

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.



300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

Order Number 8822604

BAGS: The Brown Animation Generation System

Strauss, Paul Steven, Ph.D.

Brown University, 1988

Copyright ©1988 by Strauss, Paul Steven. All rights reserved.

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background ✓
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received ✓
16. Other _____

U·M·I

BAGS

**The Brown Animation
Generation System**

Paul Steven Strauss

Sc.B., Brown University, 1981

Sc.M., University of California, Berkeley, 1982

Thesis

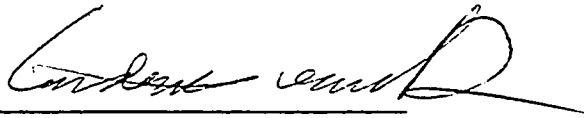
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
at Brown University

May, 1988

This dissertation by Paul Steven Strauss
is accepted in its present form
by the Department of Computer Science
as satisfying the dissertation requirements
for the degree of Doctor of Philosophy

Date

5/11/88

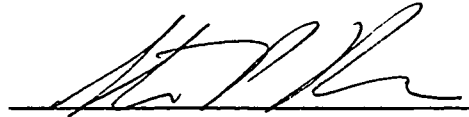


Andries van Dam

Recommended to the Graduate Council

Date

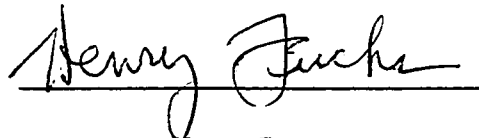
5/11/88



Steven P. Reiss

Date

9 May 1988

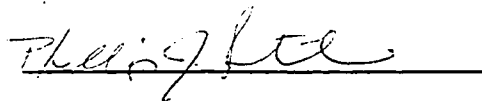


Henry Fuchs

Approved by the Graduate Council

Date

5/23/88



Copyright © 1988 by Paul Steven Strauss

All rights reserved

Vita

I was born in New York City on the twenty-ninth of September in 1959. Five years later, I began attending school, an activity that I have only recently discontinued. I was valedictorian of the Bethpage (Long Island) High School class of 1977. In 1981, I earned my Bachelor of Science degree in computer science, *magna cum laude*, from Brown University.

Upon graduation from Brown, I accepted employment with Bell Telephone Laboratories in Piscataway, New Jersey. I enrolled at the University of California, Berkeley, the following fall, as part of Bell's One Year on Campus program. I completed the Master of Science program in computer science in 1982. My Master's project was the Unigrafix system, developed under the advisement of Carlo Sequin.

I returned to Brown in the fall of 1982 to begin my study towards a doctoral degree. I rejoined Andy van Dam's Computer Graphics Group, attaining the position of Nominal Head. Six academic years later, I wrote this sentence.

Acknowledgements

Working in the Computer Graphics Group at Brown is a unique experience. The group consists of a handful of graduate students, undergraduates, research staff, and the occasional faculty member. These people have two things in common: an attraction to computer graphics and a martyr complex. The former brings people into the group; the latter keeps them there.

The communal nature of the Graphics Group makes work in isolation anathema (and almost impossible), and the development of BAGS is no exception. I would be remiss if I did not at least mention all of the people who contributed to the design, implementation, and testing of BAGS. Here goes (in alphabetical order):

Acknowledgements

v

Rashid Ahmad	David Laidlaw
John Airey	Trey Mattison
Scott Anderson	Barb Meier
Prof. Thomas Banchoff	David Mott
Ronen Barzel	Tom "T-Tom" Mott
Tom Beyer	Hank Obermayer
Jonathan Blumberg	Joe Pato
Robert Blumofe	Kelan Putbrese
Yan-Ming Chang	Rick Rafey
John "Boris" Durand	James Rinzler
Steven Feiner	Evan Schrier
Ron Gery	Ruth "Phizz" Seltzer
Jonathan Goldberg	J. R. Semple
John Hartman	Mike "Mikey" Shantzis
Jennifer Inman	David Sklar
David Johnson	Jeff Vroom
Henry "Mookie" Kaufman	Jerry "Cloth Guy" Weil

Certain members of the graphics group have been so helpful to me they warrant separate mention (at the risk of swelling their heads).

John "Spike" Hughes, on loan from the Brown Mathematics Department, has been a very valuable resource. He has given our group a firm grounding in linear algebra and geometry and has shown us elegant ways of viewing seemingly difficult problems. He has also helped me with innumerable design and implementation decisions.

Mike "Fuzzy" Natkin, Ed Chang, and Eric Bloch have also earned special mention for their willingness to take on more than their share of work. The size of the Graphics Group has varied over the past few years and is currently too small, but these three people have kept our software humming. They are also

primarily responsible for the controlled lunacy that makes the Graphics Lab a great place to spend sixteen hours a day.

I must also express my gratitude to Ben Trumbore, who is now doing graphics research at Cornell University. Ben's dedication and pertinacity are sorely missed at Brown, as is his sense of humor.

My advisor, Andy van Dam, is a legendary figure in computer graphics. He has gone out on many a thin limb to ensure the continued existence of our group. To me he has given inspiration and the freedom to choose my own direction within the field of computer graphics. Andy is one of the main reasons I was eager to return to Brown to pursue a doctoral degree.

Steve Reiss and Henry Fuchs, the other two members of my thesis committee, have provided me with many useful comments and criticisms. Steve has been also helped me with programming and systems issues over the last few years. Although illness prevented Henry from attending my defense, he managed to make time in his 26-hours-a-day schedule to scrutinize my dissertation, for which I am grateful.

Several members of the Brown computer science faculty have assisted me with technical issues in my research. Paris Kanellakis and Jeff Vitter helped me find solutions to computational geometry problems in the caustics algorithms; Jeff led me to the Greene and Yao algorithm, saving me a great deal of frustra-

tion. In a similar vein, Bernard Chazelle of Princeton University (formerly of Brown) acquainted me with the point location algorithm for region searching.

Dan Greene of Xerox PARC helped me greatly with the implementation of the Greene and Yao algorithm. He sent me code for his implementation and was willing to discuss the finer points of mine, which was sufficiently different to create problems.

I thank IBM for their support in the form of a Graduate Student Fellowship. IBM, DEC, and HP are also acknowledged for their continuing support of the Graphics Group at Brown.

The members of the Computer Science Department technical staff have had to cope with supporting a diverse collection of hardware and software, while trying to please too many people with too many different opinions. The staff has done an exemplary job, especially with those special favors I promise I won't request so often from now on. The department's administrative staff also has an unenviable job: guiding the rest of us through the bureaucracy. I cannot thank them enough for all of the help and support they have provided through the years.

Speaking of support, I doubt I would have made it through a single year without the constant support of friends and family. My parents, Malcolm and Ann Strauss, have been extremely patient with me. Despite their fears that I

would turn into a professional student, and with few tangible results, they encouraged me to continue my education. I am also grateful to my brother, Lee, who showed me by example that it could have been worse if I had gone to medical school.

I would like to thank Felicia Gershberg for many things: for not complaining when I worked fifteen hours a day, for putting all my commas in the right places, and for adjusting her life to accommodate mine. I would also like to thank her for everything else. I thank my other friends, including Bob Walsh, Carol Schenk, Linda Sanches, and Karen Seiler, for not asking, "When are you going to be done?" more than once a week for six years and for not fainting when I was finally able to answer.

*P.S.S.
Providence, RI*

Table of Contents

Chapter 0: Introduction	1
Characteristics of Animation Systems	1
The Problem	5
Thesis Contributions	7
Thesis Overview	11
Chapter 1: Related Work	13
Production Systems	14
ASAS	14
BBOP	15
Hanrahan and Sturman	15
Anima II and ANTTs	16
MIRANIM	17
TWIXT	17

S-Dynamics	18
Pacific Data Images	18
Reyes	19
OSCAR	20
Research Testbeds	21
Whitted and Weimer	22
Hall and Greenberg	22
GRAPE	23
FRAMES	23
UNC	24
Summary	25
Chapter 2: An Overview of BAGS	26
The Problem	26
Architectural Highlights	29
Terminology	30
Modeling	34
Object Modeling	35
Scene Assembly and Animation	39
Rendering	42

<i>Table of Contents</i>	xi
Post-Production	45
Summary	46
Chapter 3: The SCEFO Language	47
Animation Languages	47
Other Languages	49
SCEFO	50
Object Creation	50
Object Hierarchies	56
Action-Based Changes	58
Interpolation	63
Action Hierarchies	64
Object Communication	66
Other Features	69
Extensibility	74
Summary	76
Chapter 4: Rendering	78
The BAGS Rendering Process	80
The Lighting Model	83
Other Models	85

The BAGS Model	92
Implementation	107
Polygonal Approximation	110
Ray Tracing	113
Speed Optimization	117
Antialiasing and Caustics	120
Chapter 5: Triangle Tracing	121
The Aliasing Problem	121
Related work	123
The Triangle Tracing Algorithm	127
Comparative Analysis	133
Algorithm Accuracy	133
Performance	138
Triangle Subdivision	144
Summary	152
Chapter 6: Caustics	154
The Problem	154
Related Work	156
Overview of the Algorithm	159

<i>Table of Contents</i>	xiii
Visible Regions	160
uv Coordinate System Mapping	161
Region Overlap Processing	163
Tracing Rays from Light Sources	166
Region Searching	167
Results	168
Complexity Considerations	170
Summary	174
Chapter 7: Conclusions	176
Contributions	176
Future Work	178
Interface Programs	178
Object Classes	179
Rendering	179
Appendix A: A SCEFO Reference	181
General Comments	182
Object Creation Statements	183
Change Statements	186
Action and Apply Statements	191

Miscellaneous Statements	194
User-Defined Additions	197
Appendix B: The BAGS Lighting Model	200
Appendix C: Triangle Tracing Code	220
Data Structures	221
Mainline	222
Ray Tree Comparison and Creation	223
Computing Boundary Rays	225
Triangle Subdivision	227
Triangle Shading	234
References	236

List of Figures

1. CSG union, intersection, and difference operations	33
2. Complex objects created with CSG	38
3. Templates and instances, and rendered image	54
4. Construction of a rivet object with CSG	56
5. A view camera modeled with CSG	56
6. A simple object hierarchy using groups	57
7. A sample change statement	60
8. Translation graph of the previous example	61
9. Time graph of the motions in the previous example	62
10. SCEFO action and apply statements	65
11. Using a field within a change	68
12. Changing an object within a context	71
13. A change statement using arithmetic expressions	72

14. A simple SCEFO for-loop	73
15. Using prefixing to avoid name duplication	74
16. A SCEFO variable	75
17. BAGS rendering block diagram	82
18. Rendering comparison. Clockwise from upper left, software depth buffer, hardware depth buffer, ray tracing, scanline	84
19. OFF interface between renderers and object classes	85
20. Lighting geometry	86
21. Lambert lighting	87
22. Phong lighting	88
23. Phong lighting of an inconsistent object	90
24. Torrance-Cook lighting	91
25. BAGS lighting	94
26. Spheres of varying smoothness: 0 at upper left, 1 at lower right	96
27. Plastic sphere, metallic sphere	97
28. Varying transparency: 0 at upper left, 1 at lower right	98
29. Varying index of refraction, from 0.1 to 1.6	99
30. Comparison of Fresnel formula curve and BAGS approximation	102
31. Geometric attenuation curve and adjustment curve	103

32. Pattern mapping examples: color mapping and normal mapping	109
33. Mapping a pattern from a stored bitmap onto a sphere	110
34. Generic ray tracing procedure	114
35. Tracing secondary rays	115
36. Ray tracing with shadows	115
37. CSG operations along a ray	116
38. Aliasing in ray tracing	122
39. Tracing vertex rays through half-pixel triangle corners	128
40. A pair of boundary rays on a triangle edge	129
41. Recursive binary search for boundary rays	130
42. Subdividing a triangle by joining boundary points	132
43. Single-sample ray tracing vs. triangle tracing, and details	134
44. Adaptive supersampling vs. triangle tracing, and details	135
45. Linear boundary approximation leading to sub-pixel error	136
46. Pattern-mapped object: one ray per pixel (top), adaptive supersam- pling (bottom left), triangle tracing (bottom right)	139
47. Homogeneous pixel ratio for two sample images	141
48. Adaptive supersampling for a simple two-object pixel	142
49. Simple image rendered with varying boundary angle ratios	145

50. A simple case of partitioning ambiguity	146
51. Solid partition edges at triangle corners	147
52. Graphical display of partitioning test program	149
53. Comparison of triangle partitioning methods	151
54. A caustic interaction undetected by conventional ray tracing	156
55. Parametric mapping of a square planar object	162
56. Parametric mapping of a sphere, using two zones	162
57. Region overlap caused by reflection	164
58. Dividing a light source cube face into triangles	166
59. Caustics from one plane to another	169
60. Caustics from an off-camera object	170
61. A large object with respect to the camera, but small with respect to a light source	172
62. Diverging secondary rays; triangle subdivision by median	173
63. Caustics from two planes to a third	175

0. Introduction

Computer image synthesis can be defined as the sum of two components: *modeling*, the specification of data to be viewed, and *rendering*, the creation of an image depicting the modeled data. If the model varies over time, and a series of images are produced, the result is computer animation. This definition encompasses many disparate forms of computer animation, such as algorithm animation and computer-assisted cartooning. One particular form of computer animation is *three-dimensional*, in which the models are three-dimensional objects, and the task of rendering is to derive a two-dimensional representation of the model for display.

Characteristics of Animation Systems

Several properties can be used to distinguish animation systems described in the literature. The most important characteristic is the *intended set of users*. Most systems can be placed into one of two categories using this criterion:

production systems versus *research testbeds*. Systems in the former category are used typically by computer graphics production houses to create animations for advertising and entertainment. Users of these system range from skilled programmers to artists with little or no computer training. Graphical interaction and other non-programming interfaces are used to accommodate this varied audience.

Because their users are often motivated by intense deadline pressures, production systems must have a relatively fast turnaround time from concept to finished product. Speed is usually achieved by limiting the modeling and rendering methods to a minimal set, which can then be highly optimized. It is often the case that the need for high performance supersedes the desire to minimize the user's time and required expertise.

Research testbeds, on the other hand, are designed to allow skilled programmers to experiment with new modeling and rendering techniques. These systems must be modular and flexible, allowing new code to be inserted for easy testing. They must also be extensible so that the results of successful experiments can be added. Researchers will put up with almost any interface, so testbeds usually are not conducive to producing animation. Furthermore, testbeds are often designed with a particular domain in mind, such as molecular simulation or articulated figure animation. In these cases it may be very

difficult or even impossible to produce other types of animation.

The means of *model specification* is a second identifying characteristic. Using *interactive, visual* tools to define a model is probably the easiest and fastest way to produce animation. It is also the easiest method for non-technical users to learn; anyone with spatial design abilities can build a model with a graphical tool.

Programming languages, including those designed explicitly for animation, are harder to use because they require computer programming experience. They are used most often to accomplish specific modeling tasks, especially those that are inherently procedural, such as complex interaction between objects. The programming language approach also facilitates system extension and testing.

Script languages lie between the extremes of graphical interaction and programming. A script language is a problem-directed animation description format, differing from conventional programming languages in several ways that make them easier to use. The data structures correspond to objects in the scene, not abstractions. Control flow constructs are often limited to linear sequences, since that is how animation appears. The result is a language that does not require a high level of programming expertise, but offers more precise control and more power than interactive techniques.

Another category of model specification tools includes *simulation* and similar rule-based approaches. A set of rules, such as physical laws, and initial conditions is input to the simulation. Other variables representing changes to the system may be introduced after the simulation commences. The result of the simulation is a description of the changes made to each component object; this description can be in one of the other forms of model specification, such as a script language. Therefore, this approach can be thought of as a “front end” to the modeling task. Simulation is very useful for special animation needs, but does not address the general motion modeling problem.

Animation systems can also be distinguished by *animation paradigm*. Some systems use *keyframes*, so the user must specify properties of objects in the animation at certain instants, and the computer interpolates intermediate frames with some mathematical function. While this approach works well for two-dimensional and simple three-dimensional models, it is infeasible for complex animations. When two simple changes, such as a translation and a rotation, are applied to the same object, it can be difficult for the user to specify the object's position at any particular frame. *Action-based* (or *track-based*) systems remove this difficulty. Each change is described as a separate action, and the actions are then composed. While each action is itself keyframed, the resulting complex change is not.

The last characteristic described here is an implementation issue: the storage interface between the modeler and the renderer. Most systems store the model in a form that is acceptable to a particular renderer. Since most common rendering methods require polygonal approximation of objects, polygons are very popular storage primitives. While this approach results in an increase in rendering speed, it introduces error and restricts both the range of possible rendering techniques and the user's creativity. If the model is stored as renderer-independent primitives, rendering is not limited to any one method, and the user does not have to be concerned with any particular representation.

The Problem

Computer graphics today is predominantly two-dimensional, static, presentation graphics. Advances in computer hardware have now made animation possible in high-performance workstations, making three-dimensional computer animation the new frontier. With this trend, new groups of users are acquiring the means to create animation. Educators can use dynamic images to accompany texts or demonstrations. Scientists in all disciplines can take advantage of new ways to visualize data for analysis. Artists have a new, unfamiliar medium to explore.

The upshot of this expansion is that computer graphics systems can no longer be built with only graphics experts in mind. Non-technical users — i.e.,

those with little or no computer graphics background — need a system that allows them to reach their goals without having to become expert in another field. In fact, they should be able to use whatever interface feels most comfortable. A scientist may feel confident in his or her ability to define models based on equations or data points, while an artist may want to interactively model interesting shapes. There should not be any restrictions inherent in a graphics system that prohibit the use of a comfortable interface.

Rendering quality is another important interface issue. At times a stylized, unrealistic representation of a model is more appropriate, while at other times realism is important. The quality of the visualization should be chosen by the user, not restricted by the system. This criterion is especially important in light of hardware and software speed limitations. If a realistic image takes a long time to compute, a user may want several quick, lower-quality, preview renderings beforehand.

While the needs of non-technical users must be considered, computer graphics researchers also need a suitable environment for their work. The state of computer animation is constantly improving; a system should be flexible and extensible enough to incorporate new techniques. A stable existing system to augment is a valuable research testbed, while a system that accepts only one type of model, or requires the model to be viewed in a limited set of ways,

hampers experimentation. Similarly, a system that is fine-tuned to run well on one particular type of hardware is one step away from obsolescence.

All of these goals traditionally have been treated as being mutually exclusive; this is evidenced by the lack of systems designed to achieve all of them. The apparent contradictions between ease of use, power, speed, and extensibility have caused system designers to sacrifice one or more of these desired properties in favor of others, instead of incorporating all of them. The aim of the work presented in this thesis is to accomplish all of these goals to an acceptable level.

Thesis Contributions

The primary contributions of this dissertation are the following.

- BAGS, a framework for animation production and research. Unlike other animation systems, BAGS combines the flexibility and extensibility of a research testbed with the utility of a production system.
- SCEFO, a script language that is powerful enough to serve as the interface between modeling and rendering in BAGS, and yet may be used by non-programmers in defining animations.

- Validation of an object-oriented approach in certain aspects of animation system design.
- A flexible lighting model that, unlike other models, translates intuitive input parameters into realistic lighting effects.
- New extensions to the ray tracing rendering technique that provide area-sampling solutions to the aliasing and caustics problems.

BAGS, the Brown Animation Generation System, is designed with non-technical users in mind but is also a useful testbed for state-of-the-art graphics research. We will show how the system reconciles the conflicting goals of power, ease of use, and flexibility.

The interface between modeling and rendering in BAGS is a textual language called SCEFO. A SCEFO script contains a description of the objects in a scene, including lights and cameras, along with action-based changes to those objects over time. SCEFO is designed to be an “assembly-level” language: it is high-level enough to allow experienced animators to specify it directly, and low-level enough that it can be produced as the output of interactive graphical editors, simulations, and other front-end programs. The language also permits advanced users to add C-language extensions for additional power and control.

Because SCEFO is a conceptual representation, it does not require objects to be stored in any particular format and can be used as input to any rendering

method. The model can be chosen to best suit the given application and rendering format. The renderer-specific knowledge is stored with the objects, as is information about how to transform objects. This design is an example of an object-oriented graphical approach. “Object-oriented” in graphics systems means that the modeled objects found in animated scenes are treated as abstract constructs and grouped into classes. Associated with object classes are methods (procedures) that affect an object in the class when prompted by the reception of a message. The methods are inherited by all objects in the class. Use of an object-oriented paradigm in this context adds consistency and extensibility to BAGS, making additions of new rendering techniques and object classes easier.

The design of BAGS also provides for extensibility in many other respects. New modeling front ends can be added easily, as long as they produce SCEFO output. Interpolation methods and object transformation operators can be created by the user and can be incorporated into the system if desired. Additionally, the lighting model used in BAGS is designed to be flexible enough to allow testing of other lighting models.

BAGS has proven to be a useful production system, having been used to produce animations of architectural studies[9], molecular simulations[1], mathematical visualizations[47,14], and general animation[39,40]. However,

BAGS's value as a research tool is even more significant. Several research advances have resulted from the development of BAGS, including an improved camera motion interpolation model[46] and a robust polyhedral constructive solid geometry algorithm[54].

Another advance in research is the BAGS lighting model, which translates a small set of surface material parameters into realistic lighting effects. The parameters are intuitive, so that a user with no materials research training can describe a desired surface. The model also allows a user to modify any of the parameters or computed values for more realism, stylized lighting, or other effects.

Yet another important research contribution facilitated by the BAGS framework is a new technique for rendering highly realistic images. This technique consists of two important extensions to an existing rendering method called *ray tracing*. The first extension is a remedy for the aliasing (staircase pixel) problem, which is an artifact of the point-sampling nature of the conventional ray tracing algorithm. The algorithm provides area sampling for correct antialiasing, but requires only the conventional object intersection tests found in standard ray tracing. The second extension, which is similar in design to the first, incorporates into an image the phenomena called *caustics*, which are the reflection and refraction of light from one object to another. In many cases, this

technique dramatically improves the realism of the rendered image.

Thesis Overview

The next chapter describes work on related three-dimensional animation systems. This provides some context for critical analysis of the design of BAGS, which is presented in Chapter 2. In that chapter we describe primarily those design decisions that allow BAGS to succeed as an animation system, and only touch on those topics common to most systems. Chapter 3 is a description of the SCEFO language and its uses for modeling. Again, the emphasis is on language features that are unique or that provide needed power or flexibility to BAGS.

Chapter 4 examines the rendering portion of BAGS, including the range of available rendering techniques and the lighting model. The chapter ends with a description of the ray tracing rendering method, which is the basis for the new rendering techniques described in Chapters 5 and 6. Chapter 5 describes a way to solve the aliasing problem for ray tracing, while Chapter 6 examines a method for adding caustic effects to the rendering process. In Chapter 7 we present conclusions and possibilities for future work.

Appendix A is a syntax reference for the SCEFO language. Appendix B is a summary of the lighting model presented in Chapter 4. Appendix C contains

some of the code used to implement the triangle tracing algorithm described in Chapter 5.

1. Related Work

In the next chapter, the advantages of the architecture and features of BAGS are described. In this chapter we provide a basis for comparison by examining several animation systems that are described in the literature or available commercially. The systems are divided into two categories based on the intended set of users. The systems that are primarily production systems are in the first section, while those that serve as research testbeds are in the second. Overlaps between the two categories are noted in the system descriptions.

We will attempt to use the other criteria developed in the first chapter to further characterize the systems. Unfortunately, the literature is often incomplete in this regard. If a description in this chapter omits mention of certain features, such as interactive interfaces or model specification, it is because there is no mention of them in the literature. The result is therefore a summary of the highlights of each system, although certain obvious drawbacks are noted

when appropriate.

Several advantageous features of these systems are also present in BAGS. The basic architecture of BAGS was designed before many of these systems were; their development occurred in parallel.

Production Systems

The systems in this section are for use primarily as production systems. They are designed to produce animations for any of a variety of applications, whether commercial, entertainment, or scientific visualization. A common theme in most of these systems is an emphasis on minimizing rendering time and also on producing images that “look good,” whether or not these goals involve extra effort on the part of the animator.

ASAS

ASAS[67] is a LISP-based language for programming animation. Objects to be animated are defined as *actors*, each with a polygonal data definition and an associated piece of animation code. The code for each actor is executed once per frame, to determine the effect of that actor in the animated scene. Actors may communicate with each other through message-passing, making it possible to program objects that interact. ASAS is useful for modeling procedural

animations, but is difficult for non-programmers to learn and use.

BBOP

The BBOP system[70] is a keyframed production system developed at NYIT. The BBOP input model is a set of hierarchical, polyhedral, jointed objects. The animator has control over transformation matrices at each of the joints. An interactive program allows graphical specification of these transformations at each keyframe. A motion editor can be used to define interpolation curves.

The keyframing setup makes a model difficult to modify. Changing the duration of one action may require editing of several neighboring keyframes. In general, this approach makes complex motions almost impossible to specify. Dick Lundin[58] describes a way to add a motion simulation capability to BBOP. The animator, who must be an experienced programmer, can specify simulation routines that produce BBOP input.

Hanrahan and Sturman

Hanrahan and Sturman[44] describe a program that combines interactive and programming language approaches to model specification. Scenes are constructed using a geometric surface modeling language. An interactive program

reads the language and allows a user to change the model in real time. Animation is defined by interpolating keyframed model parameter sets that are used in object motions. The interactive nature of the system greatly aids productivity, but the restrictions of keyframing often overshadows this advantage.

Anima II and ANTTTS

Anima II is an animation production system designed to accommodate animators, educators, and artists[41]. It is intended to balance the ability to produce high-quality, complex animation with production-level efficiency. The Anima II user interacts with a graphical polyhedral modeling program to create objects to animate. The user then enters (textually) an animation script, written in the Anima II language. This language contains primitive keyframed operations on objects in the scene.

The ANTTTS system described in[25] is an attempt to improve on Anima II, which had implementation limits on model complexity and rendering quality. ANTTTS still views a model as a collection of polygons, but it is able to handle a large number of polygons. The model specification is also in the form of a script language.

MIRANIM

Nadia Magnenat-Thalmann and Daniel Thalmann's MIRANIM[59] is an artist-oriented animation production system. It is designed to allow artists to define animated models without any programming. However, its interactive editing program, ANIMEDIT, requires the user to position and move objects by entering textual commands; there is no graphical interface. The polyhedral objects to be animated are built outside the editor, and are stored in files. The MIRANIM system can be extended by programming in the CINEMIRA-2 language, which is an extension of Pascal. For example, a transformation that moves an object along a particular path can be programmed and then accessed from within ANIMEDIT.

TWIXT

Julian Gomez's *twixt* animation system[34] is action-based and interactive. Objects are specified as polygon meshes or bicubic patches and may be placed into hierarchies. An interactive program with textual input and graphical display allows the animator to define the changes to the objects over time. When a useful input device such as a knob bank is available, *twixt* allows it to be used for value input. The text input language combines modeling and animation primitives with rendering directives.

S-Dynamics

Symbolics's S-Dynamics[77] is a production system available to users of Symbolics Lisp Machine workstations. It is primarily an interactive animation editor that allows a user to define object motion graphically. The S-Geometry modeling program[78] is an interactive polyhedral modeler for creating objects to be animated.

The animation model in S-Dynamics is action-based, so that the user defines each subsequence (action) of an object separately, and the subsequences are composed when the scene is animated. Each of the subsequences, such as a rotation, scale, or translation, has keyframed endpoints. Interpolation is done by one of several available methods, or can be defined by sketching the interpolating curve. For extra flexibility and extensibility, advanced users of the system can write LISP functions to be used in the animation.

Pacific Data Images

Pacific Data Images (PDI) is one of the leaders in commercial computer animation. The system in use at PDI[15] is designed for efficient production of animation for entertainment and advertising. Modeling is accomplished with a C-based language; it is more like a programming language than a script format. Most modeling tasks are performed by special-purpose programs that generate a

set of polygons representing objects in the scene. Other programs are used for adding special effects, such as perturbing surface normals to produce moving highlights. An interactive editor is used for the difficult task of lighting the model. A range of renderers allow real-time previewing or high-quality image production.

Reyes

Reyes[21] is a production animation rendering system in use at Pixar. It is designed to accommodate very complex and diverse models and is optimized for speed. In Reyes, the model is usually defined as surface patches, which are “diced” into primitives called *micropolygons*. The micropolygons are shaded, with texture being added if desired, and the results are added to a depth buffer. The depth buffer stores the front-most polygons, thereby determining which surfaces are visible. All polygons visible at a pixel are stored in the buffer, with a measure of how much of the pixel each polygon covers. This technique results in antialiased images.

A “back door” into the depth buffer allows other rendering methods to contribute to images, although it is not clear how a user would specify this. In fact, there is little mention in the literature of the modeling aspects of the system in use at Pixar. Most of their effort is concentrated on rendering speed and other

methods for fast animation production. One important technique is *image compositing*, which allows several images to be combined to produce a single image, much like optical processes used in conventional filmmaking.

OSCAR

OSCAR[57] is a system designed at General Electric specifically for the animation of computer-generated industrial analysis experiments. Although it is limited in its utility, it incorporates several important general design principles, such as an object-oriented approach[56]. OSCAR uses this approach to direct the actions of objects in over sixty classes, including scenes, cues (timing information), actors (geometric objects), cameras, and lights.

The actors are created with any of several compatible polyhedral modeling programs. Objects called *liaisons* serve as interfaces with the rest of the system, performing data conversions as necessary. Once the actor is known to the system, any of several messages can be sent to it to change its position or color. Animation is initiated by sending timing messages to *cues*, which in turn can cause objects to move along predefined paths. A variety of rendering methods can be specified, including real-time wire-frame preview and higher-quality shaded display.

OSCAR has its own animation language interface. The language allows the user to define initial conditions such as position and color, to specify how the animation should proceed, and how to render the scene. It is designed to be high-level enough for direct specification, and OSCAR includes interactive programs that produce files containing transformation specifications.

OSCAR is a good example of how an object-oriented approach can be used effectively in animation. Because the system is designed primarily for industrial analysis, however, it is limited in function. Actors are rigid polyhedra that can be transformed only at joints. Motions are interpolated only linearly, and methods for defining motion are limited to a predefined set. However, these are just limitations of the implementation, not the design.

Research Testbeds

This section describes systems designed for use by graphics researchers in developing new modeling and rendering techniques. Of course, these systems also allow for the production of animation, but that is not their primary function.

Whitted and Weimer

The testbed described by Whitted and Weimer[84] allows them to test different polygon shaders. It is designed to accommodate any objects that can be approximated by convex polygons. Objects are described by the user in terms of vertices, edges, and polygons. The system scan-converts the objects' polygons into a *span buffer*, which holds information about objects intersecting a scanline. This information is given to a shader, which produces pixels in the final image. The system was intended to be extended eventually to include interactive model editors and an animation language.

Hall and Greenberg

Hall and Greenberg developed an image synthesis testbed[43] for the exploration of models to improve image realism. Modeling in the system consists of a set of programs that generate geometric objects defined by bounding surfaces. An *environment builder*, another set of routines, is responsible for positioning the objects and defining their surface characteristics.

The system is implemented with only ray tracing rendering, although the authors claim that any rendering algorithm could be added. This is one of the few examples where the rendering and modeling are decoupled enough to allow this extension, but it has not been implemented. The system has been

successful as a testbed, in that it has been used to develop an improved lighting model.

GRAPE

The GRAPE system[62] is a testbed for modeling primitives and rendering techniques. The components of the system are implemented as procedural nodes which are connected into directed acyclic graphs. Communication between nodes is in the form of standardized “appel” (appearance element) and synchronization messages. This approach has several advantages. The message scheme provides a great deal of flexibility, since any node conforming to the standards can be inserted easily. Standardization also means that the system is not biased towards any rendering method.

This system is intended for experienced researchers with programming skill. In addition, it suffers from inefficiency due to the overhead of converting data to standardized forms and the large number of procedure calls needed to transfer data between nodes. Although the system is useful for experimentation, the speed drawback prevents its use for production work.

FRAMES

FRAMES[66] consists of a collection of tools, each of which is a program. The tools are connected via linear UNIX* pipes. Input to the tools is specified in a text file containing commands for each of the programs in the pipeline. The programs can be considered “filters” that change the input in some way and output the result.

Modeling is performed by a set of filters that generate polygonal objects. There are no interactive front-end programs in the system, only command-directed generators. All other aspects of the modeling and rendering, including transformations, shading, camera placement, visible surface algorithms, and display, are specified by commands to the appropriate filters.

The pipeline system is designed to be flexible so filters can be swapped in and out. However, the data flow requires a polygonal model. It is difficult to determine how an alternate model, such as one required for ray tracing, could be included. Furthermore, the nature of the pipeline prohibits feedback loops and other non-linear control sequences.

UNC

A system under development by Bergman, Fuchs, et al., at the University of North Carolina[8] has a unique approach to rendering a polygonal model. An

*UNIX is a trademark of AT&T Bell Laboratories.

interactive, adaptive rendering system provides successively higher-quality levels of rendering, starting with vertex and edge indicators, and progressing to antialiased, Phong-shaded polygons with shadows.

Summary

Each of the systems described above has been designed to resolve the power/friendliness/flexibility continuum by focusing on a small range of the spectrum, although not all systems resolve it in the same way. In addition, most systems are locked into one method for modeling, such as polygonal approximation, or for rendering, such as a depth buffer. This is done primarily to enhance performance but prevents the systems from expanding or improving when new techniques are invented.

2. An Overview of BAGS

This chapter is an overview of the BAGS architecture. First, we state the goals of animation system design, to provide a basis for the explanation of our solutions. We then introduce some terminology that will be used in our descriptions of the system. The last two sections are devoted to the exploration of the modeling and rendering components of BAGS, with emphasis on the design decisions that allow the system to achieve the desired goals.

The Problem

A general animation system should provide the means to do both production work and computer graphics research, providing features for non-technical animators and for graphics experts. The three most important attributes of such a system are *usability*, *flexibility*, and *extensibility*.

Usability includes providing an appropriate interface to each type of user. An artist may require an interactive, graphical program to define a model; a physicist may need the ability to simulate laws of nature; and a mathematician may feel most comfortable specifying complex equations. An animation system should be able to accommodate all of these users.

Usability also means providing reasonable default actions to the user, so that the computer does the hard work. For example, an animator should not have to spend much time to set up realistic lighting for a scene; this should be done automatically. Furthermore, advanced users should be able to override defaults when desired.

Flexibility in an animation system requires variety in modeling and rendering. For example, each of the modeling interfaces described above should be available. Other aspects of modeling, such as available objects, operations on objects, and interpolation methods, should be varied as well.

A variety of rendering techniques is necessary to solve the time/quality trade-off. High-quality images often take too much time to render, wasting the user's time and leaving the user without any feel for animation timing. Lower-quality rendering methods are typically faster but are inappropriate for final products. If several methods are available, an animator may choose one based on the current need for quality and the amount of time available.

Extensibility is perhaps the most important attribute. If a system is easily augmented, none of its inadequacies is too harmful. In an animation system, extensible areas should include objects, operations on objects, interpolation methods, modeling programs, and rendering techniques. Some of these areas, like objects and interpolation methods, should be extensible by a user, while others, such as rendering techniques, require the work of a system programmer.

No other system in the literature attempts to achieve these goals to the extent that BAGS does. In particular, modeling in most systems is restricted to a particular paradigm. For example, the PDI[15], ASAS[67], and BBOP[70] systems all require the animator to program an animation in a high-level language, while MIRANIM[59] and *twixt*[34] couple a simple, interactive, terminal-based interface with a visual display.

Two systems stand out as being somewhat more general in the area of modeling specification. Symbolics's S-Dynamics[77] provides an interactive graphical interface as the primary modeling tool and also allows an experienced programmer to develop high-level systems to run on top of S-Dynamics. OSCAR[57], on the other hand, uses a script language approach. A script may be written directly by an animator or may be generated as the output of an interactive program.

Full flexibility in rendering also is not attained by available systems. Most systems provide a few different shading methods (faceted, Gouraud, Phong), but only one or two rendering methods (wire-frame, z-buffer). A particular feature missing from all of these systems is the ability to render the same model with a polygonal-based renderer and a ray tracer. Ray tracing, a high-quality rendering technique, works best with procedural representations of objects rather than collections of polygons. The reason for the lack of compatibility is that almost all of the systems store objects in terms of approximating polygons.

The remainder of this chapter explores the ways in which BAGS solves these problems. A brief description of the architecture of the system is followed by a glossary of terms commonly used in BAGS. The remaining sections describe the modeling and rendering components of the system.

Architectural Highlights

BAGS consists of about ninety software packages and about ninety executable programs, all written in the C language for the UNIX operating system. The software packages compose the bulk of the system, while the programs are mostly drivers. The package-oriented approach is the result of an effort to reduce code duplication, increase reusability, and maintain modularity. All code adheres to a set of software standards[71] that allows the packages to be used in

tandem without conflict. Part of the standard calls for documentation, so that online manual pages are available for all system components.

The system is divided into four main areas:

- Object modeling
- Scene assembly
- Image rendering
- Image post-production and recording

Terminology

Many terms are used interchangeably in computer graphics literature, often to the complete confusion of the reader. We include this section to avoid this problem, at least within the scope of this thesis. These definitions are used within BAGS; they may be different or even contradict those used in other papers or systems, but we will try to use them consistently here.

