Performance Evaluation for Computer Image Synthesis Systems

Ricki Blau

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

December 1992

Dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of the
University of California at Berkeley

Report No. UCB/CSD 93/736

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.

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.

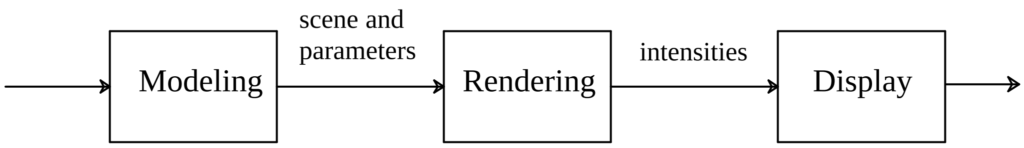


Figure 2.1. Image synthesis pipeline.

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 sub-patches are nearly flat or until they are smaller than some maximum screen size. The resulting sub-patches 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.

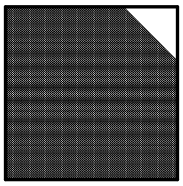


Figure 2.2. Weighted intensity contributions. If the white polygon is closer to the viewer than the dark polygon, a simple algorithm without antialiasing might display a completely white pixel. The most accurate shade for this pixel would be a dark gray that is a weighted blend of the two shades. We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

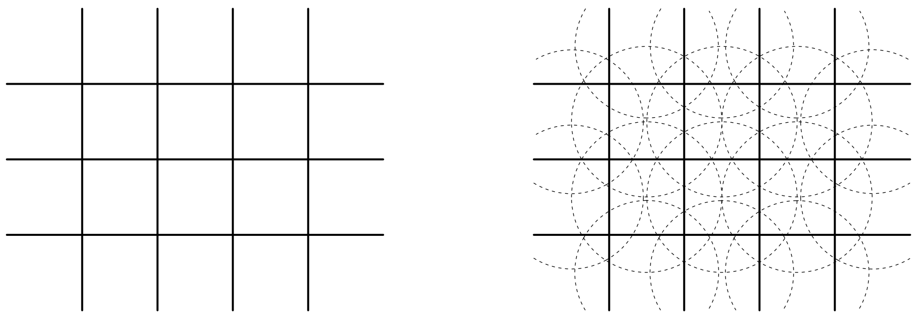
1. Post-processing strategies blur the computed image to make some visual artifacts less apparent. For example, image processing algorithms can smooth jagged edges. Unfortunately, blurring reduces the quality of the image and destroys information. These methods tend to be ineffective with surface details 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

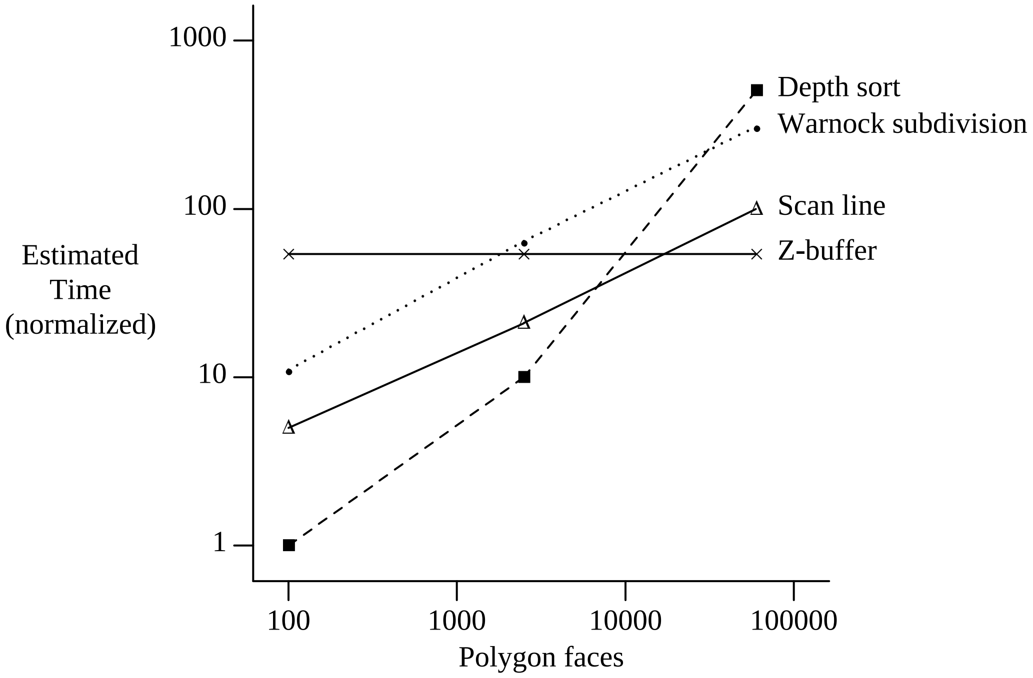


Figure 2.4. Estimated run time for four visible-surface algorithms.

