeted for very high performance: a two card-cage system is expected to render 2.5 million Gouraud-shaded polygons per second, 2 million Phong-shaded polygons per second, 870 thousand antialiased polygons per second, and generate 1280x1024-pixel images at up to 90 frames per second. Additional card cages can be added for even higher performance.

A two card-cage version of the prototype system should have higher performance than any system demonstrated to date.[1] The system is realizable with current technology and directly supports a wide variety of primitives and rendering algorithms.

The prototype system is not described in complete detail, but many of the parts have been demonstrated in existing machines (particularly Pixel-Planes 5). The majority of Part II is devoted to motivating, explaining, and justifying parts of the prototype system that are different from Pixel-Planes 5 or have not been demonstrated before. The aim is to offer convincing evidence that the system is feasible as described, but not to document it in full detail.[2]

1.5 ORGANIZATION OF THIS DISSERTATION

The body of this dissertation is divided into two parts: Part I (Chapters 2–5) describes image-composition architectures in general; Part II (Chapters 6-9) describes the prototype system.

Part I (Chapters 2–5): Image-composition architectures

Chapter 2 provides background for the remaining chapters. It discusses the requirements for real-time graphics systems, summarizes approaches that have been taken in previous systems, and places image composition in a taxonomy of multiprocessor graphics architectures. Chapter 3 reviews related work by other researchers and describes the basic variations of image-composition architectures. Chapter 4 examines solutions to the aliasing problem. Chapter 5 discusses other issues that impact system performance or flexibility, such as the structure of the image-composition network, load-balancing, latency, and the suitability of image-composition architectures for advanced rendering algorithms. It presents a taxonomy of image-composition architectures and analyzes the economics of image-composition architectures compared to other approaches.

Part II: (Chapters 6–9): A prototype system design

Chapter 6 describes the motivations behind the prototype system design and describes the system at high level. Chapter 7 describes the image-composition network, a very high bandwidth network that supports real-time update rates with supersampling antialiasing. Chapter 8 describes the Renderer/Shader board, the heart of the prototype system. Chapter 9 describes the synchronization and control methods for prototype system and presents results of simulation and performance analysis.

Related documents

---

[1] In 1991 the fastest commercial graphics system is Silicon Graphics' SkyWriter, which has a maximum speed of 2.2 million flat-shaded, unlit, triangles per second [ROHL91]. Pixel-Planes 5, an experimental graphics system built at UNC, renders 2.3 million Phong-shaded triangles per second (demonstrated at Siggraph '91).

[2] We have plans to build a machine based on these ideas. It uses newer technology, runs at higher clock speeds, and achieves significantly higher performance than the system described here. It is not the focus of this dissertation because of its speculative nature, but is described briefly in Chapter 10.

The author has written (or co-authored) two documents that support this work and are referenced frequently within the dissertation:

MOLN90 Molnar, Steven and Henry Fuchs, "Advanced Raster Graphics Architecture", Chapter 18 in *Computer Graphics: Principles and Practice* by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, Addison-Wesley, New York, 1990, pp. 855–922.

This chapter describes the field of raster graphics architecture, the computational requirements for graphics systems, and the spectrum of approaches used to build high-performance graphics systems over the past 20 years.

MOLN91 Molnar, Steven, "Efficient Supersampling Antialiasing for High-Performance Architectures," Technical Report TR-91-023, Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

This technical report reviews the theory of supersampling and describes techniques to minimize the number of samples needed to antialias images in high-performance graphics systems, including image-composition systems.

PART I

IMAGE-COMPOSITION ARCHITECTURES

CHAPTER 2
REAL-TIME IMAGE GENERATION

The pioneer graphics systems of the 1960s used vector displays and could draw only dozens to hundreds of lines in real time [SUTH63]. Since then, the following features have become available in real-time systems:

- **Higher-complexity line drawings.** (examples: Evans and Sutherland's *Line Drawing System* in 1971 and Evans and Sutherland's *PS300* in 1981 [FOLE82]).
- **Shaded display with flat shading.** (examples: Evans and Sutherland's *NOVOVIEW SP1* in 1977 [SCHA83] and Megatek's *7200* in 1981 [FOLE82]).
- **Gouraud shading hundreds of polygons.** (examples: Evans and Sutherland's *CT-4* in 1977 [SCHA83]).
- **Direct display of non-polygon primitives.** (examples: Pixel-Planes 4 (spheres and Constructive Solid Geometry) in 1986 [FUCH85]).
- **Gouraud shading of thousands of polygons.** (examples: Silicon Graphics' *GTX* [AKEL88] and Stellar's *GS1000* [APGA88] in 1988).
- **Antialiasing.** (examples: various flight simulators, Stellar's *GS1000* in 1988, and Silicon Graphics' *VGX* in 1990 [HAEB90]).
- **Textures.** (examples: various flight simulators and Silicon Graphics' *VGX*.).

Until 1990, the only systems that could generate antialiased images with textures in real time were multi-million dollar flight simulators. These systems generally had relatively low polygon performance compared to graphics workstations [SCHA83]. Graphics workstations traditionally have focused on displaying large numbers of primitives, which are necessary for computer-aided design and other modeling applications [AKEL88, APGA88, BORD89, SGI90].

In the last several years the two approaches have begun to converge: flight simulators have increased their polygon performance and graphics workstations have begun to support antialiasing and texturing [EVAN91, AKEL90]. The consensus appears to be that real-time image-generation systems of the future must both increase polygon performance and provide realistic rendering to be successful [SPRO90].

This chapter presents background information on real-time image generation. Section 2.1 discusses objectives for performance and realism in real-time graphics systems. Section 2.2 discusses the computational requirements for such systems. Section 2.3 describes three classes of multiprocessor image-generation architectures (image composition is the third of these classes).

2.1 OBJECTIVES FOR REAL-TIME SYSTEMS

Current systems display extremely complex datasets (tens of thousands of primitives) in real time and support fairly realistic lighting and shading models. Future systems must meet or exceed these performance levels, should be free of distracting artifacts, and should provide support for more realistic rendering. We briefly review what we believe to be the important objectives for future real-time image-generation systems. (By their nature, such judgments are subjective, but we must have a target at which to aim a new design).

2.1.1 Screen Resolution

Ideally, a graphics system's display resolution should match the resolution of the human eye. Experiments on the human visual system indicate that the human eye can resolve features separated by 3 to 10 arc-minutes, depending on the brightness and contrast of the features [ROSE73]. If we assume an 18-inch wide display screen viewed from 18 inches (typical for a person viewing a desktop workstation monitor), this corresponds to a linear screen resolution of 350 to 1150 pixels. Current high-resolution monitors display 1280x1024 pixels. This appears to be a reasonable standard for high-performance image-generation systems.

2.1.2 Color Fidelity

The human visual system can distinguish a gamut of hundreds of thousands of colors. Color CRTs can display a large fraction, but not all, of these colors. To represent as much of the color gamut as possible, and to allow the linear combinations of color necessary for shading and antialiasing, a linear, three-component color model is needed (even more complex models may be desirable). RGB is the most common choice for graphics systems. To provide the illusion of continuous-tone images, at least 8 bits are required to represent each color component. (See [FOLE90] for a discussion of color in computer graphics and pointers to the literature).

2.1.3 Lighting and Shading

A graphics system must accurately model the interaction between light and the elements of a scene. The physics of light transport is fairly well understood. Unfortunately, to model the physics exactly for non-trivial scenes requires a prohibitive amount of computation. Rather than model the physics exactly, rendering algorithms make approximations that are less costly to compute.

The most realistic methods in common use are ray tracing and radiosity. These are extremely compute-intensive, often requiring days to compute a single image (radiosity has the advantage that it can be precomputed for static scenes). Generally, the more realistic the lighting and shading model, the more computation that is required.

Most current high-performance systems support Gouraud or Phong lighting and Gouraud shading. A few systems support Phong shading. Future systems should support Phong shading with multiple, local light sources.

2.1.4 Antialiasing

Aliasing is a common artifact, which results from attempts to display a continuous geometric image containing high spatial frequencies on a discrete display device with a comparatively low sampling rate. High-frequency components from the underlying image map incorrectly to lower frequencies, causing stair-stepping and moiré patterns in static images and crawling and scintillation in moving images (see Figure 4.2 for an example of a static image with severe aliasing). [FOLE90] provides a good introduction to the area and references to the literature.

The aim of any antialiasing method is to attenuate frequency components in the underlying image that are higher than the Nyquist frequency (the maximum spatial frequency that can be represented) of the display device. Antialiasing methods can be divided into two classes: *object-precision methods*, which determine pixel coverage analytically, and *image-precision methods*, which estimate pixel coverage by sampling [FOLE90].

Object-precision methods require a great deal of computation and are prone to special cases; consequently, they are expensive to implement. Image-precision methods are simpler and have fewer special cases; they have become the universal choice in high-performance graphics systems. Image-precision methods generally involve sampling scene geometry at subpixel resolution. This requires accurate point-sampling during rasterization.

Antialiasing is a compute-intensive process, however it is implemented. Most current systems that perform antialiasing do so with steep performance penalties (in some systems, such as flight simulators, antialiasing is a fundamental part of the application; even in systems such as these, where antialiasing cannot be "turned off", it consumes significant hardware resources that could have been spent elsewhere). Antialiasing plays a large role in determining which image-composition architectures are feasible. We will return to this subject at length in Chapter 4.

2.1.5 Frame Rate

To create the illusion of a moving image, the image must be updated rapidly. 24 Hz is a lower limit for motion to appear smooth. Even this is insufficient if the view or elements in the scene move rapidly. Flight simulation, one of the most demanding applications, requires update rates of 30 to 72 Hz. Frames must be double-buffered, so that only finished frames are presented to the user (unless the system is fast enough to refresh the screen ahead of the CRT beam).

2.1.6 Latency

Latency, the time between sampling user inputs and displaying the image, is a crucial issue for interactive, real-time systems. Users sense latency as a lag between movement of controls and a response in the visible image. Latency reduces controllability and the illusion of being immersed in the simulated environment. In certain applications, such as head-mounted displays or flight simulators, high latency can cause motion sickness [DEYO89].

Latency is an issue distinct from update rate. Many high-performance graphics systems increase their update rate by pipelining. For example, primitives of one frame can be transformed, while primitives from the preceding frame are rasterized, while primitives from the preceding frame are refreshed from a double-buffered frame buffer. Pipelining increases the frame rate, but does not improve latency. The lower bound for latency is the frame update time. Flight simulators perform predictive tracking to minimize the apparent latency, but user motions can not be predicted with complete accuracy and, therefore, the results are not perfect.

2.1.7 Scene Complexity

The factor that generally receives the most attention is the number of primitives that can be displayed per unit time. This determines the maximum complexity of scene that can be displayed at a given frame rate. The highest-performance systems available in 1991 display approximately 1–2 million polygons per second [SGI90, FUCH89]. At 30 Hz update rates, this corresponds to a scene complexity of approximately 30–60 thousand polygons. Such scenes do not approach the visual complexity of scenes in everyday life. Much higher performance is needed to display realistic scenes; performance in the range of several million triangles per second is desirable in the near future.

2.1.8 Sophisticated Effects

Sophisticated primitive types, such as curved surfaces and volume data, and advanced rendering methods, such as texturing, environment mapping, transparency, and shadows, greatly enhance realism, but are computationally expensive. Until recently, few graphics systems supported these capabilities in real time. As polygon performance grows, however, there seems to be a feeling that some of the increased performance should be spent on more sophisticated primitives and rendering methods.

Specialized hardware, which has traditionally been used to build high-performance systems, skews the system's performance toward some capabilities, but makes others less cost-effective. To be able to trade rendering performance for realistic effects, the system architecture must be flexible enough so that the same hardware resources can be used for either purpose, leaving the application designer or user to choose appropriately between the two.

2.2 THE RENDERING TASK

Image generation is extremely compute-intensive, particularly when real-time frame rates are required. It is also very regular and can be parallelized in many different ways. Consequently, image-generation has been a prime candidate for acceleration by special-purpose hardware.

[MOLN90], written by the author and co-author Henry Fuchs, is a general review of parallel rendering techniques. (It contains the traditional literature review and references for parallel image generation; Chapter 3 contains a literature review and references for image-composition).

In this section we describe the basic computations required to generate images. In Section 2.3 we introduce multiprocessing techniques that are needed to generate images at extremely high speeds.

2.2.1 Standard Rendering Pipeline

The basic image-generation process has changed little in the last 20 years [WATK70]. It can be modeled as a sequence of computational stages that transforms geometric descriptions of primitives into screen coordinates, then converts these descriptions into pixel values. Figure 2.1 shows a version of the rendering pipeline that is appropriate for *z*-buffer rendering of Gouraud or Phong-shaded polygons (other rendering algorithms require variations to the pipeline).

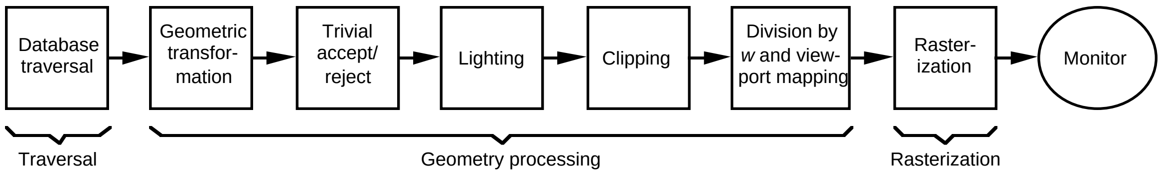


Figure 2.1: Standard rendering pipeline for *z*-buffer rendering of Gouraud or Phong-shaded polygons (adapted from [MOLN90]).

[MOLN90] describes each of these stages in detail and the architectural approaches that have been used to build high-performance systems. We refer the interested reader there for more information.

2.2.2 Computational Requirements

[MOLN90] also estimates the number of computations required in each stage to render a sample database containing 10,000 polygons. For a 1280x1024 image updated at 24 Hz, 81 million floating-point operations are required per second in the geometry stages, and 78 million integer operations and 122 million frame-buffer–memory accesses are required per second for rasterization.

The fastest floating-point processors currently available compute approximately 30 million floating-point operations per second; the fastest integer processors compute approximately 50 million integer operations per second; and the fastest DRAM memory systems have cycle times of approximately 70 nsec. The performance requirements for this modest application, therefore, exceed the capabilities of a single processor or single memory system. Current systems, such as the Silicon Graphics VGX, can display 46,000 polygons at 24 Hz, nearly five times higher complexity than this sample application [SGI90]. The VGX, and virtually all commercial high-performance graphics systems, use some form of parallelism.

2.3 MULTIPROCESSOR ARCHITECTURES

As mentioned above, the target of this dissertation is much higher performance than existing commercial graphics systems. A rough goal might be 100,000 polygons at 30 Hz update rates. Extrapolating the computation estimates above implies that approximately 1 billion floating-point operations, 100 million integer operations, and 150 million frame-buffer accesses are required per second (the floating-point calculations could be reduced by a factor of about 3 if mesh primitives are used).

If we want to render polygons (or other primitives) at these rates, the rendering task must be distributed over multiple processors at every stage in the rendering pipeline. We call architectures of this type *multiprocessor graphics architectures*. They are a relatively new topic in the field of graphics architecture.

2.3.1 Classes of Multiprocessor Architectures[1]

The 3D rendering task can be divided into three major parts: database traversal, geometry processing and rasterization (see Figure 2.1). The compute-intensive portions are geometry processing, which consists of floating-point operations on primitives in object coordinates, and rasterization, which consists of integer operations on primitives in screen coordinates. Extremely high performance systems must perform both in parallel.[2]

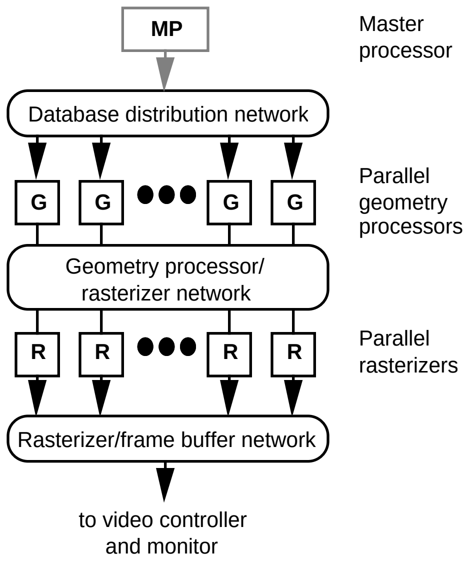

MP
Master processor
Database distribution network
G
Parallel geometry processors
Geometry processor/ rasterizer network
R
Parallel rasterizers
Rasterizer/frame buffer network
to video controller and monitor

Figure 2.2: Canonical multiprocessor graphics system. G units are geometry processors. R units are rasterizers.

We can describe a multiprocessor graphics architecture as a cluster of geometry processors and a cluster of rasterization processors connected by an appropriate set of communication paths, as shown in Figure 2.2. Primitives are initially specified in object coordinates and may be transformed to fall anywhere on (or off) the screen. The communication paths convey primitives from one set of processors to another, redistributing primitives (or fractions of primitives) where necessary.

The redistribution or *sorting* of primitives onto the screen is a major problem to solve in distributing the rendering task across multiple processors. There appear to be three distinct ways of solving the problem, illustrated in Figure 2.3.

[1]This classification is the result of extensive group discussions among the Pixel-Planes team. It does not fit existing machines particularly well, since few of them are designed for the performance levels we assume here (and, therefore, do not require such high degrees of parallelism). The classification provides a way to think about future systems with extremely high performance.

[2]Database traversal can be problematic as well. It is generally considered to be part of the application program, and, therefore, has received little attention in the literature. Attempts to parallelize geometry processing can affect the traversal method and vice versa.

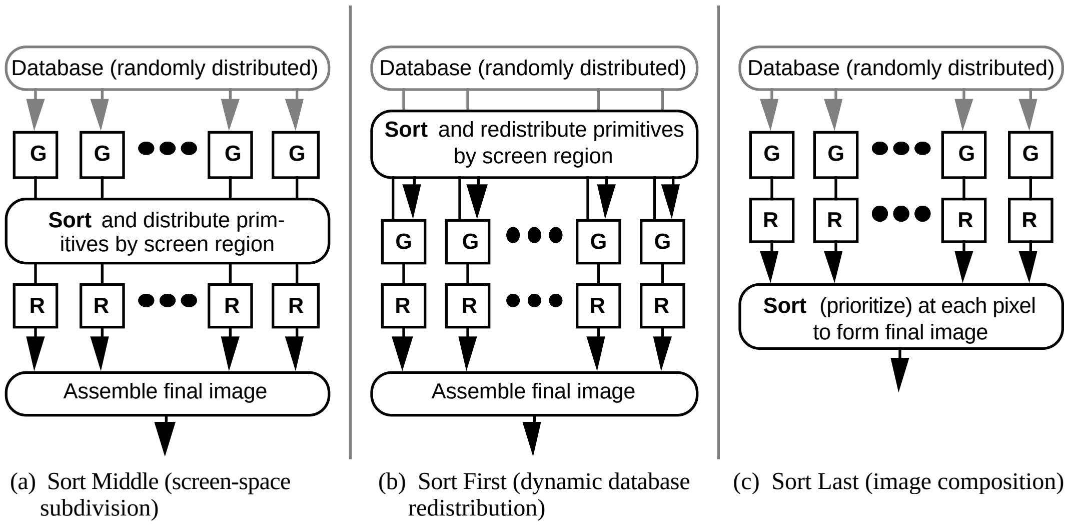

Database (randomly distributed)
G
Sort and distribute primitives by screen region
R
Assemble final image
Database (randomly distributed)
Sort and redistribute primitives by screen region
G
R
Assemble final image
Database (randomly distributed)
G
R
Sort (prioritize) at each pixel to form final image
(a) Sort Middle (screen-space subdivision)
(b) Sort First (dynamic database redistribution)
(c) Sort Last (image composition)

Figure 2.3: Three classes of multiprocessor graphics architectures: (a) Sort Middle, (b) Sort First, and (c) Sort Last.