An *animation* is the product of modeling and rendering; it is the final, viewable sequence of moving images. It may be represented in any of several media, such as film, video tape, or even a series of changing images on a graphics display device.

An animation is constructed from one or more *scenes*, which are analogous to scenes in a movie. Each scene lasts for some duration, in which motion or other changes occur. Scenes may be intercut, as they are in motion pictures, during the editing process. However, it is natural to think of each uncut segment as an individual scene, regardless of the final editing.

Scenes can be broken down further into *frames*, each of which represents the state of the model at an instant or short interval in time, analogous to a frame of film or video. A static rendering of one frame of a scene is called an *image*. The sequence of frame images when displayed in succession simulates motion, as in conventional (non-computer-generated) animation.

An *object* is a potentially visible component of a scene. In any animation system certain objects are *primitive*; these are the objects that the system can operate on at the lowest level. In many systems, polygons, polyhedra, or polygonal meshes are the only primitive objects. Others also allow bicubic patches, surfaces defined by parametric cubic equations. In BAGS, there are several classes of primitive objects. Some are canonical solids, such as spheres, cubes, cylinders, and cones, while others are solids that require some data for their definition. In some uses, the word “object” encompasses cameras and lights, which are discussed below.

Many systems allow primitive objects to be modified and combined in various ways to produce more complex objects. BAGS uses a common combination method called *constructive solid geometry* (or CSG), the application of the Boolean *union*, *intersection*, and *difference* operators to solid objects. The result of these construction operations on two cubes is shown in Figure 1. BAGS also allows the use of the non-linear *bend*, *taper*, *twist*, and *wave* unary deformation operators formulated by Barr[5].

A *camera* represents a view of a scene. It has a position and orientation in space, and other attributes that define the *film area*, a two-dimensional rectangle upon which the image of the scene is projected. There is typically only one active camera at any frame, although this is not a restriction. A stereoscopic renderer could have two active cameras, for example.

There are three basic types of *lights* that provide scene illumination. *Ambient* lights illuminate all surfaces evenly, regardless of orientation. A *point* light is an infinitesimal illumination source at a given location in space. It illuminates outward from the point in all directions. A *directional* light can be thought of as a point source that is an infinite distance away. It illuminates along rays parallel to a given direction vector. Other types of light sources, such as flood lights, spot lights, and area lights, can be created from the basic types.

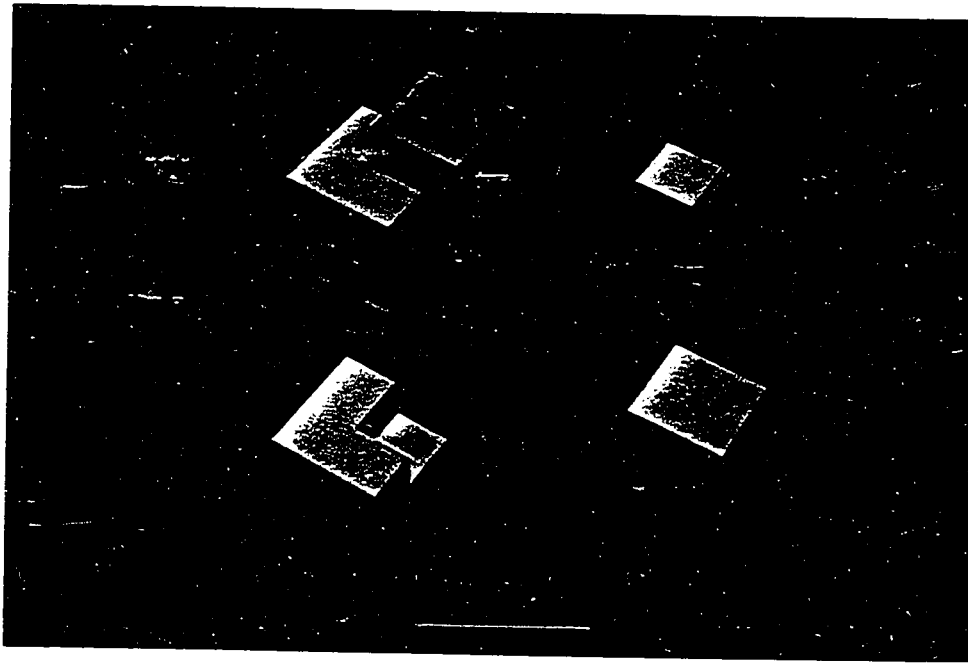


Figure 1: CSG union, intersection, and difference operations

The basic unit of animation is a *change*, which can be applied to an object, including light sources and cameras. It can be a change in position, orientation, geometry, surface characteristics, light intensity, or other properties of the object. In an action-based system like BAGS, a change that occurs over a time interval is defined by two or more *control points*, each of which is a value associated with an instant in time. The control points are interpolated over the interval to attain the continuous change.

Modeling

Modeling in BAGS results in the creation of SCEFO scripts or script fragments. The scripts, as described in Chapter 3, contain specifications of object transformations, constructions, and animation. The scripts can be created or modified directly with a text editor, but this is not intended to be the method of choice. Instead, “compilers” can be used to produce the scripts. The compilers are front-end programs that translate some other model description into SCEFO. Examples of compilers are interactive modeling programs that translate user input into scripts[35], translators that convert some other specification into SCEFO[28], or simulation software that models some behavior and outputs the results in SCEFO.

There are two conceptual phases to BAGS modeling: *object modeling*, which consists of defining primitive objects, transforming them, and using CSG operations to create complex objects; and *scene assembly*, which consists of positioning objects, lights, and cameras with respect to each other and specifying how each changes over time. In many cases the division between these two phases is vague. For example, pieces of a single object may change and move away from each other, although they are conceptually the same object. The distinction is made primarily to satisfy intuition, which tells us that actors (objects) in a movie scene are distinguishable from other actors, although each part of an

actor may undergo separate changes.

Each of the two phases is examined in more detail in the following sections.

Object Modeling

Primitive objects are categorized into *classes*, each of which represents a collection of objects that have some properties in common. As is true in other object-oriented applications, all communication with an object is transmitted through the object's class. Since classes in BAGS are responsible for the transformation and rendering of their member objects, all objects in a class must use similar methods for these processes. Therefore, the definition of each BAGS object class is procedural: an object belonging to the class must be able to be transformed and rendered by the same methods as any other object in that class.

A software package called *OFF* (the object format format, for historical reasons), is responsible for all object handling in BAGS. Part of OFF is a standardized format for object storage. Typically, each object is stored in its own OFF file, although this is not a restriction. A header on each file specifies the name of the object and its class, so that it can be read correctly.

Certain primitive object classes, such as *sphere*, *cube*, *cylinder*, and *cone*, have only one member (and one OFF file) each. Variations of each of these

objects can be obtained by transformation. For example, the one sphere in the sphere class is centered about the origin and has a radius of one unit. Other spheres and ellipsoids can be created from this one by scaling, rotating, and applying other transformations.

Other primitive object classes require data for complete specification. The *revolve* class, for example, consists of objects of revolution: solids formed by revolving a profile curve about a vertical axis. An object of this class must include, in its OFF file, data defining the profile curve. However, one set of transformation and rendering procedures can be used for all objects in the class, regardless of profile definition.

Some classes require data to be specified in a different way. For example, consider the *text* class, which contains objects that are three-dimensional extruded text strings. It would be inefficient to create a unique OFF file for each string used in an animation, especially since most files will never be used again. Instead, the *text* class contains only one object (in one file), a “generic” text object. Any text object can be created from this one by sending a particular character string, through OFF, to the object. The string is specified from within a SCEFO script.

There are approximately fifteen primitive object classes currently in BAGS. These, when coupled with the ability to add complexity with CSG, provide a

rich set of objects for use in animation. Occasionally, algorithms for a new class of objects are developed and added to BAGS. Extending OFF to contain a new object class is straightforward and fairly easy. The procedures for transforming and rendering objects in the class must be written. Once this is done, the new class is registered with OFF and is available to animators.

The use of constructive solid geometry greatly extends the range of objects. For example, all of the objects in Figure 2 are derived from CSG combinations of spheres, cubes, and cylinders. The CSG operations are all defined to be as flexible as possible. A point on the surface of the object formed by Boolean combination of two other objects has the properties of whichever surface it came from. For example, subtracting a green cube from a red sphere to produce a hemisphere causes the flat circular face of the hemisphere to be green, because it is derived from a face of the cube.

The SCEFO language contains built-in operators for CSG construction (see Chapter 3). The operations, as specified in the scripts, are in terms of the original objects. For example, the difference of a sphere and a cube is stored as just that: a sphere with a cube subtracted from it. This approach is needed to maintain SCEFO's renderer-independence. If, for example, the CSG combinations were computed by polygonal approximation of the operand objects followed by Boolean polyhedral operations, the geometry of the original objects would be

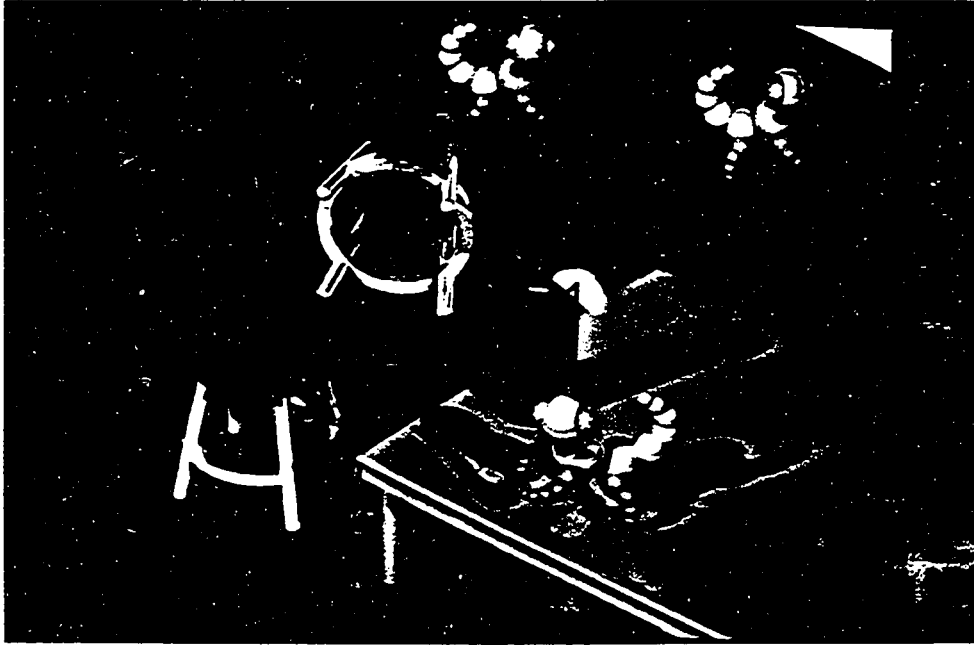


Figure 2: Complex objects created with CSG

corrupted, introducing error. Instead, the conceptual sphere and cube remain unaffected by the CSG operations.

Another advantage to this approach is that the operand objects in a CSG operation may move in relation to each other, since their geometry is not fixed by the operation. For example, a cube that is subtracted from a sphere can be directed to move in relation to the sphere, producing a dynamic slicing operation. This provides a great deal more flexibility than is found in systems that

restrict motion to joints between constructed parts.

These features illustrate the differences between two of the most prevalent forms of model storage, CSG and *boundary representation*. The latter, usually referred to as “boundary-rep” or “B-rep,” requires that models be stored as vertex-edge-face polyhedral descriptions. B-rep modeling systems are restricted by this format, since the polygonal boundaries may not be as accurate as the original object description. The hybrid approach used in BAGS, where the original CSG model is converted to B-rep for polygonal rendering, eliminates this problem. However, extra processing time is needed to compute CSG combinations for each frame of a polygonally-rendered animation. Because the added time is considerable, every attempt to exploit object coherency has been incorporated into BAGS; a CSG combination whose components remain fixed in relation to each other is represented with the same set of polygons from one frame to the next.

Scene Assembly and Animation

The task of scene assembly is to place objects in relation to each other and to define how they change over time. This is inherently more difficult than object modeling because a fourth dimension, time, is involved. In BAGS, scene assembly results in the production of a SCEFO script that contains information

about changes to objects in the scene.

When more than one change is applied to an object, the interaction between the changes must be defined. For example, the result of a translation followed by a scale is different from the result of the transformations applied in the opposite order. It is also desirable to keep the two changes distinct, so that one may be modified without affecting the other. Therefore, defining animation entails specifying each distinct change to an object and the order in which the changes are to be applied. The SCEFO language is designed for this modeling paradigm, as described in the next chapter.

The ability to apply the same change to several objects is also useful, especially in the context of *object hierarchy*, when changing an item high in the hierarchy affects all objects below it. For example, consider a hierarchical solar system consisting of a sun, a planet, and a moon. To make the planet-moon system revolve around the sun requires changing both the planet and the moon, while making the moon revolve around the planet involves only the moon. This animation is easy if the model is structured hierarchically.

Sometimes, however, hierarchies change. The moon in the above example could break away from the planet to orbit a different planet, becoming part of another hierarchy. Many animation systems restrict hierarchies to be disjoint trees, so that an object could not be a leaf in two trees. BAGS does not have

this restriction. SCEFO *groups* are hierarchically-defined collections of objects and are not necessarily disjoint. Thus, the moon could be placed into two different groups, one for each of the planet systems, and the motion could easily be modeled as desired.

Interpolation is another aspect of specifying how objects change over time. Many systems require interpolation to be linear, while some also provide splines, parabolic paths, and a few other types. However, a change in these systems is typically defined by a starting point and an ending point, with perhaps some intermediate interpolation parameters such as spline tension. An interpolation method, such as a Bezier spline, that requires non-local information about the interpolation path would be difficult to implement in such a case.

BAGS provides additional flexibility by allowing for the specification of control points and by not restricting interpolation methods. Any number of control points can be used to define a change, so that non-local interpolation data can be specified in extra control points. Any interpolation method can be used for any change, providing complete orthogonality. SCEFO also provides a way for users to define their own interpolation methods, in case none of the various predefined ones is acceptable for a desired change.

Just as objects can be structured into hierarchies, so can changes to objects. SCEFO *actions* are procedure-like groups of changes. An action may

be interpolated by any method when it is applied to an object in a scene. Since actions may be nested, multiple degrees of interpolation can be modeled in this manner. This feature provides extra flexibility to the animator during scene assembly.

Cameras must be treated specially when assembling a scene. Motions that look natural when applied to regular objects appear clumsy when applied to cameras[46]. Therefore, cameras in BAGS are handled differently, and there is a specialized positioning operation that can be applied to them. This operation is also interpolated differently, since the motion must be restricted to avoid discontinuities.

Rendering

Rendering provides the means by which a model can be viewed. It is used not only to display a completed scene, but also as an aid during the modeling process. If the modeling is done with an interactive graphical program, the rendering has to be fast enough to keep reasonable pace with the user's input. In such cases, rendering tends to be of lower quality than is desired for the final animation because of the necessity for speed. With a few orders of magnitude improvement in graphics hardware, it may be possible to render high-quality images in real time, so viewing while modeling will be the same as viewing the

final image.

With current technology, however, there must be a tradeoff between image quality and rendering time. BAGS includes several renderers that cover the speed-realism spectrum. At one end are those methods that provide a fast, less-realistic view of a model. Two renderers, an animation previewer and a single frame viewing program, use the Evans and Sutherland PS300, a high-speed vector graphics device, to provide real-time interactive capability. A relatively fast shaded static image can be rendered with a faceted-shaded painter's algorithm, a Gouraud-shaded Z-buffer algorithm (for the Lexidata Solidview and Rastertech I/380 devices), or a Gouraud- or Phong-shaded software Z-buffer algorithm. For higher image quality, a scan-line renderer with transparency and anti-aliasing is available[81]. The highest level of image quality (and typically the longest execution time) is provided by the ray tracing rendering method, which incorporates global lighting effects, such as shadows, reflections, and refractions[83].

A polygonal approximation of the model is used by all the rendering techniques mentioned above except for ray tracing. Rendering with ray tracing involves intersecting rays with objects, so objects are modeled by procedures that solve the equation of the intersection. This requirement is why most systems do not provide ray tracing in addition to polygonal rendering methods: the models are different. Of course, it is possible to ray trace a model after convert-

ing it to polygons, but that would be slower, less accurate, and less attractive than using the original representation.

Because it stores its model in renderer-independent SCEFO, and because it uses object-oriented rendering, BAGS does not have this limitation. Procedures provided by each object class can either produce a polygonal approximation of an object in the class or compute the intersection of a ray with the object. This approach is extensible, as well. If another rendering technique were developed, such as the one described in Chapters 5 and 6, it could be added by extending each of the object classes to include procedures for that technique.

The object-oriented approach is used for object transformations, as well. This may seem to be overkill: since in most cases all object classes will do the same thing to implement a transformation, such as multiplying by a scale matrix, the operations could be done at a higher level, outside the classes. However, there are types of transformations for which the object-oriented approach is necessary. For example, the non-linear deformation operators developed by Barr[5] are easy to implement for polygonal models, but are very difficult (and slow) for ray tracing[6]. Therefore, classes can implement the deformations by approximating the objects with polyhedra, and then applying the polygonal version of the operators. While this does affect the model used for ray tracing, it allows the deformations to be used in a consistent fashion, and does not affect

the rendering efficiency too much. If someone developed good methods for ray tracing deformations for a particular object class, then that class would not have to convert to polygons.

Post-Production

Included in the category of post-production are image storage, compositing, and recording. *Compositing* involves several techniques, including image overlaying and dissolving[65], techniques which are standard in film post-processing.

Few if any of the post-production facilities in BAGS are different enough from what is provided in other systems to warrant mention here. What is worth noting, however, is the small degree to which a BAGS user must rely on these facilities. In many animation production systems, image compositing is used to a large extent to save rendering time. For example, a dynamic foreground against a static background can be modeled and rendered in two pieces. Since the background is unchanging, it can be rendered once, and the resulting image composited with the foreground image for each frame. This approach saves rendering time at the expense of user time, since the animator must separate the models, render them separately, and specify that they are to be composited. It is also renderer-specific: if the background contains shiny surfaces, for example, a realistic renderer might show the changing reflections of the foreground.

This means that changing the rendering technique necessitates a user-specified change in the model. This requirement is contrary to the design philosophy of BAGS, as is valuing computer time over the user's time.

Summary

In this chapter we have described the overall design of BAGS and how that design meets the goals required for a general animation system. We have shown how use of the SCEFO script language allows a variety of modeling techniques to be incorporated into the system. These techniques include direct textual specification, interactive graphical programs, and simulation front-ends. Thus, the system is not restricted to any single modeling paradigm.

Rendering is flexible as well. Any of the variety of rendering techniques, from real-time animation previewing to photo-realistic ray tracing, can be driven from the same model. In fact, the **render_scene** command in BAGS is used to produce all types of rendering; the rendering method is specified as an option to the command.

3. The SCEFO Language

In Chapter 0, we introduced the idea behind script languages and how they serve as a non-programming text interface for model specification. In this chapter, we describe SCEFO, the animation script language used in BAGS. We pay particular attention to the language features that allow BAGS to achieve the animation system goals described in the previous chapter.

Animation Languages

What qualities are desired in an animation script language? Perhaps generality is the most important feature. A system should be general enough that it is easy for an animator to transfer a model from his or her mind to a script in a straightforward fashion: the process by which the animator creates the conceptual model should be duplicated when creating the computer model. Methods that facilitate this process include object instantiation, hierarchical

modeling, action-based animation, and hierarchical actions.

An animation language should be easier to use than a conventional programming language. Animation languages are typically problem-oriented, so that scripts are limited to a particular domain. That is, the data types in an animation language are “concrete” instead of abstract: they are objects, lights, cameras, and animated changes. It makes sense to similarly limit the structure of the language, rather than to use a high-level language (as in ASAS[67] or MIRANIM[59]), for several reasons. The animator should not be required to have general programming abilities. Also, a limited language allows error checking to be more extensive and precise. Furthermore, scripts can be generated more easily by front-end modeling programs if the animation language syntax is simple and straightforward.

An animation language should also be more precise and flexible than interactive interfaces. Graphical interaction serves its purposes in most cases, but there are times when more control is needed. That is, there are certain animation sequences that are difficult or impossible to describe adequately through a graphical interface. For example, complex changes to surface properties of an object are hard to model on a display device. Similarly, it may not be possible to describe complex forms of motion interpolation in that way. An escape to an animation language is useful in such cases.

Another desired feature is extensibility. No matter how feature-ridden an animation system is, an animator may require something else. It must be possible to add new features to the system easily. Even better is the ability for advanced users, not just system programmers, to add features.

Other Languages

There are very few animation systems built upon a language that addresses the issues outlined in the previous section. Research testbeds are usually designed for programmers, so there is no need for a non-programming interface. Commercial production systems are often designed so that jobs can be done as quickly as possible; demands are placed on the users, requiring a high degree of experience.

Some systems have very tight coupling between modeling and rendering and therefore do not use an intermediate script language. For example, the S-Dynamics system[77] creates a polygonal model via an interactive interface, and sends the polygons directly to the renderer. The model is defined by the actions of the animator using the system, and translates directly into a collection of polygons.

A tight connection between modeling and rendering is possible only in systems where the model must be specified in a particular manner. This can be an

interactive interface, as in the case of S-Dynamics, or a programming approach[15,59,67]. While this coupling leads to a well-integrated system, it restricts modeling and rendering flexibility.

SCEFO

SCEFO is designed to incorporate the desirable features described above. That is, the language serves not only as model storage and input to renderers, but a tool by which animators may produce models. Although SCEFO is a reasonable way in which to specify animation, it is designed to be terse and structured enough to be output by front-end modeling programs, such as interactive interfaces and simulation software.

The descriptions in this chapter serve to illustrate certain features of SCEFO and are not intended to fully document the language. Appendix A contains a reference guide to SCEFO; a more complete language description can be found in a tutorial guide[72].

Object Creation

The creation of objects used in a scene is an important aspect of modeling. In systems like *twixt*[34] and OSCAR[57], all objects used in an animation are created separately and instantiated as necessary in a script. This forces the

animator to model all objects beforehand and to use those objects as they are within the script. Geometric changes, such as rigid transformations and polygonal displacements, are possible within an object, but the overall object assembly must remain pretty much unchanged.

SCEFO, on the other hand, uses a more general approach. Primitive OFF objects (described in the previous chapter) can be read from their storage files to create *templates*. A primitive object can be a standard canonical object, such as a sphere or cube, or a user-defined object, like an object of revolution or a parametrically-defined surface.

A template may be copied any number of times to create new templates. The new templates may be copied as well, creating a tree of modeled objects. (This should not be confused with hierarchical grouping of objects described in the next section.) Changes in the tree propagate only towards the leaves: a change to a template affects all templates derived from it. This means that multiple copies of a template inherit any changes made to that template. The reverse is not true: changes made to each of the copies do not affect the other copies.

Templates do not appear in rendered scenes; they are used only for developing models. However, an *instance* of a template does appear in a scene. As with template copying, any number of instances may be made from a single

template. Changes to the template affect the instance, but not vice-versa. Thus, instantiation of templates is just another level of modeling, but it is one that creates visible objects.

Consider the example in Figure 3. The **read** statement reads a primitive object from an OFF file; in this case, the canonical cube is read from its standard file. This statement creates a template named `cube` in the process. Two more templates, `big_cube` and `little_cube` are created from that template through use of the **template** statement. Each of the two new templates is modified by a **change** statement (described in a later section) to modify its size by scaling in three dimensions.

Two instances are made from each of the two new templates. The `Big_red_cube` and `Big_blue_cube` are both made from the `big_cube` template, and therefore inherit the size change to that template. (The capitalization of instance names is a stylistic convention that makes SCEFO scripts easier to understand and is adopted in these examples.) Similarly, two instances of `little_cube` are created. Each of the four instances has a color assigned to it. The result of this (after instantiation of cameras and lights, and appropriate changes to position) is shown in Figure 3

Templates may also be combined using the constructive solid geometry (CSG) operations *union*, *intersection*, and *difference*. The **template** statement is

```

read      (cube.off)           cube;

template (big_cube)           cube;
template (little_cube)       cube;

change    (big_cube)           scale <0, {4, 4, 4}>;
change    (little_cube)       scale <0, {2, 2, 2}>;

instance  (Big_red_cube)      big_cube;
instance  (Big_blue_cube)     big_cube;
instance  (Little_yellow_cube) little_cube;
instance  (Little_green_cube) little_cube;

change    (Big_red_cube)      set_color <0, RED>;
change    (Big_blue_cube)     set_color <0, BLUE>;
change    (Little_yellow_cube) set_color <0, YELLOW>;
change    (Little_green_cube) set_color <0, GREEN>;

```

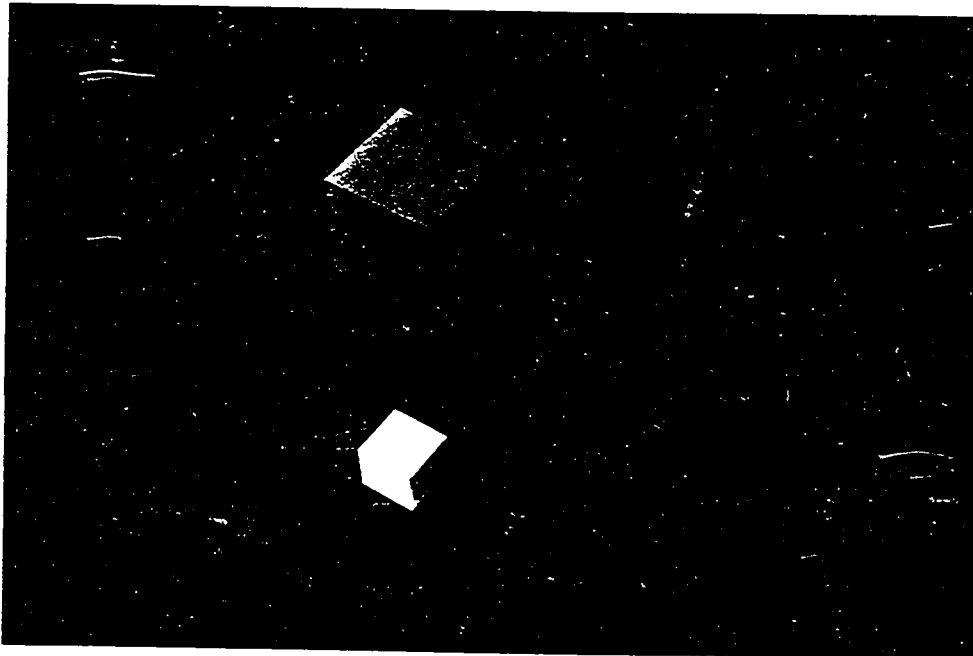


Figure 3: Templates and instances, and rendered image

used for this as well. If the right-hand side of the statement is a CSG expression of templates, the resulting template is created from a combination of the named templates. The example in Figure 4 shows how a simple rivet template can be created from a sphere, cube, and cylinder. The cube is positioned with respect to the sphere and is used to cut it in half with a difference (symbol $-$) operation, creating a hemisphere. The union (symbol \cup) of the hemisphere and the cylinder produces the final rivet object. The component objects and the CSG rivet are displayed in the image in the figure.

Note that the surface properties of the CSG object in the figure are derived from the component objects. For example, the part of the rivet's surface that came from the sphere is colored like the sphere. This is true even for the face of the hemisphere derived from the cube. This surface is colored like the cube, which does not appear in the final image because it was subtracted from the sphere. This rule used to determine the surface properties of a CSG object is designed to be general, so that it is possible to show such effects as these. If the system constrained the flat face of the hemisphere to have the same surface properties as the sphere, it would not have been as easy to create this image.

CSG templates may be treated like any other templates. They may be copied, instantiated, or used in other CSG expressions. It is therefore possible to create complex CSG hierarchies of objects. An example is shown in the image in

```
read    ("sphere.off")  sphere;
read    ("cube.off")    cube;
read    ("cylinder.off") cylinder;

change  (sphere)        set_color    <0, BLUE>;

change  (cube)          scale        <0, {2, 1, 2}>,
                           translate   <0, {0, -1, 0}>,
                           set_color    <0, RED>;

change  (cylinder)      scale        <0, {.4, 1, .4}>,
                           translate   <0, {0, -.9, 0}>,
                           set_color    <0, GREEN>;

template (rivet)        (sphere - cube) | cylinder;

instance (Rivet)        rivet;
```

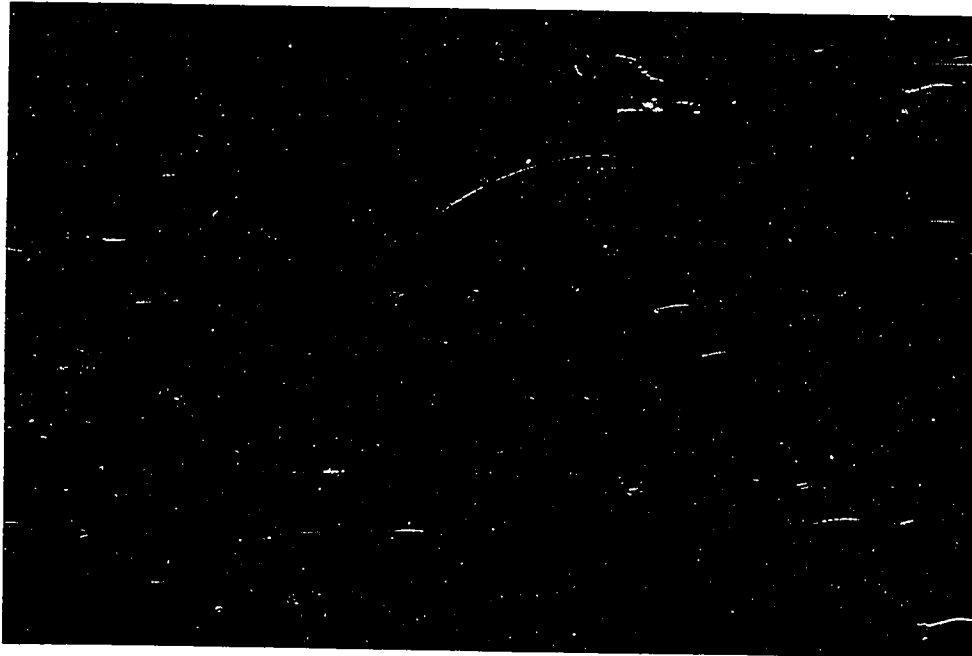


Figure 4: Construction of a rivet object with CSG

Figure 5, which is a view camera modeled as CSG combinations of spheres, cubes, and cylinders.

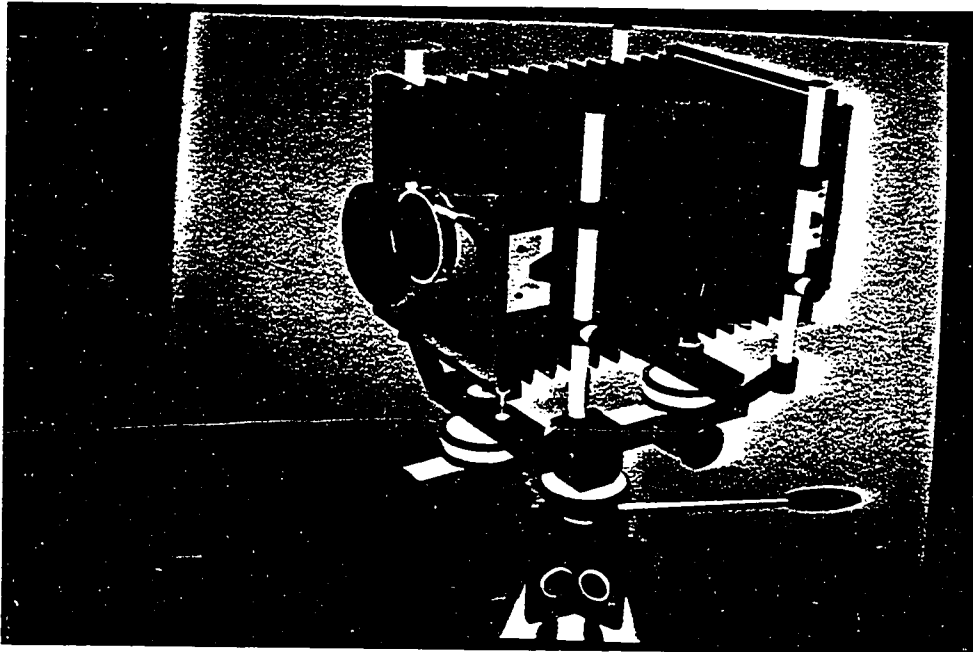


Figure 5: A view camera modeled with CSG

Object Hierarchies

The ability to create hierarchies of objects is found in many systems. One of the more flexible implementations is found in *twixt*[34], which allows a forest of object trees to be created. The trees are dynamic, so that an object can be a

part of one tree at one frame and a part of a different tree at another frame.

However, even the *twixt* hierarchy model is too restrictive. There is no reason that an object could not be considered to be part of two hierarchies at the same time. For example, consider people marching in a parade. Each person is “attached” to the rank and file in which he or she is marching. If either the rank or the file moves or turns, the person should move with it.

Because of this, SCEFO allows overlapping hierarchies of objects. An object can be placed into one or more *groups*. Groups, in turn, can be nested inside other groups. The **group** statement is used to create hierarchies, as in Figure 6. The script fragment in the figure creates two groups, a `planet_system`, composed of the planet and moon, and a `solar_system`, composed of the sun and the planet-moon system.

```
group    (planet_system) planet, moon;
group    (solar_system)  sun, planet_system;
```

Figure 6: A simple object hierarchy using groups

Once the hierarchy is created, changes can be made to any level in the hierarchy. In the solar system example, changes could be made to just the planet, moon, or sun; none of these changes would affect the other objects. A

change to the `planet_system` would affect the planet and the moon. This could be used, for example, to make the whole system orbit the sun. Similarly, a change to the `solar_system` group would affect all three objects.

Groups can consist of templates or instances, but not both. A *template group* can be treated like a template. It may be copied, instantiated, or used in a CSG expression or template group. A CSG combination involving a group operates on the union of all objects within that group. An *instance group* is just like an instance; it can only be changed or included in another instance group.

Action-Based Changes

Early attempts at computer animation, based on conventional forms of animation, relied on a keyframe approach to interpolation. In cel animation, which is used primarily for animated cartoons, the artist would paint or draw certain “key” frames at which some action caused the picture to change. The artist would then give the collection of keyframes to an assistant, who had the repetitious task of drawing the frames between the keyframes. This process, sometimes called “in-betweening,” is often well-suited to computer interpolation.

In three dimensions, however, the keyframe approach is more difficult. Specification of a keyframe requires knowing the positions of all objects in the scene at that frame. For a two-dimensional cartoon this is not a significant

problem, but when there are many three-dimensional objects flying about, it is harder to keep track. Also, the memory needed to store object states at each of the keyframes can become excessive, as users of BBOP discovered[76].

Another difficulty arises when editing a keyframed animation. Consider the case where the time interval of one particular motion must be lengthened or shortened. Doing so requires that all keyframes involving that motion be modified to contain the correct new data. This is not a trivial task, since other motions to the object may be occurring at the same time.

An *action-based* approach is much more convenient for modeling. Each change to an object is modeled as a separate action, which may be layered with other actions to produce complex motions and other effects. This approach has been implemented in other systems and has been proven to be effective. For example, *twixt* uses actions called *tracks* to layer compound motions of objects.

In SCEFO, the **change** statement implements a similar, but more general, scheme to modify the state of an object. The modification can be a geometric transformation, a change in surface characteristics, or a change in any other property that affects the object. The body of the **change** statement consists of one or more *change-ops* (change operations), each of which alters the object's state in some way. The parameters to a change-op are in the form of *control points*, each with a time value and corresponding control value.

The example statement in Figure 7 has three change-ops: a `scale` (change in size), a `translate` (change in position), and a `rotate`. The `scale` operation has a single control point at time 0, meaning that the change in size is to take place instantaneously at that time. The control value for the operation consists of a list of three values, representing the size change in three dimensions.

<code>change</code>	<code>(box)</code>	<code>scale</code>	<code><0, {1, 1.5, 2}></code> ,
		<code>translate</code>	<code><4, {0, 0, 0}></code> <code><10, {5, 0, 0}></code> <code><20, {6, 5, 0}></code> ,
		<code>rotate</code>	<code><12, {{0,0,0}, {0,1,0}, 0}></code> <code><22, {{0,0,0}, {0,1,0}, 90}></code> ;

Figure 7: A sample change statement

The `translate` in the example has three control points, defining how the translation is to occur over time. At time 4 the translation is by 0 in all three dimensions, so no movement occurs. Between time 4 and time 20, the object will be moved by 6 in x and 5 in y . The control point at time 10 indicates that the movement is in the positive x direction until time 10, at which time the motion changes to be in both the positive x and y directions. The translation is graphed in Figure 9. The rotation in the example occurs between times 12 and 22. It

rotates the object by 90 degrees around the axis defined by the origin and the positive y -axis.

