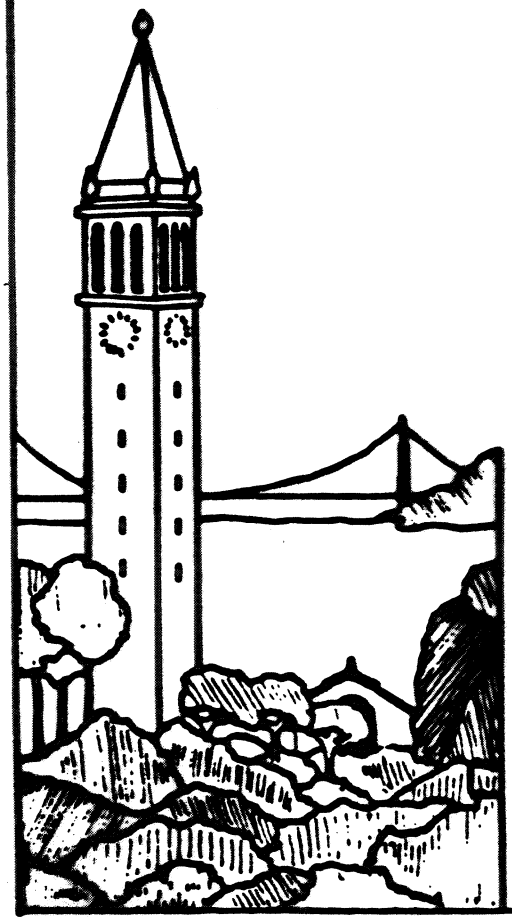
Performance Evaluation for Computer Image Synthesis Systems

Ricki Blau



Report No. UCB/CSD 93/736

March 1993

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Performance Evaluation for Computer Image Synthesis Systems

Ricki Blau

Computer Science Division
University of California
Berkeley, California 94720

ABSTRACT

This dissertation applies performance analysis to the problem of computing complex three-dimensional images. First, it identifies factors that affect the cost of image synthesis and characterizes the complexity of realistic images. Four categories of performance factors are defined: scene characteristics, viewing specifications, rendering parameters, and the computing environment. This classification provides a framework for discussing image complexity and designing performance experiments. The complexity of several complex images from an actual animation workload is described in detail.

A methodology is presented for the construction of reproducible and controllable performance measurement experiments. To measure the performance of a rendering system, an experimenter provides a set of test data, including image specifications. The dissertation describes a portable tool that generates test cases, varying the scene characteristics and viewing specifications under the control of a set of parameters. This model generator, Mg, has been implemented for two different rendering systems. Its test cases have been used to detect performance differences between the two systems and to evaluate the effects of varying the scene characteristics.

Finally, we address the workload partitioning problem for a MIMD rendering system. A simple, low-overhead adaptive algorithm balances the workload effectively on a sixteen-node rendering accelerator. The algorithm uses the rendering time observed for one frame of an animated sequence to predict costs for the next frame. The resulting cost estimates can be used by a second algorithm to divide the work among the available processing nodes. Our cost estimates are approximate, but they are obtained with very little overhead. The net result is an improvement of thirty to eighty percent over the previous load balancing schemes for production-quality rendering of animated sequences. An analysis of several competing schemes demonstrates that tradeoffs between balancing the load and preserving locality are a key consideration in the design of a parallel rendering system.

Performance Evaluation for Computer Image Synthesis Systems

by

Ricki Blau

Acknowledgements

I would like to thank my advisor, Alan Smith, for his making it possible for me to pursue my own topic, for his help in finding funding, for his constructive criticism, and for his patience. My other readers have helped with their interest in my research. In particular, I would like to thank Ed Catmull for his help in focusing on the goal of controlled variation in Chapter 4, Brian Barsky for making Chapter 2 both more rigorous and more clear, and Michael Cooper for his enthusiastic teaching, his friendship, and his getting me into this in the first place. While not an official reader, Scott Baden read Chapter 5 very carefully. I am grateful to him for his comments on the chapter, for many discussions about load balancing, and for his lead in examining the factors contributing to lost efficiency.

I would like to thank Pixar for the use of hardware, software, and data. Many people at Pixar, past and present, have helped make my research possible. Bill Reeves first suggested the load balancing problem. He has spent much time preparing models for my experiments, answering questions, and retrieving files that I should have backed up (all great ways to avoid housework). Tony Apodaca, Darwyn Peachey, Rob Cook, and Jim Lawson answered innumerable questions about the inner workings of the rendering systems Reyes, Prman, and Opal. Discussing load balancing with Pat Hanrahan helped me see my way through to the essence of the problem. I'd like to thank Don Conway, Deirdre Warin, and Craig Good for help obtaining slides, prints, and videos.

The images in Figures 3.3 were created by Pixar's Animation Division. I am grateful to Pixar for permission to reproduce these images and for allowing me to create illustrations for Chapter 5 using Pixar's models and software. I would also like to thank Paramount Pictures Corporation and Amblin' Entertainment for permission to reproduce Figure 3.3(b) from the film *Young Sherlock Holmes*.

Many people have made my stay in the department a much warmer experience, including Jean Root, Teddy Diaz, Liza Gabato, and Kathryn Crabtree. Sheila Humphreys deserves special thanks for her friendship and and encouragement, and for her work on behalf of women in science and engineering. Scott Baden, Paul Hansen, Rafael Saavedra, and Barb Tockey helped make my offices places where I enjoyed working. Doloros Anderson, Brandi Blair, and Karen-Sue Taussig allowed me to work knowing that my children were safe and happy.

Thanks to my parents, Rita and Norman Blau, for a lifetime of encouragement and to Dr. Blau for her example. And, most of all, big hugs for Julia, Oliver, Ian, and Bill.

This work was supported in part by the Computer Measurement Group through a CMG Fellowship, by a California Fellowship in Microelectronics, and by Philips Laboratories/Signetics, the National Science Foundation under grants MIP-8713274, MIP-9116578 and CCR-9117028, by NASA under Grant NCC 2-550, by the State of California under the MICRO program, and by the Digital Equipment Corporation, International Business Machines Corporation, Apple Computer Corporation, Mitsubishi Corporation, Sun Microsystems, and Intel Corporation.

Table of Contents

List of Figures

List of Tables

1
Introduction

The research described in this dissertation applies performance evaluation to the field of realistic image synthesis. Realistic computer graphics has many applications in design, engineering, entertainment, scientific visualization, medicine, and advertising. Advanced image synthesis systems produce color images of complex, three-dimensional scenes using naturalistic shading techniques. Researchers in the forefront of image synthesis have been extending the capabilities of computer graphics to simulate visual reality, seeking a quality some call "photo-realism." Two key properties of realistic images are geometric complexity and a naturalistic simulation of surfaces and illumination. The geometry of a scene describes the shapes and positions of objects, while sophisticated shading algorithms simulate the interactions between light and the surface of objects. Realistic images tend to include many objects, and the objects themselves can be very intricate. Advanced computer graphics algorithms can simulate visual qualities such as transparency, reflection, refraction, shadows, motion blur, depth-of-field and complex surface textures.

Unfortunately, high-quality image synthesis is computationally expensive. Nature creates high quality images by sending many light rays, traveling at the speed of light, in parallel to the viewer or camera. Computers have to generate each point in the image separately by sequential computation. By these standards, photo-realism will, obviously, be very computationally intensive, and computers will never create images as fast or as well as nature. While added detail and sophisticated algorithms lead to greater realism in computer graphics, they also increase the cost of computing an image. Applications that demand motion, or animation, intensify the problem, because the system must compute from ten to thirty different frames for each second of animation that the user sees. The challenge facing researchers and industry is to offer more realism and image complexity in higher speed, or even interactive, graphics systems. In the past, realistic image synthesis was usually limited to research labs. Researchers often reported using hours, or even days, of computer time to produce a single image. Now, technological advancements and the maturation of image synthesis algorithms have put realistic image synthesis and animation into the reach of a wide range of users, who rely on graphics as a tool in everyday work. These users need efficient graphics tools, and they want to be able to compare the performance of different systems and algorithms. Performance analysis tools and methodology are needed to design and evaluate practical algorithms and systems for realistic graphics.

Because photorealistic graphics requires a great deal of computation, there is a lot of interest in using multiprocessor systems to produce images more quickly. To use a multiprocessor effectively, we must first solve the workload partitioning problem: to divide the work into an efficient set of sub-tasks that can be distributed among the processing units. For the multiprocessor to perform efficiently, the workload partitioning algorithm must (1) distribute the work evenly among the processors, and (2) avoid introducing additional overhead in the way it subdivides the task.

This dissertation explores three topics in the application of performance evaluation to realistic image synthesis. First, it identifies factors that influence the cost of image generation and characterizes the complexity of realistic images. Second, it describes a methodology and tools for reproducible, controlled graphics performance experiments. Third, it presents a workload partitioning scheme for a multiprocessor graphics architecture and compares its performance against several existing schemes.

A computer-generated image is the visual representation of a set of object descriptions, or *models*. A model database specifies the geometry of the objects, as defined by three-dimensional coordinates, and it describes the appearance of the objects' surfaces. The final image is composed of a two-dimensional array of individual picture elements, or *pixels*. To produce an image, the three-dimensional scene description is mapped onto the two-dimensional coordinate system of the screen during a process called *rendering*. Rendering performs geometric processing, determines which objects are visible from the specified viewpoint, and calculates the intensity of the surfaces in the image.

System designers must make compromises, because no attainable system will generate arbitrarily complex images in real time. Historically, there have been two approaches to the tradeoffs between image complexity and compute time, one emphasizing speed and the other emphasizing realism. Interactive systems aim for real-time speeds, at, or close to, the real time video rate of thirty frames a second. They produce pictures that are as complicated as possible given the time constraints. In contrast, image quality and complexity take precedence in the field of realistic image synthesis. The primary goal is to achieve the desired image qualities, and performance goals are secondary.

This dissertation emphasizes the latter approach, that is, the problem of realistic image synthesis. It considers systems that favor quality over speed in the trade-offs noted above. In particular, it concentrates on the process of rendering, which transforms the three-dimensional specifications of a scene into a shaded, two-dimensional image.

Algorithms for rendering realistic images are described in Chapter 2. It summarizes the techniques of image synthesis, concentrating on the themes of image complexity and rendering costs. It also examines the performance of rendering algorithms and the effect of image characteristics on their costs. In this chapter, I define four categories of performance parameters: scene characteristics, viewing specifications, rendering parameters, and the computing environment. Scene characteristics describe the inherent properties of the model, including its geometry, illumination, and surfaces. Viewing specifications give the position and direction of the viewer, as well as a model for the eye or camera. Rendering parameters are independent of the scene, but they influence the execution of the rendering system or the way that the model is mapped to the two-dimensional image. The computing environment includes the hardware and the systems software that are external to the rendering environment. Throughout the dissertation, this classification provides a framework for discussing image complexity and designing performance experiments.

The major tasks of a rendering system include geometric processing, visibility determination, antialiasing, shading, and texturing. For each task, Chapter 2 describes approaches to the problem and discusses the factors that tend to influence performance. Because some performance issues depend on the global structure of the rendering system, a set of examples illustrates different approaches to organizing a rendering system. An important property that influences the performance of rendering systems is *coherence*, or the tendency for images to be similar in a spatial or temporal locality. Many efficient algorithms take advantage of coherence to reduce the cost of calculations or to improve the locality of memory and disk references.

Having established the background, the dissertation turns to three topics in the performance of image synthesis systems.

First, it presents a qualitative and quantitative characterization of the workload. Chapter 3 examines two sets of data. The first is extracted from the computer graphics literature, and shows trends in image complexity over the past twenty-five years. The second describes the characteristics of several complex images taken from an actual computer animation workload; these data were obtained by examining the model specifications, by instrumenting and profiling a sophisticated rendering system, and by analyzing images.

Although we informally talk about "image complexity," we are really interested in the complexity of the input to the renderer. The workload of a rendering system has two components. The first

component includes the data needed to specify an image: the scene description, the viewing parameters, and the motion in an animated sequence. The second component consists of system-specific parameters that control the execution of the rendering system.

The dissertation's next topic is performance measurement, which is considered in Chapter 4. To measure the performance of an image synthesis system, we observe the resources it requires to generate an image. The image is specified by a database with a detailed model of the scene, controls for viewing the scene, and controls for displaying the image. Together, the model and controls form a test case, or benchmark. The chapter describes an approach to constructing reproducible, controlled performance experiments. In this model, the experimenter varies a performance parameter while holding other factors constant. Chapter 2 groups performance factors into four categories, and Chapter 4 uses this categorization to describe corresponding types of performance experiments. In a performance experiment, we can vary the scene characteristics, the viewing specifications, the rendering parameters, or the computing environment. For example, a benchmarking effort that compares different computer systems is an experiment that varies the computing environment. Because the model's characteristics and the viewing specifications affect the performance of rendering algorithms, other interesting experiments vary the scene definition.

Chapter 4 describes a portable tool, *Mg* (*m*odel *g*enerator) that creates model data for performance experiments. Mg varies the scene characteristics and viewing specifications under the control of a set of parameters. The model generator consists of programs that generate the scene specifications, general utilities, and an output library. To port Mg to a new system, a programmer modifies a small set of procedures that output the scene specifications. Mg differs from a static benchmark suite, because the experimenter can adjust the complexity and characteristics of the scenes to simulate different workloads or to evaluate specific aspects of a system's performance. The interface supports geometric features such as parametric patches, quadric surfaces, polygons, and nested transformation matrices. The lighting and shading procedures support multiple distant and local light sources, texture mapping, reflections, and transparency with refraction. Chapter 4 describes Mg's implementation on two different rendering systems and experiments that demonstrate its use.

Finally, we addresses the workload partitioning problem for a multiprocessor rendering system. Chapter 5 explores the performance of a class of algorithms based on a spatial subdivision of the image plane, and describes a simple, low-overhead load balancing scheme designed for rendering animated sequences. The algorithm uses the rendering time observed for one frame to predict costs for the next frame. The resulting cost estimates can be used by a second algorithm to divide the work among the available processing nodes. The algorithm depends on frame-to-frame similarities, or coherence, in the input. Compression schemes for digital video also exploit frame-to-frame coherence with success. Our cost estimates are approximate, but they are obtained with very little overhead. The net result is an improvement of thirty to eighty percent over the previous schemes for production-quality rendering.

Chapter 5 analyzes the performance of eight related spatial subdivision schemes and examines three factors in the loss of multiprocessor efficiency: imbalance, contention for disk accesses, and reduced coherence (or, locality). There is an inherent conflict between load balancing and maintaining coherence. To balance the load, we divide the work into smaller units, but by dividing the work we lose coherence. The partitioning scheme described in this chapter balances the load reasonably well, while maintaining much of the workload's coherence.

Chapter 6 concludes the dissertation with a summary and suggestions for future research.

2
Image Synthesis Techniques

"Unfortunately, the price of increased realism is a huge increase in computation costs."
Turner Whitted [Whit82]

2.1. Introduction

A realistic computer-generated image is the visual representation of some three-dimensional *scene*. In contrast to the scene, the *image* is two-dimensional, an array of discrete raster display elements called *pixels*. The scene is described by a set of object descriptions, or *models*, in a model database. Creating an image requires three major steps: modeling, rendering, and display (Figure 2.1). Modeling is the process of specifying the geometry and the visual properties of the scene. Rendering transforms the three-dimensional model into its two-dimensional representation, as seen from a specified viewpoint and according to a given model for the viewer. The rendering algorithms project the scene's geometry onto the image plane, decide which objects are visible, determine the colors reflected by the surfaces, and assign a color to each pixel. The final step is to display the image. The image can be stored in a special-purpose memory, called a frame buffer, and converted from a digital format into analog signals by a dedicated display processor. Alternatively, the image may be recorded directly onto film or video, stored in main memory, or written to a file for later display. In real-time or interactive graphics systems, the rendering system and the display hardware must cooperate closely. With photorealistic graphics, the interface between rendering and display can be looser.

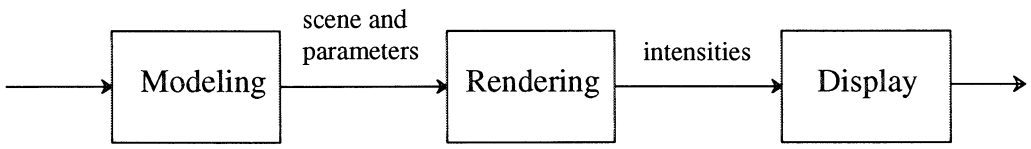


Figure 2.1. Image synthesis pipeline.

This chapter summarizes the techniques of image synthesis. It has three goals: to provide a background for readers who are not familiar with image synthesis,[1] to define the terminology used in this thesis, and to identify the factors that affect the cost of rendering an image. Because the characteristics of the models bear significantly on rendering costs, this survey covers both modeling and rendering. Image display is a separate problem, which we will not consider in detail.

Realistic image synthesis is an active research area, and new algorithms are constantly being developed. This survey concentrates on the general principles of image synthesis and aspects that are related to performance. In a previous survey, Whitted [Whit82] discussed the processing requirements for rendering, but there have been many advances in realistic graphics since then. Two more recent

[1] Throughout this thesis, I will use footnotes to discuss graphics technicalities, which are not essential to the main text.

surveys take alternative points-of-view: Amanatides emphasizes the pursuit of realism [Aman87], and Dennis summarizes key algorithms [Denn90]. Several textbooks provide more complete explanations of the basic methods of computer graphics, including the comprehensive and up-to-date work of Foley et al. [Fole90].

Two tasks dominate the cost of rendering a scene: visibility and shading. The visible surface algorithm determines what parts of the objects are visible, and therefore are to be displayed. Visibility is a geometric problem, and the performance of visible surface algorithms depends mainly on well-defined geometric quantities, such as the number of objects in a scene. The shading algorithm determines color values for the surfaces in the scene. It must consider such information as the color of an object, the reflectance properties of its surface, its location, its orientation, the light sources, and any textures that may cover the surface. The most realistic algorithms also take into account the location, orientation, and reflectance properties of the other objects in the scene. Naturally, the color can vary over the surface of an object. The number and complexity of the geometric primitives influence the size of the problem, but shading performance is not explained by geometric quantities alone. The computational effort also depends on the characteristics of the surfaces, which are hard to quantify.

In rendering, there are important tradeoffs between realism and efficiency. Realism is subjective, but we can identify properties of images that contribute to our perception of realism. One essential property is the realistic simulation of surfaces and light sources, and of the interactions among them. A second important property is a sense of geometric complexity. A third property is a careful rendering that reduces the distortion introduced by displaying the image as a grid of pixels. In general, the pursuit of these properties raises the computational cost. Realistic shading and illumination tend to require complicated algorithms. Increasing the geometric complexity adds to the size of the problem, and thus increases the number of operations that must be performed for any fixed set of algorithms. There are several approaches to reducing display artifacts. Some complicate the rendering algorithms, some mainly increase the size of the problem, and others do both.

Solutions to the visible surface problem can be evaluated objectively. Given a scene and viewing parameters, a solution is either correct or incorrect. A correct solution to the visible surface problem is essential to any shaded image. In the early days of raster graphics, the visible surface problem consumed most of the rendering time and most of the research and development effort. In the mid-seventies, Sutherland, Sproull, and Schumacker [Suth74] published a landmark survey of the problem, analyzing ten distinct algorithms and offering a taxonomy of approaches to its solution. By now, many correct and practical solutions have been implemented. Joy et al. [Joy88] present a more recent survey, which includes Grant's expanded taxonomy of visible surface algorithms [Gran87].

In comparison, shading and illumination are open-ended problems. Graphics researchers are still learning how to make surfaces look real. Because it is too expensive to simulate the actual physics of vision, most algorithms merely attempt to simulate a realistic appearance. Hall, discussing the evolution of illumination models for computer graphics [Hall89, especially Chapter 4], offers a classification of algorithms. The earliest algorithms, beginning in the late 1960's, supported simple empirical models. Starting in the mid-1970's, transitional models added detail and realism, drawing upon the principles of physics and optics to enhance the underlying empirical models. Analytical models began to appear in the early- to mid-1980's, attempting to simulate physical processes more closely. Hall's final category, the hybrid approaches, combine elements of the transitional and the analytical models.

When more realistic shading and illumination techniques are developed, they tend to complicate the work of shading a scene and increase the proportion of rendering time devoted to shading. However, the factors that influence shading costs are less well-defined and dependent on a wider variety of workload characteristics.

This chapter examines the factors that influence the performance of rendering algorithms and the effect of image characteristics on the cost of rendering. It considers general classes of performance parameters; a few affect the performance of most graphics systems, while others are important only for

certain classes of algorithms or implementations. We can group these performance factors into four classes: scene characteristics, viewing specifications, rendering parameters, and the computing environment. The scene characteristics specify the inherent properties of the model, including its geometry, illumination, and surfaces. Viewing specifications include the viewpoint, the view direction, the angles of view in the horizontal and vertical directions, and a model for the eye or camera. Rendering parameters, such as the image resolution, are independent of the scene; they influence the execution of the rendering system or the scene's representation in the image. The computing environment undeniably affects rendering performance, but it is external to the rendering environment. So, this chapter will mostly ignore the computing environment.

For each of the major rendering tasks, this chapter will describe approaches to the problem and discuss the factors that tend to influence performance. Some performance issues depend more on the global structure of the rendering system, so a set of examples will illustrate different approaches to organizing a rendering system.

This chapter will emphasize compute time costs and only occasionally will describe how memory costs have affected the development of algorithms. Nevertheless, many time-efficient rendering algorithms require substantial amounts of main memory.

2.2. Scene Modeling

Two types of information describe the objects in a scene: geometric specifications and surface characteristics. (Table 2.1). Models are usually created in a separate process preceding image generation. Typically, a special-purpose modeling language is used to describe the scene.

The shape and location of an object is specified mathematically by a set of three-dimensional modeling primitives. Some objects can be represented exactly by simple geometric primitives, such as a sphere, a torus, or a set of polygons. More complex surfaces must be approximated. Early computer graphics systems approximated the surface of objects with sets of planar polygons. Although polygons are still often used, curved surfaces are represented more smoothly by parametric curves and surfaces. Different mechanisms can be used to generate and combine the basic mathematical elements. One method, called constructive solid geometry (or CSG), combines geometric solids with set operations. Another approach is a sweep representation, which defines a three-dimensional object by sweeping a two-dimensional shape through space. Among the other possibilities are boundary representations, which specify solids with the vertices, edges, and faces that define their boundaries.

In applications such as engineering, design, and scientific visualization, the model is derived from real data. In other applications, models can be created by hand, often by a designer at a graphics workstation. For highly complex scenes, it is too tedious to specify every detail by hand. Instead, procedural modeling algorithms follow a set of guidelines to generate detailed object specifications. Deterministic algorithms are appropriate for objects with regular geometry, such as a bicycle wheel with many spokes. Stochastic algorithms can be used to create objects with more variation, such as models of natural phenomena. Properties such as length, orientation, or even shape and color are taken from a distribution specified by the scene designer. Fractals [Four82] and particle systems [Reev83] are two approaches to stochastic procedural modeling.

Some rendering algorithms can interpret a variety of primitives directly. Many others require all objects to be defined in terms of a single intermediate representation, such as planar polygons. This intermediate representation may be hidden from the scene designer by a algorithms that convert other primitives to the required form.

To complete an object's description, the designer assigns surface characteristics such as color, transparency, and textures. Information about the material properties and visual characteristics are stored in the model database along with the geometric data. Shading is computed according to an illumination model, which requires light sources to illuminate the scene. Realistic scenes can contain many light

Category		Information Specified
The Scene	Object Geometry	shape dimensions position orientation
	Surface Properties	color texture optical properties
	Light Sources	number position direction shape color
The Viewer	Viewing Specifications	position direction field of view model for camera or eye

Table 2.1. Image specifications.

sources with varied characteristics, such as color and shape.

The scene itself is specified by the object descriptions and the lighting. Viewing specifications are external to the scene, but they are required for image creation. The model for the camera or eye is often created along with the model for the scene, so Table 2.1 includes viewing specifications.

For computer-generated animation the scene specifications are dynamic over a sequence of individual *frames*. An animator or an algorithm defines the movement of objects within the scene. The shape or appearance of objects, the viewpoint, and the lighting conditions can change during an animated sequence.

2.2.1. Modeling Costs

Ordinarily, modeling is independent of rendering, so its costs are not directly relevant. However, some modeling operations are performed on-the-fly by rendering systems.

For example, the initial modeling phase specifies only algorithms and general parameters for procedural models. Some typical parameters include the number of objects to be generated, their mean size, and the variability of their sizes. To save space and time, the model is not fully instantiated until it is needed. The detailed specifications for procedural objects may be generated by a preprocessor or at rendering time.

Some rendering systems operate on only one type of geometric primitive, such as planar polygons. The renderer may support a wider variety of primitives by converting them on-the-fly to the required intermediate representation. For example, bicubic patches may be subdivided recursively until the subpatches are nearly flat or until they are smaller than some maximum screen size. The resulting subpatches are then approximated by a simple polygon.

The complexity of these modeling functions depends on the type of modeling operations required, the number of objects involved, their size, and their geometric complexity. The following section

discusses measures of geometric complexity in more detail. Viewing specifications also influence the cost of on-the-fly modeling, because less detail is visible when objects are distant. Adaptive algorithms for subdivision and procedural modeling adjust the level of detail according to the object's size on the screen, which changes with the viewing distance. When an object is viewed from nearby, a detailed model is necessary. When the same object is viewed from a distance, a simpler model can represent it adequately.

Table 2.2 summarizes the costs of on-the-fly modeling operations that may be performed by the rendering system.[2]

Category	Cost Factors
Scene Characteristics	type of modeling needed number of objects geometric properties of objects
Viewing Specifications	viewing distance

Table 2.2. Modeling cost factors. The list is limited to modeling operations performed at rendering time.

2.3. Geometric Operations and Complexity Measures

Geometric computation is the foundation of computer graphics and occurs in all stages of the image synthesis pipeline. Before describing any rendering algorithms, we will discuss some important geometric concepts and cost factors that apply throughout the rendering process.

2.3.1. Coordinate Systems and Transformations

Typically, a series of coordinate systems, or spaces, is used during the image synthesis process. A matrix multiplication transforms the data from one coordinate system to the next. A sequence of transformation matrices can be composed into a compound transformation matrix by pre-multiplying. With the compound matrix, a single matrix multiplication applies the entire sequence of transformations to each data point. In the image synthesis literature, individual coordinate systems have been given many names, which are sometimes conflicting. This thesis uses one name for each coordinate system. Foley et al. provide a comprehensive list of coordinate system names [Fole90, pages 279-281].

For convenience, each object, or part of an object, can be modeled in its own *local* coordinate system. For more complicated objects and scenes, all of the primitives are transformed into a single *world* coordinate system. This is accomplished by a modeling transformation that composes a sequence of operations such as rotation, translation, and scaling.

The local and world coordinate systems are view-independent. After the model is transformed into *eye* space, coordinates are defined relative to the viewer's position. Typically, the viewer looks in the positive z direction, with x to the right and y up. Next, a viewing transformation projects the scene onto the *screen* coordinate system. This transformation depends on the field of view. Its x and y coordinates describe the location of pixels on the screen. They may be integers corresponding to the image

[2] It can be argued that some applications of texture maps serve a modeling function. Some examples of quasi-modeling applications are varying the surface texture, varying the surface normal, or describing surface displacements. Because these texturing applications are more commonly considered part of shading and because their costs are incurred during shading, texturing is considered along with shading in Section 2.6. Other texture map applications simulate global illumination effects, such as reflections or shadows. They clearly serve a shading function.

resolution, or they may be normalized values with the range [0,1] describing the extent of the screen. Thus, integer arithmetic or lower-precision calculations are often suitable. Many rendering algorithms determine visibility in screen space, so depth information is retained and the model is still described in a three-dimensional coordinate system. For realistic images a perspective projection is used, so that objects appear smaller as their distance from the viewer increases. The perspective transformation preserves the depth order, straight lines, and flat planes of the original scene; visible-surface algorithms can therefore operate on the perspective data.

Eventually, the scene is mapped onto *device* space, the two-dimensional coordinate system of the display. Because the perspective transformation has already been applied, a simple orthographic projection of the transformed scene onto the screen is the equivalent of a perspective projection of the original scene. The orthographic projection drops the z coordinate and transforms x and y directly into display addresses.

Table 2.3 summarizes this sequence of coordinate systems. All are three-dimensional until the final transformation into two-dimensional device coordinates. Most algorithms use only some of these coordinate systems. Some, such as ray casting algorithms, operate on world coordinates. They project rays into the scene from discrete points on the screen. The nearest object that intersects a ray is visible at the corresponding sample point. In theory, many classical algorithms follow the progression of Table 2.3. They project objects onto the image plane and operate on them in screen space. But in practice, many of these algorithms combine transformations and do not explicitly use every coordinate system.

Name	Description
local	many local spaces for subsets of the scene; often hierarchical
world	describes the complete scene; independent of viewpoint
eye	relative to viewpoint and view direction
screen	screen coordinates (may be normalized); after perspective transformation
device	device-specific addresses

Table 2.3. Coordinate systems.

2.3.2. Geometric Cost Factors

The cost of rendering realistic images is dominated by shading and the visible surface problem. Coordinate transformations and the geometric aspects of runtime modeling typically consume less compute time.[3] However, there is a geometric aspect to all stages of image synthesis, and the geometric complexity of the scene affects the costs of operations throughout the rendering process.

In 1974, Sutherland, Sproull, and Schumacker published an influential analysis of the visible surface problem [Suth74]. They identified a set of "statistical measures of environmental complexity" that affect visible-surface performance. These statistics characterize the geometric complexity of the models and, to some extent, the spatial distribution of objects. Because algorithms for polygons dominated computer graphics in 1974, the complexity measures are based on faces and edges. These complexity measures are a good starting point, but we must now include a broader range of primitives.

The list of geometric complexity factors in Table 2.4 is not exhaustive, but it summarizes the primary categories. Unlike Sutherland, Sproull, and Schumacker, it separates the intrinsic characteristics of the model from the view-dependent characteristics of its projection into screen space. In addition, it recognizes geometric entities that the rendering system may create by subdividing primitives or converting objects to an intermediate form.

[3] Some profiling results are given in Chapter 3 and by Schoeler and Fournier [Scho86].

Category	Cost Factors
View-independent	types of modeling primitives number of primitives size of primitives relationships of primitives, e.g. intersection 3D spatial distribution of primitives variability of these properties
View-dependent	screen size of primitives sum of screen area covered by all primitives number of relevant primitives size of relevant primitives number of visible primitives size of visible primitives screen area covered by visible primitives 2D spatial distribution of primitives in the image depth complexity variability of these properties
Rendering-dependent	number of rendering primitives size of rendering primitives distribution of rendering primitives

Table 2.4. Geometric complexity factors. Rendering factors describe internal primitives generated by the rendering system.

The view-independent factors in Table 2.4 describe the intrinsic properties of the complete scene. A scene might contain only polygons, or it might contain many types of geometric primitives. Algorithms that manipulate primitives vary in complexity according to the geometric properties of the primitive. Other factors include the number of primitives of each type and their dimensions. Relationships among the objects can also affect costs. Sutherland, Sproull, and Schumacker cite intersecting polygons as an example; some algorithms split polygons or create an implied edge at the line of intersection.

Another set of complexity statistics are view-dependent. Given the viewing specifications, we can determine which objects are visible or potentially visible. Some invisible objects can be eliminated with simple *culling* operations, before the system applies the full visible surface algorithm. The aim of culling is to quickly eliminate surfaces that cannot possibly be visible, such as objects located behind the viewer or the back-facing sides of opaque surfaces. The cost of some rendering systems is dominated by the characteristics of the *relevant* primitives that remain after culling.

Many objects survive an initial cull but are still wholly or partially outside of the screen area. A *clipping* algorithm determines the intersection of a primitive's projection with the screen. If a primitive straddles the edge of the screen, the clipping algorithms trims it and discards the portion that falls outside of the image area.

Normally, only a subset of the relevant primitives are actually visible in the image. The performance of certain algorithms is sensitive to the characteristics of the visible primitives. For example, most rendering systems shade only visible objects.

Given the viewing specifications, we can also determine the dimensions of primitives, in pixels, when projected on the screen. This *screen size* is an important cost factor for many rendering systems. The complexity of many algorithms is proportional to the number of pixels covered by the objects in the scene. Some algorithms consider an object's apparent size when deciding the level of detail to generate during procedural modeling or to process while rendering.

A primitive that covers several pixels of the display is typically specified by only a few coordinates, polygon vertices for example. In a process called *scan conversion*, the rendering system examines the coordinates and outputs values for all pixels covered by the primitive. Some algorithms scan convert whole primitives directly, by filling the pixels they cover. Other algorithms perform the operation indirectly. They sample the scene at each pixel, finding the visible objects and determining the proper shade.

Different expressions more closely describe the performance factors for different algorithms. We will consider a variety of size statistics, even though some are easily derived from others. One example is the depth complexity, which measures the average number of primitives that project onto each point of the image. Assume that the objects have been clipped to the screen, leaving only primitives whose projections are contained within the screen area. Then, let n be the number of primitives and a_i be the screen area of primitive i. The total screen area of all objects, visible or hidden, is defined by:

$$A_t = \sum_{i=1}^{n} a_i$$

If A_v is the area of the screen covered by visible objects, we can calculate the depth complexity, D, as the ratio:

$$D = \frac{A_t}{A_v}$$

The final category in Table 2.4 covers cost factors that vary with the rendering system. These factors describe properties of any intermediate primitives that the renderer may create internally, for example as the result of polygonalization. Their values may depend on run-time controls that the system provides.

Sutherland, Sproull, and Schumacker focused on averages and sums to describe geometric complexity. Many parallel graphics systems use spatial or screen subdivision, assigning regions of world space or of screen space to different processing nodes. Other rendering systems use spatial or screen subdivision as part of a divide-and-conquer strategy to improve uniprocessor performance. These approaches are most efficient when the sub-problems are approximately equal in complexity. Thus, the distribution of primitives in 2D and 3D space and the variability of the complexity measures have become increasingly important.

The performance of a rendering system is also sensitive to its ability to exploit locality in the data. The tendency for images to be similar in a spatial or temporal neighborhood is called *coherence*. This property was discussed extensively by Sutherland, Sproull, and Schumacker. On the image plane, coherence is exhibited by the similarities among contiguous locations, which typically display the same object or similar objects. Furthermore, the shading and texture of an object is usually similar at adjacent locations on the screen. Spatial coherence is also seen in the three-dimensional relationships among objects in a scene, and temporal coherence is seen in consecutive frames of an animated sequence. Algorithms that capitalize on coherence can reduce the cost of many calculations and improve the locality of memory and disk references. It is often possible to avoid repeating complex calculations by performing simple incremental adjustments. The degree of coherence is reflected in many of the complexity measures, such as the number of primitives, their types, and their sizes both in world space and in image space. Images have become more complex since 1974, and increases in complexity tend to reduce some forms of coherence. For example, a simple image might display a few large primitives. In this case, the typical primitive would cover many pixels. In contrast, complex images tend to display large numbers of small

primitives; the average primitive may be smaller than a pixel.

In summary, geometric cost factors characterize the number, type, and sizes of primitives, their variability, and their spatial distribution. We can apply these complexity measures to the modeling primitives, to their projection on the screen, to the visible subset of their projection, or to internal rendering primitives. We must choose an appropriate set of statistics to analyze any specific renderer, since the effect of the geometric cost factors depends on the structure of a rendering system and the algorithms that it uses.

2.4. Antialiasing

The scene is modeled by continuous, high-precision primitives, but the final image displays a grid of discrete dots. A high-quality rendering system must avoid or reduce the undesirable visual artifacts that will arise if the mapping from the continuous to the discrete is not done carefully. The problem of avoiding these visual defects is called antialiasing.

The appearance of jagged edges in place of smooth silhouettes is a classic defect of simple rendering algorithms. But the potential problems are even more extensive, occurring in the interior of surfaces as well as at the edges. Small or thin objects can fall into the cracks between pixels, although they should make some contribution to the image. Long, thin objects may appear at some pixels and disappear at others, taking on a beaded or segmented look. Sub-pixel detail, such as textures or highlights, may be represented inaccurately. More subtle problems surface in animation. For example, motion may appear jerky rather than smooth. A small object can appear in some frames and not in others, depending on how it intersects pixel boundaries, and it will appear to ''twinkle'' or ''pop'' in and out of the animated sequence.

Signal processing theory explains that these artifacts are due to sampling the high-frequency, continuous model at an inadequate low frequency. In other words, the resolution of the image is too coarse to accurately represent all the detail in the model. This leads to the problem called aliasing. Antialiasing is the process of reducing, or avoiding, the defects introduced by aliasing. In image synthesis, the display frequency is dictated by the grid of pixels, whereas the frequencies represented in the model may be arbitrarily high. Crow discusses the aliasing problem in computer graphics and reviews some early antialiasing algorithms [Crow77, Crow81].

2.4.1. Approaches to Antialiasing

Let's view each pixel as representing not just a point, but a finite area. In an antialiased image, the color for a pixel is determined by contributions from all objects that are partially visible within its area. The contribution of an object to the pixel's intensity is weighted by the proportion of the pixel's area in which it is visible. Figure 2.2 illustrates this concept.

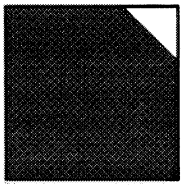


Figure 2.2. Weighted intensity contributions. If the white polygon is closer to the viewer than the dark polygon, a simple algorithm without antialiasing might display a completely white pixel. The most accurate shade for this pixel would be a dark gray that is a weighted blend of the two shades. We would, in fact, prefer the color of the darker polygon to white.

The problems of antialiasing and visible surface determination are closely tied. Antialiasing requires information about all objects partially or wholly visible in any part of the pixel. Ideally, this information is generated by the visible surface algorithm. Simple algorithms that consider only the nearest visible object for each pixel do not, in general, provide adequate support for antialiasing. Antialiasing algorithms use different approaches to obtain either approximate or more exact information about partial pixel coverage and they produce images of varying quality. Some algorithms continue to consult the full three-dimensional object descriptions. Others extract only some approximate information, such as the percentage of pixel coverage. Three approaches to antialiasing are discussed below.

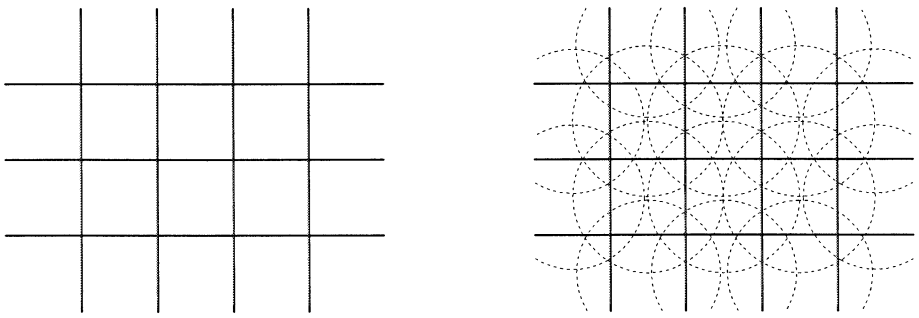
1. Post-processing strategies blur the computed image to make some visual artifacts less apparent. For example, image processing algorithms can smooth jagged edges. Unfortunately, blurring reduces the quality of the image and destroys information. These methods tend to be ineffective with surface details such as textures or highlights on glossy surface, and we will not consider them further.

2. Area Sampling, proposed by Catmull in 1974 [Catm74], supports highly accurate antialiasing. The idea of area sampling is to integrate the contributions of all visible objects or fragments over the screen area associated with each pixel. The intensity of a pixel is set to the average over its region. An object's contribution is determined according to a filter function, called the *kernel*. The kernel specifies the region of the image plane that affects the intensity of the pixel.

The box filter is a simple form of area sampling. The contribution of an object to the pixel's color is proportional to the area of the square that it covers. A box filter models the image as a grid of non-overlapping square pixels, as in Figure 2.3(a). The filter is unweighted, which means that an object's contribution is determined only by its area and not by its location.

For greater accuracy, objects within a larger radius can contribute to the pixel, and the contributions can be given more weight closer to the pixel's center. Weighted filters model pixels more like the dots on a video monitor. In Figure 2.3(b), for example, a pixel is represented by a circle with a diameter of two pixel widths; therefore, the regions of adjacent pixels overlap. To visualize the weights increasing towards the center of the pixel, we can imagine each circle in Figure 2.3(b) describing the base of a three-dimensional object, such as a cone. The height of the object, which represents the filter's weighting function, increases from the perimeter towards the center.

One weighted filter that produces good results is a Gaussian filter. It weights an object's contribution according to a Gaussian distribution with the origin at the pixel's center. This distribution gives greater weight to objects nearer the center of the pixel and is radially symmetric. In practice, such a distribution is truncated so that only objects within a finite radius of the pixel's center (typically about one pixel width) affect the intensity of a pixel. The extent of this region is called the filter's *support*. Feibush, Levoy, and Cook [Feib80] describe the implementation of a such a filter and the rationale for its design.



(a) (b)

Figure 2.3. Filters.

Exact analytic algorithms for area sampling tend to be computationally intensive, but cheaper approximations have been developed. For example, coverage mask strategies can give discrete approximations to an unweighted filter [Carp84, Fium83, Schu80]. A coverage mask is a bitmap that records coverage information for a regular grid of subpixels within each pixel. Another efficient approximation is the use of lookup tables for sampling with either weighted or unweighted filters [Abra85, Feib80].

3. Supersampling computes the image at a higher resolution, thereby increasing the sampling rate. This approach is generally associated with point sampling rendering algorithms, which determine visibility and intensity at discrete points. Simple point sampling algorithms select only one point per pixel, often its center. They compute the color of whichever object is visible at the sample point and assign that color to the pixel. Because of their coarse resolution, these simple algorithms tend to have many problems with aliasing. With supersampling, the renderer computes not one, but many, intensity values per pixel and then averages the sub-pixel values to calculate the pixel's intensity. The samples can be combined with either a weighted or unweighted filter.

More abstractly, point sampling is an approximation to area sampling, and Fiume shows that it converges to area sampling as the sampling frequency increases [Fium89].

Supersampling is simple to implement and reasonably effective, but it cannot solve all problems. If the scene contains patterns of sufficiently high frequency, any given sampling rate will be inadequate. A second disadvantage to supersampling is its cost. The computation time tends to increase with the sampling resolution, and satisfactory antialiasing may require a much higher resolution.

Stochastic sampling addresses these problems with an approach supported by signal processing theory. Its principles and applications to image synthesis have been described by Cook [Cook86] and by Dippé and Wold [Dipp85]. Conventional algorithms, whether they use one or many samples per pixel, sample on a regular grid of uniformly-spaced locations. Stochastic sampling differs by selecting an irregular or randomized pattern of samples, usually in combination with supersampling. Low sampling rates still introduce errors, but the advantage of stochastic sampling is that the errors appear as noise, without the objectionable visual patterns of aliasing. Various types of sampling patterns and filters may be used, but in practical implementations both sampling and filtering may be table-driven. Another advantage of stochastic sampling is that it leads to natural solutions to such problems as motion blur, penumbras, and depth of field [Cook84b].

Adaptive supersampling is another approach to lowering the cost of antialiasing. The idea is to increase the sampling rate only where the image contains high-frequency information. An early example of adaptive sampling is described by Whitted [Whit80]. He samples at the corners of each pixel and averages the intensity values; if the intensity at the corners differs from the average by more than a given threshold, the pixel is subdivided for further sampling. Lee, Redner, and Uselton consider statistical criteria for determining the sampling rate [Lee85]. Adaptive sampling can be combined with stochastic supersampling.

2.4.2. Antialiasing Costs

Because of the ties between antialiasing and visibility, their costs are closely related. Several geometric factors help determine the size of both problems, including the number of objects visible in the scene, the screen area they cover, and the average number of objects intersecting a pixel. The scene geometry is not directly important for postprocessing algorithms, which operate on the rendered image. They are affected by the number of pixels covered by visible objects and by the frequency of detail in the two-dimensional image.

The frequency of detail in the scene determines the sampling rate of adaptive algorithms and has a large effect on their performance. This detail includes not only the geometric structure of objects, but also surface features such as textures, highlights, and reflections. The frequency of detail with respect to the image is affected by the screen size of objects, which, in turn, depends on the viewing distance and

image resolution.

Antialiasing costs are also influenced by parameters that are set at rendering time. The display resolution affects the cost of both visibility and antialiasing. Antialiasing introduces a related factor: the sampling resolution, or the computed resolution for area-averaging algorithms. The choice of a filter, especially its area, further affects the cost. Section 2.3.2 discusses the geometric cost factors that affect most image synthesis operations, while Table 2.5 below lists the factors that are special to antialiasing.

In general, an algorithm that supports good antialiasing needs more information than a less sophisticated algorithm, so we can expect its memory costs to be higher.

Category	Cost Factors
Scene Characteristics	frequency of detail
Viewing Specifications	viewing distance
Rendering Parameters	filter area image resolution sampling frequency

Table 2.5. Antialiasing cost factors. This table lists factors that specifically affect antialiasing costs. The geometric cost factors discussed in Section 2.3.2 also affect antialiasing algorithms, except for post-processing algorithms.

2.5. Visibility

The visible surface algorithm examines the model to determine what is visible with respect to the specified viewpoint.

2.5.1. Approaches to the Visible Surface Problem

Visibility algorithms are often classified as either *image-space* or *object-space* algorithms[Suth74]. Image-space algorithms approach the problem from each pixel in the image. They examine the objects in the scene to determine which are visible in the area represented by the pixel. These algorithms typically operate at the precision of the image resolution, or of the sampling resolution. In general, the complexity of image-space algorithms is proportional to the product of the number of objects and the number of pixels. Object-space algorithms approach the problem by considering the relationships among the objects. They compare primitives to determine if one obscures the other, either partially or completely. These algorithms tend to operate at the precision of the model's definition, which is usually higher than image precision. In general, the complexity of straightforward object-space algorithms is proportional to the square of the number of objects. Foley et al. [Fole90, page 650] compared the complexity of image-space algorithms (np, where n is the number of objects and p is the number of pixels) and object-space algorithms (n^2): "Although this second approach might seem superior for $n<p$, its individual steps are typically more complex and time consuming, ... so it is often slower and more difficult to implement."

Visible surface algorithms, like antialiasing algorithms, may also be classified as either point sampling or area sampling. When discussing rendering performance, another useful way to categorize the algorithms is by the order in which objects are processed. We will follow this approach, describing four classes of visible surface algorithms and a hybrid category.

1. Arbitrary order algorithms process objects one-by-one, in any order. The primary example is the simple z-buffer algorithm, introduced by Catmull [Catm74], and its variants. This image-space algorithm requires a z-buffer, or depth memory, which has an entry corresponding to each pixel stored in the

frame buffer. The z-buffer is initialized to the depth of an assumed background located behind the scene. Objects are scan-converted one at a time, in any order. At each pixel, the object's depth (that is, its z coordinate) is compared to the value already in the z-buffer. If the new object is closer to the viewer, its intensity is stored in the frame buffer and its depth is stored in the corresponding cell of the z-buffer.

Because the simplest z-buffer algorithm maintains no sub-pixel information, it cannot support antialiasing. Sometimes a z-buffer is combined with supersampling to enable antialiasing. A modified z-buffer can also support antialiasing by maintaining per-pixel lists of visible objects. Other antialiasing z-buffers retain approximate information, such as the fraction of the pixel area covered by a set of objects.

The z-buffer algorithm makes a single pass over an unsorted data base, performing a constant number of operations for each pixel covered by an object. Thus, its costs rise linearly with the total number of pixels covered by all objects. The basic operations are simple, and there are no direct pairwise comparisons of objects. The algorithm is, therefore, suitable for moderately complex scenes. If the workload contains large primitives, the renderer can use incremental calculations to interpolate the depth across scan lines and from one scan line to the next.

The z-buffer requires a second large memory, but the algorithm may save some object memory. When objects are rendered in an arbitrary order, they may be discarded after processing. Antialiasing algorithms usually require extra memory to maintain more extensive information about partial pixel coverage. The extra memory can take the form of coverage masks (Section 2.4.1), sub-pixel information for supersampling, or retained object descriptions. Approximate methods may consume less memory, but at the same time they may limit image quality.

2. Screen-area order algorithms subdivide the screen and process together all objects that appear in the same region. Objects are pre-sorted, often with a cheap bucket sort on the line where the object first enters the image. One approach to screen-area subdivision is illustrated by scan-line algorithms, which examine together all objects that intersect a given scan line. Other types of subdivision algorithms partition the screen into rectangles. The main objective of these algorithms is to exploit coherence by processing objects in a suitable spatial ordering. By working on smaller sub-problems, they can often reduce the processing time for steps that are worse than linear. Spatial subdivision algorithms may also save memory; they can usually discard object descriptions after each region is processed.

Scan-line algorithms typically step along the scan line horizontally, doing all work for a pixel at one time. Classical image-space scan-line algorithms, such as Watkins' [Watk70], were developed for rendering polygonal models. They take advantage of scan line coherence and process polygons with simple incremental calculations. At each pixel, the polygon's depth and color are adjusted by a per-pixel increment to compute values for the next pixel. This optimization works for large primitives that cover many pixels, but it is ineffective for the tiny primitives that are common in complex images. This approach was later extended to other types of primitives. Blinn, Whitted, and the team of Lane and Carpenter developed algorithms for rendering bicubic patches in scan line order [Blin80].

Some screen subdivision algorithms, including classical scanline algorithms, use a fixed set of uniform regions. Other algorithms generate adaptive, rather than regular, regions. Warnock [Warn69] suggested a recursive subdivision algorithm. It divides each region along pixel boundaries into four equal subregions. If no more than one primitive is visible in the subregion, visibility is determined for the subregion. Otherwise, the area is subdivided recursively. The recursion ends when visibility is determined for all pixels, or when the subdivision reaches the sampling resolution. Dividing objects along subregion boundaries dominates the cost of this algorithm. The object-space ''cookie cutter'' algorithm of Weiler and Atherton [Weil77] also subdivides the image until a single primitive is visible. It differs from Warnock's algorithm, because it clips primitives along the boundaries of other primitives. The algorithm is based on the observation that visibility can change only at the boundary of objects. It requires complicated geometric operations that are most practical for polygonal models. In the worst case, the Weiler-Atherton algorithm could require that each primitive be compared with every other primitive. As with Warnock's algorithm, it depends on object coherence for efficiency and is most suitable for

images that display large primitives.

Screen-order algorithms seem well-suited to antialiasing, since it is natural to consider the contributions of all objects that are visible at a single pixel.

3. Depth order algorithms pre-sort primitives by their depth relationships and render them in sorted order. Screen-space algorithms sort the primitives once for each view of the scene. The ordering may be either back-to-front or front-to-back. Either order accommodates transparency naturally and permits antialiasing. Another category of algorithms, called depth-priority or list-priority, determine view-independent visibility relationships once per static scene in a separate pre-processing step.

The ''painter's'' algorithm suggested by Newell, Newell, and Sancha [Newe72] sorts all objects back-to-front in screen space and renders them, one-by-one, into the frame buffer. Nearer objects are painted on top of more distant objects. If an object is tagged as transparent, the background is allowed to show through by blending the existing image with the color of the transparent object. Back-to-front algorithms must process everything, even if the nearest primitive is opaque and covers the entire pixel. Reeves used an antialiased variation of the painter's algorithm to render complex images of natural phenomena [Reev85].

The Evans and Sutherland CT-5 flight simulator renders objects in a front-to-back order [Schu80]. A high resolution coverage mask records which portions of each pixel have been filled. This coverage mask supplies sub-pixel information for approximate antialiasing. Once a pixel has been completely covered by opaque objects, more distant primitives can be ignored.

Depth-priority algorithms pre-process a static data base in world space to obtain depth relationships that are independent of the viewpoint [Fuch80, Hubs82, Suth74]. Because the priority computation is expensive, these algorithms are most useful in applications such as flight simulation and architectural ''walk throughs,'' where many views will be generated of the same static scene. A linear traversal of the ordered data renders each view efficiently.

The performance of all of the depth order algorithms is determined by the number of primitives and their depth relationships. When objects overlap, the algorithms must split them before resolving depth order. Thus, overlap in the scene increases the number of internal rendering primitives and the compute time. The number of visible objects and the size of transparent surfaces further influence the performance of front-to-back algorithms, which can discard more distant objects once the screen is covered by opaque surfaces. When objects are rendered back-to-front, as with the painter's algorithm, all surfaces must be shaded, even surfaces that are obscured by closer objects.

4. Random order processing is typified by the recursive ray tracing algorithm, which determines visibility and shading concurrently. Ray tracing models the properties of light and can simulate reflection, refraction, shadows, and transparency in a unified way [Kay79, Whit80]. Rays are projected from each pixel into the scene, typically in scan line order. The rays are intersected with the primitives in the scene to determine the nearest object. When a ray hits a primitive, illumination information is computed. Depending on the surface characteristics of the intersected object, *secondary* rays are sent out in the direction of transmission and reflection. Thus, ray tracing models the reflection of one object on another.

The secondary rays can be transmitted anywhere in world space, and they potentially intersect any object in the scene. Consequently, the order in which objects are accessed depends not only on the order in which the primary rays are cast from the image plane but also on the lighting and reflectance properties of the model, which determine the direction of the subsequent rays.

A non-recursive ray tracing algorithm, also called ray casting, solves only the visible surface problem. It casts just the first level rays from the screen to determine the nearest visible object. Given a suitable organization of the model data, a non-recursive ray tracer need not access primitives in a truly random order.

The cost of a brute-force ray tracing algorithm is dominated by ray-primitive intersections. It is prohibitively expensive to test all rays against all primitives in a highly complex scene. There are many strategies to reduce the number of intersections. Instead of examining all objects for each ray, they examine only likely candidates. Object partitioning, by either spatial subdivision or some hierarchical arrangement of the data, is the key to reducing the number of intersections.

The cost of intersecting a ray with an object depends on the type of primitive. Thus, the cost of ray tracing is affected not only by the number of primitives but also by their types. Some systems simplify the problem of solving for intersection by enclosing objects in simple bounding volumes, for example, [Rubi80] or [Kay86]. The cost is also affected by the image resolution, which determines the number of primary rays. Supersampling and stochastic sampling have been applied to ray tracing to produce antialiased images [Cook84b]. For antialiased images, the sampling frequency further influences the number of primary rays. Adaptive supersampling algorithms concentrate rays in areas of greater detail, and their cost increases with the frequency of detail in the image.

5. Hybrid schemes combine scene partitioning with one of the other approaches to save time, memory, or both. In many cases, the object ordering is not an intrinsic part of the underlying algorithm, but an independent optimization.

Spatial subdivision is often combined with another approach. For example, the CT-5 flight simulator and its successors have used spatial subdivision in hardware implementations of a front-to-back visibility algorithm. Multiple processing units perform the visible surface algorithm on several groups of small sub-regions because of the prohibitive cost of enough hardware to process the entire image at once. Two different rendering systems developed at Lucasfilm use modified z-buffers and subdivide the screen. Carpenter's A-buffer algorithm [Carp84] supports approximate antialiasing with a coverage mask. The Reyes image synthesis architecture uses a stochastic sampling visibility algorithm [Cook86]. Both of these systems could theoretically process objects in any order, but they subdivide the screen to save memory. The A-buffer system processes objects in scan line order, and Reyes uses rectangular regions.

An approximate pre-ordering of the data is another cost reduction technique. Consider the z-buffer algorithm, which paints primitives into the frame buffer in an arbitrary order. The algorithm shades many objects that it later overwrites. A preliminary front-to-back sort based on approximate depth bounds may eliminate much of the unnecessary shading. A similar optimization has been applied to Reyes, as described in Chapter 3.

Ray tracing systems use many forms of object partitioning. If the objects in the scene are organized suitably, only a subset need to be tested for intersection with a given ray. These approaches lead to more efficient uniprocessor implementations and can also be the basis for parallel implementations. Spatial subdivision schemes have used either adaptive partitions (for example, [Dipp84] or [Glas84]) or regular partitions (for example, [Fuji86]). Other ray-tracing algorithms use bounding volumes and object hierarchies to reduce the number of intersections [Kay86, Rubi80, Wegh84]. Hierarchical object grouping can also be applied to image-space algorithms [Clar76].

2.5.2. Visible Surface Costs

Since visibility is a geometric problem, the geometric cost factors discussed in Section 2.3.2 predominate. Table 2.6 lists additional factors that influence the cost of the visible surface problem. In general, the number and size of primitives in the scene determine the size of the problem. Sometimes all of the primitives in the scene affect the runtime, as with the z-buffer. In other cases, the subset of visible primitives dominates, as with front-to-back rendering. In general, the types of primitives in the scene affect the complexity of the geometric calculations. Other geometric factors are important to certain algorithms. For example, depth relationships and overlap affect depth-priority algorithms, and the spatial distribution of objects affects spatial subdivision algorithms. The number and size of transparent surfaces affect algorithms that support transparency.

Frame-to-frame coherence characterizes the similarities between consecutive frames of an animated sequence of images. Some algorithms, such as the depth priority methods, depend on frame-to-frame coherence and are efficient when many images are generated of a static scene.

Appropriate data structures speed processing. Some rendering systems use supplementary data. For example, bounding boxes describe the minimum and maximum x and y coordinates of an object, and bounding volumes can delimit the extent of objects in three-space. Simple calculations with bounding boxes often replace more complicated operations on the actual object geometry. Bounding boxes are also useful for bucket sorts that partition the data spatially. Some rendering systems group related objects in hierarchical data structures that provide extra coherence.