The sort can take place before geometry processing (Sort First), between geometry processing and rasterization (Sort Middle), or after rasterization (Sort Last). We will discuss the properties of each in turn.

2.3.2 Sort Middle (Screen-Space Subdivision)

A natural way to parallelize the geometry calculations is to assign each geometry processor a fraction of the primitives. A natural way to parallelize rasterization is to assign each rasterization processor a fraction of the pixels of the screen (a time-proven method to accelerate rendering).

In traditional architectures, there is a choice of whether to assign each rasterization processor a contiguous region of pixels (Figure 2.4(a)) or an interleaved set of pixels (Figure 2.4(b)). In the interleaved scheme, each rasterizer must handle all of the primitives. For the performance levels we are assuming, there simply are too many primitives for this to be feasible. Dividing the screen into contiguous regions reduces the number of primitives that each rasterization processor must handle.

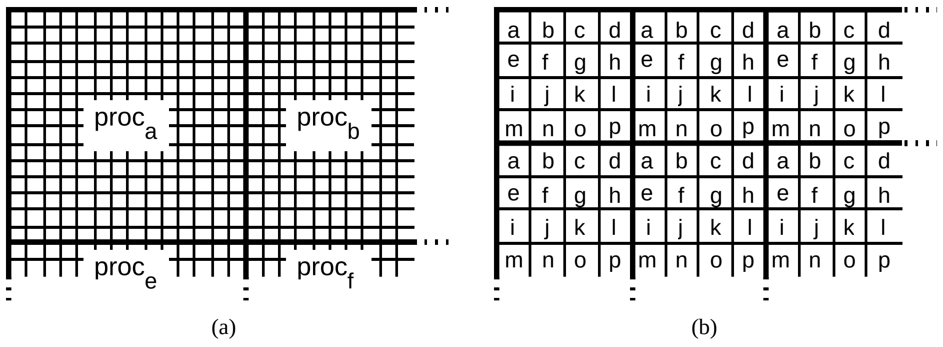


Figure 2.4: Screen-partitioning methods: (a) each processor is assigned a contiguous region of pixels; (b) each processor is assigned pixels in an interleaved pattern.

This distribution of work gives rise to the sort-middle architecture. We need a sorting network to connect the parallel geometry processors and parallel rasterization processors, since the primitives on any geometry processor may fall anywhere on the screen. This network must take transformed primitives, determine which region (or regions) they affect, and convey them to the appropriate rasterizer. It must be a global network, since it receives primitives from all of the geometry processors and must send them to all of the rasterizers. It must have high bandwidth, since it transfers a description of the entire dataset between geometry processors and rasterizers every frame.

In a sort-middle architecture, the rendering process proceeds as follows: Primitives are transformed, sorted by screen region, and routed from geometry processors to rasterizers over a high-performance network. Each rasterizer then paints a complete image of its region of the screen. When all of the regions are complete, the image fragments are collected and assembled in an image buffer for display.

Sort Middle has the following advantages:

- The database distribution method is unconstrained by the assignment of rasterizers to screen regions or the particular view.
- Each rasterizer handles all of the primitives in a screen region, so the rasterization method is unconstrained.

Sort Middle has the following disadvantages:

- It requires a distributed display database.
- It requires a global communication network, which must transfer all of the primitives between geometry processors and rasterizers every frame. The bandwidth requirements for this network scale linearly with performance.
- It is susceptible to load imbalances between rasterizers when primitives are distributed unevenly over the screen.
- It has difficulty with order-dependent primitives. Primitives from different geometry processors can arrive at the same rasterizer in a non-deterministic order. (A synchronization method, such as described in [TORB87], can be used.)
- Its latency is high. All of the primitives must be sorted into screen regions before the rasterization of any region can finish.

To the author's knowledge, only one sort-middle system has ever been implemented: Pixel-Planes 5 [FUCH89]. The system's demonstrated performance of 2.3 million Phong-shaded triangles per second is the highest reported to date. Its communication network has a limited (but large) amount of bandwidth available. This limits its maximum performance.

2.3.3 Sort First (Dynamic Database Redistribution)

Sort First uses the same screen-space subdivision approach as Sort Middle. However, geometry processors and rasterizers are coupled together, forming complete rendering units called *renderers.* Each renderer is made responsible for a single screen region. The distinguishing feature of this approach is that after the initial random distribution of primitives to renderers, primitives are redistributed by screen region to the appropriate renderer. This requires transforming primitives into screen coordinates, classifying them with respect to region boundaries, and transferring any primitives that fall outside of its renderer's region to the appropriate renderer—all before rasterization can begin.

Sort First can be efficient if there is frame-to-frame coherence—similarities between successive frames. The hope is that, with high frame rates, most primitives processed by a renderer in one frame will be processed by the same renderer in the next frame. Only primitives that cross region boundaries must be transferred to adjacent renderers. Some provision must be made for primitives that fall off the screen, since they may reappear in future frames.

Sort First has the following advantages:

- Rendering is simple after sorting. After sorting, renderers are independent; each computes a single partial-screen image.

- Its bandwidth requirements may be low. Frame-to-frame coherence should minimize the number of primitives that must be transferred each frame.
- Only local communication is required. Primitives generally are transferred to neighboring renderers.

Sort First has the following disadvantages:

- It is susceptible to load imbalances. Primitives may clump, concentrating the work on a few renderers.
- Screen boundaries cause complications. It is not clear where to send primitives that fall off the screen. They may reappear somewhere else.
- Its programming model and data structures are complex. The database must be redistributed dynamically.

Because of its inflexibility and load-imbalance problems, Sort First probably is impractical except for specialized applications, where the distribution of primitives across the screen is relatively uniform and frame rates are high (flight simulators may be one such example). To the author's knowledge, this technique has never been implemented.

2.3.4 Sort Last (Image Composition)

In Sort Last, geometry processors and rasterizers are paired to form renderers, as in Sort First. Each renderer is assigned a random portion of the database, as in Sort Middle. The difference is that, now, renderers are made responsible for the entire screen. Each renderer computes a full-screen image of its portion of the primitives. These partial images are then *composited* together, a priority sorting that eliminates surfaces in one partial image hidden by those in another.

Sort Last or *image composition* has the advantage that no sorting or redistribution of primitives is required in the renderers; each renderer computes the image of its primitives as if it were the only one in the system. The "sorting" network in this architecture takes the form of a high-bandwidth "smart" network that composites images. The simplest way to do this is to composite images pair-wise. This results in either a binary-tree or pipeline structure for the image-composition network. Since the network must operate at high speeds to maintain real-time frame rates, special composition processors or *compositors* are required. The compositor implementation may restrict the kinds of rendering algorithms that the architecture can support.

Since a tree-structured or pipelined image-composition network can accommodate an arbitrary number of nodes, and renderers compute their subimages independently, image-composition architectures have an intriguing property: they can be scaled to arbitrarily high performance by adding renderers and compositors.

Sort Last has the following advantages:

- Renderers have a simple programming model. They compute their partial images independently.
- Load balance is automatic. Each renderer can be given approximately the same amount of work.
- Communication is local and the bandwidth is constant. It is determined by the maximum rate images can be composited.
- The architecture is linearly scalable.

Sort Last has the following disadvantages:

- It requires a distributed display database.
- It restricts the class of rendering methods. The rendering method must produce pixels in format suitable for compositing.
- It requires an image-composition network with very high bandwidth.

Because of the high bandwidth requirements and expense of the image-composition network, Sort Last has been explored very little until recently. Its property of arbitrary scalability and simple programming model, however, make it appealing for very high-performance systems.

2.3.5 Comparison of Approaches

Each multiprocessing strategy has advantages and disadvantages for certain applications and rendering algorithms. The sort operation is disruptive wherever it occurs. In sort-first architectures, the distribution of the database is constrained; in sort-middle architectures, the rendering process must be "interrupted" to allow for primitive redistribution; in sort-last architectures, all images must have a consistent format, suitable for compositing.

If sorting is performed first or last, geometry processors and rasterizers can be combined into complete renderers, allowing flexible and high-bandwidth communication between the two parts. Since renderers are complete graphics systems, many of the standard approaches for building graphics systems can be employed.

Sort First appears to be of limited use because of its load-balance and screen-boundary problems. Sort Middle has proven itself to be a powerful approach in Pixel-Planes 5. David Ellsworth is analyzing the properties of this architecture as part of his PhD dissertation [ELLS91]. Sort Last has received little attention so far, but appears very promising. It is the focus of the remainder of this dissertation.

CHAPTER 3

IMAGE COMPOSITION

As we saw in the Chapter 2, multiprocessing appears to be inevitable if we wish to compute high-resolution images of complex datasets at high update rates. The sort from object coordinates to screen coordinates is a fundamental operation in all such systems. In this chapter and in the remainder of this dissertation, we will be concerned with sort-last or image-composition architectures—architectures in which the sort takes place after individual images (of part of the dataset) have been rendered.

As we saw in Chapter 2, image-composition architectures offer the following potential advantages:

- **Scalability.** Since renderers operate independently and a tree-structured or pipeline image-composition network can accommodate an arbitrary number of renderers and compositors, the performance of the system can be scaled by adding renderers and compositors to the system.
- **Simple programming model.** Aside from database distribution and editing, the rendering process on each renderer is independent of all of the others. The parallelism of the overall system is hidden from the point of view of the programmer.
- **Flexibility.** The composition method is independent of the algorithm used to generate the sub-images. Subimages can be generated by any algorithm that produces images of the proper format.

In this chapter we consider several variations on the image composition theme. Each has its own set of advantages and disadvantages. Indeed, image composition forms the basis for a variety of rendering architectures with widely differing properties.

Section 3.1 reviews previous work in image-composition architectures. Section 3.2 describes systems that overlay images in which every pixel has the same priority. Section 3.3 describes systems that provide separate priorities for each pixel.

3.1 PREVIOUS WORK

Image composition has been used in various forms for many years—not all of them related to computer graphics. We now review previous research results and system implementations based on, or related to, image composition.

3.1.1 Processor-per-Primitive Systems

Systems with multiple processors that are each responsible for a single primitive (so-called *processor-per-primitive* systems), are a simple form of image-composition architecture. Generally, pixel streams produced by each processor are composited together by passing the streams between processors or feeding them into a composition network.

This approach was used to build some of the earliest real-time graphics systems. General Electric's NASA II flight simulator, built in 1967, was one of the first such systems [BUNK89]. The NASA II allocated one polygon (face) per processor (face card). Each face card computed the image of its polygon in scan-line order at video rates. Faces were each assigned a fixed priority. A hardware priority multiplexer chose the visible pixel with the highest priority and forwarded it on to the display. The NASA II could only accommodate 60 face cards, and the polygon on each face card had a single color. The performance of the system was, therefore, quite limited.

Cohen and Demetrescu proposed a processor-per-primitive system in which priorities (depth or z values) are associated with individual pixels, rather than for entire polygons [DEME80]. Pixels would stream through a pipeline of triangle processors at video rates, with the color value of the closest surface emerging from the end.

Fussell proposed a similar system, in which triangle processors would feed their pixel streams (with *z* values) into a binary tree of comparators [FUSS82]. The advantage of the binary-tree organization was that triangle processors could work on the same pixel at the same time.

Weinberg proposed an enhancement to Demetrescu's system for antialiasing. Variable-length packets containing subpixel information would be sent between pipelined polygon processors, and a filtering module would filter the subpixels appropriately [WEIN81]. A difficulty with this approach is that the amount of information representing each pixel can increase and decrease as the pixel flowed through the pipeline. Weinberg estimated that the number of surfaces per pixel packet would be less than two for typical images.

Deering et. al. proposed a pipelined, triangle-processor system similar to Cohen and Demetrescu's with enhancements to compute Phong shading and to improve the efficiency of the triangle processors [DEER88]. Rather than computing color values in each processor and sending them down the pipeline, the processors would compute the components of the surface-normal vector for each pixel and send these, together with the polygon's intrinsic color, down the pipeline. A separate shading processor called a Normal Vector Shader, at the end of the triangle-processor pipeline, would evaluate the Phong lighting model for each pixel. The design also allowed triangle processors to be loaded with several triangles during a frame time, so long as the triangles do not overlap in *y*. This enhancement allows the a system with *n* triangle processors to render scenes containing significantly more than *n* triangles.

Schneider and Claussen proposed a pipelined triangle-processor system called PROOF (Pipeline for Rendering in an Object-Oriented Framework), that combines elements of Weinberg's and Deering's proposed systems [SCHN88]. Like Weinberg's system, PROOF would pass a variable-length list of objects between processors for each pixel. Like Deering's system, shading would be performed by a shading processor at the end of the triangle-processor pipeline.

Kedem and Ellis proposed and built a system for ray-casting Constructive Solid Geometry (CSG) objects [KEDE84]. In this system, primitives of a CSG tree are assigned to custom processors called *Primitive Classifiers*. Other custom processors, called *Combine Classifiers*, at the interior nodes of the CSG tree perform *z*-comparisons and the in/out classifications required for CSG. Kedem and Ellis have built a prototype ray-casting system, which has 256 Primitive Classifiers and 256x8 Combine Classifiers. The system performs 100–150 million ray/primitive classifications per second, but cannot display frames in real time because of a low-speed connection to the frame buffer [KEDE91].

All of the processor-per-primitive systems described above share a common limitation: their performance degrades rapidly when the number of primitives exceeds the number of processors.

3.1.2 Video-Overlay Systems

Simple overlays have been used to composite images in video editors, video games, and in flight simulators—applications in which one image has absolute priority over another image. Video editors typically use chroma-keying, in which pixels in the primary image having a particular color (such as saturated blue) are overwritten by pixels in the secondary image. A technique used in some flight simulators is to build a pipeline of processors, each of which generates an image of increasing priority. Pixels generated by one processor overwrite those generated by previous processors. This is effective, for example, when aircraft always occlude lights, which in turn always occlude surface terrain.

3.1.3 Image-Composition Algorithms

The general image-composition problem allows pixels in each image to have independent priorities. The simplest solution is to compare priority or *z* values of corresponding pixels in each image to determine which one has highest priority. This composition method suffers from the same types of image aliasing as the conventional *z*-buffer algorithm.

Duff described a more sophisticated composition algorithm that uses *z*-values at the four corners of each pixel to approximate the fractional contribution from each image [DUFF85]. Although economical, this technique does not handle every case correctly (it is adequate for compositing uncorrelated images produced by separate renderers, such as airplanes over terrain, but fails when primitives from different images share common edges). Duff recognized this limitation, but argued that it was of little consequence for the applications he intended. We will discuss Duff's algorithm in more detail in Section 3.3.1.

Nakamae et. al. described a technique based on decomposing multiple images into visible spans on scanlines with higher *y* resolution than the raster display [NAKA89]. The composite image is computed by merging spans from multiple images. The subscanline spans allow an antialiased color to be calculated for each pixel.

3.1.4 Large-Grain Image-Composition Systems

We discussed image-composition systems with a process per primitive in Section 3.1.1. Several image-composition systems with multi-primitive renderers have been proposed as well. Fuchs and Johnson sketched a graphics system architecture that computes images of portions of a display list on separate processors and combines them using a priority encoder [FUCH79]. The portion of the paper that included the sketch was not included in the conference proceedings.

Leray proposed a device for combining two video streams containing *z* information as well as color information [LERA87]. The *z* values would be used to determine priorities for each pixel of the composite image. He proposed using these devices to place synthetic objects in real scenes or to combine the outputs of two *z*-buffer frame buffers.

Bender et. al. proposed a system based on Inmos Transputers that uses a BSP-tree algorithm to compute polygon priorities and a DMA engine to combine pixels from several frame buffers into a composite frame buffer [BEND89].

Molnar proposed combining the outputs of several *z*-buffer renderers using a binary tree of *z*-comparator/multiplexers and argued that such a system could achieve linearly scalable performance at constant performance vs. price [MOLN88].

Shaw et. al. proposed implementing a simplified version of Duff's algorithm in VLSI chips to make possible a multi-renderer system with antialiasing [SHAW88]. The compositor chip was fabricated, but a prototype system was not completed.

3.1.5 Contribution of this Work

This dissertation uses many of the ideas just summarized. It makes the following new contributions:

- It identifies the issues involved in building high-performance image-composition systems with large-grain renderers.
- It addresses the antialiasing problem comprehensively.
- It describes a prototype design that demonstrates the feasibility of the approach.

The goal of this research is to explore this new, promising architectural space and to show that many of its potential benefits can be realized in practical systems.

3.2 FIXED-PRIORITY IMAGE COMPOSITION

The simplest variant of image composition assigns a fixed priority to each subimage. Pixels within subimages may encode, in addition to color, the transparency (or opacity) of the subimage at that pixel. Multiple subimages can be overlaid, creating a complicated image from a number of simple ones.

3.2.1 Chroma Key

A simple and early method of overlaying images is analog chroma-key. In this method a particular color that does not appear in the image is designated as the *key* color. Fully saturated blue is a common choice. Each subimage is recorded over a background of the key color (this can be done simply by videotaping a subject against a blue background). An analog chroma-key device, diagrammed in Figure 3.1, composites the image. The chroma keyer compares the color of the foreground image to the key color. If the colors differ, it transmits the foreground color; if the colors match, it transmits the background color.

Multiple chroma-key devices can be cascaded to composite multiple several images in real time. Alternatively, images can be stored on videotape after compositing and composited with additional images in successive passes. Chroma key is commonly used in the video and special effects fields. Lucasfilm used it to create the space combat special effects in Star Wars. Television news broadcasts and commercials frequently use it to overlay commentators over background images or video footage.

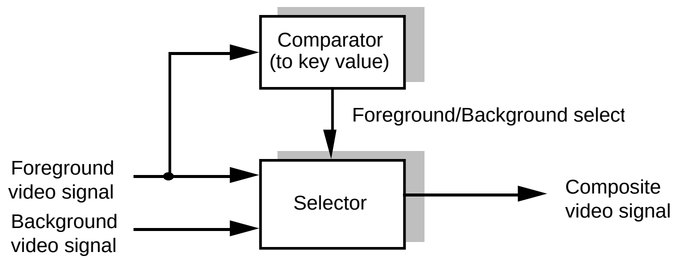


Figure 3.1: Block diagram of chroma-key compositor.

Figure 3.2 shows a 160x128-pixel image of three polygons composited using chroma-key. Separate images were computed for each polygon over a fully-saturated blue background (the key color). The images were composited in front to back order on top of the blue background image. Note that a blue boundary can be seen around the polygon silhouettes. This occurs because the key color bleeds into pixels that are only partially covered by the front image. This effect can be seen occasionally on television images.

Figure 3.2: Chroma-key overlay of three 160x128-pixel images (one for each polygon) over a blue background.

3.2.2 RGBα Overlay

Although chroma-key compositors typically operate on analog video, they can be designed to operate on digital video as well. A more appealing technique for digital video is to encode transparency information in a channel separate from the color [PORT84]. This prevents ambiguities when an image contains colors close to the key color and allows blends between the two images. The extra channel is generally called *alpha* or α. An alpha value of 1 indicates an opaque foreground pixel; an alpha value of 0 indicates a transparent foreground pixel. Intermediate values of alpha, if allowed, represent varying degrees of transparency.

A digital RGBα compositor is more complicated than its chroma-key counterpart. It must linearly interpolate between the background and foreground color based on alpha. This requires multipliers and adders for each channel. The composited color is given by the composition equations:

$$R_{\text{comp}} = \alpha_{\text{front}} \bullet R_{\text{front}} + (1 - \alpha_{\text{front}}) \bullet R_{\text{back}}$$
$$G_{\text{comp}} = \alpha_{\text{front}} \bullet G_{\text{front}} + (1 - \alpha_{\text{front}}) \bullet G_{\text{back}}$$
$$B_{\text{comp}} = \alpha_{\text{front}} \bullet B_{\text{front}} + (1 - \alpha_{\text{front}}) \bullet B_{\text{back}}$$

Since each channel typically is represented in eight or ten bits, these functions cn be implemented in relatively simple hardware. Figure 3.3 shows a block diagram of the compositor for one color channel.