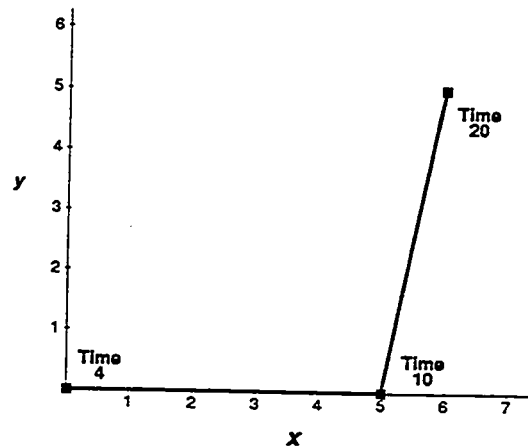


Figure 8: Translation graph of the previous example

These change specifications are action-based; the only keyframes that are specified are on a per-action basis at each of the control points. The object motions in the previous example are specified separately in each of the change-ops, which are layered together automatically. Figure 9 is a graph against time of these three motions. The translation and the rotation overlap in time and therefore are ambiguous. Since these operations are not commutative, the order in which they are applied is important, and must be known to the animator.

SCEFO uses a very simple mechanism to resolve this ambiguity. The order in which statements or change-ops are specified in the script is the order in which they are applied; in the example, the translation will be performed before the rotation at each frame where they both have effect. This rule provides a clear, easy to remember, precise method for resolving overlapping changes.

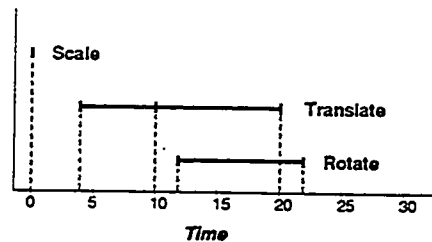


Figure 9: Time graph of the motions in the previous example

The time values in control points do not correspond to seconds, frames, or any other “real” time units; they are merely figurative. When the scene is rendered, these values are mapped linearly to real frame times as specified by the animator. It is possible to define an entire scene between time 0 and time 1, for example, and map that interval to any other time values for rendering. It is typical, however, to have the SCEFO times correspond to seconds or frames, for clarity.

Similarly, the distance units in the script are arbitrary. Whatever scale is most convenient for the animator's model can be used. As long as the relative sizes of the objects and camera in the scene are compatible, the script is valid. This gives the animator more leeway in model definition, since no size conversions are necessary.

Interpolation

Interpolation is used to compute the state of an object between keyframes or, in the case of an action-based system, within the effect of an action. In some cases, such as a change in surface color, linear interpolation is sufficient. For motion, however, linear interpolation is too "jerky," because it ignores the derivatives of the change. For this reason, most action-based systems offer a choice of interpolation methods to apply to each action.

In *twixt*, a user has a choice of several interpolation methods, including linear, parabolic, and cubic spline[34]. The system also allows ease-in and ease-out specification for accelerated and decelerated motions. The S-Dynamics system provides linear and exponentially-eased interpolation and also allows a user to sketch an interpolation curve[77].

In SCEFO, a user can designate an interpolation method for each change-op. A variety of methods are provided, including linear, eased, accelerating,

decelerating, and spline. In cases where none of these methods is appropriate, a user may define a new one, as described later. Each change operator has a default interpolation method associated with it. For example, rotations of objects are best interpolated with quaternions[69], so that is the default method for the rotate operator.

Action Hierarchies

Creating hierarchies of actions is a useful technique not found in other systems. It is clear that a hierarchy of objects is more manageable than an unstructured collection. Similarly, a single complex action is easier to work with than a list of many component actions.

The SCEFO **action** and **apply** statements implement action hierarchies. A SCEFO action is a parameter-substituted collection of changes, somewhat like a procedure in a programming language. The parameters, which may be objects to change, time values, operator names, control values, or any other component of changes, are substituted with real values when the action is applied in an **apply** statement. A simple example of the application of an action is shown in Figure 10. The action called `grow_and_move` is defined to take three parameters: the object to change, the amount by which the object is to grow, and the translation factors. Names enclosed in ampersands within the action are

dummy parameter names that are to be substituted with values when the action is applied. In the example, the `object` is scaled by the `growth` factor and then translated by the `movement` vector, which should contain three values.

```

action (grow_and_move) object, growth, movement {
  change (&object&)  scale    <0, {1, 1, 1}>
                        <10, {&growth&, &growth&, &growth&}>,
                        translate <0, {0, 0, 0}>
                        <10, &movement&>;
}

apply (grow_and_move)  Sled, 2.1, {3, 0, 0}  <30, 0>  <90, 10>;

```

Figure 10: SCEFO action and apply statements

The **apply** statement in the example changes the `Sled` object, scaling it by 2.1 and translating it by 3 units in the positive x direction. The parameters' values are substituted in the order specified in the action. Each value can be a name, value, value list, or whatever the action requires; in the example, the third parameter in the **apply** is a vector of three numbers, representing the translation values in three dimensions.

The action in the example is defined to occur between times 0 and 10, but these times have no correspondence to scene times until the action is applied. The control points at the end of the **apply** statements determine how the time interval of the action gets mapped to scene time. In the example, the two

control points in the **apply** state that action time 0 corresponds to scene time 30, and action time 10 maps to scene time 90. By default, the times are interpolated linearly in this interval, although the animator may specify a different interpolation method, as is done for changes.

An action may have applications of other actions within its body, creating a multiple-level action hierarchy. This also provides a way to implement double and higher-order interpolation. That is, an application of an action is interpolated once, and inside the action is the application of another action, which is interpolated as well. This may be used, for example, to define a spline motion (one interpolation), which is then stretched or squashed in time (the other interpolation).

Object Communication

When designing an animation, it is often useful to think of objects acting in concert, rather than independently. Because of this consideration, some systems provide means to communicate information between objects. In ASAS, for example, a user can program message-passing code into object definitions, such as having an object ask another object for its position and use that information to position itself[67]. In OSCAR, some of the methods defined for each object deal with inquiring information associated with that object. This information

may be used to direct the actions of other objects[57]. The approach used in SCEFO combines the flexibility of the ASAS method with the user friendliness in OSCAR. Any type of information may be communicated, as in ASAS, but the user has an easy way to specify the message-passing, as in OSCAR.

In SCEFO, *fields* are used to store information with an object and to communicate information between objects. Figure 11 has an example of the use of a field within a **change** statement. The value of the `mass` field is set for each object with the `set_field` operator. The value of the field, referenced by name, is used within the `change` to determine (using the `TIME` preprocessor macro) the time necessary to accelerate each mass a given distance when applying a given constant force. The `accel` interpolation method is used to enforce constant acceleration in the interval. The result is that the `change` takes different amounts of time for the two objects in the group since they have different masses: the `marble` will finish its translation at time 10, while the `cannonball` will take 100 time units.

All objects have certain predefined fields set by default. These fields are denoted by names beginning with underscores and include `_position`, which is the three-dimensional location at which the origin would arrive if transformed by all changes applied to the object so far; `_color`, the RGB color of the object's surface; and a variety of others that determine the object's state.

```

/* Amount of time needed to move object with mass = MASS by
 * DIST units, if applying constant FORCE. This is derived
 * from the formulas  $f = ma$  and  $s = 1/2 a t^2$ . */
#define TIME(MASS,DIST,FORCE)  sqrt(2 * MASS * DIST / FORCE)

change  (marble)      set_field  <0, {"mass", 10}>;
change  (cannonball)  set_field  <0, {"mass", 1000}>;

group   (masses)      marble, cannonball;

#define FORCE          20 /* Constant force applied to masses */
#define DISTANCE      100 /* Distance masses are to move      */

change  (masses)      translate <0, {0, 0, 0}>
                        <TIME(mass, DISTANCE, FORCE),
                        {DISTANCE, 0, 0}> : accel;

```

Figure 11: Using a field within a change

Predefined fields can be used to model interactions between objects, if the position of one object is used to affect the behavior of another.

A field associated with object `foo` can be accessed within a change to object `bar` as `foo:fieldname`. This is a simple mechanism by which one object can influence another. It may be used, for example, to keep the camera pointed at the center of a particular object, no matter how that object moves.

Fields are also used to send information to OFF object classes. For example, the *text* object class receives as a field the string of characters used to create a particular three-dimensional text object. Another special operator, `send_field`, is used to communicate this information to the OFF class.

Fields are useful to simulation front ends that need to store nonstandard modeling information, such as the object masses in the previous example. Constraint-based interactive modelers are a type of program that must rely on such information. *Constraints* are used to help the animator position and move objects in relation to each other. For example, a constraint can be defined between two objects so that they remain a fixed distance from each other. The information necessary to compute the constraint satisfaction criteria can be associated with the objects via fields. Predefined fields are especially helpful to these programs, since they allow the animator to refer easily to connection points on various objects.

Other Features

In this section we describe some other features of SCEFO that increase its effectiveness as an animation language.

Format. The format of SCEFO is designed to be as readable as possible without being wordy. Each statement in a SCEFO script is denoted by a keyword, which is followed by arguments for that statement type. White space (blanks, tabs, and newlines) may appear anywhere between statement tokens and is ignored. As is true for most free-format languages, indentation is not enforced by the language parser but is used for clarity.

SCEFO animation scripts are stored in one or more files. By default, script files are filtered through the C language preprocessor[53] before they are parsed. The preprocessor provides commenting, constant and macro definition, file inclusion, and conditional inclusion facilities. These features enhance readability, provide flexibility, and allow SCEFO scripts to be defined in a modular fashion. For example, an interactive object modeler can produce SCEFO files to be **#included** inside an animation script.

Object Context. SCEFO provides a mechanism to refer to any object within a context, such as a template used to create an instance, or an instance used within a group of instances. Figure 12 contains an example of such a reference. The `bike` template is a CSG combination of three templates, a `frame`, a `front_wheel`, and a `rear_wheel`. Two instances of the template are made. The color of the `bike` template is then set to red; this affects both instances, changing them (all three parts) in the process. The change to `Mike_bike'bike'frame` (the apostrophes are used to separate names within a context specification), however, affects only the frame component of the `Mike_bike` instance of the bicycle. The other instance and all the templates remain unchanged.

Changes within context allow the animator to make minor modifications to one copy of an object without having to create an entire, slightly different, new

```
template (bike) frame | front_wheel | rear_wheel;

instance (Mike_bike) bike;
instance (Spike_bike) bike;

change (bike) set_color <O, RED>;
change (Mike_bike'bike'frame) set_color <O, OCHRE>;
```

Figure 12: Changing an object within a context

object. Context is not restricted to a CSG component of an instance. Other possible contexts include members of groups, templates within CSG trees, and sub-instances created from template group instantiation.

Cameras and Lights. Cameras and lights are treated like any other objects, at least as far as the SCEFO language is concerned. They are read from their own OFF files and may be transformed by any of the standard operators. There are also specialized operators defined for use only with cameras and lights. For example, the `set_intensity` operator is used to define the RGB intensities of a light source, and the `cam_define` operator sets the focal length, film size, and film tilt of an orthographic or perspective camera. Because smooth camera motion requires specialized modeling techniques, there is a `cam_position` operator (with its own special interpolation method) that is used to move cameras[46].

Expressions. Any numeric value used in a control point may be an arithmetic expression. SCEFO provides the standard arithmetic operations of addition, subtraction, multiplication, division (integer and floating-point), and modulo. There are unary negation and logical-not operators and parentheses for precedence. Vector arithmetic (i.e., operating on lists of values) and string operations are also allowed. Any of a set of predefined functions may be used, as well. The example in Figure 13 shows some of these features.

```

/* Translate 16 units along vector at 23 degrees to horizontal */
change (Climber)  translate  <0, {22, 43.1, 17}>
                        <10, {22  + 16 * cos(23),
                          43.1 + 16 * sin(23),
                          17}>;

```

Figure 13: A change statement using arithmetic expressions

Loops. Looping constructs — for-loop and foreach-loop types — can be defined with the **loop** statement, as in Figure 14. The example in the figure is a for-loop that is iterated 4 times, with the loop variable `count` taking on the values 0, 1, 2, and 3. An occurrence of the variable name within at-signs in the body of the loop is replaced by the current value of the loop variable. Each iteration in the example loop produces a new instance of the `turtle` template, named `Turtle0`, `Turtle1`, and so on. The first instance is translated by 15

units, the second by 30, and the others by 45 and 60 units, respectively.

```
loop (count = 0 to 3) {  
  instance (Turtle@count@) turtle;  
  change   (Turtle@count@) translate   <0, {15 * @count@, 0, 0}>;  
}
```

Figure 14: A simple SCEFO for-loop

Prefixing. Another useful feature is provided by the **prefix** statement. Since all names in a SCEFO scene must be unique, the animator must make sure not to use the same name twice. If a scene is designed modularly, with perhaps one main object per file, duplicate names may be used unintentionally. Name prefixing is designed to avoid this problem. The example in Figure 15 shows how the prefix statement can be used to ensure unique names. The first **prefix** statement indicates that all template, instance, group, and action names in the following statements are to be prepended with `boat_`. (Obviously, they are all related to the construction of a boat.) For example, the template defined in the example is actually called `boat_mast`. The tilde in front of the name `cylinder` is used as an escape mechanism, since that name should not be prefixed. Finally, the last statement cancels the effect of the prefix. Prefixes can be nested, so the **prefix** statement actually “pushes” and “pops” strings.

```
prefix (boat_);  
  
template (mast) ~cylinder;  
  /* ... Other boat construction statements ... */  
  
prefix ();
```

Figure 15: Using prefixing to avoid name duplication

Variables. The **assign** statement can be used to define SCEFO variables. Variable values, which may be interpolated over time, can be accessed in any change. An example is shown in Figure 16, in which the variable `random_move` is set to a list of three random numbers, each ranging from 0 to 2. (Each number will change value at each rendered frame.) This random vector is added to a translation vector for each of two objects, moving in opposite directions. The variable is used in this example to hold the random values, so that the same vector can be added to both translations.

Extensibility

No animation system is complete without a way for a user to add features. ASAS and S-Dynamics, for example, allow certain extensions to be added through LISP programming[67,77]. This ability not only gives animators a

```

assign  (random_move)  <0, {random(2), random(2), random(2)}>;

change  (Fly1) translate  <0, {0, 0, 0} + $random_move>
                           <0, { 100, 0, 0} + $random_move>;
change  (Fly2) translate  <0, {0, 0, 0} + $random_move>
                           <0, {-100, 0, 0} + $random_move>;

```

Figure 16: A SCEFO variable

means for adding effects, but also allows researchers to test new techniques easily and quickly without having to recompile the system or change existing software.

There are three types of extensions that can be added to SCEFO: operators, interpolation methods, and value functions. Each requires some code to be written in C and compiled. The resulting object code may then be loaded dynamically into any program, such as a renderer, that is processing the SCEFO script[63]. The SCEFO **loadfile** statement is used to make the executing program load the object file. The calling sequence for each of the extensions is defined uniquely. Unlike ASAS and S-Dynamics users, a SCEFO animator does not have to worry about how the new code interfaces with the system, as long as the procedure matches the sequence.

In the case of operators, the user must write code that applies the operator to an object, affecting its state. The procedure is passed the necessary control

values after interpolation at the frame being processed. A user-provided specification ensures that these values are acceptable to the procedure. The code can implement a complex motion, a change in surface characteristics or geometry, a combination of these, or anything else.

An interpolation method is implemented as a procedure that takes a set of times, control values, and the frame time being interpolated. They may also be passed extra parameters specified in a SCEFO script. An interpolation method, like an operator, has an associated value specification so that it can check incoming control values for compatibility.

The third possible user-defined extension involves value functions. These are functions, like `sin` and `cos` used in a previous example, that take values as operands and produce other values. The functions are defined in a format compatible with the VAL value-handling package[73]. New functions can be written to perform complex calculations or any other non-standard functions.

Summary

In this chapter we have described the features of the SCEFO script language that allow animators to define models in a consistent and straightforward manner. The template/instance distinction, CSG operators, grouping and action constructs, contextual changes, and fields provide flexibility, while the

unambiguous statement-ordering rules ensure predictability. The extra syntactic features, such as loops, prefixes, and variables, make scriptwriting a more manageable task. Furthermore, any feature that does not exist in the language but is helpful for a particular model can be added as a user extension.

4. Rendering

Rendering produces a view of a scene, either for feedback while modeling, or as part of a finished animation. An animator who wants a rendered scene for immediate feedback is usually willing to sacrifice some image quality for fast turnaround. When the model is complete, the opposite is often the case. For this reason, and because of current hardware limitations, several levels of realism and speed in rendering are required. The goal is to provide the best performance for the current hardware but to be flexible enough that if hardware improves, or if a new rendering technique is developed, the system does not become obsolete.

To provide this flexibility requires that the modeling and rendering components of the animation system be decoupled, so that one does not depend on the other. If a model is stored as a collection of polygons or other low-level primitives, for example, the renderer has no choice but to produce an image of

polygonal objects. This particular situation is common to all other known systems, preventing them from producing smooth ray traced images. The *twixt* system[34] and the system developed at the University of North Carolina[8] were both designed with multiple renderer capabilities, but neither is set up to do procedural ray tracing.

In BAGS, rendering must impose a particular view upon a model stored in the SCEFO language. Because SCEFO does not contain any renderer-specific information such as polygons, a renderer is free to represent each object in whatever way is best; the object-oriented OFF interface allows the objects to be treated in this manner. Therefore, both polygonal and ray tracing rendering methods work from the same model. Furthermore, this approach permits the most accurate representation of a model to be created for a given frame. For example, the number of polygons used to approximate an object can be selected automatically for each frame, depending on the importance of the object in that frame.

The choice of a lighting model for use in rendering is very important. A significant amount of modeling time is devoted to lighting a scene, so it is necessary that the lighting model be easy to use. The model should provide a high degree of realism by default, but should be flexible enough so that advanced users can override those defaults. The parameters for defining an object's sur-

face should be understandable and intuitive; an animator should not need a degree in physics to create a new material specification. It should also be possible to interpolate the parameters, so that transitions between surface materials can be animated smoothly.

The BAGS lighting model incorporates these features. The default model produces lighting that is more realistic than that provided by the Phong model, which is used in most other systems. Unlike other realistic models, such as Torrance-Cook[17], the input parameters are intuitive and can be interpolated. The model is also flexible, in that any of the parameters or computed lighting values may be set or modified by user-defined functions. This is similar to the flexibility built into Cook's shade trees[18] and Perlin's image synthesizer[64], but is available to non-programming animators.

The BAGS Rendering Process

Rendering may produce a single image or a series of images that becomes an animation when displayed in sequence. Either way, rendering is the task of representing a view of a scene at a single frame as a two-dimensional picture. If an animation is to be rendered, the process is repeated for each frame of the animation.

Rendering a frame can be separated into four primary steps:

- interpolating changes at the given frame;
- computing the states of objects, lights, and cameras at the frame;
- determining visible surfaces;
- lighting and shading surfaces.

The last two items are necessary only for shaded images; wire-frame renderers do not need to hide or shade surfaces.

Input to a renderer is usually in the form of a SCEFO script. The SCENE software package[74] parses a SCEFO script and sets up data structures that correspond to the scene. If the rendering is being done by an interactive modeling program, that program is already working with a SCENE data structure, and therefore does not need to parse a script. SCENE also includes code to write out part or all of a scene to a file, so that modelers can create SCEFO scripts. Figure 17 is a block diagram of the central packages in BAGS which are involved in the rendering process.

Given a SCENE data structure, the FRAME package can access the contents of a single frame. It uses the SOI (Standard Operators and Interpolation methods) package to interpolate control points at the frame and to determine what transformations result from each SCEFO operator. The result is a data structure that corresponds to the contents of the frame. This includes the

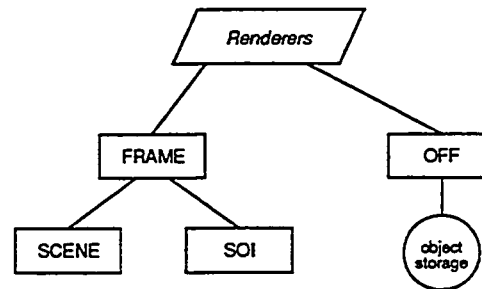


Figure 17: BAGS rendering block diagram

objects, lights, and cameras that are visible and transformations that are to be applied to each of them.

Determining the state of an object is then a matter of applying its transformations. Each object transformation is sent to the object-oriented OFF package, which sends it to the appropriate object class. The class then determines how the transformation affects the object's state and acts accordingly. OFF stores state information with each object, so the information is available when the object is rendered.

The remaining steps are all renderer-dependent. Currently in BAGS there are two broad categories of renderers: polygonal approximation and ray tracing. Polygonal approximation is used for several rendering methods, including wire-frame, faceted, depth-buffered, and scanline. Each of these methods (except

wire-frame, which displays all edges of all polygons) includes some mechanism for determining which polygons, or pieces of polygons, lie in front of others. This step is called *visible surface determination*. The ray tracing technique traces rays from the camera to objects in the scene to determine visible surfaces. The first object that is intersected by a ray is the one that is visible. Each of these methods is described in more detail later in this chapter. Images produced from the same model by four different rendering methods are shown in Figure 18.

There are OFF class routines for both of the rendering formats for each class of objects. Every class must be able, upon request, to provide a polygonal approximation of an object in its class or to intersect a ray with an object. Since these procedures are implemented on a class-by-class basis, they may be tailored for maximum efficiency and quality. The OFF software package serves as the interface between renderers and the object classes, as diagrammed in Figure 19.

The Lighting Model

For all but wire-frame renderers, lighting and shading are the next steps in the rendering process. *Lighting* is the process of determining the light intensity reaching the camera from a point on a visible surface. *Shading* is the process by which each pixel (picture element) of the rendered image is set to some value

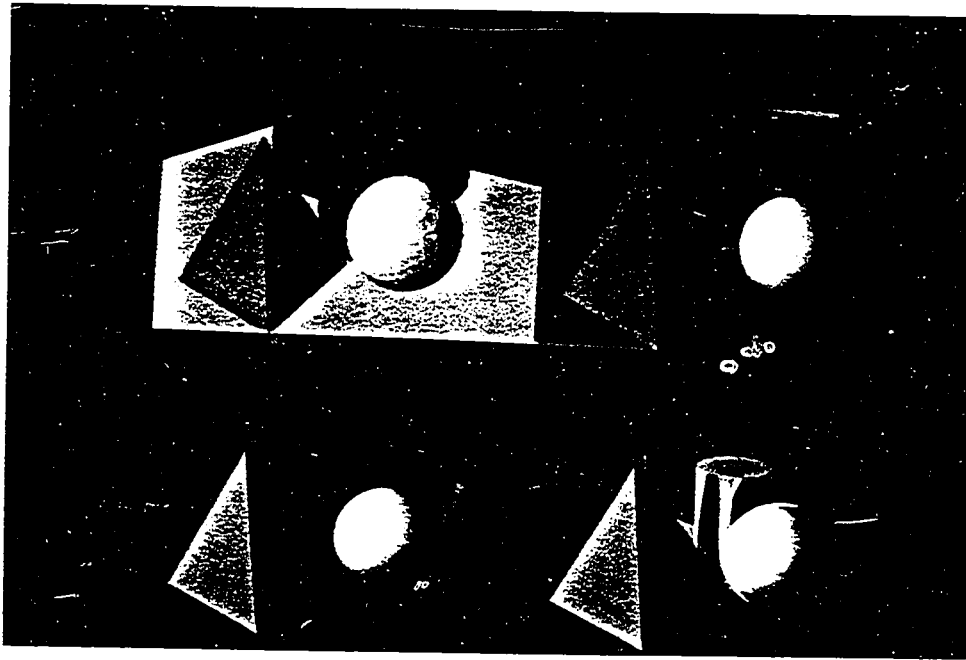


Figure 18: Rendering comparison. Clockwise from upper left, software depth buffer, hardware depth buffer, ray tracing, scanline

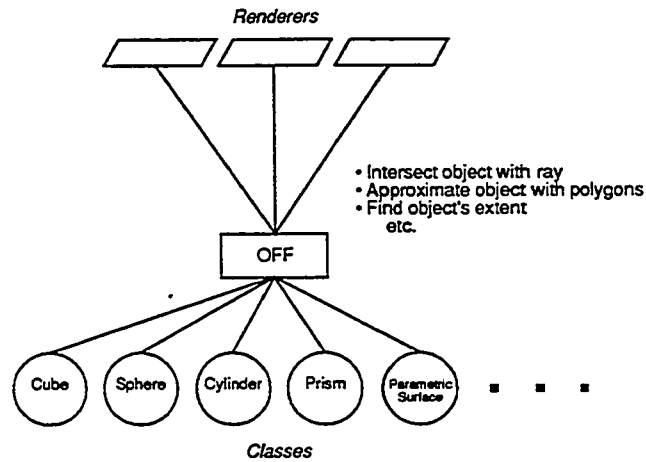


Figure 19: OFF interface between renderers and object classes

related to the intensity reaching the corresponding point on the camera's film area. Shading is tied in with the particular rendering method, but all renderers in BAGS use the same lighting model.

Other Models

The realism of an image is dependent largely upon the quality of the lighting model. Most models compute the light intensity reflected from a surface point as a function of the relative geometry of the light, view, and surface, as diagrammed in Figure 20. More complex models use additional information, as described later.

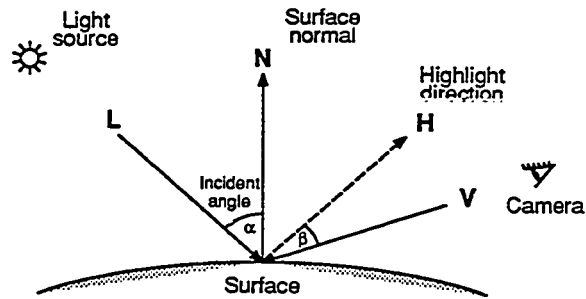


Figure 20: Lighting geometry

Early renderers used a model based on Lambert's cosine law, which states that the light intensity, I_r , reflected from a surface point to the eye is equal to the incident intensity, I , times the product of the diffuse reflectivity, r_d , of the surface and the cosine of the angle of incidence, α :

$$I_r = I r_d \cos \alpha.$$

This model is still used in some graphics systems because it is easy to implement and fast to compute. The cosine can be computed as the dot product of the surface normal, N , and the unit vector, L , from the surface point to the light source. An example of Lambert lighting is shown in Figure 21.

Since the Lambertian reflected intensity is independent of the view direction, this model tends to produce diffuse, "chalky" surface shading. Specular

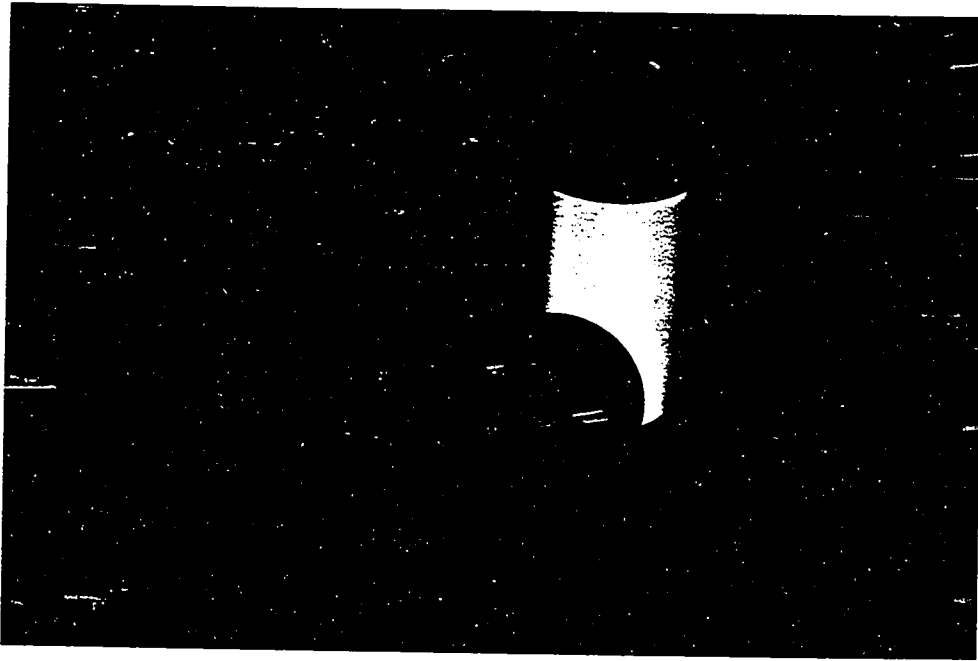


Figure 21: Lambert lighting

(shiny highlight) reflections from smooth surfaces are ignored. The Phong lighting model[12] adds these reflections:

$$I_r = I \left[r_d \cos \alpha + r_s \cos^k \beta \right],$$

where r_s is the specular reflectivity of the surface, and β is the angle between the direction of view and the highlight vector, H , the vector along which light from the source will travel after reflecting from the surface point. The cosine can be computed as the dot product of H and V . The exponent k is a number

that corresponds to the “shininess” of the surface; the higher the exponent, the shinier the surface and the smaller the spread of the specular reflection. Figure 22 is an example of Phong lighting.

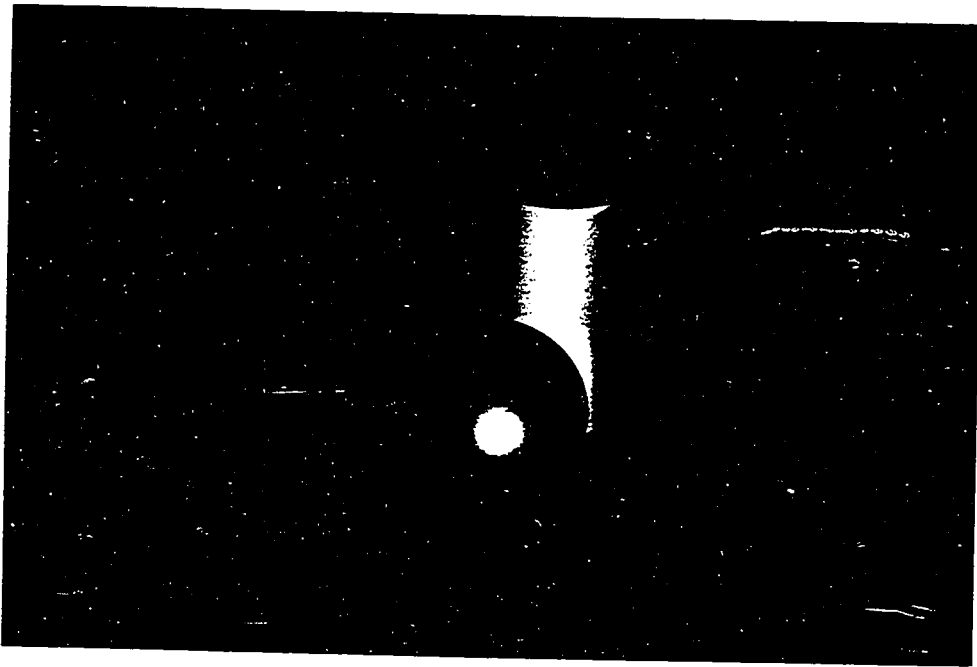


Figure 22: Phong lighting

The Phong model is slower than the Lambert, since it is usually implemented with Phong shading[12] for reasonable fidelity. Phong shading requires the interpolation of the surface normal at each image pixel, a relatively slow operation. Lambert lighting, on the other hand, can be used with Gouraud

shading[37], in which pixel intensities are interpolated. Advances have been made in improving the speed of Phong shading[10], and more hardware systems with Phong capability are appearing. However, many hardware devices still provide only Gouraud shading, even with Phong lighting.

One problem with Phong lighting is that a user of the model is required to select the values defining a surface material and to make sure that they are consistent. Because the values of r_d , r_s , and h are independent, there is nothing to prevent a user from creating inconsistent, and therefore unrealistic, materials. For example, a surface could be defined with a high specular reflectivity and a small specular exponent, meaning that the surface material is very smooth, causing most of the incident light to be reflected specularly, but is not smooth enough to have a small (shinier) highlight. The sphere in Figure 23 is an example of an inconsistent object.

Furthermore, the Phong model is not very realistic. Several improvements have been made, each bringing image quality closer to photorealism. Blinn[11], Whitted[83], Torrance and Cook[17], and Hall[43] have all developed improvements to computer lighting models. One important inclusion is the Fresnel formula, which accounts for the change in reflectivity of a surface as the angle of incidence varies. This accounts for the effect known as *grazing incidence*, when the light direction is nearly perpendicular to the surface normal, causing the

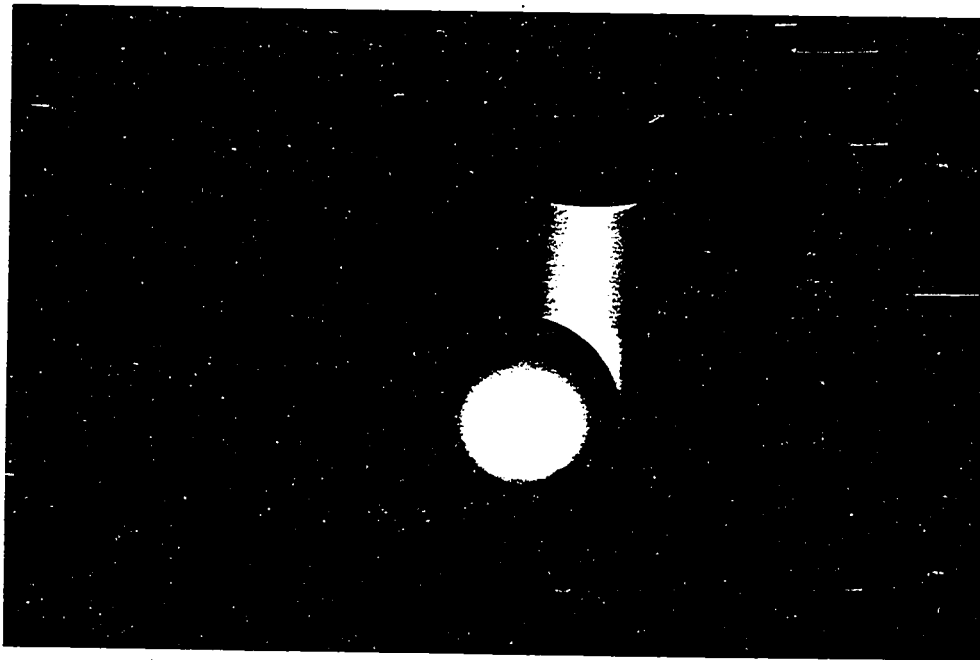


Figure 23: Phong lighting of an inconsistent object

reflectivity to approach unity (perfect reflection). Torrance and Cook also modeled the difference in reflection from metallic and non-metallic surfaces. An example of Torrance-Cook lighting is shown in Figure 24.

Each of these advanced models represents a slightly different approximation of physical phenomena, based on some set of input parameters. In some models, such as Phong, the input parameters are defined solely to make the model work, without regard to user-friendliness. For example, the specular

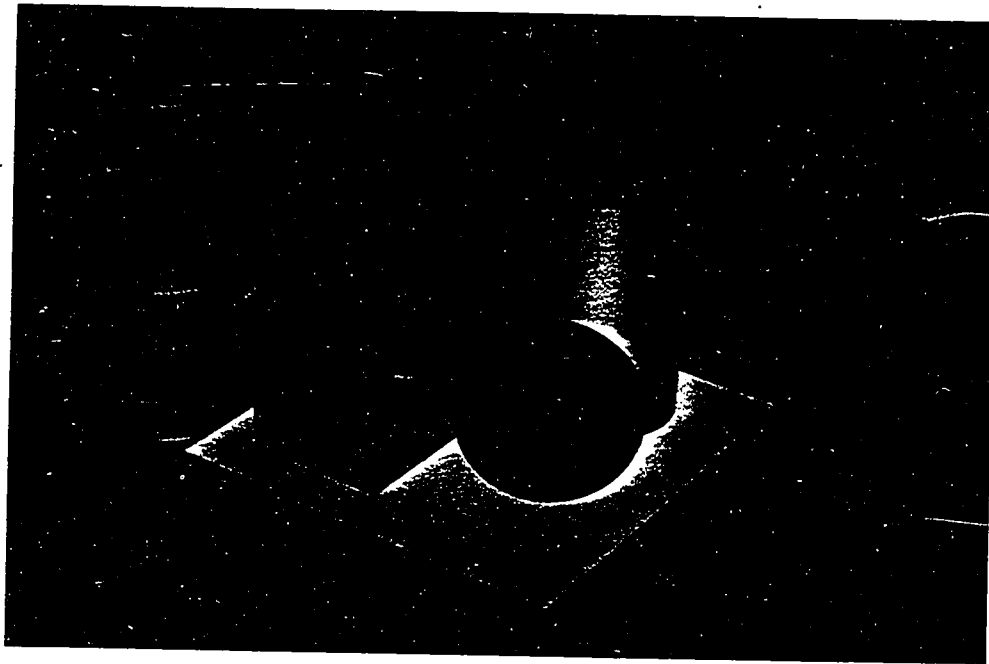


Figure 24: Torrance-Cook lighting

reflectivity and specular exponent are numbers that have to be guessed for a particular surface. Furthermore, the guesswork can only be done by someone familiar with the model. The more realistic models require more precise input parameters that are typically derived from materials science. For example, the Torrance-Cook model requires the specification of the microfacet slope distribution for a surface, which is a measure of how rough the surface is at the microscopic level.