Sutherland, Sproull, and Schumacker emphasized the importance of sorting to the visible surface problem. Primitives are sorted by depth and into the proper location on the screen. Sorting performance is influenced by the algorithm, the size of the problem, and by the initial ordering of the data. It is important to choose efficient sorting techniques. For example, radix sorts group objects into scan line buckets with a linear pass over the data. Many algorithms partition the image into small sub-problems. In this case, a bubble sort can be efficient, especially when the objects are almost in order.

Category	**Cost Factors**
Scene Characteristics	changes in objects' shapes over time movement of objects over time
Rendering Parameters	number of images (frames) per scene initial ordering of model data data structures (hierarchy, bounding volumes, etc.)

Table 2.6. Visibility cost factors. This table lists factors that affect visibility costs only. The geometric cost factors discussed in Section 2.3.2 are also critically important.

Sutherland, Sproull, and Schumacker estimated the performance of several visible-surface algorithms as a function of the number of polygons in the scene [Suth74, page 346], and Foley et al. normalized their results [Fole90, page 716]. Figure 2.4 is based on Dennis' graph of the normalized data [Denn90]. Sutherland, Sproull, and Schumacker estimated both the timing data and the workload complexity without making any measurements. They suggested that their estimates be used only for order-of-magnitude comparisons. Their performance analysis estimated the visible surface processing time for three hypothetical scenes with varying complexity. In these scenes, the aggregate screen area is approximately constant. Consequently, the average screen size of a polygon varies inversely with the number of polygons in the scene.

As Figure 2.4 shows, the depth sort algorithm is efficient for small scenes. When the number of polygons increases, more complicated depth relationships and overlap problems make the algorithm very expensive. We have already noted that two algorithms, the scan-line algorithm and Warnock's subdivision, are more efficient for larger primitives. Their estimated runtimes increase with the scene complexity, as primitives tend to become smaller. In contrast, the cost of the z-buffer depends primarily on the total number of pixels covered by all primitives. Its runtime stays constant under this workload complexity model. However, the z-buffer normally shades all objects in the scene, while ray tracing shades only surfaces that affect the final image. For very complex scenes, Kajiya argues that ray tracing becomes more efficient [Kaji88].

Other characteristics of the workload, besides the size and number of primitives, influence the choice of an algorithm. The types of geometric primitives in the workload may be an important

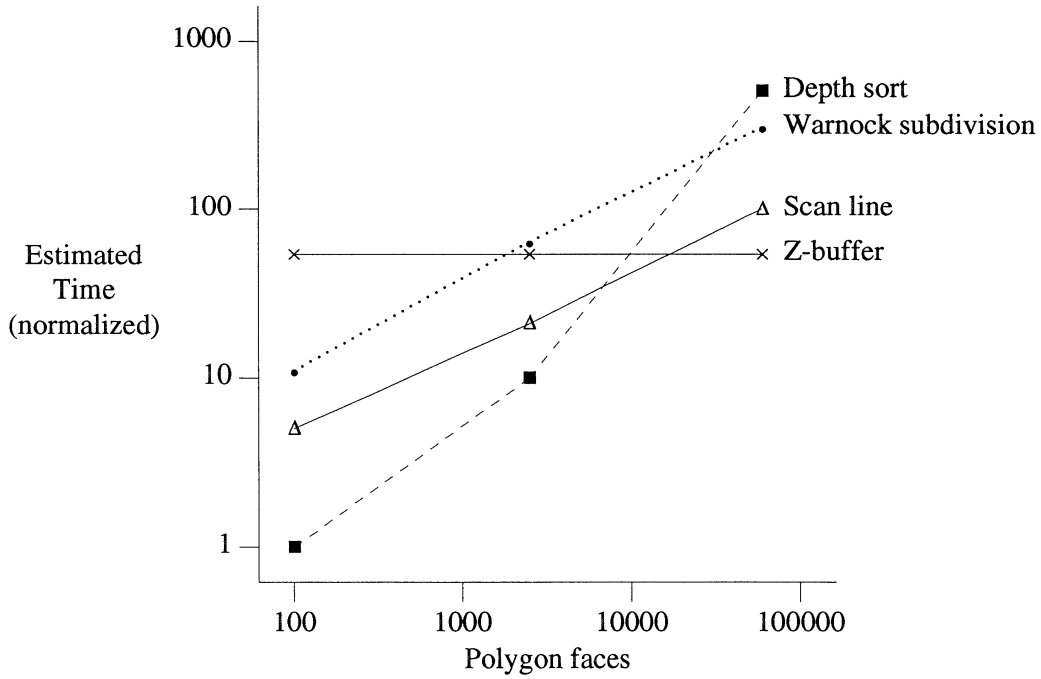


Figure 2.4. Estimated run time for four visible-surface algorithms.

consideration, because some visibility algorithms are efficient only for limited types of primitives. For example, the Weiler-Atherton algorithm requires complex clipping algorithms that are practical only for polygons. The choice of an algorithm is also influenced by the visual effects, such as transparency, that the application requires. Recursive ray tracing, scan-line, and depth order algorithms support transparency easily, but straightforward z-buffer algorithms cannot render transparent surfaces accurately.[4] Some algorithms are optimized for applications that render many views of a static scene. For example, a single rendering would probably not recoup the cost of a depth-priority algorithm's extensive preprocessing.

Systems issues such as parallelism, hardware support, and time/memory tradeoffs favor certain algorithms. In a successful parallel implementation, the data must be partitioned without destroying any type of coherence needed by the chosen algorithms. The independence of each primary ray makes ray tracing amenable to parallel implementation, but its random access to a potentially large data base complicates data distribution and communications issues. Hardware implementations are practical for simple z-buffer algorithms and the scan conversion portion of depth-order algorithms, if we assume a simple shading model. Tradeoffs between time and memory have shifted since Sutherland, Sproull, and Schumacker compared visible surface algorithms. They advocated scan-line algorithms over a z-buffer for complex images, because the additional depth memory was impractical at the time. Commercial z-buffer systems have now been common for many years. Spatial subdivision schemes often save memory, although they tend to increase the complexity of a system and may increase runtime.

[4] See Foley et al. for a detailed explanation [Fole90, page 754-755].

2.6. Shading and Texturing

The rendering system must determine a color value, or *intensity*, for each visible surface in the scene. The intensity describes the light reflected by the surface towards the viewer. Its value is normally a triple of red, green, and blue color components. In real scenes, the intensity perceived by a viewer is affected by the material properties of the surface, its position and orientation with respect to the viewer, the properties and position of light sources, and inter-reflectance among the objects in the scene. The computer graphics community has pursued two contrasting approaches to shading: imitating a realistic appearance with heuristics and simulating physical processes. Simulating the physics of illumination is a very hard problem, and even the most realistic shading algorithms are only approximate. On the other hand, many simple conventions produce acceptable results. Realistic shading is an active research area. Researchers are continuing to introduce improved shading techniques, based increasingly on models of actual physical processes. Hall surveys the history, theory, and practice of illumination [Hall89].

An image is a representation of the light reflected by objects in the scene. The shading process models two types of reflection. *Diffuse* reflection is characteristic of dull surfaces, which scatter light equally in all directions. The direction to the viewer does not affect diffuse reflection. *Specular* reflections are seen in the highlights of shiny surfaces, and they are strongest in a *direction of reflection*, which depends on the direction to the source. Because specular surfaces do not reflect light evenly in all directions, the intensity of the highlight depends on the viewer's position. Most surfaces have a combination of diffuse and specular components.

The *illumination model* describes how to calculate the intensity of light at a specified point on a surface. The *shading model* specifies at which points to apply the illumination model and determines the environment in which it is applied. A *local* illumination model considers only a point on the surface, the viewing specifications, and the light sources. A *global* illumination model also considers the transmittance of light throughout an environment. Global illumination is inherently more complex than local illumination, since it includes other surfaces in the scene to simulate phenomena such as reflection, shadows, and refraction. One approach to global illumination is recursive ray tracing, which models specular reflection, transparency, refraction, and shadows by following secondary rays in the directions of reflection and transmittance and towards the light sources. Another approach to global illumination is *radiosity*, which is based on thermal engineering methods. It simulates color bleeding and diffuse interreflection by modeling the energy equilibrium in a closed environment. Global illumination algorithms are described in more detail later in this section.

2.6.1. Light Source Models and Cost Factors

The intensity of reflected light is determined not only by the properties of the objects but also by the light arriving from light sources in the environment. Light sources may either reflect or emit light. Conventionally, a single *ambient light* computation approximates the combined effect of reflected light from nearby objects in the environment.[5]

Light sources such as the sun, lamps, and candles emit light. A scene may contain multiple emitting sources. Each light source requires a separate shading calculation, so the cost of shading increases with the number of light sources. Ambient light alone produces unrealistic images, because it gives the same intensity to all points of a surface. When an object is illuminated by an emitting light source, the intensity varies over the surface according to the position of the light.

Geometric properties of the light sources affect the cost of shading a scene. Shading computations apply Lambert's cosine law, which states that the intensity of diffuse reflection depends on the angle of illumination. Accounting for the direction normally requires a dot product operation to compute the

[5] Radiosity models ambient lighting more precisely by calculating the interactions of reflected light on diffuse surfaces.

cosine. But if the light has no direction, as with ambient light, the dot product may be skipped.

The cost of shading depends on whether the light source is modeled as infinitely distant or nearby. The simple *point light source* model describes an emitting source that is distant and small relative to the scene. Its light rays come from a single point. This model is adequate for light sources like the sun. It is reasonably accurate to assume that all light rays reaching a surface from a point source are parallel, because the model assumes that the light source is distant. In effect, the angle of illumination is constant over the surface. With nearby lights, the angle changes, and the shading algorithm must re-evaluate the dot product or interpolate its value over a surface. The intensity of light falls off as the distance from the light source to the surface increases (specifically as the inverse of the square of the distance). If the light is distant, the effect of distance is approximately the same for the entire scene. If the light is nearby, more computation is required to simulate the attenuation of light for objects that are far from the light source.

The size of the light source also affects shading costs. A point light source has no area, which is another simplification of the model. In contrast, large and nearby lights have area. They are simulated more accurately by a *distributed* light source model. Several properties of distributed light sources increase the computational costs. Sometimes a distributed source is approximated by a group of point sources, but ideally the illumination is integrated over the light's area. Light rays from a distributed source are not parallel, so the cosine varies at each point. Interpolation can approximate the change in the cosine, but the results are not always satisfactory. Furthermore, distributed light sources generate soft shadows, which are more complicated to compute than hard-edge shadows.

Not all light sources emit light evenly over their surfaces. Realistic models can assign different properties to regions of the light's surface, thus increasing the complexity of the shading operations. Comprehensive and unrestricted approaches to modeling luminous function distributions are rare; Verbeck and Greenberg describe an exception [Verb84].

2.6.2. Local Illumination

Simple, empirical illumination models are common in computer graphics. A typical model uses ambient light and point light sources to illuminate surfaces with diffuse and specular components. The diffuse and specular coefficients that describe surface materials are usually assigned empirically. The final intensity is the sum of three terms: the contributions of ambient, diffuse, and specular reflection.

Diffuse reflection, according to Lambert's law, varies with the intensity of the point light sources, the value of the diffuse reflectance coefficient, and the angle between the surface normal and the light direction. Specular reflection is simulated with an empirical model developed by Phong Bui-Tong [BuiT75]. A non-linear factor accounts for the declining specular intensity as the view direction diverges from the direction of reflection. Phong's model raises a cosine to a power to simulate the nonlinear decline; the empirically-derived exponent is greater for shinier materials. The intensity of the point light sources and the value of the specular coefficient are also needed to compute the specular contribution. The equation for this simple illumination model is given in Figure 2.5.[6] More realistic variants of this model may support other light source models or light-source attenuation.

The shading model specifies how frequently the illumination model is evaluated. In general, intensity varies over the surface of each primitive. Suppose that the rendering system is shading a polygon that covers several pixels. The simplest approach is to shade the polygon at only one point and use the same intensity for all pixels that the polygon covers. Unfortunately, this would give the image an unrealistic faceted appearance. The most accurate, and most costly, approach is to integrate the illumination over the area corresponding to each pixel. Between these two extreme lies a range of approaches, often

[6] A different formulation for the specular component was suggested by Blinn [Blin77]. Instead of using $R_i \cdot V$, it uses the "halfway" vector bisector H, between V and L_i. In the restricted case where both the viewer and the light source are at an infinite distance, H is constant throughout the scene and can be evaluated once [Whit82].

$$I = I_a k_a + \sum_{i=1}^{l} I_{p_i} \left[k_d (N \cdot L_i) + k_s (R_i \cdot V)^n \right]$$

Notation:

I	computed intensity
I_a	intensity of ambient light
I_{p_i}	intensity of point light source i
l	number of point light sources
k_a	ambient reflection coefficient; may be the same as k_d
k_d	diffuse reflection coefficient
k_s	specular reflection coefficient
n	Phong specular reflection exponent
N	surface normal at specified point (normalized)
V	direction to viewpoint from surface (normalized)
L_i	direction to light source i from surface (normalized)
R_i	direction of reflection from light source i (normalized)

Figure 2.5. Local illumination. This equation, or a variation, is commonly used to calculate the intensity of light at a point on a surface. This model is based on Phong's model for specular reflection and allows multiple point light sources.

related to antialiasing choices. In general, the shading quality increases with the computational cost.

A method described by Gouraud in 1971 [Gour71], interpolates the intensity to give a smooth appearance to curved objects approximated by polygons. It evaluates the illumination model only at the vertices of a polygon and interpolates the shade linearly in the interior. The algorithm uses incremental calculations to interpolate along polygon edges from scan line to scan line, and along scan lines from one pixel to the next. Gouraud's interpolation renders highlights inaccurately. The surface normal of a curved object changes from point to point, so it is more accurate to vary the normal and re-evaluate the illumination model in the interior of the polygon. Phong Bui Tuong [BuiT75] improved upon Gouraud's model by approximating the surface normal at each pixel to render more accurate highlights. Phong's algorithm uses the same type of incremental interpolation as Gouraud's, but instead of interpolating the intensity it interpolates the surface normal between vertices. It calculates the pixel's intensity using the approximate normal as N in Figure 2.5. Phong shading is more expensive than Gouraud shading because it evaluates the illumination model in the interior of the polygon, computing a new normal for each evaluation.

Interpolation, in general, introduces errors because it misses high-frequency detail and because intensity changes are not really linear in screen space. Furthermore, when polygons approximate a curved surface, inaccuracies in the approximation introduce shading discontinuities. A more expensive, but more accurate, alternative is to avoid interpolation and evaluate the illumination model in the interior of the object. For an antialiased image, this may imply multiple shading calculations for a single pixel.

The illumination model of Figure 2.5 is based on appearances and not on physical processes. More realistic shading algorithms use *physically-based illumination models* to describe material properties and reflection. Blinn [Blin76] and Cook and Torrance [Cook82] introduced more accurate algorithms based on the Torrance-Sparrow model of reflective surfaces. In this model, a surface consists of a set microscopic facets with non-uniform orientations. Simpler surface models make most objects look like plastic. With the improved models, surfaces may resemble many real materials, such as various metals or types of stone. As usual, the increased realism comes at the cost of increased computational effort.

2.6.3. Texture Mapping

Geometric modeling is not the only way to add detail to an image. An alternative is to map a texture onto a three-dimensional surface when the surface is shaded, as first described by Catmull [Catm74]. Texture maps have been used to modify transparency, surface normals, reflection, shadows, and other surface properties, but they are most commonly applied to the surface color. Most texture maps are two-dimensional, but it is possible to use three-dimensional texture functions to simulate surfaces carved from solid textures, such as marble or wood. Heckbert provides a comprehensive survey of texture mapping techniques [Heck86].

To apply a texture to a geometric primitive, two types of information are required: a texture name and a function mapping points in the texture map onto the three-dimensional surface. This function may be an arbitrary function of the surface parameters or a pair of indices into the texture map. The mapping is easier for some types of primitives, such as bicubic patches, than for others. The value retrieved from the texture map is used to modify some property of the surface, most often its color.

One special use of texture maps is to create the illusion of bumpy or wrinkled surfaces. The values in a *bump map* are small perturbations that vary the direction of the surface normal [Blin78]. The perturbed normal produces changes in intensity that mimic the appearance of a bumpy surface.

Texture maps can also approximate reflections of the surrounding environment on the shiny surfaces of objects. This application, called *reflection mapping* or *environment mapping*, was introduced by Blinn and Newell [Blin76]. The environment mapping algorithm assumes that a sphere surrounds the scene.[7] The viewer does not see the sphere itself. However, an image of the reflected environment is mapped onto its interior surface. When a shiny objected is shaded, any reflections are found by accessing the appropriate location on the surface of the sphere. This location is determined by direction of reflection from the point being shaded to the surface of the surrounding sphere. The direction indexes the corresponding environment map, and the value retrieved from the map represents the image of a reflection.

With any type of texture map, several texture pixels or portions of texture pixels, may map to a single display pixel. If the texture is not filtered, the image will be aliased. In the general case, the texture value is a weighted sum of the relevant texture pixels. Accurate, but expensive, analytic algorithms access the relevant texture pixels and apply a weighted filter [Feib80]. This approach is called direct convolution. Supersampling the texture can also produce acceptable results. The sampling pattern can be adaptive, to sample more heavily in regions of high-frequency information. It can also be stochastic, to minimize the visual artifacts of aliasing.

The cost of texturing corresponds approximately to the number of texture accesses. Both direct convolution and supersampling may access a large, and highly variable, number of texture pixels for each screen pixel. Algorithms that pre-filter textures can ensure a constant, relatively small ratio of texture pixels to screen pixels. Pre-filtering algorithms process textures before rendering and store a filtered version of the textures [Crow84, Will83]. Pre-filtered textures consume more memory and incur a setup cost. They can be efficient when the texture is re-used, either in a single image or in several images. If the

[7] Another object, such as a cube, may be used instead. For the sake of simplicity, we can assume a sphere.

texture will be used only once, a different algorithm may perform better. If the texture is an arbitrary function, and not an array, pre-filtering is not generally feasible.

Rendering costs can vary with the characteristics of the texture. Typically, the texture parameters index an array. More generally, the texture value can be computed as an arbitrary function of the parameters. It is usually more costly to compute an arbitrary texture function than to retrieve a value from an array. Once the texture value is retrieved, subsequent computation may be more expensive for certain texture applications. For example, bump mapping varies the surface normal and rules out simple interpolated shading for bump-mapped objects. High-frequency detail can increase the cost of adaptive antialiasing, whatever the type of texture.

Texture maps can be very big, and using a number of textures in a single image introduces important systems and memory issues. The renderer may implement a cache for the portions of texture maps that it accesses. The degree of coherence (that is, the locality) in a sequence of texture requests naturally affects the system's performance. The order in which texture pixels are accessed depends heavily on the visibility algorithm. Locality is stronger when an algorithm shades one object at a time, as do the z-buffer and the depth-order algorithms. Locality is weaker when an algorithm processes several objects concurrently, as do the scan-line algorithms.

2.6.4. Global Illumination

Realistic images must give a convincing representation of global illumination effects such as shadows, transparency, refraction, and inter-object reflection. It is possible to imitate the appearance of global illumination with heuristic techniques. For example, texture maps may simulate shadows or reflection. However, true global illumination models are more powerful and complex, producing the most accurate representation of inter-object reflection. Three approaches to global illumination are discussed below.

1. Heuristic approximations have been developed for shadows, transparency, and reflection. The shadow problem is a variation of the visible surface problem. If an object is visible from the light source, then it is not in shadow with respect to that light source. Rendering systems have augmented local illumination models with shadows by adapting a variety of visible-surface techniques. This class of algorithms works best with point light sources, which cast hard shadows. Both object-space and image-space methods have been used, as well as approximate shadow algorithms. Williams first proposed a two-pass z-buffer shadowing algorithm [Will78]. The first pass computes the scene from the point of view of the light source and creates a z-buffer. This z-buffer, or *shadow map*, contains the depth of the object closest to the light. The second pass renders the scene. When a surface is shaded, its position is transformed into coordinates in the light source's space. Using the transformed x and y, the shadow map z value is retrieved. If the shadow map z is less than the transformed z of the surface, then another object is closer to the light source, and the surface is in shadow. The shadow map can be accessed as a variation of a texture map. Generally, the cost of computing shadows depends on the same geometric factors that affect visibility costs. The cost is multiplied by the number of light sources that cast shadows.

Distributed light sources cast shadows with soft edges. Because a distributed source has area, only part of the light it emits is blocked by the edge of an object. Soft edges are sometimes imitated successfully with heuristics [Verb84], but an accurate representation requires a more complicated global illumination model [Aman84, Cohe85, Cook84b].

Transparent surfaces transmit the light from objects located behind them; typically, they also reflect some light. A coefficient of transmission weights the contributions of reflected and transmitted light. Transparency has ties to the visible surface problem, since the shader needs to know all of the objects that are visible at each point on the screen.

Refraction bends light as it travels through a transparent material. The index of refraction describes how a specific material refracts light. The angle of refraction is computed with dot product operations

and a square root. To avoid this computation, some systems overlay objects in screen space without computing an angle of refraction. Ignoring refraction gives transparent objects the appearance of very thin glass.

Texture maps are sometimes used to simulate inter-object reflections, as described in Section 2.6.3. These reflections are inaccurate, because the mapping considers only the direction to the environment map and not the location of the shaded surface within the environment.

2. Ray tracing simulates specular inter-object reflection, shadows, and transparency with refraction. After shading the nearest visible point with a local illumination model, it generates secondary rays in the directions of reflection and transmittance. Rays are generated recursively until there are no more intersections. The recursion can also end at a user-specified depth or when the algorithm determines that further rays will have a negligible effect on the pixel's intensity. Ray tracing handles sharp specular reflection well and produces shadows by sending secondary rays in the direction of light sources. It does a poor job of simulating diffuse inter-reflection among objects.

As discussed in Section 2.5.1, ray-object intersections dominate the cost of ray tracing. Two complementary strategies make ray tracers more efficient: speeding intersection calculations and reducing the number of intersections. Object partitioning and object bounding are useful techniques for improving ray tracing performance following the latter strategy.

3. Radiosity is a global illumination approach based on thermal engineering models of radiative heat transfer. It calculates more accurately the diffuse inter-object reflection that conventional ambient lighting only approximates. Goral et al. [Gora84] first applied radiosity to image synthesis. In the radiosity model, all surfaces emit and reflect light. The energy that leaves a surface is absorbed by or reflected from other surfaces in the environment.

Radiosity algorithms subdivide each primitive into a set of *patches*, such that the illumination of each patch is uniform over its surface. For a scene with n patches, a set of n^2 *form factors* describes the proportion of energy leaving one patch that arrives at each other patch. Form factors are geometric; they are independent of the viewpoint, the lighting, and the surface characteristics. A set of simultaneous equations determines the form factors. This computation is very expensive, but it is required only once per static scene. Any number of views of the scene may be rendered without recomputing the form factors. For each view, the rendering system first determines visibility, for example with a scan-line algorithm or a z-buffer. It then shades the image with a simple interpolated shading model that considers the stored form factors and information about the surface characteristics.

Approximation techniques have been developed to speed the form factor computation. These include progressive refinement algorithms that approximate the form factors in stages [Cohe88a]. Adaptive subdivision can be used to reduce the number of patches, and therefore the number of form factors. Scenes with large, uniform surfaces require fewer patches. In contrast to other algorithms, radiosity handles large light sources more efficiently than point sources. Cohen examines some performance aspects of radiosity algorithms [Cohe88b].

Specular reflection varies with the viewpoint. Because radiosity offers a view-independent solution to inter-object reflections, it handles diffuse reflection but not specular effects. There have been some efforts to combine ray tracing and radiosity, such as the work of Wallace, Cohen, and Greenberg [Wall87], Heckbert [Heck91], and others.

2.6.5. Costs of Shading and Texturing Operations

Realistic shading is still an open-ended problem; the realism, and corresponding costs, vary widely with the algorithm. The choice of an illumination model may depend on the characteristics of the objects. If the scene exhibits little specular reflection or transparency, an expensive ray tracing algorithm may be inappropriate. So far, we have assumed that a renderer applies a uniform illumination model to all surfaces. Some more flexible rendering systems interpret shading languages, which can specify different

illumination equations for different materials. In any case, it is not the colors of the objects, but the complexity of the illumination model, that affects the cost of shading calculations.

Given a specified model, the factors summarized in Table 2.7 affect the costs of shading and texturing. The material properties of the surfaces have performance implications that are very important, but hard to quantify. In general, reflectivity and transparency affect shading costs in proportion to a surface's size. Geometric factors influence the size of the problem, that is, the area that must be shaded. The types of geometric primitives also affect the complexity of some calculations. Current radiosity algorithms handle polygonal patches, so curved surfaces must be converted to a polygonal approximation.

True global illumination algorithms, such as radiosity and recursive ray tracing, are compute intensive. Texture maps approximate global illumination effects. This alternative requires less compute time and summarizes illumination information in a texture file.

Category	Cost Factors
Scene Characteristics	number of light sources directional or non-directional light sources distant or near light sources size of light sources luminous function distribution of light sources types of primitives number of primitives material properties of surfaces number, size, and distribution of transparent surfaces number, size, and distribution of reflective surfaces number of texture maps types of texture maps
Viewing Specifications	number of visible transparent surfaces screen size and distribution of transparent surfaces screen size and distribution of reflective surfaces number of texture requests number of texture pixels accessed
Rendering Parameters	shading frequency maximum recursion depth for ray tracing number of views or frames rendered of each scene number of times each texture map is used texture filtering parameters

Table 2.7. Shading and texturing cost factors. Other geometric cost factors discussed in Section 2.3.2 also affect the cost of shading.

2.7. The Structure of Rendering Systems

The performance of a rendering system is more than a sum of isolated parts. The structure of the system and interactions among the various algorithms influence the strengths and weaknesses of the system with respect to both image quality and performance. Choices that characterize the system's structure include the order of operations, data structures, specific algorithms, and the information that is kept or

discarded at each processing stage. For example, the visibility algorithm and the order of operations affect the scope of the shading problem. If the system determines visibility before shading, it may avoid shading invisible surfaces. If the system calculates illumination entirely in screen space, it lacks the information needed for true global illumination. The algorithms that implement true global illumination, such as ray tracing and radiosity, must do at least some shading computation before the transformation to screen space. Figure 2.6 illustrates the order of operations and the coordinate systems used in different rendering stages, summarizing the information in Sections 2.5 and 2.6. All of the examples support modeling in the local object space and display the image in device coordinates. They differ in the order of the visibility and shading operations and in the timing of coordinate transformations.

Approach	Coordinate System					
	Local	**World**	**Eye**	**Screen**		**Device**
classical image space	Modeling			Visibility	Shading	Display
classical object space	Modeling	Visibility			Shading	Display
Reyes	Modeling		Shading	Visibility		Display
ray tracing	Modeling	Visibility ↓ ↑ Shading				Display
radiosity	Modeling	Per-scene Shading (Form Factors)		Visibility	Per-view Shading	Display

Figure 2.6. Structure of five rendering approaches: (1) classical image space, (2) classical object space, (3) Reyes, (4) ray tracing, (5) radiosity. The figure shows the order of operations and the coordinate systems in which they are performed.

1. Classical image space algorithms determine visibility and illumination in screen space, in that order. One example is the combination of a scan-line visibility algorithm with Gouraud or Phong shading. Another example is a z-buffer with Gouraud or Phong shading.

The scan-line algorithm completely determines visibility for a pixel before shading, so it only shades visible surfaces. The z-buffer processes one primitive at a time, first checking its visibility and then shading the visible fragments. Because it processes primitives in an arbitrary order, the z-buffer algorithm may shade surfaces that are later obscured.

The performance strategy of scan-line rendering and many other classical screen-space algorithms is to speed scan conversion by exploiting object coherence. This strategy succeeds when the image is

composed of large primitives, especially polygons. For complex scenes with many tiny surfaces and a variety of primitives, the strategy falls apart.

The scan-line algorithm processes together all objects that intersect the current scan line. For each object, it considers only the fragment that intersects the current scan line. If fragments of many different surfaces are visible in one scan line, shading and texturing operations will tend to exhibit little locality. For this reason, scan-line algorithms can fail to perform well for workloads with heavy texturing. In contrast, the z-buffer preserves locality by shading an entire primitive at one time.

Both algorithms perform shading without access to the full scene description or to the world space geometry. Therefore, they cannot support true global illumination models.

2. Classical object space algorithms, like Weiler and Atherton's, differ by determining visibility in world space. Like the classical image space algorithms, they shade in screen space and do not compute global illumination. By determining visibility first, these algorithms are able to avoid shading invisible surfaces. Compared with image-space algorithms, object-space approaches frequently involve more complicated geometric calculations. Image space algorithms determine visibility relative to the viewpoint. Object-space algorithms can compute view-independent visibility information and re-use the results to render many views of a static scene. The depth priority algorithm is an example of this approach.

3. The Reyes image rendering architecture [Cook87] is unusual because it shades all surfaces before deciding which are visible. Traditional rendering systems determine visibility first, and then shade only visible surfaces. Frequently, they shade the visible fragments of a given object at different times. Reyes, on the other hand, shades an object's entire surface at once. Thus, it can more effectively exploit coherence in shading and texturing. The Reyes rendering system is optimized for scenes with extensive texture mapping, since it simulates shadows and reflections with texturing. The design of the system is based on the hypothesis that the efficiency gained from increased coherence offsets the losses incurred by shading hidden surfaces.

Reyes uses a modified z-buffer to determine visibility. Its high-resolution z-buffer supports stochastic supersampling, and the system creates a list of all surface fragments that are visible at each pixel. A filtering pass examines the visible fragment lists to compute the final intensities. To save memory, Reyes partitions the screen and renders one rectangular sub-region at a time.

4. Ray tracing determines visibility and illumination concurrently using world coordinates. Global illumination is computed with full access to the scene data in world space. With proper organization of the scene description, the system can reduce the references to objects that do not contribute to the image. Thus, it can avoid shading surfaces that are not visible either directly, through reflection or refraction, or in shadows.

The algorithm has elements of both screen-space and world-space processing. Primary rays are projected from the screen into the scene. The image resolution, therefore, affects the sampling resolution, and adaptive algorithms may increase the sampling frequency to suit the detail in the image. To subdivide the scene, a parallel ray tracer may partition the screen to distribute primary rays among the processors. On the other hand, the system can partition the three-dimensional world space and distribute the scene among processors.

5. Radiosity, separates shading into two steps: a view-independent form factors computation in world space and view-dependent interpolated shading in screen space. The approach solves for the form factors once for each static scene. For each view, it then completes the rendering much like a classical image space algorithm. Radiosity can be implemented with a front end that computes form factors and a back end that performs classical screen space rendering. While the front end processing is expensive, the back end is relatively simple. Once the form factors are available for a scene, a graphics workstation can generate many views quickly. This is attractive for applications such as architectural walk throughs. Typically, radiosity algorithms handle only diffuse reflection in world space and do a poor job of handling global specular effects.

2.8. Summary

Two considerations affect rendering costs: the size of the problem and the complexity of the problem. The factors that determine the problem size are easier to identify and to quantify. The complexity of image space algorithms is often described as np where n is number of objects and p is number of pixels. Similarly, the complexity of object space algorithms is described as n^2. In general, the number of objects and the image resolution are the primary factors that describe the size of the problem.

More specifically, the geometric factors that most strongly influence the size of the problem are the number of primitives and the size of the primitives when projected on the screen. The resolution is more than just the number of pixels in the image. The sampling frequency and the shading frequency both affect the computed resolution. In one respect, the image resolution is independent of the model, because it can be set arbitrarily at rendering time. However, in practice, the sampling frequency and shading frequency depend on the frequency of detail, which is a characteristic of the model.

The types of geometric primitives affect the complexity of the problem by determining the complexity of the geometric expressions and by influencing the types of algorithms that may be used.

The number of primitives and their screen size affect the size of the shading problem as well as the visible surface problem. Other factors that determine the size of the shading problem are the number of light sources, the number of textures, and the number of texture accesses. The characteristics of the surfaces, the illumination models, and the light source models all contribute to the complexity of the shading problem. The shading complexity of a scene is very difficult to characterize, and the complexity of shading algorithms varies tremendously.

3
An Analysis of Image Characteristics

3.1. Introduction

Workload characterization provides a necessary foundation for performance studies. This chapter characterizes, both quantitatively and qualitatively, the image synthesis workload. There are several motivations for quantifying the complexity of images. The first goal is to explore the performance of rendering algorithms. Measurements of actual images can help us understand the implications of theoretical performance bounds and predict average case behavior. The second goal is to obtain data to use in the design and evaluation of new systems.

The first set of data in this chapter is extracted from the computer graphics literature. It documents trends in image complexity over the past twenty-five years. The remaining data describe the characteristics of several complex images taken from an actual computer animation workload. These measurements were obtained by instrumenting and profiling a sophisticated rendering system.

For a rendering system, the workload consists of the data that specify an image: the scene description, the viewing parameters, and the motion in an animated sequence. The workload is modified by system-specific rendering parameters that the user supplies at runtime.[1] Chapter 2 identified a set of factors that affect the cost of rendering operations. Metrics of image complexity are derived naturally from these cost factors.

Some characteristics of models can be determined by studying the input specifications directly. These include scene complexity statistics like the number and types of modeling primitives or light sources. Other characteristics can be determined only by evaluating the model, for instance, by fully instantiating an object that is described tersely by procedural modeling specifications. Although a stochastic element might add variation to models, in production the variability is usually controlled and repeatable.

The rendering algorithms uncover further characteristics of the scene. For example, the visible surface algorithm discovers the number of visible objects, and, perhaps, the depth complexity. The geometric computations calculate characteristics such as the screen size of primitives.

A direct examination of the image provides further information, such as the percentage of the screen covered by the scene. A visual inspection of the image can indicate the importance of qualities such as texture, reflection, shadowing, or transparency.

This chapter presents information obtained by examining scene specifications, by instrumenting or profiling the renderer, and by analyzing images. Informally, we refer to "image complexity," although our true interest is in the complexity of the input to the renderer.

What is image complexity? In one respect, a complex image is a picture with a visual richness and perceptual clues that contribute to a sense of realism. Photorealism, an approach that attempts to render a scene as a photograph would show it, has been suggested as one standard for perceptual complexity.

[1] If the system is interactive, the workload includes further commands and dynamic specifications from the user. The research described in this chapter does not address interactive graphics, and the rendering systems do not support real-time image manipulation.

Thus, we can apply the "Turing test" to try to distinguish between a synthetic image and a photograph. Hagen presents an extensive examination of perceptual issues and realism in two-dimensional imagery [Hage86].

The computer graphics community has often focused on geometric complexity, especially on metrics that are easy to quantify. The number of polygons in a scene is a common metric, but this supports only a simple view of complexity. It fails to account for the range of primitives in computer graphics, which vary widely in complexity. It also ignores the complexity associated with shading and rendering effects.

This thesis addresses the performance of rendering systems, so it is natural to take an operational approach to image complexity. That is, the complexity of an image is reflected by the complexity of computing the image. This view encompasses both the geometric complexity of the model (which determines the number of rendering operations) and the complexity of the algorithms (which determines the cost of the rendering operations).

3.1.1. Previous Work

In the past decade, computer-generated images have become increasingly more complex, and the workload has changed accordingly. The changeability of the workload is only one obstacle to characterizing image complexity. Because of the considerable effort required to create models, research installations typically rely on small test suites. In production installations, information about graphics workloads is often proprietary.

Studies of image synthesis workloads are rare, although the literature reports scattered measurements about individual images. The measurements most often document the number of geometric primitives and sometimes count other elements, such as light sources and texture maps. Typically, the purpose of the statistics is to demonstrate the capabilities and speed of a new algorithm or system. Image complexity statistics also allow designers to compare systems. At times, the actual model files are shared with other sites as benchmarks. For the purpose of comparison, graphics researchers tend to emphasize repeatability over representativeness.

Whelan presents one of the few quantitative studies of image characteristics in the graphics literature [Whel85] . He characterizes the distribution of objects on the screen to assist in the design of a multiprocessor system. The six test images are varied, but they are not part of any production workload. His target application is real-time animation, rather than photorealism.

Dunwoody and Linton also target real-time graphics in their studies of interactive workloads on graphics workstations. They investigated both two-dimensional applications under the X window system [Dunw88] and three-dimensional applications [Dunw90]. In the second publication, they describe tools that dynamically capture the graphics commands generated by interactive programs. A profiler tool interprets the trace and analyzes the workload.

3.2. Historical Survey

The most common metric of image complexity is the number of modeling primitives. To document trends in geometric complexity, I scanned the computer graphics research literature of the past two decades and identified references that reported the number of modeling primitives in three-dimensional images. The data were taken as reported, with little interpretation. To simplify comparisons, only data for conventional surface modeling techniques are included.[2] To emphasize realistic image synthesis,

[2] Images based on volume rendering, height fields, and particle systems were, therefore, excluded. So were data about internal rendering primitives or other intermediate representations, such as *micropolygons* in the Reyes system [Cook87], or the tessellated models of Snyder and Barr [Snyd87]. Data about patches and elements for radiosity algorithms were also excluded, because they are not comparable to geometric complexity statistics for classical surface modeling.

flight simulators, real-time applications, and simple illustrations of modeling techniques are excluded. In general, the survey is limited to realistically shaded images, although the definition of realism is subjective and depends on the state-of-the-art at the time an image is created. In order to show early data points, it was necessary to include some hidden-line images published before 1973.

Figure 3.1 plots the number of geometric primitives in a scene for one hundred sixteen data points from thirty-eight publications. Appendix A cites the sources and gives more detailed information about each of the images; a separate bibliography appears at the end of the Appendix. The graph displays one data point for each distinct scene in a publication, even if it appears in multiple views.

Figure 3.1 divides the models into three categories: hidden-line drawings of polygonal models, shaded renderings of polygonal models, and models with a mixture of three-dimensional primitives. The first two categories include scenes that contain only polygons. A mixed model may contain polygons, but it must also have other geometric primitives. A different symbol marks each of the three categories.

The data in Figure 3.1 represent only a fraction of the images published each year, so I cannot claim that they are truly representative. Most would probably have been considered typical of the state of the art at the time of their publication. Because of the nature of the sample, broad observations are more appropriate than a statistical analysis of the data.

Polygonal models predominated until the mid-1980's and are still common. Algorithms for non-polygonal representations appeared in the graphics literature in the 1970's, but complex examples followed several years later. Figure 3.1 shows few mixed models before the mid 1980's, although one appeared as early as 1971.

Instead of a gradual growth that follows some continuous function, Figure 3.1 shows two clusters of data points, one before 1979 and one after. Ninety percent of the scenes published before 1979 have fewer than 500 primitives. From 1979 on, most scenes have at least 1,000 primitives. The cluster between 1,000 and 100,000 contains three-fourths of the data points after 1978. One interpretation of Figure 3.1 is that changes in technology in the late 1970's and early 1980's allowed the geometric complexity to advance to a new level. Procedural modeling probably played a large role in the increasing geometric complexity of synthetic images. The growth in model size occurred at about same time that stochastic modeling with fractals was introduced.[3] From 1979 to 1991, the data values are spread over a wide range. Even though complex geometric models are possible, designers still want to model some scenes that have simpler geometry.

Some of the small clusters in Figure 3.1 are not accidental. In the mid-1980's certain images gained popularity as common ray tracing benchmarks. In 1987, Haines added five new scenes to a popular test case and created a benchmark suite. He deliberately constructed the scenes to contain around eight thousand primitives [Hain87]. Independent or not, these data points suggest the capabilities of rendering systems at the time of publication. In the early 1970's, researchers also shared models or tried to reproduce images published by others, but these early models contained no more than a few hundred primitives (see Appendix A).

Even in the 1990's, some models are small. But the corresponding images are complex in other ways. For example, the two lowest data points for 1990 represent collision simulations with rigid-body dynamics. The images are rendered with shadows and reflections.

Figure 3.1 shows a difference in the sizes of polygonal models and mixed models. On the whole, the polygonal models have more primitives than the mixed models. Proportionately more of the polygonal models contain over 10,000 primitives. In general, mixed models can describe a scene more economically and more exactly than polygons. Modeling polygons are typically triangles or convex planar

[3] Two fractal modeling papers were given at Siggraph '80, one by Carpenter and the other by Fournier and Fussel. Only the abstracts appeared in the proceedings, and a combined paper by the three authors was published two years later [Four82].

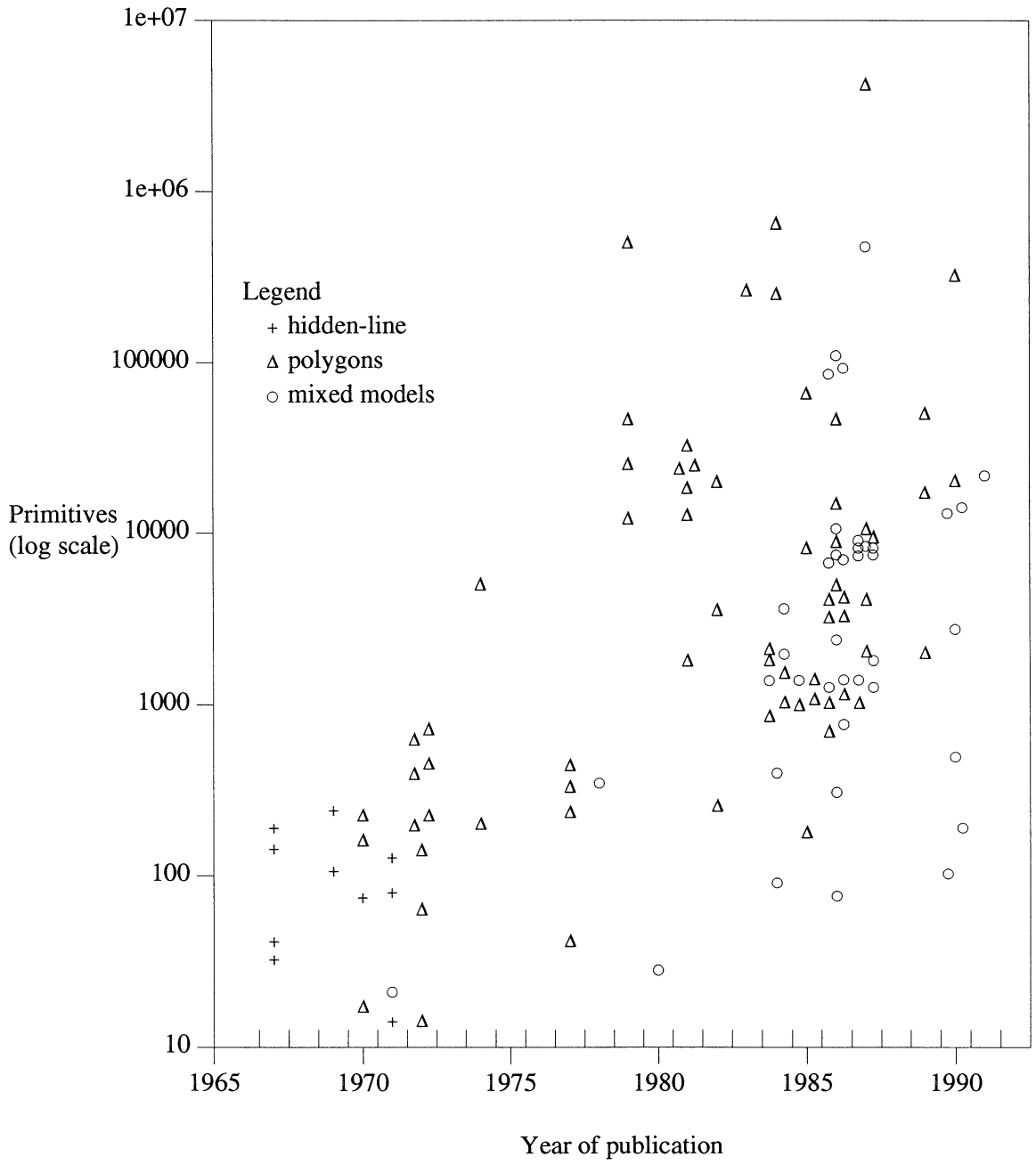


Figure 3.1. Number of geometric primitives. The data describe images published in the computer graphics literature, 1967-1991. Appendix A lists the sources of the data. The data points for a single year are spread horizontally over the corresponding interval. Within one year, the horizontal positions of data points were chosen strictly for clarity and are independent of the month of publication.

quadrilaterals, and it takes many to approximate complex surfaces. Also, the greater computational cost of rendering complex primitives may tend to limit the size of mixed models.

The increase in geometric complexity has been accompanied by an increase in the complexity of surface representation, shading, and rendering. Appendix A shows that transparency, reflections, shadows, textures, and antialiasing have become more common over the last twenty years.

Given past trends, how many primitives can we expect to find in models in the future? In 1991, Greenberg predicted that model databases will grow by two orders of magnitude in two decades [Gree91]. In the previous two decades, as Figure 3.1 shows, the growth was somewhat less than two orders of magnitude. In the years around 1970, the models contained between 10^1 and 10^3 primitives. In the late 1980's and early 1990's, the largest cluster of models contained between 10^3 and 10^4 primitives and a large subset contained 10^4 to 10^5 primitives. However, if we include only polygonal models in the comparison, the growth is indeed close to two orders of magnitude. Greenberg [Gree91] and Reeves, Ostby, and Leffler [Reev90] suggest that the problem of modeling complex scenes now limits geometric complexity more than the problem of rendering complex scenes.

3.3. The Measurement Environment

To characterize the complexity of synthetic images in an animation environment, I measured several complex scenes obtained from Pixar. Pixar is a Richmond, California, computer graphics company, formerly a division of Lucasfilm Ltd. Its animation production and research groups have created state-of-the-art computer animated films and commercials. To produce these films, Pixar has developed languages, algorithms, software, and hardware for modeling and rendering.

This chapter presents three types of measurements about the complexity of the Pixar images. Characteristics of the models were measured by examining the input to the rendering system. Further geometric data and information about texture usage were generated by instrumentation code in the rendering software. CPU time statistics were obtained by profiling the rendering system. This section introduces the rendering environment, and the following section presents the metrics, measurement methodology, and results.

The Pixar software has been developed continuously to add features, improve performance, and exploit new hardware. Two rendering systems served as measurement vehicles: the Reyes image rendering architecture [Cook87] and its successor PhotoRealistic RenderMan[4], or *prman*. Both systems follow the same philosophy, although their input languages, implementations, and some sub-algorithms differ. To simplify the discussion, we will refer to Reyes[5], but the description also applies to prman, except where noted.

Reyes was designed to support complex animated scenes, specifically, scenes with many geometric primitives and extensive texture mapping. The performance goals for Reyes address problems associated with large models. Compute time should increase only linearly with geometric complexity, assuming a constant shading complexity. The system should also preserve locality in referencing the model data, because complex scenes require large amounts of data. One way that Reyes improves locality is to focus on local illumination models, using texture maps to approximate global effects such as shadows and reflections. The system provides mechanisms to include ray-traced elements, but the primary algorithms do not perform raytracing. Instead, a modified z-buffer algorithm with good locality processes objects in $O(n)$ time. Stochastic super-sampling meets the goal of high-quality rendering with good antialiasing.

---

[4] RenderMan is a trademark of Pixar.

[5] The Lucasfilm graphics division produced two different rendering systems and named them both Reyes. Carpenter created the first Reyes in the early 1980's using the A-buffer visibility algorithm [Carp84]. This thesis describes the second Reyes, which takes an entirely new approach to the visible surface problem. It was the production rendering system for the Lucasfilm/Pixar group from 1984 through 1988.

Models can contain a variety of three-dimensional primitives, including polygons, parametric surface patches, and quadric surfaces. The modeling language may be written directly, but it is more commonly generated by interactive or procedural modeling tools. The input language for Reyes is called *model*, while prman supports the RenderMan Interface [Pixa89, Upst90]. The shapes of objects, as well as their positions, can change during an animated sequence. The modeling software interprets the scene description and generates a complete model file for each frame.

Reyes reduces all primitives to a common internal representation, called *micropolygons*. Micropolygons are very small triangles or quadrilaterals; by default they are no longer than one pixel-width in any direction. Because a micropolygon is very small, it can be shaded with a single flat color. Thus, the rendering parameter that limits the size of micropolygons controls the shading frequency. Adaptive subdivision routines ensure that all micropolygons are sufficiently small and flat. Each set of micropolygons closely approximates the original object. To support a higher-level modeling primitive, an implementer provides a module that either subdivides the primitive into micropolygons or reduces it to other primitives that Reyes can subdivide.

Micropolygons are created in two-dimensional meshes called *grids*. To improve texture locality and support vectorized shading, Reyes shades all micropolygons in a grid at one time, before determining visibility. Consequently, some invisible micropolygons are shaded. The designers of Reyes believed that the savings from improved shading coherence would outweigh the costs of any unnecessary shading operations. The amount of extra shading work is a function of the scene's depth complexity. If the depth complexity is low, fewer objects are hidden and more of the shaded surfaces are displayed.

The system supports an extensible shading language, which describes surfaces, light sources, and textures. The Reyes shading language embeds operations in graphs called *shade trees* [Cook84a], while prman supports the RenderMan Interface shading language [Hanr90]. The cost of shading a single micropolygon can vary greatly, because the illumination model is not constant. The number of texture maps, the number of light sources, and the light source complexity also change from surface to surface and from scene to scene, with considerable impact on the cost.

3.3.1. Visibility Determination in reyes

Reyes determines visibility with point sampling and stores sampling results in a modified z-buffer. A stochastic supersampling scheme developed by Cook [Cook86] supports high-quality antialiasing. It subdivides pixels into uniform regions. Each sub-pixel has one sample point, offset from the region's center by random amounts in x and y. To support transparency and constructive solid geometry, each z-buffer entry contains a list of all micropolygons that intersect the corresponding sample point.

Figure 3.2 illustrates the three processing stages of the visibility algorithm: sampling, depth sorting, and filtering. On average, each stage of the algorithm reduces the size of the input passed to the next stage.

The first stage discovers which sample points intersect, or *hit*, a micropolygon. Each micropolygon is associated with a *bounding box*, which contains the rectangular area defined by its minimum and maximum x and y coordinates. The algorithm samples each micropolygon over its bounding box, testing for intersection with each sample point. When a sample hits a micropolygon, the surface is potentially visible at that sample point. The output from the sampling stage is one list per sample point, consisting of all micropolygons hit by the sample point.

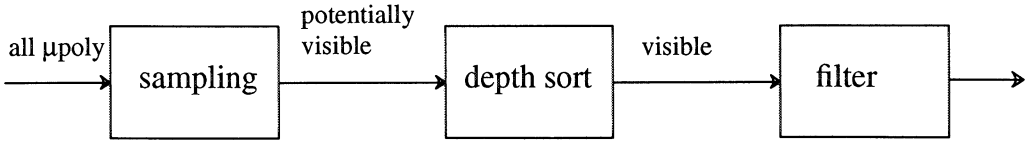


Figure 3.2. Visible surface processing stages.

The second stage sorts each list by increasing depth into the scene. In general, the first micropolygon on the list is visible. If it is transparent, any objects behind it are also visible, but only to the degree that the front object transmits light. The algorithm processes the list iteratively until it finds an opaque object that blocks the view of any obj behind it. Its output is a list of all micropolygons that are visible at the sample point and contribute to the pixel's color.

The final processing stage traverses the visible micropolygon list to determine the contribution of each micropolygon to the final color of the pixel. For high-quality rendering, a weighted filter considers the sample point's location within the pixel to determine its contribution both to the current pixel and to the surrounding pixels.

The Reyes visible surface algorithm could, theoretically, process objects in an arbitrary order, but it would have to maintain a prohibitively large amount of data until each frame is completed. Instead, a spatial subdivision scheme reduces memory costs and improves locality. The system partitions the screen into rectangular *buckets*, typically four to sixteen pixels square. It sorts objects into buckets and processes them from top to bottom and from left to right. After rendering a bucket, Reyes can discard most of the micropolygons. If a surface falls into more than one bucket, its micropolygons are added to the list for the next bucket that they intersect. Spatial subdivision saves memory by restricting the size of the z-buffer and by reducing the amount of time that micropolygons are retained.

3.3.2. Differences Between Reyes and Prman

Although Reyes and prman are closely related, some differences affect the image complexity analysis. Both systems are written in C, but they have been implemented on different hardware and under different versions of the Unix operating system. Thus, we can neither compare runtime measurements nor profile the systems with the same tools. In addition, their shading and texture mapping subsystems use different algorithms and report different statistics. However, the model files contain equivalent information even though the model languages differ.

PhotoRealistic RenderMan (specifically, prman Version 3.1) introduced improvements to the visible surface algorithm that affect the complexity measurements substantially. An approach called *invisibility processing* reduces the performance problems associated with shading before determining visibility. The goals of invisibility processing are to avoid shading invisible objects and to reduce the size of the visible surface problem. The new algorithm bounds the depth of all objects and uses the estimated depths for a preliminary sort [Apod91]. Then, the system processes the objects in an order that is roughly front-to-back. Tests at three stages of rendering detect and discard many of the invisible surfaces before performing the complete visible surface algorithm. The first test checks micropolygon meshes (called *grids*), the second checks individual micropolygons, and the third checks individual samples. The tests rely on estimated bounds that may be loose but are always correct. They might pass an invisible object on for further processing, but they never reject something that is visible.

The system tests for invisible grids after subdividing a surface but before creating the grid data structure. The algorithm bounds the grid in x, y, and z and tests the z-buffer over the entire x, y bounding box. It compares the existing values in the zbuffer with the grid's minimum z bound. If the bounding box is completely obscured by opaque objects, the grid is discarded. The surviving grids may be entirely visible, partially visible, or invisible. Prman shades them all, but before sampling it tests for invisible micropolygons. It discards a micropolygon if its entire bounding box is obscured. Finally, prman tests each sample point before calculating its intersection with any micropolygon. If the value in the z-buffer is closer than the micropolygon's z bound, the sample is discarded. Measurements in Section 3.4.3 characterize the proportion of objects that prman discards at each invisibility test.

Because of these optimizations to the visibility algorithm, prman cannot produce the same statistics as Reyes. Consider, for example the *depth complexity*, which is the number of objects, both visible and invisible, intersected by a ray projected through the image plane. For Reyes, the depth complexity at a sample point is the same as the number of micropolygons hit at that point. Reyes can report the depth

complexity at every sample point, because it samples each micropolygon over its entire x, y bounding box. In contrast, prman discards many objects before sampling. It only guarantees to hit the visible surfaces, although it may also intersect some or all of the hidden surfaces. Thus, the number of micropolygons that prman hits at a given sample point is only a lower bound for the true depth.

Although the statistics vary, this chapter contains similar groups of measurements for images produced by Reyes and prman.

3.4. Measurement Results

A set of images were selected to represent the range of geometric complexity, applications, and visual style found in the actual animation workload at Pixar (Table 3.1). Reproductions of the images appear in Section 3.4.2. Because the availability of statistics depends on the rendering system, the images are divided into groups that we will discuss separately. Group I contains images that were rendered by Reyes; the corresponding models were created between 1984 and 1987. Group II contains images that were rendered with prman between 1989 and 1991. One 1988 scene, that depicts a character named Tinny, comes from a transitional period and belongs in neither group. It was defined in an early version of the RenderMan language, but rendered with the multi-processor implementation of Reyes that is discussed in Chapter 5. The runtime statistics available for this scene are incompatible with the statistics for both Group I and Group II, so they are not included in this chapter.

Two of the Group II scenes correspond to Group I scenes: Luxo Jr. and the bike shop window. Although these models date from 1986 and 1987, their creators still consider them representative of the workload. The Luxo Jr. model was used in new animation projects in 1989 and 1991. Pixar personnel converted the bike shop to the RenderMan format in 1991 for use in performance studies. Some changes to the geometry were introduced when the models were converted to the new format. Although the pairs of corresponding scenes are not exactly alike, they are similar enough for general comparisons.

Group II also contains two related views of the bike shop. In one, the camera is on the street looking through the shop's front window. In the other, the camera is inside at the front of the shop, looking towards the back wall. The first scene uses a less detailed model of the shop's contents, but its brick wall and glass window add an extra layer of depth.

Most of the images are stills from short animated films. One scene, the stained glass knight, was a special effect in the feature film *Young Sherlock Holmes* (1985). Another, conga, is a frame from a television commercial.

Some of the models are characterized by simple geometry with complex shading or heavy texturing. These include Andre, the stained glass knight, Luxo Jr., and conga. The Luxo Jr. scenes are inherently simple, but the rendered images have a convincing realism. Shadows contribute significantly to the realistic appearance. Another scene, waves, has complex geometry and moderately complex shading. Both the geometry and the shading are complex in the bike shop scenes. The final model, Tinny, is a relatively simple element that was combined with textured backgrounds and other, more complex, objects.

Group I	Group II	multi-processor
andre		tinny
luxo jr. I	luxo jr. II	
bike shop window I	bike shop window II	
waves	bike shop interior	
stained glass knight	conga	

Table 3.1. Images in the test suite.

3.4.1. Complexity Metrics

Chapter 2 identifies four classes of parameters that affect rendering costs: scene characteristics, view-dependent characteristics, rendering parameters, and the computing environment. Most of the rendering parameters and characteristics of the computing environment do not indicate the intrinsic complexity of the images. Because they do affect many of the measurements, they are listed in Table 3.2.