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 inter-reflection 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].

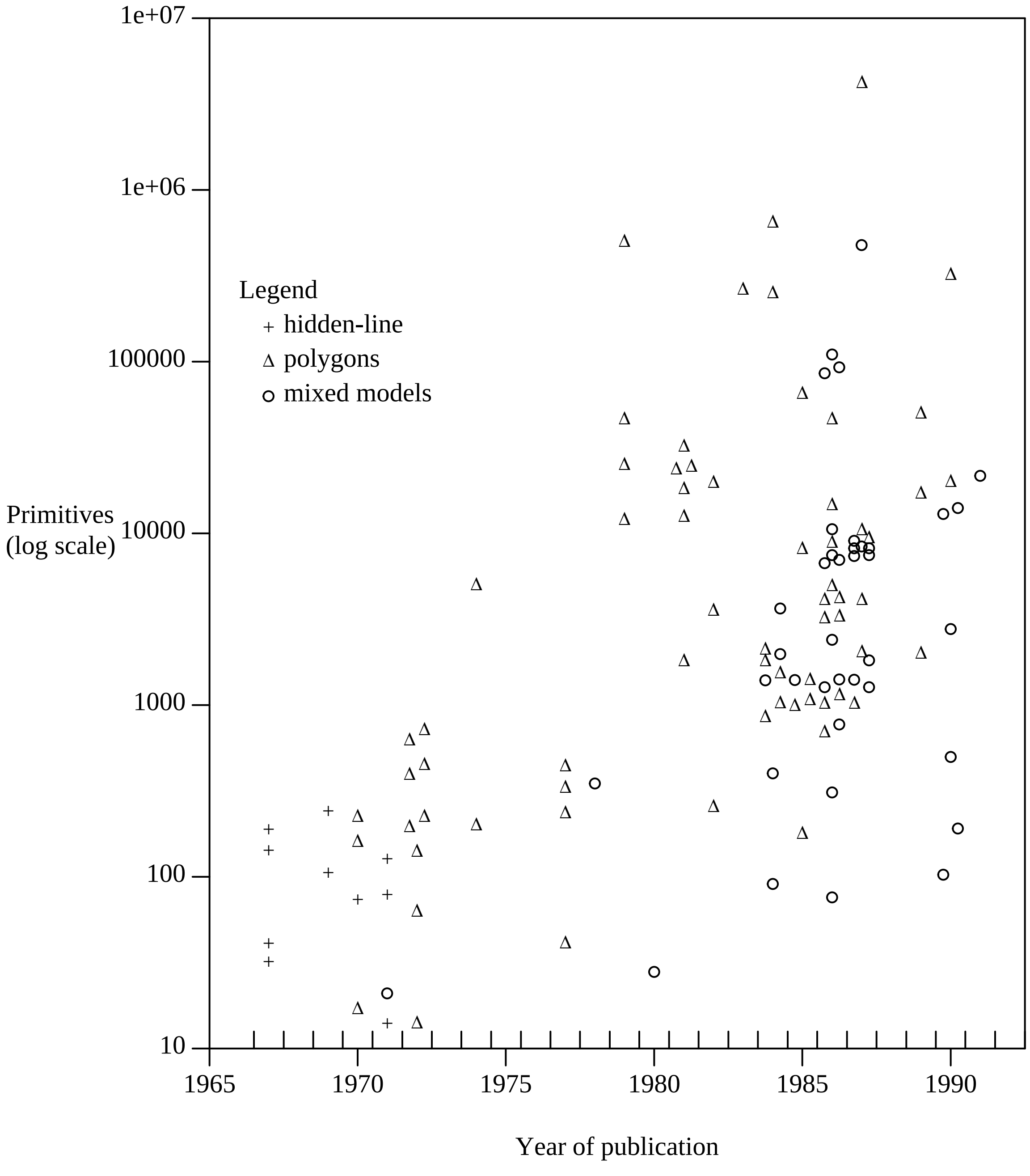


Figure 3.1. Number of geometric primitives. The data describe images published in the computer graphics literature, 1967-1991. Appendix A lists the sources of the data. The data points for a single year are spread horizontally over the corresponding interval. Within one year, the horizontal positions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

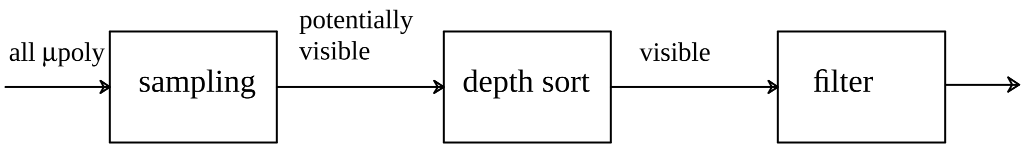


Figure 3.2. Visible surface processing stages.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. Please consult the references above for color reproductions of similar images.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. Please consult the references above for color reproductions of similar images.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. Please consult the references above for color reproductions of the image.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. Please consult the references above for color reproductions of similar images.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. The references in Table 3.3(h) contain color reproductions of another image from the same film.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. Please consult the references above for color reproductions of similar images.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report. Please consult the references above for color reproductions of the image.