One problem with these models is that a naive user would have difficulty specifying the parameters for a surface. Most non-technical animators would not know the microfacet slope distribution or even the specular reflectivity of a surface. While most common materials used in images, such as copper and glass, could be predefined by a system, these models would limit the animator's freedom in defining surface materials.

The BAGS Model

The lighting model used in BAGS is designed to be used by non-technical animators. It can be interpolated easily and is flexible. These two features are very important in an animation system: the former is needed so that it is possible to animate changing surface properties, while the latter is needed to model nonuniform and nonstandard materials.

The realism produced by the model is high when compared to Phong lighting, as shown in Figure 25. It is accomplished by modeling the following effects in addition to conventional Lambertian/Phong lighting:

- *Fresnel's formula*: The reflectivity of a surface increases as the angle of incidence approaches the grazing angle.
- *Geometric attenuation*: Fresnel reflectivity is attenuated by the geometry of the surface, so that rougher surfaces do not reach perfect reflectivity

near the grazing angle.

- *Metallic vs. non-metallic reflection*: Specular reflections from metallic surfaces usually take on the color of the surface material, while those from non-metallic surfaces take on the color of the light source.
- *Snell's Law*: Light refracting through a transparent surface refracts according to Snell's law; in some cases the light is not transmitted, but instead reflects internally.
- *Conservation of light energy*: The sum of the light intensities absorbed by a surface, reflected from it, and transmitted through it should equal the incident intensity.

These rules are all internal to the system; the user does not have to make any effort to obey them. The realistic effects are all derived from an intuitive set of input parameters. The following description is intended to provide an overview of the way in which these effects are achieved; Appendix B contains a more complete description of the lighting model.

The input parameters to the model are the view direction, V ; the red-green-blue (RGB) light source intensity, I , and direction, L ; the surface normal, N ; and a set of five material parameters. These are the RGB surface color, C ; smoothness, s ; "metalness",* m ; transparency, t ; and index of refraction, n . All

*We realize that "metalness" is not a real word. "Metallicity," "metallicness," "metalositude," and "metallaceousness" are no better.

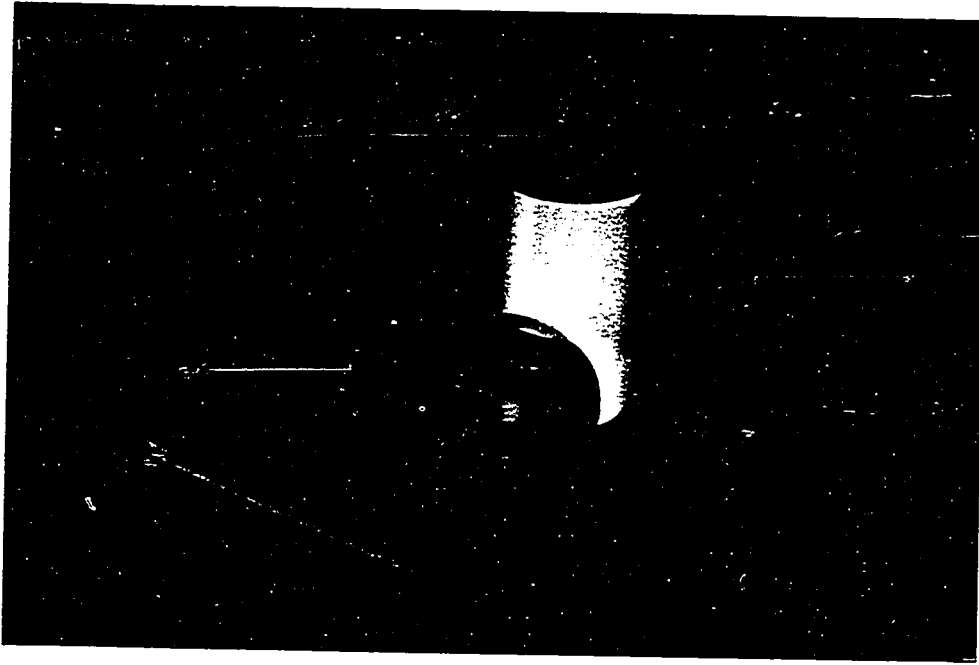


Figure 25: BAGS lighting

of these parameters are numbers ranging from 0.0 to 1.0, except for the index of refraction, which is a physical constant (e.g., 1.5 for certain types of glass).

A smoothness of 0 indicates that the surface is diffuse and chalky, while a smoothness of 1 gives a perfect mirror, reflecting all incident light along the highlight vector. Figure 26 shows sixteen spheres of varying smoothness, from 0 at the upper left to 1 at the lower right. The smoothness parameter is used to compute the diffuse reflectivity and specular reflectivity at normal incidence.

Because Fresnel effects are modeled, the specular reflectivity is not constant, but instead depends on the geometry of the normal, the view, and the light source.

The lighting model computes all of these effects from the smoothness and the lighting geometry. Other models require extra effort on the user's part to make sure that the effects look correct together. With the Phong model, for example, the diffuse and specular reflectivities are typically specified by the user, as is the specular exponent. Finding the right combination of these parameters to produce a realistic material can be time-consuming for an animator. With the BAGS model, these effects are coupled by default; advanced users can decouple them when desired.

The “metalness” is used to differentiate between metallic surfaces (value 1) and non-metallic surfaces (value 0). Specular reflections from metallic surfaces take on the color of the surface material, while reflections from nonmetallic surfaces remain the same color as the incident light source. Figure 27 shows two spheres, one plastic (metalness 0) and one metal (metalness 1). Metalness values between 0 and 1, while non-realistic, are valid as well.

The transparency parameter denotes how much light passes through the surface of an object. A value of 0 defines a completely opaque surface. However, a transparency of 1 does not mean that all light passes through the surface. First of all, the surface may be smooth, and thus reflect some light from

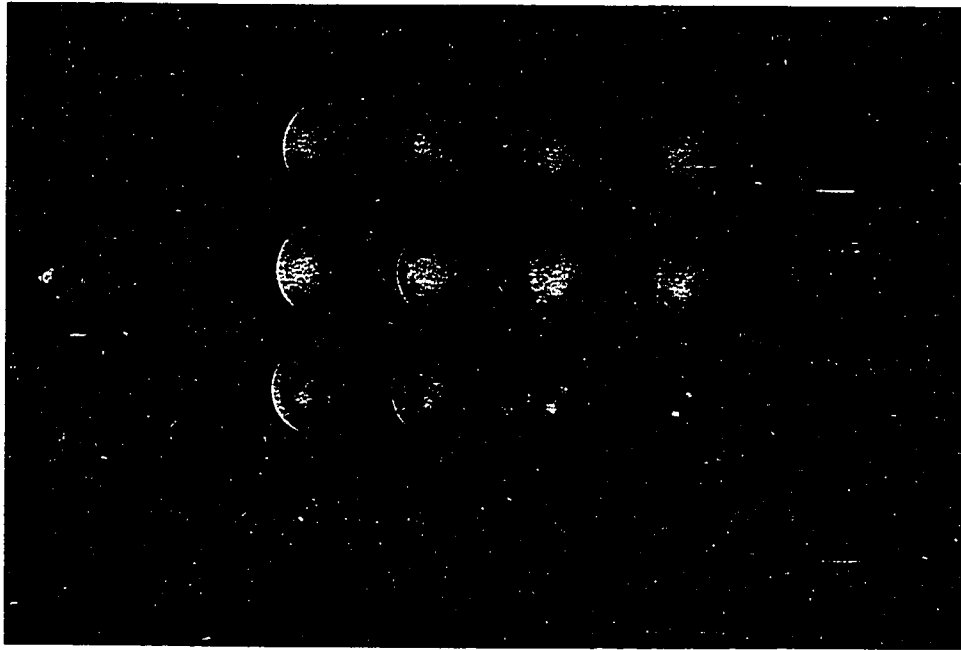


Figure 28: Spheres of varying smoothness: 0 at upper left, 1 at lower right

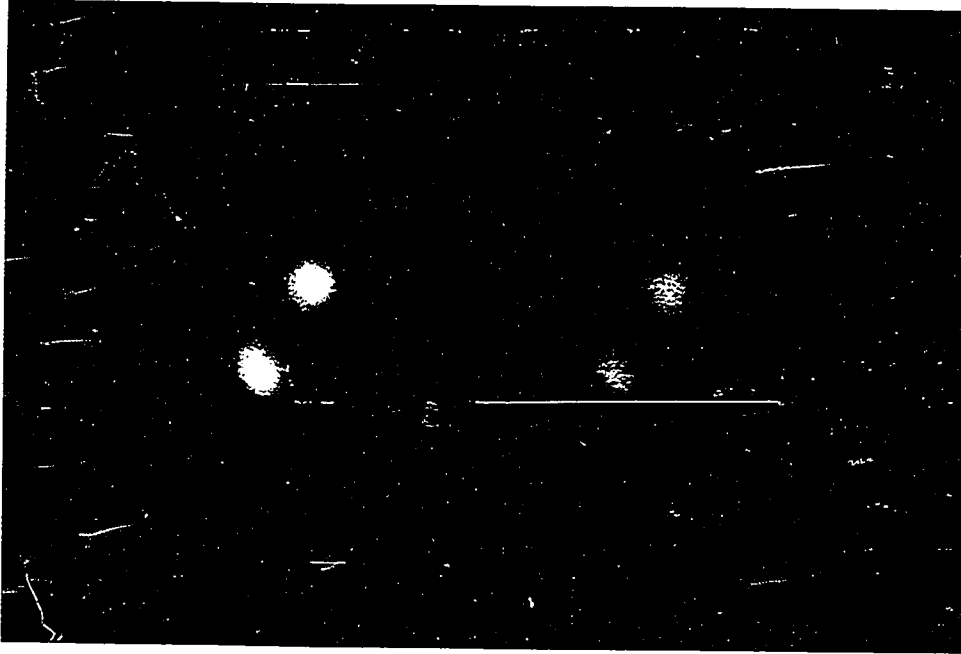


Figure 27: Plastic sphere, metallic sphere

the surface. All surfaces reflect light specularly near the grazing angle, so transparency effects diminish in those circumstances. Figure 28 shows the effect of varying the transparency parameter, again from 0 at the upper left to 1 at the lower right.

The index of refraction is a physical constant that determines how a light ray is deflected when passing through the interface of the surface material with air. Since this number represents a ratio between the speed of light through the

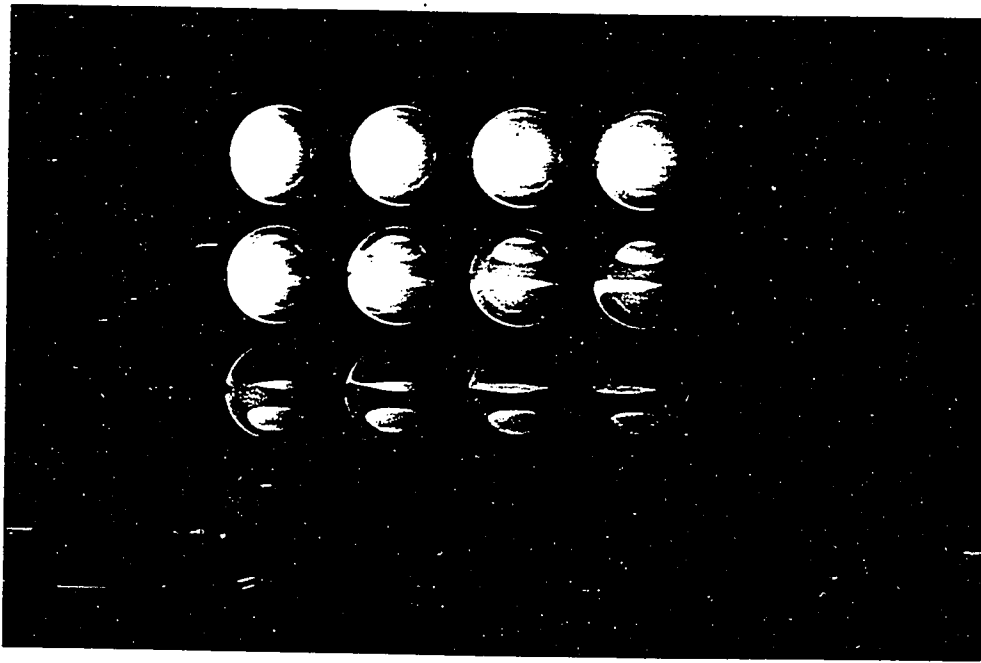


Figure 28: Varying transparency: 0 at upper left, 1 at lower right

two materials on either side of the interface, it cannot be constrained to lie between 0 and 1. An index of 1 is that of a vacuum (or air, for all intents and purposes); a surface with that index would not bend light rays at all. The spheres in Figure 29 have indices of refraction varying from 0.1 at the upper left to 1.6 at the lower right. The second sphere from the left in the second row from the top has an index of 1.

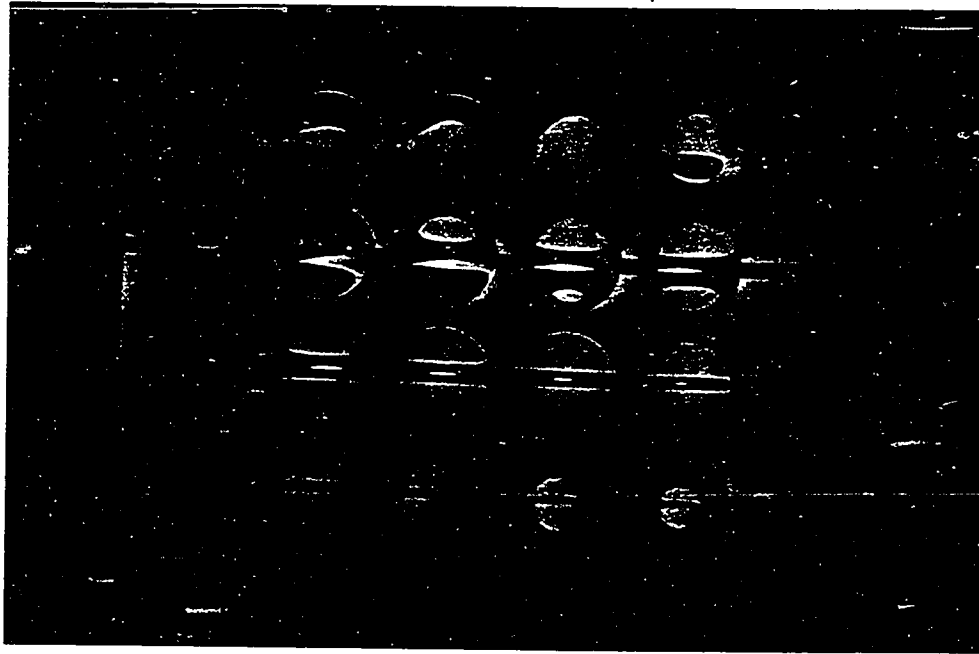


Figure 29: Varying index of refraction, from 0.1 to 1.6

The implementation of the BAGS lighting model computes the reflected intensity in a series of steps. Each computation results in an intermediate value corresponding to some aspect of the lighting model. Some of the more important values are described below.

The diffuse reflectivity, r_d , is the reflectivity of a surface with respect to diffuse light. The equation

$$r_d = (1 - s^3)(1 - t)$$

gives the value. Smooth or transparent surfaces have lower diffuse reflectivities, since they reflect specularly or transmit more of the incident light. The product of the diffuse reflectivity with the cosine of the angle of incidence (computed as a dot product)

$$l = (N \cdot L)r_d$$

yields the Lambert lighting term, l . The diffuse contribution, K_d , is then

$$K_d = lC,$$

the product of the Lambert term and the surface color.

Modeling the specular contribution is a little more complicated. The highlight vector, H , is first computed from the light direction vector, L ; the angle of incidence; and the surface normal:

$$H = L + 2(N \cdot L)N.$$

This formula is used when computing the specular angle, β , between the highlight vector and the view vector. The cosine of this angle is raised to the power of the specular exponent h . This exponent is computed as

$$h = \frac{3}{1 - s},$$

which is an empirically derived formula. The result of raising the cosine to this power is called the specular spread, p :

$$p = \left[- (H \cdot V) \right]^h.$$

This value is, in effect, the amount by which the specularly reflected light will

spread from the highlight direction and, consequently, determines the size of a specular highlight. The spread is similar in effect to the specular exponent in the Phong model but does not require any guesswork on the user's part.

Fresnel reflectivity is modeled by the f term, which is based on F , a function of the angle of incidence. The original formula was derived by Fresnel for materials with a known index of refraction. For other materials, some form of approximation is needed. Torrance and Cook[17] suggest measuring reflectivity at normal incidence and fitting a Fresnel-like curve to the results. Our approach uses an approximation to the Fresnel formula:

$$F(\alpha') = \frac{\frac{1}{(\alpha' - k_f)^2} - \frac{1}{k_f^2}}{\frac{1}{(1 - k_f)^2} - \frac{1}{k_f^2}},$$

where $\alpha' = \frac{\alpha}{\pi/2}$ to bring it in the range from 0 to 1, and k_f is a constant that tailors the curve to fit the actual Fresnel curve. Figure 30 shows a comparison of the Fresnel curve and the BAGS approximation, with a k_f value of 1.12. Because we already have found the cosine of α , the angle of incidence, we use a slightly different version of the F function which takes the cosine as an operand.

The Fresnel reflectivity is attenuated for rough surfaces by geometric obstruction of light. Torrance and Sparrow[79] formulated a theory to account for this effect, and Blinn[11] applied the theory to computer graphics. Blinn's formulation was subsequently used by Torrance and Cook in their model.

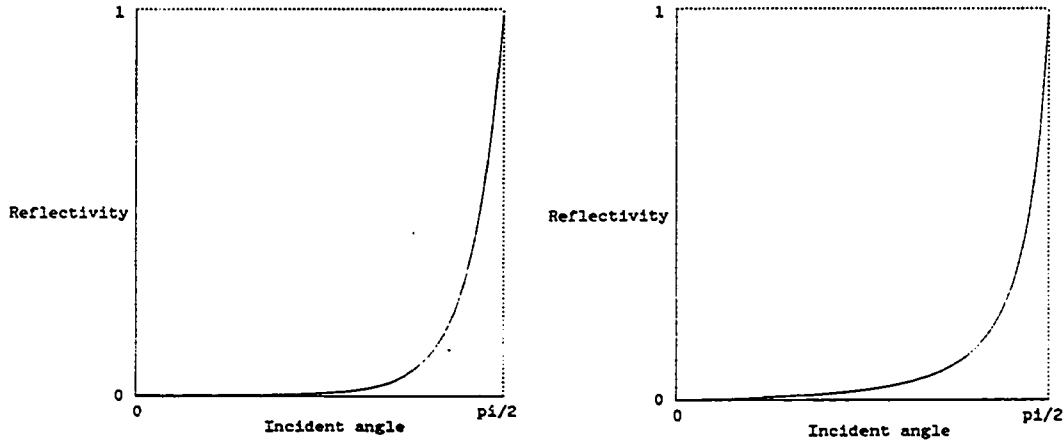


Figure 30: Comparison of Fresnel formula curve and BAGS approximation

Unfortunately, the geometric attenuation factor based on this theory accounts for only certain lighting geometries; there are cases in which the model falls apart drastically[22]. The geometric attenuation factor g in BAGS is based on an approximation similar to that of the Fresnel curve, avoiding the discontinuity:

$$G(\alpha') = \frac{\frac{1}{(1 - k_g)^2} - \frac{1}{(\alpha' - k_g)^2}}{\frac{1}{(1 - k_g)^2} - \frac{1}{k_g^2}},$$

where k_g is another constant, again affecting the curve's shape. The curve is graphed against the incident angle in Figure 31 for a k_g value of 1.01. As with

the Fresnel curve, the geometric attenuation is computed from the cosine of the incident angle, instead of from the angle itself.

The Fresnel term causes the reflectivity to increase as the incident angle approaches grazing incidence, while the geometric attenuation factor attenuates the reflectivity at the same time. Because the constants for the two curves differ, the effects do not merely cancel each other out. The product of the two factors

$$a = fg$$

is called the reflectivity adjustment, and is graphed in Figure 31.

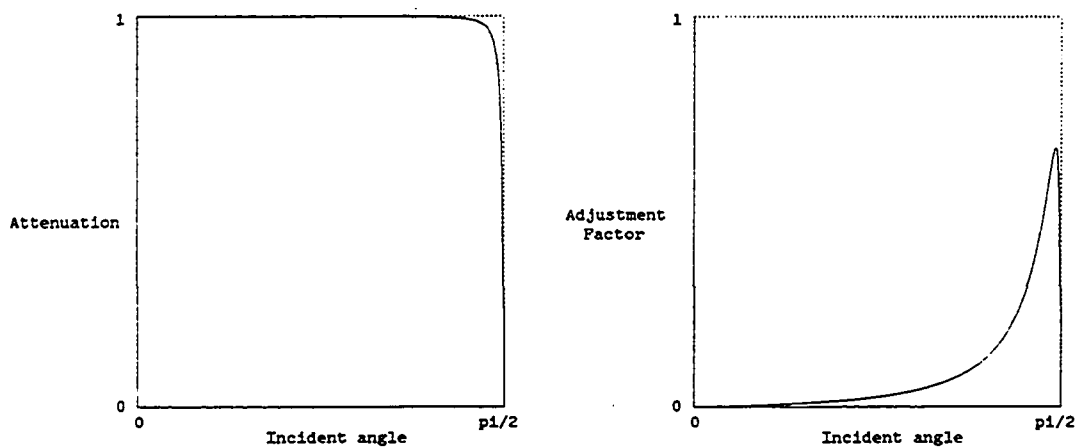


Figure 31: Geometric attenuation curve and adjustment curve

The adjusted reflectivity, r_a , is based on the normal reflectivity, r_n , but also takes into account the specular spread and adjustment:

$$r_a = r_n + (1 - r_n)a.$$

Because the value of a is 0 at normal incidence (i.e., when $\alpha = 0$), the value of r_a is the same as r_n , which is defined as the reflectivity at normal incidence. As the incident angle increases, the adjusted reflectivity increases towards unity like the a curve does.

The product of the adjusted reflectivity and specular spread

$$r_s = r_a p$$

is the specular reflectivity. It is the amount of light that reflects specularly from the surface along the direction of view.

The color of the specular reflection depends on the metalness, m ; the angle of incidence; and the color of the surface. Specular reflections from metallic surfaces tend to be the color of the surface, except when Fresnel reflectivity increases rapidly (we can use the attenuation factor to determine when this occurs). The specular color is then

$$C_s = C_1 + m(1 - a)(C - C_1),$$

where C_1 represents pure white, with red, green, and blue values of 1. This formula gives a color somewhere between the color of the surface and pure white; this color will be multiplied by the color of the light source to arrive at the final specularly-reflected intensity.

The specular contribution K_s is then

$$K_s = r_s C_s,$$

the product of the reflectivity and the color. This value is added to the diffuse contribution, and the result is multiplied by the incident light intensity,

$$I_r = I(K_d + K_s),$$

to arrive at the reflected intensity.

The effects described so far have only been local. That is, the reflected intensity is a function only of information from the surface point, light, and view. For more realism, global information can be included as well. Whitted introduced this notion in his seminal paper on ray tracing[83]. Adding the contributions of reflected and transmitted rays to the intensity calculation for a surface point increases image realism tremendously.

The BAGS lighting model incorporates these global lighting effects when possible, which is typically only when rendering with ray tracing. The computation of a reflected ray is similar to that of the highlight vector, H . The contribution of the intensity along this ray is attenuated by the adjusted reflectivity of the surface with respect to this ray. This value can be computed in the same way as r_a for the local lighting, with the reflected vector substituted for the incident light vector.

Refraction of light through a transparent object follows Snell's law. Kay and Greenberg devised a clever way to compute the transmitted ray according to this law, with minimal computational overhead[51]. BAGS uses this technique and also accounts for total internal reflection, a situation in which Snell's law has no solution. This effect, which is essential to realistic lighting, occurs when the incident angle exceeds the critical angle, based on the index of refraction of the surface material. The contribution of the transmitted ray to a surface point is attenuated by the transmittance, z , of the surface at that point. This value is proportional to the surface transparency, t , but decreases as the reflectivity increases, obeying the conservation law:

$$z = \frac{(1 - r_r)t}{1 - r_n},$$

where r_r and r_n are computed with respect to the incident view ray, not the light ray. At normal incidence, the transmittance is equal to the transparency of the surface. At larger incident angles, the reflectivity increases, so the transmittance decreases proportionately. This attenuation produces more realistic transparency effects than are found in lighting models, such as Phong, that do not account for Fresnel reflectivity.

Implementation

The BAGS lighting model is implemented as a software package called LASH (Lighting And SHading)[75]. The implementation is designed to provide realistic shading as the default and to allow users to modify the lighting model as desired.

Each of the input parameters to the lighting model (color, smoothness, etc.) is specified in a **change** statement for an object in a SCEFO script. (If none are specified, reasonable default values are used.) A parameter may be set to a constant value for the entire object or can be a user-defined function. Functions may depend on other surface parameters. For example, a user can specify that the color of a point on an object depends on the coordinates of that point. User-defined functions have specific calling sequences, much like the other extensibility features of BAGS described in Chapter 3.

The input parameters to LASH represent *basic items*, which are used to find *computed items*, such as the final reflected intensity. The computations proceed in a given order, based on dependencies on other items. For example, the specular reflectivity depends on the specular spread, so the spread must be computed first. Any of the computed items may be set by a user from a SCEFO script, in the same way the basic items are set. For example, a user could set the specular reflectivity of an object to have a constant value of 0.3. LASH

would know not to compute that item or any item it depended on.

Functions applied to items may need to modify the order in which items are computed. LASH provides a way (through the SCEFO language) to specify that a particular item depends on one or more other items, so the correct ordering may be achieved.

Many common and useful surface modification functions are provided by the PMAP pattern mapping package[80]. *Pattern mapping* is the process of modifying surface parameters to produce or simulate surface detail. For example, a mathematical function can be used to modify the color of a surface, producing a paint- or wallpaper-like effect. Surface normals can be mapped to simulate fine geometric features. LASH allows any of the input parameters to be modified in this way. Figure 32 shows two pattern-mapped cylinders. A function was used to modify the surface color of the cylinder on the left, while a similar function was used to modify the surface normals of the cylinder on the right. The “dents” in the cylinder on the right are produced by lighting, not by geometric features; the smooth silhouette of the cylinder’s edge exposes this trick.

Pattern mapping is used most often to wrap an image stored as a bitmap around an object. Bitmaps containing rendered images, painted pictures, or digitized (scanned-in) surface textures are often mapped onto surfaces to add

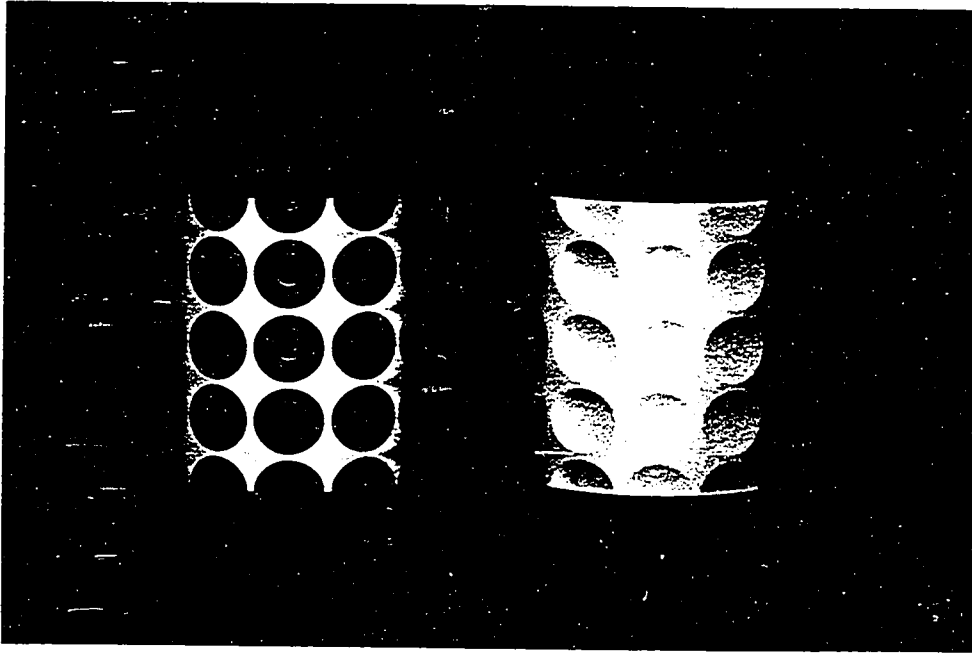


Figure 32: Pattern mapping examples: color mapping and normal mapping

detail without increasing geometric complexity. An example of this technique is illustrated in Figure 33. The image at the upper left of the image has been mapped onto the surface of the sphere at the lower right.

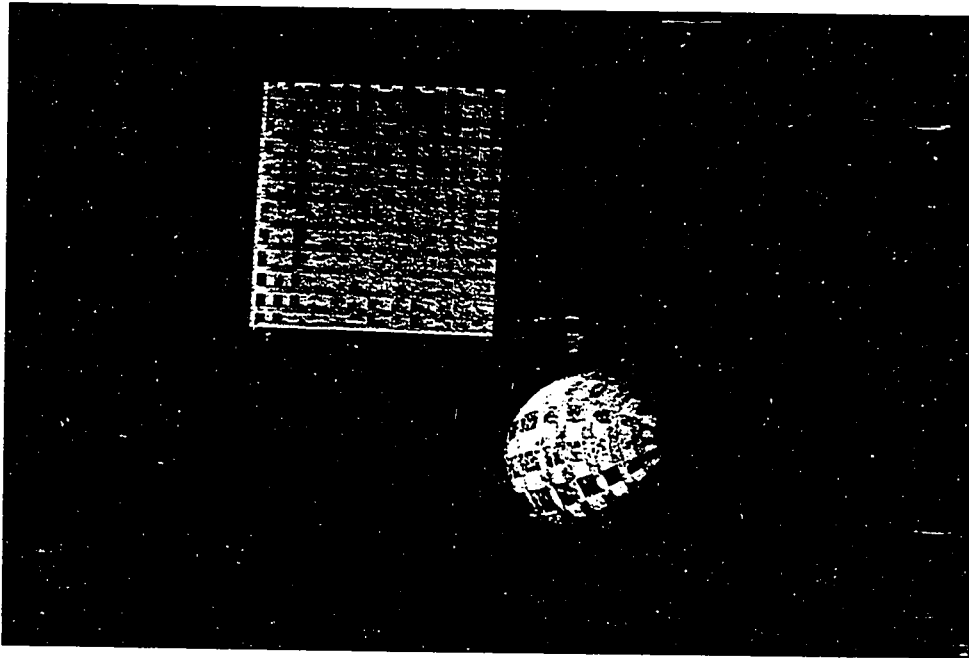


Figure 33: Mapping a pattern from a stored bitmap onto a sphere

Polygonal Approximation

Most rendering methods in use today treat objects as collections of polygons. Polygonal approximation makes objects easier to deal with: polygons can be constrained to be planar, non-intersecting, or even convex. They can be sorted to solve the visible surface problem. They can be flat-shaded for fast rendering or interpolated for smoother shading.

BAGS includes several rendering techniques that require a polygonal version of a model. Most of the standard methods are included, such as wire-frame, depth buffer (hardware and software), and the painter's algorithm. There is also a scan-line rendering method that incorporates antialiasing and transparency, can handle pattern mapping, and produces high-quality images[81].

Since models are stored in SCEFO as render-independent sets of objects, the objects must first be approximated. Each object class can produce such an approximation for any object in that class. Constructive solid geometry (CSG) combinations of polygonal objects must be computed; a robust algorithm for doing so has been developed for use in BAGS[54].

One disadvantage of the modeling-rendering separation in BAGS is the performance penalty that stems from the need to recompute the rendering for each frame from the model. However, the system includes some escape mechanisms to alleviate this problem. First of all, there is coherency checking built into the software responsible for accessing polygonal versions of frames. If an object does not change from one frame to the next, except by rigid transformations, the same representation of the object may be used for both frames. Similarly, two objects within the same frame that are similar enough can share the same representation. Also, an animator may create a polygonal version of an often-rendered object and use that version for quick preview renderings.

On the other hand, an advantage to this philosophy is the ability for the representation of the model to change over time. A good example of this involves *polygonal resolution*. The resolution of a polygonal approximation is a measure of the number of polygons used. For example, a cylinder may have any number of polygons forming its shell, plus one for the top and one for the bottom. The number of polygons to use depends on the appearance of the cylinder in the scene. If it occupies a large part of the image, many polygons are needed to keep the profile of the cylinder smooth. If it is small, or far from the camera, fewer polygons suffice. If the choice of resolution is made during modeling, the number of polygons must remain constant. Either there will be too few polygons when the cylinder is large in the image or too many when it is small. In the former case, the image may look bad. In the latter case, too much time may be spent rendering an object that does not contribute significantly to the image.

In BAGS, the cylinder is stored as a conceptual object, and the choice of resolution can be delayed until rendering time. At that time, the importance of the object with respect to the image can be determined and a resolution computed automatically by the renderer. If frame coherency is being used, and the resolution does not change significantly from frame to frame, the same representation of an object may be used more than once.

Ray Tracing

Ray tracing is a very straightforward and concise rendering technique. In its simplest and purest form, it is easy to understand and implement. Unfortunately, it is also slow and prone to aliasing (i.e., artifacts produced by insufficient sampling). But since ray tracing does produce near-realistic images, it is worthwhile to include it in the rendering arsenal, and it is worth spending extra programming time for optimization and improvement.

The basic ray tracing algorithm as developed by Appel[3] and Whitted[83] determines visible surfaces by tracing rays from the camera. One ray is sent through the center of each image pixel (Figure 34). The ray is tested against each object in the scene for intersections. If no intersections occur, the pixel takes on the value of the background. If at least one object is hit, the intersection point of the object closest to the camera is used for the pixel, after lighting.

Much of the realism in ray traced images is provided by reflections, refractions, and shadows. If an intersected object's surface is smooth, a secondary reflection ray may be propagated from the intersection point. If that ray intersects an object, it is shaded and a proportionate amount of the result is added to the pixel through which the primary ray was sent. Similarly, transparent objects may transmit refracted rays through their surfaces. Figure 35 illustrates these secondary rays. This process can recurse to any desired depth, producing

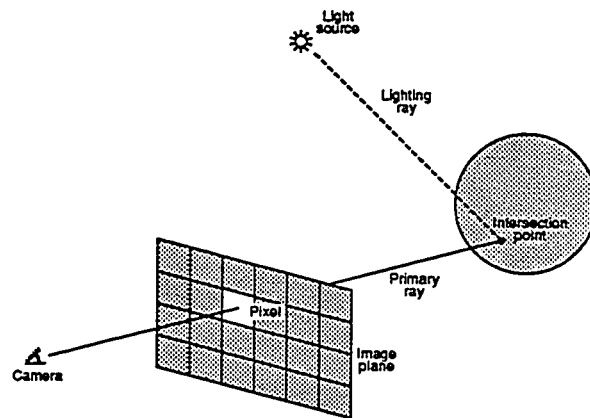


Figure 34: Generic ray tracing procedure

a tree of rays.

Shadows are implemented with some extra work prior to the lighting computations. When considering the contribution of a point or directional light source to the lighting at a surface point, a ray is sent from the surface point to the light source. (For a directional source, the ray is sent in the direction opposite to that of the light.) If the ray intersects any object before reaching the light source, the surface point is in shadow with respect to that source (Figure 36).

While it is possible to ray trace polygonal models, it is far more accurate, straightforward, and efficient to use a procedural model, in which each object is

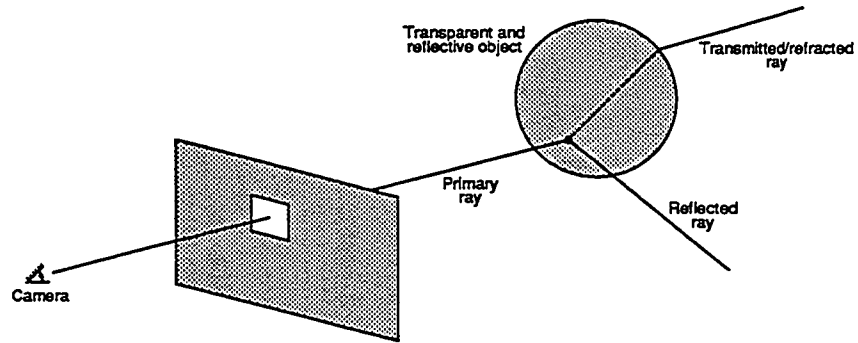


Figure 35: Tracing secondary rays

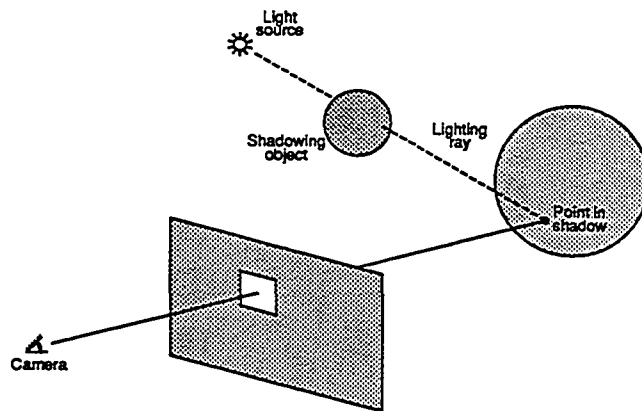


Figure 36: Ray tracing with shadows

represented by code that intersects a ray with the object. The SCEFO represen-

tation and object-oriented OFF design provide such an interface. The ray tracer in BAGS requests OFF to intersect rays with objects, returning the point of intersection and the surface normal at that point. The class-specific code is optimized for efficient object intersection calculations.