This chapter characterizes the test suite in terms of the relevant scene characteristics and view-dependent characteristics. Some scene characteristics are described as static properties of the model database: the number and type of modeling primitives, the number and type and size of light sources, and the number and types of texture maps. The description of each scene also describes in general terms the animation characteristics, transparency, shadows, and reflections. Because the visible surface algorithm operates on screen coordinates, world-space dimensions are not considered. View-dependent characteristics are measured dynamically by rendering system instrumentation. These include the depth complexity, the screen size of objects, and measurements about visible or relevant objects. The renderer also documents certain properties of the internal rendering primitives,

3.4.2. Model File Statistics

Characteristics of the model files are shown in Table 3.3, which also describes the applications from which the scenes were taken. Figure 3.3 contains black and white reproductions of the corresponding images, which give a feeling for their content. Without high-quality color photographs it is impossible to show the detail and shading complexity of the original images. The table cites publications and films that contain more accurate color reproductions of these, or related, images as well as publications that describe the related graphics research. The suite contains two pairs of similar images. Although the two Luxo Jr. images are not exactly alike, they are very similar, and only the Group I version is shown, in Figure 3.3(d). The bike shop window is also included only once, in Figure 3.3(e).

Each entry in Table 3.3 describes the characteristics of the three-dimensional model of a scene, not its representation as a two-dimensional image. Some of the Group I models contain information that is not required for all renderings of the scene, such as texture maps that are visible only from certain viewpoints. Table 3.3, however, describes all of the model's data.

Some of the texture maps, particularly those used to simulate shadows or reflections, may be computed in separate rendering passes. When discussing model complexity, this chapter ignores any auxiliary data or computation that may be needed to create such texture maps.

Collectively, the scenes use four standard types of light sources. Every scene has *ambient light*, a non-directional source with only an intensity and a color. A *distant light*, such as the sun, has a direction but no position. Its light rays are parallel, because the light is cast in a single direction. A *point light* source is local. Its light is cast from a single point equally in all directions, but it falls off with increasing distance from the source according to an inverse square law. A *spotlight* is a type of point light source that has both a position and a direction. It casts light within a cone, and the light is most intense in the center of the cone.

	Group I	Group II
Computer system	CCI Power 6/32	Silicon Graphics 4D/220
Micropolygon size	≤ 1 pixel in any direction	≤ 1 square pixel
Bucket size	4x4 pixels	4x4 pixels
Screen size	1024x614	1024x614
Samples/pixel	16, in a jittered 4x4 grid	16, in a jittered 4x4 grid
Backfacing objects	culled	culled

Table 3.2. Computing environment and rendering parameter values.

andre (cartoon character)	
Year, references	1984 [Cook86, Lass87]
Application	research in animation, rendering, and modeling of natural phenomena short animated film
Visual properties	simple surface textures
Animation	character moves and changes expressions, but remains seated shapes change camera static
Total modeling primitives	91 (character only, excluding background)
Description of primitives	74 spheres 2 cones 13 cylinders 24 bicubic patches 2 "teardrops" 13 constructive solid geometry (csg) operations
Texture maps	11 surface color
Light sources	1 distant light (the sun), no shadows

Table 3.3(a). Characteristics of the Andre model.

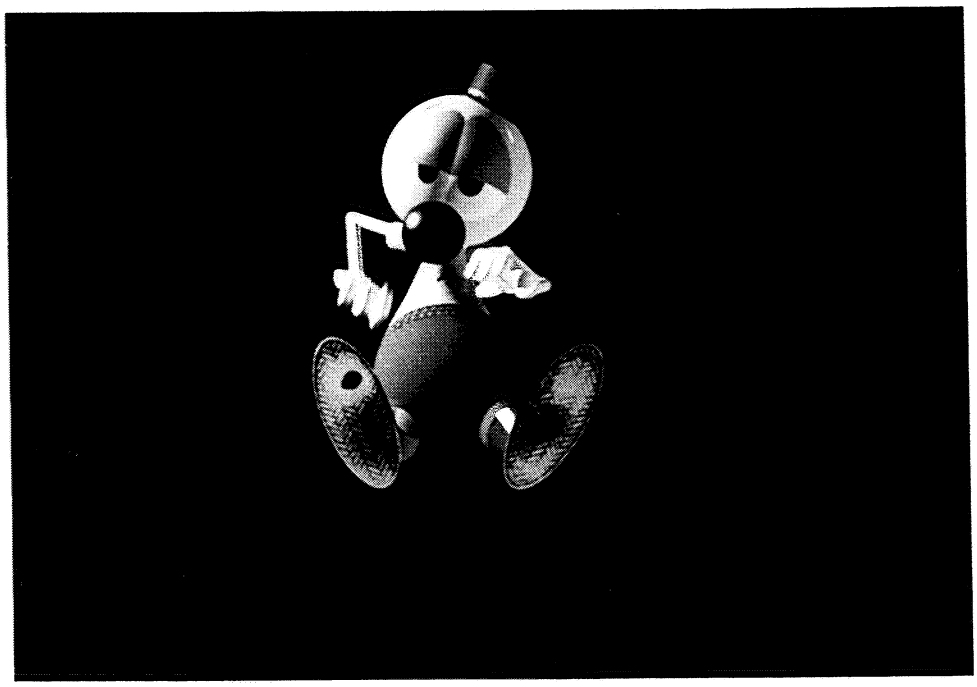


Figure 3.3(a). Andre. ©1984 Pixar

stained glass knight	
Year, references	1985 [Cook86, Fole90, Plate IV.24]
Application	special effect for a feature film [Para85], composited with live action
Visual properties	transparency, refraction many surface textures surface normal perturbations (bump maps) many light sources
Animation	character walks towards camera, moves body parts static shapes camera tilts up (rack focus in original, but not in instrumented runs)
Total modeling primitives	1,403
Description of primitives	1,403 cubic bezier patches (describing 82 pieces of thick glass, 6 surfaces each)
Texture maps	51 (each object needs 2 bump maps, 1 background, and 2 or 3 colors)
Light sources	15 spotlights, no shadows

Table 3.3(b). Characteristics of the stained glass knight model.

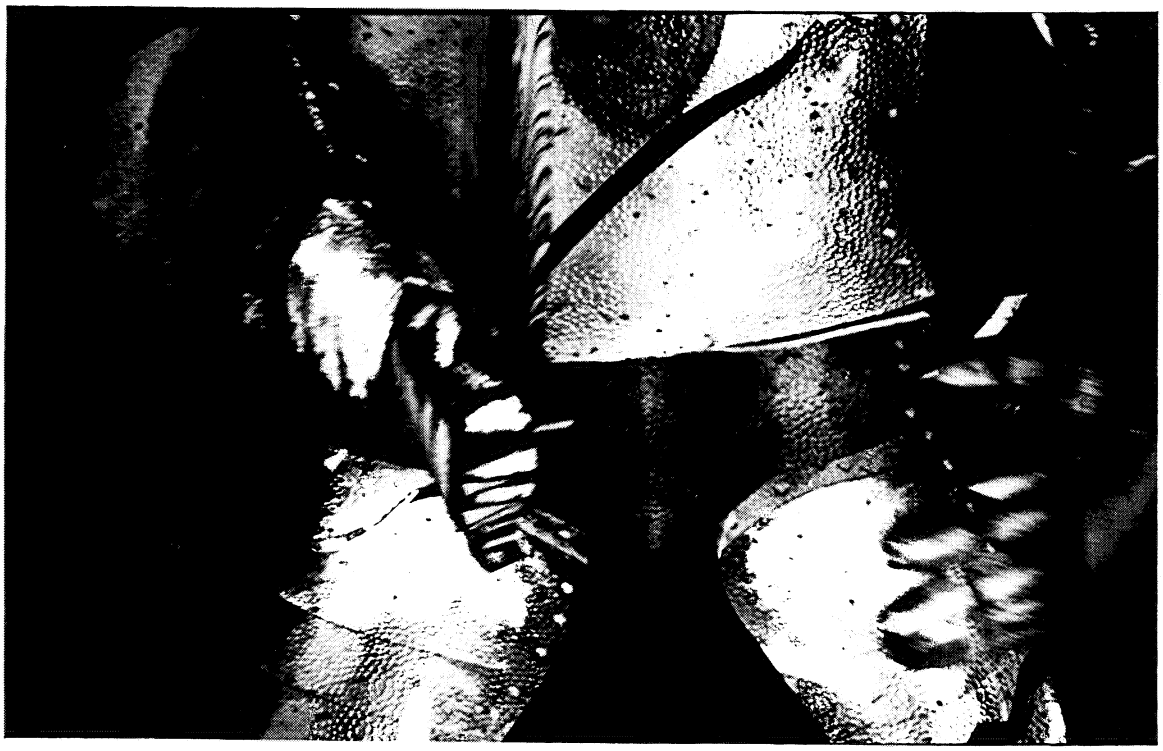


Figure 3.3(b). Stained glass knight. © 1985 Paramount Pictures Corporation and Amblin' Entertainment

waves	
Year, references	1986 [Four86] and [Fole90, Plate IV.19]
Application	research in modeling natural phenomena short animated film
Visual properties	complex geometry simulated reflections surface normal perturbations (bump maps)
Animation	waves roll in and crash on beach; beach and cliffs are static primitives move and change shape with each frame vertical camera move
Total modeling primitives	54,440
Description of primitives	45,366 bicubic patches 8,580 fractal specifications 494 bilinear patches
Texture maps	2
Light sources	1 distant light (sun), no shadows

Table 3.3(c). Characteristics of the waves model.

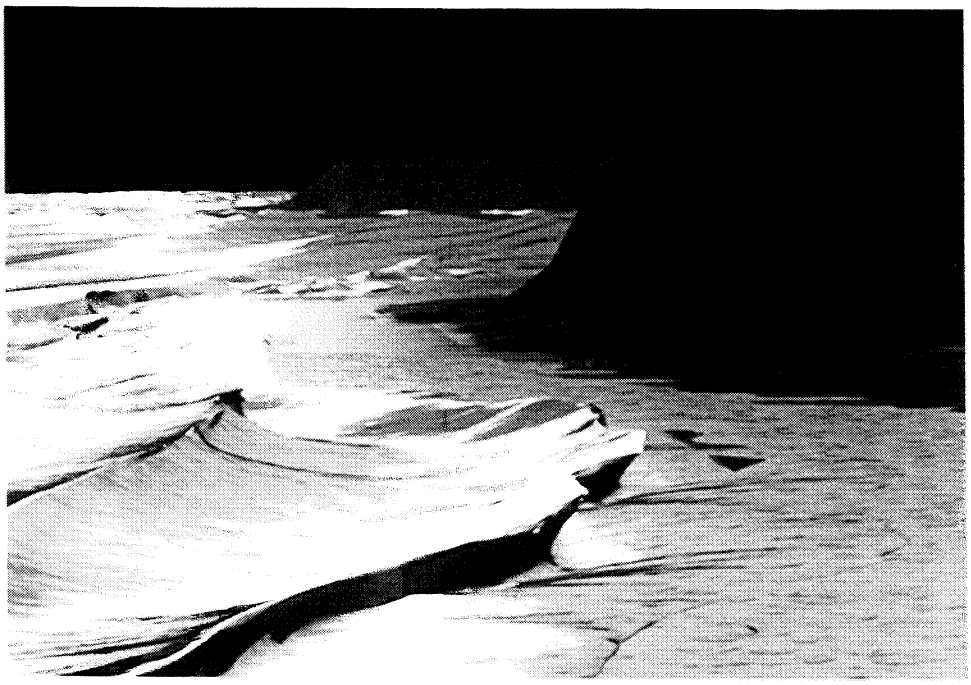


Figure 3.3(c). Waves. ©1986 Pixar

luxo jr. I	
Year, references	1986 [Lass87] and [Fole90, Plate D]
Application	rendering research (shadow map algorithm [Reev87]) short animated film
Visual properties	shadows from three light sources simple surface texture (no reflections)
Animation	two lamps move, ball rolls, lamp cord moves cord shapes change, but lamp and floor geometry static camera static
Total modeling primitives	454
Description of primitives	51 spheres 78 cones 54 cylinders 9 tori 8 ''teardrops'' 254 bilinear patches
Texture maps	4
Light sources	1 spotlight off-camera, casts shadows 2 spotlights visible in scene, cast shadows

Table 3.3(d). Characteristics of the Luxo Jr. I model.

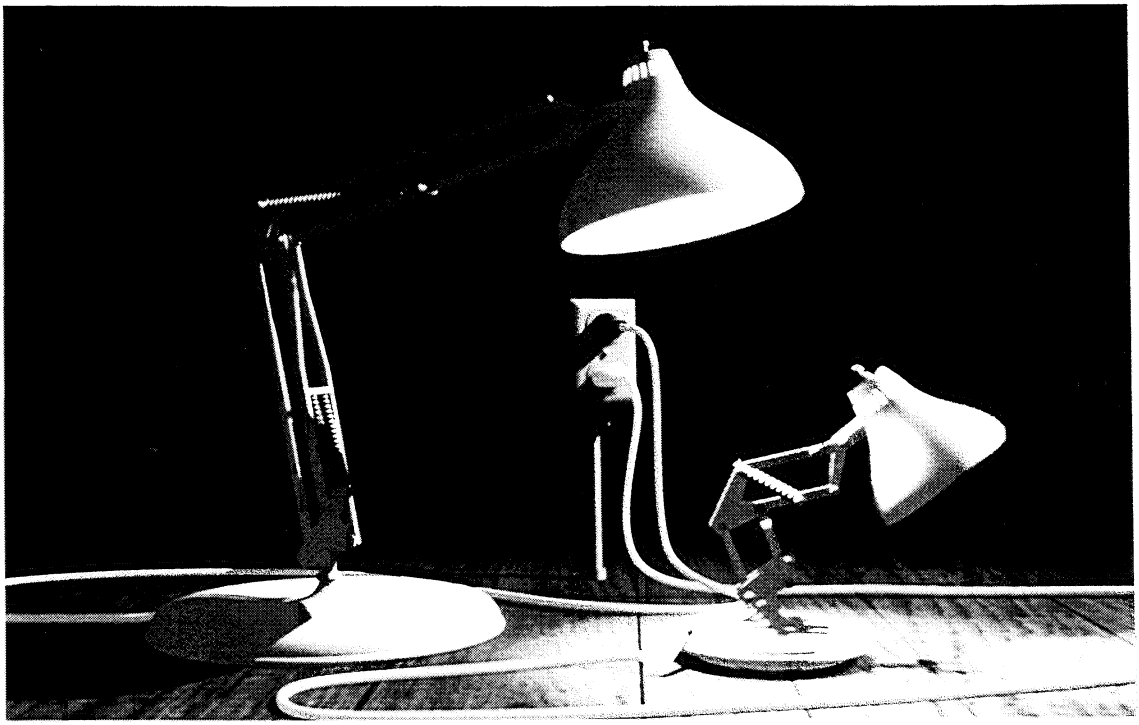


Figure 3.3(d). Luxo Jr. I. ©1986 Pixar

bike shop window I	
Year	1987
Application	animation and rendering research and development research in modeling natural phenomena short animated film, *Red's Dream*
Visual properties	complex geometry transparency, surface textures, simulated reflections sophisticated lighting and shadows
Animation	geometry static; camera tilts up and lighting changes
Total modeling primitives	10,406
Description of primitives	334 spheres 541 cones 3,274 cylinders 1,683 tori 1,300 cubic Bezier patches 3,206 bilinear patches 68 "teardrops"
Texture maps	22
Light sources	2 spotlights, cast shadows 3 spotlights, no shadows 1 neon sign, modeled as bi-directional spotlight simulated moonlight, modeled as a distant light

Table 3.3(e). Characteristics of the bike shop window I model.

Figure 3.3(e). Bike shop window (I and II). ©1987 Pixar

tinny	
Year, references	1988 [Upst90, Plates 5 and 6]
Application	short animated film
Visual properties	shadows, including self-shadowing simple surface textures
Animation	in this scene, character marches across screen camera static
Total modeling primitives	1,510
Description of primitives	113 spheres 217 cylinders 350 hyperboloids 14 tori 545 bicubic patches 271 bilinear patches
Texture maps	2 surface textures
Light sources	2 distant lights

Table 3.3(f). Characteristics of the Tinny model.

Figure 3.3(f). Tinny. ©1988 Pixar. The measurements include only the figure at the lower left.

luxo jr. II (revised model)	
Year, reference	1989, 1991 [see Figure 3.3(d) and Table 3.3(d)]
Application	short animated films
Visual properties	adds simulated reflections to original described in Table 3.3(d)) adds more complex spring geometry stereo 3D in 1988 film; video format for 1991 film
Animation	similar to original; 1988 film has one lamp and one ball; statistics bem
Animation	similar to original; 1988 film has one lamp and one ball; statistics below describe 1991 film with two lamps and two balls
Total modeling primitives	4,445
Description of primitives	24 cylinders 74 hyperboloids 46 spheres 8 tori 4,036 bicubic patches 257 bilinear patches
Texture maps	6
Light sources	same as Table 3.3(d) 1 spotlight off-camera, casts shadows 2 spotlights visible in scene, cast shadows

Table 3.3(g). Characteristics of the Luxo jr. II model.

bike shop interior	
Year, references	1987, rendered 1991 [Reev87] and [Fole90, Plate F]
Application	still from 1987 animated film, *Red's Dream* used in 1991 as a benchmark for machine comparisons
Visual properties	complex geometry transparency, surface textures sophisticated lighting and shadows
Animation	n.a.
Total modeling primitives	18,258
Description of primitives	690 spheres 3,136 cylinders 1,839 tori 883 hyperboloids 1,030 cubic Bezier patches 10,680 bilinear patches 4 constructive solid geometry (csg) intersections
Texture maps	13
Light sources	2 spotlights, cast shadows 5 spotlights, no shadows

Table 3.3(h). Characteristics of the bike shop interior model.

Figure 3.3(g). Bike shop interior. ©1987 Pixar

bike shop window II	
Year, reference	1987, rendered 1991 [see Figure 3.3(e) and Table 3.3(e)]
Application	still from 1987 animated film, *Red's Dream* used in 1991 as a benchmark
Visual properties	complex geometry transparency, surface textures, simulated reflections sophisticated lighting and shadows
Animation	n.a.
Total modeling primitives	11,311
Description of primitives	334 spheres 526 hyperboloids 3,328 cylinders 1,683 tori 2,233 cubic Bezier patches 3,207 bilinear patches
Texture maps	12
Light sources	2 spotlights, cast shadows 3 spotlights, no shadows 1 neon sign, modeled as bi-directional spotlight simulated moonlight, modeled as distant light

Table 3.3(i). Characteristics of the bike shop window II model.

	conga
Year	1990
Application	television commercial
Visual properties	simple geometry displacement mapping transparency, surface textures, simulated reflections large number of light sources
Animation	dancing gummy candies; geometry "bends" camera slowly moves upwards; lighting changes
Total modeling primitives	968
Description of primitives	18 spheres 64 cylinders 166 tori 168 hyperboloids 533 cubic Bezier patches 19 bilinear patches
Texture maps	9
Light sources	1 spotlight, casts shadows 15 spotlights, no shadows 15 point light sources 1 distant light

Table 3.3(j). Characteristics of the Conga model.

Figure 3.3(h). Conga. Pixar/Colossal Pictures 1991

Some images, especially the early ones, cover only part of the screen. This reflects a model of computing pictures that relies on compositing separate elements to create a complete image [Duff85, Port84]. Subdividing the scene into smaller subproblems can save memory or compute time. Backgrounds and other elements may be computed and rendered using different techniques, and rendered elements can be used again in different settings. The stained glass knight inherently follows this model, because the computer graphics elements were designed to be composited with live action film footage. Composited elements must be separable. Partitioning the scene and resolving visibility among the elements is usually a manageable manual task, but this requirement imposes design constraints.

To what extent has the workload become more complex over time? Andre is the earliest and simplest scene of the test suite, but it was created while the renderer was under development. The other scenes were created with more mature rendering software. In general, image complexity depends more on the requirements of the application than on the year created.

Figure 3.4 plots the number of primitives in the scenes, and compares the test suite with data from the literature. There are both large and small models throughout the time period shown in the graph. However, the RenderMan versions of Luxo Jr. and the bike shop do contain more primitives than the earlier versions, mostly because the geometry of the lamp's springs is more exact. (There is a Luxo lamp on the bike shop's counter.) Better modeling tools, as well as increased rendering capacity, encouraged this improvement in the model.

Shading complexity also depends on the needs of the application. The greatest number of light sources are seen both in one of the earliest images, the stained glass knight in 1985, and in one of the latest, conga in 1990. The use of texture maps has changed little over the years, except for the introduction in 1986 of an algorithm that uses texture maps to simulate shadows [Reev87]. Bump maps to vary

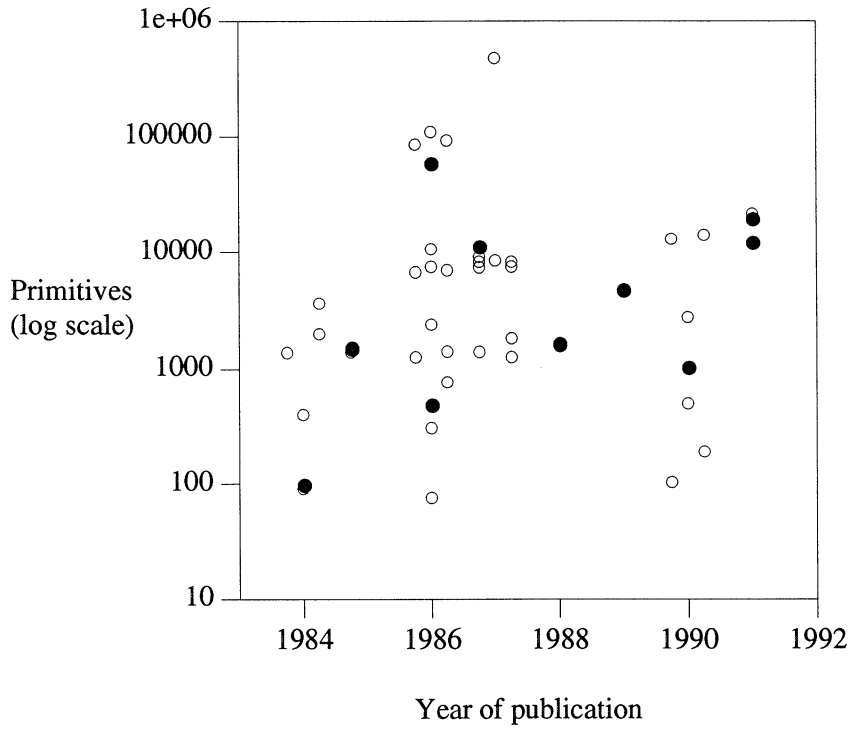


Figure 3.4. Number of modeling primitives. The figure compares the Pixar test suite with other models in the literature. Data points corresponding to the test suite are marked with bullets: ●. The mixed models data from Figure 3.1 are marked with circles: ○.

the surface normal, environment maps to simulate reflections, and displacement maps to offset the location of points on the surface were all available in Reyes as early as 1984 [Cook84a]. However, these applications of texture mapping do not appear in the test suite until later.

In summary, the test images exhibit a general consistency in their complexity. One explanation is that they were all created by a family of tools with similar capabilities and approaches to rendering.

3.4.3. Rendering-time Model Statistics

The rendering system gathers information dynamically about certain properties of the scene, including

- its internal representation in rendering primitives,
- the screen size of objects,
- the proportion of the screen covered by the image, and
- the number of objects visible at different locations on the screen.

Pixar's production rendering systems report statistics about rendering primitives and provide limited instrumentation of the visible surface algorithm. I added measurement code to Reyes that reports more detailed information about objects in screen space and about the performance of the visible surface algorithm. The instrumentation is contained in a visible surface module that substitutes for the default code. It can produce statistics with or without generating an image. For all statistics, the code records not only the mean but also the standard deviation and a histogram. Variability is important because it affects how evenly the workload may be distributed among multiple processors. I did not add the equivalent instrumentation to prman, because of the differences in the visible surface algorithm. Since prman does not sample all objects, it cannot provide complete information about objects in screen space.

The additional instrumentation in Reyes measures characteristics of both the visibility algorithm and the scenes. The code measures the number of objects processed at each sample point and in each bucket during the three stages of the visible surface algorithm. It measures, as well, the number of visibility operations applied to each micropolygon. The depth complexity statistics are obtained in an unusual manner, by sampling the depth at a large number of locations distributed pseudo-randomly throughout the screen area. This instrumentation piggybacks on the stochastic sampling code. For an image with many sub-pixel size features, supersampling indicates the depth of the scene more accurately than counting the number of objects per pixel.

Table 3.4 shows some of the rendering statistics for Group I, and Table 3.5 shows the corresponding information for Group II. Because of differences in the rendering system, some of the statistics are computed differently for the two groups. We will consider the table entries in order, from top to bottom.

Resolution. All of the images were computed at the same resolution except for Luxo Jr. II. It has a different aspect ratio because it was computed for video rather than for film.

%Coverage. For Group I, the screen coverage is defined as the percentage of *sample points* that have at least one hit. This percentage was determined by instrumenting the sampling code. In production, Andre and waves were composited with other computer-generated elements, the stained glass knight was composited with live-action film, and the other images contained some black background. All of the images except the bike shop have many empty pixels. For Group II, the screen coverage is defined as the percentage of *pixels* that contain something other than empty background. This percentage was determined by a program that examines the image data. The Group II images are all complete frames, which indicates a trend away from compositing. Although compositing saves rendering time, it requires extra human intervention and can introduce errors. It is also more difficult in scenes with shadows. With faster rendering and the more frequent use of shadows, compositing has become less common.

Sample Hits. The total number of sample hits is a function of the image geometry, the sampling rate, and the image resolution. For Group I, the number of hits is proportional to the total screen area of

Image	andre	stained glass knight	waves	luxo jr. I	bike shop window I
Resolution	1024x614[6]	1024x614[6]	1024x614	1024x614	1024x614[6]
%Coverage (sample points)	15.49	67.10	51.54	36.71	95.87
Sample Hits	4,940,937	10,814,982	9,144,470	5,474,264	21,682,346
Depth Complexity					
covered sample points	3.17	1.58	1.75	1.47	2.48
all sample points	0.49	1.08	0.91	0.54	2.47
Micropolygons	485,531	819,089	1,738,291	777,607	3,065,513
Hits per Micropolygon	10.18	13.20	5.26	7.04	7.07
Modeling Primitives	91	1,403	54,440	454	10,406
Pixels per Primitive (est.)	3,394	482	11	754	130

Table 3.4. Rendering statistics, Group I.

Image	luxo jr. II	bike shop interior	bike shop window II	conga
Resolution	1024x768	1024x614	1024x614	1024x614
%Coverage (pixels)	45.40	100.00	100.00	100.00
Sample Hits	5,890,317	14,010,635	12,866,921	15,790,568
Hits per sample point				
covered sample points	1.03	1.39	1.28	1.57
all sample points	0.59	1.39	1.28	1.57
Micropolygons Shaded	806,187	2,152,432	1,344,626	2,573,665
Micropolygons Sampled	744,200	1,896,737	1,268,163	2,487,586
Hits per Micropolygon	7.92	7.39	10.15	6.35

Table 3.5. Rendering statistics, Group II.

[6] Because of memory limitations, the left and right halves of these three images were computed separately and composited by a post-processing task.

all surfaces in the scene. This statistic is an important property of the scene, because the cost of a z-buffer algorithm increases in direct proportion to the total screen area. The number of hits varies by about a factor of four for Group I. We can not use the number of hits to estimate the total projected area of the Group II scenes, because prman does not sample all surfaces. For these images, the number of hits measures the renderer's effort rather than any intrinsic property of the scene. There is a large difference in the number of hits between versions I and II of the bike shop window. This difference demonstrates prman's ability to discard many objects that are hidden behind the front wall of the shop. Between versions I and II of Luxo Jr, the number of hits increased by about eight percent. However, the screen coverage increased by over twenty percent.

Depth Complexity. A useful way to think about the total projected area of a scene is to average the number of hits over the sample points. The quotient is the mean sample point depth complexity, that is, the average number of surfaces intersected by a ray projected from a sample point into the scene. The depth complexity is an important property of the scene, because it estimates the amount of extra shading performed by a renderer that shades all surfaces, visible or not.

For Group I, the depth complexity is computed as the average number of hits per sample point. Table 3.4 shows two depth complexity statistics for each image. The first ignores the background, and averages the depth over the non-empty sample points. For three of the five images, the mean is less than two, and it is always less than four. Table 3.6 shows the median and other percentiles of this distribution. Some sample points do have a large number of hits, but they are rare. The second depth complexity statistic in Table 3.4 averages the number of hits over all sample points in the image, including the empty background.

Although these images are generally considered complex, their depth complexity is low. Does the low depth complexity reflect inadequate rendering capacity or the difficulty of modeling complex scenes? Modeling is labor-intensive, so scene designers tend to avoid modeling objects that would be hidden behind other elements of the scene.

Depth complexity measurements are rare in the literature, but Whelan gives distributions for six images [Whel85]. The median for his images ranges from one to five, and it is greater than three in half of the images. The ninetieth percentiles range from two to over thirty. Although Whelan reports higher depth complexities than Table 3.6, his images do not appear to have much more depth. Two factors help explain the differences between his measurements and Table 3.6. First, Whelan measured the *pixel* depth complexity, which counts all objects that intersect any part of a pixel. Suppose a pixel contains parts of three non-intersecting fragments. The pixel depth complexity is then three. This chapter reports a statistic that depends less on the image resolution, the *sample point* depth complexity. If a pixel contains three non-intersecting fragments, the mean sample point depth complexity is at most one, since a ray projected from a sample point hits either one of the fragments or none. Furthermore, the mean sample point depth complexity can be less than one if the fragments do not completely cover the pixel. The second difference in the measurements is that Whelan's scenes apparently contain an explicit background polygon. Each pixel then has a minimum depth complexity of one. The Pixar images, which contain no explicit background plane, have a minimum depth complexity of zero.

Hits per Sample Point. For Group II, there are no depth complexity estimates. When prman discards an invisible object before sampling, it loses information about the true depth of the scene. The mean hits per sample point is a lower bound that underestimates the true depth of the scene. As expected, the means for the Group II images are lower than for Group I.

Micropolygons. The rendering system also reports on rendering primitives. For Group I, Table 3.4 shows the number of micropolygons generated for each image. For Group II, the table shows both the number of micropolygons that are shaded and the number that are later sampled after prman discards some invisible micropolygons.

Image	median	90th percentile	99th percentile	maximum
andre	2	8	14	16
stained glass knight	1	3	6	15
waves	2	3	6	40
luxo jr. I	1	2	4	8
bike shop window	2	4	8	27

Table 3.6. Sample point depth complexity, Group I. The percentiles exclude sample points in empty background pixels.

The number of micropolygons ranges from half a million to three million. In other words, the rendering primitives are commonly about three orders of magnitude more numerous than the input modeling primitives. These numbers are large in comparison with the number of intermediate primitives reported for some other systems. Two examples in the literature describe systems that render large numbers of small intermediate primitives, but in neither case are the numbers of primitives as consistently large. Snyder and Barr tessellated surfaces into small triangles for ray-tracing [Snyd87]. One scene generated 4×10^{11} intermediate primitives but most had fewer than half-a-million triangles. Reeves' particle systems algorithm generated many simple, tiny primitives [Reev83, Reev85]. The most complex contained nearly two million particles, but most had fewer than one million primitives.

Hits per Micropolygon. Reyes subdivides each pixel into sixteen equal regions and places one sample point pseudo-randomly within each sub-pixel. The stochastic sampling algorithm can be use to estimate the screen size of micropolygons in the Group I images. The algorithm guarantees one hit for each sub-pixel that is completely covered by a micropolygon, and on average the number of hits is proportional to the size of the object. The maximum micropolygon size is controlled by a rendering parameter, which limits the longest dimension to the width of one pixel. The minimum size depends on the shape of the surface. The renderer subdivides objects until the resulting micropolygons are flat. Micropolygons can be very small if the shape of a surface requires a fine subdivision or if the surface itself is very small. In the test suite, the mean number of hits per micropolygon varies by more than a factor of two, ranging from five to thirteen.

The number of hits per micropolygon is computed the same way for Group II. However, because of the optimizations that avoid unnecessary sampling, this statistic underestimates the screen size of objects. Rather than describing the intrinsic complexity of a scene, this statistic describes the amount of work performed by prman.

Coarse polygonal approximations of curved surfaces tend to produce unrealistic, faceted scenes. Because Reyes generates such a large number of tiny micropolygons, it can represent geometric detail accurately. This level of accuracy requires a large amount of data and many operations.

Pixels per Primitive. For Group I, Table 3.4 gives an estimate of the mean screen size of modeling primitives. To compute the estimate, first divide the total number of sample hits by the number of modeling primitives. Then, divide the quotient by the number of samples per pixel. Because back-facing surfaces have been culled, the result estimates the average projected screen area of the modeling primitives. Table 3.5 omits the equivalent statistic for Group II because the sampling optimizations do not preserve enough information to produce a meaningful estimate. The average number of pixels per primitive varies greatly, by a factor of three hundred (from 11 to 3,394). Much of the variation is due to the different requirements of the applications. For example, the largest primitives are in the images that show close-up views of a few objects. However, the statistics also hint at a trend towards modeling with smaller primitives. Andre has the largest primitives, but it was designed to be a simple first production with the rendering software. Luxo Jr. I has large primitives, and it was also given a simplified design. The later version,

Luxo Jr. II, was modified to to represent the geometry more exactly; it has nearly ten times as many primitives. If the earlier version had the same representation, it would average about one hundred pixels per modeling primitive instead of 754. Improved modeling tools, as well as increased rendering capacity, have made more detailed models practical.

3.4.4. Rendering-time Texture Statistics

The rendering system also provides information about the dynamic use of texture maps, as shown in Table 3.7. Five different types of texture maps were used: surface textures, bump maps, shadow maps, reflection maps, and environment maps. As many as four types were used in a single scene. The number of distinct texture maps defined for a scene varied from two to fifty-one, but not all of the texture maps were visible in the views that were rendered.

Image	Texture Type	Number of Maps		Texture Pixels	
		defined	used	accessed	per shaded μp
andre	surface color	11	11	n.a.	
stained glass knight	surface color	48	26	n.a	
	bump map	2	2	n.a	
	background color	1	1	n.a	
waves	bump map	1	1	n.a.	
	reflection map	1	1	n.a.	
luxo jr. I	surface color	1	1	n.a.	
	shadow map	3	3	n.a.	
bike shop window I	surface color	17	14	n.a.	
	shadow map	2	2	n.a.	
	reflection map	3	1	n.a.	
luxo jr. II	surface color	2	2	4,083,613	5.07
	shadow map	3	3	11,213,746	13.91
	reflection map	1	1	7,032,033	8.72
	total			22,329,392	27.70
bike shop interior	surface color	11	11	12,724,655	5.91
	shadow map	2	2	15,541,683	7.22
	total			28,266,338	13.13
bike shop window II	surface color	9	9	8,272,392	6.15
	shadow map	2	2	8,698,121	6.47
	reflection map	1	1	3,453,864	2.57
	total			20,424,377	15.19
conga	surface color	6	6	2,435,799	0.95
	shadow map	1	1	3,286,309	1.28
	reflection map	1	1	10,144,884	3.94
	displacement map	1	1	6,386,704	2.48
	total			22,253,696	8.65

Table 3.7. Texture usage statistics. For images in Group II, the table shows the total number of texture pixels used and the average number of texture pixels per shaded micropolygon. This information is not available for the images in Group I.

Unfortunately, the texture statistics supply only aggregate data for each texture map, without describing the distribution of texture requests over different surfaces. Two examples show how texture usage can vary with the application. For Andre, each texture map adds a pattern to one object, such as the tread on the sole of the shoes. In this scene, a single surface receives at most one texture and many surfaces are not textured. In the stained glass knight, every surface uses five or six varied maps to create a richly-textured appearance.

Reyes supports efficient access to texture data by caching texture pages obtained from the server or local disk. Overall, the Reyes texture statistics document the performance of the texture cache. They are helpful in tuning the system but have little to say about the system-independent, intrinsic properties of the scene. Table 3.7 simply notes that two Group I models, the ones with the most texture maps, define more textures than the specified views require. Prman's texturing subsystem provides a different set of statistics, including information about the number of pixels accessed from each texture map. Table 3.7 sums the number of pixels accessed by the type of texture map. It also averages the number of texture pixels over all of the shaded micropolygons and summarizes the statistics for each scene. Depending on the surface, the actual number of texture accesses may vary greatly from the mean.

There are some differences between the Group I and Group II versions of similar images. Luxo Jr. II has two more texture maps than the earlier version, including a reflection map. On average, the additional textures increase the amount of texturing work per micropolygon. Bike shop window II uses fewer texture maps than before, because a change in the camera angle removes some textures from the field of view. However, more of the textured brick wall is visible, and the aggregate screen area covered by textured surfaces is about the same. Although the texture map count is lower in the Group II version, the number of texturing operations is probably similar.

3.4.5. Profiling Results

For each image, I profiled the execution of the rendering system to learn the proportion of time consumed by the major categories of rendering tasks:

- visibility determination
- shading and texturing
- geometry
- rendering-time modeling
- input and display

Tables 3.8 and 3.9 show the timing results, and Figure 3.5 presents the information graphically. Reyes and prman run on different hardware and under different version of the Unix operating system, so the profiling tools are not the same. Because of profiling and implementation differences, categories vary slightly for Groups I and II. The differences are explained below.

I measured Reyes on the CCI Power 6/32 under Berkeley Unix using *gprof*, which reports the results of program counter sampling. A gprof report for a procedure includes not only the time spent executing its own code, but also the time spent in sub-procedures on its behalf. If a subroutine is called from several locations, gprof estimates a proportion of the execution time to attribute to each caller. From gprof's hierarchical report, we can read the percentage of runtime charged to a given phase of rendering. Some of the outer loop overhead, initialization, and data structure management fall outside of the defined tasks. I include these costs and some instrumentation overhead in the ''Other'' category.

I measured prman on a Silicon Graphics 4D/220 with a 25 MHz R3000/3010 and 16 MB of memory. The execution profiling support for the MIPS compiler under Unix System V counts basic blocks rather than sampling the program counter. It provides only a flat listing of the user cpu time charged to each procedure. To calculate the percentage of time devoted to a specific rendering task, I grouped the procedures by their source files. This categorizes most of the code accurately. However,

Image	Total	Visibility	Shading & Texture	Geometry	Fractal & CSG	Input & Display	Other
	hh:mm:ss	%	%	%	%	%	%
andre	50:07	64.6	15.2	8.2	5.1	2.7	4.2
stained glass knight	2:19:46	45.5	46.1	3.2	0.0	3.4	1.8
waves	4:51:06	32.5	50.1	8.1	3.4	2.0	3.9
luxo jr. I	1:20:01	51.2	36.6	5.4	0.0	1.4	5.4
bike shop window	5:59:55	40.8	44.6	6.8	< 0.01	2.7	5.1
average	3:04:11	46.9	38.5	6.3	1.7	2.4	4.1

Table 3.8. User CPU Time, Group I. The table lists the total user cpu time to compute each image on the CCI Power 6/32, and the proportion of time devoted to major rendering functions. **Geometry** is the time to adaptively subdivide primitives other than fractals and to create micropolygons. **Fractal & CSG** is the time to initialize, subdivide, and bound fractals or to compute constructive solid geometry operations. The column labeled **Other** covers costs such as initialization, some of the loop overhead and data structure management, and any easily separable instrumentation.

Image	Total	Visibility	Shading & Texture	Geometry	Input & Display	Other
	hh:mm:ss	%	%	%	%	%
luxo jr. II	29:44	45.8	45.7	1.2	2.0	5.3
bike shop interior	39:17	47.7	41.1	3.0	1.1	7.1
bike shop window II	28:05	54.9	34.7	3.3	1.4	5.7
conga	1:35:20	29.8	61.6	1.0	0.4	7.2
average	48:07	44.6	45.8	2.1	1.2	6.3

Table 3.9. User CPU Time, Group II. The table lists the total user cpu time to compute each image on and the proportion of time devoted to major rendering functions. The images were rendered on a Silicon Graphics 4D/220 with a 25 MHz R3000/3010 and 16 MB of memory. **Geometry** is the time to adaptively subdivide primitives other than fractals and to create micropolygons. The column labeled **Other** includes all time spent in routines from the math library, the matrix library, and other libraries called from multiple locations. It also includes some initialization and loop overhead.

many library routines, including the math library and matrix manipulation procedures, are called from several locations, so there is not enough information to show precisely how much execution time to attribute to each rendering task. The time spent in these shared procedures appears in the "Other" category.

All of the runtime measurements include only the time to render the image directly. They exclude any pre-processing needed to create texture maps, such as shadow maps. (The performance of the shadow algorithm is discussed by Reeves, Salesin, and Cook [Reev87].)

The measurements confirm that two tasks dominate rendering costs: visibility determination and shading and texturing. For Group I, the two tasks combined account for 79.8 to 91.6 percent of the execution time. For Group II, they use from 88.8 to 91.5 percent of the execution time. The spread is much wider for Group I, because two of its images require runtime modeling and increased geometric computation. Once scene uses stochastic procedural modeling with fractals, and the other uses constructive solid geometry (CSG). The balance between the two tasks, visibility and shading, shifts from scene

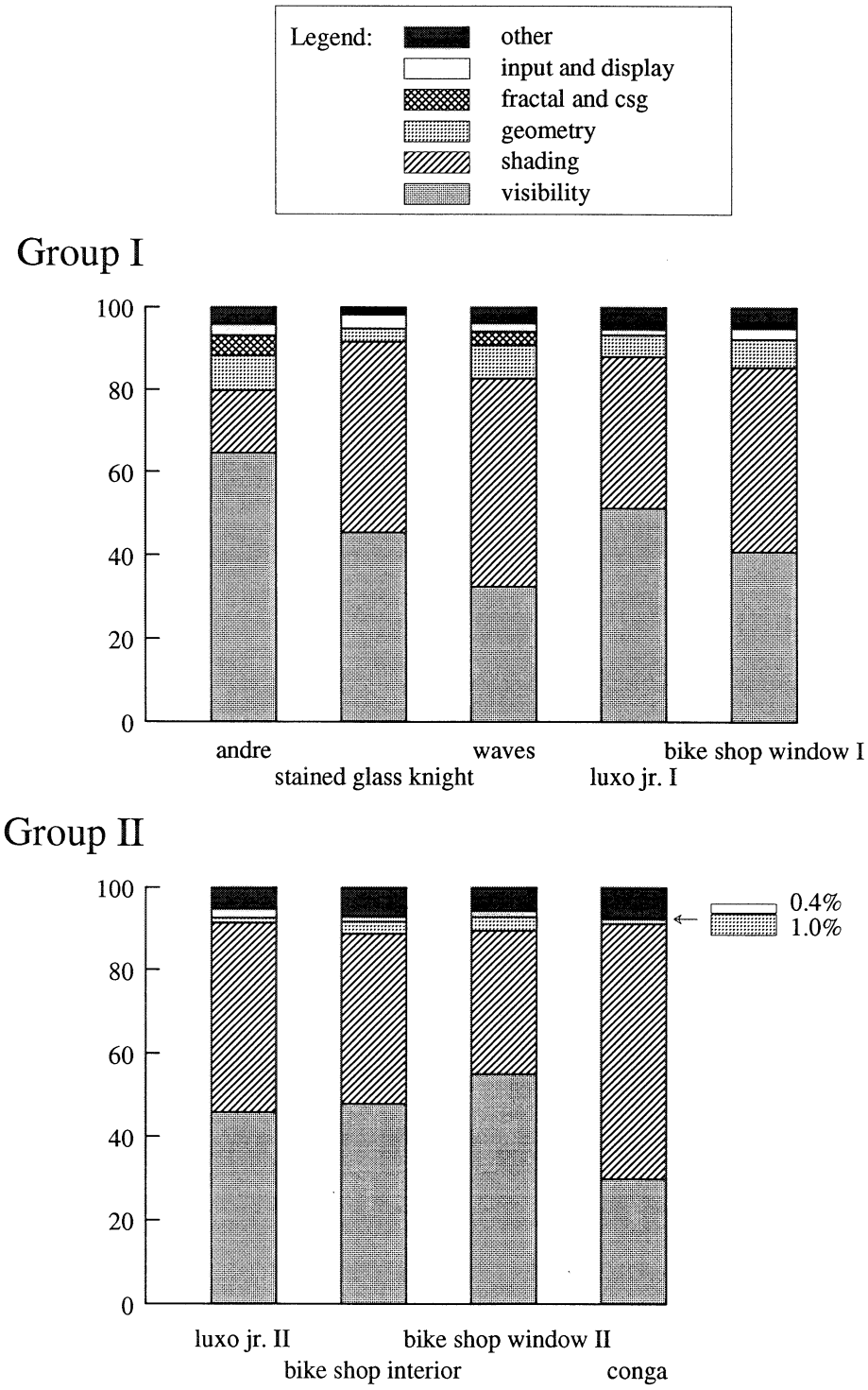


Figure 3.5. Percentage of runtime charged to major rendering activities. The graph uses the same categories as Tables 3.8 and 3.9.

to scene. The visible surface problem dominates three times and the shading and texturing problem dominates twice. In the other four cases, the costs are fairly even.

Because Reyes supports an extensible shading language instead of a constant local illumination model, shading costs vary among the images. The average time to shade a micropolygon varies by a factor of 6.6 for Group I and by a factor of 3.1 for Group II. The cost of the visible surface problem is less variable, because the algorithm does not vary.[7] Excluding CSG operations, the average time per micropolygon varies by less than a factor of two for each of the two groups. The time to process a micropolygon is affected by its screen size, the screen size of its bounding box (which determines the number of samples), and the presence of transparent surfaces.

The ''Geometry'' category describes the preprocessing that subdivides modeling primitives into grids of micropolygons. This is the third most expensive task for Group I, although it consumes no more than 8.2% of the execution time. For Group II, we cannot include the cost of calls to the mathematics library; only one to three percent of the execution time is charged directly to the geometric preprocessing. Other geometric operations are necessary for shading, texturing, visibility determination, and modeling. The modeling costs for fractal generation and csg are low.

The Input and Display category includes computation as well as input/output operations. The input module parses the model, while display requires initialization and encoding. At most 3.4% of the execution time is spent in these tasks. The input category includes only the operations that read the model data. Texture mapping requires additional input. The ratio of texture reads to model reads varies greatly, depending on the length of the model files, the amount of texturing, and the locality of the texture accesses. For example, the texture mapping module made thirty-five percent of the read system calls for the stained glass knight and about ninety percent for Luxo Jr. I.

3.4.6. Visible Surface Measurements

The measurements in the previous section show a substantial speedup from the 1985 technology of Group to the 1990 technology of Group II. The speedup comes not only from faster hardware, but also from the new software, such as the improved visible surface algorithm.

A more detailed analysis of visible surface measurements reveals further characteristics of the algorithms and the workload. Table 3.10 shows the execution time and input size for each of the three stages of the visible surface algorithm. This data is obtained from the hierarchical profiles and additional instrumentation, which is available only for Group I. The time for each stage is given both as a proportion of the total user time and in seconds. The input size is the number of samples processed in each stage, in thousands.

The workload is not balanced among the three stages. Without exception, each stage is smaller and faster than the previous stage. Table 3.11 shows the reduction in input size more clearly, by expressing the size of each stage as a fraction of the initial input to the algorithm. The reduction in execution time from one stage to the next is only partially explained by the smaller input size. As Table 3.12 shows, the computation is less complex in each successive stage. For a given stage, the average cost per sample is remarkably consistent across all five images.

Many parallel rendering algorithms subdivide the screen and assign one or more regions to each processing node. The variability of the input size at different locations affects the system's ability to balance the load among the processors. Table 3.13 shows the distribution of input among the sample points, and Figure 3.6 graphs the mean input size per sample point. The variability is expressed by a relative measure, the coefficient of variation, which expresses the standard deviation as a fraction of the mean.

---

[7] Reyes allows a different visible surface module to be substituted for the default module. As Section 3.4.3 notes, this mechanism was used to instrument the visible surface algorithm. It is also possible to vary the algorithm, but, in practice, the default stochastic supersampling algorithm is always used.

Image	sample			sort			filter		
	%	sec	size	%	sec	size	%	sec	size
andre	49.7	1,495	16,397	14.4	435	4,789	3.0	92	2,543
stained glass knight	31.1	2,611	31,924	7.7	648	10,815	4.1	342	9,010
waves	23.9	4,174	49,074	3.1	541	9,144	1.3	227	7,917
luxo jr. I	36.9	1,772	18,704	7.0	336	5,474	4.4	211	5,024
bike shop window	30.4	6,565	69,199	5.4	1,158	21,682	2.4	521	16,092
average	34.4	3,323	37,059	7.5	624	10,381	3.0	279	8,117

Table 3.10. Visible surface time. For each stage of the visible surface algorithm the table shows the percent of execution time, seconds of cpu time, and the size of its input expressed in thousands of samples.

Image	size of input		
	sample	sort	filter
andre	1.00	0.29	0.16
stained glass knight	1.00	0.34	0.28
waves	1.00	0.19	0.16
luxo jr. I	1.00	0.29	0.27
bike shop window	1.00	0.31	0.23
average	1.00	0.28	0.21

Table 3.11. Reduction of size in the visible surface algorithm. The size of the input to each of the three stages is expressed as a fraction of the initial input to the first stage. The average is the geometric mean.

Image	ms/1000 samples		
	sample	sort	filter
andre	91.18	58.56[8]	36.20
stained glass knight	81.77	59.90	37.95
waves	85.06	59.21	28.68
luxo jr. I	94.72	61.40	42.04
bike shop window	94.87	53.41	32.35
average	89.52	58.50	35.44

Table 3.12. Reduction of cost in the visible surface algorithm.

[8] For Andre, the depth sort statistic excludes csg operations. Including csg, the average time to process one thousand samples is 90.83ms.

Image	sample		sort		filter	
	mean	CV	mean	CV	mean	CV
andre	9.77	1.05	2.94	0.88	1.64	0.29
stained glass knight	4.52	0.88	1.53	0.76	1.32	0.35
waves	9.32	1.16	1.74	0.75	1.52	0.33
luxo jr I	4.73	0.80	1.38	0.58	1.35	0.35
bike shop window	7.84	1.63	2.46	0.64	1.83	0.34
average	7.24	1.10	2.01	0.72	1.53	0.33

Table 3.13. Input size per sample point. The entries describe each of the three stages of visible surface processing for Group I. The mean input size is the average number of micropolygons processed, per sample point, during each stage. CV is the coefficient of variation, or the ratio of the standard deviation to the mean. The table excludes sample points in empty background pixels.

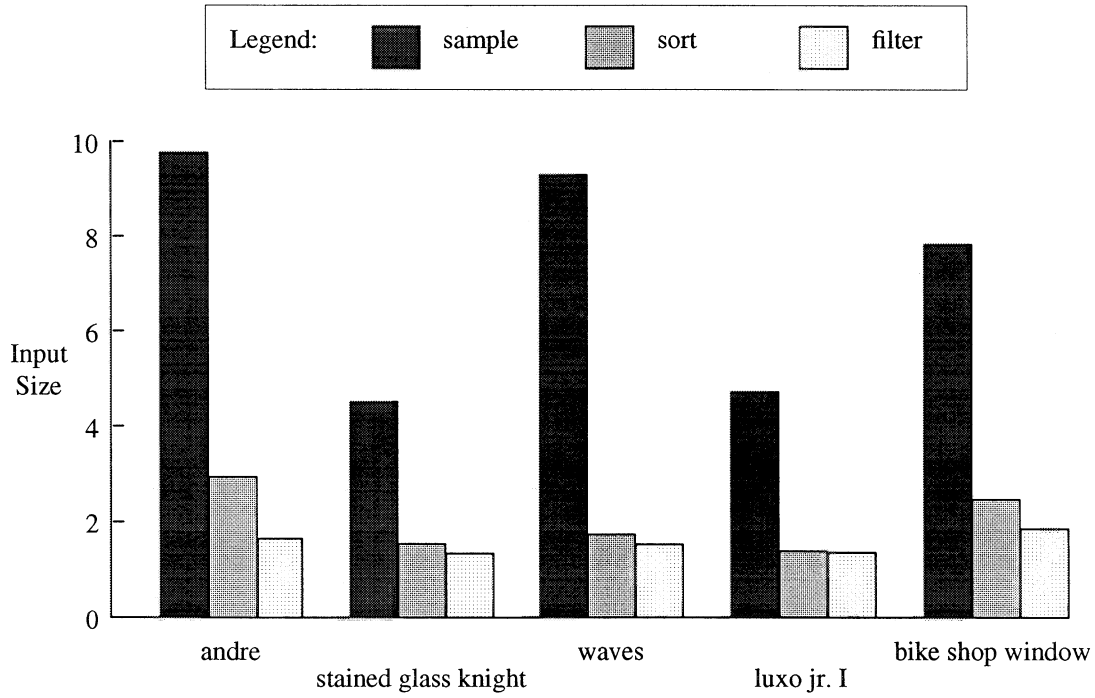


Figure 3.6. Mean input size per sample point. The data describe each of the three stages of visible surface processing for Group I. This graph plots the data in Table 3.13.

There is considerable variation in input size to the sampling stage, which consumes most of the execution time. With each stage, the input becomes not only smaller, but also less variable. This analysis considers only the variability among sample points. Chapter 5 addresses the workload variability among larger regions.

The approach called invisibility processing attempts to reduce the number of sampling operations. Prman reports the number of discarded objects, and Table 3.14 summarizes these statistics for Group II. The success of these optimizations depends heavily on the depth complexity, and it can also be influenced by the relative size of the primitives. The bike shop window has the highest mean depth complexity of the Group I images, and prman is able to eliminate nearly 70% of the grids from further processing. The opaque wall in the foreground of the scene makes it easy for the renderer to discard large portions of many surfaces. By discarding grids, the systems not only reduces the size of the visible surface problem but it also avoids shading invisible surfaces. The optimizations are also effective with the interior view, although the system discards more of the smaller objects and fewer whole grids. The algorithm is less successful with the other images, which are flatter.

The algorithm's designer experimented with invisibility processing on a variety of scenes. For images with very low depth complexity, he found that the required tests can consume more time than they save [Apod91]. In general, invisibility processing pays off for the images with higher depth complexity. For the typical image in Pixar's current workload, the algorithm pays off. It reduces the problems of shading before determining visibility while preserving the advantage of shading large pieces of a coherent surface at one time.

We can compare the performance of prman and Reyes for two pairs of scenes. Prman renders the bike shop window twelve times faster than Reyes. Because prman discards most of the grids before shading, it spends a much smaller proportion of time in shading and texturing. The average number of hits per sample point is reduced from 2.48 to 1.57. For Luxo Jr., the average number of hits per sample point shows a smaller decrease, from 1.47 to 1.03. The proportion of time spent shading and texturing Luxo Jr. is actually greater under prman. Two factors explain this increase. First, the Group II version has a more complicated shading model that includes reflection mapping. Second, the low depth complexity limits the benefits of invisibility processing.

		luxo jr. II	**bike shop interior**	**bike shop window II**	**conga**	**average**
Grids	shaded	8,437	53,841	25,720	15,418	25,854
	discarded	733	41,186	56,285	2,518	25,181
	% discarded	8.0%	43.3%	68.6%	14.0%	33.5%
Micropolygons	sampled	717,707	1,676,167	1,192,292	2,340,643	1,481,702
	discarded	26,493	221,865	75,871	136,385	115,154
	% discarded	3.6%	13.2%	6.4%	5.8%	7.3%
Samples	tested	17,741,221	37,043,356	29,361,513	51,511,258	33,914,337
	discarded	399,290	6,001,405	2,107,101	1,921,371	2.607,292
	% discarded	2.2%	13.9%	6.7%	3.7%	6.6%

Table 3.14. Invisibility processing, Group II. The table shows the number of items discarded after each of the three tests that identify invisible objects.

3.5. Summary

The images in the test suite vary in geometric complexity, depth complexity, shading models, lighting, and texturing requirements. On a whole, however, they are more alike than different, reflecting a uniform style of modeling and rendering. An examination of the historical data in Section 3.2, finds growth in geometric complexity, but also clusters that represent periods when the complexity tends to range within similar bounds. For example, when we extract the data for the years 1984 to 1991 (Figure 3.4), the growth is not obvious. Improvements in technology often push complexity to a new plateau, where it may settle until the next large change. In the Pixar workload, there has been a mild trend towards more complex scenes over the years. On the other hand, Pixar's workload has been shifting more from research and development towards commercial production. Tight production schedules usually allow less time for model development, so the complexity of Pixar's scenes may now level off.

The test images lack true global illumination effects, but in other respects the workload is still complex compared with other examples in the literature. The geometric intricacy, local illumination, texturing, and rendering quality are very advanced and the animation tools support dynamic changes in geometry.

The measurements presented in this chapter document the variability in the workload. For the visible surface algorithm, the workload is distributed unevenly over the sample points. The variability is the greatest in the initial, most costly stage of the processing. However, the visibility costs per unit of input are consistent for all the images. The costs of shading and texturing are even more unbalanced, because the complexity of the shading model and the number of texture operations varies from surface to surface and from image to image.