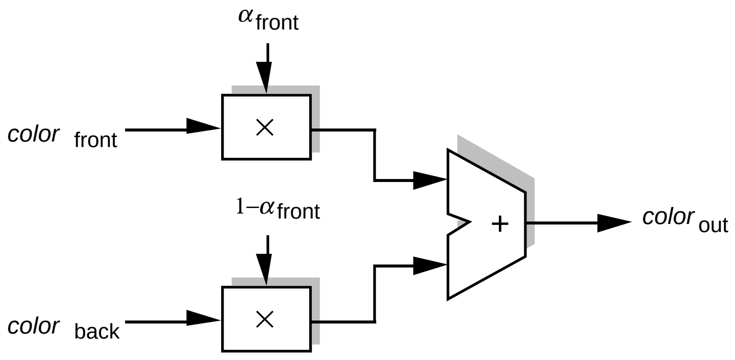


Figure 3.3: Block diagram of one channel of an RGBα compositor.

Alpha can be used to smooth the edges of images. For example, the boundary of an object in the foreground image may not coincide with pixel boundaries. It would be improper to color border pixels completely in either the foreground or background colors. If alpha is set to the fraction of the pixel actually covered in the foreground image, it can be used to smoothly blend the foreground and background colors together, reducing stair-step effects or *aliasing*.

Figure 3.4 shows an RGBα overlay of the three-polygon image of Figure 3.2. Notice that a hint of red and blue is apparent in the pixels lying on the boundary between the two white polygons. Such *bleed-through* artifacts can occur when a pixel is partially covered in two or more images. Although the seam between the two white polygons obscures the background completely, the compositing algorithm has no way to distinguish this case from cases in which two polygons partially cover a pixel, but leave the background exposed.

Figure 3.4: RGBα overlay of images above.

Figures 3.5(a–c) show the possibilities that can occur when more than one polygon partially covers a pixel. Each figure is a blow-up of one pixel, in which two polygons (in separate images) partially cover a background image. Polygon *A* is in the front image; polygon *B* is in the middle image; the background is in the far image. Although α_A and α_B are the same in all three cases, the amount of background image visible differs in the three cases.

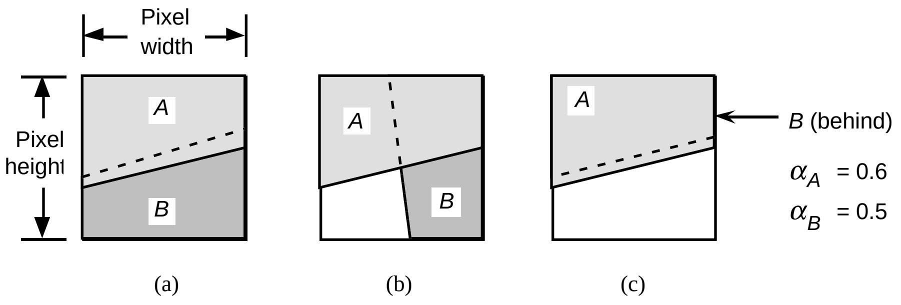


Figure 3.5: Front polygon *A* and middle polygon *B* have the same coverage in cases (a), (b), and (c), but expose differing amounts of the background color.

Since there is no way to distinguish between these cases when compositing, an arbitrary choice has to be made. Typically, the coverage fractions of the original pixel are used to determine the coverage fractions of the area left uncovered by the front surface, as in Figure 3.5(b). When this assumption fails, artifacts are produced. These artifacts are most noticeable when subimages contain correlated features, such as the shared edge between polygons in the example above. When component images are independent, as in several spaceships over a field of stars, the artifacts occur at random pixels and are difficult to detect.

RGBα compositing has been used in animation systems, to composite images from different renderers [PORT84], and in flight simulators, such as the IVEX VDS-2000, to superimpose images of lights and aircraft over images of terrain features and the runway environment [GREE91].

3.3 PIXEL-PRIORITY IMAGE COMPOSITION

Fixed-priority compositing methods are severely limited in that every pixel in a subimage is assigned the same priority. They cannot handle images that overlap in nontrivial ways, such as aircraft flying between mountains in a flight simulator, or a person standing in a room with furnishings. A more general form of image composition, called *pixel-priority* image composition, remedies this by providing each pixel with an independent priority. Priorities generally are based on the distance from the viewer to the nearest surface visible at each pixel—the pixel's *z* value. Pixel priorities or *z* values are generally encoded as an additional channel beside the color (and, optionally, alpha) channels.

Pixel-priority image composition, in its simplest form, is easy to implement. Renderers scan out *z* values along with color values. To compute each pixel in the final image, *z* values of corresponding pixels of each subimage are compared. The pixel with the lowest *z* value "wins"; its color is used in the final image. *Z* values can indicate coverage as well as priority: pixels whose *z* value is set to *zmax* (the largest representable *z* value) are effectively transparent, since they have the lowest possible priority. This method does not allow intermediate transparency values, however.

3.3.1 The Aliasing Problem

Although pixel-priority image composition is more flexible than fixed-priority composition, it makes antialiasing more difficult. We could include an alpha channel with each image to encode partial coverage, as we did before, but a new problem emerges: the result depends on the order in which images are composited.

Consider the side view of a pixel shown in Figure 3.6. Here, three polygon fragments *A*, *B*, and *C* are visible in the final pixel. Assume that each polygon fragment is stored in a separate image, and that each pixel is represented by three components: color (RGB), α, and *z*. We wish to calculate the composited pixel value *A comp B comp C*.

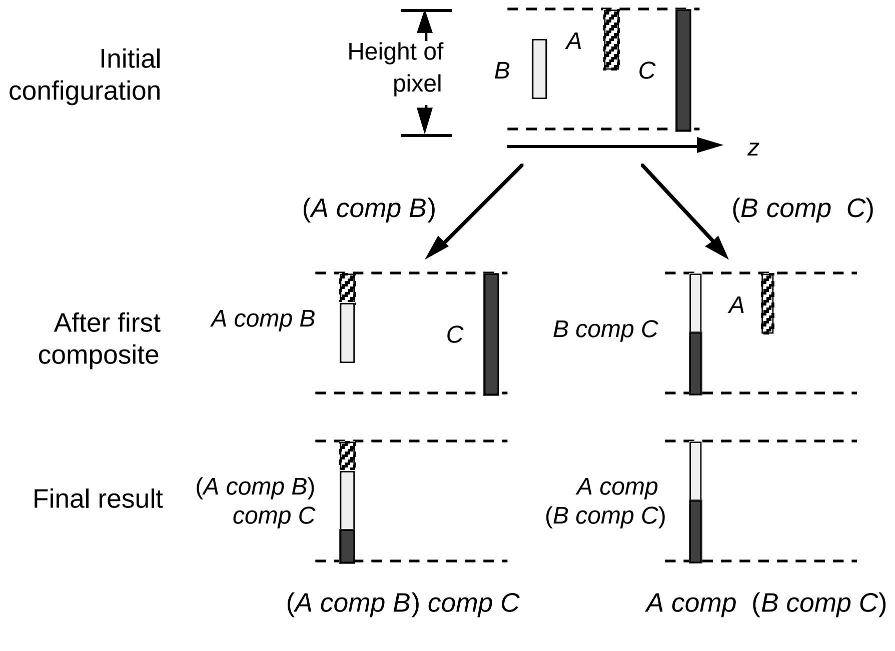


Figure 3.6: Non-associativity of alpha-blend compositing. *A*, *B*, and *C* are polygon fragments that partially cover the pixel (pixel is viewed from the side; *z* increases to the right).

Polygons *A* and *B* each cover half of the pixel. Polygon *C* covers the pixel completely, but lies behind the other two polygons. To compute *A comp B comp C*, we can compute either *A comp B* or *B comp C* first, indicated by the two arrows in the figure. The figure shows the pixel's state after the first step in both cases.

After the second composite, we see that the final result differs in each case. The result of (*A comp B*) *comp C* is correct except for possible bleed-through artifacts. The result of *A comp* (*B comp C*) is incorrect, however, since the contribution of *B* is lost. We see from this example that *comp* is not associative and produces errors if pixels are composited in the wrong order. Since each image can have arbitrary *z* values at each pixel, there is no way to assure the correct compositing order for every pixel in an image.

Duff described an alternate compositing algorithm that mitigates this problem under certain circumstances [DUFF85]. He proposed representing pixels by RGBα, as before, but sampling *z* values at pixel corners, rather than at their centers. Since corner *z* values are shared by four adjacent pixels, an entire image of this form can be stored in nearly the same space as a regular image (only one extra row and column are required).

The advantage of sampling *z* values at the corners of pixels is that extra context information is available at each pixel to reduce compositing errors. The first step in compositing two pixels *A* and *B* is to determine which *z* value is nearest at each of the four corners. There are $2^4 = 16$ possibilities, shown in Figure 3.7.

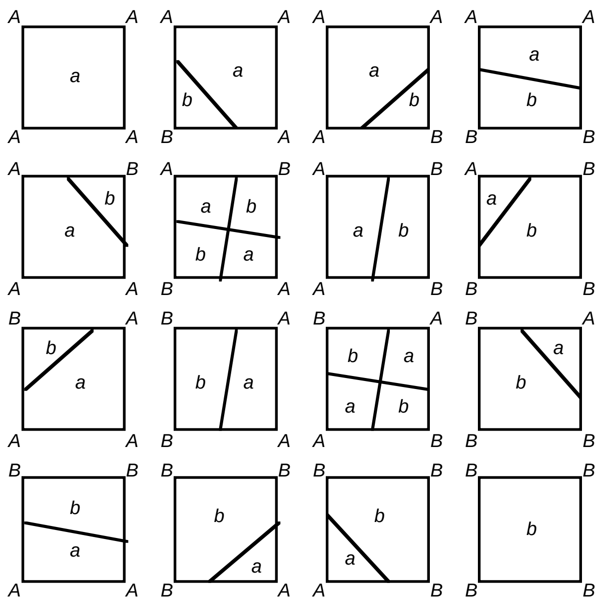


Figure 3.7: 16 possible coverage cases for Duff's algorithm. Pixel corners are labelled *A* or *B* depending on whether z_A or z_B is in front. Labels *a* and *b* indicate regions where *A* or *B* are assumed visible (figure adapted from [DUFF85]).

The fractional coverage of *A* and *B* visible in the composited pixel is determined by linearly interpolating *z* values for *A* and *B* along each pixel edge and determining the point where they intersect. If pixels have unit area, then area *a* is the fraction that *A* is assumed to be in front and *b* is the fraction that *B* is assumed to be in front. The final pixel value is given by $a \bullet \mathrm{RGB}\alpha_A + b \bullet \mathrm{RGB}\alpha_B$, where $\mathrm{RGB}\alpha_A$ is the composited pixel value assuming *A* is in front and $\mathrm{RGB}\alpha_B$ is the composited pixel value assuming *B* is in front.

Duff's algorithm works correctly when one pixel covers the other at all four *z* values and when a single covering polygon in *A* intersects a single covering polygon in *B*. The method breaks down, however, when polygon edges fall within a pixel, a limitation recognized by Duff. Figures 3.8(a) and 3.8(b) show close-up views of two pixels that demonstrate this problem. In each case polygons *A* and *B* have the same RGB, α, and *z* values. In Figure 3.8(a), the polygons share a common edge and completely obscure the background, whereas in Figure 3.8(b), the polygons leave a substantial portion of the background visible. Duff's algorithm cannot distinguish between the two cases and makes an intermediate assumption. This results in color bleeding similar to that produced by fixed-priority alpha-blending.

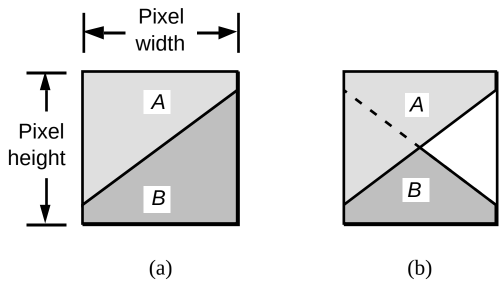


Figure 3.8: Coverage ambiguity in Duff's algorithm. Cases (a) and (b) result in the same composited pixel value.

If uncorrelated images are composited, such as independent objects in a complex scene, artifacts occur only at isolated pixels and are relatively benign. When images are correlated—for example when primitives in separate images share an edge—the artifacts are systematic. Unfortunately, shared edges appear to be a necessary evil to assure load balance across the renderers in an image-composition system (see Section 5.4). This makes Duff's algorithm a poor choice for general-purpose image-composition systems.

Compositing algorithm requirements. We have seen that the compositing algorithm for a pixel-priority image-composition architecture must have several properties to avoid errors such as color bleeding and dependence on the compositing order:

- **Commutativity.** The final image should not depend on the order in which images are composited.
- **Associativity.** The final image should not depend on the order in which images are composited.
- **Explicit subpixel geometry.** To avoid color bleeding, we need a method that retains the coverage geometry of primitives that partially cover pixels. This can be done analytically (difficult) or discretely (by sampling).

These requirements preclude the simple antialiasing schemes that were effective in fixed-priority image-composition systems. They force us to us to investigate more general approaches, such as supersampling and the A-buffer algorithm. We will return to this issue at length in Chapter 4.

3.3.2 A Simple Pixel-Priority System[1]

If we ignore antialiasing for the moment, the composition operation is very simple: *z* values are compared and the nearer pixel is chosen. This can easily be implemented in software or hardware. Figure 3.9 shows a block diagram of a simple hardware implementation. Such a hardware compositor could run at real-time video rates or higher.

Pixel-priority image composition can be used as the basis for high-performance image-generation architectures. Separate priorities at each pixel allow the visibility-determination algorithm to be distributed over the image-composition network, providing the architecture with its tantalizing properties of scalability and near-constant performance vs. price.

[1]This section is based largely on [MOLN88] and has elements from several previously published proposals, reviewed in Section 3.1.4.

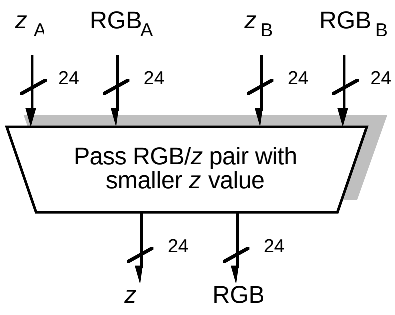


Figure 3.9: Block diagram of a simple pixel-priority compositor.

Figure 3.10 shows a block diagram of a simple *z*-buffer image-composition system. Eight rendering engines, each capable of rendering 400,000 Gouraud-shaded triangles per second (typical of current commercial systems) are harnessed together, providing a net rendering speed of 3.2 million triangles per second. Each renderer in this system is equipped with its own full-screen color and *z*-buffer.

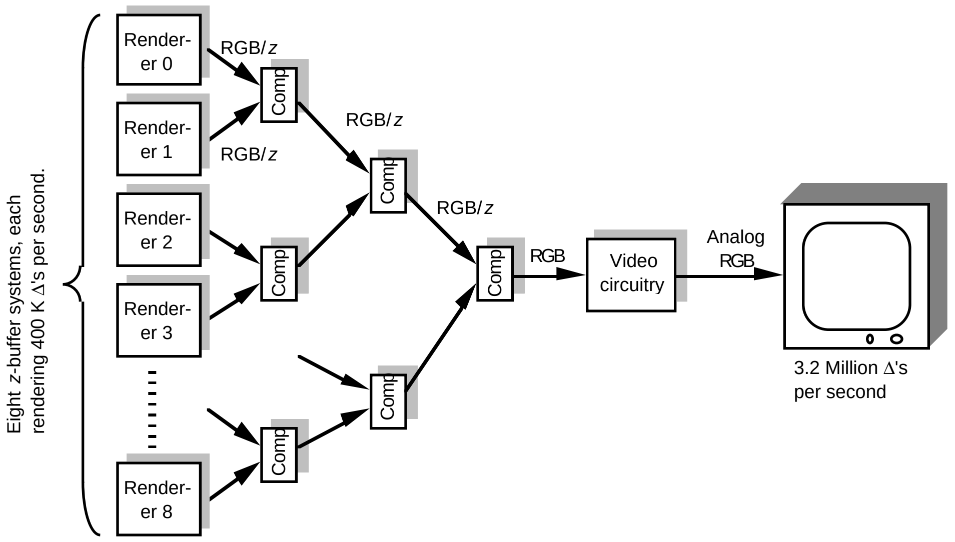


Figure 3.10: Pixel-priority system for displaying 3.2 million triangles per second.

Each renderer is assigned a fraction of the display database during system initialization. Rendering proceeds as follows: renderers transform their respective primitives into screen coordinates using identical modeling and viewing transformations, and rasterize them into their own color and *z* buffers. Next, the systems synchronously scan out color and *z* values into a binary tree of compositors, which successively combine them into the single RGB/*z* stream that emerges from the root of the composition tree. The RGB stream is fed into conventional color look-up-tables and digital-to-analog converters to drive the display monitor. This system operates at video rates and, therefore, requires no final frame buffer. Each additional renderer requires one additional compositor, so performance vs. price is constant.

***Z*-buffer Renderers.** The video scan-out mechanism of a conventional frame buffer provides a ready-made interface to the image-composition network. Normally RGB values for each pixel are scanned out in raster order and sent to the display device (see Figure 3.11). If we modify the frame buffer to scan out *z* values as well as RGB values, and to do so synchronously on all renderers, we can build a *z*-buffer composition system out of conventional rendering engines.

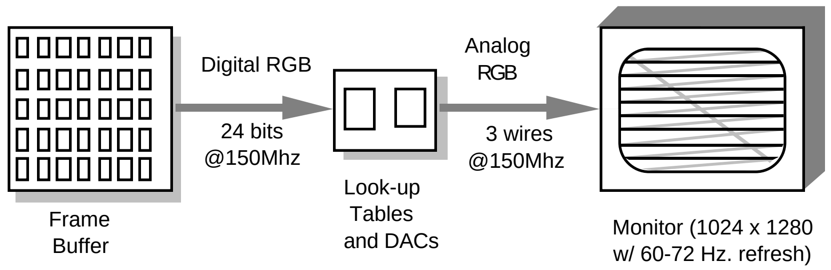


Figure 3.11: Conventional high-resolution video system

The frame buffers in most commercial rendering systems are built out of DRAMs or VRAMs. Single-ported DRAMs are generally used to store *z*-values and other pixel data not for display, while dual-ported VRAMs are used to store the color values. An easy way scan out *z* values is to store them in VRAMs in the same manner as color values. VRAM memory densities approach DRAM densities, and the same sequencing logic that drives the RGB scanout circuitry can drive the *z* scanout circuitry. A method for synchronizing the scanout circuitry on each of the boards is needed. A straightforward way to do this is to provide a global, synchronous scanout clock to each frame buffer board, and to connect this to the genlock circuit of each frame buffer.[1] Note that if rendering is to occur concurrently with video display, the *z*-buffer, in addition to the color buffer, must be double-buffered. This requires extra memory. These changes require modifications to the frame-buffer board, but the changes are largely confined to the video output portion of the system.

***Z*-buffer Compositors.** The compositors in the system merge two RGB/*z* data streams into one. Even though they are simple, they must run at very high speeds—80 to 140 MHz for a 1280x1024 monitor refreshed at 60 Hz. The compositor can be pipelined by sending *z*-values one clock cycle ahead of RGB values, as shown in Figure 3.12. The compositor shown here can be built using 100K ECL parts for approximately US \$110[2], plus board and connector costs.