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.

Due to copyright restrictions, this image cannot be reproduced in machine-readable form. A black and white reproduction of the image is available in the printed version of this technical report.

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

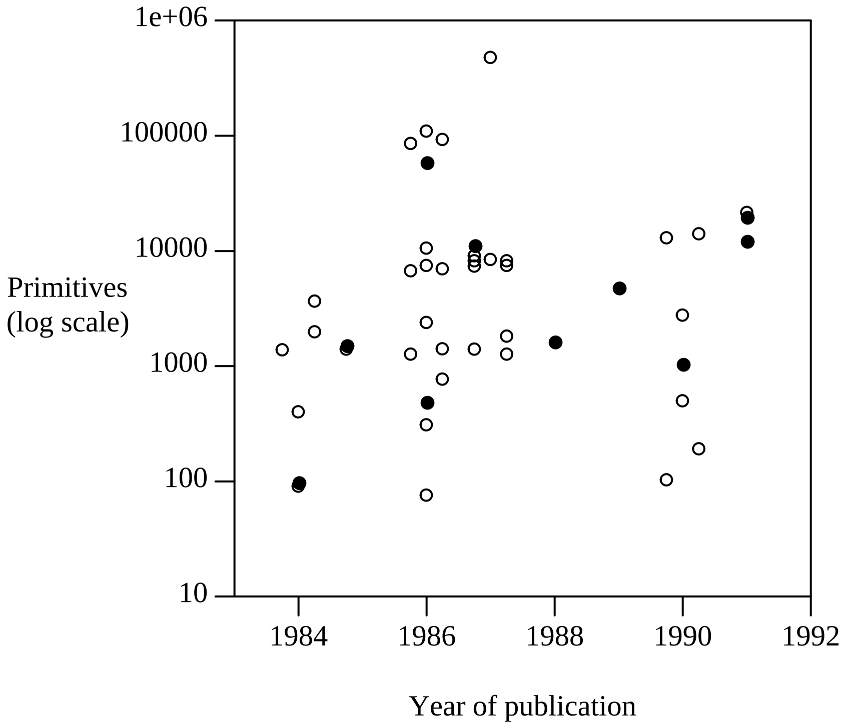


Figure 3.4. Number of modeling primitives. The figure compares the Pixar test suite with other models in the literature. Data points corresponding to the test suite are marked with bullets: ●. The mixed models data from Figure 3.1 are marked with circles: ○.