Reyes attempts to improve locality and make shading more efficient by shading large pieces of a surface at one time. Its approach involves shading all surfaces before determining which are visible. One concern with this algorithm is that the renderer will spend a lot of time unnecessarily shading invisible objects. The measurements in this chapter point out two mitigating factors. First, there may be fewer invisible surfaces than expected. The depth complexity indicates how much extra shading is required, assuming that most surfaces are opaque. In general, the depth complexity of the images in the test suite is seldom high. Second, improvements to the algorithm discard many of the invisible surfaces prior to shading. Chapter 4 describes the results of experiments that explore these issues further.

4

A Tool for Measuring the Performance of Rendering Systems

4.1. Introduction

To measure the performance of an image synthesis system, we observe the resources it requires to generate an image. The image is specified by a database with a detailed model of the scene, controls for viewing the scene, and controls for displaying the image. Together, the model and controls form a test case, or benchmark. Meaningful performance measurements depend on good test cases.

It can be hard to obtain a varied and appropriate set of benchmarks to test a system. Typically, individual sites have developed their own suites of test images. The number and scope of the tests is often limited, because it is time-consuming to create the models for complex new images. Researchers frequently create images that illustrate the functionality of a new algorithm without simulating any actual or anticipated workload. Scene specifications are often proprietary and are seldom portable, since no standard for describing complex, realistically-shaded, three-dimensional images is widely supported. A portable test suite and scene description language would provide a source of additional test cases, support performance comparisons, and facilitate the sharing of scene data.

This chapter describes the design of a portable model generator which creates test cases useful for studying the performance of rendering systems. The *m*odel *g*enerator, or *Mg*, is a group of programs, written in C. Under the control of a set of parameters, they generate scene specifications, including model geometry, surface properties, textures, light sources, and viewing parameters necessary to define an image. To port Mg to a new system, a programmer modifies a small set of procedures that output the model data. Mg differs from a static benchmark suite, because the experimenter can adjust the complexity and characteristics of the scenes to simulate different workloads or to evaluate specific aspects of a system's performance. Two test cases with different characteristics have been implemented, and the tool's format allows new scenes to be added as image synthesis workloads and technology evolve.

The primary goal of this project is to demonstrate a methodology and a tool for reproducible, controlled performance experiments. A reproducible experiment yields essentially the same results when repeated by different people or when executed on different, but equivalent, systems. A controlled experiment studies the effect of a specific parameter by varying that parameter while holding other factors constant. To study an image synthesis system, an experimenter may need to vary some geometric characteristics of the model, such as the number of texture maps, the type of primitives, or the depth complexity. In this case, a single set of static models would not be useful. Mg promotes a different approach: model data are generated to the specifications of the experimenter. The same model is always generated for a given set of specifications, so that experiments are reproducible. By providing a framework for generating parameterized models and reporting results, Mg supports performance experiments that are both reproducible and controlled.

By generating geometric data on the fly, Mg also meets design goals concerning portability. First, the programs that generate models are more compact and easier to distribute than raw data files. Second, it can be easier and more efficient to modify a clearly-defined set of output routines than to translate large data files when porting benchmarks to a new system.

No benchmark suite can be appropriate for all graphics systems or applications. The types of operations and the contents of the scenes together define a subset of graphics systems for which a measurement tool is suited. Mg targets systems for realistic image synthesis and omits consideration of issues raised by real time graphics and user interaction. Its geometric interface supports parametric patches, quadric surfaces, polygons, and nested transformation matrices. Its lighting and shading procedures support multiple distant and local light sources, textures, reflective surfaces, and transparency with refraction.

Having defined an interface, it is still hard to design scenes that are characteristic of a range of applications. Rendering workloads vary widely in the complexity of scenes, their specific characteristics, and the balance between application computing and rendering. For this reason, many people in graphics favor tools that help users create portable benchmarks from their own workloads [Dunw90, Tice88] over standardized test scenes. Mg's aim is different, because it generates scenes with controlled variation and permits the experimenter to evaluate the effect of changing specific workload characteristics.

In a performance experiment, we can separate the quantitative properties from the qualitative. Mg is designed to help measure how fast a system renders images, not how well it renders. It is not a validation suite, and it is not intended to assess either rendering accuracy or functional completeness. Although Mg does not evaluate qualitative properties, they can be incorporated into the experimental design. For example, we can compare the time to render a given image with different antialiasing options, separating any subjective evaluation of rendering quality from the performance analysis.

Benchmarks are commonly used to compare the performance of competing systems. Although inter-system comparison is not a major goal for Mg, a well-defined set of test cases serves as a benchmark suite. Even when systems have similar capabilities, comparisons raise many tough issues about the rendering system interface, portability, and fairness. One problem is to find the right balance between completeness and generality in the selection of geometric and shading operations. The operations should be widely available, but they should not be so limited that they omit important workload characteristics. A second problem is to design an interface that is fair to different systems and implementations. For example, the format that represents geometric data most efficiently for one system may be inefficient for another. Some biases may be reduced by allowing the data formats to vary, but others are harder to eliminate. Workload characteristics can also introduce biases, because systems that are tuned for specific workloads often perform poorly on other workloads. Finally, qualitative differences among systems can affect performance as well as portability. Systems that support complexity and realism typically have higher overhead than less powerful renderers and take longer to render even simple scenes. Flexible systems that offer a range of algorithms and approaches are typically slower than systems that are optimized for specific algorithms.

Some of the problems with inter-system comparisons must be addressed in the design of benchmarks, while others can be handled with clear documentation. It is important to carefully define the category of systems that are evaluated and report test conditions. In practice, there is seldom a strong need to compare systems with vastly different capabilities. However, even if we ignore inter-system comparisons, we must address problems with bias and portability if a measurement tool is to be useful for studying many systems.

The following section surveys issues and related work in graphics benchmarking, and Section 4.3 presents a framework for controlled rendering performance experiments. Sections 4.4 to 4.7 describe Mg's organization, its rendering system interface, the sample test cases, and guidelines for reporting results. The remaining sections document the implementation of Mg on two different rendering systems and demonstrate its use.

4.2. A Survey of Benchmarking Tools for Graphics Systems

Four major issues in the design of performance measurement tools for graphics are

- the overall structure of the tools,
- the interface used to transmit the scene data to the graphics system,
- the primitives and operations supported by the interface, and
- the characteristics of the scenes used as test cases.

The first three issues are important to almost all graphics performance tools. However, not all tools include test scenes.

1. Structure. For most applications, benchmarks are programs that are run with standardized sets of input data, and the elapsed time to run the benchmarks is the primary metric. Usually, benchmarks are executable: sometimes real applications programs, either small or large, sometimes representative portions of code, and sometimes synthetic programs that are constructed to simulate a workload. To study the performance of a graphics system, one usually submits *input*, and not executable programs to the system. In this respect, graphics test images resemble the Wisconsin database benchmarks [Bora84], composed of artificial databases and carefully-selected queries. The DARPA Image Understanding Benchmark for parallel computers contains elements of both data and programs [Weem91]. It consists of a set of inputs and a specified method of solving a problem; although a reference implementation is given, the solution must generally be implemented afresh for each target architecture.

The relationship between a performance tool and the test data is a major factor in the tool's overall structure. Standard rendering benchmarks emphasize test data, which may be contained in a static database or generated on the fly. Benchmarks suites may or may not specify measurement procedures, and they may offer little support for running or timing the tests. Another type of tool helps users capture their own data for test cases. Such tools typically define a portable scene definition language and provide software to translate that language for various target systems. They sometimes include procedures for running and timing tests. The scene data can be created by editing data files or by tracing graphics commands while executing an application program. A third class of tools studies low-level operations outside the context of a picture. These tools typically target interactive applications and emphasize measurement procedures and timing methodology. Information about low-level operations is not generally useful for image synthesis systems, because the cost of an operation is very sensitive to interactions with other elements in the scene.

2. Interface. The data for portable benchmarks are generally expressed in some intermediate *interface language*. Decisions in the design of the interface can give some systems performance advantages or deny systems the information they need to use optimal data formats or commands. Some of the low level issues in interface design are the amount of state a translator must maintain, the coordinate systems used to define primitives, the representation of vertex information, and support for data hierarchy. For example, consider a flat, stateless interface for polygonal data. Each polygon definition includes a complete description for all of its vertices. Because this interface does not show when a vertex is shared by adjacent polygons, it rules out more efficient, list-based representations.

3. Primitives and operations. High level issues in interface design include the types of primitives and operations that are supported. These design decisions bias the interface towards certain types of applications, workloads, and rendering systems. They also influence the complexity of images that can be represented. Generality trades off against complexity in the interface design. By demanding certain levels of sophistication, the performance tool may rule out some systems. On the other hand, by limiting the interface, it may fail to represent the complexity of many applications.

4. Scene characteristics. The scenes used as test cases determine the specific workload characteristics, which strongly affect the relative performance of different rendering algorithms. Because image synthesis workloads vary greatly and there has been little research on workload characterization, it is hard

to argue that any set of benchmarks adequately represents the workload of more than one application or installation. One solution to this problem is to design tools that allow users to capture their own applications for benchmarks. Mg proposes another solution, to vary selected workload characteristics under parametric control.

The rest of this section surveys previous work in graphics performance methodology and benchmarking and discusses the approach other researchers have taken to the design issues introduced above.

4.2.1. Methodology for Rendering Test Cases

Schoeler and Fournier [Scho86] proposed a methodology for constructing test cases in their analysis of different rendering systems. Most test cases in the literature are static, that is they cannot easily be varied by changing parameter values. Schoeler and Fournier took a different approach, advocating a controlled manner of changing the image complexity, so that one well-defined type of variable varies while others remain fixed. The simple geometry of their two scenes allowed them to port the models easily to different rendering systems, but there is a need for more varied and complex scenes and for test cases that are more comprehensive in their treatment of shading, visibility, and special effects. Schoeler and Fournier's methodology influenced the parameterized controls in Mg's design.

4.2.2. Standard Rendering Benchmarks

Among the graphics community, there has been a growing interest in more serious performance analysis and a desire to compare the performance of different algorithms or systems. Some of the first steps in this direction were taken by researchers who shared their model data or attempted to reproduce the images published by others. Appendix A documents some duplicated images as early as 1971. More recently, Glassner introduced a recursive tetrahedron test case [Glas84] that others have reproduced [Arvo87, Kay86].

In 1987, Haines released the Standard Procedural Databases (SPD) [Hain87], the most prominent example of image synthesis benchmarks. The SPD is a set of programs that generate the descriptions for six different images in a simple, easily-translated language. One of the six scenes is the tetrahedron described above. The package is strongly oriented towards two types of rendering systems: ray tracers and polygon-based graphics hardware. The programs are accompanied by detailed specifications for timing tests and statistics describing the images.

The six scenes are described in Table 4.1: a recursively-defined collection of reflective balls, a two-dimensional array of meshed gears, glass spheres in front of a fractal mountain, layers of intertwined rings, the recursive tetrahedron, and a bare tree. The scenes contain specular reflection, transparency with refraction, and shadows. They use ambient light and between one and seven positional light sources.

The models contain four types of primitives: polygons, spheres, cones, and cylinders. The simplest primitives, polygons and spheres, predominate. Haines designed the scenes to contain roughly ten thousand modeling primitives. All of the databases can be generated in a polygonal form by subdividing quadric surfaces into polygonal patches, which retain the surface normal of the underlying primitive at each vertex. Half of the scenes are modeled directly with polygons or using mostly polygons; they have about the same number of primitives in both the original and the polygonal formats. The remaining scenes, which have many quadric surfaces, generate around a million polygonal patches.

In four scenes (balls, gears, rings, and tree), a single large polygon models a floor or background. Objects cast shadows on these surfaces, and one floor is reflective. The screen coverage, as shown in Table 4.1, is greater for the SPD images than for the Reyes workload, largely because the SPD uses background planes while the Reyes scenes assume blank backgrounds. The SPD background polygons tend to cover a large portion of the screen, but the scene complexity is often clustered in a smaller region. For example, the balls image covers the entire screen when computed with its background polygon. Without the background, the objects cover only about a third of the screen. Table 4.1 also shows the sample point

Scene	balls	gears	mountain	rings	tetra	tree
Modeling primitives						
cones						4,095
cylinders				4,200		
polygons	1	9,345	8,192	1	4,096	1
spheres	7,381		4	4,200		4,095
Total	7,382	9,345	8,196	8,401	4,096	8,191
Polygon version						
#primitives	1,417,153	9,345	8,960	873,601	4,096	851,761
%Coverage	100	93	65	99	19	64
Depth Complexity						
covered sample points	1.7	3.1	1.6	4.9	1.8	1.1
all sample points	1.7	2.9	1.1	4.9	0.3	0.7
#Lights	3	5	1	3	1	7
Specular reflection	yes	yes	yes	yes	no	no
Transparency	no	yes	yes	no	no	no

Table 4.1. Characteristics of Haines' Standard Procedural Databases.

depth complexity, computed by rendering the scenes with the instrumented version of Reyes described in Chapter 3. We can compare statistics for the SPD with the corresponding entries for the Reyes Group I images in Table 3.4. The SPD scenes also tend to have greater depth complexity. This difference is due in part to the background polygons and in part to the inclusion of two images which display many objects stacked together.

Standard versions of each scene make up a benchmark suite. The package also offers some coarse control over the database complexity for researchers who want to generate their own test cases. Most of the scenes are defined recursively: balls, mountain, tetra, and tree. Increasing the complexity of these scenes increases the number of primitives, adds detail, and lowers the average screen size of the primitives. However, the depth complexity and screen coverage remain the same or increase only slightly. Increasing the complexity of the other two scenes, gears and rings, adds a layer of objects to a three-dimensional array. Thus, the depth complexity increases, but the screen coverage and the variability of geometric characteristics changes little.

The interface language consists of eight commands. Four of the commands define the modeling primitives. A fifth command gives the surface characteristics, or material properties, for a surface. Another specifies a positional light source, and the remaining commands specify viewing and display parameters or the background color. All objects are described in a flat world coordinate system, because the interface does not support any type of geometric transformations or object hierarchy. With this simple interface, a translator need not maintain any state while converting a scene database to the target system's format.

The SPD emphasizes two types of renderers: ray tracers and polygon-based systems with hardware assists. The testing guidelines and statistics distributed with the code demonstrate these priorities. The ray tracing orientation is further shown by the choice of primitives and the support for specular reflection, refraction, and shadows. For other types of renderers, the surface descriptions are well-suited to Phong specular highlights and Gouraud interpolation; these models are commonly available on graphics workstations. The scenes and the interface of the SPD ignore features that are important to other types of image

synthesis systems, such as parametric patches, a wider range of surface and lighting models, and texture mapping.

4.2.3. Workstation Benchmarks and Measurement Tools

Most of the work in graphics performance has been motivated by the needs of marketing and procurement, and has consequently been directed towards evaluating interactive workstations. An early example is Linton's workstation benchmarks, which assess processor, file access, multitasking, and graphics performance [Lint86]. These benchmarks are structured as a set of programs that are run on the target system. The graphics tests cover a limited range of low-level, two-dimensional graphics and text operations.

The now inactive Technical Interest Group in Performance Evaluation (TIGPE) of the Bay Area ACM/SIGGRAPH chapter discussed methodology and data for benchmarking workstation graphics. One of TIGPE's contributions was to define four levels of graphics performance characterization: primitive operations, pictures, systems, and applications. The group concentrated on the first level, characterizing workstation performance on primitive operations for drawing vectors, polygons, and characters. The second level measures the time to generate a complete image, the third level adds interactive input and display, and the fourth level tests the performance of entire applications. TIGPE produced prototypes for some measurement tools. These tools emphasized timing methodology and controlled experiments that varied a single, well-defined property of the data. For example, a test of vector drawing time would analyze the effect of varying the length or orientation of the vectors. TIGPE's analysis of the problem was well-constructed and comprehensive, but inappropriate for measuring the cost of realistic shading or visibility in a complex, three-dimensional environment.

The Graphics Performance Characterization (GPC) group is a consortium of workstation vendors. Influenced by early contact with TIGPE, the GPC chose to focus on the second level of graphics measurement, the ''picture'' level. The main goal of the GPC is to help customers create portable benchmarks from their own picture data. They defined a benchmark interchange format for three-dimensional scenes [Tice88] and contracted a reference implementation. The interface is oriented towards the classes of applications that are currently supported on commercially-available workstations. Each participating vendor ports the interface to its own system. Customers who create data files in the interchange format should be able to run their benchmarks on any of these systems. A small set of reference picture files has attracted some interest as a benchmark suite. These files are taken from applications such as circuit board design and mechanical CAD.

Another approach to evaluating workstation performance is to use a complete application as a benchmark. If the benchmark is truly representative of the anticipated workload, this approach is accurate and it allows users to assess interactive responsiveness. However, it requires much more effort to implement a truly portable, unbiased application benchmark. Zyda, Fichten, and Jennings have used military visual simulators to measure the performance of graphics workstations [Zyda90]. They compare the vendor's claimed drawing rates with the rates observed in the context of their application.

4.2.4. Summary

The graphics performance community tends to emphasize tools rather than standard benchmarks. This bias acknowledges the wide variety of workloads and the inadequacy of ''single figure of merit'' benchmarks, which provide little information beyond the observed runtime. The only well-known suite of standard test cases is Haines' SPD, which grew out of an implementer's effort to develop test cases for himself and to share them with others.

There has been little work on performance measurement for realistic image synthesis systems. An evaluation of a realistic renderer often concentrates on quality, the interface to modeling systems, and support for desired features. Customers often accept performance that is ''fast enough,'' given acceptable rendering quality and modeling support. For this reason, a performance tool that can be used to design

and improve a single system is very attractive.

4.3. The Structure of Graphics Performance Experiments

Most image synthesis systems contain several separate components, such as a host computer, a graphics processor, rendering software, and a frame buffer. Let us define an abstract *graphics computer* as the configuration of hardware and software elements that processes the scene specifications to produce a raster image in the frame buffer or a file. The graphics computer encompasses all factors in the environment that affect rendering performance, such as whether or not a rendering package is embedded within a window system. A performance experiment measures the resources that the graphics computer requires to generate a specified image. A reproducible experiment must specify the graphics computer completely. This abstract view of the rendering process provides a unified framework for studying diverse hardware and software architectures.

We construct a rendering performance experiment by varying a parameter that affects rendering and observing the effect on the system's performance. Chapter 2 introduced four categories of parameters that affect the contents of an image and the resources required to render the image. The *computing environment* describes the rendering system as embodied in the graphics computer. *Rendering parameters* control the way in which the rendering system processes or displays the image. *Viewing specifications* describe the way that the simulated viewer or camera observes the scene. Finally, the *scene characteristics* specify the inherent properties of the modeled scene, including its geometry, illumination, and surfaces. Each category is discussed below in more detail.

4.3.1. Computing Environment

The abstract graphics computer defines the computing environment. Environment parameters include the type of host computer, the availability of special hardware assists, and the amount of memory. Changes in the graphics computer affect the resources required to compute an image, but do not, in general, affect the resulting picture. Occasionally, changes in the environment do have subtle effects on the image contents. For example, the use of integer arithmetic instead of floating point can alter the results of geometric or shading computations.

A common type of performance experiment is to change part of the environment while keeping the image specifications constant. Indeed, this is the basis for inter-machine comparisons. For example, the same image synthesis software might render the same image on different host computers. The control of environment parameters is independent of Mg's operation, in that none of its input or output changes when the computing environment changes. A full report of the environmental conditions is necessary to give a context for understanding the results of measurement experiments.

4.3.2. Rendering Parameters

Rendering parameters control the execution of the graphics algorithm on the graphics computer. Unlike the computing environment, rendering parameters frequently affect the image's appearance. But like the computing environment, rendering parameters do not, in general, affect the scene description. One class of rendering parameters includes the rendering system's runtime parameters. For instance, an image might be rendered at different resolutions. Or, it might be rendered with and without a special effect, such as depth-of-field. Another class of rendering parameters describes the internal operation of the rendering system, its algorithms, and its implementation. For example, Crow considered changes to rendering parameters in his cost-effectiveness study of antialiasing techniques [Crow81]. Some important rendering parameters deserve further discussion.

The display resolution is a special case of a rendering parameter, because it can affect the scene's geometry in addition to the number of pixels in the image. Procedural modeling algorithms often consider the screen size of an object to determine how much detail to generate. Mg knows the resolution, and passes the information to the rendering system. Currently, none of the scenes are affected by the

display resolution, but future scenes may be resolution-dependent.

Antialiasing is often controlled by runtime options. They may enable and disable antialiasing or select an antialiasing algorithm. Options may also specify the degree of antialiasing, the degree of supersampling, or the type of filter. Antialiasing parameters do not affect the model, but the cost or effectiveness of an antialiasing algorithm depends on the characteristics of the scene.

Not all rendering systems support properties such as reflection, refraction, shadows, and depth-of-field. Some systems provide these effects as options, and others approximate them with varying degrees of realism. Rendering parameters enable and disable visual effects, select options, or adjust approximations. Given a fixed set of values for the rendering parameters, the actual cost of computing an effect depends on the scene characteristics. A rendering system can ignore information if the corresponding effects are unavailable or unwanted, but the experimenter should report the omission.

4.3.3. Viewing Specifications

Viewing specifications include the viewing position and direction, the angles of view, and a model for the eye or camera. Mg generates viewing specifications and allows them to vary. These parameters are an integral part of the image definition, but they are generally independent of the model definition. The viewing specifications determine which parts of the model are visible and which parts of the screen are covered. Thus, changes to the viewing specifications directly affect the image and the rendering costs, although they do not normally affect the model. Suppose we zoom in on a set of objects that are displayed against an empty background. Without modifying the model definition, we vary the screen size of the objects and the screen coverage.

Occasionally, changes to the viewing specifications affect the model's definition. If an object's screen size changes, some applications or systems will adjust the level of detail in the model.

4.3.4. Scene Characteristics

The scene characteristics describe the geometry, surface properties, and lighting of the scene. They define the complexity of the scene, which in turn determines the cost of computing an image. Chapter 2 lists the most important scene characteristics and discusses their potential effects on rendering costs.

There are many ways to vary a scene's geometric complexity. For example, experiments can change the number of objects, modify the structural complexity of a fixed set of objects, or vary the distribution of objects in space. Simple changes to the scene geometry can exhibit a complex range of effects. Suppose we want to add new primitives to an existing scene of some objects displayed against a blank background. We can maintain the screen coverage and depth complexity by subdividing the existing objects into smaller primitive elements. Alternatively, we could add new objects with the same average size as the existing objects. In this case, the spatial distribution of the objects determines the effect on the screen coverage and depth complexity. If all of the objects are visible, the screen coverage increases but the maximum depth remains about the same. If the new objects are all hidden, the depth complexity increases but the screen coverage remains the same.

Other experiments can vary surface properties such as the amount of transparent or reflective surfaces, the number of texture maps, or the amount of textured surfaces. Varying the number of light sources and their characteristics affects the cost of shading and shadowing.

4.4. The Structure of Mg

Mg is made from two types of source files: scene generators and library code (see Table 4.2). Each scene generator outputs the model for one family of scenes. The scene generators describe the contents of the scene in an *interface language* composed of procedure calls. The library source files implement the interface to the rendering system; they also provide definitions and general utilities. The interface procedures translate the scene description into the format required by the target rendering system. To port

Mg to a new system, a programmer modifies the interface procedures in *mg_output.c* to output the system's own scene description language. Once a scene generator is compiled with Mg's library, running the scene generator will produce a model description file.

The scene generators vary the scene characteristics under the control of a set of parameters. Some general parameters, such as the image resolution, are applied all scenes. Others are defined individually by the scene generator. The options are specified as command-line arguments. Every option has a default, so it is not an error to omit the options. To document the test conditions, each scene generator reports the values of all of its options.

Mg's interface is closely related to the RenderMan Interface [Pixa89]. RenderMan, unlike most other current or proposed graphics standards, supports all of the features for realistic image synthesis that Mg uses. The interface is not proprietary, documentation is widely available [Upst90], and implementations are available for a number of different platforms. Mg's interface is not a subset of RenderMan, but it can be easily translated into the RenderMan interface. In the interest of portability and generality, it differs from RenderMan in some respects.

This rest of this section describes the interface language. Appendix B specifies the complete interface in C syntax. Later sections describe the scenes and the options. Complete specifications for porting Mg to a rendering system are documented in the source files. Upstill provides a more detailed explanation of the RenderMan Interface [Upst90].

Type	File	Description
library	mg_defs.h	definitions
library	mg_lib.c	general utilities: matrix routines, random number generator, etc. .sp .15
library	mg_output.c	interface procedures
library	mg_texture.c	utilities to generate texture data
scene generator	mg_spheres.c	spheres model
	mg_spheres.h	coordinate data for patches and polygons
scene generator	mg_terrain.c	textured terrain

Table 4.2. Mg source files.

4.4.1. Frame Initialization

Before generating any data for an image, the scene generator calls the routine *OutputBegin* with a name for the picture as its only argument. After completing the image specifications, the scene generator calls *OutputEnd*. These calls are provided as a convenience for porting Mg. The two routines have no required function, but they can be used to output any prologue or epilogue that is useful for the target system.

A scene is initialized with viewing and lighting specifications that affect the entire scene. *OutputResolution* is called once per frame to specify the image resolution. This is the only rendering parameter that Mg specifies, because it can potentially affect the level of detail in procedural models.[1] Other rendering parameters, such as sampling and filtering specifications, do not affect the scene data generated by Mg. An experimenter can vary any of the other rendering parameters without modifying Mg's input or output.

[1] None of the images currently generated by Mg are affected by the display resolution.

OutputViewpoint is called once per frame, after *OutputResolution*, to specify the viewing position, field of view, and near and far clipping planes. The eye or simulated camera is located at the specified position and aimed at the "look at" point. Together, the viewing position and the "look at" point define a viewing direction. The camera's orientation is defined by "up," a point in the up direction from the camera's position. The field of view is specified as an angle; the RenderMan interface requires an angle less than 180 degrees. The near and far clipping planes are specified by distances from the camera; the planes are perpendicular to the z axis in camera space. Mg assumes a perspective projection for all images. The current viewing model does not support visual artifacts such as depth-of-field, which describes a range of depth over which objects appear in sharp focus.

OutputLight is called once per light source to describe its lighting model and its light source properties. The light source models are a subset of RenderMan's standard light source shaders: ambient, distant, and point light sources [Upst90]. Mg favors generality over completeness by omitting more complex models, such as area light sources. All light sources have a scalar intensity and a color with red, green, and blue components. Ambient light has no position or direction; a distant light has a direction, but no position; and a point light source has a position but no direction. The unused properties have a null value.

The interface does not enforce any maximum number of light sources in a scene. Each scene generator creates a fixed set of light sources, which are initially turned on. Options to the scene generator can control the number of light sources by turning off selected lights.

OutputViewpoint (position, look_at, up, field_of_view, near, far)
OutputResolution (xres, yres)
OutputLight (model, intensity, color, position, direction)

Table 4.3. Frame initialization routines.

4.4.2. Geometric Specifications

Most of the data for a scene are detailed geometric specifications. No matter what format is used internally by the scene generator, the geometric specifications will probably have to be translated into another format. The interface must provide enough geometric information to enable a programmer to translate its data structures into the format required by another system.

The interface supports nested transformations to simulate the characteristics of complex, hierarchical models. This design contrasts with Haines' Standard Procedural Databases, which use absolute world coordinates for all geometry. In fact, the RenderMan interface requires transformations. It defines many primitives relative to the origin and positions them with a sequences of transformations. This decision raises the issue of bias in comparing a system that can use absolute coordinates against a system that implements the RenderMan interface. If we insist on including transformations as part of the renderer's workload, we deprive the first system of a potential performance advantage. However, models with nested transformations probably represent more accurately the workload of production-quality, realistic renderers. This is the major factor behind Mg's design.[2]

Geometric data are described in a left-handed coordinate system, in which the positive x, y, and z axes point right, up, and forward, respectively. In this system, polygon vertices are given in clockwise order.

---

[2] It is, of course possible, to perform the transformations within Mg's output module and supply the rendering system with absolute world coordinates. Such a perverse deviation from the interface violates the intent of the benchmarks and calls for careful documentation when reporting test results.

A rendering system that supports nested transformations maintains at all times a *current* transformation matrix. Any new transformation operation modifies the current transformation. Mg supports nested transformations with *OutputSave*, to save a copy of the current transformation matrix, and *OutputRestore*, to restore the matrix from the top of the stack. *OutputIdentity* initializes the current transformation matrix to the identity matrix. Mg does not maintain the transformation stack itself, but passes the commands to the renderer.

Three transformations can modify the current transformation matrix. *OutputTranslate* moves objects by the specified distance in x, y, and z. *OutputScale* changes the size of objects by the specified factors in x, y, and z. *OutputRotate* specifies a rotation in degrees. The axis of rotation passes through the origin and the specified point. The direction of the rotation is defined by a left-handed coordinate system; with the left thumb pointing from the origin to the specified point, the rotation follows the direction of the curled fingers.

OutputSave()
OutputRestore()
OutputIdentity()
OutputTranslate(dx, dy, dz)
OutputScale(sx, sy, sz)
OutputRotate(angle, x, y, z)

Table 4.4. Matrix commands.

4.4.3. Primitives

Mg uses three types of primitives: quadric surfaces, polygons, and bicubic patches. Polygons are planar and convex; they do not have holes. Initially, Mg supports only bézier patches, but support for other types of patches could be added in the future.

The quadric surfaces include spheres, cones, and cylinders. *OutputSphere* specifies the radius of a sphere centered on the origin. *OutputCone* describes a cone by its height and the radius of its base. The base lies on the x,y plane. The z axis passes through the center of the base and the apex. *OutputCylinder* specifies the radius of a cylinder about the z axis. Its height is given by z coordinates for the bottom and top. Cylinders are open on top and bottom, and cones are open on the bottom. All of these primitives are positioned through a sequence of transformations.

Both polygons and parametric patches are described by structured lists of vertices. Two data structures are commonly used for vertex lists. Consider the case of a four-sided polygon. In one representation, the data is fully expanded, and a list of four triples gives the vertex coordinates. In the other representation, two lists specify the data hierarchically. First, a vertex list gives the coordinates of each vertex. Then, a polygon list points to the four vertices used by the primitive.

Each representation has advantages and disadvantages. The fully-expanded representation normally requires more data to be transmitted, because shared vertices must be repeated in full. On the other hand, it does not require that any state be maintained. The hierarchical representation is more compact, but it requires state. Because the the hierarchical representation specifies each vertex just once, it reduces problems with inconsistencies and round-off errors. Mg's interface uses the hierarchical representation in the interest of portability; it is easier to go from the hierarchical to the fully expanded representation than the reverse.

Vertices are numbered implicitly from zero. The counter is incremented when *OutputVertex* is called to specify the position for a new vertex. *OutputVertexReset* resets the counter. The interface

OutputSphere(radius)
OutputCone(radius, height)
OutputCylinder(radius, bottom, top)
OutputVertex(position)
OutputVertexReset()
OutputVertexNormal(xyz)
OutputVertexColor(rgb)
OutputVertexST(s, t)
OutputPolygon(nvertices, vertex_pointers[], normal_flag, color_flag, texture_flag)
OutputPointsPolygons(npolygons, nvertices[], vertex_pointers[], normal_flag, color_flag, texture_flag)
OutputPatch(basis, vertex_pointers[])

Table 4.5. Geometric primitives.

supports additional, but optional, information for polygon vertices. When polygons are used to approximate a curved surface, *OutputVertexNormal* specifies the direction of the true surface normal at the current vertex. *OutputVertexColor* specifies a color for the current vertex, and *OutputVertexST* specifies texture coordinates, s and t, for the current vertex.

Polygons and patches are described by lists of vertex numbers. The first two arguments to *OutputPolygon* give the number of vertices in the list and the array of vertex numbers. Three flags indicate whether the vertices have normals, colors, or texture coordinates. *OutputPointsPolygons* specifies multiple polygons that share a common set of vertices. Its first argument gives the number of polygons. The second argument is an array of integers, with one entry for each polygon in the list. This array gives the number of vertices in each polygon. The vertex pointer array lists the vertices for all of the polygons. The flag arguments are the same as for *OutputPolygon*. *OutputPatch* specifies the type of bicubic patch and a vertex list with sixteen control points in a four-by-four matrix.

4.4.4. Surface Properties and Shading Information

Each surface is described by its material properties, including color, opacity, coefficients of reflection, and the index of refraction. These attributes are set by three procedures: *OutputColor*, *OutputOpacity*, and *OutputSurface*. Typically, a single set of surface properties describes several primitives that form a high-level object. It is not necessary to specify the properties individually for each primitive. Instead, a surface attribute can be set once, and it will remain in effect until it is explicitly changed. If a color has been specified individually for a polygon vertex, it overrides the current color.

Color consists of red, green, and blue components. Both color and opacity are specified on a scale of 0 to 1. For a completely transparent surface opacity is set to 0, and for an opaque surface it is set to 1. The opacity is specified for separately for the red, green, and blue channels. With transparent materials, the index of refraction is supplied for shaders that compute more accurate refraction effects.

Mg supports three of RenderMan's standard surface types: constant, matte, and plastic [Upst90]. The constant surface model ignores light sources and uses the current color for all points on the surface. This is the most simple model, useful for baseline measurements. A plastic surface uses ambient, diffuse, and specular reflectance coefficients, while a matte surface uses only ambient and diffuse (Figure 2.5). A fourth surface model, reflective, is used for metallic surfaces that reflect objects in the environment. It uses ambient and diffuse coefficients and assumes that the reflection coefficient is 1. *OutputSurface* describes the material's reflectance properties. It specifies a surface model, ambient, diffuse, and specular coefficients, and a roughness factor. The roughness factor controls specular reflection. Specular

OutputColor(rgb)
OutputOpacity(percent, refraction)
OutputSurface(model, Ka, Kd, Ks, roughness)

Table 4.6. Shading commands.

highlights are sharpest when the surface is most smooth, as indicated by a roughness value close to 0. With a rougher surface, the highlight becomes more broad.

Mg generates surface attributes, because they are intrinsic characteristics of a scene. It does not, however, specify details of the shading model, such as the shading frequency or the interpolation algorithm for smooth shading. More sophisticated surface models and atmospheric affects are omitted from the interface in order to limit Mg's complexity and make its interface more general. Many rendering systems will not support all of the surface attributes that are in the interface, and any deviations from the specified surface models should be reported.

4.4.5. Texture Maps

Mg supports texture maps that modify the surface color. Its textures have horizontal or vertical stripes in varying widths and colors. Multiple texture maps may affect a surface. The utility in *mg_texture.c* generates the specifications for ten texture images with three channels of information: red, green, and blue. Texture images can be converted to the texture map format required by the rendering system. Normally, the conversion is a pre-processing step that precedes rendering. Because the colors in a texture map do not generally affect performance, the experimenter may substitute locally available texture maps.[3] To match the performance characteristics of Mg's textures, any texture maps should contain three channels, with eight bits per channel. If a subjective evaluation of antialiasing is important, textures with regular detail or hard edges are useful.

One or more textures can be applied to a single surface. The final color is a blend of all textures applied to the surface, $\frac{1}{n}\sum_{i=1}^{n} tex_i$, where tex_i is the color obtained from the ith texture.

Texture mapping requires two types of information: a texture name and a function that maps points in the texture map onto a three-dimensional surface. Mg leaves the actual naming of the texture files to the system-dependent output routines. *OutputTxSurface* simply specifies the number of texture maps to be applied to a surface. It also gives ambient and diffuse coefficients for shading the textured surface. Mg conforms to the RenderMan conventions for mapping between a surface's parameter space and the texture space. The texture map always covers the unit square in its s and t coordinates. By default, the texture coordinates of a parametric or quadric surface also range from zero to one, so that the texture covers the surface exactly. A different mapping is specified by *OutputTxCoords* which gives four pairs of texture coordinates. They corresponding to (0,0), (1,0), (0,1), and (1,1) in the surface's parameter space. The mapping specified by *OutputTxCoords* applies to all subsequent patches and quadric surfaces until another call changes the mapping. Texture coordinates for polygons must be specified for individual vertices. *OutputVertexST* assigns texture coordinates to polygon vertices, as documented in Section 4.4.3.

[3] In the case of an adaptive sampling algorithm, the frequency of detail in the texture maps can indeed affect performance. When comparing such a system with another, the experimenter should use Mg's textures, or others with the same frequency of detail.

OutputTxSurface(n, ka, kd)
OutputTxCoords(s[4], t[4])
OutputVertexST(s, t)

Table 4.7. Texture map commands.

4.5. Controllable Parameters

One of Mg's most important features is its support for constructing performance experiments by varying the values of a carefully chosen set of parameters. This section gives an overview of the controllable parameters. Some parameters apply to all scene generators; in general these are the parameters that control viewing and lighting. Other parameters, which directly affect the scene description, are unique to each scene generator.

Three types of parameters are common to all scene generators: the resolution, the field of view, and the lighting controls. The default values all depend on the scene. Changing the field of view is similar to changing the focal length of a lens on a camera. With a wider field of view, objects appear smaller on the screen, and with a narrower field of view they appear larger. Moving the viewpoint would also change the apparent size of objects, but, unlike the viewing position, the field of view is independent of the coordinates used in the scene definition. Each scene defines a fixed set of light sources. By default all lights in the scene are on. To control the number of light sources, one can turn off selected lights. By convention, the light sources are numbered from zero, and the first is an ambient light source.

Geometric options vary with the scene, and they have no uniform syntax. The options for each scene are described in the following section. Typically, the types of geometric primitives can be varied. Other options may control the number, size, or spatial distribution of objects. The spatial distribution affects the performance of visibility algorithms by varying the number of visible surfaces, the screen area of visible surfaces, the depth complexity, and the screen coverage. The spatial distribution also affects the performance of multi-processor systems that use a spatial subdivision to partition the workload.

Shading and texturing options also depend on the scene. Options can control the number of transparent or reflective objects, the number of texture maps, and the number of textured surfaces.

function	**flag**	**arguments**	**default**
resolution	–r	*xres yres*	depends on scene
field of view	–f	*angle*	depends on scene
turn $light_i$ off	–L*i*		on

Table 4.8. Universal scene generator options.

4.6. Test Scenes

The primary design goal for the test scenes is to support controlled variation of the scene characteristics, and to make the effects of the variation easy to understand. It should be possible to test the limits of systems with characteristics like very high depth complexity or many small objects. It is less important that the resulting images look like pictures from any typical workload. The types of scenes are designed to display different characteristics and to allow different types of controlled variation. Two scene generators have been implemented for Mg. *Spheres* creates scenes that contain a variable number of spheres. It varies the spatial distribution of objects and the material properties of objects, including transparency and reflection. *Terrain* creates layered, terrain-like surfaces with surface textures. This section describes the types of images that each scene generator produces, and the options that apply to the individual scenes. Although the scenes can be varied, there is a default setting for each parameter.

4.6.1. Spheres

Spheres uses simple geometry, but it allows the experimenter to vary many properties. Its scenes display only spheres, but the underlying primitives vary. Each object can be modeled by one sphere, eight bézier patches, or one hundred ninety-two triangles. The scene can be varied by changing the number, size, and spatial distribution of the spheres. By default, the radius varies inversely with the number of objects, so that the total projected screen area remains approximately constant. Thus, the experimenter can evaluate the cost of increasing the number of objects without increasing the total rendered area. The size option specifies a scale factor, which multiplies the default radius. Visibility characteristics are varied by selecting one of three types of spatial arrangements: randomly distributed spheres within a three-dimensional bounding volume, grids of non-overlapping spheres, or stacked grids of spheres with many surfaces obscured. The model generator uses pseudo-random numbers to position the randomly scattered spheres. The x, y, and z coordinates can be taken from either a uniform or a Gaussian distribution. When the spheres are scattered randomly, many of the spheres intersect. When the spheres are placed in grids, they do not intersect unless the size factor is greater than 1.2.

One option generates a background polygon, covered with one texture. Besides adding depth to the scene, the background polygon makes the effects of shadowing and refraction more visible. By default, there is no background polygon and the image is empty wherever there are no spheres.

function	flag	arguments	description	default
name	–N	*filename*	use *filename* for model output	spheres
resolution	–r	*xres yres*		512 by 512
field of view	–f	*angle*		30 degrees
turn light$_i$ off	–L0		white ambient light	on
	–L1		white point light	on
	–L2		red point light	on
	–L3		green point light	on
spatial distribution	–du		random, uniform distribution	–du
	–dG		random, Gaussian distribution	
	–dg	*n*	*n* by *n* grid, not overlapped	
	–ds	*n m*	*n* by *n* grid, stacked *m* deep	
number of spheres	–n	*n*	make *n* spheres	100
type of primitive	–pb		bezier patches	spheres
	–pp		polygons	
	–pP		points polygons format	
	–ps		spheres	
size of spheres	–s	*s*	scale size of spheres by factor *s*	1.0
background	–b		add textured background polygon	none
constant shading	–c		for all surfaces	off
transparent	–t	[*p*]	p = proportion transparent $0 \le p \le 1$ p defaults to 0.2 if –t specified	0.
metallic	–m	[*p*]	p = proportion reflective $0 \le p \le 1$ p defaults to 0.2 if –m specified	0.

Table 4.9. Spheres scene generator options and their defaults. Spheres are never both transparent and metallic; if both **–t** and **–m** are given, the sum of the percentages must not exceed 1.

Surface characteristic can also vary. One option controls the fraction of spheres that are transparent, and another controls the fraction that are reflective. (No spheres are both transparent and reflective.) By default, all of the spheres are opaque and non-reflective. Random numbers are used to assign surface characteristics, so the exact proportions of the different surface types are approximate. However, the same sequence of pseudo random numbers is always generated, so the proportions are repeatable. Table 4.9 details all of the options that control scene generation.

4.6.2. Terrain

Terrain covers much of the screen with terrain-like collections of parametric patches or polygons. It simulates scenes such as landscapes or backgrounds for animation, and it can be used to explore texture mapping performance. The model can contain from one to ten separate terrain surfaces, each composed of nine hundred primitives. Table 4.10 details all of the options that control scene generation. Five views of the scene are available. The first is a closeup view of one surface and resembles a landscape. The other views look at a stack of surfaces from either the front or the top. The terrain elements can be stacked directly on top of each other or offset so that more of each surface is visible. Texture maps can be used to specify surface colors. Parameters control the number of textured surfaces and the number of textures per surface. By using finely-detailed textures that are prone to aliasing, an experimenter can also study antialiasing and filtering.

function	**flag**	**arguments**	**description**	**default**
name	–N	*filename*	use *filename* for model output	terrain
resolution field of view	–r –f	*xres yres* *angle*		640 by 480 45 degrees
constant shading	–c		for all surfaces	off
turn $light_i$ off	–L0 –L1		white ambient light white distant light	on on
number of surfaces	–n	*n*	make *n* terrain surfaces	1 (view 0) 10 (views 1-4)
type of primitive	–pb –pp –pP		bezier patches polygons points polygons format	patches
texture maps	–t	*m n*	apply textures to *m* surfaces *n* textures per surface	no textures
select view	–v0 –v1 –v2 –v3 –v4		landscape front top front, "spread out" top, "spread out"	view 0

Table 4.10. Terrain scene generator options and their defaults.

4.6.3. A Benchmark Suite

A set of five varied test cases has been selected as a benchmark suite. The suite consists of three sphere scenes and two terrain scenes. Table 4.11 specifies the suite by listing the command line options necessary to generate each of the scenes. Some characteristics of the scenes are summarized in Table 4.12, and Figure 4.1 shows the resulting images. The images are reproduced in black and white. For clarity, white backgrounds were added to some of the images.

The first test case, *sphere100*, has very simple geometry and somewhat more complex shading requirements. It contains one hundred randomly scattered spheres. Approximately half of the objects have a simple plastic surface model, but the rest are either transparent or reflective. The coordinates that position the objects are taken from a uniform distribution. A textured polygon fills the background.

The second scene, *sphere1600*, contains sixteen hundred randomly distributed spheres. The position coordinates are taken from a Gaussian distribution, so that the spheres tend to be less evenly distributed over the screen. The depth complexity has more variability and a greater maximum value with the Gaussian distribution than with a uniform random distribution. Because of the small size of the objects and the central cluster, this scene is more sparse than the others.

The third scene, *stack*, displays a large number of primitives and covers most of the screen with a dense, but regular, arrangement of objects. The spheres are arranged in a grid, thirty by thirty by five deep. Each sphere is modeled by eight bézier patches, for a total of 36,000 primitives. The spheres all have the same radius, but the perspective transformation gives the more distant spheres a smaller screen size. This scene has the most complex geometry of the suite, but the shading is simple.

name	**specification**
sphere100	spheres –n 100 –t .3 –m .2 –s 1.5
sphere1600	spheres –n 1600 –dG
stack	spheres –ds 30 5 –pb
landscape	terrain –v0 –t 1 8
layers	terrain –v4 –pp –T 6 1

Table 4.11. Suite of five benchmarks.

Scene	**sphere100**	**sphere1600**	**stack**	**landscape**	**layers**
Primitives	100 spheres 1 polygon	1600 spheres	36,000 patches	900 patches	9,000 polygons
Resolution	512 by 512	512 by 512	512 by 512	640 by 480	640 by 480
%Coverage	100	35	77	63	76
Light sources (type)	3 (point)	3 (point)	3 (point)	1 (distant)	1 (distant)
Reflection	yes	no	no	no	no
Transparency	yes	no	no	no	no
Textures	1	0	0	8	6

Table 4.12. Characteristics of five benchmark images.

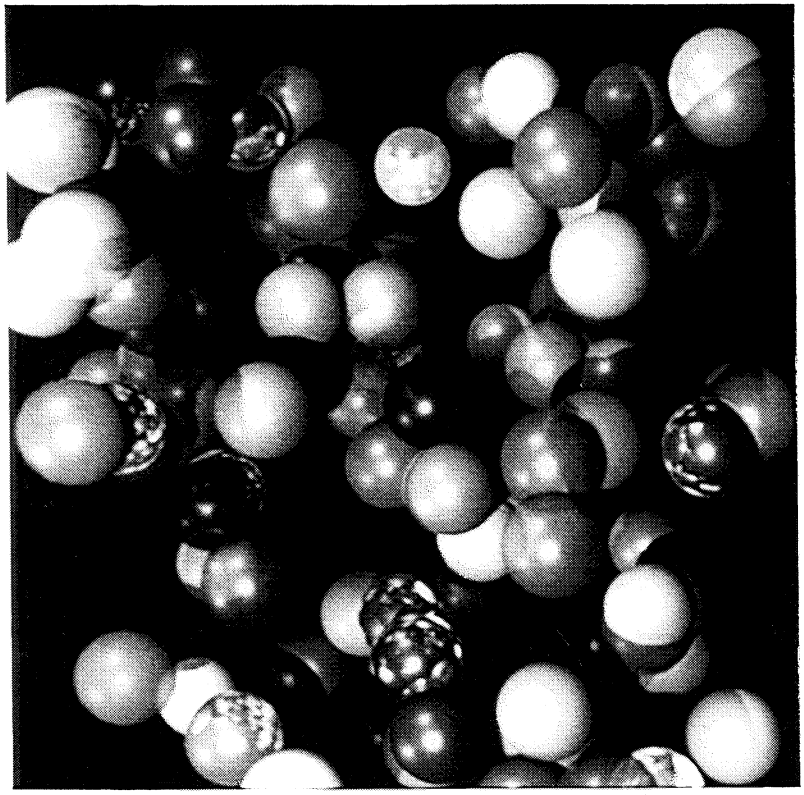


Figure 4.1(a). Sphere100.

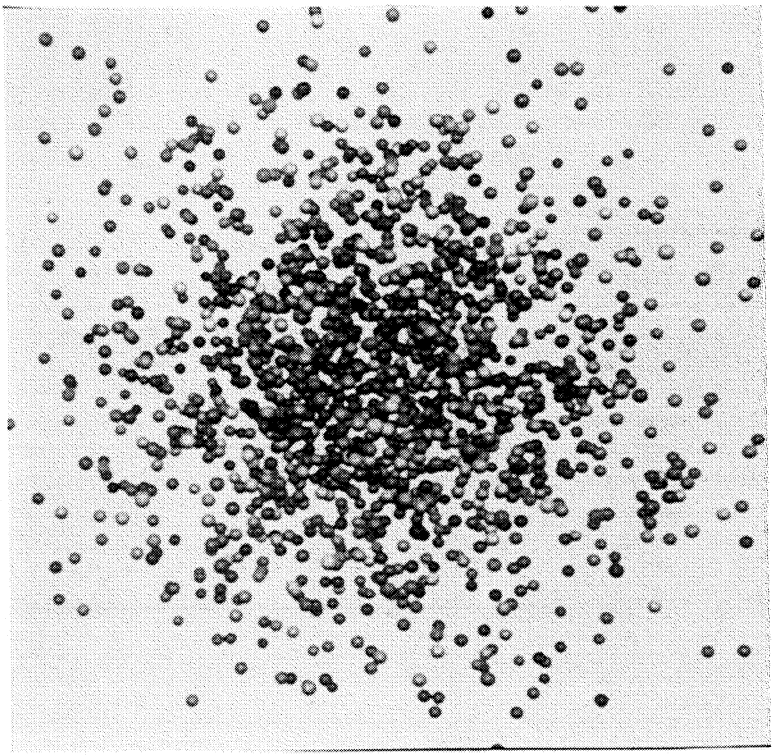


Figure 4.1(b). Sphere1600.

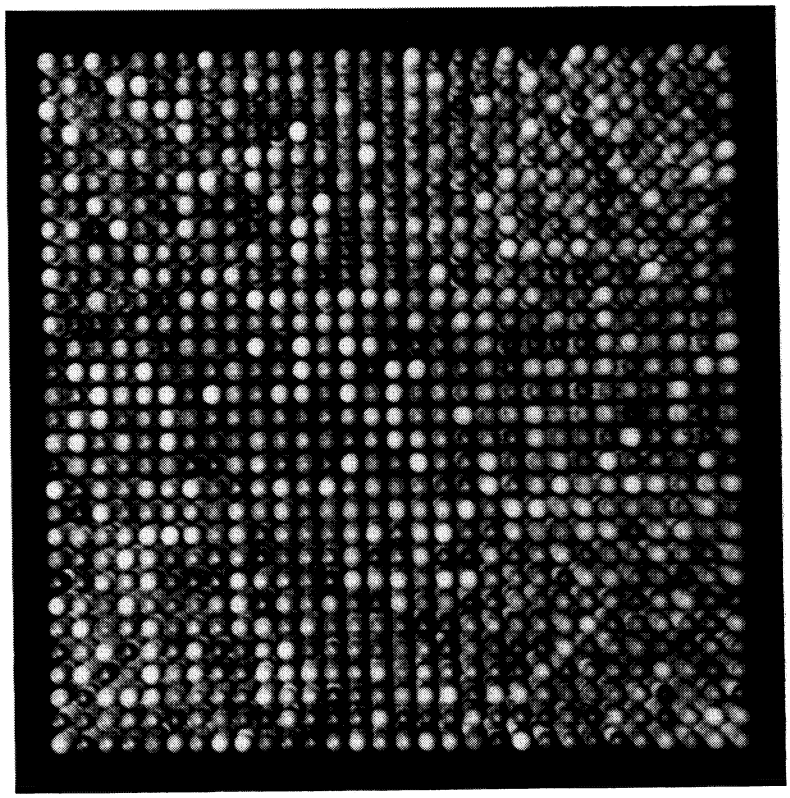


Figure 4.1(c). Stack.

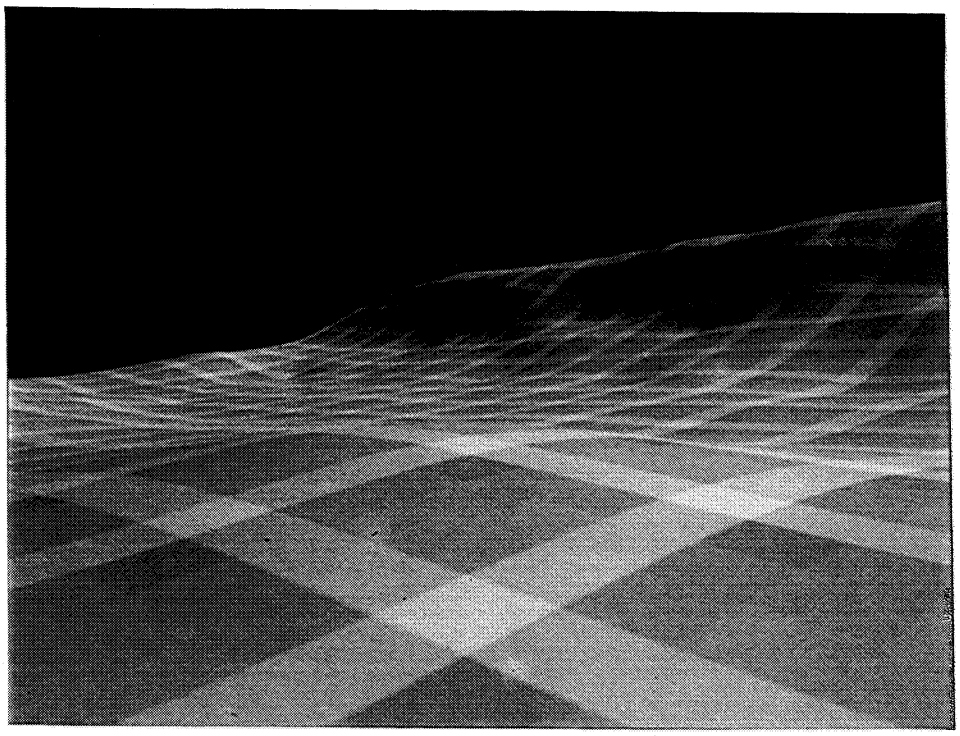


Figure 4.1(d). Landscape.

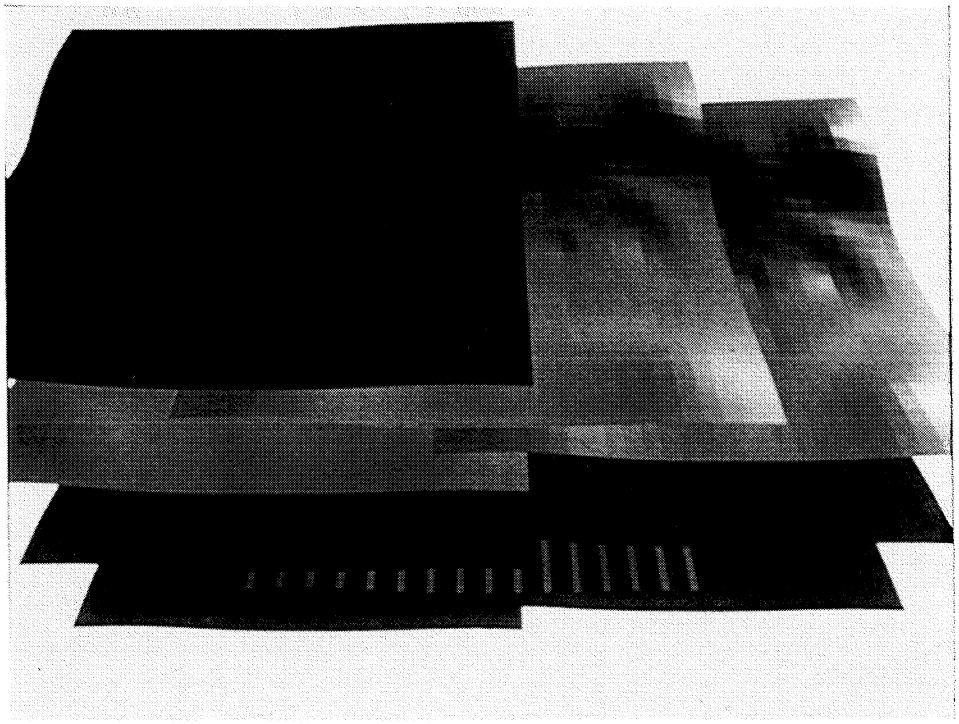


Figure 4.1(e). Layers.

One terrain scene, *landscape*, is a close-up view of a heavily-textured terrain surface. Eight separate texture maps determine its color. The surface is modeled by 900 bézier patches, and approximately one-third of the surface is visible in the image. The second terrain scene, *layers*, views ten terrain surfaces from a position above, and slightly to the front of, the layered objects. The bottom six surfaces have one texture each. The top four surfaces, which are untextured, obscure most of the textured surfaces. Although more than half of the objects have textures, less than a quarter of the visible pixels are textured.

4.7. Reporting the Results

Unless test conditions are fully documented, it is difficult to reproduce an experiment. When experiments are based on Mg, there are four types of information that should be included in a complete report.

First, the report must contain enough information to allow a reader to generate the same model file. The fixed benchmark suite may be used, and each scene generator outputs a default version when invoked with no arguments. But, scene characteristics can be varied by manipulating parameters. Therefore, the report must specify any deviations from the default values. Parameters are varied by changing the command-line options. If the scene generator program is run again with the same options, it will generate the same scene and viewing specifications. Thus, the most convenient documentation of the scene specifications is a listing of the command line. In addition, each scene generator produces a report that details the results of all of the options with which it was invoked.

Second, the report should describe the the computing environment, including both hardware and software. The report should identify which operations are included in the rendering time, and which operations are performed in separate pre-processing steps.