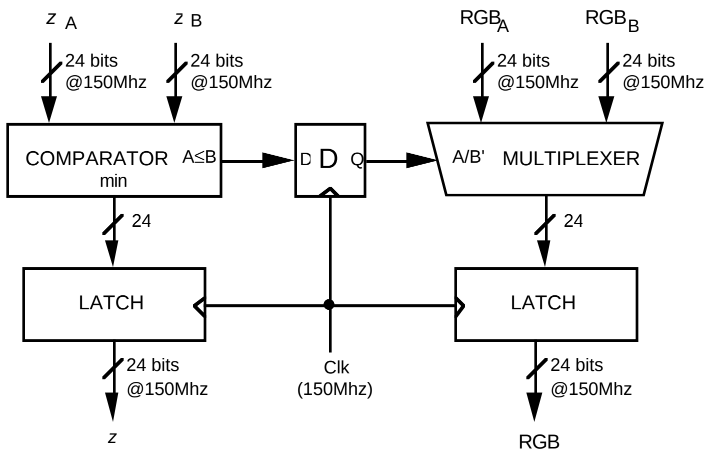


Figure 3.12: Block diagram of a high-speed ECL compositor.

Although *z*-buffer image-composition systems can achieve extremely high performance and are simple to

[1]High-speed synchronous clocks can be distributed in large systems using *salphasic* (constant-phase) clock technology demonstrated in Pixel-Planes 5 [CHI90].

[2]Price of six Fairchild F10016 and 12 Fairchild F100155 ECL parts (Hamilton-Avnet Electronics, Raleigh NC, prices quoted 8/30/88).

construct, they suffer from a major limitation—aliasing. We consider solutions to the aliasing problem in Chapter 4.

CHAPTER 4
SOLVING THE ALIASING PROBLEM

We now consider the problem of aliasing in pixel-priority architectures. Since high-performance systems must be able to generate high-quality, antialiased images, the aliasing problem must be solved to make image-composition architectures feasible. This chapter explores the main alternatives and the architectural families they lead to. Section 4.1 introduces two image-precision antialiasing approaches: supersampling and A-buffer algorithms. Section 4.2 describes the supersampling family of image-composition architectures. Section 4.3 describes the A-buffer family of image-composition architectures.

4.1 ANTIALIASING APPROACHES

We saw in Section 3.3 that simple antialiasing methods, such as alpha-blending and Duff's algorithm, are inadequate for pixel-priority image-composition systems, because they are not associative and do not preserve subpixel geometry. We now turn our attention to two general (but more expensive) methods: supersampling and A-buffer algorithms.

4.1.1 Supersampling

Supersampling is a general approach to antialiasing in which the image is sampled at higher-than-pixel resolution and filtered down to pixel resolution using a low-pass filter. It is a brute force algorithm, requiring a great deal of computation, but is the simplest and most general image-precision antialiasing algorithm known. Supersampling is a large and active research area within computer graphics. A full discussion of its subtleties is beyond the scope of this dissertation. [FOLE90] provides a good introduction to the area and pointers to the literature. This section provides a brief overview of supersampling as it pertains to image-composition architectures.

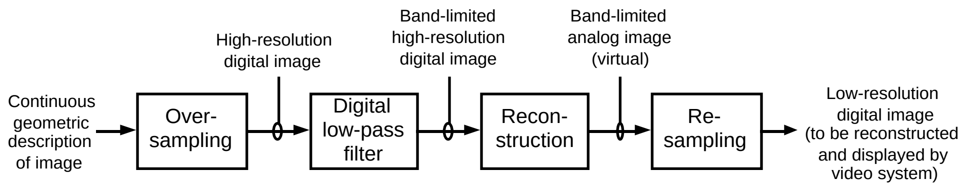

Continuous geometric description of image
Over-sampling
High-resolution digital image
Digital low-pass filter
Band-limited high-resolution digital image
Recon-struction
Band-limited analog image (virtual)
Re-sampling
Low-resolution digital image (to be reconstructed and displayed by video system)

Figure 4.1: Conceptual pipeline of operations performed during supersampling.

Supersampling has the four conceptual steps shown in Figure 4.1. First, the image is sampled at several times the pixel resolution. This results in a high-resolution digital image that can represent components of higher frequency than can a single-sample-per-pixel image. Next, a digital low-pass filter attenuates frequency components above the Nyquist frequency of the one-sample-per-pixel image. Third, a continuous image is reconstructed from the high-resolution digital image. Finally, the image is resampled with a single sample per pixel.

In an actual implementation, the third and fourth steps, reconstruction and resampling, are never performed explicitly. Instead, the image is only sampled at the locations needed to compute the pixel values in the low-resolution image. These samples are averaged together, using blending coefficients determined by the low-pass filter, to produce a single, antialiased color for each pixel.

Supersampling is a very general approach. Since it determines the visible surface and color for each sample point, it accurately antialiases images containing any type of primitive, shaded in any fashion—so long as enough samples are taken to represent the high-frequency components in the underlying image. (Since any image that contains lines or edges of polygons has components of infinite spatial frequency, this can seldom be done perfectly). Figure 4.2 shows a 160x128-pixel image of a space station sampled with 1 sample per pixel (no antialiasing), and supersampled with 5, 16, and 32 samples per pixel.

Figure 4.2: Space station scene with 1 sample per pixel (upper left), 5 samples per pixel (upper right), 16 samples per pixel (lower left), and 32 samples per pixel (lower right).

Unfortunately, supersampling an image with *k* samples per pixel requires rendering the image at *k* times the resolution or rendering it *k* times with slightly different coordinate offsets—a *k*-fold increase in computational cost over the unantialiased image. For good quality images, *k* is typically chosen to be 16 or greater. This is a burdensome computational expense for any system, particularly real-time ones. We can minimize *k* by cleverly choosing sampling patterns and blending coefficients. [MOLN91] contains a discussion of the issues involved. Nevertheless, *k* values of at least 5 appear necessary for tolerable antialiasing, no matter how samples and weights are chosen, and much higher *k* values would be preferable if they were less expensive.

[FUCH85] and [EYLE88] describe a supersampling technique called *successive refinement.* A crude image is presented to the user at full speed as long as the user manipulates the view or any rendering parameters. When the image is held stationary (or is changed less frequently than the update rate), further samples are computed and averaged in with the previous samples, producing a supersampled image in incremental fashion. This technique is useful in applications where positioning the dataset and analyzing can be done separately. It less useful in real-time applications, where high-quality moving images are required in every frame.

4.1.2 A-Buffer

The second general antialiasing approach is to vary adaptively the amount of information used to represent each pixel, using more information where spatial frequencies are high and less information where spatial frequencies are low. We will refer to algorithms of this type, conveniently, but somewhat incorrectly, as

A-buffer algorithms.[1] A-buffer algorithms address the main difficulties of supersampling: its high computational cost and its high bandwidth requirements. Their main disadvantages are added complexity and artifacts under certain circumstances.

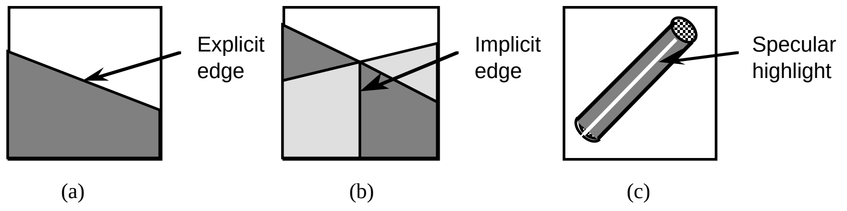


Figure 4.3: Sources of high-frequency components in a displayed image: (a) Explicit primitive edge, (b) implicit edge where primitives intersect, and (c) specular highlight.

Computer-generated images typically contain three sources of high-frequency components:

- **Explicit edges.** The zero-width boundaries of primitives have infinite spatial frequencies (Figure 4.3(a)).
- **Implicit edges.** Interpenetrating primitives cause implicit edges, which also have infinite spatial frequencies (Figure 4.3(b)).
- **High-frequency shading effects.** Certain shading effects, such as specular highlights, shadows, and textures can contain high-frequency components. For example, a Phong-shaded cylinder several pixels wide can have a specular highlight much narrower than a pixel (Figure 4.3(c)).

If we ignore texturing, which requires special treatment anyway (see Section 5.5.3), the main source of high frequency components in most images is polygon edges. Pixels near polygon edges need extra sampling; pixels in the interior of polygons can generally be sampled once per pixel. A-buffer algorithms take advantage of this by storing extra information for pixels that contain polygon edges, and simply storing color and *z* values for the others.

The potential savings are substantial. The three largest sample datasets in Appendix A contain between 120,000 to 350,000 polygons. In high-resolution (1280x1024) images of these datasets, fewer than 50% of the pixels contain contributions from two or more primitives. If we can take advantage of this, we can produce high quality images and substantially reduce the bandwidth requirements for the image-composition network.

The original A-buffer algorithm described by Carpenter in [CARP84] assumes that rendering is performed using a screen-sized image buffer. The entry for each pixel can contain either a single color and *z* value, or else a pointer to a linked list of partially-covering primitive surfaces. Initially, each pixel in the screen buffer is set to the color and *z* value of the background. As primitives are rasterized, they are diced into pixel-sized pieces called *fragments*. Fragments may completely cover a pixel, or may only partially cover it (if an edge of the polygon passes through the pixel, for example).

At each pixel affected by a polygon, a determination is made as to whether the fragment covers the pixel completely or partially. If it covers the pixel completely and is the closest surface so far, the fragment's color and *z* value replace the values stored in the screen buffer. If it covers the pixel only partially, a record is made containing the fragment's color, *z*, and partial-coverage information (generally an alpha value or a subpixel coverage bitmask). The entry in the screen buffer is changed to a pointer, which points to this newly created record. This record contains a pointer to another such record, containing the color and *z* value of the background (see Figure 4.4). As additional primitives are rasterized, they are merged into the screen buffer in this fashion.

[1]The name A-buffer was coined by Carpenter in [CARP84] to refer to a particular algorithm of this type. Since that time, it has informally taken on a wider meaning. We will use the term A-buffer to refer to all antialiasing algorithms that use a variable-length list of fragments to represent each pixel.

In the original A-buffer paper, there is no restriction on the length of the linked lists of fragments. In pathological cases, where many primitives partially cover a single pixel, the linked lists can become arbitrarily long. However, fragments that completely cover a pixel obscure all of the fragments behind them and truncate the list. This tends to limit the length that lists actually achieve.

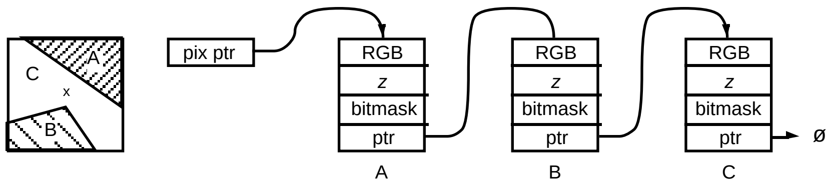


Figure 4.4: Magnified view of pixel and linked list of fragments for the three surfaces A, B, and C visible at the pixel.

The original A-buffer algorithm was used to produce high-quality antialiased images in the initial Reyes rendering system [CARP84]. This system was designed to run on virtual-memory uniprocessors, such as the VAX 11/780, a much more flexible software environment than parallel renderers in an image-composition system.

A-buffer algorithms can be adapted to image-composition architectures by feeding the fragment lists into the image-composition network, where they are composited and eventually resolved. This approach has been proposed by several researchers [WEIN81, SCHN88], but (to the author's knowledge) has never been implemented. The approach has a number of problems:

- It requires sophisticated processing at the compositors to interpret the variable-format pixel streams and composite them correctly.
- It requires transmitting extra information for complex pixels. Even though this is normally less than the *k*-fold increase with supersampling (in pathological cases it can be more), it exacerbates the bandwidth problem to some degree.
- It produces artifacts that are not found in supersampled images

Fortunately, most of these problems can be solved, or at least mitigated, and the approach offers a number of advantages, including high-quality images with modest composition-network bandwidth and a natural way to implement transparency. We will discuss the A-buffer family of image-composition architectures in detail in Section 4.3.

4.2 SUPERSAMPLING ARCHITECTURES

As explained in Section 3.3.1, supersampling is a brute-force algorithm that requires a great deal of computation, but it is the simplest and most general image-precision antialiasing algorithm known. It is the basis for the supersampling family of image-composition architectures, whose properties we will discuss in the following sections.

4.2.1 The Factor of *k*

Brute-force algorithms have brute-force costs: to compute *k* samples per pixel, we must do *k* times as much work as for an unantialiased image. Brute-force algorithms have advantages, however. For example, the *z*-buffer algorithm was at one time thought too extravagant in memory usage to be practical [SUTH74]. Because of its simplicity and the decreasing cost of memory, nearly all graphics systems now use the *z*-buffer algorithm. Perhaps supersampling antialiasing has similar advantages.

If we wish to supersample without sacrificing frame rate, we must overcome two challenges:

- We must compute *k* times as many samples per second as in an unantialiased image.

- We must increase the bandwidth of the image composition network by a factor of *k*, since we must composite subpixel samples before resampling.

The first point has implications for the renderers, the second for both the renderers and the image-composition network.

4.2.2 Computing Additional Samples

The computational power required of each renderer is proportional roughly to:

# of primitives • # of subpixel samples • frame rate

If we wish to increase the number of samples by a factor of *k*, we either increase the renderer's power by a factor of *k*, reduce the number of primitives, or reduce the frame rate. If we knew how to increase the renderers' performance easily, we would not be worried about image composition. We are left, then, with trading the number of primitives or frame rate for the extra samples.

Fortunately, things are not be as bad as they seem. There is considerable coherence between the multiple, slightly offset, images required for supersampling. Depending on the renderer design, some calculations may be factored out and performed just once for all of the subsamples. For example, vertex normal vectors need to be calculated only once. Taking advantage of such coherence can reduce the computational expense below the nominal factor of *k*.

4.2.3 Increasing the Bandwidth

The most serious obstacle to supersampling is the increased pixel bandwidth required. The numbers are enormous and affect the renderer and every aspect of the image-composition network. For example, consider a system that updates a 1280x1024 screen with 5 samples per pixel at 30 Hz. The composition bandwidth is:

1280x1024 pixels • 5 samples/pixel • 8 bytes/sample • 30 frames/second

= 1.57 Gbyte/sec

No individual factor is outlandish, but the product certainly appears to be. To achieve this bandwidth requires either:

- 1 wire at 12.6 GHz.
- 32 wires at 393 MHz.
- 128 wires at 98 MHz.
- 512 wires at 24.5 MHz.

This bandwidth requirement is not for a single, localized datapath within a single board; rather, it is necessary at the output port of the renderer and at every link in the image-composition network, many of which span board boundaries.

Fortunately, this may not be a problem for the renderers. *Z*-buffer renderers have huge bandwidth requirements at the *input* ports of their frame buffers (see [MOLN90]). Current commercial systems generally use 16-40 banks of frame-buffer memory to achieve this bandwidth [AKEL88, POTM89]. If they are built using VRAMs, as many are, even greater bandwidth is available at the output port. For example, a 20-bank VRAM memory system with 20 MHz serial ports provides 20 • 20 Mword/sec = 1.6 Gbyte/sec at its output port. This satisfies the bandwidth requirement for the renderer output port.

The image-composition network, however, bears the brunt of the factor of *k*. It must span several boards, and has many high-speed connections, which are extremely expensive in board area and dollars.

If real-time update rates are not required for antialiased images, there is an escape route: we can use successive refinement. Successive refinement requires no hardware modifications for the renderers or image-composition network. It does, however, require a final accumulator frame buffer capable of blending new sample values with previous ones [HAEB90]. Accumulator frame buffers are relatively easy to implement. Such a system can compute high quality images, but with a penalty in speed.

If we want high frame rates we have no choice but to increase the image-composition network's bandwidth *k*-fold, or to choose a different antialiasing algorithm. We initially believed that the composition-network bandwidth would dominate the other costs of the system, making it infeasible unless the system contained very high-performance renderers. Consequently, we began investigating lower-bandwidth alternatives, such as the A-buffer method described in the following section. After considering the properties of each method, the tradeoffs became less certain; much can be said for the feasibility of implementing simple operations in parallel, a strongpoint of the supersampling approach. The prototype system design described in Part II of this dissertation uses the supersampling method.

4.3 A-BUFFER ARCHITECTURES

A-buffer antialiasing is the second general approach for antialiasing in image-composition architectures. Its primary advantage over supersampling is that pixels contain extra information only where needed. Each pixel is represented by a variable number of partially-visible primitive fragments; the number of fragments depends on the number of primitives that are potentially visible at the pixel. Since most pixels in most images can be represented by one or two fragments, we might expect that less information per pixel is needed in an A-buffer system than in a supersampling system.

Unfortunately, A-buffer algorithms are complex and produce several types of undesirable artifacts. These artifacts stem from the fact that depth and surface values are only computed at the center of each pixel, and this is insufficient when edges pass through a pixel or surfaces interpenetrate.

In this section we will take a closer look at the problems and opportunities of the A-buffer approach. Section 4.3.1 describes how A-buffer algorithms can be implemented on an image-composition system. Section 4.3.2 describes enhancements to the simple A-buffer algorithm to mitigate artifacts. Sections 4.3.3 to 4.3.5 discuss possible implementations for various system components.

4.3.1 Applying the A-Buffer to Image Composition

The classic A-buffer algorithm uses a screen buffer to maintain a sorted list of partially visible primitives at each pixel. In an image-composition architecture, each renderer computes its own image. The image-composition network must combine the partial-coverage information from the multiple renderers into a single pixel stream.

In a simple A-buffer image-composition system, fragments contain the following information:

- **RGB—**the color value for the fragment (obtained by sampling the linear equations for R, G, and B at the center of each pixel).
- ***z*** —the depth value for the fragment (obtained by sampling the linear equation for *z* at the center of the pixel).
- **Coverage bitmask**—an *N*-bit value which contains a coverage bit for each subsample within the pixel (each bit is set if the corresponding sample point lies inside the fragment).

To render an image, each renderer must compute, for each pixel in the image, an ordered list of fragments resulting from the primitives in its database partition. This list contains the fragments that are potentially visible at that pixel in increasing *z* order, ending with the nearest fragment that completely covers the pixel. We call the list of fragments for each pixel a *pixel packet*. The renderer places pixel packets onto the image-composition network in scan-line order.

The compositors in an A-buffer system have a more demanding task than the compositors we have discussed before. They must parse the pixel streams coming from each input and interleave fragments within a pixel so that the pixel stream has the same format at the compositor's output as it does at its two inputs. Since fragments from one compositor input can cover fragments from the other input, the compositors must be smart enough to truncate the fragment list when the pixel becomes completely covered. The list of fragments for each pixel, therefore, can increase if partially-visible fragments are added to the front of the list, or shrink if completely covering fragments obscure fragments behind them.

A-buffer systems require an extra module not found in supersampling systems: a *pixel assembler* to convert the multiple fragments per pixel into RGB values. It does this by blending the colors of each fragment within a pixel packet based on the coverage information in the fragments. The result is a single RGB value (assembled pixel) for each pixel. Since the pixel stream entering the pixel assembler contains coverage information for all of the primitives in the scene, it is the point of highest bandwidth in the system.

The stream of assembled pixels (RGB values) emerging from the pixel assembler can be stored in a double-buffered frame buffer, to allow the screen to be refreshed at any rate desired.

4.3.2 Minimizing Artifacts

The standard A-buffer algorithm leads to several types of artifacts:

1) Visibility errors.
2) Shading errors.
3) Aliasing of high-frequency shading effects.

All of these artifacts stem from sampling interpolated parameters at pixel centers only. This can result in erroneous *z* or color values when the sample point lies outside the polygon and can result in aliasing if the sample point misses a high-frequency shading effect. We now examine each of these artifacts more closely and suggest enhancements to the standard algorithm to minimize these artifacts.

Visibility errors. The visible fragment at each pixel is determined by comparing the *z* values of each fragment in a pixel packet. *Z* values are computed by sampling the linear expression for *z* (by forward differences or some other means) at the pixel center. Since fragments are constructed for polygons that cover pixels only partially, the sample point for a fragment may lie outside the corresponding polygon, and may vary significantly from the *z* value at the closest polygon edge. When polygons are close to edge-on, the *z* gradient may be very high, and the *z* value at the sample point can be larger or smaller than the maximum or minimum *z* value of the entire polygon.