It has been shown that CSG is comparatively easy to handle in a ray tracing algorithm[68]. Boolean operations on objects are reduced to one-dimensional cases (Figure 37) that are easy to compute. The model representation in BAGS accommodates this algorithm, since the CSG combinations are not resolved until the scene is rendered.

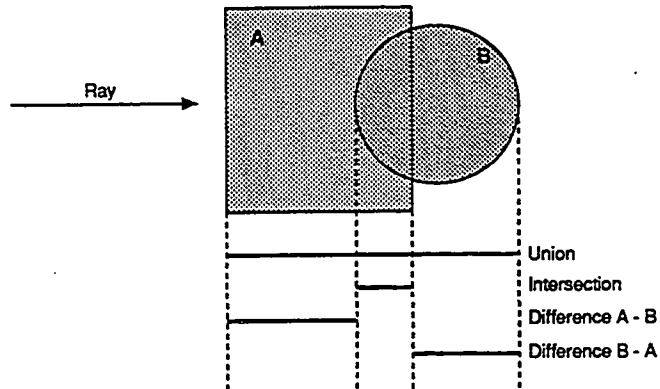


Figure 37: CSG operations along a ray

Speed Optimization

The main drawback of the ray tracing technique is its relative slowness. A ray traced image can take hours to compute, whereas a polygonal version may be finished in a few minutes. However, the extra computation time is justified by the substantial increase in image quality. The quest for realism has prompted researchers to improve the performance of the ray tracing algorithm.

The primary area of improvement has come in ray-object intersection testing. In the naive algorithm, every object is tested for intersections with every ray. The use of object extents, coupled with space or object subdivision, can increase execution speed dramatically by reducing the number of complex intersection tests. Various methods for doing so have been implemented [50, 30, 29, 52].

However, none of these optimizations is ideal for use in BAGS. First, all the space subdivision algorithms partition a scene based on the intersections of object surfaces with cuboid extents. In a system with only simple primitive objects like spheres and cylinders, the preprocessing time needed for such intersection testing is relatively small when compared with the great savings during ray tracing. This is not true in BAGS; some object classes, such as parametric surfaces, would require so much time for extent intersection testing that this approach would be intractable.

The other problem with these methods is handling CSG. None of the optimization methods mentioned above explicitly includes CSG combinations of objects. Glassner has addressed the issue of optimizing CSG ray tracing[31]. He uses a bottom-up approach in which rays are intersected first with the leaf objects of CSG trees. If an intersection is found, the processing passes up the CSG tree to the top node. If this propagation is successful, the ray intersects the top-level CSG object. The bottom-up procedure is suitable for a model with few CSG objects but not for one that makes heavy use of CSG, such as BAGS[33]; the subdivision tree quickly becomes cluttered with leaf objects, many of which are subtracted from other objects and therefore never appear in the image.

The optimization technique used in BAGS combines extent-based space subdivision with a top-down CSG approach. The rectilinear extents of all non-CSG and top-level CSG objects are used to create a 3-D tree of cells. Each plane of a cell is derived from a plane of an object extent. The object extent planes are sorted, and the dimension with the most planes is chosen to partition the next level in the 3-D tree. The centermost plane in that dimension is used to divide the object extents. The procedure recurses until all planes are used.

Each cell in the resulting tree contains pointers to all objects whose volume occupies that cell. Typically, there is a small number of objects per cell. The

starting point of a ray can be located in the tree by binary search. Each object in the cell containing that point can be compared with the ray. The tree cells are linked to allow a ray to pass from one cell to the next in constant time. At each cell along the way, the ray is tested against objects. The cell traversal can stop as soon as an intersection is found.

If all CSG leaf objects were used to build the subdivision tree, the tree would quickly become too cluttered to be efficient. Also, objects with no effect on the final image would fill cells; a very large object used only in a difference operation to cut away part of another object would be stored in the tree, even though it is never seen. Thus, only the top-level objects are used. The extent of each top-level object is computed from the extents of the sub-objects.

When a ray intersects the extent of a CSG object, the ray is tested against the sub-objects in the CSG tree. A few optimizations are used to avoid unnecessary intersections. For example, a ray does not hit the Boolean intersection of two objects if the first object tested is not hit. Union operations are preprocessed to divide the extent into sub-extents, each of which contains neither, one, or both of the operand objects.

These optimizations are all performed in software. Other work has been done to parallelize ray tracing code and to develop special-purpose hardware. Using such techniques, images that previously took hours to render can be ray

traced in minutes, or even seconds. These advances further justify the inclusion of ray tracing renderers in BAGS.

Antialiasing and Caustics

As ray tracing performance increases, the other problems with the algorithm become more noticeable. One such problem is called *aliasing*, and is most noticeable as a staircase effect at object boundaries. The reason for this effect is the inherent point-sampling nature of ray tracing. If one ray is sent through each pixel, that pixel will either have the color of an object, if the ray hits it, or the background color, if the ray misses it. A more detailed discussion of this problem and various solutions is given in Chapter 5.

The other main problem concerns effects that are not modeled by conventional ray tracing. These effects are called *caustics*, the interplay of light reflected from one object to another or transmitted through an object onto another. Chapter 6 addresses this problem.

5. Triangle Tracing

In this chapter we present a new technique that improves the antialiasing performance of ray tracing. Unlike some other methods discussed in the next section, this technique is built on the conventional ray tracing operation of intersecting rays with objects. It does not involve new forms of intersection calculations and can therefore be incorporated with existing ray tracing programs.

The Aliasing Problem

Aliasing may occur in signal processing when the sampling frequency is less than twice the maximum frequency of the sampled signal. Conventional ray tracing samples a scene once per pixel. The frequency of the signal (i.e., intensity reaching the eye) in a scene is potentially infinite, so there is a good chance that aliasing will happen. High-frequency areas, such as where several object boundaries meet, or where a complex pattern is mapped onto a surface, are par-

ticularly problematic.

Figure 38 illustrates how aliases in the form of “staircases” occur in a ray traced image. A ray is sent from the camera through the center of each pixel in the image plane. If the ray intersects an object, the pixel is filled with the shaded color of that object. Otherwise, the pixel is set to the background color. The pixels representing an edge of the cube in Figure 38 depend on whether the ray hits the cube or not. The result is the staircase of pixels shown in the figure.

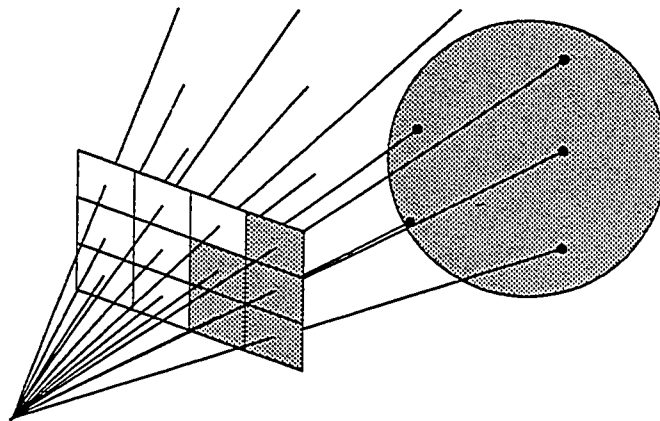


Figure 38: Aliasing in ray tracing

Related work

The correct way to antialias an image is to first *filter*, (remove) from the signal all frequencies that are above half the sampling rate[23]. Convolving the signal with an appropriate function to accomplish the filtering is equivalent to examining the signal over the area of a pixel, rather than at just a single point. The simplest filter of this type is a rectangular box over each pixel, meaning that the intensity contribution of an object to a pixel is equal to the proportion of the pixel's area covered by that object. Catmull used the term *area sampling* for this type of filter[13]. More complex filters, such as the sinc function, may be used for better antialiasing.

Area sampling is fairly easy to implement in scanline polygonal renderers, because enough information about the polygon is known to compute its overlap with each pixel[13]. With ray tracing, however, this cannot be done. Each point being shaded is an abstract point representing the intersection of a ray with an object. The point is zero-dimensional, and therefore does not accurately represent what is visible at a pixel. Non-local information cannot be determined without compensating for the infinite thinness of a ray.

One way to overcome the sampling problem is to send out “thick” rays. Heckbert and Hanrahan at NYIT use a technique called *beam tracing*, which renders an antialiased image of a polygonal model[45]. The initial beam is the

viewing pyramid formed by the camera point and the sides of the film plane. The polygons in the scene are sorted by distance from the beam origin and are then tested for intersection with the beam. The front-most intersecting polygon is stored in a data structure and is subtracted from the beam.

The beam tracing procedure is repeated recursively to produce a *beam tree* of visible polygons. Reflections and refractions are modeled by originating new beams at the point of reflection or transmission. The visible polygons are then passed to a scan-converter to produce the image.

The obvious drawback to this method is the requirement that the model consist of polygons. The beam tracing technique can be considered a way to incorporate the ray tracing lighting model into polygonal rendering, rather than a modification to conventional ray tracing that performs antialiasing.

Rather than using polygonal beams, Amanatides traces circular cones[2]. A cone is defined with the apex at the base point of the ray and a spread angle measured from the ray vector, which is used as the center line of the cone. The spread angle is chosen so that the intersection of the cone with the image plane is approximately the size of a pixel. The cone is tested for intersections with the objects in the scene to produce a sorted list of the eight closest intersected objects. Associated with each intersection is the fraction of the cone's cross-section covered by the intersected object. The shaded point of intersection

is weighted by this coverage factor to arrive at the contribution to the pixel. Reflected and refracted rays spawn new cones from the point of intersection. The spread angles for these new cones depend on the curvature of the surface at that point.

Sampling areas with cones instead of rays produces antialiased images. It also can be used to produce “fuzzy” shadows, glossy reflections, and translucency effects by using area information in shading computations.

However, some difficulty arises in the intersection calculations. Amanatides is successful in computing intersections of cones with spheres, planes, and polygons. Other classes of objects are too complex to allow efficient intersection calculations. Another problem lies with the way partially obscured objects are treated, as explained in the paper. Only the coverage value for each object is retained, so there is no way to determine if two objects that each cover half the pixel are both visible or if just one is. Doing so would require extra information about the objects to be maintained, such as an approximate geometry within the cone or at least a coverage mask. These methods would produce approximate, but more accurate, results, at the expense of even more difficult intersection tests.

Both of the methods just discussed are attempts to solve the ray tracing aliasing problem by enlarging the rays. Because neither of the techniques is

entirely successful, researchers have turned to strategies that trace more rays per pixel. Tracing several regularly-spaced samples per pixel is called *supersampling*, which is a simple way to raise the sampling frequency, thereby decreasing aliasing. However, this technique does not remove aliases in all frequencies.

Another antialiasing sampling technique, called *adaptive supersampling*, was first proposed by Whitted[83]. He traces rays at all four corner points of a pixel. If the four calculated intensities are similar, they are averaged and the pixel is set to the result. Otherwise, the pixel square is subdivided into smaller squares, new rays are traced, and the process is repeated. The recursion continues until some resolution threshold is reached.

This method approaches area sampling in results but uses only point sampling, meaning that the standard ray-object intersections can be used. A problem exists with the pixel subdivision scheme, though. The subdivision always produces squares that are some regular division of the original pixel. Any suitably high-frequency object configuration can exceed the sampling rate imposed by such a regular pattern, causing aliasing.

Cook, et al. describe a different method for supersampling that diminishes the problems caused by a regularly-spaced sampling grid[19,20]. Their approach, called *distributed ray tracing*, sends several sampling rays per pixel, but perturbs each ray spatially by a small amount. The resulting stochastic

sample is less regular than is Whitted's method and therefore converts high-frequency information into noise, instead of into aliases. (This is a complicated phenomenon which is described in much more detail in the literature.) Distributing the same rays in other domains simultaneously can produce glossiness, translucency, soft shadows, depth of field, and motion blur. Methods for optimizing the sampling and for sampling adaptively have been described in the literature[55,26,60]. These methods help attenuate the increased processing time necessary to trace the extra rays.

The Triangle Tracing Algorithm

We now present a new algorithm for producing antialiased ray traced images. It is essentially an area-sampling algorithm, so it can filter the intensity signal over an area of an object's surface. Because it uses only point samples to compute the areas, the conventional ray-object intersection procedures may be used, as opposed to the special-purpose intersections required by cone tracing. However, unless care is taken the point-sampling nature of the algorithm can lead to common under-sampling artifacts, such as motion jump and strobing.

In the basic triangle-tracing algorithm, *vertex rays* are sent from the camera point through the corners of an image pixel. The pixel is divided by a diagonal into two triangles, so that three rays are sent at once, forming a triangular

cone (Figure 39). Each of the three vertex rays is tested for object intersections, as in conventional ray tracing. For each ray, the closest intersected object, if any, is saved.

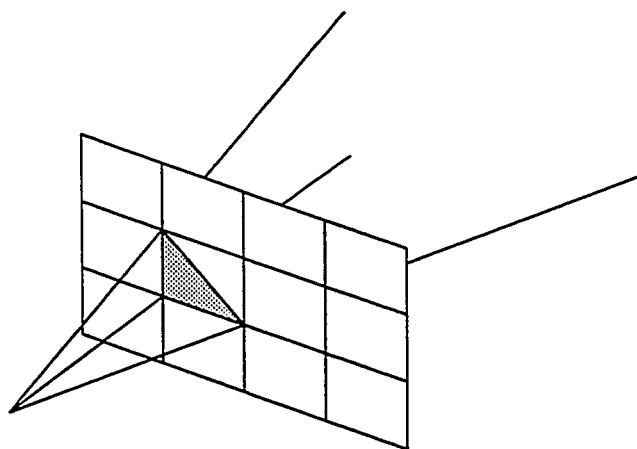


Figure 39: Tracing vertex rays through half-pixel triangle corners

A triangle is considered to be *homogeneous* if none of its three vertex rays hit an object or if all three hit the same object. If none of the rays hits an object, the triangle has no effect on the image and therefore nothing else is done with it. If all three rays intersect the same object, lighting calculations are performed. The intensities computed at the three intersection points are averaged, and the result is halved (to compensate for the pixel division) and added to the

pixel.

If the three rays hit different objects, or if one or two hit an object and the rest do not, the *heterogeneous* triangle is subdivided into smaller triangles. The first step in the subdivision procedure is the computation of *boundary rays*, which are near-identical rays that intersect different objects. The boundary rays intersect the pixel triangle at *boundary points*. Figure 40 shows a pair of boundary rays.

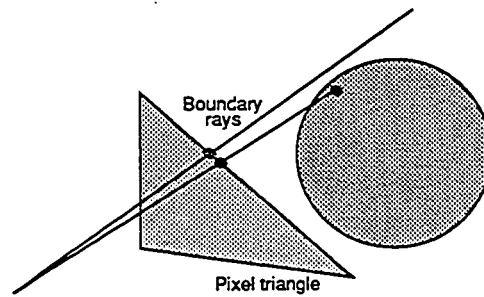


Figure 40: A pair of boundary rays on a triangle edge

The boundary points and rays are found by successive binary division of the triangle edge. Initially, the boundary points are chosen to be the endpoints of the edge, so the initial boundary rays are vertex rays. At each division, a ray is sent through the midpoint of the edge segment joining the two current boun-

dary points. The midpoint ray replaces one of the two boundary rays in the binary search, depending on which object it hits. If the midpoint ray hits a third object, the process is repeated recursively for the segments on both sides of the midpoint. Bisection continues until the angle between two rays is smaller than the *maximum bound angle*, ψ , at which point the desired boundary rays have been found. Figure 41 illustrates this process.

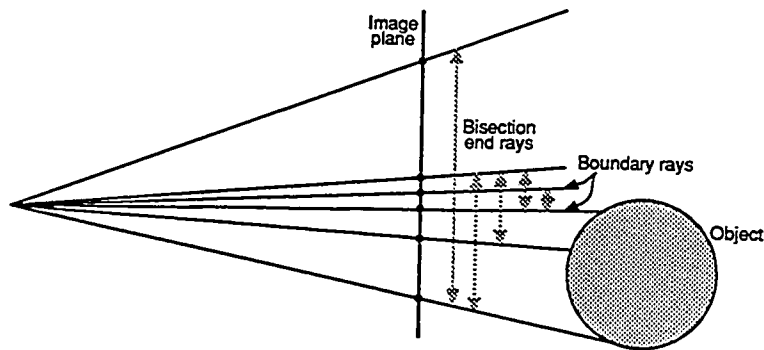


Figure 41: Recursive binary search for boundary rays

Boundaries can occur within objects as well as at their edges. Antialiasing must be performed at sharp edges of certain objects, such as cubes and cylinders. Because of this, objects with edges are divided into *zones*. For example, a cube is divided into six zones, representing the six faces. Two rays intersecting an object in different zones are considered to be on opposite sides of a

boundary. Thus, subdivision of triangles occurs over these boundaries, resulting in antialiasing.

Once all pairs of boundary rays are found, the triangle is subdivided. The subdivision method used in the algorithm joins corresponding boundary points on two edges of the triangle, as illustrated in Figure 42. The result is a triangle and a quadrilateral, which is split by a diagonal into two more triangles. The added *boundary edges* may themselves contain boundary points (represented by gray dots in the figure); the algorithm performs another binary search to detect these points. The subdivision process continues until no new boundary points are discovered. When all three edges of a triangle have no boundary points, the vertex rays are known to be consistent, and the homogeneous triangle is processed.

When triangles are subdivided, the areas of the resulting pieces are calculated and used to weight any subsequent shading computation. The weighted shading results in area-sampling antialiasing. The almost negligible area between boundary edges is added to one of the three triangles to avoid cumulative error.

The preceding discussion ignores the fact that rays may reflect from or be transmitted through objects. Two rays may intersect the same object initially but hit different objects after being reflected or refracted. Therefore, entire *ray*

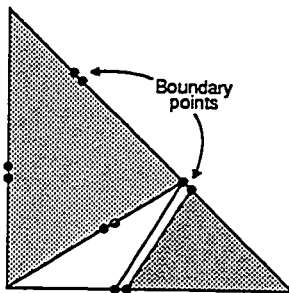


Figure 42: Subdividing a triangle by joining boundary points

trees are compared. Subdivision stops only when the three vertex rays hit the same objects throughout their trees. When this occurs, each object encountered along the tree is shaded. Contributions from objects encountered by reflection or refraction are weighted by the RGB reflectivity or transparency, as well as the original triangle area. To allow this weighting, contributions must be maintained as RGB triples, rather than single values. This is conceptually similar to Whitted's adaptive subdivision, but with a different geometric basis (triangles instead of squares) and a different comparison criterion (object geometry instead of intensity variation).

Some of the code used to implement the triangle tracing algorithm is given in Appendix C.

Comparative Analysis

We next analyze the accuracy and performance of the triangle tracing algorithm in relation to the other antialiasing algorithms described previously. We restrict the comparisons to those algorithms that are general-purpose enough to handle all forms of input objects. For this reason, the beam tracing and cone tracing methods are not considered; the former requires a polygonal model, while the latter works only with simple primitive objects such as spheres and planes. The remaining algorithms are adaptively supersampled ray tracing, distributed ray tracing, and triangle tracing. Kajiya's adaptively supersampled distributed ray tracing[49] is another algorithm to consider, but insufficient data has been collected for that technique.

Algorithm Accuracy

The triangle tracing algorithm produces well-antialiased images. The image on the left of Figure 43 was ray traced naively with one sample per pixel to show the worst case of aliasing. The one on the right was rendered with triangle tracing. The details of both images show aliasing and antialiasing around the edges of objects.

Figure 44 is a comparison of adaptive supersampling and triangle tracing. The adaptive supersampling was done to a maximum of three levels of

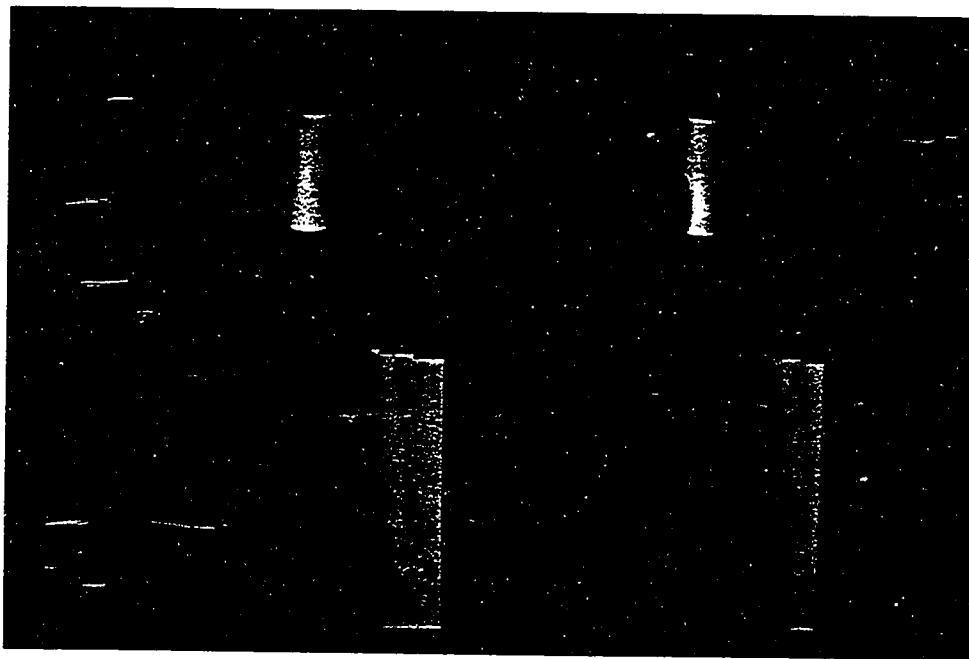


Figure 43: Single-sample ray tracing vs. triangle tracing, and details

subdivision.

The accuracy of the triangle tracing antialiasing is very high, primarily because the pixel area coverage approximation is almost exact in most cases. There are only two instances in which the accuracy is diminished. Once is for objects with very small (sub-pixel) features with high curvature. Since triangle tracing approximates intrapixel object boundaries with straight lines (the triangle edges), highly curved surfaces within a pixel could be misrepresented, as

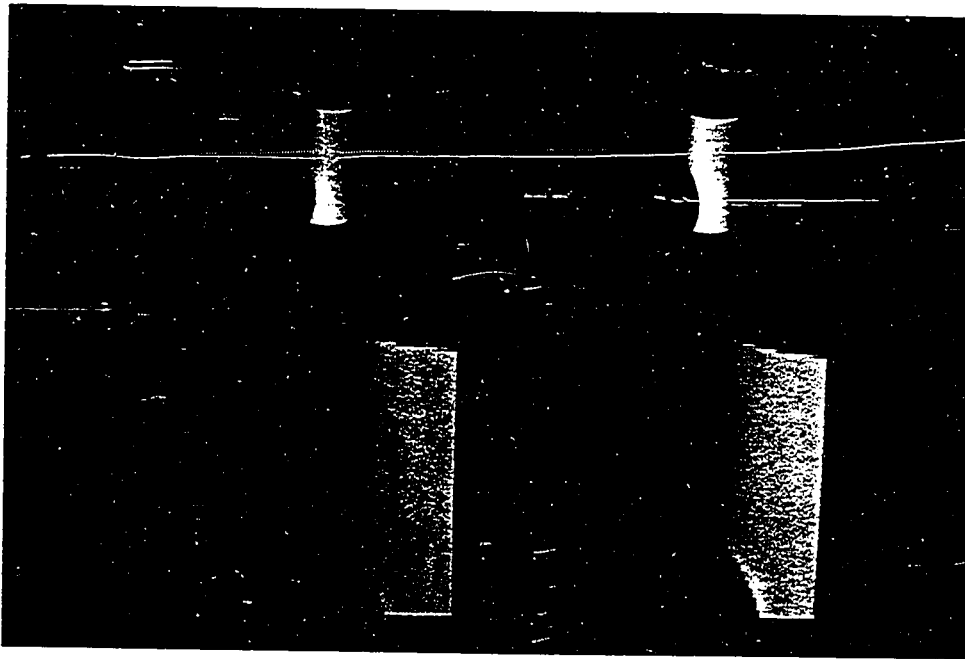


Figure 44: Adaptive supersampling vs. triangle tracing, and details

shown in Figure 45. These cases occur very rarely, as most objects and object features are larger than the size of a pixel. More likely than not, the error caused by such a discrepancy would be unnoticeable.

The other possible problem occurs with sub-pixel objects that lie between vertex rays and therefore go undetected. Again, such objects make very small contributions to the image, but there are cases where it would be noticeable, especially in high-frequency regular clusters of such objects. It makes sense to

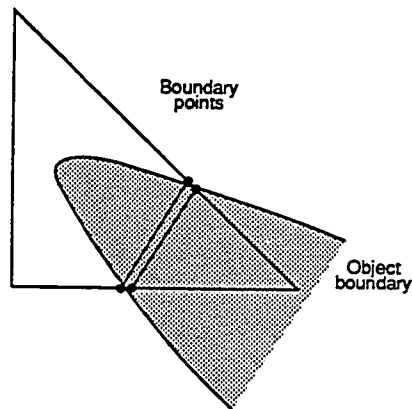


Figure 45: Linear boundary approximation leading to sub-pixel error

select a *minimum feature size* for the algorithm. Any object smaller than this size could, under the right conditions, remain undetected. If a minimum feature size smaller than half the pixel size is needed, the triangle tracing algorithm can be modified to compensate. For example, a ray could be sent through each pixel midpoint, thereby dividing the pixel into four triangles and dividing the minimum feature size in half. Another possible technique would be to increase the extent of very small objects to the apparent size of a pixel. Any ray through the pixel would intersect the object's extent, indicating that the object should not be ignored.

Perhaps the most important advantage of triangle tracing over the other two methods is the way it handles high-frequency information not caused by object geometries. This phenomena is most often encountered as a result of mapping patterns onto surface properties of an object, such as colors or normals. A pattern consisting of features that map to small areas with respect to pixel size will result in severe aliases if not handled correctly, as shown at the top of Figure 46. The pattern mapped onto the plane in the figure consists of one-pixel-wide vertical lines on a white background.

The adaptive supersampling ray tracing method uses variations in the intensity function as a basis for sampling and filtering. That is, the RGB intensities arriving at the camera as the result of sending rays into the scene are compared to each other. This comparison guides the adaptive sampling procedure. As a result, high-frequency intensity changes from pattern mapping cause extra rays to be cast to compensate for the greater amount of information in the area. It is therefore possible for many rays to be cast for a single pixel, even though the pixel is covered by only one object. This oversampling is necessary for antialiasing the pattern. The image at the lower left of Figure 46 is an adaptively-supersampled rendering of the image at the top; an average of about 14 rays per pixel were traced to antialias the image. Distributed ray tracing method, whether adaptively sampled or not, also requires several rays to be cast per pixel to antialias pattern-mapped surfaces.

The triangle tracing approach, on the other hand, does not have this problem: no extra rays need to be sent to antialias a pattern. Unlike the other two methods, the algorithm can distinguish between intensity variations caused by geometry and those caused by surface pattern mapping. Because the ray trees obtained during triangle tracing contain geometric information instead of intensities, the triangle subdivision ends when the object intersections are consistent, whether or not the surface of the object is pattern mapped. Pattern antialiasing can be achieved with any of several techniques that map areas on the surface of an object into areas in a pattern space, such as the *summed-area tables* method[24]. The result of triangle tracing a pattern is shown at the lower right of Figure 46. The same number of rays were needed to trace this image as were needed to trace the aliased image at the top of the figure.

Performance

A strict analysis of ray tracing performance is very difficult. In fact, a consistent performance test for implementations of conventional ray tracing programs has not been developed, despite considerable effort[32]. This is due partly to incompatibility among various implementations and models and partly to the difficulty of classifying data. A best/worse/average case analysis is almost meaningless: “best” cases are scenes devoid of objects, “worst” cases are impossible to construct and of no interest, and “average” cases are difficult to

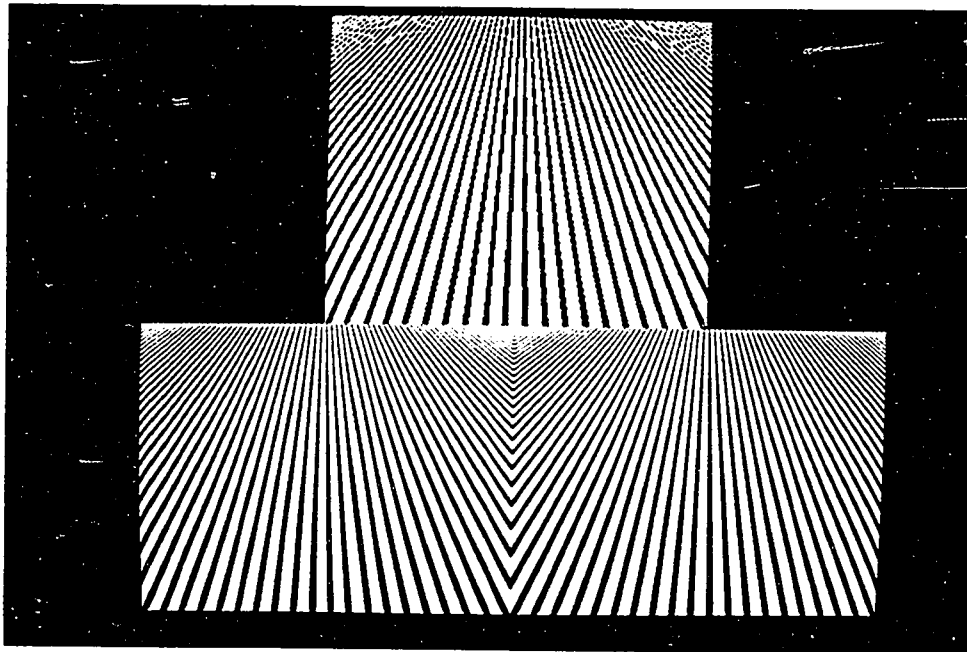


Figure 46: Pattern-mapped object: one ray per pixel (top), adaptive supersampling (bottom left), triangle tracing (bottom right)

establish as being average.

However, because we are interested in comparing triangle tracing with other antialiasing algorithms, we will make a simplifying assumption. The assumption is that the time needed to compute object intersections for a single ray tree is the same for all algorithms. (We also assume that all algorithms compute identical trees for identical rays.) This means that if two algorithms

cast the same number of primary rays, they will have approximately the same running time. We can make this assumption because we are not really comparing ray-object intersection routines, and therefore their execution times are unimportant.

Given this assumption, the running time of an algorithm is then a function of complexity at each pixel. For homogeneous pixels — i.e., those pixels in which no object is visible or which are occupied entirely by one homogeneous object — all three algorithms have approximately the same performance. All samples in such a pixel are similar, so there is no need for supersampling or subdivision. A large proportion of pixels in many images are homogeneous, as indicated in Figure 47. For images with large homogeneous pixel ratios, all three algorithms run in approximately the same time.

Pattern mapping is often used to enhance image complexity by varying surface properties of objects. This is an important technique that often covers a significant portion of an image's pixels. For pixels covered by a pattern-mapped object, the triangle tracing algorithm is considerably faster than the other two. The adaptive sampling algorithms detect the pattern as intensity function variations, and require more samples for antialiasing. This results in many extra rays being cast, especially if the pattern contains high-frequency information with respect to the pixel size. Triangle tracing does not need to subdivide such

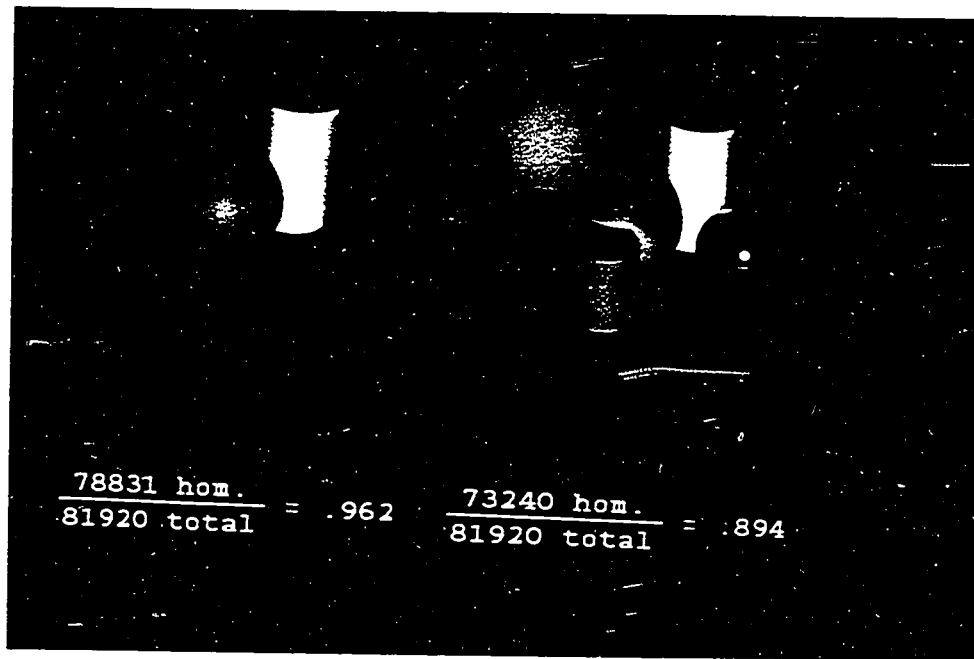


Figure 47: Homogeneous pixel ratio for two sample images

a case, since the object geometry is consistent over all pixel corner samples; no extra rays need be sent. Thus, triangle tracing provides a significant performance advantage in such cases.

Pixels in which more than one object appears, either directly or through reflection or refraction, require more work for all three algorithms. In Whitted's adaptive subdivision algorithm, such a pixel may be subdivided m times for a maximum of 4^m sub-pixel squares and a maximum of $(2^m + 1)^2$ rays.

Implementations must limit the recursion level m so that the algorithm terminates. A typical limit is $m = 3$, so that at most 81 rays may be cast for a single pixel. (The simple case of a pixel bisected by a diagonal line requires 39 rays to be traced, as shown in Figure 48.) This level of subdivision is usually enough for reasonable antialiasing, except in cases where the regularity of the sampling produces aliasing artifacts. Increasing the number of rays sent in those instances decreases the likelihood of aliasing problems but can never eliminate it entirely.

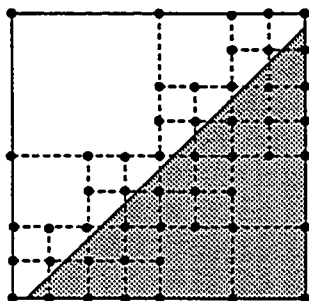


Figure 48: Adaptive supersampling for a simple two-object pixel

Adaptive sampling for distributed ray tracing leads to similar performance results. Experimental results with sample images by Lee, et al[55], show that a small number of rays, such as 8, is an adequate sample for low-frequency areas, while pixels with high-frequency information could require over 90 rays for

reasonable antialiasing. In their examples, the fraction of pixels requiring the maximum number of rays to be cast ranges from 10 to 43 per cent. The average number of rays per pixel is approximately twenty, while those pixels requiring more than the minimum rays average about 50 rays apiece. As with adaptive regular subdivision, the number of rays needed is very high in areas with pattern-mapped objects.

The number of rays traced by the triangle tracing algorithm is related to the number of subdivisions for each of the pixel triangles. If the pixel is homogeneous, no subdivisions are required; only the three vertex rays are sent, for a total of 4 per pixel. The implementation of the algorithm saves ray trees from one row of pixels to the next and from one pixel to the next within the row, so that most homogeneous pixels (those not in the top row or first column) require only one new ray to be traced.

Finding boundary rays is done by angle bisection until the boundary ray angle is smaller than the maximum bound angle, ψ . The largest angle that may be bisected is one subtended by the diagonal of a pixel with respect to the camera point. This angle is ϕ , the *minimum feature angle*; an object or feature with apparent diameter smaller than this angle is not guaranteed to have an effect on the image. The number of rays needed to find any boundary ray pair is bounded by $\lg(\psi/\phi)$. The *boundary angle ratio*, ψ/ϕ , determines how accurately

the bounding rays represent object edges. Larger ratios leave larger gaps between triangle subdivision edges, and therefore result in less accurate approximations. Figure 49 shows a sample image with varying angle ratios. A ratio of 0.1 appears to be adequate for this simple image, resulting in an upper bound of 4 on the binary search. A more complex image may require a slightly smaller ratio. Of course, each subdivision produces smaller triangle edges, so that the bound decreases as the recursive subdivision proceeds.