Third, the report should note any important features of Mg's implementation on the target system. For example, Mg provides surface characteristics as input to the renderer's shading routines. How the renderer's illumination model uses these values can vary from system to system. The report should also document any unimplemented features, such as refraction for transparent surfaces. Finally, any deviations from the interface specifications should be noted.

Fourth, the report should document the relevant rendering parameters, as described in Section 4.3. These include various run-time options given to the rendering system.

4.8. Implementation Experience

Mg has been ported to two different rendering systems. The first is PhotoRealistic RenderMan (*prman*), which is described in Section 3.3. The second is a ray tracer called *opal* [Apod92a]. One goal of the implementation effort is to demonstrate that Mg can help identify performance differences among algorithms. These systems were selected because of their different approaches to rendering and because both renderers were available on the same computers.

To port Mg, the output library procedures in *mg_output.c* were modified. Both the prman and opal versions write model files using the RenderMan Interface Bytestream Protocol, or RIB [Pixa89]. The differences in their output correspond to the systems' varying support for transparency and reflection. Appendix B contains all of the prman version of *mg_output.c* and the code for the procedures which differ in the opal version.

4.8.1. An Image-Space Renderer

The prman implementation supports all of Mg's geometric features, including intersecting objects. It outputs the RenderMan Interface Bytestream Protocol, or RIB. The RIB requests for geometric primitives and matrix operations correspond closely to Mg's own interface, except in the handling of vertex information. When colors, normals, or texture coordinates are specified for polygon vertices, the output module stores the values. The vertex information is later written to the RIB file by OutputPolygon or OutputPointsPolygons.

Table 4.13 summarizes the surface models supported by the prman implementation. Three types use standard RenderMan shaders: constant, matte, and plastic. There is no support for refraction with transparent surfaces or for truly reflective surfaces. The index of refraction is ignored, and reflective surfaces are rendered with the standard "metallic" shader which does not provide reflections of objects in the environment. Textures are supported by invoking custom shaders. When one texture is applied to a surface, the shader rotates among the ten texture maps that are supplied. When multiple textures are specified, a fixed set of textures are used and the shader averages the values obtained from the texture maps. The light sources use the standard RenderMan models: ambient, distant, and point light sources. The renderer does not implement shadowing directly, but it can simulate shadows with texture mapping techniques. None of the experiments described in this chapter include shadows.[4]

surface type	description
constant	color given by most recent call to OutputColor()
matte	ambient and diffuse components
plastic	ambient, diffuse, and specular components
reflective	non-reflective *metal* surface ambient and specular components
texture	average of all texture map values, ambient and diffuse components

Table 4.13. Surface types for prman implementation.

4.8.2. A Ray Tracer

The opal ray tracer uses the same RIB input as prman. It performs stochastic supersampling and improves its performance with the Kay-Kajiya bounding volume algorithm [Kay86] and shadow ray caching. Its proprietary adaptive supersampling algorithm is still under development. Opal's support for the RenderMan shading language precludes some optimizations commonly found in other ray tracers, which can assume a uniform illumination model [Apod92b].

The version of opal used in the experiments is a prototype, compiled for debugging and without optimization. Thus, it is not always meaningful to compare its runtimes directly against prman. Instead, the analysis will compare each system to itself and examine how runtimes change in response to changes in the renderer's input.

The opal implementation of Mg differs from the prman version only in its support for refraction and reflection. The output module invokes custom shaders that cast secondary rays in the direction of reflection or transmittance. (Surfaces generated by Mg can be either transparent or reflective, but not both.) Opal allows secondary rays to be disabled at runtime, in which case it casts only primary rays from the eye. In the experiments, secondary rays were enabled for scenes with reflective or transparent surfaces. Shadow rays are also controlled separately at runtime. The expense of computing shadows ruled out the use of shadow rays in most of the experiments.

4.9. Experiments using Mg

This section documents a few simple experiments using Mg. These experiments demonstrate the use of controlled variation and show how the performance characteristics of prman and opal are affected by their different algorithms. One set of experiments explores prman's success in culling hidden surfaces before shading.

[4] The performance of the shadow map algorithm has been discussed by its implementers [Reev87].

The experiments were run on an SGI Iris Crimson, a fifty megahertz system with a MIPS R4000 cpu and sixty-four megabytes of main memory. The operating system is System V UNIX. In all of the experiments, images were written to a file in the TIFF format (Tagged Image File Format). Files for input and output were accessed over a local network. Unless otherwise noted, images were computed without shadows, which reduced the compute time considerably and gave the renderers more comparable workloads. Except where noted, rendering system options took their default values. Where applicable, the defaults are the same for prman and opal. In particular, the images were computed with four jittered samples per pixel using a 2 by 2 Gaussian filter. The shading rate, which controls the maximum screen distance between shading samples, was one pixel. Compute times are an average of two or more trials for most of the runs under one minute.

4.9.1. Comparing Two Rendering Systems

Both systems rendered the five images in the benchmark suite (), varying the resolution. The spheres models were rendered at 128 by 128, 256 by 256, 512 by 512, and 1024 by 1024. The terrain models were rendered at 160 by 120, 320 by 240, 640 by 480, and 1280 by 960. Tables 4.14 and 4.15 list the cpu time measurements. The times are plotted in Figures 4.2 and 4.3.

The measurements show how the compute times increase for both renderers as the image resolution increases. For the ray tracer, the increase is close to linear in the number of pixels (doubling the resolution in each direction results in a four-fold increase in pixels). This is expected, since the number of primary rays is determined by the number of pixels. Prman's time also increases with the resolution, but more slowly. This increase is also expected, because the shading and sampling frequencies depend on the resolution.

	User CPU (hh:mm:ss)				User CPU (normalized)			
resolution	**0.25**	**0.5**	**1**	**2**	**0.25**	**0.5**	**1**	**2**
sphere100	0:16	0:55	2:36	7:52	1.00	3.44	9.75	29.50
sphere1600	0:44	1:00	1:39	4:06	1.00	1.36	2.25	5.59
stack	1:23	2:59	8:14	16:02	1.00	2.16	5.95	11.59
landscape	0:22	1:08	3:51	14:02	1.00	3.09	10.50	38.27
layers	0:08	0:16	0:48	2:59	1.00	2.00	6.00	22.38
average	0:35	1:16	3:26	9:00	1.00	2.29	6.07	17.49

Table 4.14. Prman. User cpu time for rendering five benchmark images. The resolution is expressed relative to the default resolution along one axis. The line labeled *average* shows the arithmetic mean of the raw cpu times and the geometric mean of the normalized times.

	User CPU (hh:mm:ss)				User CPU (normalized)			
resolution	**0.25**	**0.5**	**1**	**2**	**0.25**	**0.5**	**1**	**2**
sphere100	2:41	10:42	42:14	2:48:23	1.00	3.99	15.74	62.75
sphere1600	0:25	1:19	4:40	17:17	1.00	3.16	11.20	41.48
stack	5:14	16:37	1:03:46	4:01:41	1.00	3.18	12.30	46.18
landscape	4:37	18:14	42:56	4:46:01	1.00	3.95	9.30	61.95
layers	0:58	3:02	10:46	41:10	1.00	3.14	11.14	42.59
average	2:57	9:59	32:52	2:30:42	1.00	3.46	11.76	50.15

Table 4.15. Opal. User cpu time for rendering five benchmark images. The resolution is expressed relative to the default resolution along one axis. The line labeled *average* shows the arithmetic mean of the raw cpu times and the geometric mean of the normalized times.

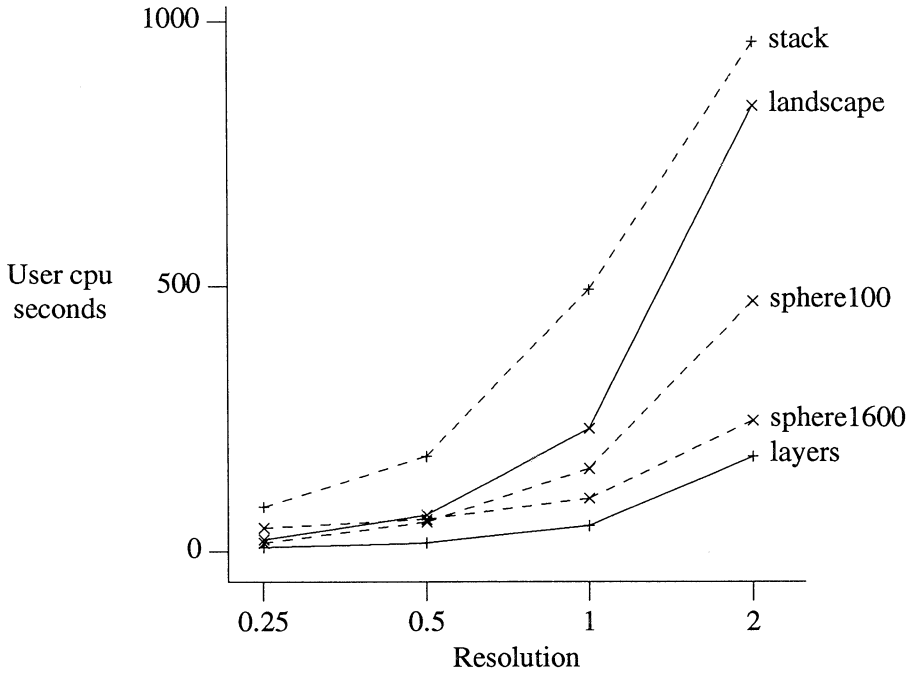


Figure 4.2. Prman. User cpu time for rendering five benchmark images. The x axis plots the resolution (as a fraction of the default resolution in one dimension) on a log scale.

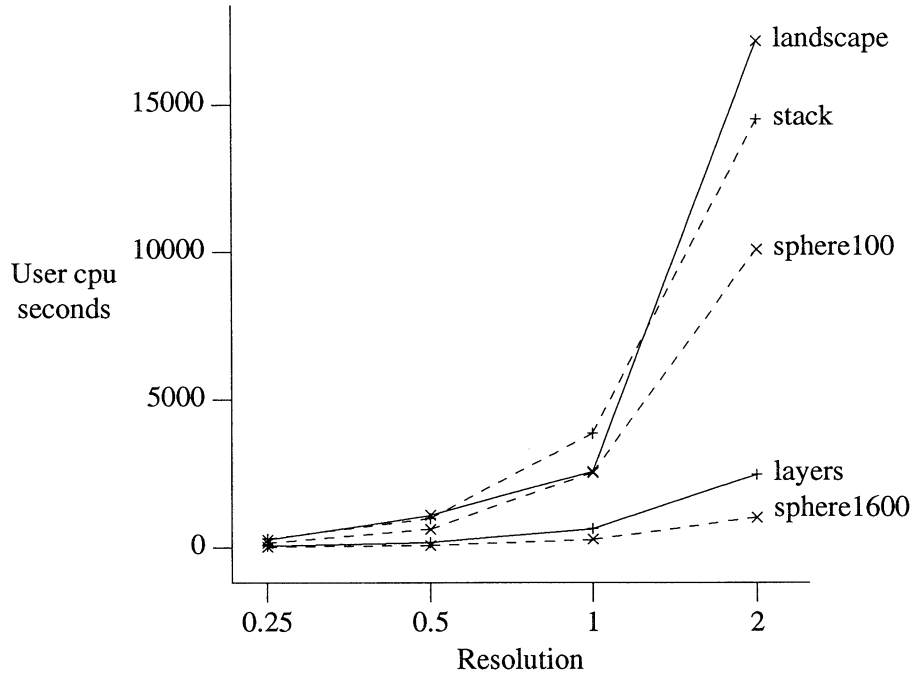


Figure 4.3. Opal. User cpu time for rendering five benchmark images. The x axis plots the resolution (as a fraction of the default resolution in one dimension) on a log scale.

Sphere100, which contains transparent and reflective objects, is moderately difficult for both renderers. However, opal solves a more difficult problem by rendering transparency with refraction and inter-object reflections. Prman rendered the transparent surfaces without refraction and did not simulate reflections.

Stack is the most difficult test case for prman at all resolutions. It is hard because of its depth complexity and the large number of primitives. Prman reduces, but does not eliminate, the costs associate with depth complexity with algorithms that cull hidden surfaces before shading. Opal is effected less by depth complexity, because of algorithms that organize objects into spatial hierarchies. Section 4.9.3 explores in more detail the effect of depth complexity on prman's performance.

For opal, no one image stands out as the most expensive at all resolutions. *Stack* is one of the most difficult for opal as well as prman, but probably for a different reason. The model contains a large number of bézier patches. Intersecting a ray with a patch is costly, while ray-sphere intersection is comparatively easy. Despite opal's relatively efficient ray-patch intersection code, patches still slow down the renderer.

To compare the relative costs of spheres and patches for both renderers, *stack* was modeled and rendered with spheres instead of patches.[5] Figure 4.4 plots the compute time with both primitive types for the two systems, and Table 4.16 lists the data. For opal, the patches are more costly than spheres by approximately a factor of six at all of the tested resolutions. With spheres, opal is so efficient that it renders the low-resolution images more quickly than prman. For prman, the costs are more even. At different resolutions, the tradeoff between processing more objects (patches) and performing more subdivision (spheres) favors different types.

The spheres scene generator uses eight patches to approximate one sphere. Whereas the patch model for *stack* contains 36,000 primitives, the sphere model has only 4,500. How are the two renderers affected by varying the number of primitives in the scene? A simple experiment varied the number of spheres, increasing n by a factor of four each time: 25, 100, 400, 1,600, 6,400, and 25,600. Figures 4.5 and 4.6 plot the compute time against the number of spheres. First, the spheres were arranged in a $\sqrt{n}$ by $\sqrt{n}$ grid. As n increased, the radius of the spheres decreased, so that the total screen area remained approximately the same. According to opal's statistics, the number of primary rays that intersected spheres varied by no more than eight percent. This statistic is proportional to the screen area covered by the objects in the scene. The largest example showed the most variation, but its objects are extremely small. If we consider only the images with 6,400 spheres or less, the number of intersections varied by less than two percent. All images were rendered at a resolution of 512 by 512.

When the spheres are arranged in a regular grid, no sphere is obscured by another. The spheres were also scattered randomly, hiding some spheres behind others (Figure 4.6). The screen area, as estimated by the number of primary rays that intersected spheres, was about thirty percent less than for the grid arrangement. This statistic varied by no more than eight percent as the number of spheres increased.

The compute time for opal is virtually unaffected by the increase in the number of objects when the spheres are placed in a grid. For the scattered spheres, opal's runtime still increases slowly. (As the spheres become smaller, they are less likely to hide each other and the visible surfaces cover more screen area.) We saw above that opal renders patches much more slowly than spheres. This experiment establishes that the eight-to-one ratio of patches to spheres is not the problem. In contrast to opal, prman's effort increases in proportion to the number of spheres. The cross-over point between the two algorithms

[5] It was not practical to create the same scene with polygons, because the model file would be too large, and the renderers would have problems with memory and swap space. The sphere scene generator would create 864,000 polygons for the 30 by 30 by 5 grid displayed in *stack*. The model file would be over 32Mb in the points-polygons format and almost 150Mb using individual polygons.

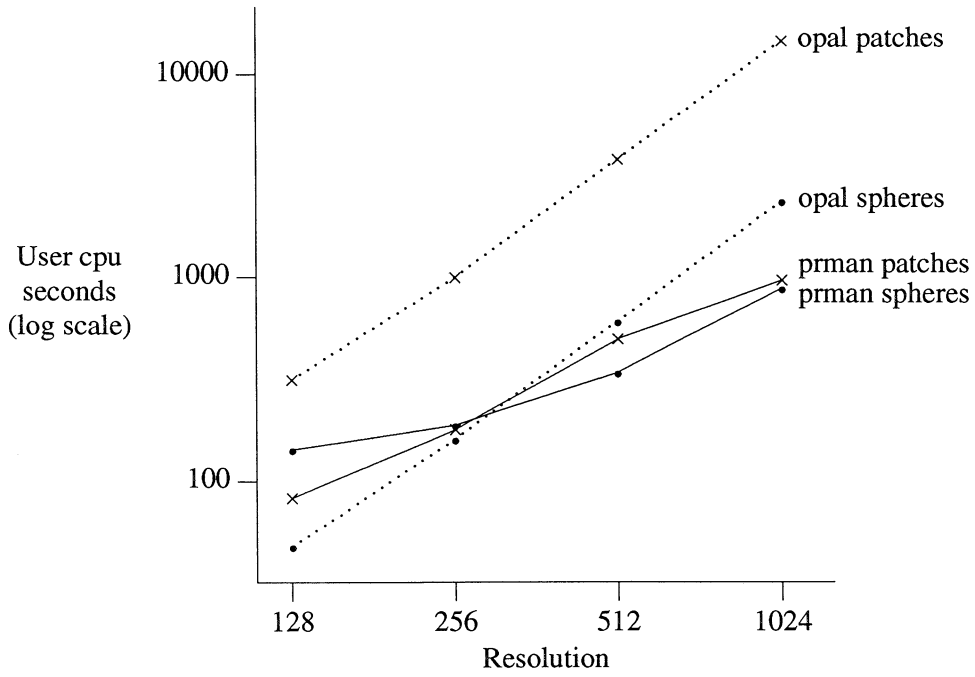


Figure 4.4. Comparison of spheres and patches, prman and opal.

	resolution			
	128	**256**	**512**	**1024**
prman spheres	2:23	3:08	5:41	14:41
prman patches	1:23	2:59	8:14	16:02
opal spheres	0:48	2:41	10:04	39:15
opal patches	5:14	16:37	1:03:46	4:01:41

Table 4.16. Comparison of spheres and patches, prman and opal.

depends on many characteristics of the workload. As we have seen, a ray tracer can process spheres very efficiently, and opal's performance would suffer with less efficient primitives.

It is also worth noting that the test scene contained no transparency or reflections and it was rendered without shadows. Therefore, opal performed only one-level ray casting as opposed to true recursive ray tracing. There are some significant differences between the two systems in shading and texturing performance. Prman's algorithms were designed to improve shading coherence and optimize texture mapping. On the other hand, ray tracers perform texturing and shading with less coherence. Three of the benchmark images, *sphere100*, *landscape*, and *layers*, contain textures. In general, prman shows a better relative performance on the textured images. *Landscape*, which has extensive texturing, is the most difficult image for opal at the highest resolution. The next section discusses shading and texturing in more detail.

4.9.2. Shading and Texturing Issues

Ray tracers, such as opal, support global illumination effects such as reflection, transparency with refraction, and shadows by following secondary rays in the appropriate directions. In the five-image benchmark suite, only *sphere100* contains reflective and transparent surfaces. The opal timings in the

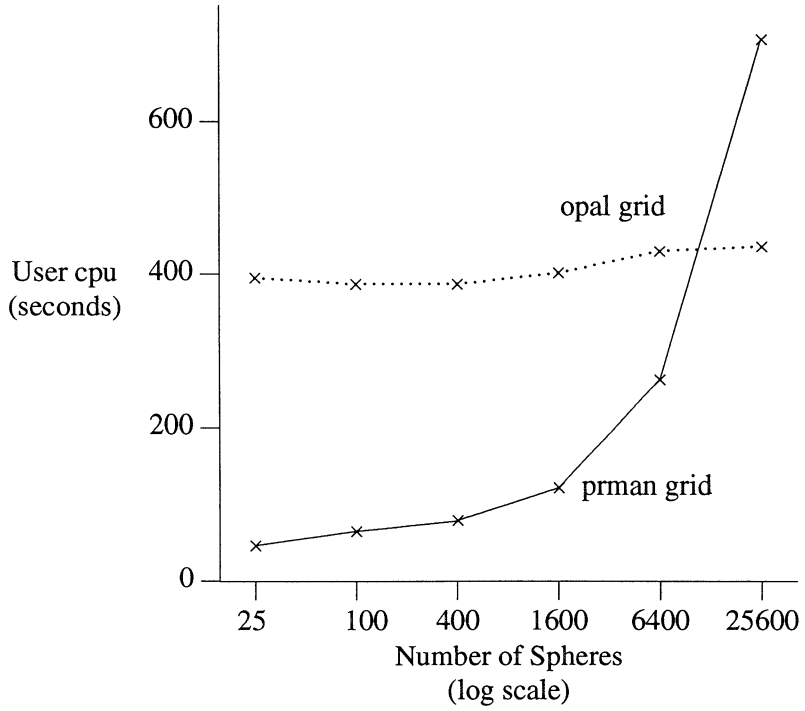


Figure 4.5. Grid. Varying the number of spheres, prman and opal. Each image contains spheres in a $\sqrt{n}$ by $\sqrt{n}$ grid.

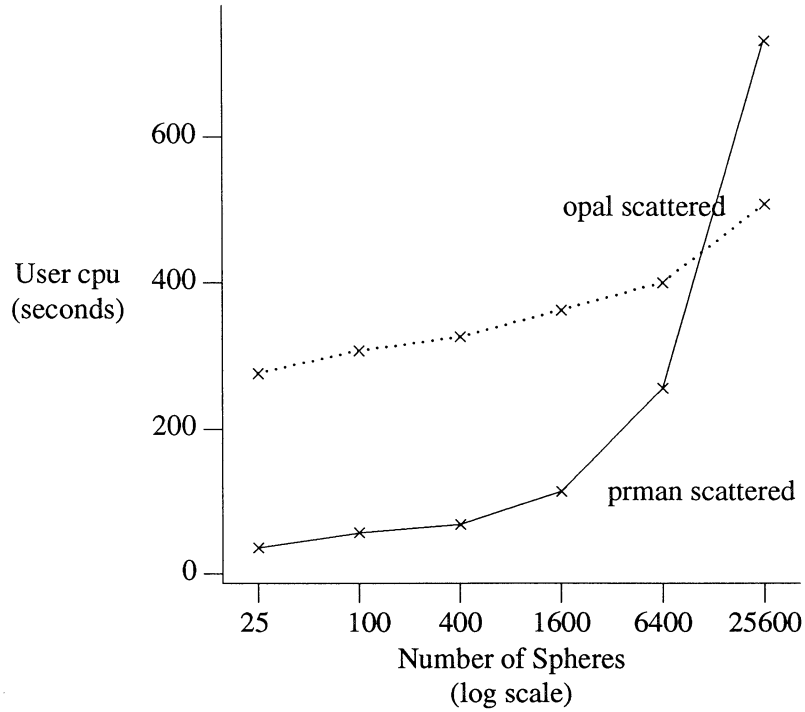


Figure 4.6. Scattered spheres. Varying the number of spheres, prman and opal.

previous section include reflection and transparency. To explore the compute time costs of global illumination effects, *sphere100* was rendered with two additional shading options. The resulting runtimes are shown in Table 4.17. The first line shows the time to render the image with only primary rays, using the same local illumination model as prman. The second line shows the time when we add reflection and transparency. The third line adds the cost of shadows in addition to reflection and transparency.

Omitting secondary rays entirely, opal is still much slower than prman (Table 4.14). Adding secondary rays for transparency and reflection nearly doubles the compute time at all of the tested resolutions. The shadow rays increase the cost by about the same amount.

Prman follows the Reyes rendering architecture [Cook87] by shading objects before performing the full visible surface algorithm. This approach incurs the cost of shading many surfaces that are not visible. However, it improves the coherence (or locality) of the shading and texturing problem by processing large pieces of a single surface at one time. In contrast, opal's ray tracing algorithm fails to preserve coherence in shading and texturing. This difference is illustrated by the texture access statistics reported by the renderers. Table 4.18 shows the number of texture pixels accessed by prman and by opal for *sphere100*, *landscape*, and *layers*. These three are the only images in the suite that contain textures.

The difference between the two rendering algorithms is striking. The contrast is most pronounced for sphere100, which presents the hardest problem for opal. Because of the reflections and refraction in the image, the texture map is accessed in response to secondary rays as well as primary rays. The secondary rays can be directed to any point in scene. Therefore, texture map pixels are accessed in a less coherent order.

Prman's coherence is due to its shading before determining visibility. This approach has advantages, as seen in Table 4.18, but it incurs the cost of shading invisible surfaces. The next section discusses this problem.

	User CPU (hh:mm:ss and normalized)							
resolution	**128**		**256**		**512**		**1024**	
primary	01:25	1.00	05:32	1.00	21:43	1.00	1:26:51	1.00
secondary	02:41	1.89	10:42	1.93	42:14	1.94	2:48:23	1.94
shadows	04:05	2.88	16:30	2.98	1:06:21	3.05	4:20:21	3.00

Table 4.17. Opal. User cpu time with different levels of ray tracing. Times are for *sphere100* rendered with primary rays only, with primary rays and secondary rays in the directions of reflection and transmittance, or with primary, secondary, and shadow rays. The row labeled "secondary" contains the same data as Table 4.15.

scene	**resolution**	**texture pixels (millions)**	
		prman	**opal**
landscape	640 by 480	79.2	135.1
landscape	1280 by 960	235.5	895.4
layers	640 by 480	1.9	8.5
layers	1280 by 960	7.2	33.8
sphere100	512 by 512	4.5	34.7
sphere100	1024 by 1024	14.0	138.4

Table 4.18. Texture access statistics.

4.9.3. Prman's Visible Surface Algorithm

Section 3.3.2 describes an approach called *invisibility processing* that prman uses to avoid shading invisible objects and to reduce the input to the full visible surface algorithm. Two experiments evaluated the success of this approach. The first experiment varied two scene characteristics, depth complexity and transparency. The images contained spheres in ten-by-ten grids. Up to eight grids appeared in the scene, stacked one behind the other. Each configuration was rendered twice, with transparent spheres and with opaque spheres. The commands used to generate the models were of the form

spheres –t *tpct* –ds 10 *depth*

where *tpct* was either zero or one, and *depth* varied from one to eight.

When all spheres are transparent, every surfaces is shaded and processed by the visible surface algorithm. But when the spheres are opaque, the grids closet to the camera obscure those further away. If the invisibility processing is successful, the renderer should be able to avoid processing these obscured surfaces. Table 4.19 gives the results of varying the depth of the scene and the opacity. It contains two statistics in addition to the runtime. The size of the shading problem is indicated by the number of micropolygons shaded by the renderer, and the size of the visible surface problem is indicated by the number of samples.

In the case of transparent spheres, the size of the shading and visible surface problems increase at about the same rate as the number of layers in the scene. In the case of opaque spheres, the work increases more slowly, because prman is eliminating a large proportion of the hidden surfaces. (With only one layer, the opaque spheres require less processing than the transparent spheres because the sides facing away from the camera are not visible.)

The scenes for the second experiment contained from one to ten terrain surfaces, composed of patches. The surfaces were stacked directly above each other and viewed from a position in front of and slightly above the objects. All of the surfaces were the same size in world space, but only the top layer was visible in its entirety. The screen sizes of the surfaces varied slightly because of the effects of the perspective projection. Two shading options were used. In one case, each surfaces was modeled as untextured plastic. In the other, each surface had three texture maps. The commands used to generate the models were of the form:

terrain –pb –v1 –n *nsurfaces*,
and
terrain –pb –v1 –n *nsurfaces* –t *nsurfaces*3

where *nsurfaces* varied from one to ten.

Opacity	Layers	User cpu		Samples		Micropolygons	
		m:ss	ratio	number	ratio	number	ratio
transparent	1	1:27	1.00	4,416,306	1.00	571,920	1.00
	2	2:44	1.89	8,518,543	1.93	1,099,640	1.92
	4	5:16	3.63	15,843,296	3.59	2,050,520	3.59
	8	9:42	6.69	27,818,344	6.30	3,623,384	6.34
opaque	1	1:08	1.00	2,846,954	1.00	480,576	1.00
	2	1:47	1.57	3,993,524	1.40	793,320	1.65
	4	2:48	2.47	5,606,529	1.97	1,272,274	2.65
	8	3:45	3.31	6,954,053	2.44	1,763,870	3.67

Table 4.19 Sphere depth experiment. Each layer of the scene contains a ten-by-ten grid of spheres. Scenes have one, two, four, or eight layers. Each configuration was rendered with opaque surfaces and with transparent surfaces. For each statistic, the table gives both the raw numbers and the problem size relative to the scene with one layer of spheres.

Figure 4.7 shows the effect of adding depth to the scene. It plots the number of non-empty pixels and the number of micropolygons. Table 4.20 contains the corresponding data and other statistics. The number of micropolygons increases only slightly faster than the number of visible pixels. This increase is less than the total screen area of the objects (both visible and obscured), which increase approximately linearly with the number of surfaces. In summary, prman is able to avoid many of the performance problems associated with shading before determining visibility.

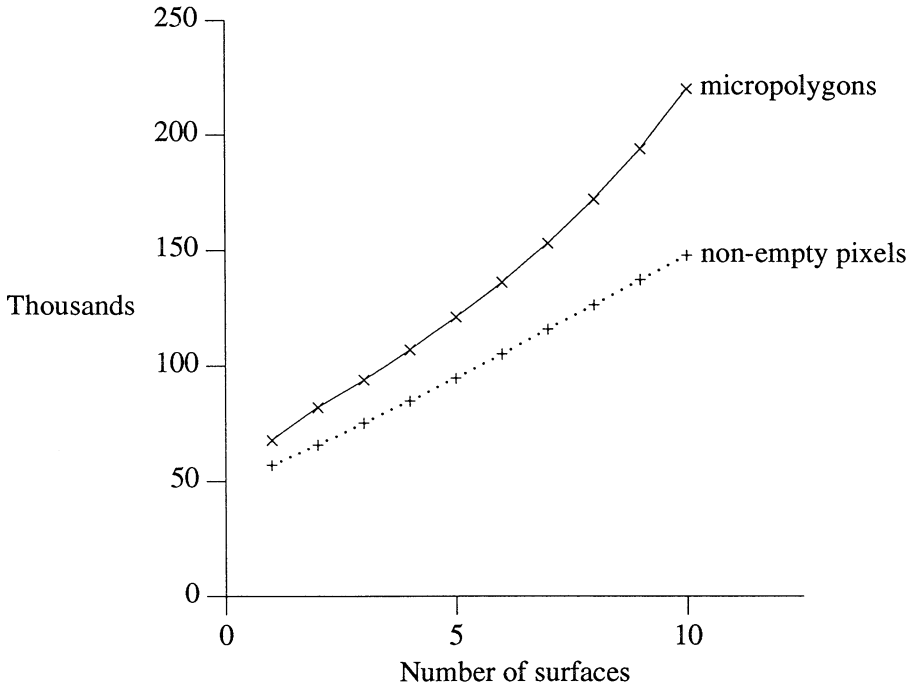


Figure 4.7. Terrain depth experiment. The number of micropolygons is plotted with a solid line, and the number of non-empty pixels (at a resolution of 640 by 480 pixels) is plotted with a dotted line.

Surfaces	Pixels ratio	User cpu (m:ss) solid	textured	Samples number	ratio	Micropolygons shaded number	ratio
1	1.00	0:12	0:21	68,100	1.00	645,361	1.00
2	1.16	0:15	0:25	81,541	1.20	765,978	1.19
3	1.32	0:17	0:29	93,703	1.38	886,332	1.37
4	1.49	0:20	0:33	107,063	1.57	1,017,412	1.58
5	1.66	0:22	0:37	120,869	1.77	1,156,433	1.79
6	1.84	0:25	0:42	135,780	1.99	1,307,791	2.03
7	2.03	0:28	0:49	153,014	2.25	1,474,801	2.29
8	2.22	0:30	0:53	171,851	2.52	1,652,867	2.56
9	2.40	0:34	1:00	193,831	2.85	1,846,140	2.86
10	2.59	0:37	1:08	219,745	3.23	2,049,638	3.18

Table 4.20 Terrain depth experiment. The number of non-empty pixels is expressed relative to the image with one surface. For visible surface samples and shaded micropolygons, the table gives both the count and the ratio.

4.9.4. Summary

The experiments described in this section used renderers with contrasting approaches and known performance differences. Using models generated by Mg, we could detect these differences and quantify their effect. For example, Figures 4.5 and 4.6 document clearly that increasing the number of objects in the scene has little effect on the ray tracer, but slows prman considerably, especially if there are more than a few thousand objects. This result is expected [Kaji88], although the extent of prman's per-object overhead is somewhat surprising. Figure 4.4 shows that the type of modeling primitive affects the ray tracer far more. This result is also expected. Ray tracing centers on the operation of intersecting rays with primitives; its cost varies with the primitive type and is especially high for parametric patches. On the other hand, prman subdivides all objects into an intermediate form and does relatively little work on the original primitives, so it hardly notices the differences between patches and spheres.

These experiments are successful in having documented these, and other, expected results. They validate Mg's ability both to detect the effects of varying the renderer's input and to distinguish between systems with different performance characteristics.

The experiments varied several types of parameters, including scene characteristics and rendering parameters. The scene geometry was varied to assess the impact of varying either the number of objects or the type of modeling primitive. In each case, one characteristic changed while others remained approximately constant. For instance, the aggregate screen area of the objects remained approximately the same although the number of objects increased by a factor of 1,024. The surface characteristics were varied to measure the success of prman's visible surface optimizations. Rendering parameters were changed to measure the effect of increasing the image resolution and to study the ray tracer's performance with different levels of ray tracing.

Some of the experiments used Mg's benchmark suite, while others required that special-purpose models be generated. When necessary, it was easy to vary and control the scene characteristics using Mg's scene generators.

4.10. Conclusions

This chapter addresses three problems in performance measurement for rendering systems. First, it develops a methodology and a framework for controlled rendering performance experiments. Second, it proposes an interface for portable test suites that is suitable for realistic image synthesis. Third, it demonstrates a performance measurement tool, Mg, that follows the methodology and implements the interface. Mg generates scenes for rendering performance tests, and the scene characteristics vary in a controlled manner.

Mg displays several advantages over less flexible benchmarks. It is adaptable and can be used to construct specific tests that produce more than single-figure-of-merit performance numbers. The test results give information that can be used to evaluate factors in the system's performance. Mg's approach also has disadvantages. Using such a tool requires sophistication. It presents the experimenter with many choices and variations, and it produces a potentially large amount of data, which must be carefully evaluated. Even though a fixed benchmark suite has been defined, Mg does not generate a complete, stand-alone, executable test. Instead, it provides only data files which must be given as input to the system being tested. Furthermore, the proper measurement methodology must be determined individually for each rendering system configuration. Therefore, a tool like this is most useful in the hands of an implementer or system designer who needs to understand how various factors affect the system's performance.

A major hypothesis in Mg's design was that test images do not need to have the appearance of "real" images in order to be useful. The test cases constructed for Mg have simple geometry, which allows the experimenter to understand more easily the effects of controlled variation. Experience with these models demonstrates that they can detect performance differences between systems and help answer

performance questions about a renderer's algorithms or implementation. With little effort, it is possible to to create controlled experiments that give useful information about the relationship between the renderer's input and its performance.

Mg is intended for systems that render realistic images. However, I have chosen to limit the complexity of the system to make it easier to implement and port. The interface also omits some features in order to be more general. Two additions, in particular, may be useful in the future. First, adding other types of bicubic patches and NURBS (non-uniform rational B-splines) would support a wider range of applications. Second, different types of texture maps, for instance environment maps, would make the shading interface more realistic. A major limitation in the interface's definition is the lack of support for animation. Thus, it cannot be used to evaluate motion blur algorithms or explore frame-to-frame coherence.

5
Workload Partitioning for a Multiprocessor Rendering System

"I believe that the most exciting aspect of ray tracing is that it lacks coherence. That is, it is easily parallelizable."
James T. Kajiya [Kaji88]

5.1. Introduction

Because photorealistic rendering requires a great deal of computation, there is a lot of interest in using multiprocessor systems to produce images more quickly. To use a multiprocessor effectively, we must first solve the workload partitioning problem: to divide the work into an efficient set of sub-tasks that can be distributed among the processing units. For the multiprocessor to perform efficiently, the workload partitioning algorithm must (1) distribute the work evenly among the processors, and (2) avoid introducing additional overhead in the way it subdivides the task. In other words, it is important to keep all processors busy, but they must also be kept productive.

This chapter examines the performance of a class of workload partitioning schemes, those based on a spatial subdivision of the image plane. Spatial subdivision is attractive, because it separates the rendering problem into localized sub-problems. In fact, many rendering algorithms have used spatial subdivision even on uniprocessors because it improves locality and lowers the effective size of the problem. As Chapter 3 notes, however, the rendering work is not distributed uniformly over the screen space. Therefore, a naive spatial subdivision strategy may do a poor job of balancing the workload among the available processing nodes.

This chapter describes a simple, low-overhead spatial subdivision scheme that balances the workload explicitly. Our approach is designed for rendering animated sequences, and it uses costs observed for one frame to predict costs for the next frame. A second algorithm divides the work among the processing nodes so that the estimated cost is approximately the same for all nodes. The cost estimate algorithm is implemented in a program called *adjust*, which has been used in production to compute two short animated films.

The studies presented in this chapter target a multiprocessor with distributed memory and sixteen processing nodes. Each node has a substantial amount of local memory, but there is no shared memory. The nodes do not have individual connections to disk, but they share a single connection to a host, which handles disk requests. The target system uses no special graphics hardware and implements all of its graphics functionality in software. Both the experiments and actual production experience demonstrate that the partitioning algorithm proposed in this chapter offers a noticeable improvement over the other schemes available on the system.

The performance studies assume a highly complex animation workload with large, detailed models and realistic shading and texturing. The workload requires careful antialiasing and medium to high resolution (512 to 4096 pixels per scanline). The experiments described in this chapter use scene descriptions that were taken unchanged from a full production workload. Because it takes many minutes, or even

hours, to render images of this quality on a uniprocessor, we are not attempting to achieve real or near-real time speeds.

We assume that the processing nodes will cooperate to render a single frame. We did not consider assigning a different frame to each node for two reasons. First, designing an animated film is a very labor-intensive problem, and an animator often needs to see individual frames as quickly as possible. Rendering separate frames in parallel increases the system's throughput but does not reduce the latency. Second, complex images often require large amounts of geometric and texture data. Because the shapes and colors of objects can change over time, we cannot assume that the scene specifications are the same for two frames of a sequence. The rendering system described in this chapter allows objects to change dynamically by instantiating a separate copy of the scene database for each frame. If the nodes computed sixteen frames in parallel, the aggregate working set for the system could be substantially larger.

The next section surveys previous work in spatial subdivision and load balancing for computer graphics. Section 5.3 describes the rendering system used as a base for the performance experiments. Sections 5.4 and 5.5 discuss spatial subdivision schemes and cost estimates for adaptive partitioning. Sections 5.6 and 5.7 present the experimental results and analyze the factors that contributed to the loss of multiprocessor efficiency. Section 5.8 describes our experience with using the new partitioning algorithm in production. Section 5.9 evaluates the hardware and software architectures and considers possible improvements, while Section 5.10 offers conclusions.

5.2. Literature Review

This section reviews previous research in workload partitioning and load balancing for rendering complex images and briefly considers two related applications. There are several ways to parallelize the rendering workload. With image-space parallelism, the screen is subdivided and a subset of the pixels is assigned to each processing node. With object-space parallelism, a region of the three-dimensional model space is assigned to each node. With object-per-processor parallelism, objects or geometric primitives are distributed to the nodes. Depending on the degree of parallelism and the complexity of the image, a node might be responsible for one or several objects. With functional parallelism, stages of the rendering process are assigned to separate nodes, often in a pipeline fashion. Different forms of parallelism may be combined in a single architecture. For example, multiple functional pipelines may render different objects or screen regions in parallel.

Some rendering systems have attained large-scale parallelism on SIMD (single instruction stream, multiple data stream) architectures. For example, the Pixel Planes project uses image-space parallelism with one special-purpose processor per pixel [Fuch85]. Other rendering systems have been implemented on general-purpose SIMD systems, such as the Connection Machine, using image-space subdivision and object-per-processor models [Crow89]. Large-scale SIMD architectures tend to require novel approach to rendering, in comparison with uniprocessor or MIMD implementations.

Rendering algorithms for MIMD (multiple instruction stream, multiple data stream) architectures more closely resemble those on uniprocessor systems. A very coarse granularity of parallelism may be obtained by rendering different images in parallel on a number of loosely-coupled processors. In this case, it may be possible to use uniprocessor software with little modification.

If we apply parallelism, instead, to the computation of a single image, we can improve the latency and often improve the locality of data references. First, we must partition the work of computing the image among the available processors. To utilize the processing power effectively, the work must be partitioned so that all of the nodes receive similar amounts of work. In *static* workload partitioning schemes, the partition remains the same throughout the computation of an image. In *dynamic* schemes, work assignments may change during the computation. A *fixed* scheme generates exactly the same partition for all images without considering the scene's characteristics, while an *adaptive* scheme tailors each partition to the workload's characteristics.

In summary, varied models of parallelism have been applied to computer graphics. Rather than survey the broad range of approaches, this chapter will focus on spatial subdivision and load balancing for MIMD architectures. For a broader view of parallel rendering algorithms, see the surveys by Crow [Crow88] or by Molnar and Fuchs [Moln90].

5.2.1. Spatial Subdivision for Uniprocessor Graphics

Image-space subdivision has long been used in uniprocessor rendering algorithms. Often its goal is to reduce the complexity of the problem, as demonstrated in the visible surface algorithms of Warnock [Warn69] and Franklin [Fran80]. In the Reyes rendering architecture, spatial subdivision has been used to conserve scratch memory in uniprocessor implementations [Cook87]. Because the system described in this chapter uses an implementation of Reyes, image-space subdivision is a natural partitioning strategy.

Ray tracing algorithms are executed in the three-dimensional object space, but sampling is initiated from the two-dimensional screen space. A primary ray is cast from a pixel into a pre-defined portion of the object space, searching for the nearest object to the image plane. Once a ray intersects an object, secondary rays are cast to shade the object. Certain rays simulate reflection and refraction, and others test to see if the surface is in shadow. With image-space rendering algorithms, it is often possible to partition the model database so that each node stores only the subset that it needs. The data distribution problem is inherently more difficult with a multiprocessor ray tracer because these secondary rays may need to access any part of the object space. The computation for any pixel may, therefore, access any part of the database. Spatial subdivision has been applied both in the two-dimensional image space and in the three-dimensional object space. (See, for example, Dippé and Swensen's uniprocessor results [Dipp84].) Multiprocessor ray tracers are discussed below in a separate sub-section.

5.2.2. Implicit Load Balancing with Spatial Subdivision

Screen subdivision is often tied to the management of the display memory, or frame buffer. In many cases, this requires a fixed set of relationships between the processing nodes and screen locations. There have been many attempts to structure the memory so that typical workloads are spread evenly among the associated processing elements. This is a form of *implicit* load balancing, which is not adapted to any specific workload or set of processing conditions. At one extreme, the Pixel Planes architecture [Fuch82] uses a processor-per-pixel model. Architectures that have fewer processors than pixels must assign a set of pixels to each processor. If each node processes a fixed contiguous region, the load will be unbalanced unless the image complexity is distributed evenly over the screen. In 1977, Fuchs proposed an interleaved pattern of pixels to balance the load implicitly [Fuch77]. His scheme divides the screen into small tiles (for example, 4 by 4 pixels) and gives each processor one pixel from every tile. Even when the image is concentrated in a small portion of the screen, this scheme is likely to balance the load. Interleaved pixel partitioning is used in the Pixel Machine [Potm89] and in the Silicon Graphics 4D/240GTX [Akel89].

5.2.3. Explicit Load Balancing

Explicit algorithms attempt to control the load balance directly by basing work assignments on the workload's characteristics or on the observed distribution of work. Parke [Park80] and Kaplan [Kapl79] simulated screen-based subdivision for multi-processor visible surface algorithms without doing any real implementation.

Whelan studied the load balancing problem for a proposed real-time, parallel animation system [Whel85]. His architecture subdivided the two-dimensional screen space among approximately sixteen processors. Through simulation he compared the performance of fixed partitioning schemes with one algorithm for explicit load balancing. His explicit load balancing scheme studied the scene's geometry to estimate the rendering effort for each pixel. It then used a recursive bisection algorithm to subdivide the screen into regions with approximately the same amount of predicted work. Whelan concluded that the

explicit method balanced the workload much better than the fixed methods, but it required too much overhead to be feasible for a real time system, which must render many images every second. The load balancing scheme presented in this chapter, while similar to Whelan's, is practical because of differences in the algorithms and in the workloads. The cost estimates that I will describe are simpler (though more approximate) than Whelan's, so the overhead is reduced. Even so, our system can more easily absorb the overhead because our images are much more complex and require about three orders of magnitude more rendering effort.

Crow experimented with distributed rendering on a collection of workstations. Each workstation rendered its own subset of objects, and one workstation combined the results to form a complete image [Crow86]. Crow's algorithm analyzed scene characteristics and attempted to balance the load among the workstations. In the final analysis, the overhead for partitioning, distributing the data, and compositing the image was considerable, and he gained only modest advantages from using multiple workstations.

5.2.4. Load Balancing for Multiprocessor Ray Tracers

There have been many projects to use multiprocessors for ray tracing. These efforts have been motivated by the compute-intensive nature of raytracing and by the independence of the rays. Many early designs used spatial subdivision but did not address the load balancing issue directly, for example [Clea86].

Other multiprocessor designs have addressed the problem of load balancing. Dippé and Swensen suggested dynamic load balancing based on a three-dimensional subdivision of object space for a proposed ray-tracing architecture [Dipp84]. There were open problems, however, in the implementation of their distributed load balancing mechanism. Priol and Bouatouch[Prio89] performed static load balancing by subsampling the image and assigning approximately equal numbers of rays to each node. On a sixteen-node hypercube, they reported a speedup of only 6.16 for a 128 by 128 pixel image. Boothe[Boot89] determined that a form of processor self-scheduling worked well for a shared memory system. His strategy was to distribute many small jobs to the nodes. However, he found many hard problems with implementing a dynamic scheduling strategy on a message-passing system. Because recursive ray tracing can send secondary rays to any part of world space, the workload partitioning problem for ray tracing differs from the image-space problem described in this chapter. The organization and distribution of data is an important issue not only for load balancing, but also for reducing the number of intersection calculations [Casp89, Gree90].

5.2.5. Workload Partitioning for Scientific Applications

There has been a wide range of research on partitioning scientific applications for multiprocessing. The partitioning schemes for our rendering system use a recursive bisection algorithm to generate partitions. Berger and Bokhari [Berg87] describe this strategy, which Baden [Bade87] applied successfully to an application in fluid dynamics. Baden's application has several characteristics in common with ours: the workload is distributed unevenly over a two dimensional grid, the computation is localized within the grid, and the distribution of complexity changes in time steps. Unlike ours, his application has an accurate and easily obtained cost function. Iqbal, Saltz, and Bokhari [Iqba86] and Baden and Kohn [Bade91] compare recursive bisection with other partitioning strategies.

5.2.6. Frame-to-frame Coherence and Video Compression

The algorithm presented in this chapter depends on similarities in consecutive frames to predict rendering costs. Algorithms to compress digital video, such as the MPEG standard [LeGa91], exploit frame-to-frame coherence for dramatic increases in compression. The industry standards for video compression were developed for applications that are feasible only with high rates of compression. Their success is due both to spatial coherence within individual frames (resulting in intra-frame compression)

and to temporal coherence from frame to frame (resulting in inter-frame compression). Le Gall [LeGa91] explains that the target applications "demand a very high compression not achievable with intraframe coding alone" and reports that inter-frame compression can add another factor of three in compression. The inter-frame compression algorithms include both predictive and interpolative techniques.

Video compression rates as high as 50:1 to 200:1 are possible, but such high rates of compression depend not only on coherence but also on the use of lossy techniques [Ang91]. With lossy algorithms, the reconstructed images are not identical to the originals, but they are subjectively similar to the human eye for image quality that is comparable to analog videotape. Ang, Ruetz, and Auld cite much lower compression rates for lossless methods, around 3:1. Although frame-to-frame coherence is not sufficient for very high compression rates, it still makes an important contribution in this application.

5.3. The Rendering Environment

This section describes the environment of our workload partitioning experiments: the parallel rendering hardware, the rendering software, and the workload. The rendering environment supports photorealistic rendering of complex scenes; it offers power and flexibility to the scene designer. Given the current state of technology, these goals are incompatible with real-time or near real-time image generation.

5.3.1. The RM-1 System Architecture

The RM-1 rendering accelerator was designed and constructed in 1987. It first supported animation production in 1988. The RM-1 is a VME-bus board that operates as an attached processor, and all of its graphics functionality is implemented in software. Each board has sixteen programmable processing nodes, consisting of an Inmos T800 transputer and four megabytes of local memory. The T800 cpu has floating point and runs at twenty megahertz. The memory cycle times vary from 80 nanoseconds to 120 nanoseconds. On the whole, the RM-1 has a high-performance design with more processors, more memory, and faster parts than typical transputer systems.

One transputer, the root, communicates directly with the host over the VME bus. The other processors communicate with the host indirectly through the root. Each transputer has four twenty-megabit-per-second links to other processors. At the start of any run, the communications topology can be reconfigured through a cross-point switch. However, a ternary tree configuration is almost always used. In this configuration, each processor has one link to its parent and up to three links to child nodes. The root uses its parent link to communicate with the host. Early experiments with the RM-1 showed that the ternary tree helped reduce disk access bottlenecks by shortening the average distance from the nodes to the host. All of our experiments were run with the ternary tree configuration.

Varied computers have hosted RM-1 boards, including Sun-3 and Sun-4 workstations, the Silicon Graphics 4D/70 graphics workstation, and the CCI Power 6/32, a general-purpose computer that is about five times as fast as a VAX 11/780. The role of the host is to provide centralized control and input/output services for the transputers. The host downloads graphics data and control options to the RM-1 and then initiates rendering. During the rendering process, servers running on the host handle requests to read texture data and output display information. The host does not necessarily have a local disk, and it may obtain code and texture data from servers over the Ethernet.

The system is optimized to generate highly complex images. Specifically, it is not organized for real-time display or user interaction. Single-frame compute times for our experiments varied from one to thirty-three *minutes* when rendered on sixteen processors. In contrast, real-time, or near real-time, applications require ten to thirty frames per *second*. Accordingly, the architecture has no video component or other display hardware. All display output is forwarded to the host computer, which can either transfer the image to a frame buffer or store it in a file.

5.3.2. The Software Architecture

The rendering software is an implementation of the Reyes rendering architecture described in Chapters 2 and 3 [Cook87], specifically the transitional system discussed in Section 3.4. It can render arbitrarily complex scenes and interprets a sophisticated programming language for user-defined lighting and shading algorithms. The algorithms support photo-realistic rendering, with features such as high-quality antialiasing, motion blur, and depth of field. The system adheres to the RenderMan interface for transmitting model and control information to the rendering program [Pixa89, Upst90]. The software is written in C and is downloaded from the host.

All sixteen transputers run the same rendering code. They work on a single image in parallel by subdividing the image space into disjoint, rectangular regions. The host downloads the model description and rendering control options to the RM-1 in a language that implements the RenderMan interface. Through this input stream, the host provides centralized control for the rendering process. If a node should need information about pixels outside of its region, it computes the information locally. Therefore, the nodes work independently, communicating only to transmit information to or from the host.

Because the RM-1 has no shared memory, model and control information is distributed to the local memory of each processor during a model initialization phase at the start of each frame. After the host transmits the data to the root processor, it is propagated to the other nodes by flooding the ternary tree. On each node, two processes handle the input stream. One process simply forwards the input to all of the node's children. The other process reads the input, bounds geometric primitives in screen space, and culls all objects that fall completely outside the node's assigned region. Much of the computation of the model initialization phase is inherently serial, and all nodes repeat many of the same calculations. The initialization time increases with the size of the input stream and with the depth of the tree.

The renderer conserves memory by dividing each region into small rectangles, called *buckets*, and computing pixel values for each bucket in turn. When a bucket is finished, the transputer discards temporary data structures for the bucket, sends its coordinates and pixel values to the display server on the host and starts work on the next bucket. The server can write the pixels either to the frame buffer or to a file.

The current Reyes implementation requires that the subdivision of the image space remain static for each frame. A node's local memory is too small to retain the entire model throughout the computation, and the system's designers did not want to tackle the very difficult problem of distributing data on the fly (see, for example, [Boot89]). Instead, the system requires that each processor know its assigned region in advance, so it can discard data that falls outside of the region. A processor need not, however, work on a single contiguous region. Indeed, the system already supports an option that subdivides the image into an interleaved pattern of small tiles. The region assigned to a node is not constrained to have any relationship to its parent's region. The system does not require that the entire screen be rendered, but to produce the correct image, the regions must cover all pixels that contain visible objects. The safest way to ensure this condition is to render the entire screen. However, it is possible to analyze the model file and bound the image more tightly.

In summary, the workload partitioning problem for the RM-1 accelerator with the Reyes rendering software has the following characteristics:

- The host provides centralized control.
- The screen space is subdivided into arbitrary, disjoint rectangular regions.
- The regions must completely cover the visible image.
- A static partition is specified for each frame.
- Any region may be assigned to any processor.
- No real-time constraints are imposed upon the system.

The design and implementation of the Reyes rendering architecture is unusual in several respects, and it is characterized by some uncommon performance considerations. It would be difficult to explain our measurements without first noting some of the system's performance characteristics:

The system encourages input-intensive approaches. Reyes relies heavily on texture mapping techniques, as described in Chapter 2, to add surface detail to objects and to simulate non-local illumination effects such as shadows, reflection, and refraction. The implementation is optimized to encourage the use of texture maps. A more compute-intensive alternative would be to simulate the optics of non-local illumination effects with an algorithm such as ray tracing. The texture-mapping approach can reduce the number of calculations required to determine an object's color, but it tends to generate many accesses to a number of large texture files.

Algorithms operate on small polygonal rendering primitives. The renderer subdivides geometric primitives adaptively to produce internal primitives called *micropolygons*. The largest dimension of a micropolygon is typically half the width of a pixel.

The renderer uses a lot of memory. Reyes uses about 350,000 bytes for code and initialized data. Even simple applications need another one and a half megabytes for data. Each geometric primitive is subdivided into many micropolygons, and the renderer keeps detailed information about each micropolygon until it has finished all buckets that contain the micropolygon. Furthermore, the renderer retains extensive information about all objects that intersect a pixel until the pixel is resolved.

Buckets are rendered in row-major order. Each node renders one bucket at a time, traversing its region from left to right and from top to bottom. Objects often fall into several neighboring buckets, and the data describing the object cannot be discarded until all of its buckets have been rendered. Consider an object that intersects two buckets, one above the other. If the node's region is a vertical strip, the renderer will move across the its region quickly and will soon reach the second bucket. If the region is a horizontal strip, the renderer will process many more buckets before reaching the second bucket. With horizontal regions, the renderer tends to retain information for a longer time, use more memory, and exhibit poorer locality of reference.

Caching texture pages improves performance. Typical workloads exhibit reasonably good locality of reference to the texture files. The performance of the RM-1 system is improved by caching texture pages at the nodes and at the server.

The cost of rendering an object is not distributed evenly over the area covered by the object. To increase the locality of the computations, the renderer processes entire objects, or large pieces of objects, at one time. An object is first subdivided into smaller geometric primitives and finally into meshes of micropolygons. Each primitive is bounded in screen space by a rectangular bounding box, and the cost of each level of subdivision is charged to the bucket that contains the upper left corner of the bounding box. Similarly, all of the micropolygons in a mesh are shaded at one time. No matter how many buckets the mesh intersects, the entire shading cost is charged to the bucket that contains the upper left corner of mesh.

Shading costs vary considerably. Shading and lighting algorithms are not fixed by the system, but are specified with programs written in the RenderMan shading language. Shading calculations can be arbitrarily complex, and as Chapter 3 showed, their costs can vary widely. Consequently, it is difficult to predict shading costs a priori.

5.3.3. Timing

The time required to render a frame on the RM-1 system can be expressed as the sum of three components:

$$T_{frame} = T_{model} + T_{render} + T_{display} \tag{5.1}$$

T_{model}, or the model initialization time, is the time to read the model and control information; T_{render}, or the rendering time, is the time to compute values for all of the pixels in the image; and $T_{display}$ is the time to display the pixels (or write them to a file). All of these values are measured as elapsed, or wall clock, times.

Each node reports the rendering time for its region. Since the only time reported by the transputers is the rendering time component, we will simplify the notation and let t_i denote the rendering time for node i. Timing starts after all nodes have read the model and are ready to render the image. Timing for each node stops after it has computed values for all of its pixels and has sent its display output to the host. (The host will not necessarily have received or displayed all of the completed pixels at the time that the node stops timing.) The rendering times reported by the node thus omit all of the time for model initialization and some of the display time. The time to render the entire image, which we shall call the RM-1 rendering time, is the time reported by the last transputer to finish:

$$T_{render} = \max(t_i),\ 1 \le i \le p$$

where p is the number of processing nodes.

The RM-1 nodes report only elapsed, or wallclock, times. Unfortunately, accurate idle times are unavailable. Direct measurement is difficult, because the transputer's hardwired scheduler does not allow for an idle process at the lowest priority. We could estimate idle time by sampling the program counter or profiling, but this would slow down the computation and alter the effects of multiprocessor contention. I have, instead, chosen to estimate idle time indirectly, by analyzing wait times for key events.

All of the host computers run the UNIX operating system and can report user cpu time, system cpu time, and elapsed time at the end of each run. Although the host acts as an intermediary between the transputers and the disk, the cpu time statistics show that it is mostly idle while the RM-1 is rendering. Let T_{host} be the elapsed time reported by the host. It includes not only the RM-1 rendering time, but also the time to initialize the model and display the frame. To approximate the sum of the model initialization and display overhead, we subtract the RM-1 rendering time from the host's elapsed time,

$$T_{model} + T_{display} = T_{host} - T_{render}$$

keeping in mind that part of $T_{display}$ overlaps with T_{render}. Whereas T_{render} is the time for the parallel rendering portion of the computation, the time $T_{model} + T_{display}$ describes tasks that are more serial in nature.