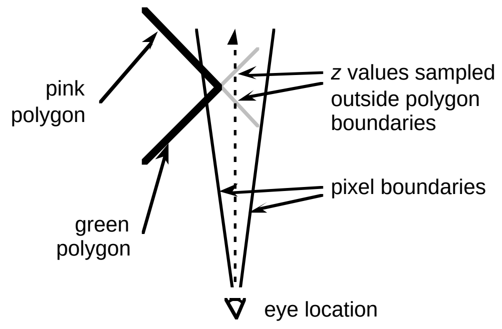


Figure 4.5: Visibility errors caused by sampling *z* outside of polygon boundaries (scene is viewed from above).

This effect can lead to visibility errors, as shown in Figures 4.5 and 4.6. Figure 4.5 shows two polygons seen from above. The front face of the front polygon is colored olive green. The front face of the back polygon is colored pink. Because of the errors in *z*, pixels along the intersection between these two polygons appear to have the pink surface in front. Figure 4.6 shows a rendered image of the same scene that exhibits these artifacts.

Although these artifacts are noticeable in static images, they twinkle from frame to frame in moving images, making them unacceptable in real-time systems. The problem can be addressed in a number of ways: One solution is to calculate the *z* value at the nearest point on the polygon edge. This might produce pleasing results, but would greatly complicate the rasterization algorithm, the inner loop of the rendering process. The pixel could be sampled at multiple locations, but then we would not need the list of fragments; the algorithm would reduce to supersampling, discussed in Section 4.2.

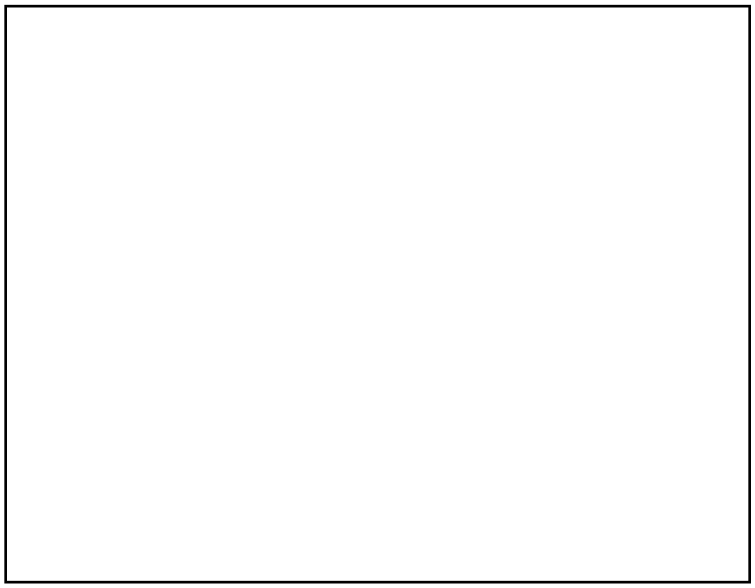


Figure 4.6: A-buffer-rendered image of the two-polygon scene in Figure 4.5. Several pixels along the intersection between polygons are colored pink, whereas only green should be visible.

Another approach is to calculate *z* values at several points within the pixel, but to defer these calculations until after image composition. This can be done by encoding *z* slope information in the pixel packet, rather than the *z* values for multiple sample points. In this approach, subpixel *z* values do not have to be calculated for fragments that are eliminated in the image-composition process, and the increase in image-composition bandwidth is modest.

The slope information required to do this is the change in *z* with respect to screen-space *x* (*dz*/*dx*) and the change in *z* with respect to screen-space *y* (*dz*/*dy*). Unfortunately, *z* values for a single surface can vary substantially within a pixel in nearly edge-on polygons. This means that *dz*/*dx* and *dz*/*dy* must be able to represent a substantial fraction of the full *z* range. On the other hand, slope information does not need to be very precise; in most cases intersecting surfaces will have slopes that differ significantly.

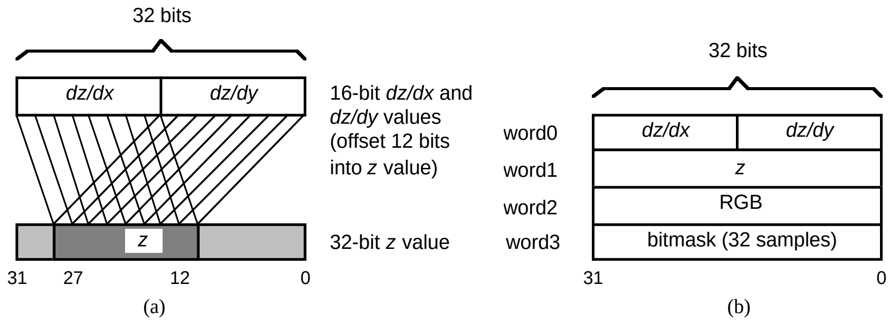


Figure 4.7: (a) Compact representation of *dz*/*dx* and *dz*/*dy*. (b) Data format for 4-word fragment.

In our software simulator, we have obtained reasonable results using 16-bit slope values with 32-bit *z* values. The slope values are offset 12-bits into the 32-bit *z* value, as shown in Figure 4.7. This means that a one-pixel displacement in *x* can alter *z* by as much as 1/64th of the 32-bit *z* range. This is a compact representation for *dz*/*dx* and *dz*/*dy*, since both slope values fit into a single 32-bit single word. This only increases the data requirements for a fragment by 1 word (from 3 words to 4).

Is this a worthwhile addition to an A-buffer composition system? Consider the costs and benefits. There are three added costs: 1) extra bandwidth in the image-composition network, 2) extra work to calculate slope values, and 3) extra work to calculate subsample *z* values during pixel assembly.

The added bandwidth is significant, but probably tolerable (4 words per fragment instead of 3). Slope values are extremely easy to compute—they already exist! These are the same slope values used to advance *z* from pixel to pixel or from scan line to scan line in virtually any renderer. The only complications are format conversion and clamping, since slopes may be represented as floating-point values in the renderer. These are both relatively simple operations, and can be implemented in hardware, if necessary. Calculating *z* values for each subpixel sample during pixel assembly is another matter. This substantially complicates the task of pixel assembly. But there are two pieces of good news: it only has to be done for potentially visible fragments (a large savings), and it is only needed at only one place in the system. Hence, it is a fixed cost for the system and does not affect its scalability.

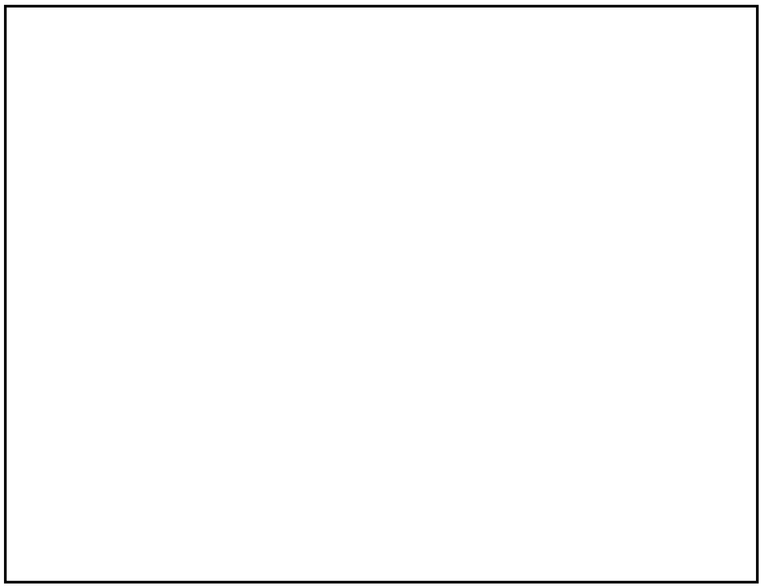


Figure 4.8: Two-polygon scene rendered using *z* slope information.

Figure 4.8 shows the same scene that was rendered in Figure 4.7, but this time using *z* slope information. The artifacts disappear completely. This approach seems to be a reasonable compromise toward rendering high-quality images without overwhelming the image-composition process.

Shading errors. Another type of artifact, also caused by sampling only once per pixel, is errors in shading. When fragments do not cover a pixel completely, interpolated values, such as R, G, and B in Gouraud shading or normal-vector components in Phong shading, can be sampled outside the polygon's boundary, resulting in errors. This is especially severe when polygons are nearly edge-on. For example, Gouraud-shaded polygons with high color gradients can have color values that overflow the minimum or maximum allowable values and wrap around.

Figure 4.9 demonstrates this phenomenon. The nearly edge-on polygons at the top and bottom of the airplane's fuselage have high color gradients. Color values at some boundary pixels overflow the 0-255 color range, wrap around, and are grossly incorrect. Errors in surface-normal vectors used for Phong shading are less pronounced, since they are renormalized after interpolation. They still can result in incorrect Phong highlights.

Figure 4.9: Gouraud-shaded image of an X-29 aircraft with color-wraparound artifacts

These artifacts could be eliminated the same way we eliminated errors in *z*: by computing slopes for each color parameter, storing them with each fragment, and evaluating them for each subpixel sample during pixel assembly. Unfortunately, there are three (or more) color or normal channels, in contrast to one *z* channel. Since color and normal values require fewer bits of precision than *z* values, the three pairs of slopes could be stored more compactly than *z* slopes. Still, handling multiple channels would be expensive.

Figure 4.10: Scene in Figure 4.9 with color clamping.

A simpler, cruder approach is to clamp interpolated parameters for each fragment to the minimum and maximum values of the polygon. These values are readily available; they are the minimum and maximum values of the parameter at each of the polygon vertices. They can be computed once for an entire polygon during geometry processing and made available to the rasterizer. This method has the additional advantage of being completely independent of the image-composition network.

Figure 4.10 shows the same scene as in Figure 4.9, but with clamped color values. For all of the images we have tested, this simple approach appears to do an acceptable job. To see if it produced any distracting time-dependent artifacts, we incorporated the method into a prototype real-time flight simulator. The artifacts were mitigated to the point of not being distracting—or even noticeable—without scrutinizing the image. Both this approach, and the more general approach using slopes, appear feasible.

High-frequency shading effects. A third type of artifact is caused by undersampling high-frequency shading effects, such as Phong highlights. For example, consider a front-lit metallic cylinder that is only two or three pixels in diameter. This cylinder will have very sharply defined highlights that are much narrower than the width of a pixel. The spatial frequency of the highlight is higher than half the A-buffer's one-sample-per-pixel sampling frequency (the Nyquist frequency for a one-sample-per-pixel image), and, therefore, will alias. Figure 4.11 is an example of such an image.

Figure 4.11: A-buffer rendering of highly specular cylinders (640x512-pixel resolution with 32 samples in the fragment bitmask).

Figure 4.12 shows the same scene rendered using supersampling (with the same resolution and the same number of samples per pixel). The supersampled image contains less aliasing because each subpixel sample is shaded independently, whereas a single color value is used for all the subpixel samples in the A-buffer algorithm.

Unfortunately, there is no easy way to extend the A-buffer algorithm to avoid these artifacts without reducing it to supersampling. One possible solution is to supersample the color values in the renderer and use the blended color during image composition. This does not reduce the amount of calculation over supersampling on the renderers, but does reduce the image-composition network bandwidth. This approach still results in errors when pixels are partially obscured, however, because some obscured sample points may contribute to the final pixel color, but it is certainly an improvement over the standard A-buffer algorithm. If we wish to avoid these artifacts entirely, the only alternative appears to be supersampling throughout the rendering process.

Figure 4.12: Supersampled rendering of same scene (also 640x512-pixel resolution with 32 samples per pixel in same locations as A-buffer).

We rendered each of the sample databases in Appendix A using both supersampling and an enhanced A-buffer algorithm using *z* slopes and color clamping (but not supersampled colors). Very few artifacts were apparent in the A-buffer images. A-buffer rendering appears to be a feasible method for generating high-quality images as long as highly specular objects can be avoided. We will compare the two approaches further in Chapter 5.

4.3.3 A-Buffer Renderers

We now focus on implementation issues for A-buffer architectures. An A-buffer image-composition system requires scan-line A-buffer renderers, special compositors, and a pixel assembler. All of these can conceivably be built, but they present challenging design problems. In this section and the following sections we briefly discuss some of the issues involved in implementing an A-buffer image-composition system.

A-buffer renderers must be able to generate a sufficient number of fragments to render complex, high-resolution images in real time. A database partition of 3,500 triangles is required to render 100,000 polygons per second at 30 Hz. Based on the statistics from the sample images in Appendix A, high-resolution images containing 3,500 to 10,000 triangles average approximately 1.04 to 1.16 fragments per sample. Renderers of this size must generate, therefore, approximately 1.16 • 1280 • 1024 • 30 = 46 million fragments per second.

A-buffer renderers could be built using a large image-buffer (as in the original A-buffer paper) or using a scan-line approach. The linked lists required to store the multiple fragments per pixel makes the image-buffer approach difficult to implement. Also, since scan-line renderers produce pixels in the order they are needed by the image-composition network, scan-line rendering appears to be a natural fit here.

Most scan-line renderers implement the following three pipelined operations [MOLN90]:

- **Transformation and bin sorting.** Primitives are transformed into screen coordinates and stored in a list, based on the first scan-line in which they are active.
- **Active segment sorting.** A list of active polygons is built for each succeeding scan line (only incremental changes are needed between scan lines). The intersection of each polygon with the

current scan line is called a *segment*.. The list is sorted by the segments' left endpoints.

- **Pixel generation.** A list of active segments is built for each pixel in the scan-line (only incremental changes are needed between pixels). The active segments are used to determine the values for each pixel.

Some of the earliest real-time raster systems used scan-line rendering [WATK70, SCHA83], though each polygon had a constant color (flat shading) and they did not perform antialiasing. These systems were extremely parsimonious in terms of numbers of parts, however. Later scan-line renderers, that incorporated Gouraud shading and antialiasing, are much more complex [BUNK89, NIIM84]. An A-buffer renderer requires even more sophisticated visibility and shading calculations: it must calculate RGB, z, dz/dx and dz/dy, and bitmasks for each fragment.

RGB and z are linearly interpolated parameters that can be calculated easily using forward differences. dz/dx and dz/dy are constants for each primitive. Bitmasks are more difficult to calculate. They could be calculated by supersampling the primitive geometry at each pixel, but this would defeat many of the benefits of the A-buffer approach. A simpler way is to use look-up-tables based on slope and intercept parameters, as described in [ABRA85]. This allows a large number (up to 32) samples to be determined in constant time.

Computing multiple fragments per pixel is straightforward: a fragment is created for each segment active at the current pixel. Conventional scan-line renderers only need to determine which segment has the smallest z value; A-buffer renderers must sort the fragments into ascending z order, an extra complication. Fortunately, this task can be moved to the input port of the compositor (see Section 4.3.4).

Buffering between the fragment generator and the image-composition network is also needed. Since different renderers may have different distributions of primitives, and the image-composition network handles the same pixels simultaneously from all of the renderers (at least, close to simultaneously), buffering is required to dissociate the two operations. We will see in Section 5.4, that this memory must be able to hold a substantial fraction of the pixel packets for an entire image.

4.3.4 A-Buffer Compositors

The second critical component in the architecture is the compositor. A-buffer compositors must perform non-trivial compositing and do so at extremely high speeds. To composite two A-buffer pixel streams requires comparing fragments from each stream one-by-one. Fragments must be interleaved and truncated when further fragments can no longer be visible.

Determining pixel coverage. Various criteria can be used to determine when a pixel is completely covered. Some require more computation, but result in minimum-length pixel packets. Others take less computation, but result in pixel packets that are unnecessarily long.

A simple, but inefficient method is to interleave pixel fragments and to truncate the list only when a fragment is encountered that completely covers the pixel. This requires only a logical *and* of the coverage bits, but is inefficient, since adjacent polygons whose seam crosses a polygon are not recognized as covering the polygon. This can be a common situation, since primitives of a tiled surface may be placed on separate renderers for load balancing, as explained in Section 5.3.

Fragment list lengths can be minimized by accumulating a coverage bitmask for the pixel. This can be done by logically *or*ing the coverage bitmask of each succeeding fragment with a cumulative coverage bitmask. When all of the bits have are set, the pixel is considered to be covered. This is computationally inexpensive and correctly handles the case of a pixel covered by multiple abutting polygons.

A complication with either approach is the fact that fragments span a range in z, rather than having a single z value. If we do not treat these correctly, we can create artifacts and undo the benefits of adding dz/dx and dz/dy to z values. Computing actual z values at each subsample point is far too much work to be performed in each compositor. A more tractable approach is to compute z values at the corners of the pixel for each fragment (the corners represent the minimum and maximum x and y locations of any subpixel sample). We can conclude that one fragment is in front of another if its z value is less than the other fragment's z value at each corner of the pixel. We should only truncate fragments that lie entirely behind the covering fragment (or fragments).

Specialized compositor inputs. If the image-composition network is structured as a pipeline, rather than a tree, the compositor input ports can be specialized: one can handle the high bandwidth from the previous compositor, and the other can be tailored to interface with a renderer. One specialization that is possible is to sort fragments from the renderer in the compositor itself.

This can be done by incorporating a *rebound sorter* into the compositor's input port. A rebound sorter is an array of processing elements that read in unsorted data and output the data in sorted order [KUNG88]. A rebound sorter has a finite depth *N*, determined by the number of processing elements in the sorter; *N* must be relatively small to be practical. These restrictions are acceptable in an image-composition system.

Fragments would be processed by the compositor as follows: the renderer would compute fragments for the current pixel and load them into the rebound sorter in the compositor in any order. Up to *N* fragments would be read from the sorter (fewer if fewer than *N* fragments were loaded in the first place). The fragments would emerge in ascending *z* order and then be merged into the pixel stream arriving at the compositor's other input. The sorter would be cleared between pixels. If a pixel contains more than *N* fragments, the extras (the ones with largest *z* values), would drop off the end of the sorter and be discarded. This can be prevented by making *N* greater than or equal to the number of subpixel samples—the maximum number of fragments per pixel. Even if *N* is less than the number of samples, *N* values greater than 8 or 16 should produce only minor errors in the most complex pixels.

Buffering between compositors. Since compositors can add fragments to the pixel stream and can truncate fragments that become covered, the amount of information for each pixel can increase or decrease as the pixel travels through the composition network. To even out these differences, some amount of buffering is required between compositors. Before designing an actual system, one must determine the amount of buffering required for the types of images that will be rendered. One must be able to guarantee that for the set of images likely to be encountered, serious bottlenecks do not arise in the composition network. Simulation or queueing analysis is required to answer this question definitively.

Bandwidth requirements. The compositor must perform all of the functions above and do so at extremely high speeds. For example, the pipes database from Appendix A, rendered at 1280x1024 resolution contains an average of 1.68 fragments per pixel—a total of 2.2 million fragments. Updating this image at 30 Hz requires that at least one compositor in the image-composition network process 66 million fragments per second, a bandwidth of 1.1 Gbyte/second. The only practical way to achieve this is using high-speed circuitry with several parallel paths.

4.3.5 Pixel Assembler

A pixel assembler is required to combine the multiple fragments of each pixel packet into a single RGB value for the pixel. This involves compositing fragments in front-to-back order, taking into account how much of the pixel each new fragment covers.

The addition of *dz*/*dx* and *dz*/*dy* to pixel fragments complicates the pixel assembly algorithm: the ordering of fragments may not be the same for all subpixel samples. To solve this in a general way requires computing the closest fragment for each subpixel sample. To achieve the required speed, this must be done in parallel for each sample. The final step is to blend the colors from each of the subpixel samples into a color value for the entire pixel. If importance sampling is used, this can be a simple average operation [MOLN91]. If samples have different weights, they must be multiplied by appropriate blend coefficients before adding.

A-buffer image-composition architectures can render transparent objects in a straightforward way if the pixel assembler can alpha-blend subpixel samples, instead of simply finding the closest one. We consider the extensions needed to support transparency in Section 5.5.2.