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

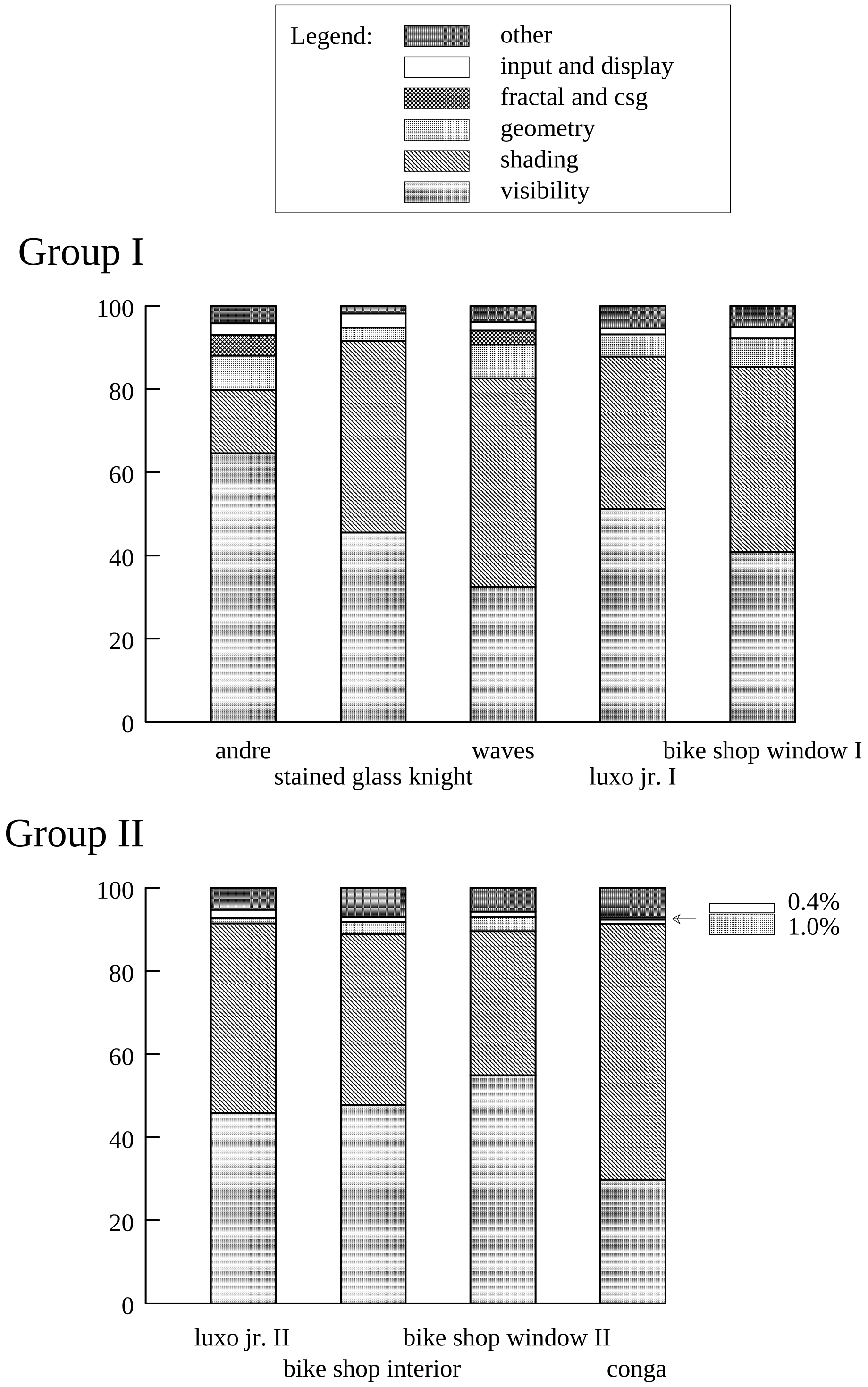


Figure 3.5. Percentage of runtime charged to major rendering activities. The graph uses the same categories as Tables 3.8 and 3.9.

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.

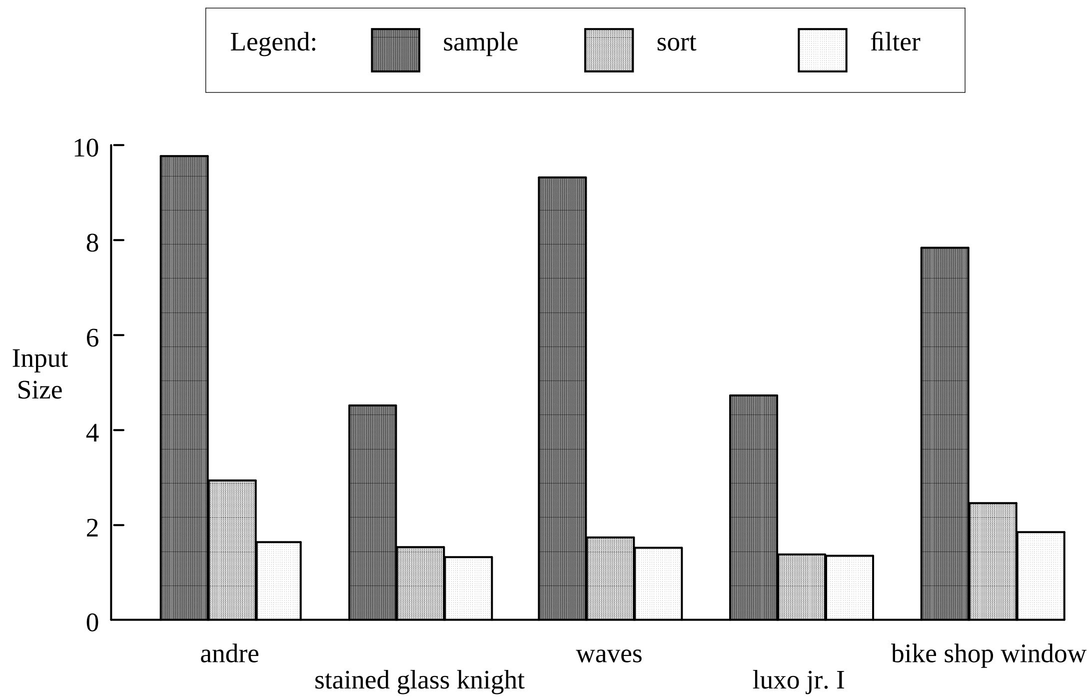


Figure 3.6. Mean input size per sample point. The data describe each of the three stages of visible surface processing for Group I. This graph plots the data in Table 3.13.

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.