It is impossible to calculate T_{model} and $T_{display}$ individually from the available statistics. To understand these costs, I timed model initialization and display with a stopwatch, using messages on the terminal as cues to start and stop timing. I timed three trials each of the three models used in my performance experiments. In these tests, T_{model} ranged from sixty to seventy-five seconds. These observed initialization times were remarkably consistent, even though the length of the model files varied by about a factor of four. However, the animation group at Pixar has worked with models that have taken as long as four minutes per frame to initialize. Post-rendering display times are negligible when the image is displayed directly into the frame buffer. When the image is written to a disk file, the display overhead time depends on the resolution and contents of the image. It can be as little as thirty seconds, or as much as several minutes.

Table 5.1 summarizes the terms and notation defined in this section and in the rest of the chapter.

Rendering time components (Sections 5.3.3 and 5.4.1)

T_{frame}	time to compute a frame on the RM-1
T_{model}	model initialization time
T_{render}, T	rendering time (time to compute pixel values)
$T_{display}$	time to display pixels
T_{host}	elapsed time reported by the host
$T_{partition}$	partitioning overhead time
T'_{frame}	time to render a frame, including partitioning overhead

Node statistics (Section 5.5)

p	number of processing nodes
t_i	rendering time for node i
$x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$	coordinates of a node's region
$estimate_i$	cost estimate for node i

Metrics (Sections 5.6.3 and 5.7)

T_p	multiprocessor rendering time with p nodes
T_1	uniprocessor rendering time
S_p	multiprocessor speedup with p nodes
E_p	multiprocessor efficiency with p nodes
W_p	amount of work required by computation with p nodes
B_p	balance achieved with p processing nodes
L_p	lost efficiency for a multiprocessor with p nodes
L_p^b	loss due to imbalance
L_p^r	loss due to read delays
L_p^c	loss due to reduced coherence
E_p^b	idealized efficiency for a perfectly balanced load
E_p^r	idealized efficiency without read contention

Renderer (Sections 5.3.2 and 5.7.3)

bucket	a small rectangular region of pixels
micropolygon (μpoly)	a small internal rendering primitive
grid	a rectangular mesh of micropolygons

Table 5.1. Terms and notation.

5.4. Spatial Subdivision Schemes

The performance analysis in this chapter studies a set of spatial subdivision schemes for workload partitioning. The RM-1 system supports only static workload partitioning schemes, in which the partition remains fixed throughout the computation of a frame. This section describes the schemes and discusses the factors that affect their performance. It introduces the performance issues briefly, while the following sections flesh them out with measurements.

5.4.1. Qualities of Spatial Subdivision Schemes

Let's first consider the factors that influence the suitability and performance of spatial subdivision schemes:

1. *Load balancing*. A desirable partitioning scheme balances the load evenly among the available processors.
2. *Overhead*. An ideal subdivision scheme introduces no overhead to compute the workload partition and to distribute assignments to the processing nodes. We may avoid delays if the host can compute partitions for future frames while the accelerator is busy rendering. If the host would otherwise be idle, then the true time to process a frame remains T_{frame} as defined in Section 5.3.3, Equation 5.1. If the partition is not computed in parallel with rendering, then the true time to render the frame must include the partitioning overhead time, $T_{partition}$:

$$T'_{frame} = T_{frame} + T_{partition}$$

3. *Correctness*. The system must produce correct results, even when the workload is distributed among multiple processors. For photo-realistic rendering applications we are concerned with image quality, including antialiasing and filtering (see Chapter 2). The shade of a pixel is computed as a weighted average of the objects that affect the pixel. If we use a one pixel by one pixel filter for antialiasing, the final shade of any pixel is completely determined by the objects that intersect the pixel. For production-quality photorealistic rendering, however, Pixar insists on filters that are two pixels wide. This means that the final color assigned to one pixel depends on the contents of neighboring pixels. This decision affects multiprocessor rendering, because a node must be able to obtain information about pixels outside of its region in order to determine the correct shade for border pixels. In the current RM-1 system, each node computes the needed information about neighboring pixels rather than depending on results from other nodes.
4. *Coherence.* In computer graphics, *coherence* is the tendency for images to be locally similar. By preserving locality, it is possible to reduce the cost of many calculations and improve the locality of memory and disk references. An ideal partitioning scheme would preserve any coherence in the workload that the renderer is prepared to exploit. In particular, Reyes improves texture locality and shading efficiency by processing large pieces of a surface at one time. A partition that scatters pieces of a coherent surface among many nodes frustrates Reyes' shading optimizations. Furthermore, a partition that often separates adjacent pixels makes a two-by-two reconstruction filter more expensive.

   There is often a tradeoff between maintaining coherence and minimizing load imbalance. If the partitioning scheme is constrained to assign large, coherent regions to each processor, it will tend to balance the load less evenly than an algorithm that can assign disjoint pixels to processors. What if the partitioning scheme violates the property of coherence in order to spread the load more evenly? The processors will all keep busy, but they may have much more work to do. They may take even longer to complete the task than if they had shared a more uneven, but more coherent load.
5. *Task independence*. Ideally, the computations assigned to different nodes will be as localized and independent as possible. Dependence results in more communication delays or duplicated computation.

6. *Systems issues*. The partitioning strategy should be a good match for the system. The communications structure and other characteristics, such as caching, affect the success of a particular subdivision scheme.

5.4.2. Fixed Spatial Subdivision on the RM-1

To render in parallel on the RM-1, the system partitions the screen space into regions and assigns a region to each processing node. The user may either specify the regions explicitly or use one of the system's fixed subdivision schemes. The system software provides two categories of fixed subdivision schemes. The *window* schemes give each processor a single contiguous region, subdividing the screen so that all windows have the same dimensions. For a sixteen-processor configuration, the partition may consist of sixteen horizontal strips, sixteen vertical strips, or a four-by-four grid. The *bucket* schemes divide the screen into small tiles and assign each processor a scattered subset of the tiles. The default tile size is four by four pixels, but the dimensions may be changed. In the default bucket scheme with p processors, each processor renders every pth tile across the screen. If the number of processors evenly divides the number of tiles across the screen, this scheme generates vertical strips. Normally, the strips are thinner than in the vertical window scheme, and each processor has several strips in different parts of the screen. A variation of the bucket scheme avoids vertical strips by repeating a rectangular pattern of assignments across the screen.

The fixed window schemes maintain good coherence. However, they are unlikely to balance the load well unless the image complexity is distributed uniformly over the entire screen. For example, if the image is concentrated in half of the screen area, several windows are likely to be empty and the about half of the nodes will do most of the work. The bucket schemes tend to balance the load evenly, but they destroy area coherence by scattering parts of large objects (or of large, coherent regions) among several processors.

Would the fixed window schemes provide better load balancing if we could avoid giving any node an empty region? To test this idea, I simulated a simple variation of the fixed grid scheme. I determined a rectangular bounding box that contained all of the objects in the image and subdivided the area within the bounding box into a uniform four-by-four grid. The bounding box improved the performance of the fixed grid schemes somewhat (by about ten to twenty percent). Still, the results were less promising than some exploratory simulations of adaptive subdivision, and I shifted my attention to the adaptive schemes.

5.4.3. Cost Estimates for Adaptive Subdivision

The problem with the schemes described above is that none finds a middle ground with good balancing and also good coherence. Our hypothesis is that we can achieve this middle ground with static scheduling by subdividing the screen *adaptively*, taking into account the characteristics of the workload in order to balance the work more evenly. An adaptive scheme has two main requirements: a set of cost estimates and a partitioning algorithm. The cost estimates attempt to predict how much time it will take to render some small portion of the image. The unit of granularity might, perhaps, be a pixel or a bucket. We will assume that each cost estimate describes a "bucket," which may be as small as a single pixel or as large as several pixels in each dimension. The partitioning algorithm assigns a subset of the buckets to each node in such a way that the summed cost estimates are approximately the same for all nodes.

There are several possible ways to estimate rendering costs. Let us consider four leading candidates in detail.

1. *The geometric model*. Geometric complexity is a good predictor of run time for many rendering algorithms, assuming that shading costs are relatively uniform for the different surfaces in the image. We can estimate geometric complexity by processing the model description with the following algorithm:

```
for each object {
        bound the object in screen space
        for each bucket intersected by the bounding box {
                increment the cost estimate for the bucket
        }
}
```

One drawback to this approach is the difficulty of mapping high-level geometry to screen space quickly, yet accurately. It can be quite expensive to parse the entire model file. Bill Reeves implemented this algorithm and used it for load balancing during the production of the film *Tin Toy* (1988). Reeves found that he had to tune his cost estimate program for different models by weighting high-level objects according to a subjective estimate of their shading complexity and other factors [Reev89]. It was also necessary to subdivide objects finely to produce tighter bounding boxes. His program balances the load well when it is tuned for the scene, but performs poorly otherwise. Because it takes several minutes to process the model file, he generates cost estimates on a workstation that otherwise would not be used for rendering and effectively eliminates partitioning overhead on the rendering machine. The resulting processor assignments are copied across the local network to the rendering host. This unwieldy procedure and the significant computational overhead are very undesirable in a production environment. Reeves's approach is one of the partitioning schemes used in my experiments. (It is called the *model* scheme in the rest of this chapter.) We did not tune the cost estimates for the different models.

2. *Low-level geometry*. We could base cost estimates on low-level primitives, such as micropolygons, assuming again that the shading complexity is fairly uniform. But because of the very high cost of subdividing objects into micropolygons, low-level geometry is not feasible for *a priori* estimates. Could we, perhaps, use micropolygons to predict costs for future frames? With some modification to the display server and to the RM-1 rendering code, we could save the number of micropolygons per bucket as a byproduct of computing a frame. With this information, we could estimate the cost of the next frame of an animated sequence. The computational overhead would be low, but we would increase the message traffic between the RM-1 nodes and the host.

   I experimented with this approach. First, I rendered an image with an instrumented system that reported the number of micropolygons and the rendering time for each bucket. Then, using the number of micropolygons as each bucket's cost estimate, I subdivided the screen so as to balance the cost estimates among the processors. I rendered the image again with the resulting partition. The result was a modest improvement over the best fixed partitioning scheme: about a ten percent decrease in runtime and an increase from 0.53 to 0.59 in efficiency. This experiment was optimistic, because it predicted the costs of an image with knowledge gained by rendering the same image. When I tried to use the cost estimates to predict the cost of the next frame in the animated sequence, the performance was predictably worse because of frame-to-frame differences. The runtime decreased only about six percent, and the efficiency rose from 0.50 to only 0.53.

   The problem with this approach is the weak correlation between the number of micropolygons in a bucket and the milliseconds reported to render the bucket; the correlation coefficient for some frames was as low as 0.25. This contrasts with the stronger correlation between the number of micropolygons per *node* and the node's rendering time, which was about 0.8 for the same scenes. Perhaps the micropolygon count fails as a cost estimate because the shading complexity varies among different objects. I believe, however, that there is an underlying problem with the renderer's implementation: because the costs of subdividing and shading an object are not spread uniformly among the pixels covered by the object (see Section 5.3.2), it is hard to make accurate fine-grained cost estimates.

3. *Shading*. In the Reyes rendering architecture, shading costs vary widely and are hard to predict. It would be useful to factor the surface complexity into cost calculations, but it is very hard to estimate costs directly by examining the scene specifications.
4. *Time*. Obviously, the best predictor of run time would be the actual time required to render the bucket. Rendering time is an appealing estimate because it indirectly accounts for both surface complexity and geometric complexity. Of course, run time is only available *a posteriori*, so it is not a practical cost estimate for individual still images. But consider animation: given adequate frame-to-frame coherence, the rendering time for one frame may be a good predictor for the next frame. As a metric, rendering time accounts for both geometric complexity and shading costs. The algorithm I developed achieves good results with this approach.

5.4.4. Partitioning Algorithms

The second component of an adaptive subdivision scheme is the partitioning algorithm. We have used a recursive bisection, or median-cut, algorithm implemented in the program *median* by Pat Hanrahan [Hanr88]. The algorithm processes a set of cost estimates corresponding to small regions of the screen. With one cut it divides the screen into two regions of approximately equal cost. It recursively bisects each part of the screen until it has created the desired number of rectangular regions. An early application of recursive bisection to graphics was in Warnock's visible surface algorithm [Warn69]. More recently, Whelan experimented with recursive bisection for real-time animation [Whel85]. The algorithm has been used in scientific applications by Berger and Bokhari [Berg87] and by Baden [Bade87]. Recursive bisection is fast and does an acceptable job of balancing the cost estimates.

Normally, median will make both horizontal and vertical cuts as it subdivides the screen. Figure 5.1 illustrates a partition generated by a recursive bisection with cuts in both directions. Median can also be constrained to make all cuts in one direction, producing either horizontal or vertical strips. Strictly speaking, median is not limited to bisection. It will produce partitions for any number of processors, not just for powers of two.

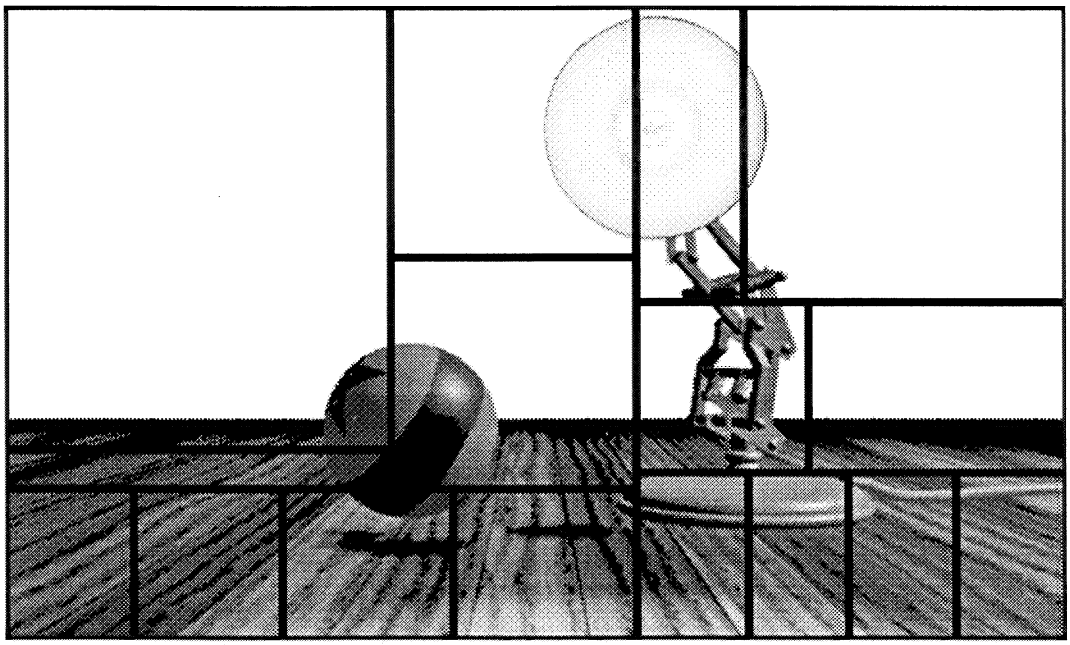


Figure 5.1. Partition generated by recursive bisection. The lines that represent the cuts are superimposed on the image.

5.5. Estimating Rendering Costs with the Program Adjust

The program *adjust* produces cost estimates, using the run time of one frame to predict the cost of the next frame. Its cost estimates are passed to median, which uses them to subdivide the screen. Together, adjust and median implement a workload partitioning algorithm. The host computer runs adjust and median after one rendering job completes and uses the results to make node assignments for the next frame. Adjust works on a sequence of frames. The first frame is rendered under one of the fixed window schemes: horizontal strips, vertical strips, or a grid. Starting with this *baseline* partition, the algorithm adjusts the boundaries of the windows after each frame to improve the balance.

The algorithm is very simple. Adjust requires two inputs from each node: the coordinates of its region (x_{min}, x_{max}, y_{min}, and y_{max}) and its rendering time in seconds (t_i). From the screen coordinates, it calculates the number of pixels rendered by each node. Adjust assigns the same cost estimate to all pixels rendered by the node. This estimate is simply the node's rendering time divided by the number of pixels rendered:

$$estimate_i = \frac{t_i}{(x_{max} - x_{min} + 1)(y_{max} - y_{min} + 1)}$$

In practice, the cost estimate is scaled in order to pass an integer to median.

In addition to the array of cost estimates, adjust's output includes the array's dimensions and the image resolution. The cost array need not have the same resolution as the image; median scales the cost array in each dimension as needed. This allows adjust to make an effective optimization: to view image partitioning as a one-dimensional problem. We can constrain the algorithm to divide the screen into vertical strips, making $p-1$ cuts along the horizontal axis, where p is the number of processing nodes. Because adjust assigns the same cost estimate for all pixels in a region, it makes no difference if it outputs one cost estimate per column or one for each pixel. Median scales the data appropriately and generates the same partition in either case. Alternatively, adjust can partition the screen into horizontal strips with one cost estimate per scanline. There is a vast reduction in the partitioning overhead with a one-dimensional cost array instead of a two-dimensional cost array (Table 5.2).

Why does adjust use such coarse-grained cost estimates? With just one estimate per node, adjust cannot pinpoint where the complexity is within a region. Despite the obvious drawbacks to the coarse granularity, there are problems with a finer granularity. Section 5.4.3 noted how difficult it is to produce accurate fine-grained estimates based on micropolygons. I also experimented with using the reported milliseconds per bucket as a cost estimate, with no better results. Again, we believe that the problem is that the rendering work is not spread evenly over the area covered by an object. If an object moves between frames, the rendering costs may shift to different buckets. Furthermore, it is more expensive to produce and process finer-grained cost estimates.

Baseline Scheme	adjust	median	adjust+median
vertical strips	0.139	0.107	0.246
horizontal strips	0.085	0.043	0.128
two-dimensional grid	45.510	25.521	71.031

Table 5.2. Partitioning overhead. All times are in user cpu seconds for partitions with sixteen regions on the slowest host, a Sun-3. Both programs, adjust and median, are written in C. The difference in run time for horizontal and vertical strips is explained by the aspect ratio of the images; there are 512 cost estimates in the x direction, but only 307 in the y direction. The times are averages for at least 29, and as many as 155, runs of each program.

Adjust's cost estimates are based on the hypothesis that one frame can predict the costs of the next frame. It can only succeed if there is sufficient frame-to-frame coherence in the workload. To test this hypothesis, I ran a series of performance experiments on the RM-1 system. The following sections describe the experiments and analyze the results.

5.6. Experimental Results

The purpose of the performance experiments was to evaluate spatial subdivision for parallel rendering and to compare the benefits of static and adaptive partitions. Specifically, I wanted to examine the feasibility of using frame-to-frame coherence to predict rendering costs. For each experiment, I tried different workload partitioning schemes and computed about ten frames of an animated sequence. I tested three distinct sequences using, in general, nine partitioning schemes. Table 5.3 describes the spatial subdivision schemes used in the performance experiments.

Four schemes used fixed spatial subdivisions. Three of these were fixed window schemes: horizontal strips, vertical strips, and a four-by-four grid. The fourth fixed subdivision was a bucket scheme which divided the screen into tiles of 8 by 8 pixels or 24 by 24 pixels; each of p nodes rendered every pth tile across the screen. Four adaptive subdivision schemes were also studied. Three of the adaptive schemes used adjust's cost estimates, but each produced regions with a different shape: horizontal strips, vertical strips, and unconstrained two-dimensional grids. The fourth adaptive scheme used cost estimates that Reeves's program derived from the model files; it produced two-dimensional grids. All of the adaptive schemes used Hanrahan's *median* program to generate partitions based on the cost estimates. Finally, to compute the multiprocessor speedup, each sequence was rendered on a single processing node.

Category	Description	Cost Estimate	Overhead
Fixed	single processor	none	negligible
Adaptive Subdivision	vertical strips	time for previous frame	< 1 second
	horizontal strips	time for previous frame	< 1 second
	two-dimensional grid	time for previous frame	1 minute
	model (2d)	geometry of current frame	3-5 minutes
Fixed Subdivision	vertical strips	none	negligible
	horizontal strips	none	negligible
	4-by-4 grid	none	negligible
	buckets	none	negligible

Table 5.3. Spatial subdivision schemes.

5.6.1. The Workload

The workload for the performance experiments consisted of three sequences taken from short animated films produced at Pixar. The experimental workload had all the complexity of a full production workload. Because modeling and animating such complex scenes requires an enormous human effort, I selected sequences that were already available in the required RenderMan format.[1] The workload was

[1] Still images from a benchmark suite, such as the scenes from Chapter 4's Mg, are inadequate to test my hypothesis that information obtained from computing one frame may be used to predict the performance of the following frame in an animated sequence. Furthermore, it would be very difficult to enhance still images with motion that is representative of real animation.

defined to include the same scene specifications and control options that were used in production runs. Certain control options can be tuned to improve rendering performance, but I accepted the production values as given and did not modify them. However, in order to reduce the compute time I had to make two changes to the experimental workload. First, I computed all images at a resolution of 512 by 307, instead of the production resolution of 1024 by 614. Second, I computed sequences of only ten or eleven frames (or less than half a second of film). Sequences from animated films tend to be much longer, as Table 5.4 shows.

I selected sequences that represent three different, but common, cases in animation rendering. Four factors characterize the differences among the sequences: screen coverage, complexity, uniformity, and motion. The sequences are described below, while Tables 5.5 and 5.6 provide further details. The simplified black and white images in Figure 5.2 document the motion in the sequences and indicate the contents of the scenes. The frames read from left to right and from top to bottom by rows.

Camera Move. This sequence of ten frames is derived from the film *Tin Toy* (1988). We see a toy through the cellophane window of a box, which rests on a wood-grained floor. A similar picture is reproduced in Figure 3.3(f). The image fills the entire frame, and the box covers a large part of the screen. Every part of the image has some texturing, but most of the geometric complexity is in the toy and its box. The scene is static, and the only motion comes from the camera, which rotates one degree per frame about the box.

Junior. This sequence of ten frames is from *Luxo Jr. in 3D* (1989), a short sequel to the film *Luxo Jr.* (1986). Models from this film are described in Table 3.3(g), and a similar image from a previous film is reproduced in Figure 3.3(d). This sequence also contains both a character and a static background, but here the camera remains fixed and the motion comes from the character, a drafting lamp that is playing with a rubber ball. The upper two-thirds of the background is empty, and the lower third shows a wooden floor. All told, the image covers about 45-50% of the screen, varying from frame to frame. Most of the geometric complexity is in the lamp, yet the complexity varies over the character. The floor has very simple geometry but complex texturing; separate texture maps simulate the wood grain, shadows, and reflections. The character's motion is confined to one part of the screen; the lamp's base is stationary while its upper body swings from a crouched position to an upright position.

Tinny. In this sequence of eleven frames from *Tin Toy*, a wind-up toy races across the screen along a diagonal path. By the end of the sequence, it has nearly disappeared off the bottom-left edge of the screen. This sequence illustrates a common rendering technique for character animation. The character ''element'' is computed alone and composited later with a background that is computed separately. This technique is cost-effective when the background is static or changes only in well-defined areas. The character covers approximately 10% of the screen, and the background is entirely empty. Table 3.3(f) describes the character element in this sequence. Figure 3.3(f) shows a complete frame, including both the character and the background.

Both the geometric complexity and the shading complexity are fairly uniform over the toy's surface. But since the image is concentrated in a small area, the complexity of the whole screen is far from uniform. Because of the rapid cross-screen motion, this sequence has the least frame-to-frame coherence and is the most challenging for our adaptive approach to workload partitioning.

Film	Total Sequences	Total Frames	Sequence Length (Frames)		
			min	max	mean
Luxo Jr.	6	2520	n.a.	n.a.	420
Red's Dream	28	6049	81	792	216
Tin Toy	57	6709	6	666	118
Luxo Jr. in 3D	1	640	640	640	640
KnickKnack	37	4461	36	318	121

Table 5.4. Length of continuous sequences. The table gives the total number of sequences and the total number of frames in Pixar's animated films. The mean sequence length is the number of frames divided by the number of sequences. Full data for *Luxo Jr.* are unavailable; the film could have been rendered as one continuous sequence but was broken arbitrarily into six sequences for more convenient filming and post-production. The data for the other films are from post-production statistics.

Sequence	Model File Size	Geometric Primitives	
		total	description
Camera Move	276,327 bytes	1354	217 cylinders, 352 hyperboloids, 113 spheres, 14 tori 348 bicubic patches, 310 bilinear patches
Junior	846,640 bytes	2005	12 cylinders, 36 hyperboloids, 24 spheres, 4 tori, 1800 bicubic patches, 129 bilinear patches
Tinny	378,444 bytes	1510	217 cylinders, 350 hyperboloids, 113 spheres, 14 tori, 545 bicubic patches, 271 bilinear patches

Table 5.5. Geometric complexity of the experimental workload. A different model file describes each frame. The table gives the median file length of all frames in the sequence.

Sequence	Texture Files	Texture Channels	Size
Camera Move	7	14	11.1 Mb
Junior	5	7	3.4 Mb
Tinny	2	5	4.4 Mb

Table 5.6. Texture data for the experimental workload. The table gives the number of texture files and the number of distinct texture channels, or components. For example, a single texture with red, green, and blue components is represented by one file with three channels. Shadow information is represented by a single-channel texture file for each light source. The texture size is the sum of the lengths of all texture files required by a scene.

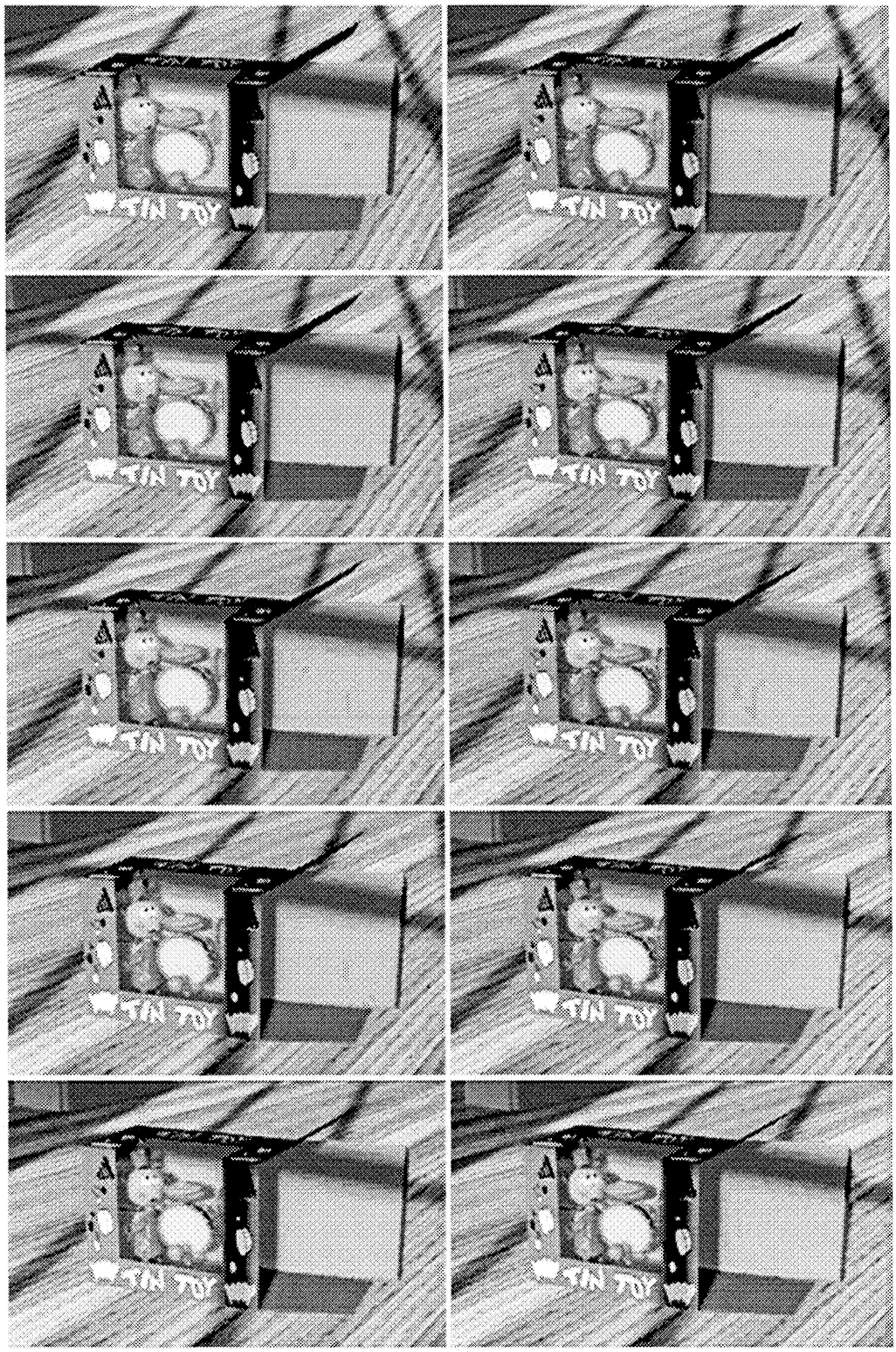


Figure 5.2(a). Motion in the Camera Move sequence. Frame-to-frame differences are most easily seen in the upper left corner and in the positions of the narrow shadows cast by a window frame.

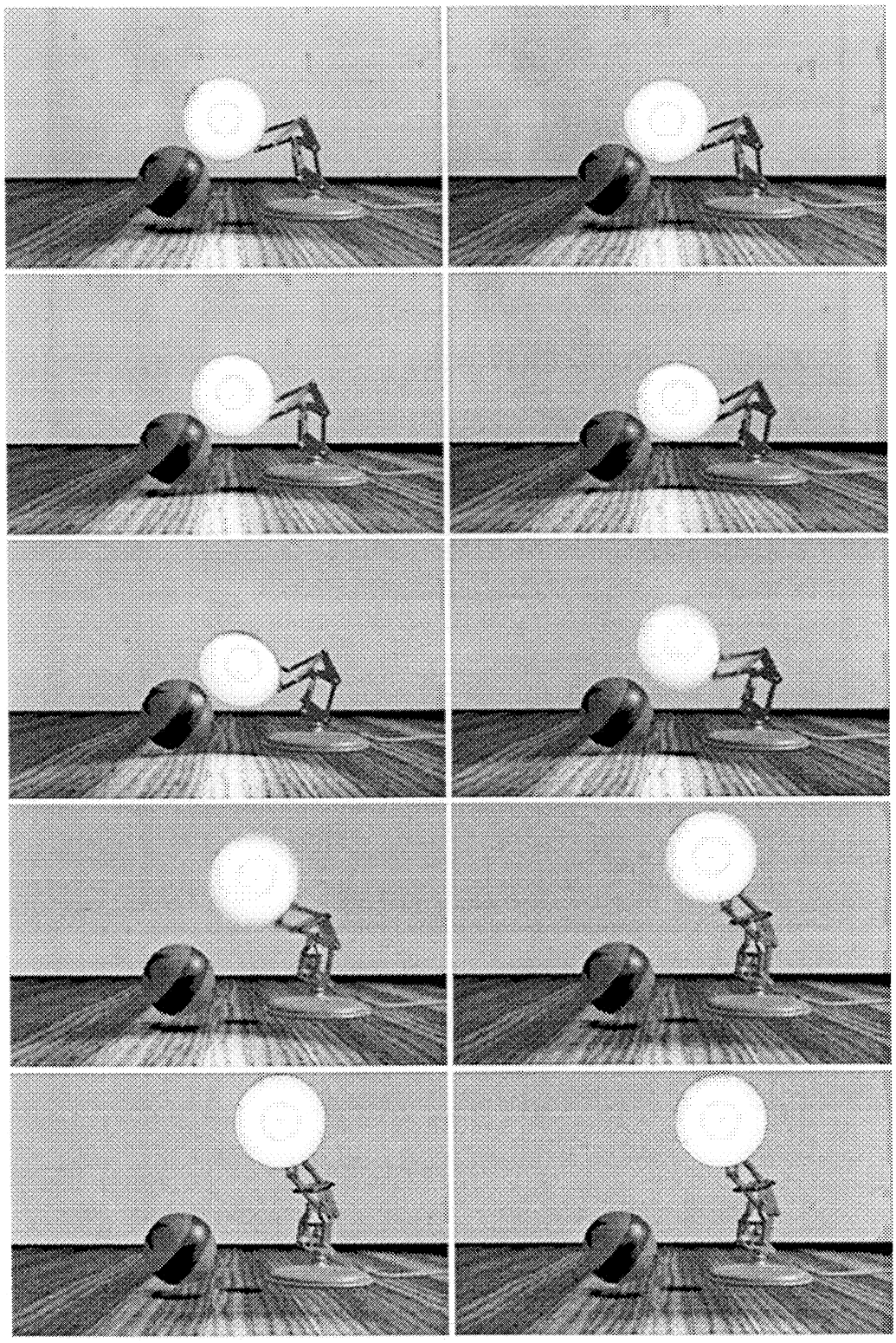


Figure 5.2(b). Motion in the Junior sequence.

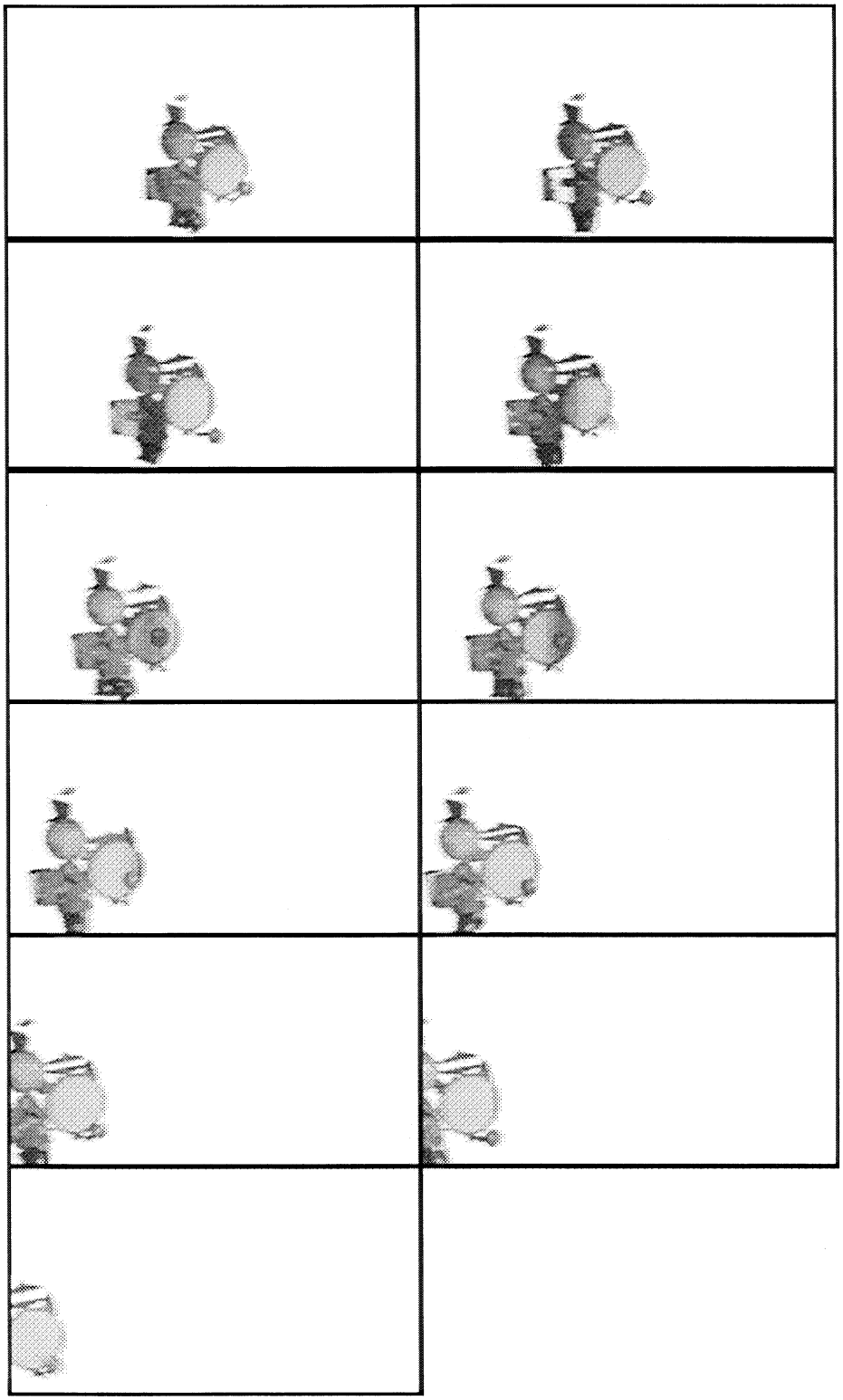


Figure 5.2(c). Motion in the Tinny sequence.

5.6.2. Processing Conditions

There was only enough dedicated machine time to run the experiments once, but the length of the computations compensate for the lack of repetitions. The quickest sequence ran in eighteen minutes, and the longest required nearly five hours of RM-1 time; most runs took one or two hours to complete. The measurements for individual frames within an experiment were consistent.

The RM-1 hardware and software are experimental and subject to frequent change. To ensure consistent performance measurements, I froze the computing environment and used the the same RM-1 board, host computer, and software releases for all runs. The host was a Sun–3 workstation that read the model file, texture data, and programs from its local disk. The RM-1 board was one of the fastest available, with a memory cycle time of 80 nanoseconds.

Using the full sixteen nodes on the RM-1 board, I rendered each of the three sequences under all of the partitioning schemes. Next, I varied the number of nodes using the best fixed scheme and the best adaptive scheme for each sequence. The tile size for the fixed bucket scheme was eight pixels square for Camera Move and Tinny, and twenty-four pixels square for Junior (as specified by the options used in production). All of the experiments were rendered with a one pixel by one pixel filter, and a subset of the experiments were run again with a two-by-two filter.[2]

5.6.3. Metrics

In Section 5.3.3 we defined T_{render} as the time to render an image on the RM-1. Because we will focus on the RM-1 rendering time as the basis for the performance analysis in this chapter, we will drop the subscript and refer to rendering time with a simple T. Assume that p is number of nodes rendering the image, and T_p is the multiprocessor rendering time with p processors. Then, T_1 is the uniprocessor rendering time. Kuck defines the multiprocessor *speedup* as [Kuck78]:

$$S_p = \frac{T_1}{T_p}$$

The multiprocessor speedup is a good intuitive metric, describing the multiprocessor's power as a multiple of the uniprocessor's power. However, speedup is a function of the number of processors. We can scale speedup to compute *multiprocessor efficiency*, a metric that is independent of the degree of parallelism [Kuck78]:

$$E_p = \frac{S_p}{p} \tag{5.2}$$

When $T_p < T_1$ for $p>1$, then $S_p \geq 1$ and $E_p \leq 1$.

Multiprocessor efficiency tells how well an algorithm performs, but it does not give enough information to explain its performance. Therefore, we will also examine other metrics. The amount of work required by the computation is the total compute time for all nodes, or

$$W_p = \sum_{i=1}^{p} t_i$$

This statistic allows us to compare how productively the processors are working under different partitioning schemes.

The processor utilization, or the proportion of available time during which processors are rendering, describes how well the workload is balanced. Let us define a *balance* metric, B_p, which reflects the processor utilization and reaches its maximum value of one when the workload is perfectly balanced:

---

[2] Specifically, the filters were a one-by-one box filter and a two-by-two Gaussian filter.

$$B_p = \frac{W_p}{pT_p} = \frac{\sum_{i=1}^{p} t_i}{pT_p} \leq 1$$

B_p is the same as the efficiency when the partitioned workload requires the same computational effort as the uniprocessor workload, that is, when $W_p = W_1 = T_1$. When partitioning the workload for $p > 1$ nodes increases the amount of work, then the balance metric is no longer equal to the multiprocessor efficiency. In this case, $W_p > T_1$ and, consequently,

$$B_p = \frac{W_p}{pT_p} > E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

5.6.4. Measurements

This section presents the measurements observed for the workload partitioning experiments. Table 5.7 describes the runs on a full, sixteen-processor configuration. The next subsection discusses the experiments that varied the number of processing nodes. Figure 5.3 compares the finish times graphically, while Table 5.8 shows the rendering time of the different partitions relative to the fastest scheme.

We will postpone a more detailed performance analysis until the following section and make just a few general observations here. The most important result is that the adjust algorithm outperformed fixed schemes in all of the test cases. For each sequence, the best fixed scheme took 30-48% longer than the best adaptive scheme. If we rule out the fixed buckets scheme, which is not used in production because of image quality considerations,[3] the best fixed window scheme took 30% to 84% longer than the best adaptive scheme. In only one case does a fixed window partition outperform one of adjust's partitions. This occurs with Tinny, when the best fixed window scheme finished more quickly than the slowest adjust scheme.

Which adaptive scheme is best depends on the workload. In general, RM-1 users have noticed that vertical strips are rendered faster than horizontal strips. One explanation for this performance difference is the row-major order for rendering buckets, as discussed in Section 5.3.2. Another explanation is that the images are wider than they are high (as is a movie screen). Because of the aspect ratio, horizontal strips are thinner than vertical strips and have a higher surface-to-volume ratio. Their boundaries tend to cut more objects, so horizontal strips are probably less coherent. Certainly, for the Camera Move sequence, we observe a large increase in the total work when we select horizontal strips. However, we can't discount the influence of the workload. Junior does well with vertical strips, either fixed or adaptive, because the image spreads across the screen. Horizontal strips are better for the Tinny sequence because it is characterized by horizontal motion. As the toy moves across the screen, it stays roughly within the same horizontal region. In our experience, it is fairly easy to choose a good baseline scheme intuitively by observing the orientation of the image and the direction of the motion.

The numbers in Table 5.7 tell an incomplete story–the rendering times do not include the partitioning overhead of the adaptive schemes. The overhead is negligible for the one-dimensional adjust schemes, both vertical and horizontal. For a sequence with n frames, we compute $n-1$ partitions. Assuming the average overhead costs in Table 5.2, the partitioning overhead would range from one and a quarter to two and a half seconds. At worst, the cost of running adjust and median in one dimension is about one-tenth of one percent of the time to render the sequence. Including this overhead in the rendering time would not change the efficiency statistics in Table 5.7. In contrast, the partitioning overhead for

[3] The bucket schemes support only one-by-one filters. Production quality images require a wider reconstruction filter, typically a Gaussian filter that is two pixels wide. It would be complicated, and more computationally expensive, to support wider filters under the bucket schemes.

Scheme	T_{16}	S_{16}	E_{16}	W_{16}	B_{16} overall	B_{16} min	B_{16} max
Single processor	75401.8	1.00	1.000	75401.8	1.00	1.00	1.00
*Adjust vertical	6494.9	11.61	0.726	91365.8	0.88	0.54	0.97
Adjust horizontal	9349.1	8.07	0.504	136602.3	0.91	0.72	0.95
Adjust 2d	6730.9	11.20	0.700	85664.0	0.80	0.41	0.97
Model	10543.3	7.15	0.447	84642.2	0.50	0.48	0.53
*Fixed vertical	9640.9	7.82	0.489	87538.9	0.57	0.54	0.60
Fixed horizontal	9690.1	7.78	0.486	110112.7	0.71	0.70	0.72
Fixed grid	13980.9	5.39	0.337	87641.6	0.39	0.39	0.41
Buckets	17703.6	4.26	0.266	261621.7	0.92	0.90	0.94

Scheme	T_{16}	S_{16}	E_{16}	W_{16}	B_{16} overall	B_{16} min	B_{16} max
Single processor	34833.2	1.00	1.000	34833.8	1.00	1.00	1.00
*Adjust vertical	3827.1	9.10	0.569	48503.6	0.79	0.69	0.93
Adjust horizontal	4316.7	8.07	0.504	50372.7	0.73	0.35	0.89
Adjust 2d	4324.4	8.06	0.503	44006.3	0.64	0.43	0.85
Model	6263.7	5.56	0.348	42643.1	0.43	0.40	0.49
*Fixed vertical	4982.3	6.99	0.437	47371.6	0.59	0.51	0.71
Fixed horizontal	7539.7	4.62	0.289	44280.0	0.37	0.35	0.41
Fixed grid	5801.6	6.00	0.375	40426.7	0.44	0.42	0.48
Buckets	5897.2	5.91	0.369	59156.5	0.63	0.57	0.71

Scheme	T_{16}	S_{16}	E_{16}	W_{16}	B_{16} overall	B_{16} min	B_{16} max
Single processor	8571.8	1.00	1.000	8571.8	1.00	1.00	1.00
Adjust vertical	2915.3	2.94	0.184	15306.8	0.33	0.18	0.69
*Adjust horizontal	1079.0	7.94	0.496	11731.1	0.68	0.30	0.87
Adjust 2d	1902.5	4.51	0.282	11146.9	0.37	0.16	0.54
Model	1145.6	7.48	0.468	11496.9	0.63	0.59	0.66
Fixed vertical	3715.8	2.31	0.144	10397.6	0.17	0.12	0.20
Fixed horizontal	1990.6	4.31	0.269	9934.7	0.31	0.27	0.36
Fixed grid	4218.0	2.03	0.127	9267.5	0.14	0.11	0.21
*Buckets	1438.5	5.96	0.372	19416.1	0.84	0.82	0.87

Table 5.7. Times and statistics. All times are reported as seconds of elapsed time, using 16 processors. The fastest adaptive scheme and the fastest fixed scheme are marked by asterisks. Three numbers are given for B_{16}. The first is the balance observed for the entire sequence. The second and third are the minimum and maximum values observed when B_{16} is calculated for each frame individually.

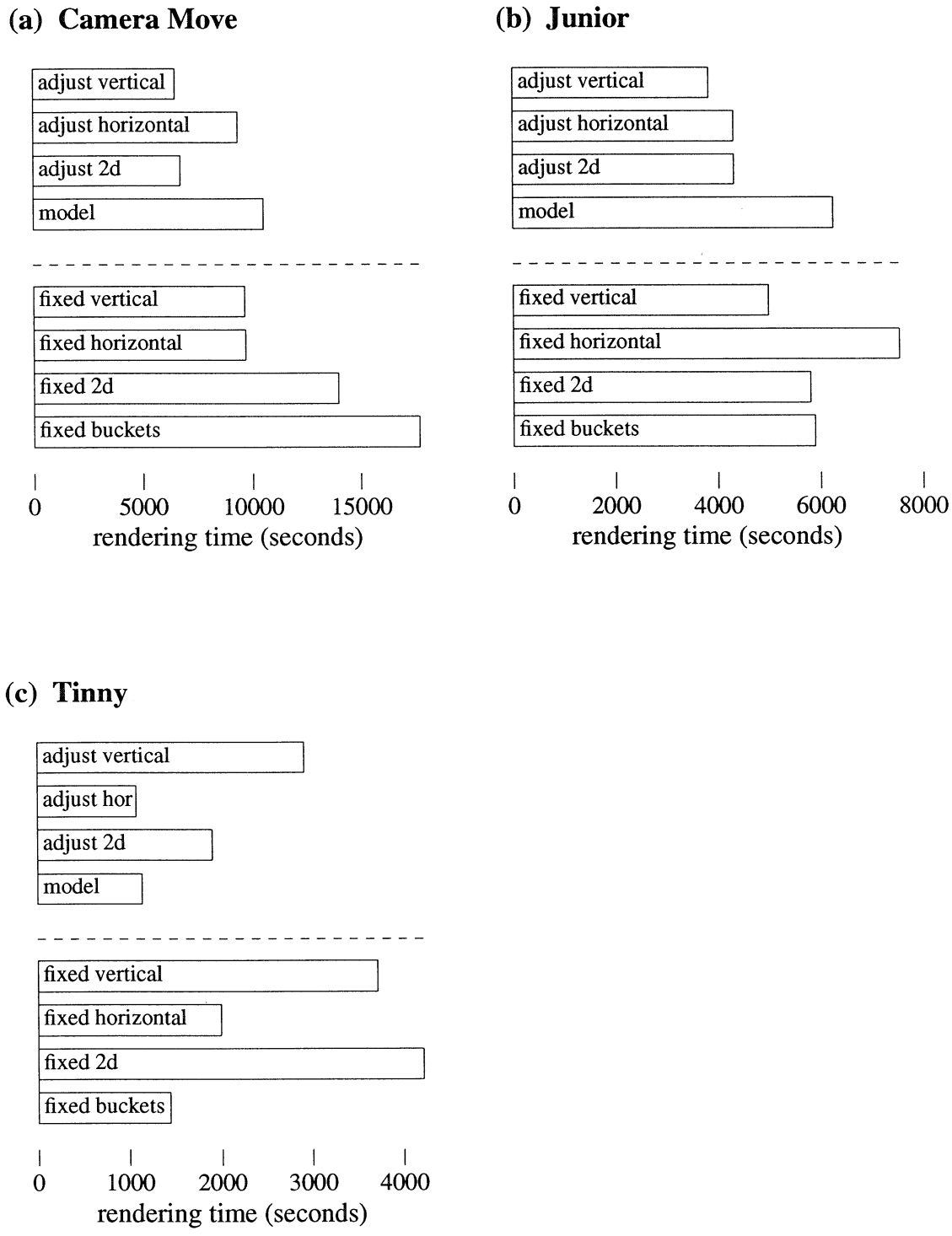


Figure 5.3. Rendering times for the adaptive partitions (above the dashed line) and fixed partitions (below the dashed line).

Scheme	Camera Move		Junior		Tinny	
	T_{16}	relative	T_{16}	relative	T_{16}	relative
Adjust vertical	6494.9	1.00	3827.1	1.00	2915.3	2.70
Adjust horizontal	9349.1	1.44	4316.7	1.13	1079.0	1.00
Adjust 2d	6730.9	1.04	4324.4	1.13	1902.5	1.76
Model	10543.3	1.62	6263.7	1.64	1145.6	1.06
Fixed vertical	9640.9	1.48	4982.3	1.30	3715.8	3.44
Fixed horizontal	9690.1	1.49	7539.7	1.97	1990.6	1.84
Fixed grid	13980.9	2.15	5801.6	1.52	4218.0	3.91
Buckets	17703.6	2.73	5897.2	1.54	1438.5	1.33

Table 5.8. Rendering time relative to the speed of the fastest partitioning scheme. The observed times are given in seconds.

Sequence	Rendering Only		With Partitioning Overhead	
	T_{16}	E_{16}	T'_{16}	E'_{16}
Camera Move	6730.9	0.700	7370.2	0.639
Junior	4324.4	0.503	4963.7	0.439
Tinny	1902.5	0.282	2612.8	0.205

Table 5.9. Partitioning overhead for Adjust 2d.

the two-dimensional adjust scheme is over ten minutes, that is, from nine to thirty-seven percent of the rendering time. Table 5.9 shows how this overhead affects the statistics. Even if we ignore the partitioning overhead, one-dimensional adjust was always faster than the two-dimensional version; only for Camera Move does the rendering time for an adjust 2d partition come close. Once we include the overhead, the two-dimensional schemes look even worse.

The two-dimensional adaptive schemes cost more because adjust and median process many more cost estimates. If we provided a smaller matrix of cost estimates, the overhead would be less, but would the load be as well balanced? It is possible to trade off the granularity of the cost estimates against the balance of the resulting partition, but I did not experiment with this trade-off. Because the two-dimensional scheme was never as fast for the experimental workload, even with fine-grained estimates, it seemed unimportant to tune the granularity of the cost estimates. For some other workloads, experimenting with the granularity of the cost estimates might yield performance improvements.

Initially, we wondered how often we would need to compute a new partition and rebalance the workload. But the very low overhead of the one-dimensional adjust schemes discouraged us from experimenting with the frequency of rebalancing. There is simply no reason not to compute a new partition for each frame.

The model scheme performed well only for Tinny, and, otherwise, it was comparable to the fixed schemes. The adjust schemes produce cost estimates by observing the previous frames. Only the model scheme uses data that describe the current frame. Unlike adjust, the model scheme is insensitive to frame-to-frame differences. Also, because it uses information about the current frame, it is the only scheme that can accurately eliminate empty pixels from the computation. However, the program that generates the cost estimates must be tuned for each model, and even then, it can yield disappointing results.

The overall performance of a partitioning scheme depends not only on how well it balances the workload but also on the additional amount of computational work that it introduces. The schemes based on two-dimensional grids (adjust 2d, model, and fixed grid) tend to introduce less additional work. Intuitively, the grid subdivisions preserve locality better because they maintain a lower surface-to-volume ratio in their regions and, therefore, cut fewer objects with the regions' edges. The scheme that generates the largest amount of rendering work is the fixed buckets scheme. Whatever it gains by balancing well, it loses by requiring a great deal of extra computation. Buckets outperformed fixed windows for only the Tinny sequence, which is very hard for the fixed window schemes to balance. The disappointing performance of fixed buckets illustrates the importance of maintaining coherence while subdividing the workload.

In summary, why a scheme performs well or poorly is a complex issue. The explanation depends both on the workload and on the partitioning scheme. A scheme that balances a given workload well may still perform poorly if it generates much extra work. On the other hand, a very efficient scheme may fail because its workload is too unbalanced. The following section will analyze the performance of the partitioning schemes in more detail.

5.6.5. Varying the Number of Processors

A second set of experiments varied the number of processing nodes from one to sixteen, using intermediate values of two, four, eight, and twelve. Table 5.10 provides a set of statistics. Figures 5.4, 5.5, and 5.6 show how rendering time and multiprocessor efficiency vary with the number of processors; the rendering times are plotted on a log scale.

There was not enough machine time available to test all of the partitioning schemes, so I selected the best fixed scheme and the best adaptive scheme for each sequence. These are the schemes marked by asterisks in Table 5.7, with one exception. For the Camera Move sequence, I noticed that fixed horizontal strips balanced better than fixed vertical strips. But vertical strips still outperformed horizontal because the horizontal scheme generated more extra work. I hypothesized that the increase in work was due to the row-major orientation of the renderer. To compensate, I assigned fixed horizontal strips and rotated the image ninety-degrees so that the strips were rendered as if they were vertical. Indeed, much of the extra work was eliminated. The rotated horizontal strips were rendered more quickly than any other fixed partition, with an observed multiprocessor efficiency of 0.560 for sixteen processors. Therefore, in varying the number of processing nodes, I used the rotated fixed horizontal scheme for Camera Move.

As expected, the multiprocessor efficiency drops as the number of nodes increases. It falls very quickly between one and four nodes for the fixed schemes. After four processors, the slopes for the fixed schemes become less steep and more similar to the slopes for the adjust schemes. The loss of efficiency is a result of two trends as we add more processors: the total amount of work tends to increase, and the work tends to become less balanced. The following section analyzes these trends and their underlying causes.

5.7. Analysis of Lost Efficiency

By definition, the multiprocessor efficiency, E_p, ranges from zero to one. Unfortunately, the maximum observed efficiency fell far below one for a sixteen-node RM-1 accelerator. In this section, we will examine the factors that contribute to the observed loss of efficiency.

Let us define the *lost efficiency*

$$L_p = 1 - E_p$$

for a multiprocessor with p nodes. In the RM-1 accelerator, we have identified three major causes of lost

Scheme	Nodes (p)	T_p	S_p	E_p	W_p	B_p overall	B_p min	B_p max
	1	75401.8	1.00	1.000	75401.8	1.00	1.00	1.00
Adjust vertical	2	39861.4	1.89	0.946	76551.6	0.96	0.82	1.00
	4	20700.1	3.64	0.911	76056.5	0.92	0.60	0.98
	8	11027.5	6.84	0.855	79018.7	0.90	0.56	0.98
	12	7925.2	9.51	0.793	84381.1	0.89	0.55	0.97
	16	6494.9	11.61	0.726	91365.8	0.88	0.54	0.97
Fixed horizontal	2	42405.7	1.78	0.889	80472.5	0.95	0.93	0.97
	4	26051.8	2.89	0.724	78464.6	0.75	0.75	0.76
	8	14195.9	5.31	0.664	79512.4	0.70	0.69	0.72
	12	10319.2	7.31	0.609	84512.4	0.68	0.68	0.69
	16	8412.2	8.96	0.560	90643.0	0.67	0.67	0.68

Scheme	Nodes (p)	T_p	S_p	E_p	W_p	B_p overall	B_p min	B_p max
	1	34833.2	1.00	1.000	34833.8	1.00	1.00	1.00
Adjust vertical	2	18138.0	1.92	0.960	35230.3	0.97	0.93	0.99
	4	9795.5	3.56	0.889	36214.0	0.92	0.82	0.98
	8	5449.9	6.39	0.799	38471.2	0.88	0.78	0.97
	12	4121.2	8.45	0.704	41957.9	0.85	0.75	0.95
	16	3827.1	9.10	0.569	48503.6	0.79	0.69	0.93
Fixed vertical	2	20152.5	1.73	0.864	35201.6	0.87	0.83	0.93
	4	12359.9	2.82	0.705	36109.1	0.73	0.67	0.82
	8	7082.0	4.92	0.615	38509.0	0.68	0.57	0.81
	12	5500.0	6.33	0.528	41564.4	0.63	0.53	0.76
	16	4982.3	6.99	0.437	47371.6	0.59	0.51	0.71

Scheme	Nodes (p)	T_p	S_p	E_p	W_p	B_p overall	B_p min	B_p max
	1	8571.8	1.00	1.000	8571.8	1.00	1.00	1.00
Adjust horizontal	2	5246.3	1.63	0.817	8781.7	0.84	0.51	0.99
	4	2813.7	3.05	0.762	9312.5	0.83	0.50	0.96
	8	1704.1	5.03	0.629	10097.8	0.74	0.37	0.93
	12	1263.3	6.79	0.565	10873.8	0.72	0.32	0.88
	16	1079.0	7.94	0.496	11731.1	0.68	0.30	0.87
Buckets	2	6527.1	1.31	0.657	12234.2	0.94	0.50	1.00
	4	4703.9	1.82	0.456	18399.2	0.98	0.95	0.99
	8	2704.7	3.17	0.396	19163.2	0.89	0.85	0.92
	12	2061.9	4.16	0.346	19295.6	0.78	0.73	0.84
	16	1438.5	5.96	0.372	19416.1	0.84	0.82	0.87

Table 5.10. Times and statistics, varying the number of processors. Times are reported in seconds of elapsed time. Three numbers are given for B_p. The first is the balance observed for the entire sequence. The second and third are the minimum and maximum values observed when B_p is calculated for each frame individually.