CHAPTER 5

ANALYSIS OF IMAGE-COMPOSITION APPROACHES

In Chapters 3 and 4 we introduced image-composition architectures and described two main methods of antialiasing. In this chapter we will discuss other factors that influence the performance, flexibility, and cost of image-composition systems, such as load balancing, latency, and support for advanced rendering algorithms. We will compare the advantages and disadvantages of different approaches and present a taxonomy of image-composition architectures.

Section 5.1 discusses trade-offs between a tree-structured and pipelined image-composition network. Sections 5.2 through 5.4 discuss performance issues such as latency and load-balancing. Section 5.5 discusses how advanced rendering algorithms, such as transparency and texturing, map onto image-composition architectures. Section 5.6 presents a taxonomy of image-composition architectures. Section 5.7 discusses the economics of image-composition compared with other image-generation approaches.

5.1 TREE VS. PIPELINE

So far we have paid little attention to the topology of the image-composition network—the way compositors and renderers are connected. Since compositors combine two input pixel streams into one output stream, the composition network has the structure of a binary tree. Binary trees can have many forms, as shown in Figure 5.1.

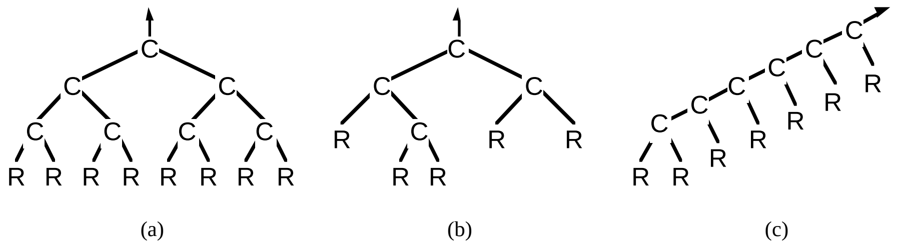


Figure 5.1: Classes of binary trees: (a) balanced tree, (b) mixed tree, (c) left-heavy tree (pipeline).

The two extreme cases, balanced trees, and left-heavy (or right-heavy) trees are the most interesting for an image-composition network. Each has advantages and disadvantages. Mixed trees have most of the disadvantages of both extremes with few of the advantages. Figure 5.2 summarizes the properties of the three approaches.

Property	Structure of composition network		
	Balanced tree	Mixed tree	Pipeline
Structure	Regular	Irregular	Linear
Connections between boards	Varying-length, non-local	Variable-length, non-local	Single-length, local
Expendability	Expansion difficult	Expansion difficult	Expandable
Latency	$\log_2$ (# renderers) • compositor latency	Intermediate	# renderers • compositor latency
Input port specialization	Must be identical	Must be identical	Specialization possible
Comp. network bandwidth (A-buffer systems)	Balanced on average	Intermediate	Increases toward end
Pixel synchronization	Synchronous on each renderer	Complicated	Staggered on each renderer

Figure 5.2: Properties of different composition-network topologies.

5.1.1 Balanced Tree

In a balanced tree, the pixel stream from every renderer flows through the same number of compositors. This means that, for A-buffer systems, the bandwidths in different parts of the image-composition network are relatively balanced and that renderers can operate synchronously, each computing the same pixel at the same time. The main disadvantage of balanced trees, however, is their complex structure. Unless the physical hardware can be arranged in a tree, the connections between compositors are complicated, and different-length wires are required to connect different parts of the system. Also, the number and types of connections depend on the number of renderers in the system, which makes it difficult to expand the system.

5.1.2 Pipeline

In a pipeline network (left-heavy or right-heavy tree), the layout problem is trivial: each renderer is paired with a compositor and the compositors are connected to the preceding and succeeding compositors, forming a pipeline. The pixel stream begins at the upstream renderer and proceeds downstream to the end of the pipeline. Additional renderers can be added by extending the pipeline, but the type of connection for each compositor remains the same. Another advantage is that compositor input ports may be distinguished. Since the right input port, for example, always connects to a renderer, it can accept data in a format suited to the renderer.

The simple structure and easy expandability of pipelines are tremendous advantages. Pipelines have disadvantages, however: The amount of information carried in the pipeline increases toward the end of the pipeline (this is only an issue for A-buffer architectures). Also, renderers in a pipelined network must calculate pixels in a staggered fashion, so that a renderer finishes a pixel as the corresponding pixel arrives at its compositor.

Another minor difference between the two approaches is latency, the length of time required for a pixel to pass through the network. Let n be the number of renderers in the system and t_c be the time required for a compositor to process a single pixel. A balanced tree has a latency of $t_c \bullet \log_2 n$, since a given pixel passes through $\log_2 n$ compositors. A pipeline network, therefore, has a latency of $t_c \bullet n$, much larger than the latency of a balanced-tree network. However, since t_c is very small (« 1 µsec), even the longer latency of a pipeline is negligible compared to a frame time. For all practical purposes, therefore, the latency of the image-composition network is negligible, and is not a serious disadvantage for pipelined networks.

The overwhelming consideration for choosing a composition-network topology is the flexibility and simplicity of the layout. Here pipelines are the clear winner. The additional advantage of allowing one port to be distinguished lets us divide function between the renderer and compositor as we wish. We use this to advantage in the prototype system described in Part II. *We will assume that image-composition networks have a pipeline topology throughout the rest of this dissertation*.

5.2 FRAME RATE AND LATENCY

Three performance issues are crucial for a graphics system:

- *Throughput*—the number of primitives that can be rendered per second.
- *Frame time*—the time between successive frames.
- *Latency*—the time required to compute a single frame.

Image-composition architectures have the property that throughput (primitive rendering performance) can be increased arbitrarily by adding renderers. The minimum frame time is determined by the image-composition network (load balancing plays a part here; we will discuss it in Sections 5.3 and 5.4). Latency is determined by the rendering algorithm. The two rendering approaches we have considered, *z*-buffer and scan-line A-buffer, incur latency in different areas. We now consider ways to reduce the latency on each.

5.2.1 Latency in a Simple *Z*-Buffer System

Consider a simple image-composition system that uses *z*-buffer renderers and does not perform antialiasing. Figure 5.3 shows a time line for the rendering process. First, control inputs (joysticks or other input devices) are sampled. Next, primitives are transformed and rasterized into a screen-sized image buffer. When all the primitives have been rendered, the images from each renderer are composited and stored in a frame buffer.

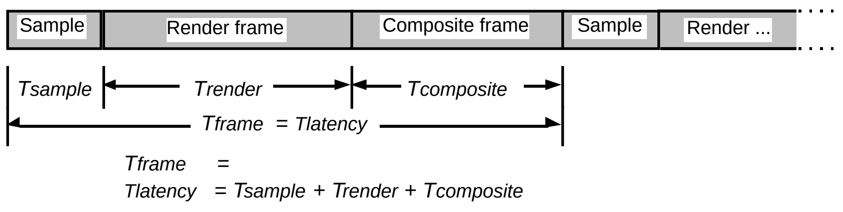
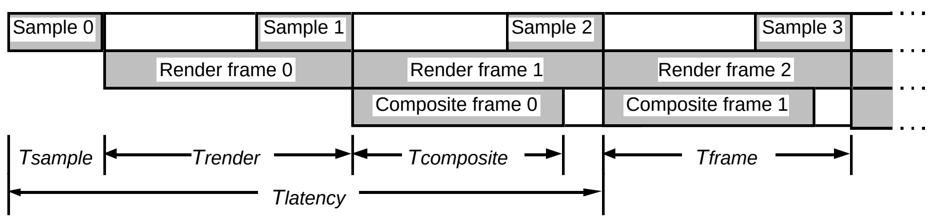


Figure 5.3: Rendering time line for a *z*-buffer image-composition system[1].

Input sampling, rendering, and compositing, are generally performed in separate hardware units. Therefore, each unit is busy only a fraction of the frame time. We can increase throughput and reduce the frame time by pipelining the three operations, as shown in Figure 5.4. Now the frame time is reduced, but the latency is unchanged. This is a general principle: pipelining increases a system's throughput, but not its latency.

[1]To be precise, we should include the effect of the frame buffer too, since we cannot view a new image until it appears on the CRT screen. When a new image has been loaded into the frame buffer, it must wait for the next vertical retrace to avoid tearing during the frame. This adds between 0 and *Trefresh* to the latency (an average of *Trefresh*/2), and adds variability to the frame time. This effect is the same regardless of the rendering or image-composition method, so we will ignore it in this discussion. Note that if *Tframe* < *Trefresh*, we can avoid this added delay by updating the frame buffer directly. This requires a high and very predictable frame rate, since the CRT *must* be refreshed every 1/74 to 1/30 of a second.



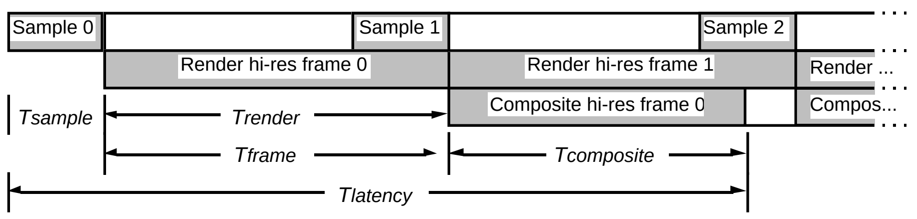
$$Tframe = \max(Tsample, Trender, Tcomposite)$$
$$Tlatency = Tsample + Trender + Tcomposite$$

Figure 5.4: Rendering time line for pipelined *z*-buffer system.

How can we reduce the system's latency? Aside from speeding up computations in the rendering pipeline, the elixir for all evils, we have one other alternative: divide the task into small chunks and pipeline these, rather than the overall task. There is no obvious way to do this in a simple *z*-buffer system, but we can use this approach when we add supersampling.

5.2.2 Latency in a Supersampling System

Consider a *z*-buffer system that antialiases using supersampling. Figure 5.5 shows a naive way that we might structure the rendering process. We could compute all the subsamples for each pixel in the whole screen, then composite these together to obtain a final supersampled image.



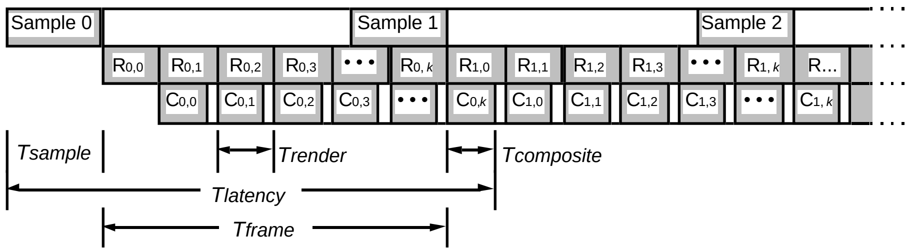
$$Tframe = \max(Tsample, Trender, Tcomposite)$$
$$Tlatency = Tsample + Trender + Tcomposite$$

Figure 5.5: Time line for naive supersampling image-composition system.

This presents two problems: First, since we cannot start compositing until the entire image is rendered, each renderer must be able to buffer an entire image, containing color and *z*, at subpixel resolution. This requires an exorbitant amount of storage. Second, the latency is high, since rendering and compositing cannot be overlapped for a single frame.

There is a much better method. Rather than computing the image at high-resolution, we can compute the image multiple times at low (single-sample-per-pixel) resolution, composite these independently, and merge them after the composition network. Figure 5.6 shows the time line for this second approach.

The frame time is unchanged, but the latency is much lower, since we have split the computational task into small parts that can be pipelined. Also, the storage requirements are greatly reduced. The only disadvantage to this approach is that the frame buffer must store the image and blend in successive samples as they are generated. Such *accumulator frame buffers* have been incorporated into existing commercial systems [APGA88, HAEB90]. The reduced latency and reduced storage requirements make this the preferred approach for supersampling image-composition systems.

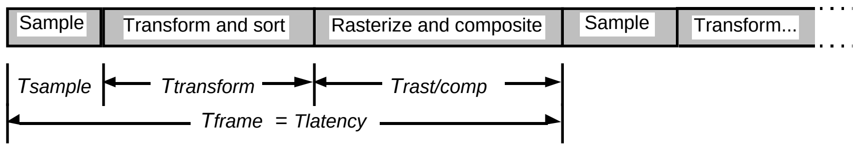


$$Tframe = \max(Tsample,\ k \bullet Trender,\ k \bullet Tcomposite)$$
$$Tlatency = Tsample + k \bullet \max(Trender, Tcomposite) + Tcomposite$$

Figure 5.6: Time line for low-latency supersampling system. (R$_{i,j}$ means render region *i*, sample *j*. C$_{i,j}$ means composite region *i*, sample *j*).

5.2.3 Latency in an A-buffer System

Since A-buffer image-composition systems use scan-line renderers, rather than *z*-buffer renderers, their rendering time lines look quite different, as shown in Figure 5.7.

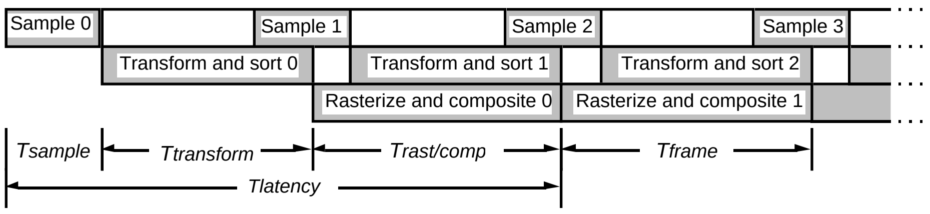


$$Tframe = Tlatency = Tsample + Ttransform + Trast/comp$$

Figure 5.7: Time line for simple A-buffer image-composition system.

The rendering algorithm has two main steps: transforming/bin-sorting primitives and rasterizing/compositing pixels in scan-line order. Because rendering occurs in scan-line order, the entire scene must be transformed and bin-sorted before rasterization can begin.

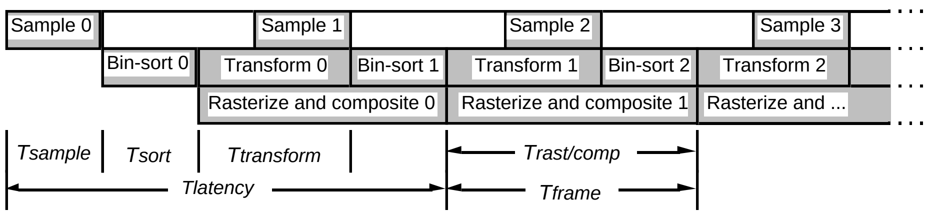
Figure 5.8 shows the rendering time line for a pipelined, A-buffer image-composition system.



$$Tframe = \max(Tsample,\ Ttransform,\ Trast/comp)$$
$$Tlatency = Tsample + Ttransform + Trast/comp$$

Figure 5.8: Time line for pipelined A-buffer image-composition system.

Is there any way to break steps into smaller parts to reduce the latency? The answer is yes—provisionally. The transformation process involves several operations: transforming vertices into screen coordinates, transforming normals, computing lighting values, etc. The only part necessary for bin-sorting is computing the *y* extents of primitives and placing them into bins for each scan line. If the application provides sufficient flexibility, we may be able do just these steps in a quick first pass, and do the rest of the transformation later. We would then have the rendering time line shown in Figure 5.9.



Tframe = max(*Tsample*, *Tsort* + *Ttransform*, *Tcomposite*)

Tlatency = *Tsample* + *Tsort* + max(*Ttransform*, *Trast/comp*)

Figure 5.9: Time line for A-buffer system with early bin-sorting pass.

Ideally, we could simply compute the screen-space *y* coordinates of each vertex of each primitive and compute the minimum and maximum *y* value for each primitive's vertices (the *y* extent of the primitive determines which scan-line bin the primitive belongs to). We could then store a pointer to the primitive in object coordinates, rather than the entire transformed primitive, as is typically done. This would require minimal storage—approximately two words per primitive—and would require only a fraction of the overall work of the transformation stage. We would not have to compute *x* and *z* screen-space coordinates, transform normal vectors, or do any setup calculations necessary for rasterization; these could all be done later.

Unfortunately, we may not be able to do things so simply. Most graphics systems use a hierarchical representation for the database. Local transformation matrices may appear anywhere within the hierarchy to provide local translations or rotations for subtrees in the hierarchy, such as the four instances of wheels on a car.

A current-transformation matrix is maintained as the hierarchical structure is traversed. This matrix is needed to transform the vertices from object coordinates to screen coordinates, as well as to transform vertex normal vectors. If we wait until the rasterization phase to compute *x* and *z* screen coordinates and vertex normal vectors, we will no longer have the current transformation matrix.

It may be possible to store references to the various transformation matrices with each bin-sorted primitive—so long as the hierarchy does not contain instances (subtrees that are referenced in more than one place in the hierarchy).   Instanced primitives have more than one transformation matrix associated with them. Because of these complications, the notion of early bin sorting is still a matter for research.

If these problems can be overcome, and they certainly can be for applications with simple display structures, latency in an A-buffer image-composition system can be reduced to slightly more than a frame time plus the time required for the bin-sorting pass—very near the optimal value.

5.3 STATIC LOAD BALANCING

An image-composition system achieves its speed by partitioning the display database over multiple renderers and rendering the partitions in parallel. For good system utilization and highest performance, the computational load must be distributed evenly across the renderers and over an entire frame time. The distribution can be uneven in several ways:

- Renderers can have unequal numbers of primitives.
- Primitives on each renderer can take unequal time to process.
- Primitives can map to different parts of the screen, making rendering the bottleneck at times and composition the bottleneck at other times.

The first two points are examples of static load imbalances: differences in the amount of work renderers have to perform. The third point is an example of dynamic load imbalance: changes in processing load that occur over time. Both types of imbalance can affect system performance. The effects of static imbalances, such as unequal numbers of primitives or unequal processing times on each renderer, are obvious: certain renderers finish early and are idle, while others take longer and prolong the frame time. The effects of dynamic imbalances are more subtle, but can create even worse performance penalties.

We will discuss static load balancing in the following sections. We will discuss dynamic load balancing in Section 5.4.

5.3.1 Database Distribution

To achieve static load balance, the computational load must be distributed evenly across the renderers and over an entire frame time. To accomplish this, we must distribute primitives so that each renderer has the same number of primitives or—even better—that the primitives on each renderer take the same time to process.

Graphics systems store and traverse a display database in one of two ways: *retained mode*, in which primitives are stored in a (mostly) static display structure that is traversed every frame, or *immediate mode*, in which the application generates primitives anew every frame [FOLE90]. Each has advantages for certain applications and each requires a distinct approach to database distribution. Consequently, we will talk about them separately.

5.3.2 Retained-Mode Traversal

In a retained-mode display structure, each renderer is initially allocated a fraction of the dataset, which it displays every frame. The database can be edited each frame (including changes to transformation matrices), but the assumption is that the vast majority of the data structure is retained from frame to frame. Since the display structure is generally implemented as part of a graphics library, and the user has only indirect access to the data, the retained-mode approach gives the rendering software control over how primitives are represented and assigned to renderers.[1]

We want to assign primitives to renderers in a way that is easy to implement, has low computational cost (so the overhead of distributing primitives does not cancel the gains from evening the load), and is robust with respect to changes in view and changes to the database. This is easily done if the display structure is "flat" or non-hierarchical. In this case, the display structure is simply a collection of primitives with a single transformation matrix.

If there are P primitives and R renderers, we can simply assign P/R primitives to each renderer. Note that this can be done in one of two ways: we can shuffle primitives in a round-robin fashion, assigning the first primitive to the first renderer, the second to the second renderer, and so forth; or we can assign the first P/R primitives to the first renderer, the next P/R primitives on the second renderer, etc. We call the first approach *scattering*, and the second *clustering*.

The primitives in most databases contain some amount of geometric coherence. That is, primitives near each other in the display structure generally lie near to each other in the scene as well. This means that clustering tends to preserve this coherence and that primitives in different partitions are likely to fall into different regions of the screen. Scattering, on the other hand, tends to make the distribution of primitives on the screen similar for each partition. We will see in Section 5.4, that this can be very important for dynamic load balance.