Triangle Subdivision

The importance of an efficient triangle partitioning method is obvious. Each subdivision results in the creation of two new edges, sometimes requiring several boundary rays to be located. It is therefore desirable to minimize the number of sub-triangles created during partitioning, although the ultimate goal is to minimize the number of boundary rays. This latter goal does not necessarily follow from the former, as will be shown by example.

The best possible subdivision method would use information about the geometry of the objects visible within a triangle. This information is unfortunately unavailable; all we have is a collection of boundary points. Consider the triangle in Figure 50. There are three objects, A, B, and C, visible in the triangle, resulting in three pairs of boundary points, p_1 , p_2 , and p_3 . Each point is

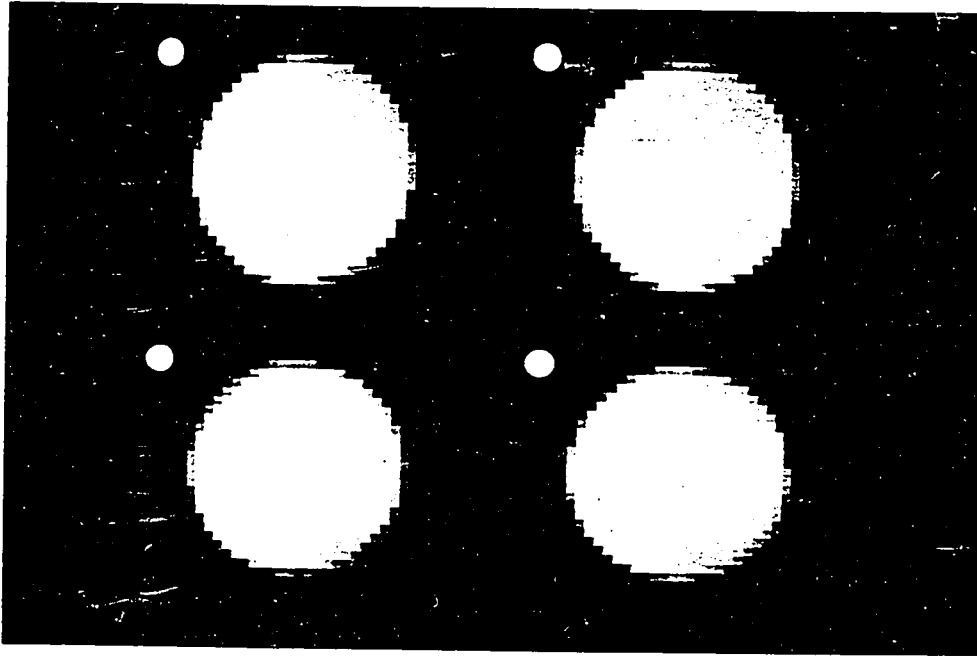


Figure 49: Simple image rendered with varying boundary angle ratios

labeled with the object which its ray intersects. There is no explicit correlation between the pairs of points; there is no way for an algorithm to know beforehand that it is better to partition the triangle by an edge joining p_1 and p_3 , rather than by one joining p_1 and p_2 . In fact, knowing that an edge of an object passes between p_1 and p_3 is beyond the scope of any algorithm. We must therefore develop heuristics to guide the subdivision process in ambiguous cases.

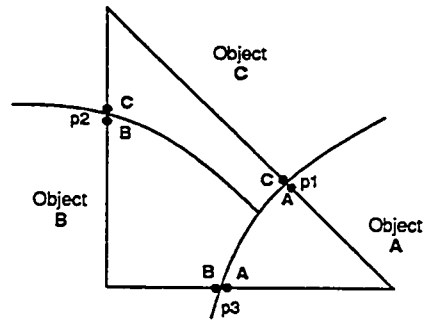


Figure 50: A simple case of partitioning ambiguity

What are the desired qualities on which to base the subdivision heuristics? We already know that the number of triangles or boundary rays is to be minimized. We would also like to avoid creating *trivial* triangles, those with a minuscule proportional area (e.g., less than 0.0000001) with respect to the original triangle. Trivial triangles are ignored by the triangle tracing algorithm, so that no time is spent on triangles with minimal contribution. The subsequent loss of accuracy is not very significant, but should be minimized.

Another heuristic measure is rapid convergence. Partition edges should be chosen to have as few bounds as possible, so that fewer subsequent subdivisions are needed. A *solid* partition edge, one with no boundary points, introduces no extra subdivisions and is ideal. This savings must be weighed against the time

needed to find such an edge; if, for example, all pairs of boundary points are compared to see if a solid edge can be used, the time savings are compromised. A simple and feasible method is to look for solid edges at the corners of a triangle (Figure 51). The underlying geometry makes such edges a good possibility when compared with the chance of finding a solid edge among internal boundaries. A solid corner partition edge can be detected by comparing the ray trees at the boundary points: if both trees hit the same objects, there is no need to compute boundary points along the joining edge, and the edge is solid.

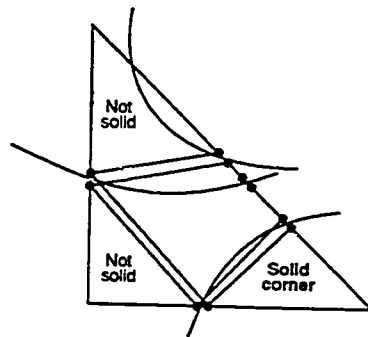


Figure 51: Solid partition edges at triangle corners

These heuristics lead us to the following method of subdividing a triangle. It is assumed that the triangle has boundary points on at least two sides, or there is no need to partition it.

- If a solid corner partition edge exists, use it to cut off a homogeneous triangular corner of the triangle, leaving a quadrilateral.
- If no such corner exists, partition the triangle as follows:
 - Find the two sides with the largest number of boundary pairs.
 - Find the numerically centermost pair on each of those two sides.
 - Join the corresponding points of the two pairs.
 - The result is another triangle, and a quadrilateral.
- To partition a quadrilateral, test whether either diagonal is solid. If so, partition by that diagonal. Otherwise, choose the diagonal that leaves the two sides with the most boundary point pairs in one triangle.

The image in Figure 52 shows the graphical output of a subdivision algorithm testing program. The image on the left is the state of the program as it begins execution. The large right triangle represents a pixel triangle, and the thick white edges represent the boundaries of objects visible in that pixel. The magenta dots are boundary points that have been found by bisection. The task of the subdivision program is to partition the triangle into smaller triangles, each of which is fully contained in one of the object regions. The image on the right of Figure 52 shows the finished partitioning, with randomly-chosen colors for the sub-triangles.

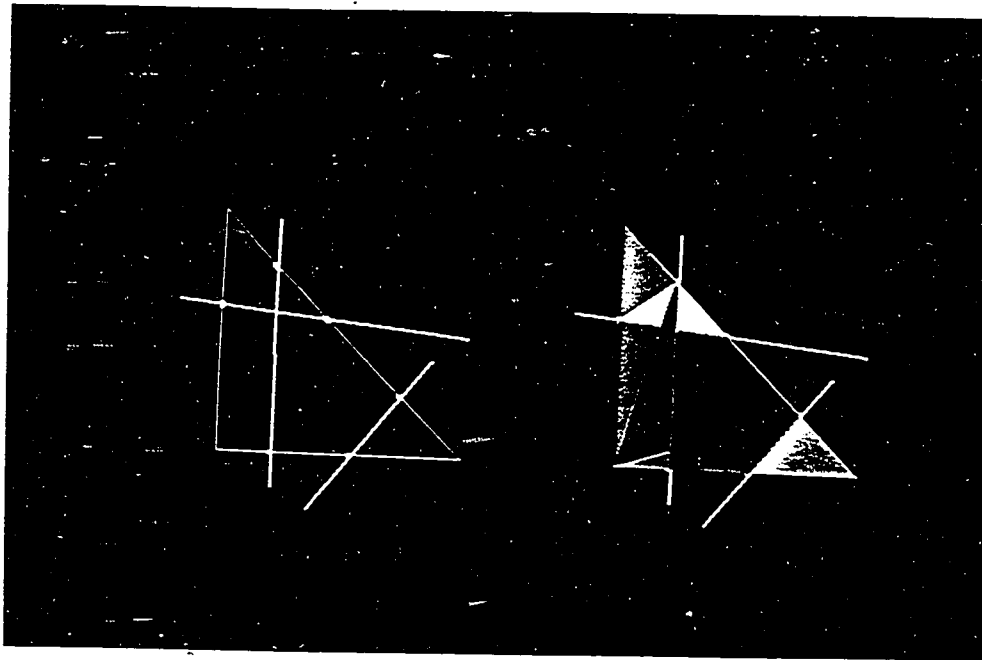


Figure 52: Graphical display of partitioning test program

The tables in Figure 53 show the results of running the subdivision program on a variety of test cases. Five different partitioning algorithms are tested:

- Subdivide by cutting off the first corner of a triangle (A).
- Subdivide by cutting off a randomly-chosen corner of a triangle (B).
- Subdivide by cutting off the triangle corner with the largest area (C).

- Subdivide by joining the centermost bounds of the two triangle sides with the most boundary pairs (D).
- Subdivide by the algorithm described previously (E).

The table shows for each case the number of input line segments and regions formed by them. The resulting number of triangles created, the number of trivial triangles (a good measure of bad partitioning), and the total number of boundary points computed, including those rejected during the binary search, are also given. The results of “optimal” partitioning are included for comparison (method O). These numbers were produced by looking at the input regions and determining by eye the minimum number of triangles needed to cover them.

The numbers in the table support some very important observations. First, it is clear that the first-corner, random-corner, and largest-corner methods are too inefficient to be practical. Second, a smaller number of sub-triangles does not always imply fewer boundary tests (see Case 8 in the table). Third, varying the maximum bound distance (corresponding to the maximum bound angle ψ) has an appreciable effect on the subdivision performance. Last, and most important, the subdivision algorithm used in triangle tracing (method E) seems to be about linear in the number of object regions within the triangle, and is therefore acceptable for our purposes.

	Method						Method					
	A	B	C	D	E	O	A	B	C	D	E	O
Boundary angle ratio = 0.001												
	Case 1: 1 line, 2 regions						Case 2: 2 lines, 3 regions					
Non-trivial Triangles	3	3	3	3	3	3	5	5	5	7	5	5
Trivial Triangles	0	0	0	0	0	-	0	0	0	2	0	-
Total Bounds	20	20	20	20	20	-	39	39	39	69	39	-
	Case 3: 2 lines, 4 regions						Case 4: 3 lines, 5 regions					
Non-trivial Triangles	93	13	24	9	9	7	31	26	30	16	13	9
Trivial Triangles	315	12	27	0	0	-	20	17	41	15	0	-
Total Bounds	*	179	358	69	69	-	334	267	459	175	96	-
	Case 5: 3 lines, 6 regions						Case 6: 5 lines, 13 regions					
Non-trivial Triangles	77	39	45	25	27	11	209	240	175	73	90	25
Trivial Triangles	381	36	78	12	10	-	473	261	320	88	85	-
Total Bounds	*	410	886	243	235	-	*	2908	3207	819	882	-
	Case 7: 8 lines, 14 regions						Case 8: 8 lines, 17 regions					
Non-trivial Triangles	228	154	136	52	66	27	206	187	212	85	90	31
Trivial Triangles	460	105	223	61	49	-	502	132	257	58	41	-
Total Bounds	*	1360	2617	702	634	-	*	1451	2913	762	657	-
Boundary angle ratio = 0.001												
	Case 1: 1 line, 2 regions						Case 2: 2 lines, 3 regions					
Non-trivial Triangles	3	3	3	3	3	3	5	5	5	7	5	5
Trivial Triangles	0	0	0	0	0	-	0	0	0	2	0	-
Total Bounds	14	14	14	14	14	-	27	27	27	48	27	-
	Case 3: 2 lines, 4 regions						Case 4: 3 lines, 5 regions					
Non-trivial Triangles	27	11	19	9	9	7	33	39	23	18	13	9
Trivial Triangles	383	14	24	0	0	-	14	17	10	13	0	-
Total Bounds	*	107	211	48	48	-	189	166	134	109	66	-
	Case 5: 3 lines, 6 regions						Case 6: 5 lines, 13 regions					
Non-trivial Triangles	69	44	33	23	25	11	163	293	154	72	81	25
Trivial Triangles	389	33	30	16	14	-	122	256	127	65	58	-
Total Bounds	*	235	286	157	152	-	636	1587	769	396	386	-
	Case 7: 8 lines, 14 regions						Case 8: 8 lines, 17 regions					
Non-trivial Triangles	208	100	108	66	66	27	180	178	162	80	90	31
Trivial Triangles	119	49	65	51	41	-	221	107	87	47	41	-
Total Bounds	1052	441	526	383	352	-	1435	723	772	399	376	-
*Recursion was too deep and the algorithm terminated prematurely.												

Key	Method Description	Key	Method Description
A	Partition first corner of triangle	D	Partition by centermost bounds
B	Partition random corner of triangle	E	Partition solid corner, else centermost bounds
C	Partition corner of triangle with largest area	O	Theoretically optimal triangle partitioning

Figure 53: Comparison of triangle partitioning methods

The number of rays needed to antialias a pixel with one boundary, even for a relatively small boundary angle ratio of 0.01 is only 14. This is less than the number required for adaptive supersampling (more than 30, as shown above) or distributed ray tracing (at least 16). Slightly more complex pixels (with three or four objects) require approximately the same number of rays for all three methods. As a pixel increases in complexity, the number of rays needed for triangle tracing increases, surpassing the number required for the other two methods. This situation suggests that some heuristic should be built into triangle tracing to avoid excess computation for extremely complex pixels. However, since such pixels are rare, this optimization may not be necessary.

Summary

We have presented triangle tracing, a new algorithm for producing area-sampled antialiased ray traced images. Unlike beam tracing, the algorithm does not require a polygonal model. Unlike cone tracing, it uses the standard ray-object intersection procedures used in conventional ray tracing. Triangle tracing requires no extra rays to be traced per pixel to antialias pattern-mapped objects, which can account for large portions of complex images. Adaptive supersampling and distributed ray tracing approaches require many rays to be traced per pixel for such objects. For pixels with low complexity (i.e., containing

fewer than about four objects), the triangle tracing algorithm is comparable in speed to either of these point-sampling methods, and produces results of equal or greater accuracy. By tuning the maximum bound angle ratio, the algorithm can be adjusted for speed, trading off accuracy.

In the next chapter we show how an extension of the triangle tracing algorithm can be used to incorporate caustics, the interaction of light between objects, into ray traced images.

6. Caustics

In Chapter 5, we described the problem of aliasing in ray tracing and how the triangle tracing algorithm could be used to solve it. In this chapter, we focus our attention on another drawback of the conventional ray tracing algorithm. The problem involves *caustics*, the reflection or refraction of light from one object to another. Conventional ray tracing does not include these effects because of the ways in which rays are traced and lighting is computed. We will present an extension to the triangle tracing algorithm that simulates caustics, increasing the realism in the resulting images.

The Problem

Conventional ray tracing is illustrated in Figure 34 in Chapter 4. Rays are traced from the camera through image pixels, and are tested for intersections with objects in the scene. When an intersection, such as the one with the sphere

in the figure, is found, the lighting at the surface point is computed and the resulting intensity is added to the pixel. Lighting is computed by tracing rays from the surface point to each of the light sources, checking for shadows, and using the resulting intensity as input to the lighting model. If the surface is reflective or transparent enough, secondary rays may be traced; their contributions are added to the pixel, representing other objects that are visible at that pixel.

Note that the lighting contribution to the surface point comes directly from the light sources, not from other objects. This is more apparent in the scene in Figure 54. The mirror object at the left of the scene reflects light from the point source onto the diffuse sphere at the right. However, when the lighting at each visible surface point on the sphere is computed, this reflected light is not detected; only the light coming directly from the source is used in the computation. When the lighting at points on the mirror is computed, a ray is propagated in the direction of reflection with respect to the view ray. This reflected ray determines what objects are visible in the mirror, but does nothing to contribute caustics.

How can caustics be incorporated into the lighting model? It is clear that doing so requires rays to be traced from the light sources, rather than from only the camera. However, it does not make sense to trace rays from only the light

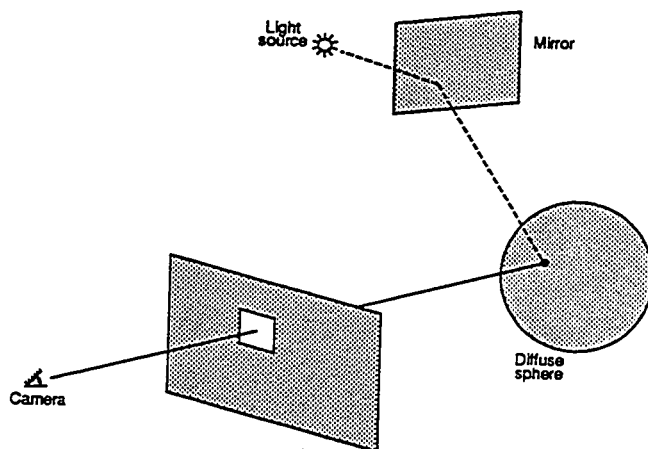


Figure 54: A caustic interaction undetected by conventional ray tracing

sources; this would result in problems analogous to the caustics problem, but from the camera. That is, reflections and refractions visible in object surfaces can be detected only when tracing rays from the camera. It is therefore clear that a solution to the caustics problem requires information to propagate in both directions.

Related Work

Arvo has proposed a method that traces rays from both the camera and light sources[4]. In the preprocessing pass, rays are traced from all light sources

toward objects. Each ray from a source is assigned some amount of energy. When a ray intersects an object, directly or through reflection or refraction, some of the energy from the ray is stored at the intersection point. This information is used in the second pass, which is essentially the conventional ray tracing algorithm. When a surface point is shaded in this second pass, the energy stored at the intersection point is used in the lighting calculations. The energy, representing the specularly reflected intensity, is added to the diffuse component, which is computed normally.

Each object in the scene has an illumination map that is used to store energy quanta for that object. The map is a 1 by 1 square that is divided into a grid of points. Each point on the grid corresponds to a point on the surface of the object, according to some parameterization function for the surface. That is, given uv -coordinates for a point on the grid, there is a corresponding xyz point on the surface of the object. When a light ray intersects the object, the intersection point is transformed into a uv point, and energy is stored at the four grid points surrounding that point.

The intensity at a surface point is computed from the energy stored at the corresponding illumination map point. The energy must be divided by the area of the surface represented by the map point. The partial derivatives of the surface parameterization function are used to approximate this area.

Arvo warns that the density of the light rays must be much greater than the density of the grid points in illumination maps, or the statistical results will be compromised. This assumes knowledge on the part of the renderer about the mapping functions, the scene geometry, and the grid subdivision. Since these would be difficult to incorporate into the program, it is up to the animator to choose the correct density proportions.

Another drawback of Arvo's scheme is its potentially unlimited memory needs. Each object in the scene requires an illumination map, whether or not that object is ever visible to the camera. Memory use is then proportional to the number of objects in the scene, with a coefficient based on the density of the illumination maps. For scenes of high complexity, the storage could become excessive.

Kajiya has developed a variation on ray tracing based on his generalization of rendering, the "rendering equation."^[49] His solution to the equation involves several Monte Carlo algorithms. The resulting images contain caustic effects, and also reflection of light between diffuse surfaces. One advantage to his approach is that very little time is spent computing lighting effects that contribute little to an image.

Kajiya's rendering method is similar to distributed ray tracing, which was discussed in the previous chapter. For each pixel, several rays are sent into the

scene. However, unlike conventional ray tracing, each primary ray does not result in an entire ray tree of reflections and refractions. Instead, one *path*, a tree with one branch at each node, is chosen for each ray. The direction of each subsequent ray in a path is chosen by probabilistic methods, using the surface properties at an intersection point as a basis.

Like distributed ray tracing, this stochastic technique needs to sample the scene more heavily in high-frequency areas. Therefore, the adaptive sampling problems of distributed ray tracing are also found in Kajiya's approach. This means that the algorithm can be foiled by very high concentrations of image information in small areas, where high sampling levels are necessary. This includes not only clusters of objects, which can similarly defeat triangle tracing, but mapped patterns as well.

Overview of the Algorithm

Our approach is a non-probabilistic one. It is similar to Arvo's, but reverses the order of the passes. In the preprocessing pass, the algorithm determines which pieces of objects are visible to the camera. The second pass traces rays from light sources to the objects. When light falls in a visible area, the appropriate amount of intensity is added to the pixel from which that area is visible.

The triangle tracing algorithm described in the previous chapter is used in both passes. In the first pass, it is used to discover the triangular visible regions on object surfaces. In the second pass, it determines the amount of light to be added to each object, and therefore to each pixel. We examine each of these passes in more detail in the following sections.

Visible Regions

The purpose of the first pass is to determine what parts of objects are visible to the camera. A modified version of the triangle tracing algorithm is used to obtain this information. When a homogeneous triangle is encountered by the algorithm, it is not shaded but is stored as a *visible region* associated with the object intersected by the triangle's rays. Thus, each object keeps track of the regions on its surface that can be seen by the camera. Associated with each visible region is the pixel through which the region's triangle was traced, and the *coverage*, the amount of the pixel's area covered by the triangle. For example, if all three rays of a half-pixel triangle intersect the same object, a visible region with coverage 0.5 is added to that object.

uv Coordinate System Mapping

Storage of regions on three-dimensional surfaces is difficult, especially when those regions must later be searched for point containment. Therefore, the visible region storage for objects is done in parametric space. That is, each vertex of a triangular region is converted via an inverse parametric mapping for the object to *uv*-coordinates, and the region is stored in terms of those coordinates. This allows regions to be stored in a two-dimensional area, rather than in three dimensions.

Therefore, a parametric mapping method must be defined for each class of object in a scene. For example, a *plane* object maps linearly from its four corners to the four corners of the one-by-one *uv*-coordinate square (Figure 55). However, other objects require extra work. The surface of a sphere, for example, cannot be mapped to a square without introducing discontinuities, usually at the poles. (This problem has plagued mapmakers for a long time.)

The problem of discontinuity can be overcome for certain object classes by dividing objects into *zones*, similar to the ones used for detecting internal edges in the previous chapter. For example, a sphere can be divided into two zones, representing the upper and lower hemispheres. Each of the zones maps by projection to a circular region in one half of the *uv*-map, as shown in Figure 56. To avoid discontinuities between zones, two rays intersecting different zones of the

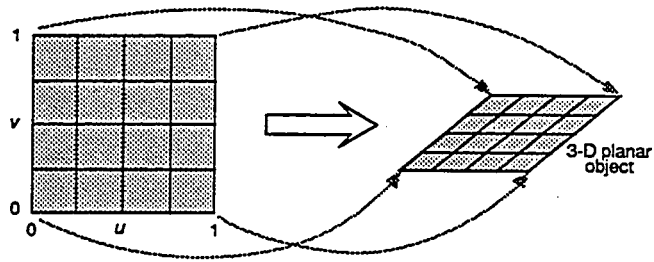


Figure 55: Parametric mapping of a square planar object

same object are treated as if they intersected different objects. Therefore, a triangle in the first pass of the caustics algorithm is considered homogeneous only if its three rays all intersect the same zone of the same object.

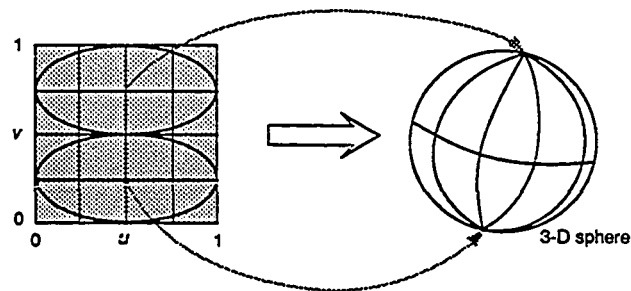


Figure 56: Parametric mapping of a sphere, using two zones

Region Overlap Processing

The preceding description glosses over the fact that the rays cast in triangle tracing result in ray trees. Each ray may reflect from or refract through an object. This fact has two effects on the algorithm. The first is that regions visible through reflection or refraction must also be added to objects, along with a measure of their contribution to the pixel, based on the reflectivity or transmittance of the intersected surface. The second consequence is that two or more regions on the same object may overlap. This can be seen from the example in Figure 57, in which two regions on the sphere can be seen to overlap. One of the regions is the result of direct visibility from the camera, while the other is created through reflection from the mirrored surface on the left.

The consequences of region overlap are serious. In the second pass, when rays are traced from light sources, the region in which each ray lands must be determined. If, because of overlap, a ray can land in several regions, the comparison of rays becomes relatively difficult and time-consuming. (This is especially bad when the vast number of such comparisons is considered.) Instead, a preprocessing step is performed in which region overlaps are removed for each object, producing a set of nonoverlapping polygons that represent visible regions of that object.

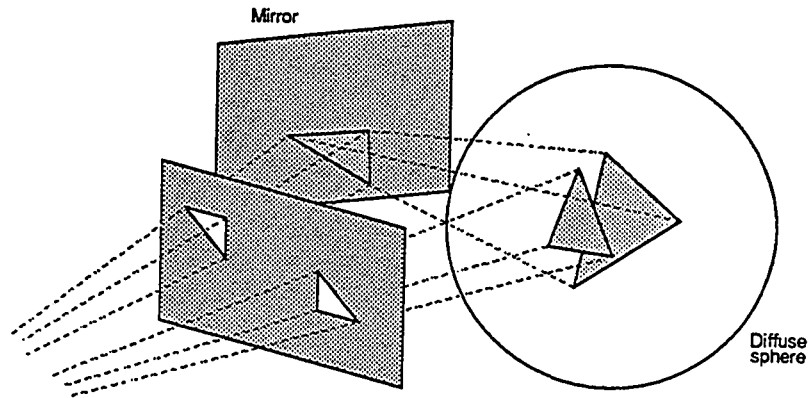


Figure 57: Region overlap caused by reflection

The overlap removal process consists of two steps. First, all intersections of edges of the triangular regions are found by a sweep-line algorithm. Second, the regions bordered by the intersected edges are resolved by another line sweep.

The first implementation of the overlap algorithm was implemented in floating-point coordinates. However, this approach quickly developed problems due to the fixed precision of floating-point computations on a computer. Because triangle tracing subdivision results in very small regions, the lengths of edges when mapped into parametric space are small enough to approach the floating-point granularity of the computer. This leads to severe problems in the intersection algorithm when intersection points cause edges to be perturbed

slightly, altering the geometry of regions.

Greene and Yao have developed an integer-based solution to the edge intersection problem[38]. Their algorithm ensures that a point added to an edge as a result of an intersection does not cause the edge to move, creating inconsistent geometries. Therefore, their solution is ideal for the region overlap problem. Their method is a variation of Bentley and Ottmann's intersection algorithm[7]; the modifications involve how edges are "redrawn" to new points when intersections are found. The resulting algorithm runs in time $O((n + i)\log n + i)$, where n is the number of original edges, and i is the number of edge intersections.

Once all edge intersections have been determined, another sweep of the mapping square is made to determine region overlaps. At each step along the sweep, edges intersecting the sweep line are examined, in order. Regions bounded by the edges are maintained in a list. Where two or more regions are found to overlap, a new region is created and marked as being contained in the original regions. This information is used in the second pass of the caustics algorithm to add intensities to regions; any light landing in the overlap region is added to all regions in which that region is contained.

Tracing Rays from Light Sources

The second pass of the caustics algorithm uses triangle tracing to cast rays from light sources towards objects. The triangles are defined by placing a cube around each point light source: each face of the cube is divided into a grid of squares, which are divided in half to form triangles. (The grid subdivision is similar to methods used for shadow[42] and radiosity[16] calculations.) Triangular divisions on one face of a cube are shown in Figure 58.

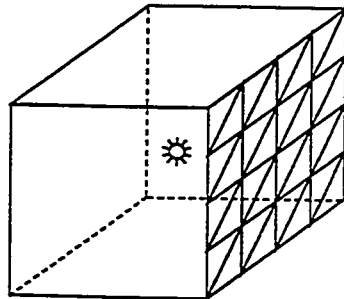


Figure 58: Dividing a light source cube face into triangles

The triangle tracing algorithm proceeds as before, with some important differences. The subdivision criterion is not based only on the objects (and zones) intersected by the rays, but also on the visible regions (if any) in which the rays land. Therefore, a homogeneous triangle is one for which all three

vertex rays intersect the same object within the same visible region.

When a homogeneous triangle is discovered, the visible region at each level in the vertex ray trees for the triangle is shaded, and the appropriate amount of intensity is added to the pixel from which that region was visible. The computed intensity is multiplied by the ratio of the area of the traced triangle to the area of the visible region. The result is multiplied by the contribution of the visible region to the pixel (based on the reflectivity and transmittance of surfaces encountered along the way) and the coverage of the region.

Region Searching

Each intersection of a light ray with an object must be tested for inclusion within a visible region. Because many such inclusion tests are made, it is vital that the test be very efficient. A time- and space-efficient point location algorithm has been developed by Edahiro, Kokubo, and Asano[27]. Their algorithm works for any set of bounded regions, with no constraints on convexity. This flexibility is important, since overlap removal usually results in non-convex regions.

The point location algorithm requires a preprocessing step in which regions are partitioned into rectangular *buckets*. The number of buckets is chosen so that neither the storage nor computation costs are excessive, based on the

number of points and edges in the region graph. The authors claim performance of $O(n)$ preprocessing time, $O(n)$ memory use, and $O(1)$ search time, for a graph with n vertices and the correct number of buckets.

Results

The images produced by this algorithm include antialiasing, reflection, refraction, shadows, and caustics. Antialiasing is a result of the area sampling in the first pass, as is the case in the triangle tracing algorithm presented in the previous chapter. Visible reflections and refractions are detected in the first pass when view rays intersect reflective or transparent objects. Shadows are obtained as a result of the illumination from light sources in the second pass: rays that intersect opaque objects do not continue through them. Caustics result from the propagation of light rays by reflection and refraction in the second pass.

A sample image is shown in Figure 59. The highlights on the diffuse blue plane at the bottom are caused by light reflecting from the smooth white plane above it. The caustics are visible also in the reflection in the white plane.

It should be noted that caustics are detected even if the reflecting or transmitting object is not visible. Light ray tracing does not stop until a ray intersects no object. That is, a light ray hitting a reflective surface will

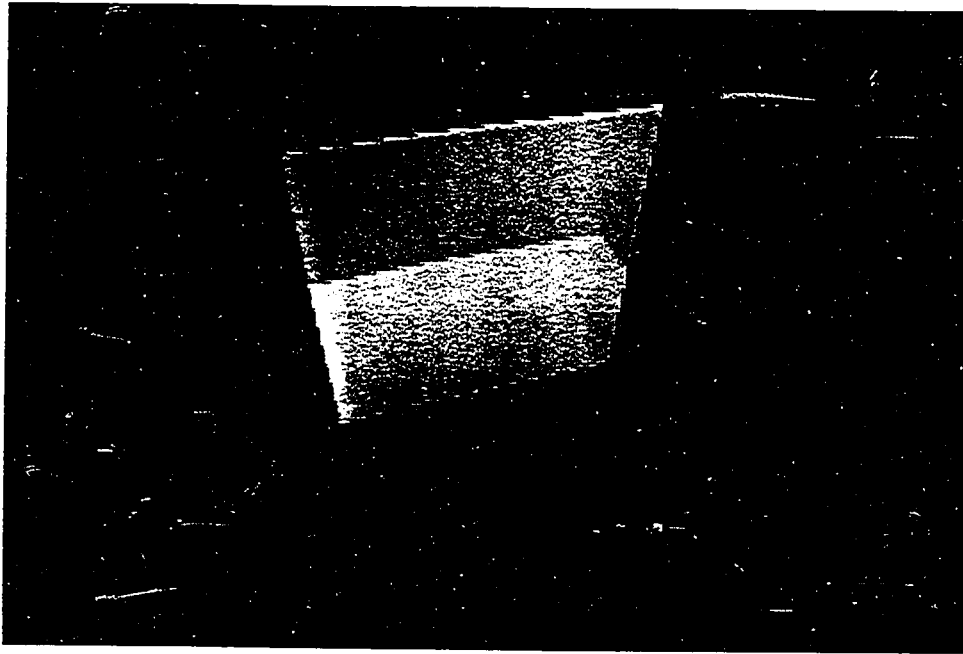


Figure 59: Caustics from one plane to another

continue, even if the ray does not land in a visible region; the ray contributes to whatever object is reached by reflection. The image in Figure 60 illustrates such a case. The trapezoidal caustic reflection on the plane in the image is caused by a reflective plane that is out of view to the left.

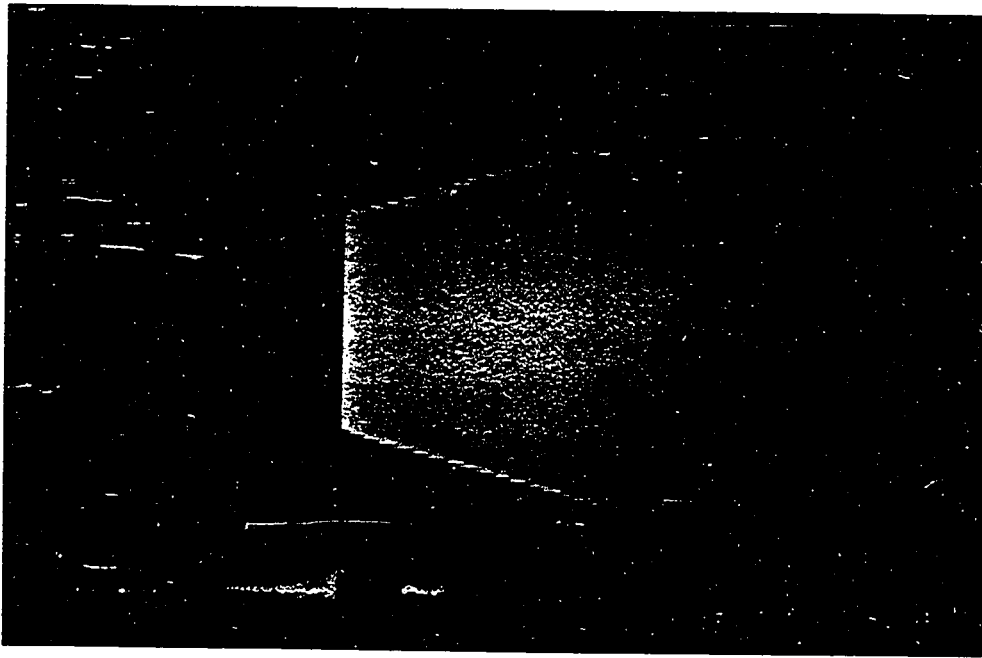


Figure 60: Caustics from an off-camera object

Complexity Considerations

The caustics algorithm is extremely computationally intensive. The first pass is computationally equivalent to the triangle tracing algorithm of Chapter 5. The second pass, performed once for each light source, is typically more expensive. The triangle subdivision by visible region boundaries often results in many triangles being cast.

The complexity of the second pass depends largely on the number of divisions used for a grid on the face of a light source cube. With few divisions, the light rays for a single triangle may miss important features, and therefore require fewer subdivisions. With many divisions, the resulting computational burden may outweigh the resulting increase in accuracy.

The storage cost depends on the number of visible regions encountered in the first pass. Discounting secondary rays, the number of regions depends on the number of pixels in the image and the subdivision level of each pixel. With reflection and refraction, this number increases, but remains proportional to the number of pixels. The storage cost is lower than that necessary for Arvo's method, because not all of the object surfaces are visible.

In triangle tracing from the camera, it is guaranteed that no important objects could "slip through the cracks" between rays, because any such objects would have to be of sub-pixel size. The same is not true in the second pass of the caustics algorithm, because what is unimportant to a light source may be very important to the camera (figure 61).

This case can be avoided by an appropriate selection of the *minimum feature angle*, ϕ . This angle, as described in the previous chapter, determines the largest object that can remain undetected by the algorithm. If ϕ is chosen to be the smallest angle (from the light source) subtended by any object, then no

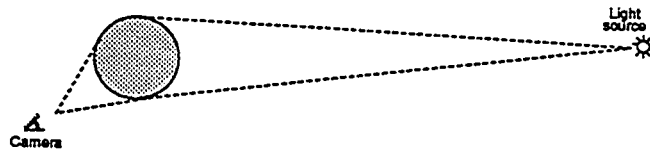


Figure 61: A large object with respect to the camera, but small with respect to a light source

object will be missed. This angle can be used to select the number of grid divisions for each face of the cube around the light source.

Reflection and refraction or primary rays may result in divergent secondary rays that subtend an angle larger than ϕ , as shown on the left in Figure 62. When this situation occurs, it may be necessary to subdivide triangles. To detect divergent rays, the *tree angle* is computed for all pairs of vertex ray trees in a homogeneous triangle. The tree angle is defined to be the maximum angle between corresponding rays in two trees. (The trees at the vertices of a homogeneous triangle are guaranteed to intersect the same objects along the way, and therefore have the same depth and structure.) A homogeneous triangle with a side that subtends an angle larger than ϕ can be divided into two triangles at the median by tracing a ray through the midpoint of that side, as depicted on the right of Figure 62.