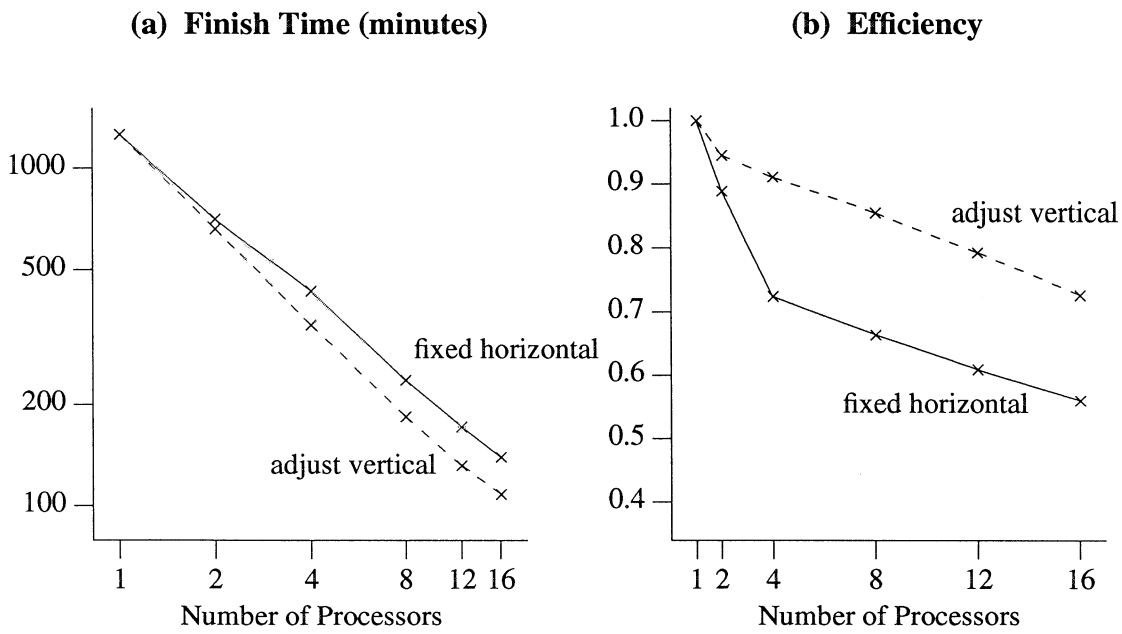


Figure 5.4. Camera Move. Increasing the number of processing nodes.

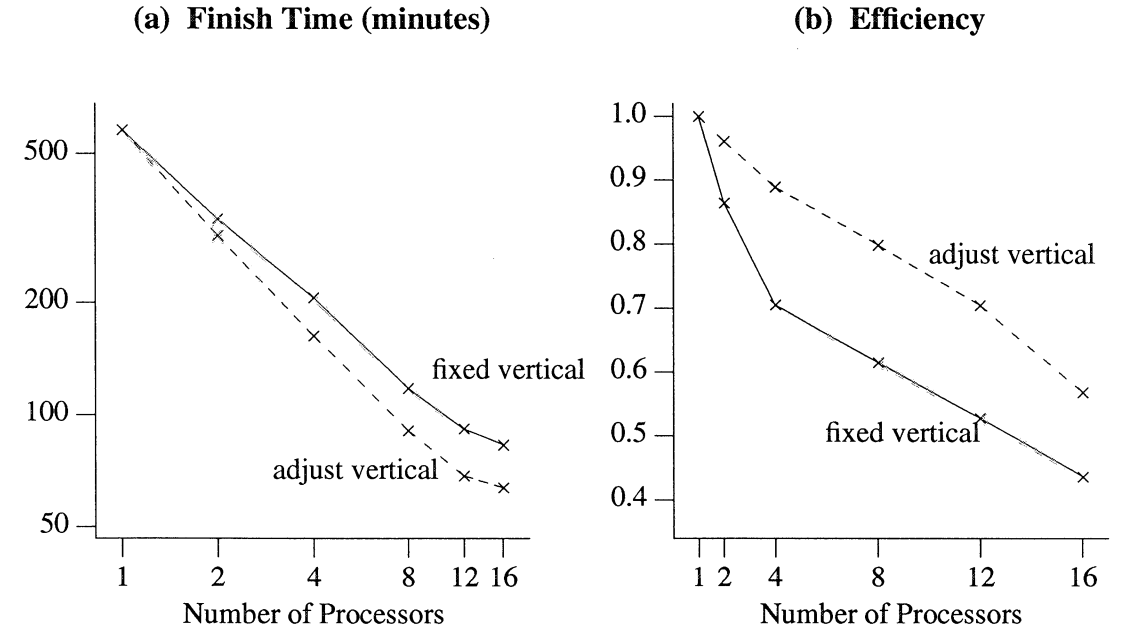


Figure 5.5. Junior. Increasing the number of processing nodes.

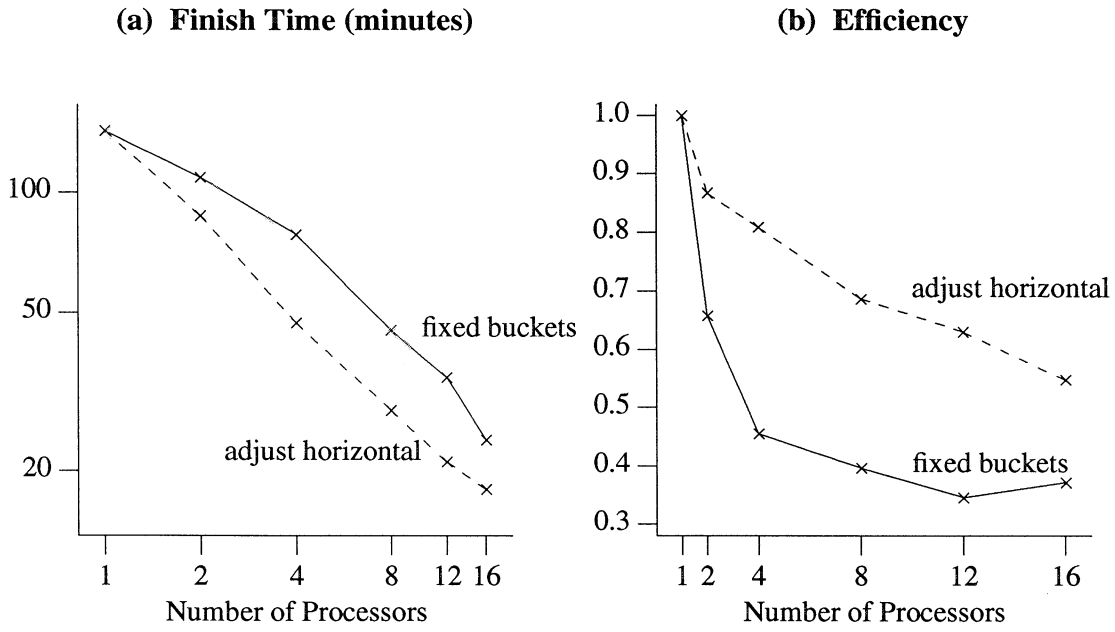


Figure 5.6. Tinny. Increasing the number of processing nodes.

rendering efficiency: imbalance, texture read delays, and reduced coherence. The total lost efficiency can be expressed as a sum of three terms:

$$L_p = L_p^b + L_p^r + L_p^c$$

where L_p^b is the loss due to imbalance, L_p^r is the loss due to read delays, and L_p^c is the loss due to reduced coherence.

In this section, we will estimate the impact of these three factors. The analysis is approximate, for two main reasons. First, not all of the effects that contribute to the loss of efficiency could be measured directly under the current RM-1 implementation, so we estimate them indirectly. Second, we could not control all of the factors that effect performance without changing the character of the system. For instance, the effects of texture caching vary for different partitioning schemes and different numbers of nodes, but to disable caching would alter the system's performance considerably.

The analysis in this section makes a major assumption, that texturing and the resulting input operations are an inherent part of the workload. Our goal, therefore, is to balance the mixture of computation and data accesses so that all nodes finish at approximately the same time. Adjust conforms to this view by using total elapsed time, and not compute time, as the basis for cost estimates. Evaluating the suitability of recursive bisection for balancing scientific workloads, Berger and Bokhari suggested that work estimates be based on a weighted combination of compute effort and communications costs [Berg87], which our approach does implicitly. As we noted above, the workload partition influences the number of texture accesses for each of the nodes. In this respect, our analysis is system dependent. Our analysis is also retrospective. Given the observed performance, we will attempt to explain where efficiency was lost.

After discussing individually the three sources of lost efficiency, we will consider the total picture for the experiments with sixteen processors. Finally we will see what happens as the number of processors varies.

5.7.1. Imbalance

To assess the impact of imbalance, we ask how much faster would the system render an image if the load were perfectly balanced? For this analysis, we assume that the total compute time, W_p, is fixed and that each node in a perfectly-balanced system would finish at time W_p/p. We substitute W_p/p for the observed rendering time, for T_p, in Equation 5.2 to calculate E_p^b, the idealized multiprocessor efficiency for a perfectly balanced load. The difference between the idealized efficiency and the observed efficiency is our estimate of the loss due to imbalance:

$$L_p^b = E_p^b - E_p$$

Table 5.11 shows the estimated loss due to imbalance for the three animated sequences.

Certain sequences are harder to balance than others, no matter which subdivision scheme we choose. These are the sequences with less uniform complexity, such as Junior and, especially, Tinny. In all cases, adjust improved considerably on the corresponding fixed window schemes. However, adjust still lost a noticeable amount of efficiency to imbalance on several occasions. Let's focus on adjust to see how it could improve its assignments. Three potential sources of imbalance are poorly-balanced initial frames, a sub-optimal partitioning algorithm, and inaccurate workload estimates.

Scheme	Camera Move L_p^b	Junior L_p^b	Tinny L_p^b
Adjust vertical	0.099	0.149	0.376
Adjust horizontal	0.048	0.188	0.235
Adjust 2d	0.180	0.289	0.487
Model	0.444	0.469	0.278
Fixed vertical	0.372	0.298	0.680
Fixed horizontal	0.199	0.498	0.594
Fixed grid	0.523	0.487	0.798
Buckets	0.022	0.220	0.069

Table 5.11. Lost efficiency due to imbalance.

The first possible source of imbalance is a poorly-balanced initial frame. Adjust computes the first frame of a sequence with one of the fixed window schemes and then tries to improve upon the original fixed partition. We have always observed a lower balance metric for the first frame than for the adjusted frames. Figure 5.7 illustrates how the balance tends to vary over a sequence. The curve for the Camera Move sequence has a typical shape. It rises sharply after the first frame and continues to rise more slowly until it hits a steady state. The balance maintains its steady state as long as there is good frame-to-frame coherence in the sequence. When motion weakens the frame-to-frame coherence, the curve dips.

Because we computed balance and efficiency as averages for the entire sequence, the influence of the slower first frame lowers the average. What happens if we omit this frame from the statistics, and compute E_p for the adjust frames only? As Table 5.12 shows, the improvement is slight. The Camera Move sequence loses the least efficiency to imbalance, but about half of this loss is accounted for by the influence of the slower first frame. The other sequences have more serious problems with imbalance, even after we compensate for the effect of the first frame. In these cases, we will have to look elsewhere for the causes of imbalance.

A second potential source of imbalance is the recursive bisection algorithm implemented by median. We cannot expect any partitioning algorithm to produce a perfectly-balanced load because of the constraints we place on the partition: it must contain one rectangle per processor, and the rectangles must cover the screen. Furthermore, recursive bisection is not an optimal algorithm, as Iqbal, Saltz, and

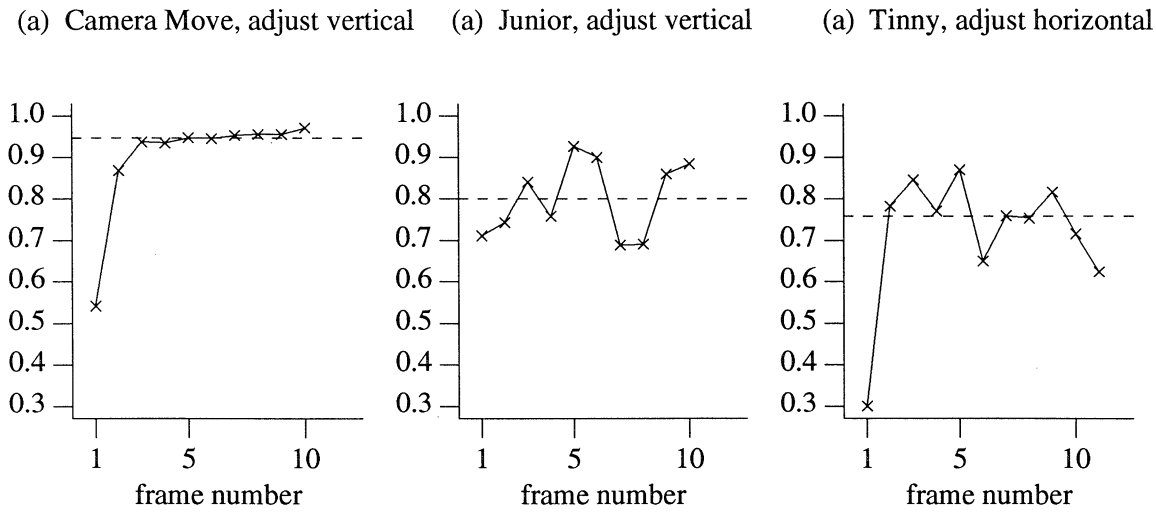


Figure 5.7. Balance improves from the start of the sequence. The balance, B_{16}, is plotted for the individual frames of each sequence. The dashed lines show the median value for each sequence.

Sequence	Adjust Scheme	Efficiency		Improvement
		all frames	adjusted frames	
Camera Move	vertical	0.726	0.772	0.046
	horizontal	0.504	0.507	0.003
	2d	0.700	0.794	0.094
Luxo	vertical	0.569	0.571	0.002
	horizontal	0.504	0.551	0.047
	2d	0.503	0.524	0.021
Tinny	vertical	0.184	0.188	0.004
	horizontal	0.496	0.547	0.051
	2d	0.282	0.312	0.030

Table 5.12. Loss of efficiency due to a less efficient fixed-window partition for the first frame of sequence.

Bokhari demonstrate with a counter-example [Iqba86]. To evaluate how well median balances, we can compute a variation of the balance statistic, B_{16}. Instead of using a node's runtime, we substitute median's cost estimates for the region. This statistic, the balance predicted by median, shows how well median has balanced the cost estimates. Table 5.13 compares the balance predicted by median with the observed balance.

The quality of median's workload assignments is indicated by the column of numbers to the left, the balance predicted by median. This is the machine utilization that would be achieved given accurate cost estimates, and it is generally within five to ten percent of perfect balance. By this standard, median produces acceptable partitions. Even an optimal solution would probably not be perfectly balanced, given our constraints on the solution. And without computing an optimal partition, it isn't possible to say how much better median could do.

The right-hand column of numbers in Table 5.13 points to the real problem–the potentially large gap between the balance predicted by the cost estimates and the actual balance observed while rendering the image. The use of inaccurate workload estimates is the remaining source of imbalance, and it does

Sequence	Adjust Scheme	Balance	
		Median's	Runtime
Camera Move	vertical	0.945	0.940
	horizontal	0.938	0.935
	2d	0.979	0.905
Luxo	vertical	0.960	0.810
	horizontal	0.871	0.818
	2d	0.973	0.676
Tinny	vertical	0.868	0.392
	horizontal	0.860	0.759
	2d	0.936	0.432

Table 5.13. Observed balance compared with balance predicted by median's cost estimates. The table shows the mean value of the balance statistic over the sequence, excluding the first frame of a sequence, which is rendered with a fixed subdivision scheme.

the most damage. We have discussed in principal two serious problems with adjust's algorithm. First, adjust assigns the same cost estimate to all of a node's pixels, even though costs are not normally distributed uniformly over the region. Second, adjust uses one frame to predict the performance of the next, even though the image usually changes between frames. As Table 5.13 shows, the predictions are more accurate when frame-to-frame coherence is stronger. On the whole, the predictions were most accurate for Camera Move which exhibited the most frame-to-frame coherence, and least accurate for Tinny which exhibited the least frame-to-frame coherence. The large discrepancies between the estimated and observed balance for Tinny can be explained by the character's motion. The direction of the motion is roughly horizontal, so the predictions for horizontal strips were more robust than the other predictions, which failed badly.

In summary, how can we improve the balance? Modest improvements can be made by computing longer sequences, and possibly by using a different partitioning algorithm. For example, Iqbal, Saltz, and Bokhari have proposed a greedy partitioning algorithm [Iqba86]. It produces partitions that are close to optimal for a class of one-dimensional problems that includes our one-dimensional spatial subdivision problem. However, significant performance gains require better cost estimates. Two important approaches to explore are increasing the granularity of cost estimates and predicting motion. We have already noted the problems with obtaining accurate fine-grained cost estimates in the system, and predicting motion may be a hard problem. Let's not forget adjust's big virtue–its simplicity. Any algorithm that takes much longer to compute cost estimates will have to be all that much more accurate to achieve the same overall performance.

5.7.2. Contention and Communications Delays

Efficiency also drops when contention and communications latency force a processor to wait. The major symptom of contention in the RM-1 system is the increase in the time to read texture pages. All messages to the texture server must go through the root node. In a single-processor configuration, only the root is active so the messages are transmitted along the shortest possible path. When additional processors are configured in a ternary tree, the average path length is increased. Furthermore, the messages contend for access to the single link to the host, and, possibly, to links lower in the tree. Conditions other than texture reads may cause a processor to wait, but, as shown by profiling, these conditions contribute little to the processor's idle time [Laws89]. Therefore, we will ignore these conditions in this analysis.

A set of experiments independently confirmed that texture read delays are the major symptom of multiprocessor contention in the system. Using partitions produced by adjust, I ran each of the sixteen regions serially on the root node. Lacking the machine time to render all three sequences this way, I

Scheme	Camera Move			Junior			Tinny		
	Faults	Avg. (ms)	Total (seconds)	Faults	Avg. (ms)	Total (seconds)	Faults	Avg. (ms)	Total (seconds)
Single Processor	91348	66	6038	24029	75	1809	93	57	5
Adjust vertical	40280	172	6935	34700	269	9343	355	68	24
Adjust horizontal	99788	494	49292	32842	254	8339	407	69	28
Adjust 2d	39619	194	7683	28734	245	7038	257	66	17
Model	40237	149	6012	27752	216	5981	423	64	27
Fixed vertical	36356	191	6950	32852	264	8663	190	67	13
Fixed horizontal	94428	279	26376	33272	177	5900	222	68	15
Fixed grid	59379	184	10935	28449	171	4876	144	69	10
Buckets	207520	539	111805	43676	339	14804	760	51	38

Table 5.14. Texture faults and read time statistics. The column labeled faults shows the number of texture page requests passed to the server. The average read time indicates the wait time (in milliseconds) while a single fault is serviced; the average is computed for all sixteen processors and all frames in the sequence. The total read time (in seconds) is the total time that all processors waited for texture read operations, summed over the entire sequence.

computed the three middle frames of each sequence using its fastest adjust scheme. Each serial run had the same intrinsic work and referencing patterns as the corresponding parallel run, so the difference between their total elapsed times came from communications delays and multiprocessor contention. The additional texture read time for the parallel runs explained about ninety-five percent of this difference in elapsed times.

Because idle time statistics are not available, texture access measurements are used to estimate the loss due to contention and communications delays. Table 5.14 describes the texture read delays. Texture pages are cached in the node's local memory, and a texture page fault generates a read request that is sent to the host's texture server. For a given sequence, the number of texture page faults varies with the partitioning scheme. When a partition preserves coherence, the renderer exhibits better locality of reference to the texture pages. Consequently, the fixed bucket partitions tend to generate many faults, and the schemes based on grids (adjust 2d, model, and fixed grid) tend to generate less.

The average read time is the mean time that a node is idle while waiting for a texture page. Many factors influence the read time. The time for a read tends to be higher when the load is well-balanced, because at any given time more nodes are actively contending for texture read services. Naturally, the read time also depends on the number of processors in the system and the hit rate for the server's disk cache. The above factors are primarily artifacts of multiprocessor contention and communications delays. However, the average read time also suffers from the loss of coherence, since it increases with the volume of requests.

In this analysis of the loss of efficiency, it would be useful to separate the effects of texture read contention from the effects of reduced coherence. Unfortunately, the two interact to drive up the cost of texture accesses. Instead, the analysis approximates the loss of efficiency due to texture read delays. This approximation, attributes the increase in the average read time solely to contention. On the other hand, it attributes the increased volume of texture faults to the loss of coherence. So, to estimate the loss of efficiency due to contention, the analysis uses a modified average read time, described below.

To estimate the loss of efficiency due to contention in the multiprocessor, we ask how much faster would the system render if texture reads were as fast as on a uniprocessor. How should we estimate this uniprocessor read time? The "Single Processor" read times in Table 5.14 were strongly affected by the texture server's disk cache hit rates. For example, the texture cache of a single node is small for rendering an entire Camera Move frame, and the node generates more faults. Consequently, the texture server

will find many of the requested pages in its disk cache and respond more quickly. A better estimate for a uniprocessor read time would be a read time for a partitioned workload, so I have used the average read time observed when the regions of a partition were rendered serially. These times are much more consistent than the average read times in Table 5.14: 76.7ms for Camera Move, 79.6ms for Junior, and 73.7ms for Tinny.[4]

The total read time for a processor is the product of the number of texture faults generated by the processor and its average read time. So, for this analysis, we calculate a new total read time for each processor by multiplying the number of its texture faults by the estimated average read time for a uniprocessor. A modified finish time for the processor, t'_i, is calculated from the observed t_i by subtracting the difference between the observed total read time and the new total. With the modified t'_i for each processor, we can calculate the idealized multiprocessor efficiency, E'_p. The approximate efficiency loss due to read contention is, then, the difference between the idealized efficiency and the observed efficiency:

$$L'_p = E'_p - E_p$$

Table 5.15 shows the approximate loss due to read delays for the three animated sequences. For the most part, the efficiency losses were low. Tinny required very few texture accesses and exhibited no loss of efficiency that is directly attributable to communications delays or contention for texture read services. Otherwise, read delays had the most noticeable influence on the partitioning schemes with a high volume of read requests. The efficiency loss can be low under a partitioning scheme that preserves locality, but it can be quite high under a less coherent scheme.

5.7.3. Reduced coherence

Partitioning the image tends to reduce coherence, because different parts of an object (or of coherent areas) may be assigned to different processors. Weakened coherence affects the system's performance in two ways. First, it reduces the locality of reference to texture pages and, therefore, increases the texture page fault traffic. Second, it increases the probability that two or more processors will have to perform the same calculations for a single object. After we have accounted for read delays and imbalance, the remaining cause of inefficiency is lost coherence. An estimate for the loss due to reduced coherence is derived by subtracting the losses due to imbalance and to read delays from the overall lost efficiency. Table 5.16 summarizes the contributions of the three major sources of lost efficiency, while Figure 5.8 presents the same information graphically.

In general, the horizontal schemes destroy more coherence than the vertical schemes, and the schemes based on two-dimensional grids (adjust 2d, model, and fixed grid) preserve coherence most successfully. The buckets scheme has the most disruptive effect on coherence.

Other measurements can confirm, and help explain, the effect of different partitioning schemes on coherence. As Section 5.3.2 explains, each node adaptively subdivides objects into meshes of small micropolygons. These meshes are called *grids*. Table 5.17 describes the rendering primitives generated for each sequence. The renderer streamlines and localizes the shading and texturing operations by shading an entire grid at one time, while the visible surface algorithm operates on individual micropolygons. Grids may straddle the boundary between two regions. In this case, each node shades the entire grid independently, making no attempt to clip the grid to the region's edge. If a region's dimensions are small, it is more likely that its edges will cut a grid and weaken the object coherence. The buckets scheme chops the screen into many small regions, causing nodes to generate many more grids and increasing the amount of duplicated computation. Two-dimensional grids preserve the most coherence, because their edges cut the fewest grids.

---

[4] The same estimate is used for each node. There was not a strong correlation between t_i and the depth of p_i in the ternary tree, except for a few frames where the workload was extremely well balanced. Typically, the strongest influence on an individual node's average read time was the number of nodes actively competing with it for resources.

Scheme	Camera Move		Junior		Tinny	
	modified total	L^r	modified total	L^r	modified total	L^r
Adjust vertical	3090	0.021	2762	0.089	26	0
Adjust horizontal	7655	0.167	2614	0.049	30	0
Adjust 2d	3039	0.029	2287	0.063	19	0
Model	3086	0.014	2209	0.008	31	0
Fixed vertical	2789	0.013	2615	0.096	14	0
Fixed horizontal	7243	0.070	2648	0.005	16	0
Fixed grid	4555	0.012	2264	0.017	11	0
Buckets	15918	0.181	3476	0.128	56	0

Table 5.15. Lost efficiency due to texture read contention.

Scheme	Camera Move			Junior			Tinny		
	L^b	L^r	L^c	L^b	L^r	L^c	L^b	L^r	L^c
Adjust vertical	0.099	0.019	0.156	0.149	0.087	0.195	0.376	0.000	0.440
Adjust horizontal	0.048	0.164	0.284	0.188	0.047	0.261	0.235	0.000	0.269
Adjust 2d	0.180	0.027	0.093	0.289	0.060	0.148	0.487	0.000	0.231
Model	0.444	0.012	0.097	0.469	0.008	0.175	0.278	0.000	0.254
Fixed vertical	0.372	0.011	0.128	0.298	0.093	0.172	0.680	0.000	0.176
Fixed horizontal	0.199	0.064	0.251	0.498	0.005	0.208	0.594	0.000	0.137
Fixed grid	0.523	0.010	0.130	0.487	0.016	0.122	0.798	0.000	0.075
Buckets	0.022	0.174	0.538	0.220	0.125	0.286	0.069	0.000	0.559

Table 5.16. Sources of lost efficiency. L^b is the estimated loss due to imbalance. L^r is the estimated loss due to read delays. The remaining loss, L^c, is attributed to reduced coherence.

Scheme	Camera Move			Junior			Tinny		
	µpoly	grids	size	µpoly	grids	size	µpoly	grids	size
Single processor	5214381	417556	12.9	2386192	184097	13.7	417251	62562	7.1
Adjust vertical	5467267	529578	13.0	2534551	221032	13.7	864562	149932	7.5
Adjust horizontal	5430446	538440	13.1	2526605	236914	13.8	501359	93939	8.0
Adjust 2d	5320462	474103	13.0	2454256	205513	13.7	550651	92637	7.5
Model	5326266	480635	12.9	2465791	205764	13.6	1197509	201689	7.7
Fixed vertical	5431836	506014	13.0	2499309	215800	13.8	550318	87396	7.4
Fixed horizontal	5402912	518435	13.1	2512337	217434	13.7	455088	76510	7.6
Fixed grid	5305701	465745	12.9	2437223	198384	13.6	450616	70348	7.3
Buckets	13692455	1036877	13.4	3361616	255920	13.8	1515146	201910	8.0

Table 5.17. Rendering primitives. The table shows the number of micropolygons (µpoly) and grids generated for each sequence and the mean size (in micropolygons) of grids.

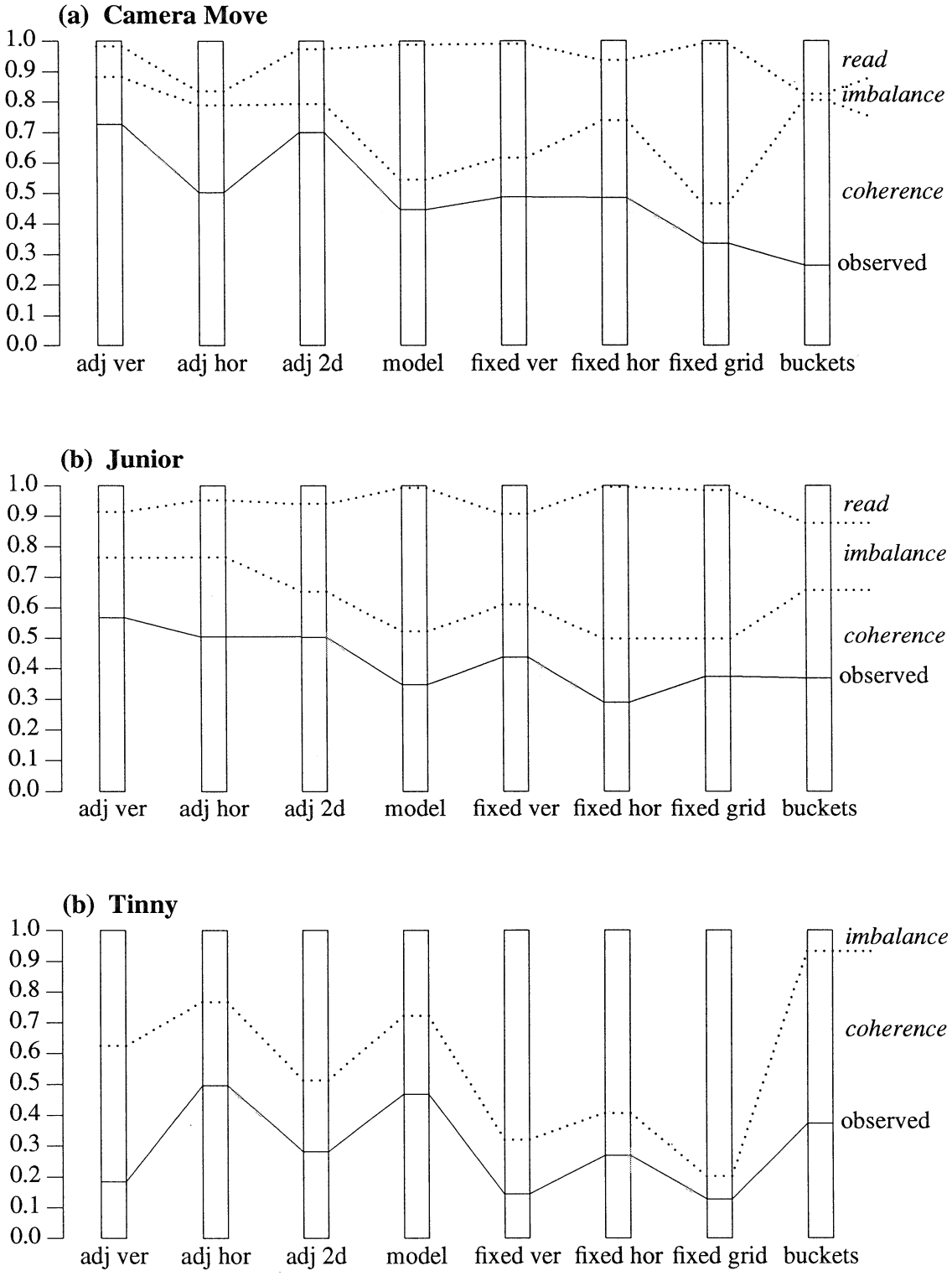


Figure 5.8. Lost efficiency. Solid horizontal lines indicate the observed efficiency. The region above the solid line represents the lost efficiency. The dotted lines divide this region into three components. From top to bottom they are: the loss due to texture read delays, the loss due to imbalance, and the loss due to reduced coherence. Tinny experienced no loss due to read delays.

5.7.4. Varying the Number of Processors

Figures 5.9, 5.10, and 5.11 describe the loss of efficiency observed when the number of processing nodes was varied. In these graphs, the solid horizontal lines bracket the observed efficiency and 100% efficiency. The region between the solid lines represents the lost efficiency. The dotted lines divide this region into three components. From top to bottom they are: the loss due to texture read delays, the loss due to imbalance, and the loss due to reduced coherence. Tinny experienced no loss due to read delays.

The major sources of inefficiency are imbalance and reduced coherence. Read delays make a relatively small contribution to lost efficiency (but they start to cause problems for Junior after twelve processors). In general, the effect of imbalance is felt quickly and holds steady after four processors. As the number of processors increases beyond four, the major source of increased inefficiency is reduced coherence. Intuitively, we expect coherence to suffer as we chop the screen into smaller pieces. More specifically, as we increase the number of nodes, regions become smaller. However, grids stay about the same size (measured in pixels). Therefore, the edges are likely to intersect more grids.

In summary, to apply spatial subdivision effectively to rendering on a multiprocessor, we must find algorithms that preserve as much coherence as possible.

5.7.5. Inter-processor dependence

In all of the experiments discussed above, the regions were completely independent. That is, no part of the computation depended on information about a pixel in another region, and the final shade of a pixel was completely determined by the objects that intersected the pixel. This was possible because a one pixel by one pixel reconstruction filter was used for antialiasing in those experiments. For production-quality photorealistic rendering, Pixar chooses a two by two filter; that is, the final shade of a pixel is influenced by objects in a two pixel by two pixel area. Thus, the shade of a pixel depends on the contents of the pixels that surround it. If a pixel is on the border of a region, its value will, therefore, depend on information about pixels assigned to other nodes.

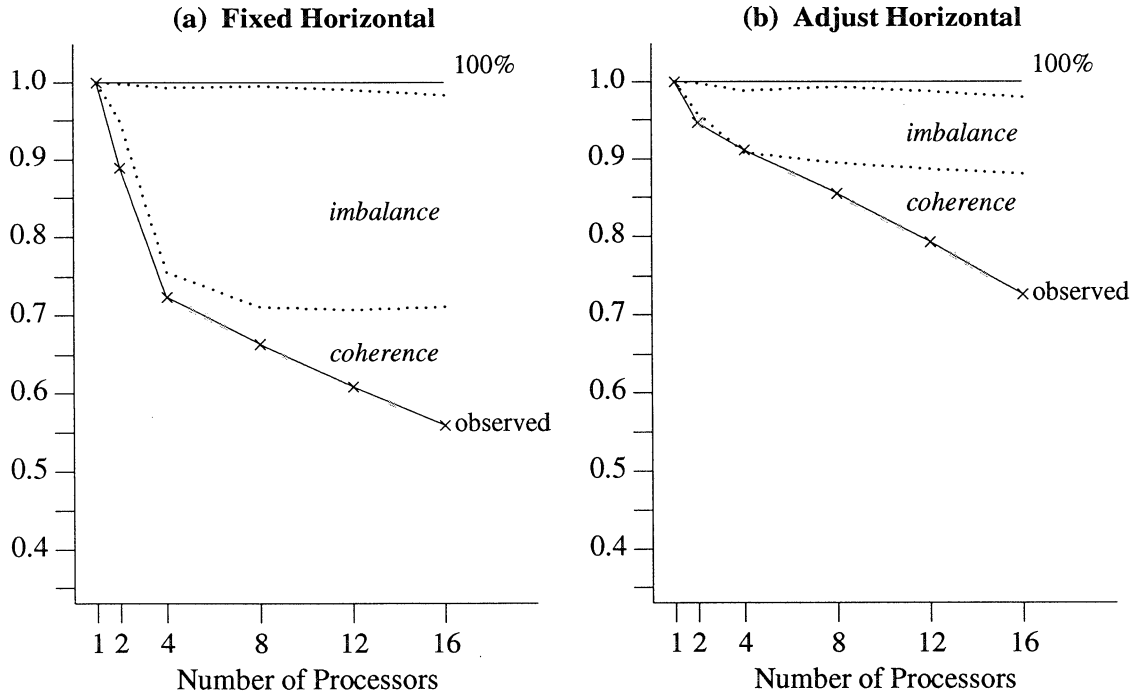


Figure 5.9. Camera Move. Loss of efficiency with a varying number of processors. In these graphs, the small unlabeled area below the solid line at 100% represents the loss of efficiency due to read contention.

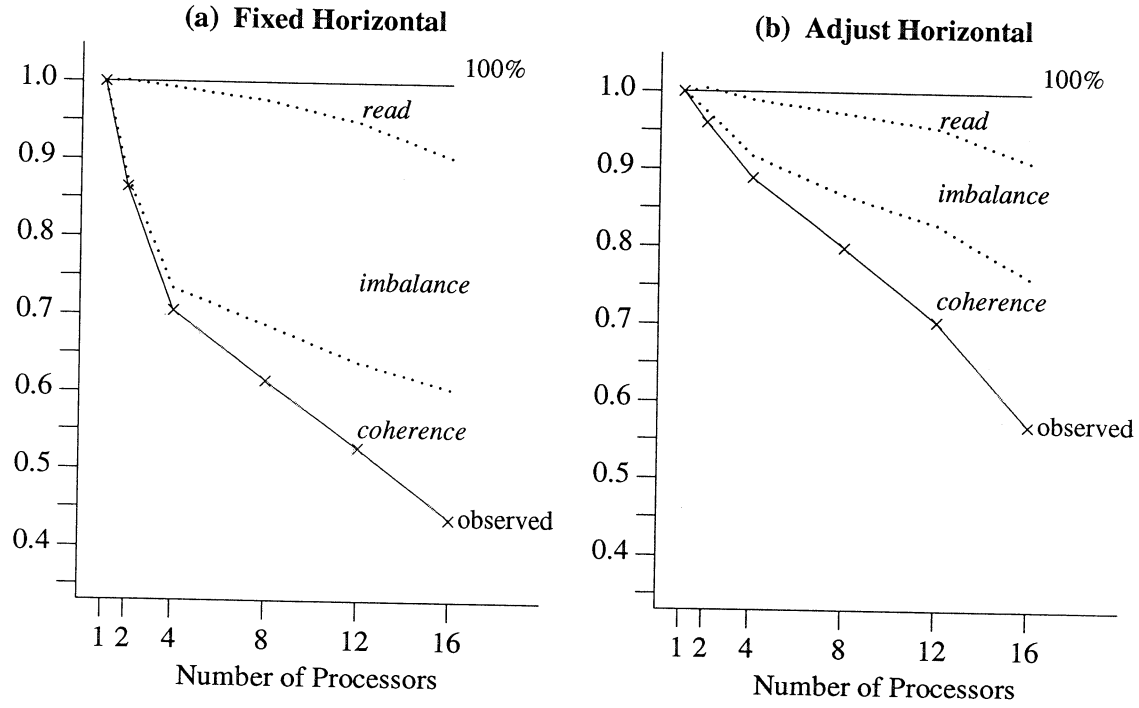


Figure 5.10. Junior. Loss of efficiency with a varying number of processors.

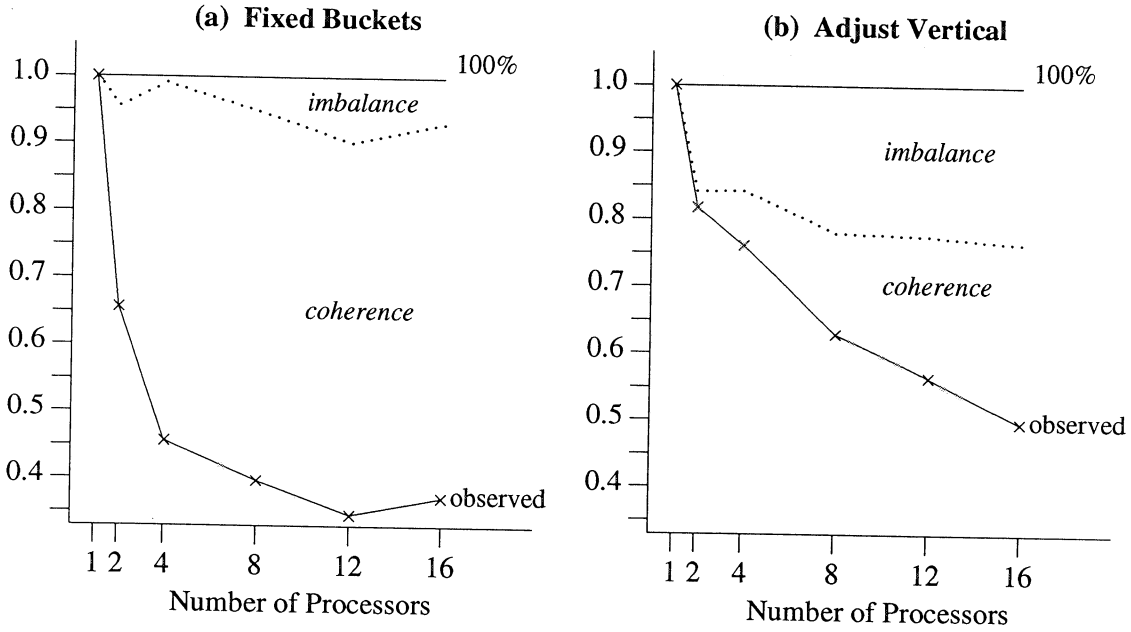


Figure 5.11. Tinny. Loss of efficiency with a varying number of processors. For this sequence, there was virtually no observed loss of efficiency due to read contention.

In some architectures, each node would compute values only for the pixels in its region. Nodes would then communicate to share information about pixels on their borders. In the RM-1 architecture, nodes do not cooperate in this fashion, and each node compute values for neighboring pixels locally. The cost of this approach is duplicate computation, rather than communications and synchronization overhead. This section will discuss the impact of this duplicate computation on runtime and on efficiency.

The Camera Move sequence was computed both with a one-by-one box filter (as documented in Table 5.7) and with a two-by-two Gaussian filter (as documented in Table 5.18). Independent of partitioning, it is inherently more expensive to use a wider filter. Comparing Tables 5.7 and 5.18, we see a six percent increase in the time to render the sequence on a single processor when we switch to a two-by-two filter. With sixteen nodes, there was an increase of eight to fourteen percent, depending on the partition.

The number of extra pixels that are computed because of inter-region dependence varies with the shape of the partition. If the partition has sixteen vertical strips, the nodes will duplicate one column of pixels on each side of the fifteen cuts. Thus, thirty of the 512 columns are duplicated, which amounts to a little less than six percent of the screen. With horizontal strips, thirty of 307 scanlines are duplicated, or almost ten percent of the screen. With a regular two-dimensional grid, there are three horizontal cuts and three vertical, and less than two percent of the pixels are duplicated. The two-dimensional partitions produced by median are not constrained to have cuts equally in each dimension, but typically the number of duplicated pixels is about the same as for a regular grid.

The extra computational effort to support a wider filter also depends on the distribution of complexity on the screen. The duplicated pixels may be empty, or they may contain a complex portion of the scene. If the image is concentrated in only part of the screen, then a fixed partition may make a number of cuts through empty pixels. But when an adaptive scheme successfully focuses computational effort on the active areas of the screen, the pixels bordering the cuts are more likely to contain data.

All told, the differences observed for Camera Move are relatively consistent among the partitioning schemes. The largest increases in work were observed for the fixed horizontal scheme, and the smallest increases were observed for fixed grid and adjust 2d. Comparing multiprocessor times with the single processor time, the difference in efficiency was no more than 0.03. This small difference in efficiency is encouraging; it says that the increased rendering time is mostly explained by the extra work inherent in supporting a wider filter and not by having partitioned the work among sixteen processors.

Adjust vertical had the fastest rendering time with the one-by-one filter. With the two-by-two filter, adjust 2d rendered more quickly, probably because fewer pixels were duplicated. However, if we add in the estimated 639 seconds of partitioning overhead (rebalancing nine frames at a cost of seventy-one seconds each, Table 5.2) adjust vertical still shows better overall performance.

Scheme	T_{16}	S_{16}	E_{16}	W_{16}	B_{16}		
					mean	min	max
Single processor	79682.0	1.00	1.000	79682.0	1.00	1.00	1.00
Adjust vertical	7143.5	11.15	0.697	100716.5	0.90	0.56	0.96
Adjust horizontal	10303.2	7.73	0.483	150119.4	0.91	0.72	0.94
*Adjust 2d	6982.3	11.41	0.713	92438.2	0.89	0.40	0.97
Model	11339.4	7.03	0.439	92744.0	0.51	0.49	0.55
*Fixed vertical	10408.3	7.66	0.478	96365.6	0.58	0.56	0.61
Fixed horizontal	10931.3	7.29	0.456	126150.1	0.72	0.71	0.73
Fixed grid	14732.5	5.41	0.338	94208.0	0.40	0.40	0.40

Table 5.18. Times and statistics with a 2x2 Gaussian filter. Times for Camera Move, reported as seconds of elapsed time, using 16 processors. The fastest adaptive scheme and the fastest fixed scheme are marked by asterisks. Two-by-two filters are not supported with the fixed buckets scheme.

5.7.6. Summary

The results confirm our hypotheses about the relationships between balance and coherence. Specifically, the fixed window schemes tend to exhibit good coherence and poor balance. The fixed bucket scheme tends to balance well, but it loses efficiency because it destroys coherence. It is possible to trade off coherence against balance by changing the size of tiles. For the Junior sequence only, the buckets were larger and more coherent. Consequently, the partition was less well balanced, but it generated less extra work.

As hypothesized, the adjust schemes were affected about equally by imbalance and by the loss of coherence. They have problems with imbalance mostly because the cost estimates are only approximate. Predicting frame-to-frame motion and refining the granularity of the estimates might provide more accurate estimates, but these techniques would probably require a great deal more overhead. The effects of reduced coherence can be minimized by choosing the appropriate baseline partition for adjust.

In general, none of the runs experienced serious problems with delays caused by slower texture access times. However, the way we have defined the loss due to read delays masks the actual load placed on the communications structure. Bear in mind that a major effect of reduced coherence is to increase the volume of the texture access traffic.

5.8. Production Experience

The Pixar animation production group has successfully used the one-dimensional vertical adjust scheme in production to render short films, including *Luxo Jr. in 3D* (1989), and *KnickKnack* (1989). All of the production frames were computed at a resolution of 1024 by 614, using a two-by-two Gaussian filter. The production runs generated some timing statistics, but they are less detailed than the statistics obtained for the experimental runs. The production frames have been run only on sixteen-node configurations, so we cannot estimate speedup or efficiency. Also, the simple statistics maintained by the production software would not allow us to analyze the factors that contribute to the loss of efficiency.

Most notably, the production runs balanced even better than predicted by the performance experiments. Adaptively partitioned frames from *Luxo Jr. in 3D* typically had $B_{16} > 0.90$. I analyzed the data for one sequence of ten frames and another of twenty frames. The median value for B_{16} was 0.96, with a lower quartile of 0.92 and an upper quartile of 0.97. This is noticeably better than the balance observed for the Junior sequence, which was selected from the same film. (Its median value of B_{16} was only 0.80.)

Initially, I hypothesized that the improved balance was due to the higher resolution of the production images. In Section 5.6.5 we noted that, given a constant resolution, the balance is better for partitions with fewer regions. As we decrease the number of nodes, we increase the dimensions of the regions and have relatively more pixels per cut. We see the same surface-to-volume effect when we increase the resolution and keep the number of nodes constant: the dimensions of the regions increase relative to the number of cuts and the workload tends to be more balanced. To test this hypothesis, I ran the Junior sequence again at a resolution of 1024 by 614, and compared the results for a 512 by 307 run of the same sequence.[5] Surprisingly, the balance statistics for the two runs were very close, differing by only 0.002 over the sequence. Apparently, the production runs were more balanced because of differences in the workload. I chose the Junior sequence at random before the animation for the film was complete, and in retrospect we discovered that it has more motion and less frame-to-frame coherence than is typical of the film on the whole.

I also hypothesized that the production runs retained more coherence. At higher resolution, the production images contained four times as many pixels as the experimental images, but the maximum grid

[5] For the higher-resolution experiment, I had to use a different RM-1 board with a slower memory cycle time and a host with a different type of local disk. I therefore ran the 512 by 307 sequence again on the new system. The measurements obtained for the two new runs are consistent, but we cannot compare them with numbers elsewhere in this chapter.

size remained a constant number of pixels. Probably, fewer grids would be cut by the edges of regions. Furthermore, the higher resolution should ease the texture read contention. Assuming the computation retains more coherence, fewer texture requests should be generated. In addition, a certain amount of computation accompanies each request so the rate of requests should generally not increase with the resolution. Therefore, the contention for texture reads should be no greater than what we observed for the experimental workload. These hypotheses were confirmed by comparing the two runs of the Junior sequence. Table 5.19 documents a remarkable improvement in the texture access statistics when we increase the image resolution. Because of the improved coherence, there are fewer texture page faults. Both the average read time for a texture page and the total texture read time drop significantly. Although the higher resolution sequence covers four times as many pixels, its rendering time is only 2.8 times greater than the rendering time for the lower resolution sequence. Since the higher resolution computations make fewer demands on the I/O and communications subsystems, we anticipate that a production workload can make efficient use of more processing nodes than the experimental workload.

One more advantage was observed during production. The Pixar animation team frequently recomputed frames, in which case they generated cost estimates from an earlier rendering of the same frame. Obviously, the adjust schemes work even better using the results of a previous run to estimate costs for the same frame.

Resolution	512 by 307	1024 by 614
texture pixels accessed	164,359,114	337,683,481
texture page faults	35,478	27,802
average texture read time	136 ms	99 ms
total texture read time	4,835 sec	2,742 sec
T_{16}	4,237.9 sec	11,817.2 sec
W_{16}	56,388.8 sec	157,715.8 sec

Table 5.19. Junior. Increasing the display resolution.

5.9. An Evaluation of the Rendering Architecture

A detailed performance evaluation of the RM-1 architecture is beyond the scope of this thesis. However, this section will briefly consider some of the key characteristics of the hardware and software and their impact on the system's performance.

5.9.1. The Hardware Architecture

The use of distributed memory, rather than shared memory, has a wide-ranging affect on implementation strategies. We assume that, in general, the entire database cannot fit in the local memory of a node. Therefore, local memory must contain the relevant subset of the database. The data may either be loaded in advance or demand paged; a subset of the model data is loaded in advance and the texture data are paged. Without shared memory, dynamic scheduling is difficult, because a new set of data must be loaded with each new work assignment and because nodes must communicate to schedule the work. The following section discusses dynamic scheduling in more detail.

The distributed memory model also accentuates the importance of coherence. In a distributed memory environment, it is more difficult to share results of a computation. When two RM-1 nodes need the same results, they compute the information independently instead of communicating the results. When coherence is destroyed, more computation is duplicated. This increases the amount of work, and, if textures are involved, it also tends to increase the disk traffic.

In the experience of production users, four megabytes of local memory is usually adequate, but more memory would be welcome. Certain workloads show a much higher rate of texture page faults than the experimental workload, and they would benefit from a larger texture cache. The size of local memory also limits the amount of intermediate results that can be cached. At times, users must adjust parameters such as the bucket size, so that the scratch data fit into the available memory. Occasionally, an image fails because the local memory is too small. In these cases, the user must manually subdivide the workload into two separate computations.

Another factor that affects performance is the single connection between the host and the board. The loss of efficiency due to texture read contention tends to be small, but it cannot be dismissed since its effect grows with the number of nodes (Figures 5.9 and 5.10). For instance, it accounted for nearly ten percent of the efficiency loss for the Junior sequence. The texture read delay does not seem to be primarily a problem with the server. As long as the server has adequate memory and local disk, it tends to have a high disk cache hit rate and consistent access times [Peac89]. Nor does the problem seem to be with communications delays on the RM-1 board. Although some nodes are as far as four hops from the host, measurements show little correlation between a node's distance from the host and its average texture read time. Having eliminated the other possible causes, we believe that the single connection to the disk server is the largest factor in the increased texture read delays.

5.9.2. Dynamic Scheduling

The system's support for static scheduling only, and not dynamic scheduling, is a feature of the software architecture that is influenced by the design of the hardware architecture. This section discusses one alternative to the RM-1 system's static algorithms and estimates its performance. Processor self-scheduling is a dynamic scheduling algorithm that balances the workload implicitly. The algorithm divides the screen into many small tiles, so that the number of tiles is much larger than the number of processors. Let us assume that the system uses the same set of regions as a fixed bucket scheme. Initially, the algorithm distributes one tile to each node. Whenever a node finishes a tile it takes another, until all of the tiles are rendered. On a shared-memory system, nodes typcially access a shared counter. On a message-based system, a server might supply a new work assignment when requested.

Because the Reyes software supports only static partitioning, we did not experiment directly with dynamic scheduling. We can, however, estimate its performance. The worst case processor utilization occurs when one node begins to render the most time-consuming tile just as the other nodes all complete their last tiles. Assume there are p processors and n tiles, and let $tile_i$ be the time to render the ith tile. Then $p-1$ nodes are idle while the final tile is rendered, which takes time $\max_{1 \le i \le n}(tile_i)$. The total idle time at the end of the computation is, thus,

$$(p-1)\max_{1 \le i \le n}(tile_i)$$

Let T_{render} be the elapsed time to render the entire image, Then, the proportion of idle time for all nodes is:

$$\frac{(p-1)\max_{1 \le i \le n}(tile_i)}{pT_{render}}$$

Subtracting the proportion of idle time from 100% gives B_n, the system utilization for this worst case scenario:

$$1 - \frac{(p-1)\max_{1 \le i \le n}(tile_i)}{pT_{render}}, \quad n > p \tag{5.3}$$

This expression holds when there are more tiles than processors, which is the only interesting case.

Let us substitute actual measurements into Equation 5.3, ignoring the scheduling overhead for now. I instrumented the renderer to report compute time per bucket and rendered a frame from the *Junior* sequence. The 512 by 307 image was divided into 9,824 tiles, each four pixels square. Over half of the tiles were empty, but 4,178 contained data to render. Evaluating Equation 5.3 with $n = 4{,}178$, $p = 16$, $\max_{1 \le i \le n}(tile_i) = 27.014$, and $T_{render} = 276.194$, gives a worst-case processor utilization (B_{16}) of 90.8% for this frame under dynamic scheduling. This compares well with the best utilization observed in Table 5.7. To estimate utilization in the typical case, I simulated processor self-scheduling. The simulator processed buckets in the same order that they were traced during the instrumented run. This order produced nearly perfect load balancing, with 99.4% processor utilization.

This approach promises excellent load balancing, but it shares the defects of the static bucket schemes. Unless we assign work carefully, we will tend to reduce coherence and, thus, increase the total amount of work. The scheduling overhead further increases the amount of work. Let's assume that the total amount of rendering work under this scenario would be the same as W_{16}, or the amount of work, for a fixed bucket scheme. (It's hard to predict the effects of texture caching, but this assumption seems reasonable.) Assume optimistically that the load would be perfectly balanced, that is, that each processor finishes at time $W_{16}/16$. Scheduling overhead aside, how quickly would the three test sequences be rendered? Table 5.20 shows the estimated rendering times under these assumptions. For Junior, the rendering time is only about four percent faster than the time observed for the fastest adaptive scheme (Table 5.7). For the other sequences, the best static adaptive scheme is still faster.

Therefore, improving the balance is not enough, unless we can retain more coherence. One approach to maintaining coherence is to vary the tile size, trading off load imbalance against increased coherence. Coherence tends to be stronger when the tiles are larger. In contrast, the load tends to be more balanced when tiles are smaller. Intuitively, dynamic scheduling produces a more balanced load when $n \gg p$. Given a constant image resolution, we achieve this inequality with smaller tiles. More formally, in Equation 5.3 the system utilization increases as $\max_{1 \le i \le n}(tile_i)/T_{render}$ decreases, since $(p-1)/p$ remains constant for a given number of processing nodes. How varying the tile size effects $\max_{1 \le i \le n}(tile_i)/T_{render}$ depends on the workload. However, in typical workloads, the ratio would normally decrease (and the utilization increase) as tiles get smaller.

Another approach to maintaining coherence is to assign adjacent tiles to the same node whenever possible. This scheme adds more scheduling complexity, but it preserves more coherence. Its effectiveness would depend on the workload and on the details of the scheduling algorithm.

So far, we have ignored the overhead of dynamic scheduling, which we expect to be much greater than for static scheduling. This approach needs either shared memory or very good interprocessor communications, both to support the scheduling overhead and to distribute the data for new work assignments. Such a scheme is most suitable for a shared-memory multiprocessor. Boothe had good success with dynamic scheduling for a ray tracer on a shared memory machine [Boot89]. However, he found many problems in implementing a similar algorithm on a distributed memory machine, and suggested that a static strategy might have been a better choice. Berger and Bokhari [Berg87] suggested that recursive bisection for scientific applications might outperform fine-grained dynamic queueing, even on shared memory multiprocessors, because recursive bisection preserves locality and reduces overhead.

Sequence	Dynamic Scheduling Estimate		Fastest Adaptive
	W_{16}	$\frac{W_{16}}{16}$	T_{16}
Camera Move	261621.7	15389.5	6494.9
Junior	59156.5	3697.3	3827.1
Tinny	19416.1	1213.5	1079.0

Table 5.20. Dynamic Scheduling.