Clearly, it is easy to assign the same number of primitives to each renderer. What if different primitives take different times to process? This is only a minor issue if the database consists of discrete polygons, but can be a major concern if it is composed of aggregate primitives, such as triangle strips, which can have widely varying lengths. The solution used in Pixel-Planes 5, is to assign a weight to each primitive, based on an estimate of the time it will take to process [ELLS90a]. Discrete polygons or triangle strips containing

[1]Distributed display structures have only recently become a topic of interest in the graphics literature. [ELLS90a] describes many of the issues involved and presents an implementation of a distributed PHIGS-like display structure on Pixel-Planes 5.

only a few triangle receive low weights; long triangle strips or complex primitives, such as Bezier patches or NURBs, receive high weights. The weight can be a simple function that depends on simple parameters, such as the number of vertices in a triangle strip. When each new primitive is encountered, it can be added to the renderer with the lowest load (determined by summing the weights of its primitives).

Weighting primitives can do an excellent job at equalizing the load between renderers. It creates other problems, however: if the user wants to edit or delete a primitive, how do we determine which renderer it resides on? How do we assure that a series of edits to the database does not destroy the load balance? There are many ways these issues can be tackled, but they are beyond the scope of this dissertation. [ELLS90a] contains a discussion of many of these issues.

5.3.3 Hierarchical Display Structures

Most graphics applications require a more sophisticated database model than a simple collection of primitives. The standard solution is a hierarchical display structure, in which state-changing commands, such as transformation matrices, are intermingled with primitives in a tree (or directed acyclic graph) structure (see Figure 5.10). This approach has been almost universally used in graphics to support articulation, instancing, and local transformations for parts of objects [FOLE90].

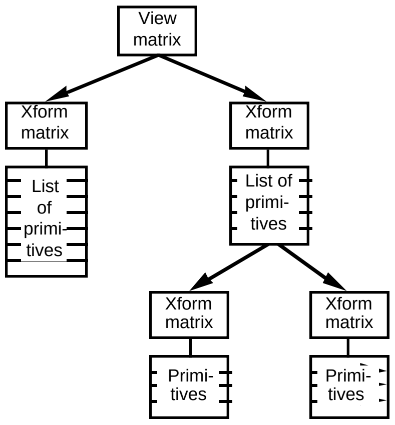


Figure 5.10: Hierarchical display structure.

Hierarchical display structures introduce context into the traversal problem: what happens to a primitive depends on more than the primitive's raw coordinates—it depends on *where* the primitive lies in the display structure. For example, a primitive is transformed by the concatenation of all the transformation matrices above it in the hierarchy.

Traversing a single display structure of this type is relatively straightforward: a stack is used to store the current context, and the hierarchy is traversed in depth-first order. When a branch in the hierarchy is encountered, a copy of the current context is pushed onto the stack so that any state-changing commands encountered below the branch point can be "undone." When the traversal reaches the branch point on the way up, the stack is popped, restoring the context to its value before the branch was taken.

A distributed database model must emulate this process; it must compute the same context information during traversal. As with flat display structures, there are two main approaches: *scattering* and *clustering*, which are generalizations of the techniques above.

Scattering. Scattering, as before, means scattering primitives across the renderers. Now, however, the display structure contains many small collections of primitives scattered throughout the hierarchy, rather than a single collection of primitives that can be treated the same. We can scatter the primitives in each of these smaller collections of primitives across the renderers, but how do we retain the context information? Since primitives from every part of the display structure are likely to reside on each renderer, the only general solution is to replicate the entire hierarchical structure at each renderer. The display structure on each renderer, therefore, is a miniature copy of the global display structure—but only contains a fraction of the primitives. Figure 5.11 shows a simple data structure scattered over two renderers.

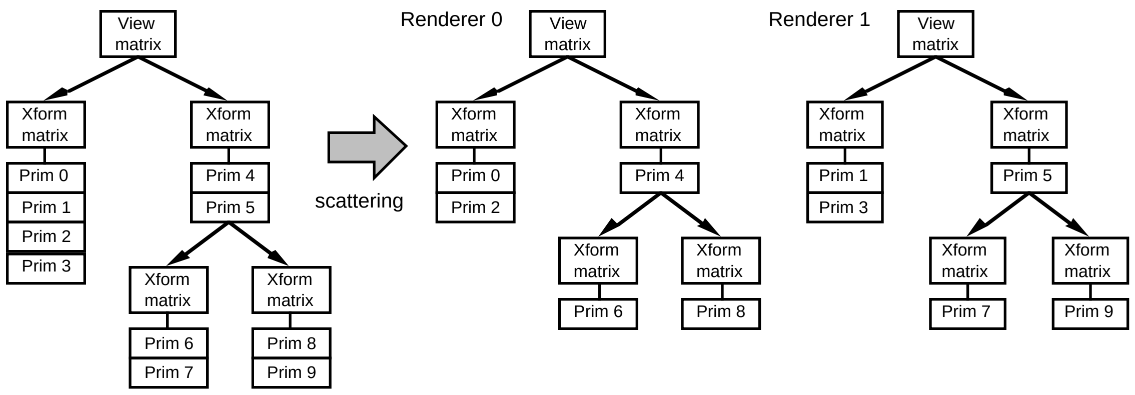


Figure 5.11: Distributing a display structure by scattering.

Scattering has the advantage of making load balancing among renderers nearly automatic under almost all conditions, but has the disadvantage of requiring extra space to store the multiple copies of the hierarchy and extra time to traverse them on each renderer. For databases with deep hierarchies, these can be very costly. However, many databases have fairly shallow hierarchies with large numbers of primitives at the nodes.

Clustering. We can generalize clustering by assigning entire subtrees of the display structure to different renderers, as shown in Figure 5.12. A renderer, therefore, need store only the portions of the hierarchy for which it is responsible; it does not need a copy of all the state-changing commands. This scheme saves space and reduces the time spent traversing the hierarchy, but makes load balancing among renderers more difficult for several reasons.

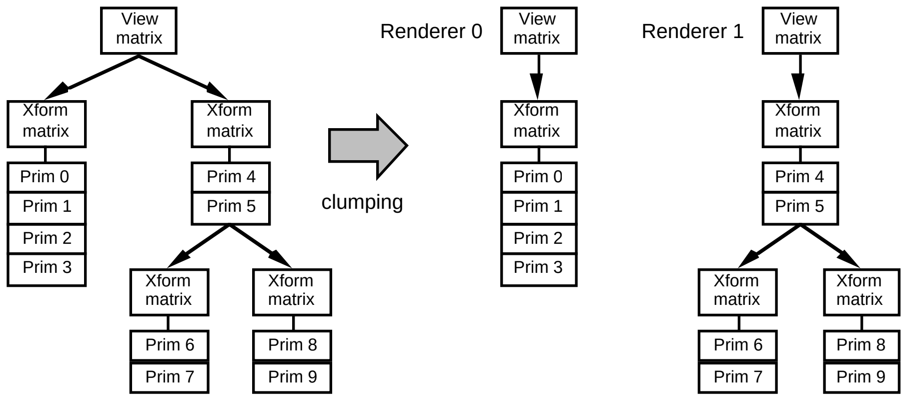


Figure 5.12: Distributing a display structure by clustering.

First, it is unlikely that there will be the same number of subtrees as renderers, and even more unlikely that each subtree will contain the same number of primitives (or total primitive weight). We must be prepared, therefore, to split subtrees across renderers, meaning that some context, at least, must be stored redundantly.

Second, subtrees are likely to be geometrically coherent. That is, primitives in a subtree are likely to occupy a similar part of object space, hence, screen space. When the viewing transformation changes, entire subtrees may enter or exit the view frustum, dramatically changing the number of primitives actually rendered on different renderers. To overcome these limitations, a dynamic load calculation must be performed on the database, and heuristics used to allocate subtrees to renderers. The overhead of allocating the database in this manner could easily overwhelm any savings in database traversal time by not replicating the hierarchy on each renderer.

It is the author's opinion that scattering is the method of choice for most applications on image-composition machines. The only exceptions are likely to be applications with large numbers of subtrees that contain few primitives, such as molecular modeling or scientific visualizations. An alternate approach for this type of application is immediate-mode traversal, which we will discuss next.

5.3.4 Immediate-Mode Traversal

The alternative to a retained-mode display structure is immediate-mode traversal. In this case, the application generates the primitives anew every frame. In a parallel graphics system, this either means that there must be separate channels from the processor (or processors) running the application to the renderers—*external immediate mode*, or that the application runs in parallel on the renderers themselves—*local immediate mode*.

In either case, the application programmer decides how primitives are generated and which primitives are sent to which renderer; the system architect or graphics library designer has no control over how this is done. Ideally, the application programmer will write his program so that similar numbers (or weights) of primitives are sent to each renderer. Immediate-mode traversal causes even more difficult problems if the distribution of primitives over the screen is uneven. We discuss this class of problems next.

5.4 DYNAMIC LOAD BALANCING

In Section 5.3, we discussed static load balancing: assuring that each renderer has the same amount of work to do each frame. We now discuss *dynamic load balancing:* assuring that load imbalances do not develop over time. Dynamic load imbalances arise for more subtle reasons that static ones, and often cannot be solved as easily. For example, a particular dataset and choice of viewing transformation may map all primitives to the top part of the screen. A scan-line renderer will bog down during the first few scanlines, while the image-composition network waits. Later, the renderer will have little to do, but the image-composition network will be saturated. There is no way to assure that primitives will cover the screen evenly. The only recourse against problems like this is to add elasticity or buffering to the system. This will be our general approach to solving dynamic load imbalances.

5.4.1 *Z*-Buffer Systems

Simple *z*-buffer systems, or *z*-buffer systems that use supersampling, are relatively immune to dynamic load imbalances. The full-screen image buffer at each renderer provides a full frame of buffering that decouples rendering from image composition. Primitives can be clumped anywhere in the image without affecting performance, since composition does not begin until all the primitives have been rendered. In some sense, the buffering to even out dynamic load imbalances has already been paid for in this type of system.

5.4.2 A-Buffer Systems

A-buffer systems, on the other hand, are prone to dynamic load imbalances. One might suppose that A-buffer systems are economical because they require little temporary storage: at first glance, they do not seem to require much storage between the renderer and composition network, since both process pixels in the same order, and bin storage can be minimized by pre-sorting, as described in Section 5.2.3. This naive approach leads to two problems, one minor, and one catastrophic.

Uneven distribution of primitives over the screen. Consider the example above, in which a particular database and viewing transformation cause every primitive in the dataset to fall into the upper fifty scan lines of the screen, as shown in Figure 5.13. Rendering is slow for the pixels in the top part of the screen, since the list of active primitives is long, and several primitives may contribute to each pixel. During the first portion of the frame time, the system's speed is limited by the renderers. When rendering proceeds past this congested part of the screen, however, the renderers have very little to do, but the compositors must process all of the remaining pixels. System performance, at this point, is limited by the composition network.

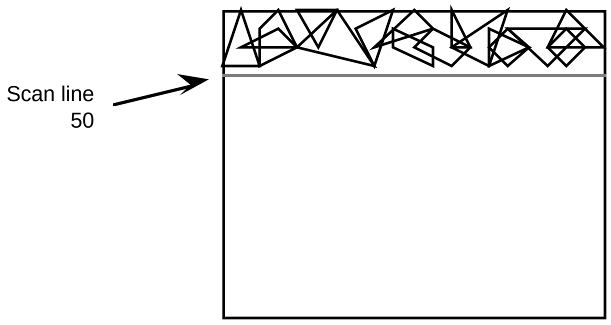


Figure 5.13: Dynamic load imbalance caused by uneven distribution of primitives across the screen.

If rendering performance is balanced with composition-network performance (i.e., the ideal rendering time equals the composition time), this type of load imbalance causes at worst a factor-of-two speed penalty. In the worst possible case, rendering and compositing cannot be overlapped, doubling the overall rendering time.

The solution to this type of problem is to add buffering between the renderers and the composition network. This allows the renderers to speed ahead when they have little work to do, and to take extra time when they are heavily loaded, without slowing the composition network down. A-buffer systems have a mitigating factor: the number of fragments per pixel increases in denser parts of the image. This slows the composition network down at the same time the renderers are heavily loaded. Sparse regions of the screen likely result in little more than one fragment per pixel, allowing the composition network to process pixels faster.

Different distributions on each renderer. Consider what happens when a dataset is distributed across the renderers so that primitives from each renderer map onto different parts of the screen, as shown in Figure 5.14.

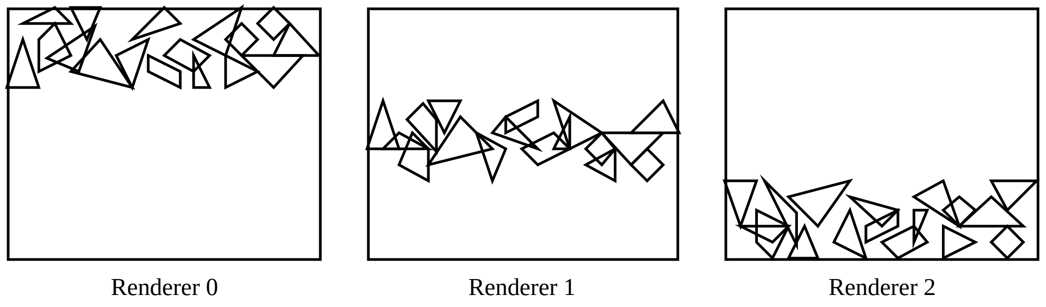


Figure 5.14: Dynamic load imbalance caused by uneven distribution of primitives across the screen *at each renderer*.

Each renderer now is loaded in different parts of the image. If composition is performed synchronously with rendering, the entire system proceeds at the speed of renderer with the heaviest load, and this changes as we move through the image. In a very unbalanced case, only one renderer is active at a time while the other renderers are idle—the effective parallelism is lost and the whole system behaves as a system with one renderer! This is a terrible effect, which we must prevent if at all possible.

5.4.3 Dynamic Load-Balance Ratio

The easiest way to deal with this problem is to ensure that primitives are well-distributed over the renderers. Scattering generally does a good job of this. To provide a quantitative measure of how well primitives are distributed in a given database, we define a quantity called the *dynamic load-balance ratio*. This is an estimate of the number of times longer a particular database will take to render on a system with tightly coupled rasterizers and compositors than the same database rendered on the same system, but with ideally distributed primitives.

To calculate the dynamic-load-balance ratio, we first divide the screen into regions that represent a fraction of the rendering task for the particular rasterization method (for scan-line A-buffer systems, we divide the screen into stripes of 5 to 10 scan lines; for the prototype system described in Part II, we divide the screen into rectangles). We then count the number of primitives from each database partition that are active in each screen region and compute two sums, S_{max} and S_{avg}:

$$S_{max} = \sum_{regions\ R} \max_{renderers} (\#\ of\ \ primitives\ \ in\ R)$$

$$S_{avg} = \sum_{regions\ R} \operatorname{average}_{renderers} (\#\ of\ \ primitives\ \ in\ R)$$

S_{max} is the sum over all screen regions R of the maximum number of primitives on any renderer falling into R. It represents the rendering effort spent given the uneven distribution of primitives. S_{avg} is the sum over all screen regions R of the average number of primitives on each renderer falling into R. It represents the theoretical amount of rendering effort required. The dynamic load-balance ratio, ρ, is defined by S_{max}/S_{avg}, and can assume values ≥ 1. It can be interpreted as the number of times longer a database can take to render than one would predict, based on the number of primitives.

Appendix A calculates ρ for the sample databases assuming rectangular regions, rather than horizontal stripes, as we have assumed so far (this makes sense for the prototype system and should give similar results to a calculation using horizontal stripes). The values range from 1.05 to 1.78 for scattered datasets, and from 6.31 to 13.17 for clustered datasets. Thus, scattering has better dynamic load-balance characteristics than clustering. This is to be expected, since scattering distributes nearby primitives over each of the renderers, whereas clustering puts nearby primitives on the same renderer.

We cannot always choose how primitives are distributed, however. Some of the sample datasets have high dynamic load-balance ratios even after scattering. One can imagine datasets where this can occur, no matter what the distribution method. Immediate-mode traversal creates an even worse problem, since we have no control over which primitives are assigned to which renderer. The most likely result is that nearby primitives will be sent to the same renderer—the worst possible scenario.

Clearly, we must be prepared to control the damage when it does occur. The only solution is to add buffering between the renderers and the image-composition network to decouple the two processes. The amount that is needed depends on the magnitude of the dynamic load imbalance. Unfortunately, it is easy to imagine situations where a large fraction of a frame's worth of buffering is needed. The example above, where the primitives on each of four renderers map to different quadrants, is one such situation.

This gives rise to a general rule that appears to apply to all image-composition systems:

Rule of image composition and buffering: Image-composition architectures require a substantial fraction of a frame of buffering on every renderer unless there is explicit control over the distribution of primitives to renderers.

Z-buffer systems satisfy the law automatically with their full-screen image buffer. A-buffer systems must have a very robust database distribution algorithm or must have substantial buffering between the renderer and composition network. The prototype system described later in this dissertation is a hybrid between *z*-buffer and A-buffer styles of image-composition architecture. The need for buffering played a large part in the development of the architecture.

5.5 ADVANCED RENDERING ALGORITHMS

Up to this point we have concentrated on rendering polygonal databases with Gouraud or Phong shading. These currently are the prevalent algorithms on real-time, 3D graphics systems, but are by no means the only ones. Other more complex and realistic primitives types and rendering algorithms have been developed and are gaining popularity. For example, there has been an explosion of new results in volume rendering in recent years. Volume rendering is a more natural method of visualizing many 3D datasets than surface rendering. Also, many recent graphics systems support texturing. Texturing vastly increases the visual detail in an image without increasing its geometric complexity.

Algorithms such as these are becoming increasingly important. For a graphics architecture to be viable, it must be flexible enough to support a variety of algorithms at reasonable performance (unless it can find an important application niche). This section considers the suitability of image composition architectures for advanced rendering algorithms and primitive representations. We will start by discussing support for complex shading models, then consider texturing and volume rendering.

5.5.1 Deferred Shading

Gouraud shading requires only three values per pixel: *red*, *green*, *blue*, all of which are linearly interpolated. Phong shading, on the other hand, requires the same number of interpolated values, several dot products, a reciprocal square root, and numerous multiplications. Shading can get even more complicated. For example, several light sources can be present; primitives can have different specular and diffuse colors; fog or atmospheric haze can be present. Complicated shading models such as these can require a great deal of computation—far more, even, than hidden-surface elimination itself.

The straightforward method of rendering is to evaluate the shading model for each pixel of each primitive as it is rasterized. This can be extravagant, however, since pixels in one primitive are frequently covered by pixels from another primitive. Furthermore, many parts of most shading calculations do not depend on the primitive: for example, when Phong-shading a pixel, the surface normal vector must be normalized, no matter what primitive the normal vector came from. This general notion suggests an alternate way of structuring the shading calculations: we can compute intermediate values, such as intrinsic colors and (unnormalized) surface normal vectors, on a per-primitive basis during rasterization, but defer shading calculations until all of the primitives have been rasterized.

This technique is commonly known as *deferred shading*. It has been proposed and/or implemented on several systems, most notably Deering's Triangle Processor and Normal Vector Shader of [DEER88], PROOF [SCHN88], and Pixel-Planes 5 [FUCH89]. Deering and Schneider each proposed triangle-processor systems, which Phong-shade their scenes using a short pipeline of shading processors. Pixel-Planes 5 shades 128x128-pixel regions after all of the primitives in the region have been rasterized.

We can use this same approach in an image-composition architecture on two levels: we can defer shading within renderers and we can defer shading throughout the entire system.

Deferred shading on individual renderers is nothing new; since renderers are complete graphics systems, we can defer shading on them as on any other system. Deferred shading on an entire image-composition system is much more powerful; if we send intermediate information over the image-composition network, rather than RGB values, we can defer shading until the entire image has been composited. This means we only have to shade each pixel once, independent of the number of renderers and the depth complexity of the pixel.