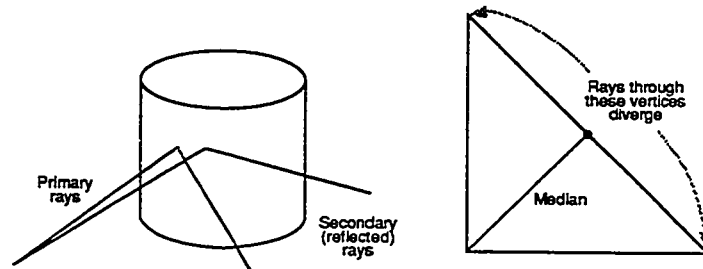


Figure 62: Diverging secondary rays; triangle subdivision by median

It is important to note that an object will be treated properly if at least one ray intersects it. This fact suggests an approach in which at least one ray is cast toward the center of every object. This method has two problems, however. First, objects encountered by reflection or refraction may still be missed. Second, the recursive subdivision may overflow a procedure stack in cases where the initial triangle crosses many region boundaries.

One feasible optimization is to avoid casting light rays that definitely will not have any effect on the final image, i.e., rays that do not intersect the cuboid extent enclosing all objects. In fact, each face of the cube surrounding the light source can be tested in this manner to see if any rays need to be traced through it. If necessary, the face can be subdivided and the test repeated.

Summary

In this chapter we have described an extension to the triangle tracing algorithm that models caustic interactions of light between objects, without sacrificing visible reflection and refraction. Because it is built on an area-sampling technique, the resulting images are antialiased. Unlike stochastic methods for computing caustics, the algorithm does not depend on variance in samples to do the antialiasing, so pattern-mapped objects can be rendered at no extra cost. Additionally, casting rays from light sources in the algorithm produces shadows by default.

A sample image containing all of these effects is shown in Figure 63. Overlapping caustics from the reflective red and blue planes are visible on the diffuse gray plane, while reflections of the gray plane are visible in the other two. Shadows are also visible. The image is magnified four times in height and width, making the antialiasing around edges is visible. The need for optimizations to the algorithm is evidenced by the time needed to compute this image. It required about 38 hours of CPU time on an Encore Multimax, using only one processor.

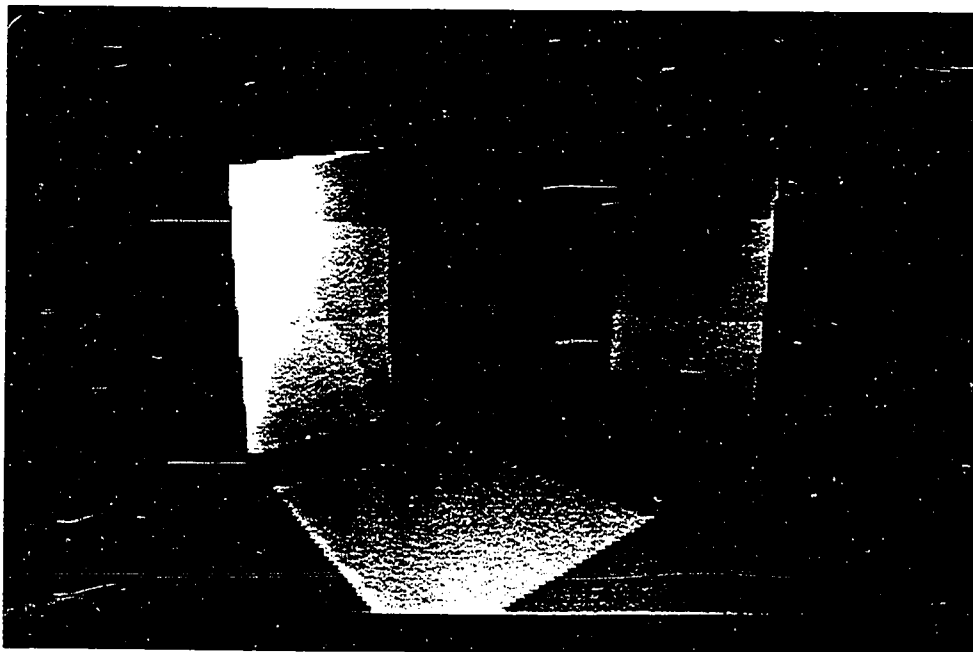


Figure 63: Caustics from two planes to a third

7. Conclusions

Advances in computer hardware are bringing three-dimensional computer animation closer to everyone, including scientists, educators, artists, and others. Designers of animation software will need to keep these people in mind when developing new systems. BAGS may serve as a prototype for such systems, since it provides a general framework upon which user-friendly interfaces may be built. In the next section we review the features of BAGS that make this framework possible and the research contributions that were developed with it. The section following that describes areas for future work.

Contributions

The Brown Animation Generation System (BAGS), outlined in Chapter 2, has been designed to be both a production system and a research testbed. We feel that combining these two broad categories of animation systems into a

single entity results in improvements in both areas. Animators can take advantage of new research developments incorporated into the system, while researchers have a broad animation base with which to work.

The SCEFO language, presented in Chapter 3 and in Appendix A, is a major contributor to the flexibility of BAGS. Localization of all modeling capabilities in the language allows the modeling and rendering components of BAGS to remain independent of each other. This independence permits various combinations of modeling and rendering approaches to be interchanged, providing a high degree of flexibility to the animator or researcher.

The renderer-independence of SCEFO allows any of a variety of rendering techniques to be used to produce a view of a modeled scene, as described in Chapter 4. BAGS includes several rendering methods, each of which represents a unique resolution to the trade-off between computation time and image quality. Common to (all but wire-frame) renderers is the BAGS lighting model, which translates intuitive, easy-to-interpolate surface parameters into realistic lighting effects.

One rendering method which has been improved through BAGS is ray tracing. In Chapter 5 we presented triangle tracing, an extension to ray tracing that uses an area-sampling technique to correct the problem of image aliasing. This method improves on existing area-sampling methods, in that it works with

any model and requires only conventional object intersection procedures. In Chapter 6 we presented another improvement to ray tracing. By using the triangle tracing algorithm for both cameras and lights, we are able to incorporate caustic interactions of light into ray traced images.

Future Work

BAGS is by no means complete. (In fact, no animation system can ever be considered “complete,” in the sense that there is always something that can be added to it.) However, it is designed to be extensible in several ways. Some areas for possible extension are presented below.

Interface Programs

While the SCEFO language is a reasonable interface for model specification, it is by no means the best way for all users to define animations. Front-end programs that “compile” into SCEFO descriptions would greatly improve the user-friendliness of BAGS. Some programs, such as a parametric surface modeler[28] and an interactive graphical animation modeler[35] with an object relation model[61], are currently being designed and implemented. Other applications, such as generalized simulation software, should be developed in the future.

Object Classes

The object-oriented treatment of objects in BAGS makes the installation of new object classes relatively easy, once the appropriate rendering methods have been developed. There are certain useful object classes currently missing from the BAGS repertoire, most notably spline patches and particle systems. Work is being done to incorporate these and other classes.

Dynamic polygonal objects compose one class of objects that is common in other production systems but has not been explored fully in BAGS. Polygonal objects are conducive to structural changes, since such changes require only movement of vertices and rearrangement of edge and face connectivity. Because BAGS is not geared towards polygons as most systems are, development of software to handle these objects has not been a priority.

Rendering

The primary challenge to developers of rendering software today is keeping up with hardware. Each new graphics workstation that comes on the market requires a slightly different form of rendering specification. Attempts to standardize these specifications may prove fruitful, but until then we will have to be tolerant of quirky interfaces. The use of the SCEFO language as rendering input allows BAGS renderers to share a great deal of code. This greatly

facilitates the development of new rendering back-ends to match devices' specifications.

The radiosity approach to lighting and rendering[36,16,48,82] incorporates diffuse-to-diffuse surface light interactions in the lighting model. This is an important idea that has produced strikingly realistic results for certain modeled environments. There may be a way to incorporate these diffuse interactions into the triangle tracing approach. One possible method is to use the caustics algorithm, with a modification: when adding intensity to a diffuse region, place a grid cube (like a light source grid cube) around the region, and trace rays through all triangles in the grid. The resulting rays would contribute light in a diffuse fashion to other surfaces. This approach may be too slow to be feasible, but may be adaptable to parallel implementations or hardware.

Other extensions to the algorithm may be developed to incorporate other "fuzzy" effects incorporated into other algorithms, such as soft shadows from distributed light sources, translucency, glossiness, motion blur, and depth of field. These effects all require information sampled over an area, so the triangle tracing algorithm may be useful in achieving them.

Appendix A

A SCEFO Reference

This appendix contains a syntactic and semantic description of the SCEFO animation language. It is not intended as a complete tutorial, but rather a reference guide. The features presented here all pertain to Version 2 of SCEFO.

The first section contains general comments about SCEFO. The four sections after that describe the SCEFO statements, organized by statement type. First are the object creation statements, **read**, **template**, **instance**, and **group**. The next section is all about the **change** statement and includes details about arithmetic expressions, fields, and other related topics. The section following that describes **action** and **apply** statements. The fourth statement section covers miscellaneous statements: **comment**, **loop**, **prefix**, and **assign**.

After the statement sections is a brief description of how new value functions, operator routines, and interpolation routines may be added to BAGS by animators.

General Comments

SCEFO is a free-format language. That is, white space (blanks, tabs, and newlines) may appear anywhere in a script and serves to separate tokens. Although indentation is optional, it is recommended for clarity. The examples in this guide may serve as a reference for style as well as syntax.

SCEFO script files are, by default, run through the C language preprocessor. The preprocessor allows commenting, constant definition, macro definition, file inclusion, and conditional acceptance of input. These features serve to make scripts more readable and are recommended when appropriate.

All names and keywords in SCEFO are case-sensitive. That is, the name `cube` is different from `Cube` or `CUBE`. A useful stylistic convention differentiates between template and instance names (and names of template and instance groups) by capitalizing the latter.

SCEFO does not allow forward references. Objects, actions, and variables must all be defined before they can be used. This rule prevents circular

definitions from occurring and generally results in more structured scripts.

Object Creation Statements

The statements in this section all produce a new object (template, instance, or group) with a given name. All object names must be unique; duplicates are reported as errors. Names are any string of fewer than 256 alphanumeric characters or underscores with no intervening white space.

The **read** statement creates a template from an OFF object stored in a file:

```
read ("cube.off") cube;
```

reads the object named `cube` from the file `cube.off`, creating a template named `cube` in the process. The template name does not have to be the same as the object name.

```
read ("cube.off") box = cube;
```

reads in the `cube` object, but creates a template named `box` from it. This feature can be used to avoid naming conflicts when reading similarly-named objects from different files.

Several objects can be read from the same OFF file with one **read** statement by separating the components by commas:

```
read ("furniture.off") bed, table, chair;
```

Objects read in this way may be renamed as well.

The **template** statement is used to create a copy of a template or to create a CSG combination of templates. Creating a copy of a template is very simple:

```
template (blue_chair) chair;
```

The name within parentheses is used for the new template, which is a copy of the other template.

A CSG template can be created from any Boolean combination of other templates. The symbols `|`, `&`, and `-` are used to denote the union, intersection, and difference operations, respectively. Parentheses may be used to change precedence, which is left-to-right by default:

```
template (widget) (doohickey | thang) - (rotor & sprocket);
```

The **instance** statement creates an instance from a template. The name of the instance appears within parentheses, followed by the template name:

```
instance (Axle) shaft;
```

creates an instance named `Axle` of the template named `shaft`.

The **group** statement creates a template group or an instance group, depending on what its members are. If `tree`, `rock`, and `mushroom` are all templates, then

```
group (lawn_items) tree, rock, mushroom;
```

creates the template group named `lawn_items` which contains the three templates. If the three items had been instances, an instance group would have been created. It is an error to combine instances and templates in the same group.

A template group may be used wherever it is valid to use a template. This includes **template** statements (both for copies and CSG) and groups. Similarly, instance groups may be used in other instance groups. In this way, hierarchically nested groups can be created.

A **change** statement modifies the state of a template, instance, or group. A change to a template affects that template and all templates and instances derived from it. A change made to a group affects all members of that group; if the group is a template group, the change also affects any templates created from that group; A change to an instance affects only that instance.

Each change consists of one or more change-ops, which contain the name of an operator and one or more control points. The change-ops are separated by commas, while the control values need no separating tokens. The operator may be one of the standard SCEFO operators or a user-defined one. Each control point has a time value and a control value, all within angle brackets. The control value may be a list of values, in which case the list appears within curly braces, as in the **scale** operator in the following example. The component values in the list may themselves be lists, as in the case of the **rotate** operator in the example.

```
change (Bottle) scale < 0, {1, 2, 3}>,
                rotate < 0, {{0,0,0}, {0,1,0},    0}>
                <10, {{0,0,0}, {0,1,0}, 43.2}>;
```

Values within control points are integers, real numbers, character strings (within double quotation marks), lists of component values. Most numbers used within control points may be either integers or real numbers, or arithmetic expressions. Expressions may contain the unary negation (symbol `-`) and logical-not (symbol `!`) operations, and the binary addition (symbol `+`), subtraction (symbol `-`), multiplication (symbol `*`), division (symbol `/`), and integer modulo (symbol `%`) operators. The C language conventions for integer-real expressions are supported, as are (recursive) operations on value lists:

```
12 + 11          -> 13
12 + 11.0        -> 13.0
1 / 2            -> 0
1.0 / 2          -> 0.5
1 + {2, 2.5, 3} -> {3, 3.5, 4}
{1, 3.2} * 5      -> {5, 16.0}
{1, 1.5} + {2.1, 4} -> {3.1, 5.5}
{3, 2} * {1.1, 2.2} -> {3.3, 4.4}
{1, {2, 3}} + {4, 5} -> {5, {7, 8}}
{1, 2, 3} + {4, 5} -> ERROR
```

A set of predefined functions is supported by SCEFO:

<code>abs(n)</code>	absolute value
<code>acos(n)</code>	arc cosine
<code>apply(func,list)</code>	applies the named function to each item in the list, returning a list with the results
<code>asin(n)</code>	arc sine
<code>atan(n)</code>	arc tangent in the range -90 to 90
<code>atan2(y,x)</code>	arc tangent in the range -180 to 180
<code>ceil(n)</code>	ceiling
<code>compare(a,b,l,e,g)</code>	returns 1 if $a < b$, e if $a = b$, and g if $a > b$
<code>cos(n)</code>	cosine
<code>cross(vec1,vec2)</code>	vector cross product
<code>dot(vec1,vec2)</code>	vector dot product
<code>exp(n)</code>	exponential function
<code>floor(n)</code>	floor
<code>i_to_s(n)</code>	converts integer to string
<code>length(vec)</code>	length of vector
<code>log(n)</code>	natural logarithm
<code>log10</code>	base 10 logarithm
<code>max(list)</code>	returns list item with greatest value
<code>min(list)</code>	returns list item with smallest value
<code>normal(vec)</code>	normalizes vector to unit length
<code>pow(a,b)</code>	raises a to power of b
<code>r_to_s(n)</code>	converts real to string
<code>random(n)</code>	returns random number between 0 and n
<code>round(n)</code>	rounds real number to integer
<code>s_to_i(string)</code>	converts string to integer
<code>s_to_r(string)</code>	converts string to real
<code>select(list,n)</code>	returns nth item in list
<code>sin(n)</code>	sine
<code>sqrt(n)</code>	square root
<code>strlen(string)</code>	string length (in characters)
<code>tan(n)</code>	tangent
<code>trunc(n)</code>	truncates real to integer

Expressions may be used for time values or control values in control points. For example,

```
change (turtle) scale <23 * .05, {1, 2, 3}>
                  <47 * .05, 5 * {1, 2, 3}>;
```

is a valid change.

User-defined and predefined fields may be used as values in or out of expressions. For example,

```
change (elevator) translate <0, {0, rope:length, 0}>;
```

translates the elevator template by the value of the length field of the rope object. When referring to a field on the object being changed, the object name prefix is optional; these two statements are equivalent:

```
change (car) translate <0, {0, 0, car:amount_to_move};
change (car) translate <0, {0, 0, :amount_to_move};
```

Predefined fields for an object are accessed in the same way. The fields currently supported in SCEFO are:

<code>_position</code>	three dimensional position of object
<code>_x_axis</code>	the x-, y-, and z-axes, after object
<code>_y_axis</code>	transformations are applied to them
<code>_z_axis</code>	
<code>_color</code>	RGB color of object
<code>_smoothness</code>	surface smoothness value
<code>_metalness</code>	surface metalness value
<code>_transparency</code>	surface transparency value
<code>_index</code>	index of refraction

The interpolation method for a change-op may be specified explicitly at the end of the change-op:

```
change (robot) rotate <0, {{0,0,0}, {1,0,0}, 0}>
                  <20, {{0,0,0}, {1,0,0}, 120}>
                  : discrete,
                  translate <0, {0, 0, 0}>
                  <10, {6, 9, 3}> : ease(3, 2.5);
```

specifies that the rotation is to be interpolated with the `discrete` method, and the translation is to be interpolated with the `ease` method. The latter method provides slow ease-in and ease-out of the translation. It takes optional parameters that specify the number of frames for the duration of the easing.

The object being changed is specified by name or by context. An object context consists of two or more object names, separated by apostrophes, that define a fully-qualified path from one object to another. The path may follow template creation (from one template or through CSG), instance creation, or

group inclusion. Examples of these are:

```

template (a)      b;
template (c)      a - (e | f);
instance (P)      c;
group (g)         a, c;

change (a'b)      ... ;
change (c'e)      ... ;
change (P'c)      ... ;
change (P'c'e)    ... ;
change (P'c'a'b)  ... ;
change (g'a)      ... ;
change (g'a'b)    ... ;

```

(In the example the ellipses represent the rest of the statement, which is unrelated to the topic at hand.) Each change to an object in context affects that object only when used in that context. For example, the change to $P'c$ in the example affects the c template, but only when used to create the instance P ; another instance created from c would not be affected by such a change.

Action and Apply Statements

The **action** statement is used to create procedure-like groups of changes. The action is given a name, which is used to refer to the action to apply it. It is defined with dummy parameters that are substituted with real values when the action is applied:


```

action (tilt_a_whirl) what, spins, tilt {
  change (&what&) rotate <0,  {{0,0,0}, {0,1,0}, 0}>
                        <10, {{0,0,0}, {0,1,0},
                          &spins& * 360}>,

                        rotate <0,  {{0,0,0}, {0,0,1}, 0}>
                        <10, {{0,0,0}, {0,0,1},
                          &tilt&}>;
}

```

The items within ampersands are substituted with the corresponding values when the action is applied. (The ampersands are used as delimiters at both ends so that the parameter names may be used as part of lexical tokens, such as names.)

The time scale used within an action is arbitrary. This scale is mapped to real scene time units when the action is applied.

The **apply** statement is used to apply an action. It specifies the name of the action, values to pass to each parameter, and control points that map action time to scene time. For example,

```

apply (tilt_a_whirl) wheel, 23, 12.7 * 3
                        <17, 0>
                        <36, 10>;

```

applies the action from the previous example to the wheel object, with the given number of spins and the tilt angle. Note that an expression may be used

as a parameter value. The control points specify that the action time 0 maps to scene time 17, and that action time 10 maps to scene time 36. The times are interpolated linearly, unless another interpolation method is specified for the apply:

```

apply (tilt_a_whirl) wheel, 23, 12.7 * 3
                                <17, 0>
                                <36, 10> : ease(4, 5);

```

specifies that the `ease` interpolation method is to be used when mapping the times. If the changes within the action also have interpolation methods specified, this has the effect of layering the interpolation methods.

An action may contain any number of **change** and **apply** statements. Using applies within an action creates hierarchical actions, which are often helpful for modeling complex motions.

Action parameters may have default values specified:

```

action (grow) what, xsize = 1, ysize = 1, zsize = 1 {
    change (&what&)
        scale <0, {&xsize&, &ysize&, &zsize&}>;
}

apply (grow) blob, 2, , 3 <5, 0>;

```

The **apply** statement in this example passes the value 2 for the `xsize` parame-

ter and the value 3 for the `ysize` parameter. The default value for the `ysize` parameter in the definition of the action is 1, so that is used for the apply.

Miscellaneous Statements

This section describes the SCEFO statements that do not fit into the other categories. These statements are provided mainly for user comfort.

The **comment** statement allows an animator to document a script. C language comments may be used in a script, since they are removed by the preprocessor. However, any program that reads the script will not see the comments. Therefore, a program that reads a script and writes out an altered version of it will not have these comments in it.

This is not the case with SCEFO comments, which are of the form:

```
comment ("This is a comment, for demo purposes only");
```

The comments have no effect on the script, but programs that read and write scripts will leave them intact. This feature may also be used by some programs to imbed extra information in scripts, leaving that information invisible to SCEFO.

The **loop** statement is used to save typing and script storage space. It allows a repetitive series of statements to be compressed into an iterated loop.

There are two forms, demonstrated in the following examples:

```
loop (i = 0 to 20 by 2) {
  template (row@i@) row;
  change (row@i@) translate <0, {@i@ * 2, 0, 0}>;
}

loop (value in {2.7, 3 * sin(23), 6}) {
  change (frantic_guy) translate <0, {@value@, 0, 0}>;
}
```

The first form implements a for-loop, substituting the updated value of the loop counter in the body of the loop. The “by” clause is optional. The second form is a foreach-loop, iterating once for each value in the value list.

The **prefix** statement is an aid to modular script design. It has two forms:

```
prefix (car_);

prefix ();
```

The first form adds the given prefix to the end of the current prefix string (which is initially the null string), while the second form removes the most recently added prefix. The prefix is added to all object and action names, unless they are escaped with the `~` character, in which case they are left as is. There is a max-

imum prefix nesting level of 32 in the current system.

The **assign** statement creates a variable and sets it to a value. It looks much like a **change** statement:

```
assign (direction) <0, {1, 1, 0}>  
                  <10, {0, 1, 1}>;
```

The named variable is assigned the values within the control points. As shown, the values may be lists. By default, linear interpolation is used, but other methods may be specified after a colon. **Assign** statements, like changes and applies, may be used within the bodies of actions.

Once a variable has been set, it may be used within another change, apply, or action; it is referred to with a dollar-sign prefix, such as `$direction` for the variable in the above example. The `$FRAME` variable is predefined by SCEFO, and always contains the current frame number.

The **loadfile** statement is used to specify that a binary object code file is to be dynamically loaded into the program processing the SCEFO script, e.g.:

```
loadfile ("/users/xyz/my_code.o");
```

This is needed to add any of the extensions described in the following section.

User-Defined Additions

Users may define change operators, interpolation methods, and value functions to augment the standard set included in SCEFO. An addition requires a **loadfile** statement to load the file containing the code, and one or more invocations of the code within the SCEFO script.

A value function is defined in terms of the VAL package's data structures. The user function is passed a function-expression item to evaluate and replace with the result of applying the function. For example, here is a (simple and unnecessary) function that returns the fourth root, by using two square root calls:

```
#include <math.h>
#include <brown/val.h>

int
fourth_root(id, item)
VALid    *id;
VALitem  *item;
{
    double arg;

    arg = VAL$REAL(VAL$FUNC_ARG(item));

    VAL$SET_REAL(item, sqrt(sqrt(arg)));

    return(1);
}
```

Of course, a better function would check the argument for validity, returning zero in error cases.

The above function could be used in a SCEFO script in the normal way:

```
loadfile ("whatever_file_the_code_is_in.o");  
change (thing)  scale  <0, {1, 1, fourth_root(231.2)}>;
```

Adding an interpolation method requires two things: a specification of value types that the method can receive as input, and the procedure that implements the interpolation. The value specification is in the form of a VALspec structure, a symbolic description of allowable VAL data types. Certain common specifications are predefined by VAL. For example, VAL\$SPEC_3R denotes a list of three reals, as might be needed for a three-dimensional vector, and VAL\$SPEC_NR denotes a list of any number of reals.

The interpolation method procedure is passed an array of control point times, an array of control values, and the frame time at which to interpolate. If the interpolation method requires or accepts parameters, they are passed as well. The procedure must determine, based on the frame time and the array of control point times, the interpolated value.

Adding a change operator is similar to adding an interpolation method. First, the user must specify the allowable argument types for the operator, using

a VALspec. The name of the default interpolation method to use when applying the operator is also required. The procedure that implements the operator is passed the interpolated values and the state of the object being affected. Most operators merely attach one or more SOI transformation nodes to the object state. These nodes are gathered and processed when the object is rendered.

Appendix B

The BAGS Lighting Model

This appendix is a summary of the lighting model implemented in BAGS by the LASH (Lighting And SHading) package. It describes the input parameters, called *basic items*, and computed values, called *computed items*.

For each item, the following information is listed: the item name, the symbol used to represent it in equations, its data type, and a description of the item, including its effect on lighting and any other important information. For a basic item, the default value is also given. This is the value that will be used if the user never sets that item. For a computed item, the items it normally depends on are listed, as is the equation used to compute its value.

The symbol used for an item depends on its type. Symbols for single-valued items are lower-case Roman letters, except those having to do with angles, which are Greek letters. Symbols for points, vectors, and RGB color triples are in upper case.

Some of the computations are different for each of the three types of light sources. Ambient sources, in particular, are treated specially, and have their own section below.

A complex renderer, such as a ray tracer, may need global lighting values as well as the ones local to the point being shaded. LASH provides some additional items to handle this. These items appear in their own section below.

The items in each section are listed in the order in which they are evaluated or computed.

Basic Items

Surface point

Symbol: P
Type: 3D point
Default value: (0, 0, 0)

This is the point on the surface whose lighting is to be computed. It is usually not a good idea to change this value with functions, since renderers may not be able to cope.

Surface normal

Symbol: N
Type: unit vector
Default value: (0, 0, 1)

This is the normal to the surface at the point being lit. It points outwards from the surface. Diffuse lighting will be brightest if the direction of a light source is exactly opposite to this vector.

Surface color

Symbol: C
Type: RGB triple
Default value: (1, 1, 1)

This is the RGB color of the surface at the point being shaded. The color affects the color of the diffusely-reflected light, and, for metallic surfaces, the specularly-reflected light. Each of the red, green, and blue components of the color should be between 0 and 1, inclusive.

Surface smoothness

Symbol: s
Type: single-valued
Default value: 0

This is an indication of how smooth a surface material is. (This does not mean that the surface does not have macro-features such as mountains, but that it doesn't have micro-features that make a surface look "chalky.") The smoothness affects how much light is reflected diffusely, and how much is reflected specularly. For specular reflection, it also determines how much the light spreads after reflecting from the surface (a smoother surface spreads the light less).

A smoothness of 0 means that the surface is completely diffuse, and will not show specular highlights (excepting Fresnel effects, described below). A smoothness of 1 indicates that the surface is a perfect mirror; all light reaching the surface will reflect from it, unless it is also transparent.

Surface metalness

Symbol: m
Type: single-valued
Default value: 0

This quantity is used to indicate whether a surface material is metallic or not. The metalness affects the color of specularly reflected light: a metallic surface reflects the color of the surface, while a non-metallic surface reflects the color of the light source. A metalness of 1 means that the material is metallic, while 0 means that it is non-metallic. Values between 0 and 1, while not entirely realistic, are valid.

Surface transparency

Symbol: t
Type: single-valued
Default value: 0

This defines how much light can pass through the surface material. A transparency of 0 means that the surface is totally opaque; no light at all will be transmitted through it. A transparency of 1 means that all of the light incident on the surface will pass through it, except for Fresnel effects, as described below. This item should not be confused with transmittance, which is defined later.

Surface index of refraction

Symbol: n
Type: single-valued
Default value: 1

The index of refraction determines how a ray of light will bend when passing through the surface. This number corresponds to the real index for materials, as listed in any general physics or optics book. For air or vacuum, it is 1, meaning that light passes straight through without bending. The denser the material, the higher its index; most types of glass have an index of about 1.5. The index of refraction should never be 0, since this may result in a division by 0. Changing the index has absolutely no effect on surfaces that have transparency of 0.

Source location

Symbol: L_p
Type: 3D point
Default value: (0, 0, 0)

This is used only for point light sources, to determine the position of the source. The vector from this point to the surface point is used as the light source direction for point sources.

Source direction

Symbol: L
Type: unit vector
Default value: (0, 0, -1)

For directional light sources, this is the vector along which the light travels. For point sources, this is a computed item (see below).

Source intensity

Symbol: I
Type: RGB triple
Default value: (1, 1, 1)

This is the amount of light radiating from a source. The red, green, and blue components can be used to change the color as well as the intensity of the light. These values are typically between 0 and 1, although perhaps “interesting” effects may be created by experimenting with other values.

View direction

Symbol: V
Type: unit vector
Default value: (0, 0, -1)

This vector defines the direction along which the surface point is viewed. It is used to determine the brightness of the specularly-reflected light (highlight). If the ray of light reflected by the surface point is parallel to the view direction vector (but in the opposite direction), the highlight will be brightest.

Computed Items for Point and Directional Sources

Source direction

Symbol:	L
Type:	unit vector
Depends on:	surface point, surface location
Equation:	$L = \frac{P - L_p}{\ P - L_p\ }$

For point sources, the source direction is computed by subtracting the surface point from the source location, and normalizing the result to get a unit vector.

Shadowed intensity

Symbol:	I_s
Type:	RGB triple
Depends on:	source intensity
Equation:	$I_s = I$

The shadowed intensity is, by default, equal to the source intensity. This item is included primarily for use by renderers that compute shadows. Those programs can provide LASH functions to set or modify the shadowed intensity, based on other information available to them, without changing the source intensity.

Diffuse reflectivity

Symbol:	r_d
Type:	single-valued
Depends on:	surface smoothness, surface transparency
Equation:	$r_d = (1 - s^3)(1 - t)$

This item is a measure of how much light will be reflected diffusely. A less smooth, or “chalkier,” surface will reflect more light diffusely than a smooth one. A transparent surface reflects less light diffusely than an opaque one. The smoothness is cubed in the equation to produce a more realistic correlation between the smoothness parameter and the reflectivity of the surface.

Cosine of incident angle

Symbol:	$\cos\alpha$
Type:	single-valued
Depends on:	surface normal, source direction
Equation:	$\cos\alpha = N \cdot L$

The incident angle determines the intensity of the diffusely-reflected light. The closer the source direction is to being directly opposite the surface normal, the higher the intensity. The cosine of the incident angle is used to model this effect. It is easily computed as the dot product of the surface normal and light source direction. It should never be negative, since that would indicate that the surface is not visible from the light source; LASH changes negative values to 0.

Lambert lighting term

Symbol: I
 Type: single-valued
 Depends on: diffuse reflectivity, cosine of incident angle
 Equation: $I = r_d \cos \alpha$

Lambert lighting models diffusely-reflected light. The reflected intensity depends on the incident angle and the reflectivity of the surface. This item will be 1 for a completely non-smooth, opaque surface being illuminated by a light directed along the surface normal (in the opposite direction), and will be between 0 and 1 for other configurations.

Diffuse contribution

Symbol: K_d
 Type: RGB triple
 Depends on: surface color, Lambert lighting term
 Equation: $K_d = IC$

This is the amount of diffuse light that will be reflected from a surface, assuming a source intensity of red = green = blue = 1. (The actual source intensity is used later.) It is the product of the Lambert term and the surface color, since diffusely-reflected light always has the same color as the surface material.

Highlight direction

Symbol: H
 Type: unit vector
 Depends on: surface normal, source direction, cosine of incident angle
 Equation: $H = L + 2\cos \alpha N$

This is the vector along which the brightest specular reflection will occur. It is calculated as the direction to which the source direction vector will be reflected by the surface, with respect to the surface normal. This uses the rule that the angle of reflection is equal to the angle of incidence.

Cosine of specular angle

Symbol: $\cos\beta$
Type: single_valued
Depends on: view direction, highlight direction
Equation: $\cos\beta = -H \cdot V$

The specular angle is the angle between the view direction vector and the highlight direction vector. The specular highlight will appear brightest when this angle is 0; i.e., when the view direction is parallel to the highlight direction, in the opposite direction. The cosine of the angle is computed by using a dot product; since the vectors are oriented in opposite directions, the result is negated to make the result lie between 0 and 1.

Specular exponent

Symbol: h
Type: single-valued
Depends on: surface smoothness
Equation: $h = \frac{3}{1 - s}$

The specular exponent helps determine the size of the spread of the specular highlight (see the description of spread below). It is the power to which the cosine of the specular angle is raised to compute the spread. A perfect mirror will have an infinite exponent, while a rough surface can have one as low as 3.

Specular spread

Symbol:	p
Type:	single-valued
Depends on:	cosine of specular angle, specular exponent
Equation:	$p = \cos^k \beta$

This is the amount by which specularly-reflected light will spread from the highlight direction, and, consequently, the relative size of a specular highlight. On a very smooth surface, the reflected light will not spread much (the spread value will be close to 0), and the highlight will appear fairly small. A rougher surface will spread the highlight more (the spread will be closer to 1).

Normal reflectivity

Symbol:	r_n
Type:	single-valued
Depends on:	surface transparency, diffuse reflectivity
Equation:	$r_n = (1 - t) - r_d$

This is the amount of light that the surface would reflect specularly if the ray of light was directed along the surface normal, but in the opposite direction. For opaque surfaces, it is the same as the diffuse reflectivity. Transparent surfaces let some of the light through, so the reflectivity will be lower for them.

Fresnel term

Symbol: f
Type: single-valued
Depends on: cosine of incident angle
Equation: $f = F(\cos\alpha)$

Fresnel's equation states that the ratio between incident and reflected intensities depends on the angle of incidence. That is, as the angle of incidence increases to 90 degrees, the reflectivity increases to 1. Although it was formulated for transparent non-metallic materials, the effect occurs for all materials. This term approximates the effects of the Fresnel equation with the **F** function. The function returns 0 when the angle of incidence is 0 (i.e., the light direction is perpendicular to the surface), and returns 1 when the angle of incidence is 90 degrees, rising sharply near the 90-degree mark. The closer the Fresnel term is to 1, the higher the reflectivity.

Geometric attenuation factor

Symbol: g
Type: single-valued
Depends on: cosine of incident angle
Equation: $g = G(\cos\alpha)$

This term attenuates the Fresnel reflectivity near the grazing angle, for surfaces that are less than perfectly smooth. It compensates for the masking of incident and reflected light by the microfeatures of the surface when the incident angle is near 90 degrees. The factor is approximated by the **G** function, which is similar to the **F** function, but inverted. The **G** function returns 1 when the angle of incidence is 0 and returns 0 when the angle of incidence is 90 degrees, falling very sharply near the 90-degree mark.

Reflectivity adjustment

Symbol: a
Type: single-valued
Depends on: Fresnel term, geometric attenuation factor
Equation: $a = fg$

This is the product of the Fresnel term and the geometric attenuation factor. It is the amount the specular reflectivity is multiplied to compensate for the Fresnel formula.

Adjusted reflectivity

Symbol: r_a
Type: single-valued
Depends on: normal reflectivity, reflectivity adjustment
Equation: $r_a = r_n + (1 - r_n)a$

This is the reflectivity of the surface after considering the Fresnel formula and geometric attenuation. If the adjustment term is 0, the adjusted reflectivity is equal to the normal reflectivity. As the adjustment term increases to 1, the adjusted reflectivity increases to 1 as well.

Specular reflectivity

Symbol: r_s
Type: single-valued
Depends on: spread, adjusted reflectivity
Equation: $r_s = r_a p$

This is the reflectivity of the surface with respect to specular light. It represents the amount of light traveling along the source direction that will be reflected by the surface. It takes into account the adjusted reflectivity and the specular spread factor.

Specular color ratio

Symbol: q
 Type: single-valued
 Depends on: surface metalness, reflectivity adjustment
 Equation: $q = m(1 - a)$

This determines whether the color of specularly-reflected light will be closer to the color of the light source or the surface. Non-metallic objects always reflect the color of the source. Metallic objects usually reflect the color of the surface material, except when the Fresnel reflectivity approaches 1, in which case the reflected light takes on the color of the incident light from the source. This effect parallels the change in reflectivity modeled by the reflectivity adjustment, so it is used again here. The specular color ratio is 0 if the color should be that of the source, and is 1 if the color should be that of the surface material. Values between 0 and 1 indicate that the color is a combination of the two.

Specular color

Symbol: C_s
 Type: RGB triple
 Depends on: surface color, specular color ratio
 Equation: $C_s = C_1 + q(C - C_1)$

This is the color of the specular reflection, assuming the color of the light source is red = green = blue = 1 (sources of other colors are compensated for later). It uses the specular color ratio to interpolate between the surface color and pure white, denoted by the C_1 term, the color with all three components equal to 1.

Specular contribution

Symbol: K_s
 Type: RGB triple
 Depends on: specular reflectivity, specular color
 Equation: $K_s = r_s C_s$

This is the amount of light that will be reflected specularly from a surface, assuming a source intensity of red = green = blue = 1. (The actual source intensity is used later.) It is the product of the specular reflectivity and the specular color.

Total contribution

Symbol: K_t
 Type: RGB triple
 Depends on: diffuse contribution, specular contribution
 Equation: $K_t = K_d + K_s$

This is the total amount of light that will be reflected from a surface, assuming a source intensity of red = green = blue = 1. It is the sum of the diffuse and specular contributions. Note that this is computed differently for ambient sources.