5.9.3. The Software Architecture

The Reyes rendering software was developed on uniprocessors, first on the VAX 11/780 and later on the faster CCI Power 6/32. Both of these systems offered a large virtual address space and, typically, sixteen megabytes of physical memory. Although Reyes was conceived of as an experimental testbed, it was, in fact, used in production for several film projects during a period of at least four years. Naturally, the renderer's design and implementation were influenced by its original computing environments. It was ported to the RM-1 multiprocessor with few changes to the rendering algorithms.

Certain features that improved uniprocessor performance scale poorly on the RM-1 system. Section 5.7.3 explains that the system shades large pieces of surfaces, called grids, at one time to increase the computation's locality. The renderer saves any results that it does not need immediately. On a uniprocessor this strategy can reduce the amount of work because each grid is shaded once, and the cached results are always available. On the RM-1 system, this strategy is less successful because multiple nodes may need to duplicate the work of shading a single grid. Another performance strategy is to save computation by using more memory. For instance, increasing the size of buckets improves coherence but requires additional scratch memory. The smaller local memories of the RM-1 system tend to limit the bucket size, because the renderer more quickly runs out of scratch memory. The result is to reduce the use of coherence.[6]

In the case of Reyes, relatively efficient uniprocessor software was ported to a multiprocessor; the algorithms were not designed directly for any specific model of parallel computation. This has a subtle effect on our estimates of speedup. The multiprocessor efficiency and speedup metrics compare the compute time on $p > 1$ nodes with the compute time on a single node. Thus, two different factors can lead to a lower multiprocessor efficiency: slow multiprocessor times or fast uniprocessor times. With Reyes' uniprocessor heritage, we probably obtain a faster baseline measurement on a single node than we would if the software had been designed directly for a distributed-memory multiprocessor architecture. The good baseline performance (and not just the failure of the performance strategies to scale well) makes it harder to obtain a good speedup running Reyes on the RM-1.

The Reyes design experimented with different ideas about parallelism, including vectorized shading. Screen-based subdivision was supported by the principle of independence; if we subdivide the image, each region can be computed correctly independently of the other regions. Although the renderer may duplicate some calculations, its algorithms never depend on non-local results. Cook, Carpenter, and Catmull formalized concerns about parallelism in the design principles [Cook87]:

> The [image rendering] architecture should be able to exploit vectorization, parallelism, and pipelining. Calculations that are similar should be done together. For example, since the shading calculations are usually similar at all points on a surface, an entire surface should be shaded at the same time.

Experience with the RM-1 system demonstrates the inherent conflict in these design goals. As we have seen, vectorized shading requires good geometric locality while screen-based subdivision reduces the locality. It is hard to design efficient parallel software without a clear idea of the hardware architecture.

5.9.4. Scaling Issues

The experimental workload tends to use about eight nodes effectively before the efficiency falls. The previous section argues that a higher-resolution workload might use sixteen nodes more efficiently. Assume that we have sixteen RM-1 nodes that can be configured either as two eight-node systems or one sixteen-node system. Running in parallel, the two eight-node boards could each compute the full Camera Move sequence in 11,027.5 seconds (see Table 5.10). A single sixteen-node system would require about

[6] Increasing the bucket size also constrains the ways in which the screen may be partitioned to balance the load among the RM-1 nodes.

18% more time (or 12,989.8) seconds to compute the same twenty frames. Similarly, the sixteen-node system would take 40% longer to compute the Junior sequence and 27% longer to compute Tinny. The advantage of a sixteen-node board is that it provides 40-70% quicker turnaround on individual frames. Two eight-node systems would also require an additional port on a host and could increase the contention for resources on the host and on the texture server.

Experimental RM-1 systems have been configured with as many as sixty-four nodes. Unfortunately, no performance measurements were made.

Denser memory chips would allow the amount of memory on an RM-1 board to be increased by a factor of four. Given a choice, we would increase the nodes' local memory, rather than use all of the new memory for additional nodes. Adding more nodes with the same local memory would only worsen the texture access bottleneck. Such a system would probably require at least one more connection to the host. Increasing local memory would permit larger texture caches and reduce the bottleneck. The best solution might be to add fewer nodes to the system and still enlarge the local memory; a good compromise could be determined experimentally.

If faster nodes were available, they would generate texture access requests more quickly. Larger local memories and, thus, larger texture caches would help reduce the extra strain on the I/O system. A second connection to the host might also be needed, to reduce the I/O bottleneck.

5.10. Conclusions

This chapter demonstrated the effectiveness of adaptive spatial partitioning for rendering using simple, low-overhead cost estimates. Our strategy balances the load reasonably well while maintaining most of the spatial coherence in the computation. Our approach should be applicable to both general-purpose and special-purpose architectures. In particular, we studied an implementation on a multiprocessor with general-purpose processing nodes. Although the algorithm was designed to use history when computing successive frames of animation, it has also been useful for recomputing still images.

The performance analysis in this chapter stresses the importance of maintaining coherence and the tradeoffs between balance and coherence. When small groups of pixels are computed independently, the total amount of computation effort rises sharply. The results we observed for the fixed bucket scheme suggest that pixel interleaving should be thought of as a hardware technique for real-time systems, as discussed in Section 5.2.2. It is not a promising strategy for implicit load balancing for photorealistic rendering.

The experiments showed how the rendering time varies with the number of nodes on the RM-1 board; after eight processors, there is relatively little improvement in the rendering time for the experimental workload. Section 5.8 predicts, however, that a higher-resolution production workload may be able to use eight to sixteen processors more effectively. With the more effective partitioning schemes, the major underlying cause of the loss of efficiency is the difficulty of maintaining coherence as the number of regions increases.

Pat Hanrahan has proposed a software architecture for the RM-1 board that would address the issue of coherence. In his object-parallel approach, the subdivision and shading of each object would be the responsibility of a single node. Shaded pieces of objects would be distributed to the nodes according to a spatial subdivision, and then the visible surfaces would be determined. This architecture would have more complex communications and synchronization problems, but it is an interesting alternative that could be simulated before implementation.

Our approximation of the efficiency loss due to texture read delays, L^r, understates the problems of the input/output architecture. The I/O subsystem is strained by the loss of coherence as well as by multiprocessor contention, because a less coherent computation generates more texture read requests. The texture access bottleneck is probably less serious in the higher-resolution production runs, because

regions are larger in relation to grids. Still, it is unlikely that a single connection between the host and the RM-1 board will be adequate for much more than sixteen nodes. The texture access bandwidth would probably need to be increased also if faster cpu's were used.

The cost estimates produced by adjust suffer from two major flaws: they are very coarse-grained, and they depend on frame-to-frame consistency. With a different rendering implementation, it would be worthwhile to experiment with finer-grained cost estimates. Bear in mind that the overhead would increase with the number of cost data points. Predicting motion is, perhaps, less promising because it would increase the overhead considerably and require the analysis of a considerable amount of data. Adjust's advantage is its very low partitioning overhead. A more accurate cost estimate algorithm would still lose to adjust unless it was also very quick. Adjust performs well because it provides modest improvements at very low cost.

6
Conclusions and Directions for Future Research

In the past, realistic image synthesis was mostly limited to research and development environments. Researchers had first to discover algorithms for simulating visual experiences, before it was reasonable to become too concerned about performance. The algorithms and technology have now reached a stage of maturity where realistic image synthesis is accessible to a wide range of users. It is not only appropriate, but important to study rigorously the performance of image synthesis systems. A major goal of this dissertation is to demonstrate the application of performance analysis methodology to the field of image synthesis.

Two topics provide a foundation for the research. The first, in Chapter 2, is a review of a variety of rendering algorithms and the factors that affect their performance. The second, in Chapter 3, is a quantitative and qualitative characterization of the image synthesis workload. After establishing the nature of the systems and the workload, the dissertation considers two applications of performance methodology to image synthesis: measuring the performance of rendering systems and partitioning the workload for a MIMD rendering system.

Image synthesis performance depends on a complex set of intertwined factors. The individual algorithms and the ways that they interact in a complete rendering system determine, for the most part, the computational complexity of the rendering process.

The geometric complexity of the model and the image resolution, in general, determine the size of the problem. The types of geometric primitives can influence the complexity of the calculations. For example, Section 4.9.1 shows the performance penalty associated with ray tracing a set of parametric patches rather than spheres. Geometric characteristics are relatively well-defined and easy to quantify, as shown by the workload studies in Sections 3.2 and 3.4. However, a raw count of high-level modeling primitives will not always reflect the true size of a problem. For instance, the frequency of detail varies with the viewpoint, and the complexity of a sub-algorithm may depend on the screen size of an object or the number of internal rendering primitives it generates.

Realistic shading and texturing is an open-ended problem. It is much harder to quantify the shading complexity of a scene than the geometric complexity. A triangle is a triangle, and a patch is a patch, but the surface characteristics of a primitive can be made arbitrarily complex. Interactions among objects affect rendering costs, and this is especially true for shading. When objects are seen through transparent surfaces or are reflected by other objects, the complexity of rendering the scene increases in ways that are hard to predict. If the rendering system offers a flexible choice of modeling primitives and illumination models, the model data can specify varying algorithms and, therefore, change the computational complexity of the problem. The dissertation's major research vehicles, reyes, prman, and opal, are all examples of renderers that support programmable shading.

Chapter 2 categorizes the key factors that influence the performance of rendering systems. The actual effect of a cost factor depends on the algorithms and structure of a given rendering system. It is possible for a scene characteristic to increase the cost of one algorithm while having little effect on another. An experiment in Section 4.9 demonstrates this. With other factors held constant, the number of objects affects prman's performance but has little impact on opal.

6.1. Workload characterization

Chapter 3 discusses the complexity of images in terms of the cost factors given in Chapter 2. The analysis looks for trends in the development of image complexity. At any given time, the complexity of the image synthesis workload is shaped by the available technology. Historical data indicate that hardware and software improvements often allow image complexity to advance to a new level. The dissertation concentrates on the complexity of rendering and on the geometric and shading characteristics that affect rendering costs. Other types of complexity are now becoming important in graphics applications, such as animation and dynamics. Some examples are algorithms that drape fabric realistically or simulate hair. As noted by Greenberg [Gree91] and by Reeves, Ostby, and Leffler [Reev90], image complexity may now be limited more by the problem of modeling complex scenes than by the problem of rendering complex scenes.

The workload data in Chapter 3 give a good characterization of the geometric properties of a set of images and a more simple qualitative description of their shading properties. It is much harder to characterize shading complexity rigorously, and this is one possible direction for future work. The cost of shading and texturing an image is influenced by subtle interactions among objects, such as the reflection of one surface in another. Therefore, the most fruitful approach may be to instrument rendering systems and quantify shading characteristics at run time. The spatial distribution of complexity in the image was characterized by instrumenting the visible surface module. This resulting data describe the variation of complexity at a very fine level of granularity. It would be worthwhile to study the spatial distribution of complexity at a coarser level of granularity, dividing the image into larger regions of varying sizes. The data in Chapter 3 has another limitation. All of the images come from a single installation and were influenced by the same model of rendering. Another direction for future work is to examine the workload from a variety of installations. Finally, the results can be extended by profiling and comparing a variety of rendering systems. Using the same data for all systems would allow us to compare the response of different algorithms to the scene characteristics.

6.2. Performance Measurement

Chapter 4 proposes a methodology for measuring the performance of rendering systems. It classifies the factors that affect the cost of rendering and advocates controlled experiments that vary these cost factors. The four categories of cost factors are scene characteristics, viewing specifications, rendering parameters, and the computing environment.

In this scheme, the first type of performance experiment varies the scene characteristics. I have implemented a tool that generates models for rendering performance experiments. Under the control of parameters, the model generator Mg varies the geometric and shading characteristics of the scene. A major hypothesis in Mg's design is that benchmark images need not have the appearance of "real" computer graphics images. What is important, is that they share the computational characteristics of the real workload. Furthermore, it is easier to understand the effects of geometric variation if the scenes are composed of simple components and if the objects are positioned according to simple algorithms.

Some simple experiments with Mg demonstrate its ability to detect performance differences between systems. In Section 4.9, Mg's models are also used to explore the effect of workload changes on a single system's performance.

Mg is a prototype, and there are many ways it can be extended in the future. With the current interface, it is hard to generate very complex models without creating an unreasonably large file. More compact output formats or the inclusion of object instantiation in the interface are two techniques that might improve Mg's usefulness. The addition of new primitives, new light source models, and environment maps might support new applications. With only two scene generators, Mg still can create models with a range of characteristics. New scene generators could emphasize different features, such as environments suitable for radiosity experiments and models that combine a variety of object sizes and types. A scene generator that controls the frequency of detail would be useful for testing adaptive sampling and

antialiasing algorithms. Mg's interface already supports this type of variation with controls that change the size of objects and the texture coordinate mappings.

Adding support for animation is, in general, a hard problem. It would not be too difficult for Mg to support one simple case, in which the scene is static but the viewpoint moves. This case includes applications such as a camera move or an architectural "walk through."

One other direction for future research is to use Mg as basis for the work suggested in the previous section, profiling and characterizing varied rendering systems.

6.3. Workload Partitioning

Chapter 5 defines two potentially conflicting goals for a multiprocessor rendering system: to distribute the work evenly among a set of processing nodes, and to avoid introducing additional costs by partitioning the problem. The additional costs can include the scheduling overhead or extra work that is introduced by weakening coherence. The additional costs can also take the form of idleness, as processors experience delays due to contention or run out of work and wait for other nodes to finish. A successful solution to the workload partitioning problem must achieve good balance while preserving as much coherence as possible.

I proposed a partitioning algorithm for computing animated sequences and implemented it in the program *adjust*. To demonstrate the effectiveness of the algorithm, I compared three types of partitions that it generates with five competing schemes. The partitions were tested on three different animated sequences. For all of the test cases, adjust gave the best multiprocessor efficiency. Two qualities are important to adjust's success, its low overhead and its finding a middle ground in the tradeoffs between coherence and balance.

One reason for adjust's low overhead is its simple, easily obtained cost estimate: the processors' compute times for the previous frame. A second reason is its use of approximation. A single data point estimates the cost for all pixels rendered by a node. To simplify the problem further, it creates partitions along one axis, reducing the problem to a single dimension. The analysis in Chapter 5 concludes that the approximate cost estimates result in somewhat imbalanced partitions. However, the scheduling overhead is so low that the net effect is a respectable speedup.

The analysis of several competing schemes documents the tradeoffs between balance and coherence. It shows that adjust balances the load reasonably well without destroying too much coherence. This is possible because the algorithm adapts to the characteristics of the workload.

Two general principles are illustrated by this analysis. First, a moderate improvement can be worthwhile if it requires little overhead. Any algorithm that takes much longer than adjust to compute cost estimates has to be much more accurate to achieve the same overall performance. Second, the tradeoff between balance and coherence is a key consideration in parallelizing graphics. Chapter 4 provides a second illustration of this principle. Ray tracing is commonly considered a good candidate for multiprocessing because the rays are independent. Section 4.9.2 shows how the corresponding lack of coherence increases the cost of texture mapping.

The measurements in Chapter 5 concentrate on compute time. One effect of the loss of coherence is an increase in the number of texture references. Many image synthesis applications require very large models or many textures. Accessing model or texture data can be a problem, especially in computing environments with remote file servers. One direction for future work is to develop algorithms and strategies to localize texture references and reduce the i/o traffic related to texturing.

References

[Abra85] G. Abram, L. Westover and T. Whitted, Efficient alias-free rendering using bit-masks and look-up tables, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 53-59.

[Akel89] K. Akeley, The Silicon Graphics 4D/240GTX superworkstation, *IEEE Computer Graphics and Applications 9*,4 (July 1989), 71-83.

[Aman84] J. Amanatides, Ray tracing with cones, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 129-135.

[Aman87] J. Amanatides, Realism in computer graphics: a survey, *IEEE Computer Graphics and Applications 7*,1 (January 1987), 44-56.

[Ang91] P. H. Ang, P. A. Ruetz and D. Auld, Video compression makes big gains, *IEEE Spectrum 28*,10 (October 1991), 16-19.

[Apod91] T. Apodaca, Private communication, Pixar, Richmond, California, May 3, 1991.

[Apod92a] T. Apodaca, D. Peachey and M. Vandewettering, Opal (computer program), Pixar, Richmond, California, 1992.

[Apod92b] T. Apodaca, Private communication, Pixar, Richmond, California, November 9, 1992.

[Arvo87] J. Arvo and D. Kirk, Fast ray tracing by ray classification, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 55-64.

[Bade87] S. B. Baden, Run-time partitioning of scientific continuum calculations running on multiprocessors, LBL-24643, Physics Division, Lawrence Berkeley Laboratory, Berkeley, California, June 1987. PhD Thesis, Computer Science Division, University of California, Berkeley.

[Bade91] S. B. Baden and S. R. Kohn, A comparison of load balancing strategies for particle methods running on MIMD multiprocessors, CS91-99, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, California, May 1991.

[Beat82] J. C. Beatty and K. S. Booth, eds., *Tutorial, computer graphics*, IEEE Computer Society Press, Washington, D.C., second edition, 1982.

[Berg87] M. J. Berger and S. H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, *IEEE Transactions on Computers C-36*,5 (May 1987), 570-580.

[Blin76] J. F. Blinn and M. E. Newell, Texture and reflection in computer generated images, *Communications of the ACM 19*,10 (October 1976), 542-547.

[Blin77] J. F. Blinn, Models of light reflection for computer synthesized pictures, *SIGGRAPH '77 Conference Proceedings, Computer Graphics 11*,2 (July 1977), 192-198.

[Blin78] J. F. Blinn, Simulation of wrinkled surfaces, *SIGGRAPH '78 Conference Proceedings, Computer Graphics 12*,3 (August 1978), 286-292.

[Blin80] J. F. Blinn, L. C. Carpenter, J. M. Lane and T. Whitted, Scan line methods for displaying parametrically defined surfaces, *Communications of the ACM 23*,1 (January 1980), 23-34.

[Boot89] B. Boothe, Multiprocessor strategies for ray-tracing, Technical Report UCB/Computer Science Department 89/534, Computer Science Division (EECS), University of California, Berkeley, September 1989.

[Bora84] H. Boral and D. J. DeWitt, A methodology for database system performance evaluation, *Proceedings SIGMOD 1984, SIGMOD Record 14*,2 (June 1984), 176-185.

[BuiT75] P. Bui Tuong, Illumination for computer generated images, *Communications of the ACM 18*,6 (June 1975), 311-317. Reprinted in [Beat82], pages 449-455.

[Carp84] L. Carpenter, The A-buffer, an antialiased hidden surface method, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 103-108.

[Casp89] E. Caspary and I. D. Scherson, A self-balanced parallel ray-tracing algorithm, in *Parallel processing for computer vision and display*, P. M. Dew, R. A. Earnshaw and T. R. Heywood (editor), Addison-Wesley, 1989, 408-419.

[Catm74] E. Catmull, A subdivison algorithm for computer display of curved surfaces, UTEC-CSc-74-133, Computer Science Dept., University of Utah, 1974.

[Clar76] J. H. Clark, Hierarchical geometric models for visible surface algorithms, *Communications of the ACM 19*,10 (October 1976), 542-554.

[Clea86] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle and R. Vatti, Multiprocessor ray tracing, *Computer Graphics Forum 5*,1 (March 1986), 3-12.

[Cohe85] M. F. Cohen and D. P. Greenberg, The hemi-cube: a radiosity solution for complex environments, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 31-40.

[Cohe88a] M. F. Cohen, S. E. Chen, J. R. Wallace and D. P. Greenberg, A progressive refinement approach to fast radiosity image generation, *SIGGRAPH '88 Conference Proceedings, Computer Graphics 22*,4 (August 1988), 75-84.

[Cohe88b] M. F. Cohen, A consumer's and developer's guide to radiosity, in *A Consumer's and Developer's Guide to Image Synthesis*, Siggraph '88 tutorial notes, August 1988, 201-218.

[Cook82] R. L. Cook and K. E. Torrance, A reflection model for computer graphics, *ACM Transactions on Graphics 1*,1 (January 1982), 7-24.

[Cook84b] R. L. Cook, T. Porter and L. Carpenter, Distributed ray tracing, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 137-145.

[Cook84a] R. L. Cook, Shade trees, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 223-231.

[Cook86] R. L. Cook, Stochastic sampling in computer graphics, *ACM Transactions on Graphics 5*,1 (January 1986), 51-72.

[Cook87] R. L. Cook, L. Carpenter and E. Catmull, The reyes image rendering architecture, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 95-102.

[Crow77] F. C. Crow, The aliasing problem in computer-generated shaded images, *Communications of the ACM 20*,11 (November 1977), 799-805.

[Crow81] F. C. Crow, A comparison of antialiasing techniques, *IEEE Computer Graphics and Applications 1*,1 (January 1981), 40-48.

[Crow84] F. C. Crow, Summed-area tables for texture mapping, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 207-212.

[Crow86] F. C. Crow, Experiences in distributed execution: a report on work in progress, *ACM SIGGRAPH '86 Course Notes #15*, August, 1986.

[Crow88] F. C. Crow, Parallelism in rendering algorithms, *Proc. Graphics Interface '88*, June 1988, 87-96.

[Crow89] F. C. Crow, G. Demos, J. Hardy, J. McLaughlin and K. Sims, 3d image synthesis on the Connection Machine, in *Parallel processing for computer vision and display*, P. M. Dew, R. A. Earnshaw and T. R. Heywood (editor), Addison-Wesley, 1989, 254-269.

[Denn90] A. R. Dennis, An overview of rendering techniques, *Computers & Graphics 14*,1 (1990), 101-115.

[Dipp84] M. Dippe and J. Swensen, An adaptive subdivision algorithm and parallel architecture for realistic image synthesis, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984).

[Dipp85] M. Dippe and E. H. Wold, Antialiasing through stochastic sampling, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 69-78.

[Duff85] T. Duff, Compositing 3-D rendered images, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 41-44.

[Dunw88] J. C. Dunwoody and M. A. Linton, A dynamic profile of window system usage, *Proceedings, 2nd IEEE Conference on Computer Workstations*, March 1988, 90-99.

[Dunw90] J. C. Dunwoody and M. A. Linton, Tracing interactive 3d graphics programs, *Proceedings, 1990 Symposium on Interactive 3D Graphics, Computer Graphics 24*,2 (March 1990), 155-163, 267.

[Feib80] E. A. Feibush, M. Levoy and R. L. Cook, Synthetic texturing using digital filters, *SIGGRAPH '80 Conference Proceedings, Computer Graphics 14*,3 (July 1980), 294-301.

[Fium83] E. Fiume, A. Fournier and L. Rudolph, A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer, *SIGGRAPH '83 Conference Proceedings, Computer Graphics 17*,3 (July 1983), 141-150.

[Fium89] E. Fiume, *The mathematical structure of raster graphics*, Academic Press, Boston, 1989.

[Fole90] J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, *Computer graphics: principles and practices, 2nd edition*, Addison-Wesley, Reading, Massachusetts, 1990.

[Four82] A. Fournier, D. Fussel and L. Carpenter, Computer rendering of stochastic models, *Communications of the ACM 25*,6 (June 1982), 371-384.

[Four86] A. Fournier and W. T. Reeves, A simple model of ocean waves, *SIGGRAPH '86 Conference Proceedings, Computer Graphics 20*,4 (August 1986), 75-84.

[Fran80] W. R. Franklin, A linear time exact hidden surface algorithm, *SIGGRAPH '80 Conference Proceedings, Computer Graphics 14*,3 (July 1980), 117-123.

[Free80] H. Freeman, ed., *Tutorial and selected readings in interactive computer graphics*, IEEE Computer Society Press, Silver Spring, MD, 1980.

[Fuch77] H. Fuchs, Distributing a visible surface algorithm over multiple processors, *Proceedings ACM 77*, October 1977, 449-450.

[Fuch80] H. Fuchs, Z. M. Kedem and B. F. Naylor, On visible surface generation by a priori tree structures, *SIGGRAPH '80 Conference Proceedings, Computer Graphics 14*,3 (July 1980), 124-133.

[Fuch82] H. Fuchs, J. Poulton, A. Paeth and A. Bell, Developing Pixel-Planes, a smart memory-based raster graphics system, *Proc. Conf. on Advanced Research in VLSI*, MIT, 1982, 137-146.

[Fuch85] H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spach, J. D. Austin, F. P. Brooks Jr., J. G. Eyles and J. Poulton, Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 111-120.

[Fuji86] A. Fujimoto, T. Tanaka and K. Iwata, ARTS: accelerated ray-tracing system, *IEEE Computer Graphics and Applications 6*,4 (April 1986), 16-26.

[Glas84] A. S. Glassner, Space subdivision for fast ray tracing, *IEEE Computer Graphics and Applications 4*,10 (October 1984), 15-22.

[Gora84] C. M. Goral, K. E. Torrance, D. P. Greenberg and B. Battaile, Modeling the interaction of light between diffuse surfaces, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 213-222.

[Gour71] H. Gouraud, Continuous shading of curved surfaces, *IEEE Transactions on Computers C-20*,6 (June 1971), 623-629. Reprinted in [Free80], pages 302-308.

[Gran87] C. W. Grant, A preliminary taxonomy of visible surface algorithms, Technical Report UCRL-95948, Lawrence Livermore National Laboratory, Livermore, California, 1987.

[Gree90] S. A. Green and D. J. Paddon, A highly flexible multiprocessor solution for ray tracing, *The Visual Computer 6*(1990), 62-73.

[Gree91] D. P. Greenberg, More accurate simulations at faster rates, *IEEE Computer Graphics and Applications 11*,1 (January 1991), 23-29.

[Hage86] M. A. Hagen, *Varieties of realism: geometries of representational art*, Cambridge, Cambridge University Press, 1986.

[Hain87] E. A. Haines, A proposal for standard graphics environments, *IEEE Computer Graphics and Applications 7*,11 (November 1987), 3-5.

[Hall89] R. Hall, *Illumination and color in computer generated imagery*, Springer-Verlag, New York, 1989.

[Hanr88] P. Hanrahan, Median (computer program), Pixar, San Rafael, California, 1988.

[Hanr90] P. Hanrahan and J. Lawson, A language for shading and lighting calculations, *SIGGRAPH '90 Conference Proceedings, Computer Graphics 24*,4 (August 1990), 289-298.

[Heck86] P. S. Heckbert, Survey of Texture Mapping, *IEEE Computer Graphics and Applications 6*,11 (November 1986), 56-67.

[Heck91] P. S. Heckbert, Simulating global illumination using adaptive meshing, PhD Thesis, Computer Science Division, University of California, Berkeley, April 1991.

[Hubs82] H. Hubschman and S. W. Zucker, Frame-to-frame coherence and the hidden surface computation: constraints for a convex world, *ACM Transactions on Graphics 1*,2 (1982), 129-162.

[Iqba86] M. A. Iqbal, J. H. Saltz and S. H. Bokhari, A comparative analysis of static and dynamic load balancing strategies, *Proceedings, International Conference on Parallel Processing*, August 1986, 1040-1047.

[Joy88] K. I. Joy, C. W. Grant, N. L. Max and L. Hatfield, eds., *Tutorial, computer graphics: image synthesis*, IEEE Computer Society Press, Washington, D.C., 1988.

[Kaji88] J. T. Kajiya, An overview and comparison of rendering methods, *ACM SIGGRAPH '88 Course Notes: A Consumer's and Developer's Guide to Image Synthesis*, August 1988.

[Kapl79] M. Kaplan and D. P. Greenberg, Parallel processing techniques for hidden surface removal, *SIGGRAPH '79 Conference Proceedings, Computer Graphics 13*,2 (August 1979), 300-307.

[Kay79] D. S. Kay, *Transparency, refraction and ray tracing for computer synthesized images*, M.S. Thesis, Program of Computer Graphics, Cornell University, January 1979.

[Kay86] T. L. Kay and J. T. Kajiya, Ray tracing complex scenes, *SIGGRAPH '86 Conference Proceedings, Computer Graphics 20*,4 (August 1986), 269-278.

[Kuck78] D. J. Kuck, *The structure of computers and computation*, John Wiley & Sons, New York, 1978.

[Lass87] J. Lasseter, Principles of traditional animation applied to 3d computer animation, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 35-44.

[Laws89] J. Lawson, Private communication, Pixar, San Rafael, California, June 29, 1989.

[LeGa91] D. Le Gall, MPEG: a video compression standard for multimedia applications, *Communications of the ACM 34*,4 (April 1991), 46-58.

[Lee85] M. E. Lee, R. A. Redner and S. P. Uselton, Statistically optimized sampling for distributed ray tracing, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 61-67.

[Lint86] M. A. Linton, Benchmarking engineering workstations, *IEEE Design & Test of Computers 3*,3 (June 1986), 25-30.

[Moln90] S. Molnar and H. Fuchs, Advanced raster graphics architecture, in *Computer graphics: principles and practices, 2nd edition*, J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes (editor), Addison-Wesley, Reading, Massachusetts, 1990, 873-907.

[Newe72] M. E. Newell, R. G. Newell and T. L. Sancha, A solution to the hidden surface problem, *Proceedings ACM National Meeting*, 1972, 443-450. Reprinted in [Free80].

[Para85] Paramount, Young Sherlock Holmes, (film), 1985.

[Park80] F. I. Parke, Simulation and expected performance analysis of multiple processor Z-Buffer Systems, *SIGGRAPH '80 Conference Proceedings, Computer Graphics 14*,3 (July 1980), 48-56.

[Peac89] D. Peachey, Private communication, Pixar, San Rafael, California, October 26, 1989.

[Pixa89] The RenderMan™ Interface, Version 3.1, Pixar, 1001 West Cutting Blvd., San Rafael CA 94804, September 1989.

[Port84] T. Porter and T. Duff, Compositing digital images, *SIGGRAPH '84 Conference Proceedings, Computer Graphics 18*,3 (July 1984), 253-259.

[Potm89] M. Potmesil and E. M. Hoffert, The Pixel Machine: a parallel image computer, *SIGGRAPH '89 Conference Proceedings, Computer Graphics 23*,3 (July 1989), 69-78.

[Prio89] T. Priol and K. Bouatouch, Static load balancing for a parallel ray tracing algorithm on a MIMD hypercube, *The Visual Computer 5*(1989), 109-119.

[Reev83] W. T. Reeves, Particle systems—a technique for modelling a class of fuzzy objects, *SIGGRAPH '83 Conference Proceedings, Computer Graphics 17*,3 (July 1983), 359-376.

[Reev85] W. T. Reeves and R. Blau, Approximate and probabilistic algorithms for shading and rendering structured particle systems, *SIGGRAPH '85 Conference Proceedings, Computer Graphics 19*,3 (July 1985), 313-322.

[Reev87] W. T. Reeves, D. H. Salesin and R. L. Cook, Rendering antialiased shadows with depth maps, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 283-291.

[Reev89] W. T. Reeves, Private communication, Pixar, San Rafael, California, March 23, 1989.

[Reev90] W. T. Reeves, E. F. Ostby and S. J. Leffler, The Menv modelling and animation environment, *Journal of Visualization and Computer Animation 1*,1 (1990), 33-40.

[Rubi80] S. M. Rubin and T. Whitted, A 3-dimensional representation for fast rendering of complex scenes, *SIGGRAPH '80 Conference Proceedings, Computer Graphics 14*,3 (July 1980), 110-116.

[Scho86] P. Schoeler and A. Fournier, Profiling graphic display systems, *Proceedings Graphics Interface '86*, Vancouver, British Columbia, May 1986, 49-55.

[Schu80] R. A. Schumacker, A new visual system architecture, *Proc. Second Interservice/Industry Training Equipment Conference*, Salt Lake City, Utah, November 18-20, 1980.

[Snyd87] J. M. Snyder and A. H. Barr, Ray tracing complex models containing surface tessellations, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 119-128.

[Suth74] I. E. Sutherland, R. F. Sproull and R. A. Schumacker, A characterization of ten hidden-surface algorithms, *Computing Surveys 6*,1 (March 1974), 1-55.

[Tice88] S. E. Tice, M. Fusco and P. Straley, The picture level benchmark, *Computer Graphics World 11*,7 (July 1988), 123.

[Upst90] S. Upstill, *The RenderMan Companion*, Addison-Wesley, 1990.

[Verb84] C. P. Verbeck and D. P. Greenberg, A comprehensive light-source description for computer graphics, *IEEE Computer Graphics and Applications 4*,7 (July 1984), 66-75.

[Wall87] J. R. Wallace, M. F. Cohen and D. P. Greenberg, A two-pass solution to the rendering equation: a synthesis of ray tracing and radiosity methods, *SIGGRAPH '87 Conference Proceedings, Computer*

Graphics 21,4 (July 1987), 311-320 .

[Warn69] J. E. Warnock, A hidden surface algorithm for computer generated halftone pictures, Technical Report 4-15, Computer Science Dept., University of Utah, June 1969.

[Watk70] G. S. Watkins, A real time visible surface algorithm, UTEC-CSc-70-101, Computer Science Dept., University of Utah, June 1970.

[Weem91] C. Weems, E. Riseman, A. Hanson and A. Rosenfeld, The DARPA image understanding benchmark for parallel computers, *Journal of Parallel and Distributed Computing 11*,1 (January 1991), 1-24.

[Wegh84] H. Weghorst, G. Hooper and D. P. Greenberg, Improved computational methods for ray tracing, *ACM Transactions on Graphics 3*,1 (January 1984), 52-69.

[Weil77] K. Weiler and P. Atherton, Hidden surface removal using polygon area sorting, *SIGGRAPH '77 Conference Proceedings, Computer Graphics 11*,2 (July 1977), 214-222.

[Whel85] D. S. Whelan, Animac: a multiprocessor architecture for real-time computer animation, Technical Report 5200:Tr:85, PhD Thesis, Computer Science Department, California Institute of Technology, May 1985.

[Whit80] T. Whitted, An improved illumination model for shaded display, *Communications of the ACM 23*,6 (June 1980), 343-349.

[Whit82] T. Whitted, Processing requirements for hidden surface elimination and realistic shading, *Proceedings, Spring COMPCON 82*, 1982, 245-250.

[Will78] L. Williams, Casting curved shadows on curved surfaces, *SIGGRAPH '78 Conference Proceedings, Computer Graphics 12*,3 (August 1978), 270-274.

[Will83] L. Williams, Pyramidal parametrics, *SIGGRAPH '83 Conference Proceedings, Computer Graphics 17*,3 (July 1983), 1-11.

[Zyda90] M. J. Zyda, M. A. Fichten and D. H. Jennings, Meaningful graphics workstation performance measurements, *Computers & Graphics 14*,3/4 (1990), 519-526.

Appendix A
Image Complexity, 1966-1991

The table on the following pages summarizes information about images published in the computer graphics literature between 1966 and 1991. The survey covers images published in the SIGGRAPH proceedings from 1977 to 1991, ACM Transactions on Graphics, IEEE Computer Graphics and Applications, and anthologies in the IEEE tutorial series. Also included are references cited in this thesis, graphics papers published in the Communications of the ACM, and, for earlier data, publications from other journals. Chapter 3 discusses the selection criteria. The notes below explain the interpretation of the fields in the table entries.

The data in this table come from the reference cited for each image. The table is not complete, because many of the references do not provide information for all of the table's columns. I have made only a few guesses, based on an inspection of the image or statements in the publication.

Year The year that the reference was published, unless it gives a different year of creation for the image.

Source A reference from the list included at the end of this Appendix, or a reference to a chapter of this thesis.

Picture The author's title for the image, possibly shortened, or a descriptive phrase to identify it. There is only one table entry for each distinct scene. If the reference contains more than one image of a scene, the entry describes the most complex version.

Type The first letter shows whether the image was created for production use in an actual application (P) or for computer graphics research (R). The second letter shows whether the image is a frame from an animated sequence (A) or an independent still image (S).

Objects No attempt has been made to reconcile varying definitions of "object," which may mean anything from a low-level primitive to a real-world object. Therefore, it is not meaningful to compare the entries in this column except among the images from a single reference.

Primitives: total
The total includes all modeling primitives, as reported in the publication. In some cases, the numbers appear to be approximate or to include only primitives relevant to the research reported in the reference; therefore, the total may not equal the sum of the next two columns (polygons and other primitives). If the size of a model can be varied, the table reports the number of primitives used to generate the published image.

Primitives: polygons
The number of polygons in the model.

Primitives: other
The number of modeling primitives other than polygons, broken down by type, if possible. For some exclusively polygonal models, this column shows the number of edges and/or vertices in the polygons.

Codes The five entries in this column hint at the shading and rendering complexity: A (antialiasing), R (reflections), S (shadows), T (transparency), and X (texture mapping). If the code letter appears in the table, either the corresponding quality can be seen in the image or it is documented in the reference. If a dash (–) appears instead, either the corresponding quality is absent or there is not enough information to confirm its presence.

Notes Other information given in the reference, such as the source of the model, the number of light sources, or an unusually high resolution. Unless otherwise indicated, all images are approximately 512 by 512.

Year	Source	Picture	Type	Objects	Primitives			Codes	Notes
					total	polygons	other		
1966	[Suth70]	blocks	RS	40					40 blocks
1967	[Appe67]	two objects	RS	2	32	32			hidden-line
		three machine parts	RS	3	41	41			
		aircraft	RS	1	143	143			
		aircraft over carrier	RS	2	190	190			
1969	[Gali69]	convex object	RS	1	242	242	372 edges		hidden-line
		German pavilion	RS		106	106	168 vertices, 252 edges		
1970	[Bouk70]	A-frame cottage	RS		17	17	60 edges		
		torus	RS	1	225	225	900 edges		
		cubes	RS	32	160	160	640 edges		
1970	[Lout70]	building	RS			74	92 vertices, 220 edges		hidden-line
1971	[Gold71]	helicopter	RA		21		21		
1971	[McGr71]	intersection (a)	RS		14	14			hidden-line
		airplane	RS		127	127			similar to [Appe67]
		house	RS		79	79			similar to [Gali69]
1972	[Newe72]	Figure 3	RS		14	14			
		Figure 4	RS		225	225			based on [Bouk70]
		Figures 5,6	RS		140	140		——T–	
		Figure 7	RS		63	63			based on [Watk70]
		Figure 8	RS		625	625		——T–	
		Figure 9	RS		625	625		——T–	
		Figure 10	RS		720	720			
		Figure 11	RS		450	450			
		Figure 12	RS		393	393			
		Figure 13	RS		195	195			
1974	[Suth74]	Roberts' House	RS		200	200			proposed "simple" scene
		Harbor	RS		5,000	5,000			hypothetical "difficult" scene
		Big Harbor	RS		120,000	120,000			speculative "giant" scene
1977	[Haml77]	cone	RS			331			hidden-line
		cylinder	RS			235			
		bottle	RS			41			
		aircraft	RS			440			
1978	[Will78]	robot	R?				> 350 bicubic patches	?–STX	

Year	Source	Picture	Type	Objects	Primitives			Codes	Notes
					total	polygons	other		
1979	[Csur79]	tribbles	R?		500,000	500,000			
		Burley, Idaho	R?		25,000	25,000			
		Beldar's hat	R?		12,000	12,000			
		cocklebur	R?		46,000	46,000			
1980	[Blin80]	teapot	RS		28		28 bicubic patches		
1981	[Crow81]	test scene	RS			24,568			
1982	[Whit82]	bottles and glasses	RS			23,554	70,661 edges	—STX	no antialiasing
		room scene	RS			32,000	44,000 edges		
		logo	RS			12,492	19,450 edges		
		fractal terrain	RS			18,188	28,421 edges		
		30-sided torus	RS			1,800	2,760 edges		
1982	[Crow82]	banana	RS		256	256			
		Easter egg	RS		3,552	3,552			
		George (skeleton)	RS		19,666	19,666			
1983	[Kaji83]	fractal surface	RS		262,144	262,144		–RS—	no antialiasing
1984	[Wegh84]	Pencils	RS	38	1806	1806		RS—	1 light or 3 lights
		Polygons	RS	21	1983	1980	3 spheres	–S—	3 lights
		Camera	RS	39	1393	1392	1 sphere	–S—	1 light
		Office Scene	RS	63	2109	2101	8 spheres	RST–	5 lights
		Pool Room	RS	38	853	835	13 spheres, 5 cylinders	RST–	5 lights
1984	Chap. 3	Andre	RA		91		74 spheres, 13 cylinders	A——X	
1984	[Brow84]	Gunstar	PA			648,000			3000x5000 pixel resolution
		Starfighter	PA			250,000			average frame from film
1984	[Glas84]	checkerboard/balls	RS	53				RS—	
		spirals	RS	401	401		401 spheres	–S—	
		recursive pyramid	RS	1,025	1,025	1,025		–S—	
		geodesic cube	RS	1,536	1,536			–S—	
		sinc function	RS				3,656 spheres	RS—	
1985	[Whel85]	House	RS			178			
		Caltech	RS			990			
		VW	RS			1,072			Sutherland's model, Univ. of Utah
		X-Wing	RS			1,407			
		Frame1100	RS			8,089			
		Fractal64	RS			65,030			
1985	[Reev85]	stained glass knight	PA	82	1403		1403 bicubic patches	AR–TX	82 pieces of glass, 23 light sources

Year	Source	Picture	Type	Objects	Primitives			Codes	Notes
					total	polygons	other		
1986	[Bish86]	chessmen	RS			14620			2048x2048 pixels
1986	[Four86]	beach at sunset	RA				92,724 bicubic patches	AR—X	2048x1228 pixel resolution
		wave refraction	RS				85,680 bicubic patches	R—X	
1986	[Fuji86]	DNA	RS	7011	7011		7011	—S—	antialiased
		CG Tokyo 85	RS		10584			A–S—	
		RX-7	RS			46,000		RS—	antialiased(?)
1986	[Berg86]	bench	RA		697			A–S—	shadow polygons excluded;
		Tony de Peltrie	RA			4211		A–S—	16 samples/pixel
		shoes	RA			1141		A–S—	
		Montreal	RS			8854		A–S—	
1986	[Hain86]	chessboard	RS	132	4944	4944			4 lights
		cards	RS	41	3286	3286			5 lights
		highchair	RS	33	6712	6702	2 cylinders, 8 quadrics		3 lights
		gazebo	RS	80	3222	3222			8 lights
		tree	RS	768	773	6	256 spheres, 511 quadrics	—T–	
		kitchen	RS	224	1413	1298	76 cylinders, 35 quadrics, 4 spheres		5 lights
1986	[Joy86]	shiny spheres	RS	61	76		60 spheres, 16 patches	–RS—	
		lamp	RS		310		310	ARST–	4 lights
1986	[Kay86]	trees	RA		110,000			ARS—	all antialiased
		pyramid**4	RS		1024	1024		A–S—	based on [Glas84]
		pyramid**5	RS		4096	4096		A–S—	
		superquadric	RS	402	2402	2400	2	ARS—	2 lights
		tree branches	RS	2	1272	1	1271	A–S—	
		tree leaves	RS	2	7456	6185	1271	A–S—	
1987	[Arvo87]	pyramid**4	RS		1024	1024			
		pyramid**10	RS		4.2×10^6	4.2×10^6			from [Kay86]
		tree leaves	RS		7455			–S—	from [Kay86]
		tree branches	RS		1272			–S—	from [Kay86]
		grove	RS	64	477,121			–S—	
		teapot	RS		1824			RS—	
		platonic solids	RS		1405			RS–X	
1987	[Dyer87]	skull1	RS	1		2034		A——	
		skull2,3,4	RS	5		10420		A——	up to 3 lights

Year	Source	Picture	Type	Objects	Primitives			Codes	Notes
					total	polygons	other		
1987	Chap. 3	bike shop window	RA		9050		see Table 3.3(a)	ARSTX	7 lights
	[Ostb87]	night in bike store	R	A		16,783		ARSTX	19 texture channels, 1024x788
1987	[Snyd87]	graphics lab	RS				100 intermediate triangles	A–S–X	diffuse shadows from 3 sources
		teapot museum piece	RS				10,000 intermediate triangles	ARS—	
		reflective bristles	RS				15,000 intermediate triangles	A–S—	
		Statue of Liberty	RS				100,000 intermediate triangles	A–S—	
		brass ornament	RS				100,000 intermediate triangles	ARSTX	
		flowers, grass, clovers	RS				200,000 intermediate triangles	A–S—	
		glass museum piece	RS				400,000 intermediate triangles	ARST–	
		grass and trees	RS				2×10^9 intermediate triangles	A–S—	
		field of grass	RS				4×10^{11} intermediate triangles	A–S—	
1987	[Hain87]	balls	RS		7382	1	7381 spheres	RS—	3 lights
		gears	RS	64	9345	9345		RST–	54 multiple-edge, concave polygons
		mountain	RS	5	8196	8192	4 spheres	RST–	
		rings	RS	141	8401	1	4200 spheres, 4200 cylinders	RS—	3 lights
		tetra	RS	1	4096	4096		–S—	based on [Glas84, Kay86]
		tree	RS		8191	1	4095 cones, 4095 spheres	–S—	7 lights
1989	[Potm89]	station	RS			17,000			
		tea room	RS			50,000		A–ST–	area light source, 16 samples/pixel
		museum room	RS			2,000		——X	13 texture maps, 4Mb texture data
1990	[Mund90]	convertible	?S			20,000		–RST	triangles; 20 lights
1990	[Bara90]	falling jack	RA			97	6 curved surfaces	–RSTX	rigid body dynamics
		falling dice	RA			89	102 curved surfaces	–RST–	
1990	[Max90]	chromatin fiber	RA				14,050 cone spheres	A——	motion blur
		branching grape root	RA				13,000 cone spheres	A—TX	motion blur, 2048x1500
		torus	RS	1			24 cone spheres	A——	
		Christmas tree	RS				1770 cone spheres, 1000 spheres	A—TX	
1990	[Sale90]	simple scene	RS				500 spheres	A–STX	
1990	[Gree91b]	printing press	RS			320,000		–RS–X	
1991	[Gree91a]	workstation	RS	2600	21,600	20,000	1600 quadric surfaces	ARSTX	9 samples/pixel, 1280x1024 pixels

Sources

[Appe67] A. Appel, The notion of quantitative invisibility and the machine rendering of solids, *Proceedings ACM National Meeting*, 1967, 214-220. Reprinted in [Free80].

[Arvo87] J. Arvo and D. Kirk, Fast ray tracing by ray classification, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 55-64.

[Bara90] D. Baraff, Curved surfaces and coherence for non-penetrating rigid-body simulation, *SIGGRAPH '90 Conference Proceedings, Computer Graphics 24*,4 (August 1990), 19-28.

[Beat82] J. C. Beatty and K. S. Booth, eds., *Tutorial, computer graphics*, IEEE Computer Society Press, Washington, D.C., second edition, 1982.

[Berg86] P. Bergeron, A general version of Crow's shadow volumes, *IEEE Computer Graphics and Applications 6*,9 (September 1986), 17-28.

[Bish86] G. Bishop and D. M. Weimer, Fast Phong shading, *SIGGRAPH '86 Conference Proceedings, Computer Graphics 20*,4 (August 1986), 103-106.

[Blin80] J. F. Blinn, L. C. Carpenter, J. M. Lane and T. Whitted, Scan line methods for displaying parametrically defined surfaces, *Communications of the ACM 23*,1 (January 1980), 23-34.

[Bouk70] W. J. Bouknight, A procedure for generation of three-dimensional half-toned computer grapics presentations, *Communications of the ACM*, September 1970, 292-301. Reprinted in [Free80].

[Brow84] M. D. Brown, The last starfighter: making a movie with computer graphics, *IEEE Computer Graphics and Applications 4*,7 (July 1984), 7-8.

[Crow81] F. C. Crow, A comparison of antialiasing techniques, *IEEE Computer Graphics and Applications 1*,1 (January 1981), 40-48.

[Crow82] F. C. Crow, A More Flexible Image Generation Environment, *SIGGRAPH '82 Conference Proceedings, Computer Graphics 16*,3 (July 1982), 9-18.

[Csur79] C. Csuri, R. Hackathorn, R. Parent, W. Carlson and M. Howard, Towards an interactive high visual complexity animation system, *SIGGRAPH '79 Conference Proceedings, Computer Graphics 13*,2 (August 1979), 289-299.

[Dyer87] S. Dyer and S. Whitman, A vectorized scan-line z-buffer rendering algorithm, *IEEE Computer Graphics and Applications 7*,7 (July 1987), 34-45.

[Four86] A. Fournier and W. T. Reeves, A simple model of ocean waves, *SIGGRAPH '86 Conference Proceedings, Computer Graphics 20*,4 (August 1986), 75-84.

[Free80] H. Freeman, ed., *Tutorial and selected readings in interactive computer graphics*, IEEE Computer Society Press, Silver Spring, MD, 1980.

[Fuji86] A. Fujimoto, T. Tanaka and K. Iwata, ARTS: accelerated ray-tracing system, *IEEE Computer Graphics and Applications 6*,4 (April 1986), 16-26.

[Gali69] R. Galimberti and U. Montanari, An algorithm for hidden line elimination, *Communications of the ACM 12*,4 (April 1969), 206 -211.

[Glas84] A. S. Glassner, Space subdivision for fast ray tracing, *IEEE Computer Graphics and Applications 4*,10 (October 1984), 15-22.

[Gold71] R. A. Goldstein and R. Nagel, 3-d visual simulation, *Simulation 16*,1 (January 1971), 25-31.

[Gree91a] D. P. Greenberg, A ray tracing simulation of a radiosity simulation, *IEEE Computer Graphics and Applications 11*,1 (January 1991), 6-7.

[Gree91b] D. P. Greenberg, More accurate simulations at faster rates, *IEEE Computer Graphics and Applications 11*,1 (January 1991), 23-29.

[Hain86] E. A. Haines and D. P. Greenberg, The light buffer: a shadow-testing accelerator, *IEEE Computer Graphics and Applications 6*,9 (September 1986), 6-16.

[Hain87] E. A. Haines, A proposal for standard graphics environments, *IEEE Computer Graphics and Applications 7*,11 (November 1987), 3-5.

[Haml77] G. Hamlin, Jr. and C. W. Gear, Raster-scan hidden surface algorithm techniques, *SIGGRAPH '77 Conference Proceedings, Computer Graphics 11*,2 (July 1977), 206-213. Reprinted in [Free80].

[Joy86] K. I. Joy and M. N. Bhetanabhotla, Ray tracing parametric surface patches utilizing numerical techniques and ray coherence, *SIGGRAPH '86 Conference Proceedings, Computer Graphics 20*,4 (August 1986), 279-285.

[Kaji83] J. T. Kajiya, New techniques for ray tracing procedurally defined objects, *ACM Transactions on Graphics 2*,3 (July 1983), 161-181.

[Kay86] T. L. Kay and J. T. Kajiya, Ray tracing complex scenes, *SIGGRAPH '86 Conference Proceedings, Computer Graphics 20*,4 (August 1986), 269-278.

[Lout70] P. P. Loutrel, A solution to the hidden-line problem for computer-drawn polyhedra, *IEEE Transactions on Computers*, March 1970, 221-229. Reprinted in [Free80].

[Max90] N. Max, Cone-spheres, *SIGGRAPH '90 Conference Proceedings, Computer Graphics 24*,4 (August 1990), 59-62.

[McGr71] F. J. McGrath, A method for eliminating hidden lines with polyhedra, *Simulation 16*,1 (January 1971), 37-41.

[Mund90] G. Mundell, Convertible Grand Prix prototype (computer-generated image), *IEEE Computer Graphics and Applications 10*,4 (July 1990), 9.

[Newe72] M. E. Newell, R. G. Newell and T. L. Sancha, A solution to the hidden surface problem, *Proceedings ACM National Meeting*, 1972, 443-450. Reprinted in [Free80].

[Ostb87] E. Ostby and B. Reeves, A night in the bike store (computer-generated image), *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 352.

[Potm89] M. Potmesil and E. M. Hoffert, The Pixel Machine: a parallel image computer, *SIGGRAPH '89 Conference Proceedings, Computer Graphics 23*,3 (July 1989), 69-78.

[Reev85] W. T. Reeves, Statistics associated with the making of *Young Sherlock Holmes*, Technical Memo 140, Computer Graphics Project, Lucasfilm Ltd, San Rafael, California, October 20, 1985.

[Sale90] D. Salesin and J. Stolfi, Rendering csg models with a zz-buffer, *SIGGRAPH '90 Conference Proceedings, Computer Graphics 24*,4 (August 1990), 67-76.

[Snyd87] J. M. Snyder and A. H. Barr, Ray tracing complex models containing surface tessellations, *SIGGRAPH '87 Conference Proceedings, Computer Graphics 21*,4 (July 1987), 119-128.

[Suth70] I. E. Sutherland, Computer displays, *Scientific American 222*,6 (June 1970). Reprinted in [Beat82], pages 4-20.

[Suth74] I. E. Sutherland, R. F. Sproull and R. A. Schumacker, A characterization of ten hidden-surface algorithms, *Computing Surveys 6*,1 (March 1974), 1-55.

[Watk70] G. S. Watkins, A real time visible surface algorithm, UTEC-CSc-70-101, Computer Science Dept., University of Utah, June 1970.

[Wegh84] H. Weghorst, G. Hooper and D. P. Greenberg, Improved computational methods for ray tracing, *ACM Transactions on Graphics 3*,1 (January 1984), 52-69.

[Whel85] D. S. Whelan, Animac: a multiprocessor architecture for real-time computer animation, Technical Report 5200:Tr:85, PhD Thesis, Computer Science Department, California Institute of Technology, May 1985.

[Whit82] T. Whitted and D. M. Weimer, A software testbed for the development of 3d raster graphics systems, *ACM Transactions on Graphics 1*,1 (January 1982), 43-58.

[Will78] L. Williams, Casting curved shadows on curved surfaces, *SIGGRAPH '78 Conference Proceedings, Computer Graphics 12*,3 (August 1978), 270-274.

Appendix B

Mg Interface Specifications and Source Code

This printed version of the technical report contains only Section B.1, which specifies the Mg interface. The complete text of this appendix, which includes all of the source listings described below, is available online. The online text and source listings may be obtained by anonymous ftp from toe.cs.berkeley.edu in the directory pub/tech–reports/cs/csd–93–736. You may also request a copy of the source listings by mail to ricki@cs.berkeley.edu.

B.1. The Mg Interface

```
/*
 *    Mg output interface definition
 *
 */

typedef     double      xyz[3];
typedef     double      rgb[3];
typedef     double      rgba[4];

enum SurfaceType {Constant, Matte, Plastic, Reflective};
enum LightType {Ambient, Distant, Point};
enum BasisType {Bezier, Bspline, Catrom};


/*
 *        Initialization
 * OutputBegin and OutputEnd are the first and last output
 * procedures called by any scene generator.
 * For convenience, they may write any prologue, epilogue, or
 * auxiliary files required by the target system.
 * OutputBegin takes the name of the picture as an argument.
 */

OutputBegin (name)
char *name;

OutputEnd ()



/*
 *        Frame Setup
 */

OutputViewpoint (position, look_at, up, field_of_view, near, far)
xyz position, look_at, up;
double field_of_view, near, far;

OutputResolution(xres, yres)
int xres, yres;


static int nlights = 0;

OutputLight (model, intensity, color, position, direction)
enum LightType model;
double intensity;
rgb color;
xyz position, direction;
```

```
/*
 *        Matrix commands
 */

OutputSave()

OutputRestore()

OutputIdentity()

OutputTranslate(dx,dy,dz)
double dx, dy, dz;

OutputScale(sx, sy, sz)
double sx, sy, sz;

OutputRotate(angle, dx, dy, dz)
double angle, dx, dy, dz;
```

```
/*
 *      Object Geometry
 */


/*
 *              vertex routines
 *
 *      OutputVertex(), OutputVertexNormal(), and OutputVertexColor()
 *      set values for the current vertex.
 *      OutputVertex() increments vcount.
 *
 */

OutputVertexReset()

OutputVertex(v)
xyz v;

OutputVertexNormal(n)
xyz n;

OutputVertexST(s, t)
double s, t;

OutputVertexColor(c)
rgba c;


OutputPolygon (n, v, nflag, cflag, tflag)       /* specify vertices in clockwise order */
int n, v[], nflag, cflag, tflag;

OutputPointsPolygons(np, nv, v, nflag, cflag, tflag)
int np, nv[], v[], nflag, cflag, tflag;

OutputPatch (basis, v)
enum BasisType basis;
int v[];

OutputSphere (radius)
double radius;

OutputCone (radius, height)
double radius, height;

OutputCylinder (radius, bottom, top)
double radius, bottom, top;
```

```
/*
 * Shading interface.  All shading properties stay in effect until overwritten.
 *      Color
 *      Opacity (percent opaque, index of refraction)
 *      Surface characteristics (surface model;
 *          coefficients of ambient, diffuse, and specular reflection;
 *          roughness for specular reflection)
 *      Texture map information
 */

OutputColor(color)
xyz color;

OutputOpacity(percent, refraction)
rgb percent;
double refraction;

OutputSurface(model, ka, kd, ks, roughness)
enum SurfaceType model;
double  ka, kd, ks, roughness;

/*
 *      s,t texture coordinates for parametric or quadric surface
 */

OutputTxCoords(s, t)
double s[4], t[4];

/*
 *      specify texture map(s) for a surface
 *  texture shading model specifies number of textures, Ka, and Kd.
 */

OutputTxSurface(n, ka, kd)
int n;
double  ka, kd;
```