This has impact on the architecture, however. We need to perform these shading calculations after the pixel streams have been composited—at the end of the composition network. We can either place a shading processor before the frame buffer, as in Deering's system, or provide a frame buffer with pixel-processing capabilities, as in Pixel-Planes.

A side benefit to including a shading processor is that it can also be used to blend sample values for supersampling, or to assemble pixels in an A-buffer system. Deferred shading is a powerful concept when applied to image-composition architectures. It plays an important role in the prototype system described in Part II.

5.5.2 Transparency

True transparency, the accurate rendering of semi-transparent surfaces, requires compositing semi-transparent surfaces that cover a pixel in front-to-back or back-to-front order. Since transparent surfaces can reside on any renderer, and an arbitrary number of transparent surfaces may cover any given pixel, compositing cannot occur until all possible surfaces are present—at the end of the image-composition network. *Z*-buffer architectures cannot implement true transparency, since they allow only a fixed amount of data per pixel, and an arbitrary amount is required. A-buffer architectures can support transparency, at the cost of increased load on the image-composition network.

***Z*-buffer (supersampling) architectures.** Since *z*-buffer architectures can transmit only a fixed amount of information per pixel, and an arbitrary amount may be required, the only general solution is to run the network multiple times, once for the opaque primitives, and once for each transparent primitive[1]. The color and *z* values from each pass must be stored in an accumulator frame buffer before the surfaces can be composited (they cannot be composited directly, since the primitive in each new pass may fall anywhere in the list of transparent polygons).

This algorithm can handle only a few transparent primitives, since the composition-network bandwidth is multiplied by the number of transparent primitives plus one (for the opaque primitives), and an accumulator frame buffer has limited storage.

The only feasible approach for *z*-buffer architectures seems to be screen-door transparency. This works by computing some samples with transparent objects present and other samples with transparent objects excluded (several samples per pixel are required for this approach). Unfortunately, only a few transparency levels are available and the number of transparent objects is limited, since overlapping transparent objects should have uncorrelated samples. Sample coverage can be computed stochastically as well. This makes it possible to increase the number of transparent objects, but adds noise to the image [ELLS91].

A-buffer architectures. A-buffer architectures can implement true transparency in a straightforward way. Fragments can be created for transparent primitives in the same way as opaque ones. These fragments can be conveyed, together with the fragments from opaque primitives, over the image-composition network to the pixel assembler. Since the fragments arrive in front-to-back order, they can be used to compute the pixel color. Three enhancements to the architecture are required:

1) Fragments must contain an alpha (transparency) value along with RGB values.
2) Transparent fragments must not affect pixel coverage in the compositors.
3) The pixel assembler must be able to composite semi-transparent objects.

The main difficulty with this approach is the added load on the image-composition network. If a single transparent object covers the entire screen, the average number of fragments per pixel increases by one. This is significant, because the average number of fragments per pixel in the test images ranges from 1.13 to 1.68. The addition of one or more fragments per pixel seriously increases the amount of data that must be conveyed over the image-composition network. In addition, the pixel assembler must composite fragments for each sample point, rather than finding the nearest fragment at each sample point. This is a significant amount of extra computation, but is restricted to a single location in the entire system.

5.5.3 Textures

Image-based textures require prefiltering to avoid aliasing artifacts. Whether the filtering is done by MIP maps [WILL83], summed-area tables [CROW84], or some other method, several table look-ups are required per sample. Procedural textures require a procedure evaluation per sample. Both require substantial amounts of calculation.

In some ways, texturing can be considered orthogonal to image composition, since it really is just a sophisticated form of shading. It could be performed on the renderers, if they are suitably equipped. However, texturing can take advantage of the deferred-shading principle. If it is performed at the end of the image-composition network, it need only be performed for visible pixels (or visible fragments, in A-buffer systems). There are added benefits for texturing: the texture tables need only be stored once.

---

[1]This is similar to multi-pass algorithms that have been developed for simple *z*-buffer renderers [MAMM89]. The difference is that in these multi-pass algorithms, each pass captures the "next-closest" surface. The new surface can be composited in with the previous ones after each pass. The number of passes required is equal to the number of transparent layers that can cover a pixel plus one (for the opaque primitives). This method does not extend to image-composition architectures unless a method is provided for conveying composited images back to the individual renderers. Without this feedback, the compositors have no way of detecting the "next-closest" surface.

Image-based texturing can only be performed in real time with special hardware to look up texture values. Procedural textures are similar in nature to other shading calculations, and can be implemented on the deferred-shading processor. Figure 5.15 shows a block diagram of a system consisting of renderers, a texturer (for image-based textures), and a deferred shader.

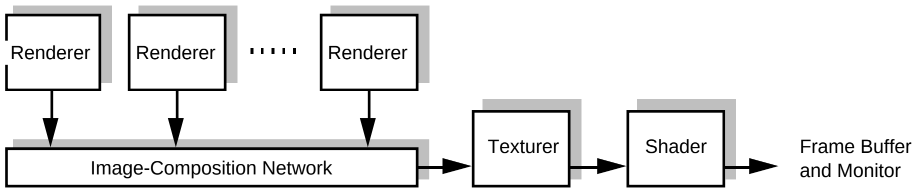


Figure 5.15: Texturing implemented as a deferred-shading process.

5.5.4 Volume Rendering

Volume rendering is the direct display of images computed from a 3D array of data. The data may represent density, emissivity of a particular type of radiation, etc. There are three competitive algorithms for rapidly displaying volume data: 1) ray-casting, in which beams are cast into the volume dataset for each pixel (or pixel subsample) [LEVO88, SABE88, UPSO88] and 2) serial transformations, in which the data array is transformed by a sequence of three 1-D shear operations to orient it correctly for a particular view direction [DREB88, HANR90], and 3) projection methods (splatting), in which voxels are mapped to the screen and composited [WEST89, SHIR90, MAX90, WILH91].

Splatting appears to be the most promising method for implementation on an image-composition system. In the standard splatting algorithm, voxels in the dataset are traversed in front-to-back or back-to-front order and composited into an image buffer using a Gaussian (or other) filter kernel, an operation very similar to polygon rendering.

On an image-composition architecture, the volume dataset can be partitioned into blocks or slabs, with one partition assigned to each renderer. The renderers traverse the voxels in their partition, projecting voxels onto the screen and compositing them into pixels lying in a neighborhood of the center of projection of the voxel. The contribution to each pixel depends on the distance and direction from the pixel to the center of projection of the voxel.

Voxels can thus be considered standard surface primitives with two odd properties:

1) They must be traversed in order.
2) They are semi-transparent. The transparency depends on the distance and direction from the pixel center to the center of projection.

Figure 5.16 shows a voxel splatted onto an image buffer. After all the pixels have been splatted, the image buffer contains a semi-transparent image of all the voxels. These are then composited over the image-composition network.

We have assumed that an image buffer is available for the rendering calculations. This is not strictly necessary. In an A-buffer image-composition system, pixels can be calculated in scan-line order by maintaining a list of active voxels and compositing them for one pixel at a time. These compositing operations could conceivably be performed over the image-composition network.

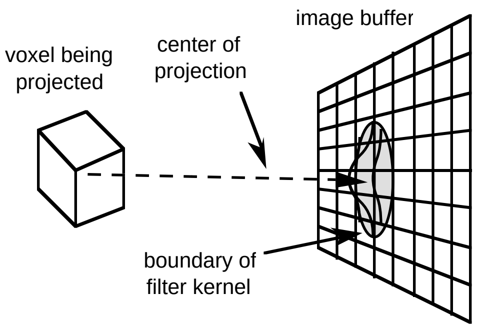


Figure 5.16: Voxel splatted onto image buffer.

If the compositors are general enough to alpha-blend, they can composite the images directly. The images must be composited in the correct order, of course. This places restrictions on the way that the database can be partitioned. If the database is partitioned into slabs, the compositing order will be correct for view directions that are near to perpendicular to the cut planes between slabs. Since this cannot be guaranteed for all viewpoints, the database must be replicated three times and partitioned along different axes so that, for any view, a set of slabs is available that allows compositing in the proper order (only one set of slabs is used for any particular frame). This means replicating the database three times, but does not increase the amount of computation during rendering [NEUM91].

If the compositors are not sophisticated enough to alpha-blend images, blending can be done in a deferred shader. To do this, each image from each renderer must be transmitted to the deferred shader, which composites them together. This results in lower performance, since the traffic on the image-composition network is multiplied by the number of renderers. Since present volume datasets generally contain 256^3 or fewer voxels, the final image may only need to have relatively low resolution. This could reduce the amount of data that must flow over the image-composition network.

Ulrich Neumann has implemented a parallel splatting algorithm on Pixel-Planes 5, with compositing performed over Pixel-Planes 5's general-purpose ring network, instead of a special-purpose image-composition network. He has achieved frame rates of up to 3 Hz on 128^3 datasets [NEUM91].

5.6 TAXONOMY OF IMAGE-COMPOSITION ARCHITECTURES

As we have seen in the previous sections, the basic notion of compositing images from multiple renderers leads to a variety of image-composition architectures. We attempt to organize these into a taxonomy in this section.

The points of differentiation we have chosen for the taxonomy are the following:

1) Priority method: fixed-priority or pixel-priority.
2) Number of primitives per renderer: one or multiple.
3) Antialiasing method: simple (α-blend or Duff), supersampling, or A-buffer.

There are other points of differentiation as well: the traversal method (retained mode or immediate mode), the rendering method (*z*-buffer or scan-line), the amount of buffering between renderers and compositors, the composition-network topology, the presence or absence of deferred shading, etc. We felt that these are secondary to the points listed above or are dependent on other characteristics of the system (for example, *z*-buffer rendering is incompatible with A-buffer antialiasing).

Figure 5.17 shows the resulting taxonomy tree. The leaves of the tree are labeled by the name of the class of system and by any examples that have been proposed or built. The taxonomy shows only branches that were deemed both feasible and interesting (for example, fixed-priority architectures do not require sophisticated antialiasing, so branches for fixed-priority architectures with supersampling or A-buffer antialiasing are not shown).

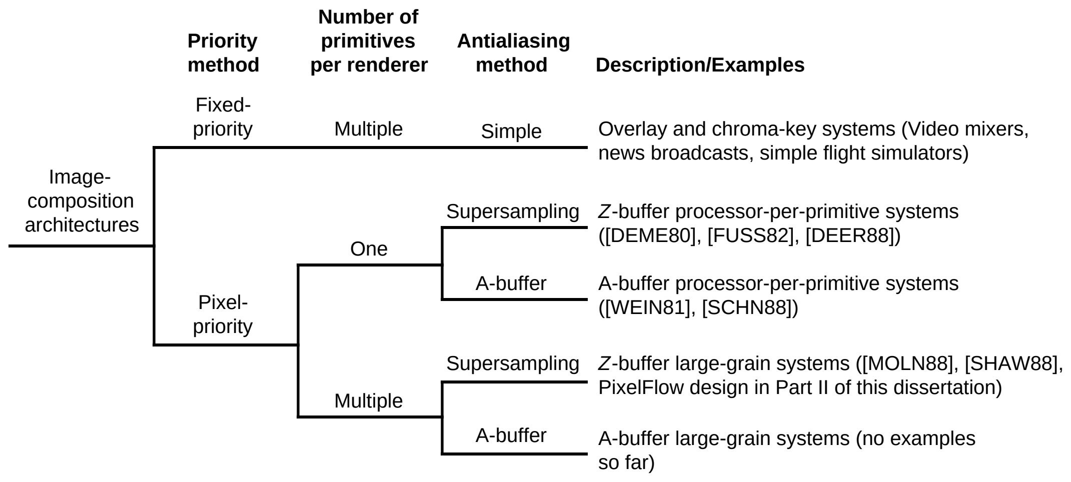

Priority method
Number of primitives per renderer
Antialiasing method
Description/Examples
Image-composition architectures
Fixed-priority
Multiple
Simple
Overlay and chroma-key systems (Video mixers, news broadcasts, simple flight simulators)
Pixel-priority
One
Supersampling
Z-buffer processor-per-primitive systems ([DEME80], [FUSS82], [DEER88])
A-buffer
A-buffer processor-per-primitive systems ([WEIN81], [SCHN88])
Multiple
Supersampling
Z-buffer large-grain systems ([MOLN88], [SHAW88], PixelFlow design in Part II of this dissertation)
A-buffer
A-buffer large-grain systems (no examples so far)

Figure 5.17: Taxonomy of image-composition architectures.

The branches that are the most interesting for high-performance systems are *z*-buffer and A-buffer, large-grain systems (discussed in Chapter 4). *Z*-buffer (supersampling) systems are simpler to implement and do not produce shading artifacts, since shading is performed independently for each sample point. However, they are poor for rendering transparent objects and require a very high bandwidth image-composition network. A-buffer systems are more complex, since the pixel-stream format is complicated. However, they easily handle transparent objects and have lower image-composition network bandwidth. Neither architecture is a clear choice for all applications. Part II presents a prototype system based on the *z*-buffer approach.

5.7 SCALABILITY AND ECONOMICS

Any parallel graphics system requires communication between different processing units. In most architectures, these communication requirements scale as a function of the rendering speed. For example: the frame-buffer memory bandwidth in architectures such as Silicon Graphics' VGX scales linearly with rendering speed; the global communication network bandwidth in a sort-middle architecture such as Pixel-Planes 5 scales linearly with rendering speed. Both of these become difficult to implement above certain performance levels: frame-buffer partitioning reaches a point of diminishing returns as the number of partitions exceeds the number of pixels in a typical primitive; global communication networks are difficult to build at high speeds.

Sort-last architectures have the property that the bandwidth between adjacent modules is constant. Since only local communication with fixed bandwidth is required between adjacent renderers, the capacity of the communication network scales as renderers are added to the system. Hence, the performance of sort-last architectures increases nearly linearly with the number of renderers, as does the price of the system.

How do image-composition architectures compare to existing architectural approaches? Figure 5.18 gives the performance and price for a number of currently-available rendering systems. The information was obtained in most cases from sales representatives of the various companies and should be interpreted with appropriate caveats. For example, the performance figures are never-to-be-exceeded, peak numbers that can be difficult to achieve in actual applications. Furthermore, systems may perform best under different conditions and can be configured with various amounts peripheral hardware, such as disk drives, memory etc. All of this makes meaningful comparisons between systems difficult.[1]

[1]Some systems contain hardware for Gouraud shading with no performance penalty and quote performance for Gouraud-shaded triangles; others quote performance for flat-shaded triangles. Some assume triangles are contained in triangle-strip meshes; others assume independent triangles. Some systems are equipped with multiple general-purpose processors and high-performance disk drives; others are stripped down and therefore look deceptively inexpensive.

The performance estimates for the PixelFlow prototype system (described in Part II) are based on rasterizer performance alone, rather than the pessimistic assumptions made in the performance analysis section of Chapter 9. These optimistic estimates are conceivable, given a sufficiently restricted application program (this is generally the criterion used for "marketing" performance in commercial systems). We calculated performance in this way to put the prototype system on a more even footing with the commercial systems. The price for the PixelFlow systems was obtained by dividing the component-cost estimate from Appendix B by 2 for production and multiplying by 3 for markup and overhead.

System	Tris/sec	Price	Notes
Hewlett-Packard 7000 T4	330 K	$118,190	50-pixel, 10x10-pixel bounding box, flat-shaded tris. Price is for Model 750, 16 Mbytes RAM, SCSI disk, 19" monitor, 1280x1024 resolution.
Silicon Graphics "Indigo"	40 K	$14,031	50-pixel, 10x10-pixel bounding box, flat-shaded tris. Single R3000 processor with 16 Mbytes RAM, 432 Mbyte disk.
Silicon Graphics 420VGX	1.1 M	$168,050	50-pixel, 10x10-pixel bounding box, flat-shaded, unlit tris. Two R4000 processors, 385 Mbyte disk, 32 Mbytes RAM, 4x10 array of image engines.
Silicon Graphics 480VGX	1.1 M	$268,050	As above, except 8 R4000 processors.
Silicon Graphics SkyWriter	2.2 M	$249,900	As above, except 4 R4000 processors and two VGX pipelines.
Sun Microsystems VX/MVX	100 K	$120,000	Estimates from a developer of the system.
Prototype system 1/2 card cage	1.3 M	$266,000[1]	Gouraud-shaded tris. 9 renderers, 2 shaders, bin-replication factor = 1.2. Performance based on rasterizer only. Price from Appendix B • 0.5 (production) • 3 (markup and overhead)
Prototype system 1 card cage	2.6 M	$435,000[5]	As above, except 18 renderers and 2 shaders.
Prototype system 2 card cages	5.2 M	$822,000[5]	As above, except 36 renderers and 4 shaders.

Figure 5.18: Performance and price for various systems available in 1991 (including estimates for the prototype system).

Unfortunately, the communication bandwidths in an image-composition network are huge, particularly for high-resolution images with supersampling. In the previous chapters, we have shown how such bandwidths can be achieved. For image-composition architectures to be cost-effective, the cost of a node in the image-composition network must be low compared to the cost of a renderer, and this requires high-performance renderers.

Figure 5.19 plots this same information. One can see that current commercial systems have a maximum rendering speed in 1991 of approximately 2 million triangles per second. The PixelFlow prototype system, on the other hand, should achieve this in a one card-cage configuration. The performance/price ratios for some of the commercial systems are higher. This means that if one desires rendering performance in the range possible in current commercial systems, the prototype system is not as economical an alternative. However, if one desires higher performance than is possible in other ways, image-composition is a feasible way to achieve it.

Defining accurate benchmarks for comparing graphics systems is an issue of heated discussion [BLAU88].

[1]See Appendix B.

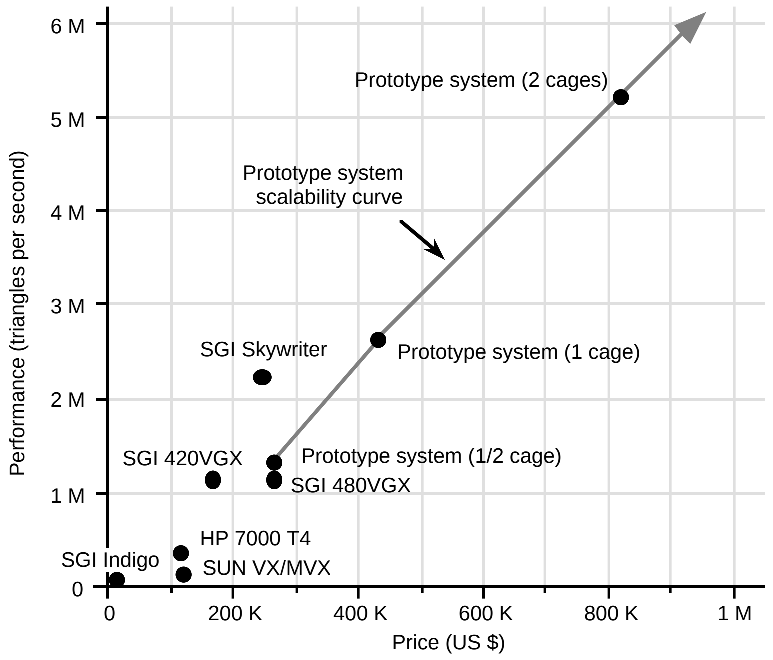


Figure 5.19: Performance vs. price for PixelFlow prototype system and representative systems in 1991.

One could argue that, if commercial systems push current architectures that do not scale to higher and higher speeds, they will come closer to fundamental limits, such as clock speed, maximum packaging density, etc., and their performance vs. price will decrease at higher performance levels. This has already happened with conventional computers: supercomputers, which require exotic packaging and technology, have lower performance vs. price than RISC workstations, for example.

The prototype system, on the other hand, has a performance-cost curve that asymptotically approaches a line of constant slope. The slope is the performance vs. price of a single renderer. The curve begins initially below this line because the prototype system contains fixed costs that are independent of system size, such as shaders and a frame buffer. For image-composition architectures to be cost-effective, the performance vs. price of a single renderer must be high, and the system must contain enough renderers to amortize the fixed costs of the system.