Reflected intensity

Symbol: I_r
 Type: RGB triple
 Depends on: shadowed intensity, total contribution
 Equation: $I_r = I_s K_t$

This is the amount of light reflected from a surface, given the actual source intensity. The shadowed intensity is used so that the reflected intensity can be attenuated or nullified for a surface point in shadow. This item represents the value most often needed when computing lighting at a surface point; it is the amount of light that reaches the eye or pixel. This is computed differently for ambient sources.

Computed Items for Ambient Sources

Diffuse reflectivity

Symbol: r_d
Type: single-valued
Depends on: surface smoothness, surface transparency
Equation: $r_d = (1 - s^3)(1 - t)$

This is computed the same way for ambient sources as for other sources, and has the same effect.

Total contribution

Symbol: K_t
Type: RGB triple
Depends on: surface color, diffuse reflectivity
Equation: $K_t = r_d C$

This is the total amount of light that will be reflected from a surface for an ambient source with an intensity of red = green = blue = 1. Note that this is computed differently for ambient sources than for others. Because ambient sources have no direction, they do not result in Lambertian lighting effects or create specular highlights. Therefore, this item is the merely the product of the color and the diffuse reflectivity of the surface.

Reflected intensity

Symbol: I_r
Type: RGB triple
Depends on: source intensity, total contribution
Equation: $I_r = IK_i$

This is the amount of light reflected from a surface, given the actual intensity of the light. This is also computed differently for ambient sources than for others. There is no need to use the shadowed intensity, since ambient lights never cast shadows.

Computed Items for Global Lighting

Important note: These items are included for renderers such as ray tracers that need global lighting information. The items deal with light that reflects from surfaces and refracts through surfaces. They are computed by following the light source direction vector as it reflects from or is transmitted through the surfaces. A conventional ray tracer, however, needs to determine the same information for the view direction, instead of the source direction. An easy way to accomplish this is to create a dummy directional source whose direction is the view direction. Using this vector, LASH will compute the required item values. For example, the specular highlight vector in such a case will be precisely the direction the view vector would travel after reflecting from the surface, with respect to the surface normal. This vector is used in conventional ray tracing to find objects that are seen as reflections in the object being shaded.

Transmittance

Symbol:	z
Type:	single-valued
Depends on:	surface transparency, normal reflectivity, specular reflectivity
Equation:	$z = \frac{(1 - r_s)t}{1 - r_n}$

This is the amount of light that actually passes through the surface along the source direction, after accounting for Fresnel effects. (If more light is reflected from a surface, less will pass through it.) This can be used to simulate transparency more accurately for any rendering technique, and can be used by a ray tracer as a coefficient for refracted rays.

Transmitted direction entering surface

Symbol:	T_1
Type:	unit vector
Depends on:	surface normal, surface index of refraction, source direction
Equation:	$T_1 = S_1 + \sqrt{1 - \ S_1\ ^2}N$

This is the direction along which the light from a source will travel after entering the surface from air. The S_1 vector in the equation is defined as

$$S_1 = \frac{1}{n}[L - (L \cdot N)N]$$

The index of refraction of the surface material determines how much the light will bend as it passes through, according to Snell's law. This vector does not always exist; in the case of total internal reflection, the value of this item will be set to the vector (0,0,0) to denote this.

Transmitted direction exiting surface

Symbol: T_2
 Type: unit vector
 Depends on: surface normal, surface index of refraction, source direction
 Equation: $T_2 = S_2 + \sqrt{1 - \|S_2\|^2}N$

This is the direction along which the light from a source will travel after exiting the surface into air. (It is assumed that the light ray originated inside the object.) The S_2 vector in the equation is defined as

$$S_2 = n[L - (L \cdot N)N]$$

The cautions listed for the previous item are true for this one as well.

Appendix C

Triangle Tracing Code

This appendix contains some C code from an implementation of the triangle tracing algorithm presented in Chapter 5. For the sake of brevity, only the code sections that are unique to triangle tracing are included. The missing subroutines, such as object intersection and lighting computation, are common to most ray tracing programs and do not warrant inclusion here.

Much of the code has been simplified to make it clearer. Because of this, certain optimizations have been removed.

Data Structures

```

/* A ray defined by a point and a direction vector */
struct Ray {
    double    p[3],          /* Starting point of ray      */
              d[3];          /* Direction vector           */
};

/* The intersection of a ray with an object */
struct Inter {
    Object    *obj;          /* Object intersected         */
    int       zone;          /* Zone of object surface     */

    <surface properties: color, reflectivity, etc.>
};

/* A node in a tree of rays */
struct Ray_tree {
    int       depth;         /* Depth of subtree rooted at node */
    Ray       ray;           /* Ray for tree node            */
    Inter     inter;         /* Intersection of ray with object */
    Ray_tree  *refl,         /* Subtree for reflected ray (or NULL) */
              *tran;        /* Subtree for transmitted ray (or NULL) */
};

/* A side of a triangle being traced */
struct Side {
    Ray_tree  *t1, *t2,      /* Ray trees at endpoints of side */
              **bounds;      /* Array of boundary ray trees     */
    int       num_bounds;    /* Number of boundary ray trees (even) */
};

/* This macro sets the contents of a Side structure. It is used for brevity */
#define SET_SIDE(S, T1, T2, B, NB) (
    S.t1      = T1, (
    S.t2      = T2, (
    S.bounds   = B, (
    S.num_bounds = NB

```

Mainline

```

main()
{
    Ray      *ul_ray, *ur_ray, *ll_ray, *rr_ray;
    Ray_tree *ul_tree, *ur_tree, *ll_tree, *rr_tree;
    Side      s1, s2, s3;
    int       pixel[2];

    /* Trace triangles for each pixel in the image plane */
    for each pixel = (i,j) in image {

        /* Compute rays from camera point to corners of pixel */
        ul_ray = new_ray(camera_point, upper-left corner of pixel);
        ur_ray = new_ray(camera_point, upper-right corner of pixel);
        ll_ray = new_ray(camera_point, lower-left corner of pixel);
        lr_ray = new_ray(camera_point, lower-right corner of pixel);

        /* Create ray trees for each of the four rays */
        ul_tree = create_ray_tree(ul_ray, 1, 1.0);
        ur_tree = create_ray_tree(ur_ray, 1, 1.0);
        ll_tree = create_ray_tree(ll_ray, 1, 1.0);
        lr_tree = create_ray_tree(lr_ray, 1, 1.0);

        /* Set up three sides of lower-left triangle */
        SET_SIDE(s1, ul_tree, ll_tree, NULL, 0); /* Left side */
        SET_SIDE(s2, ll_tree, lr_tree, NULL, 0); /* Bottom   */
        SET_SIDE(s3, lr_tree, ul_tree, NULL, 0); /* Diagonal  */

        /* Compute boundary rays on all three sides */
        bound_side(&s1);
        bound_side(&s2);
        bound_side(&s3);

        /* Process the lower-left triangle: coverage = one half pixel */
        subd_tri(pixel, &s1, &s2, &s3, 0.5);

        /* Set up 2 sides of upper-right triangle (diagonal is same as before) */
        SET_SIDE(s1, ul_tree, ur_tree, NULL, 0); /* Top      */
        SET_SIDE(s2, ur_tree, lr_tree, NULL, 0); /* Right side */

        /* Compute boundary rays on both new sides */
        bound_side(&s1);
        bound_side(&s2);

        /* Process the upper-right triangle: coverage = one half pixel */
        subd_tri(pixel, &s1, &s2, &s3, 0.5);
    }
}

```

Ray Tree Comparison and Creation

```

/* This returns TRUE if the two given trees intersect the same zones of the
 * same objects along their entire paths. This calls itself recursively, and
 * assumes that neither of the trees is NULL. */

int
same_tree(t1, t2)
Ray_tree *t1, *t2;
{
    /* See if the two rays hit the same object zone and call recursively */
    return(t1->inter->obj == t2->inter->obj &&

        (t1->inter->obj == NULL || t1->inter->zone == t2->inter->zone) &&

        (t1->refl == NULL || t2->refl == NULL ? (t1->refl == t2->refl) :
         tree_same_zone(t1->refl, t2->refl)) &&

        (t1->tran == NULL || t2->tran == NULL ? (t1->tran == t2->tran) :
         tree_same_zone(t1->tran, t2->tran)));
}

```



```

/* This creates a ray tree (or subtree) for the given ray. The depth of
 * the ray within the tree is passed in to allow pruning to a maximum depth.
 * The contribution of the ray (i.e., the product of reflectivities and
 * transparencies along the ray path) is used to terminate the recursion
 * when continuing would have little or no effect. */

Ray_tree *
create_ray_tree(ray, cur_depth, contrib)
Ray      *ray;
int      cur_depth;
double   contrib;
{
    int      hit;           /* Whether ray hits an object */
    Inter    inter;        /* Intersection of ray with object */
    Ray_tree *t;           /* Tree created for ray */
    Ray      *new_ray;      /* New ray for reflection or refraction */

    /* See if ray hits anything */
    hit = intersect(ray, &inter, ENTERING);

    /* Level-1 trees will never be NULL; others won't be if they hit an obj */
    if (cur_depth == 1 || hit) {
        /* Create a new Ray_tree node with the given contents */
        t = allocate_ray_tree(cur_depth, ray, &inter);

        /* Trace reflected and transmitted rays if necessary */
        if (hit && t->depth < MAX_TREE_DEPTH) {

            /* If reflected ray has enough of a contribution */
            if (contrib * t->inter->reflectivity > REFLEC_THRESHOLD) {
                /* Compute reflected ray */
                new_ray = reflect_ray(ray, &inter);
                /* Create sub-tree for reflected ray */
                t->refl = create_ray_tree(&new_ray, cur_depth + 1,
                                          contrib * t->inter->reflectivity);
            }
            else t->refl = NULL;

            /* Same procedure for transmitted ray, except ray has to enter
             object and then exit it */
            if (contrib * t->inter->transparency > TRANSP_THRESHOLD &&
                refract_ray(ray, &inter, ENTERING, &new_ray) &&
                intersect(&new_ray, &inter, EXITING) &&
                refract_ray(&new_ray, &inter, EXITING, &new_ray))
                t->tran = create_ray_tree(&new_ray, cur_depth + 1,
                                          contrib * t->inter->transparency);
            else t->tran = NULL;
        }
        else t->refl = t->tran = NULL;
    }
    /* If no intersection, no tree */
    else t = NULL;

    return(t);
}

```

Computing Boundary Rays

```
/* Finds ray trees for boundary rays between the two ray trees at the ends of
 * the given Side. The bounds array and num_bounds fields of the Side are
 * filled in with the results. */

void
bound_side(s)
Side *s;
{
    /* If ray trees are the same, there are no boundaries between them */
    if (same_tree(s->t1, s->t2)) {
        s->num_bounds = 0;
        s->bounds = NULL;
    }

    /* Otherwise, use bisection process to find boundaries */
    else bound_bisect_side(s);
}
```

```

/* Finds boundary rays for the given side, by bisection */

void
bound_bisect_side(s)
Side *s;
{
    Side    s1, s2;
    Ray_tree *end1, *end2, *t;
    int     i, j;

    end1 = s->t1; end2 = s->t2;

    /* Loop until boundary rays are close enough together */
    while (dot_product(end1->ray.d, end2->ray.d) < BOUND_MIN) {

        /* Bisect rays, creating a new tree */
        t = create_ray_tree(average_rays(end1->ray, end2->ray), 1, 1.0);

        /* If t is same as one end, continue bisection loop */
        if (same_tree(end1, t)) end1 = t;
        else if (same_tree(end2, t)) end2 = t;
        /* Otherwise, recurse */
        else {
            /* Find boundary lists for both halves */
            s1.t1 = end1; s1.t2 = t;
            s2.t1 = t; s2.t2 = end2;

            bound_side(&s1); bound_side(&s2);

            /* Create an array to hold the resulting sublists */
            s->num_bounds = s1.num_bounds + s2.num_bounds;
            s->bounds = allocate_bounds_array(s->num_bounds);

            /* Join the two sublists into the new array */
            for (i = j = 0; i < s1.num_bounds; i++, j++)
                s->bounds[j] = s1.bounds[i];
            for (i = 0; i < s2.num_bounds; i++, j++)
                s->bounds[j] = s2.bounds[i];

            return; /* Exit loop and procedure */
        }
    }

    /* If it gets here, there are only 2 boundary rays */
    s->bounds = allocate_bounds_array(2);
    s->bounds[0] = end1;
    s->bounds[1] = end2;
    s->num_bounds = 2;
}

```

Triangle Subdivision

```

/* This returns the index of the centermost boundary ray in an array of
 * boundary rays, given the number of rays in the array */
#define CENTER_BOUND_INDEX(NUM)    (2 * ((NUM) - 1) / 4))

/* If the given triangle of sides is homogeneous, this calls shade() to shade
 * the triangles formed by the three vertex ray trees. Otherwise, it calls
 * subd_split() to subdivide the triangle. */

void
subd_tri(pixel, s1, s2, s3, coverage)
int    pixel[2];
Side   *s1, *s2, *s3;
double coverage;
{
    int b1, b2;

    /* If triangle is too small to worry about, ignore it */
    if (coverage < TRIVIAL_COVERAGE) return;

    /* If there are no bounds on at least two sides, it's homogeneous */
    if ((s1->num_bounds > 0) + (s2->num_bounds > 0)
        + (s3->num_bounds > 0) < 2) {

        /* Shade triangles only if the primary rays hit an object */
        if (s1->t1->inter->obj != NULL)
            shade(s1->t1, s2->t1, s3->t1, pixel, coverage);
    }

    /* Otherwise, have to subdivide */
    else {
        /* Determine corner to cut off and boundaries to join, then split */
        switch(subd_corner(s1, s2, s3, &b1, &b2)) {
            case 1:
                subd_split_tri(pixel, s1, s2, s3, b1, b2, coverage);
                break;
            case 2:
                subd_split_tri(pixel, s2, s3, s1, b1, b2, coverage);
                break;
            case 3:
                subd_split_tri(pixel, s3, s1, s2, b1, b2, coverage);
                break;
        }
    }
}

```

```

/* This determines the corner of the triangle defined by the three sides that
 * should be cut off to subdivide the triangle, setting the b1 and b2
 * parameters to the indices of the bounds on the two sides being joined.
 * If any corner is a "solid" triangle - i.e., one that creates no extra
 * boundaries when it is cut off - that corner is used. Otherwise, the
 * centermost bounds on the sides with the most bounds are connected. */

int
subd_corner(s1, s2, s3, b1, b2)
Side    *s1, *s2, *s3;
int      *b1, *b2;
{
    int corner;

    /* Find first solid corner, if any */
    if (s1->num_bounds > 0 && s2->num_bounds > 0 &&
        same_tree(s1->bounds[s1->num_bounds - 2], s2->bounds[1])) corner = 1;

    else if (s2->num_bounds > 0 && s3->num_bounds > 0 &&
        same_tree(s2->bounds[s2->num_bounds - 2], s3->bounds[1])) corner = 2;

    else if (s3->num_bounds > 0 && s1->num_bounds > 0 &&
        same_tree(s3->bounds[s3->num_bounds - 2], s1->bounds[1])) corner = 3;

    else corner = 0;

    /* If a solid corner was found, return it */
    if (corner > 0) {
        *b1 = (corner == 1 ? s1->num_bounds :
                corner == 2 ? s2->num_bounds : s3->num_bounds) - 1;
        *b2 = 0;
    }

    /* If none, use centermost method */
    else corner = subd_centermost(s1, s2, s3, b1, b2);

    return(corner);
}

```

```

/* This finds and returns the centermost bounds on the two sides with the
 * most bounds. */

int
subd_centermost(s1, s2, s3, b1, b2)
Side    *s1, *s2, *s3;
int      *b1, *b2;          /* Returns bound indices on two sides */
{
    int which, num;

    /* Find side with fewest bounds */
    which = 1;
    num = s1->num_bounds;
    if (s2->num_bounds < num) {
        which = 2;
        num = s2->num_bounds;
    }
    if (s3->num_bounds < num) which = 3;

    /* Centermost bounds of other sides will be joined */
    switch (which) {
        case 1:
            which = 2;
            *b1 = CENTER_BOUND_INDEX(s2->num_bounds) + 1;
            *b2 = CENTER_BOUND_INDEX(s3->num_bounds);
            break;
        case 2:
            which = 3;
            *b1 = CENTER_BOUND_INDEX(s3->num_bounds) + 1;
            *b2 = CENTER_BOUND_INDEX(s1->num_bounds);
            break;
        case 3:
            which = 1;
            *b1 = CENTER_BOUND_INDEX(s1->num_bounds) + 1;
            *b2 = CENTER_BOUND_INDEX(s2->num_bounds);
            break;
    }

    return(which);
}

```

```

/* This subdivides the triangle (given by three sides) by connecting boundary
 * b1 of side s1 with boundary b2 of side s2. This produces a new triangle
 * and a quadrilateral. */

void
subd_split_tri(pixel, s1, s2, s3, b1, b2, coverage)
int pixel[2];
Side *s1, *s2, *s3;
int b1, b2;
double coverage;
{
    Side ts1, ts2, ts3, ts4;
    double frac1, frac2;

    /* Create sub-triangle from ts1, ts2, and ts3 */
    SET_SIDE(ts1, s1->t1, s1->bounds[b1], s1->t2,
             s1->bounds + b1 + 1, s1->num_bounds - b1 - 1);
    SET_SIDE(ts2, s2->t1, s2->bounds[b2], s2->bounds, b2);
    SET_SIDE(ts3, ts2.t2, ts1.t1, NULL, 0);

    /* Compute boundaries on new side of triangle */
    bound_side(&ts3);

    /* Figure fraction of original triangle sides in subtriangle sides */
    frac1 = side_ratio(&s1->t2->ray, &ts1.t1->ray, &s1->t1->ray);
    frac2 = side_ratio(&s2->t1->ray, &ts2.t2->ray, &s2->t2->ray);

    /* Process sub-triangle recursively */
    subd_tri(pixel, &ts1, &ts2, &ts3, coverage * frac1 * frac2);

    /* Create quadrilateral from ts1, ts2, ts3, and ts4 */
    SET_SIDE(ts1, s1->t1, s1->bounds[b1 - 1], s1->bounds, b1 - 1);
    SET_SIDE(ts2, s1->bounds[b1 - 1], s2->bounds[b2 + 1], NULL, 0);
    SET_SIDE(ts3, s2->bounds[b2 + 1], s2->t2, s2->bounds + b2 + 2,
             s2->num_bounds - b2 - 2);
    SET_SIDE(ts4, s3->t1, s3->t2, s3->bounds, s3->num_bounds);

    /* Compute boundaries on new side of quadrilateral */
    bound_side(&ts2);

    /* Process quadrilateral */
    subd_quad(pixel, &ts1, &ts2, &ts3, &ts4, frac1, frac2, coverage);
}

```

```

/* This splits a quadrilateral into two triangles by the diagonal between
 * sides s1,s2 and s3,s4. */

void
subd_split_quad(pixel, s1, s2, s3, s4, coverage1, coverage2)
int    pixel[2];
Side   *s1, *s2, *s3, *s4;
double coverage1, coverage2; /* Coverages of triangles on both sides
                             of diagonal */
{
    Side ts1, ts2, ts3;

    /* Create a side to represent diagonal */
    ts1 = *s1;
    ts2 = *s2;
    SET_SIDE(ts3, s2->t2, s1->t1, NULL, 0);

    /* Find boundary rays on that side */
    bound_side(&ts3);

    /* Subdivide triangle on one side of diagonal */
    subd_tri(pixel, &ts1, &ts2, &ts3, frac1);

    /* Repeat process on other side of diagonal */
    ts1 = *s3;
    ts2 = *s4;
    ts3.t1 = s4->t2;
    ts3.t2 = s3->t1;

    /* The third side is the same as before, but in the opposite order: reverse
       the order of the boundary rays */
    bound_reverse(&ts3);

    subd_tri(pixel, &ts1, &ts2, &ts3, frac2);
}

```



```

/* The three given rays are assumed to lie in the same plane and have the same
 * starting point. Consider the intersection of the three rays with a line
 * in the plane. Call the intersection points p1, p2, and p3. This finds the
 * ratio of the length of segment p1p2 to the length of segment p1p3. This
 * ratio is used to compute the area of subdivided triangles. */

double
side_ratio(r1, r2, r3)
Ray *r1, *r2, *r3;
{
    double cosalpha, cosbeta, a, b;

    /* Let cosalpha be the cosine between r1 and r2,
       cosbeta be the cosine between r2 and r3 */
    cosalpha = dot_product(r1->d, r2->d);
    cosbeta = dot_product(r2->d, r3->d);

    /* a and b are the lengths of the segments p1p2 and p2p3, respectively */
    a = 1.0 / (cosalpha * cosalpha) - 1.0;
    b = 1.0 / (cosbeta * cosbeta) - 1.0;

    /* Check boundary conditions */
    if (a <= 0.0) return(0.0);
    if (b <= 0.0) return(1.0);

    a = sqrt(a);
    b = sqrt(b);

    /* Return the desired ratio */
    return(a / (a + b));
}

```

```

/* This determines the diagonal along which to divide a quadrilateral into two
 * triangles. If it can find a diagonal that does not introduce new boundary
 * rays, that diagonal is used. Otherwise, it chooses the diagonal that keeps
 * the two sides with the most boundary rays together. */

void
subd_quad(pixel, s1, s2, s3, s4, frac1, frac2, coverage)
int pixel[2];
Side *s1, *s2, *s3, *s4;
double frac1, frac2,          /* Fraction of sides of original triangle
                                covered by quadrilateral sides */
coverage;                     /* Coverage of original triangle */
{
    int which, a, b;

    /* Check for diagonal that crosses no boundaries */
    if (same_tree(s1->t1, s3->t1)) which = 0;
    else if (same_tree(s2->t1, s4->t1)) which = 1;

    /* If none, pick diagonal that keeps sides with most bounds together */
    else {
        a = (s1->num_bounds + s2->num_bounds) -
            (s3->num_bounds + s4->num_bounds);

        b = (s2->num_bounds + s3->num_bounds) -
            (s4->num_bounds + s1->num_bounds);

        if (a < 0) a = -a;
        if (b < 0) b = -b;

        which = (a > b ? 0 : 1);
    }

    /* Subdivide quadrilateral: compute coverages of resulting triangles */
    if (which == 0)
        subd_split_quad(pixel, s1, s2, s3, s4, (1.0 - frac1) * frac2 * coverage,
                        (1.0 - frac2) * coverage);
    else
        subd_split_quad(pixel, s2, s3, s4, s1, (1.0 - frac2) * frac1 * coverage,
                        (1.0 - frac1) * coverage);
}

```

Triangle Shading

```
/* This is called to shade the triangles formed by the three given ray trees.
 * It calls shade_trees() to do the work. */

void
shade(t1, t2, t3, pixel, coverage)
Ray_tree *t1, *t2, *t3;
int      pixel[2];
double   coverage;
{
    double weight[3];

    /* No attenuation yet */
    weight[R] = weight[G] = weight[B] = 1.0;

    shade_trees(t1, t2, t3, pixel, coverage, weight);
}
```

```

/* Computes the intensity reflected from all triangles defined by the given
 * rays, adding the results to the given pixel. */

void
shade_trees(t1, t2, t3, pixel, coverage, weight)
Ray_tree *t1, *t2, *t3;
int      pixel[2];
double   coverage,          /* Fraction of pixel covered */
weight[3];                  /* Represents attenuation by reflection
                             or refraction along ray path */
{
    double intens[4], tweight[3];

    /* Compute lighting at three intersection points and average results
     to get values for intens */
    shade_average_intens(t1, t2, t3, intens);

    /* Factor coverage and weight into color, coverage into alpha channel */
    intens[R] *= coverage * weight[R];
    intens[G] *= coverage * weight[G];
    intens[B] *= coverage * weight[B];
    intens[A] = coverage;

    /* Add intensity to pixel */
    add_intensity(pixel, intens);

    /* Recurse for reflected ray, if necessary */
    if (t1->refl != NULL) {
        tweight[R] = weight[R] * t1->inter->color[R] * t1->inter->reflectivity;
        tweight[G] = weight[G] * t1->inter->color[G] * t1->inter->reflectivity;
        tweight[B] = weight[B] * t1->inter->color[B] * t1->inter->reflectivity;

        shade_trees(t1->refl, t2->refl, t3->refl, pixel, coverage, tweight);
    }

    /* Recurse for refracted ray, if necessary */
    if (t1->tran != NULL) {
        tweight[R] = weight[R] * t1->inter->color[R] * t1->inter->transparency;
        tweight[G] = weight[G] * t1->inter->color[G] * t1->inter->transparency;
        tweight[B] = weight[B] * t1->inter->color[B] * t1->inter->transparency;

        shade_trees(t1->tran, t2->tran, t3->tran, pixel, coverage, tweight);
    }
}

```

References

- [1] Rashid Ahmad, Guanidinium Transport through the Gramicidin Channel/Guanidinium Interactions in the Gramicidin Channel (Animation) (1988).
- [2] John Amanatides, "Ray Tracing with Cones," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 129-135 (July, 1984).
- [3] A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *AFIPS Spring Joint Computer Conference*, pp. 37-45 (1968).
- [4] James Arvo, "Backward Ray Tracing," in *SIGGRAPH '86 course notes no. 12: Developments in Ray Tracing*, (August, 1986.).
- [5] Alan H. Barr, "Global and Local Deformations of Solid Primitives." *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 21-30 (July, 1984).
- [6] Alan H. Barr, "Ray Tracing Deformed Surfaces," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 287-296 (August, 1986).

- [7] Jon L. Bentley and Thomas A. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computers* **C-28**(9) pp. 643-647 (September, 1979).
- [8] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach, "Image Rendering by Adaptive Refinement," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 29-37 (August, 1986).
- [9] Sabrina Berner and Benjamin Rubin, *The Center for Information Technology (Animation)* (1986).
- [10] Gary Bishop and David M. Weimer, "Fast Phong Shading," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 103-106 (August, 1986).
- [11] James F. Blinn, "Models of Light Reflection for Computer Synthesized Pictures," *Computer Graphics (SIGGRAPH '77 Proceedings)* **11**(2) pp. 192-198 (Summer, 1977).
- [12] Phong Bui-Tuong, "Illumination for Computer-Generated Pictures," *Communications of the ACM* **18**(6) pp. 311-317 (June, 1975).
- [13] Edwin Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *Computer Graphics (SIGGRAPH '78 Proceedings)* **12**(3) pp. 6-11 (August, 1978).
- [14] Edward S. Chang, *Cusps of Gauss Mappings (Animation)* (1988).

- [15] Richard Chuang and Glenn Entis, "3-D Shaded Computer Animation — Step by Step," *IEEE Computer Graphics and Applications* **3**(9) pp. 18-25 (December, 1983).
- [16] Michael F. Cohen and Donald P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3) pp. 31-40 (July, 1985).
- [17] Robert L. Cook and Kenneth E. Torrance, "A Reflectance Model for Computer Graphics," *Computer Graphics (SIGGRAPH '81 Proceedings)* **15**(3) pp. 307-316 (August, 1981).
- [18] Robert L. Cook, "Shade Trees," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 223-231 (July, 1984).
- [19] Robert L. Cook, Thomas Porter, and Loren Carpenter, "Distributed Ray Tracing," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 137-145 (July, 1984).
- [20] Robert L. Cook, "Stochastic Sampling in Computer Graphics," in *SIGGRAPH '86 course notes no. 15: State of the Art in Image Synthesis*, (August, 1986).
- [21] Robert L. Cook, Loren Carpenter, and Edwin Catmull, "The Reyes Image Rendering Architecture," *Computer Graphics (SIGGRAPH '87 Proceedings)* **21**(4) pp. 95-102 (July, 1987).

- [22] Robert L. Cook, private communication (1988).
- [23] Frank Crow, "The Aliasing Problem in Computer-Generated Shaded Images," *Communications of the ACM* **20**(11) pp. 799-805 (November, 1977).
- [24] Franklin C. Crow, "Summed-Area Tables for Texture Mapping," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 207-212 (July, 1984).
- [25] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, and M. Howard, "Towards an Interactive High Visual Complexity Animation System," *Computer Graphics (SIGGRAPH '79 Proceedings)* **13**(2) pp. 289-299 (August, 1979).
- [26] Mark A. Z. Dippe and Erling Henry Wold, "Antialiasing Through Stochastic Sampling," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3) pp. 29-37 (July, 1985).
- [27] Masato Edahiro, Iwao Kokubo, and Takao Asano, "A New Point-Location Algorithm and Its Practical Efficiency — Comparison with Existing Algorithms," Research Memorandum RMI 83-04, University of Tokyo (1983).
- [28] Steven Feiner, David Salesin, and Thomas Banchoff, "Dial: A Diagrammatic Animation Language," *IEEE Computer Graphics and Applications* **2**(7) pp. 43-54 (September, 1982).
- [29] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications* **6**(4) pp.

- 16-26 (April, 1986).
- [30] Andrew Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications* 4(10) pp. 15-22 (October, 1984).
 - [31] Andrew Glassner, "Efficient Boolean Evaluation of CSG Models for Ray Tracing," *The Ray Tracing News* 1(1) pp. 3-7 (September, 1987).
 - [32] Andrew Glassner, "A Summary of the Ray Tracing Roundtable at Siggraph '87," *The Ray Tracing News* 1(1) p. 2 (September, 1987).
 - [33] Andrew Glassner, private communication (1988).
 - [34] Julian E. Gomez, "TWIXT: A 3D Animation System," *Computers and Graphics* 9(9) pp. 291-298 (1985).
 - [35] Jonathan A. Goldberg, "Interactive Animation and Change Description through the Action Paradigm," Masters Thesis, Brown University (May, 1987).
 - [36] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennet Battaile, "Modeling the Interaction of Light between Diffuse Surfaces," *Computer Graphics (SIGGRAPH '84 Proceedings)* 18(3) pp. 213-222 (July, 1984).
 - [37] Henri Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers* C-20(6) pp. 623-628 (June, 1971).

- [38] Daniel H. Greene and F. Frances Yao, "Finite-Resolution Computational Geometry," *IEEE Symposium on Foundations of Computer Science*, pp. 143-152 (October, 1986).
- [39] Brown Computer Graphics Group, Accidental Nuclear War (Animation, work in progress) (1988).
- [40] Brown Computer Graphics Group, Futuristic Aircraft (Animation, work in progress) (1988).
- [41] Ronald J. Hackathorn, "ANIMA II: A 3-D Color Animation System," *Computer Graphics (SIGGRAPH '77 Proceedings)* 11(2) pp. 54-64 (Summer, 1977).
- [42] Eric A. Haines and Donald P. Greenberg, "The Light Buffer: A Shadow-Testing Accelerator," *IEEE Computer Graphics and Applications* 6(9) pp. 60-16 (September, 1986).
- [43] Roy A. Hall and Donald P. Greenberg, "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications* 3(8) pp. 10-20 (November, 1983).
- [44] Pat Hanrahan and David Sturman, "Interactive Control of Parametric Models," pp. 246-259 in *SIGGRAPH '84 course notes no. 8: Introduction to Computer Animation*, (July, 1984).

- [45] Paul S. Heckbert and Pat Hanrahan, "Beam Tracing Polygonal Objects," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 119-127 (July, 1984).
- [46] John F. Hughes, "Interpolation of Cameras in Keyframe Animation," Brown University Computer Graphics Group Technical Memorandum (January, 1987).
- [47] John F. Hughes, Boy's Surface: Polyhedral and Smooth Immersions of the Projective Plane (Animation) (1988).
- [48] David S. Immel, Michael F. Cohen, and Donald P. Greenberg, "A Radiosity Method for Non-Diffuse Environments," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 133-142 (August, 1986).
- [49] James T. Kajiya, "The Rendering Equation," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 143-150 (August, 1986).
- [50] Michael R. Kaplan, "The Uses of Spatial Coherence in Ray Tracing," in *SIGGRAPH '85 course notes no. 11: State of the Art in Image Synthesis*, (July, 1985).
- [51] Douglas Scott Kay and Donald Greenberg, "Transparency for Computer Synthesized Images," *Computer Graphics (SIGGRAPH '79 Proceedings)* **13**(2) pp. 158-164 (August, 1979).

- [52] Timothy L. Kay and James T. Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 269-278 (August, 1986).
- [53] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978).
- [54] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes, "Constructive Solid Geometry for Polyhedral Objects," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4) pp. 161-170 (August, 1986).
- [55] Mark E. Lee, Richard A. Redner, and Samuel P. Uzelton, "Statistically Optimized Sampling for Distributed Ray Tracing," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3) pp. 61-67 (July, 1985).
- [56] William Lorensen, "Object-Oriented Terminology," in *SIGGRAPH '87 course notes no. 14: Object-Oriented Geometric Modeling and Rendering*, (July, 1987).
- [57] W. Lorensen, M. Barry, D. McLachlan, and B. Yamrom, "An Object-Oriented Graphics Animation System," in *SIGGRAPH '87 course notes no. 14: Object-Oriented Geometric Modeling and Rendering*, (July, 1987).
- [58] Dick Lundin, "Beyond Bhop — Motion Simulation and Other Issues," pp. 283-295 in *SIGGRAPH '84 course notes no. 8: Introduction to Computer Animation*, (July, 1984).

- [59] Nadia Magnenat-Thalmann and Daniel Thalmann, *Computer Animation*, Springer-Verlag, Tokyo (1985).
- [60] Don P. Mitchell, "Generating Antialiased Images at Low Sampling Densities," *Computer Graphics (SIGGRAPH '87 Proceedings)* **21**(4) pp. 65-72 (July, 1987).
- [61] David C. Mott, "An Improved Articulation and Object Relation Model for Scientific Visualization," Masters Thesis, Brown University (May, 1988).
- [62] Tom Nadas and Alain Fournier, "GRAPE: An Environment to Build Display Processes," *Computer Graphics (SIGGRAPH '87 Proceedings)* **21**(4) pp. 75-84 (July, 1987).
- [63] Joseph N. Pato, "Dynamic Linking/Loading for UNIX," BWE Programmer's Manual, Brown University (June, 1985).
- [64] Ken Perlin, "An Image Synthesizer," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3) pp. 287-296 (July, 1985).
- [65] Thomas Porter and Tom Duff, "Compositing Digital Images," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3) pp. 253-259 (July, 1984).
- [66] Michael Potmesil and Eric M. Hoffert, "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," *Computer Graphics (SIGGRAPH '87 Proceedings)* **21**(4) pp. 85-93 (July, 1987).

- [67] Craig W. Reynolds, "Computer Animation with Scripts and Actors," *Computer Graphics (SIGGRAPH '82 Proceedings)* **16**(3) pp. 289-296 (July, 1982).
- [68] Scott D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing* **18**(2) pp. 109-144 (February, 1982).
- [69] Ken Shoemake, "Animating Rotation with Quaternion Curves," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3) pp. 245-254 (July, 1985).
- [70] Garland Stern, "Bbop - A System for 3D-D Keyframe Figure Animation," pp. 240-243 in *SIGGRAPH '83 course notes no. 7: Introduction to Computer Animation*, (July, 1983).
- [71] Paul S. Strauss, "Software Standards for the Brown University Computer Science Department," Brown University Computer Graphics Group Technical Memorandum (June, 1985).
- [72] Paul S. Strauss, "A Tutorial Guide to the SCEFO Language," Brown University Computer Graphics Group Technical Memorandum (April, 1987).
- [73] Paul S. Strauss, "The VAL Package," Brown University Computer Graphics Group Technical Memorandum (September, 1987).
- [74] Paul S. Strauss, "A SCENE Design Overview," Brown University Computer Graphics Group Technical Memorandum (September, 1987).

- [75] Paul S. Strauss, "A Guide to LASH for SCEFO Users and BAGS Programmers," Brown University Computer Graphics Group Technical Memorandum (July, 1987).
- [76] David Sturman, "Interactive Keyframe Animation of 3-D Articulated Models," *Proceedings of Graphics Interface*, pp. 35-40 (1984).
- [77] Symbolics, Inc., *S-Dynamics*, Symbolics, Inc. (1985).
- [78] Symbolics, Inc., *S-Geometry*, Symbolics, Inc. (1985).
- [79] K. E. Torrance and E. M. Sparrow, "Theory for Off-Specular Reflection from Roughened Surfaces," *Journal of the Optical Society of America* **57**(9) pp. 1105-1114 (September, 1967).
- [80] W. Benjamin Trumbore, "A Tutorial Guide to the PMAP Pattern Mapping Description Language," Brown University Computer Graphics Group Technical Memorandum (July, 1986).
- [81] Jeffrey J. Vroom, "ELROY: Elegant Rendering, Oh Yeah," Brown University Computer Graphics Group Technical Memorandum (June, 1986).
- [82] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," *Computer Graphics (SIGGRAPH '87 Proceedings)* **21**(4) pp. 311-320 (July, 1987).

- [83] Turner Whitted, "An Improved Illumination Model for Shaded Display,"
Communications of the ACM **23**(6) pp. 343-349 (June, 1980).
- [84] Turner Whitted and David M. Weimer, "A Software Testbed for the
Development of 3D Raster Graphics Systems," *ACM Transactions on
Graphics* **1**(1) pp. 43-58 (January, 1982).