

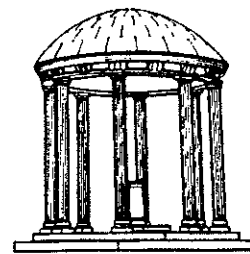
Algorithms for Efficient Image Synthesis

TR90-031

August, 1990

Andrew Stephen Glassner

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

Algorithms for Efficient Image Synthesis

by

Andrew Stephen Glassner

A Dissertation submitted to the faculty of The
University of North Carolina at Chapel Hill
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the
Department of Computer Science

Chapel Hill

July 1988

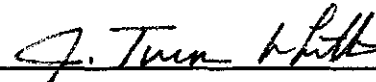
Approved by:



Adviser (Kenan Professor Frederick P. Brooks, Jr.)



Reader (Gil Professor Henry Fuchs)



Reader (Professor J. Turner Whitted)

Original material © 1988
Andrew Stephen Glassner
ALL RIGHTS RESERVED

ANDREW STEPHEN GLASSNER
Algorithms for Efficient Image Synthesis
(Under the direction of F. P. BROOKS, JR.)

Abstract

This dissertation embodies six individual papers, each directed towards the efficient synthesis of realistic, three-dimensional images and animations. The papers form four major categories: ray tracing, animation, texture mapping, and fast iterative rendering.

The ray tracing papers present algorithms for efficiently rendering static and animated scenes. I show that it is possible to make use of coherence in both object space and time to quickly find the first intersected object on a ray's path. The result is shorter rendering times with no loss of quality.

The first animation paper considers the needs of a modern animation system and suggests a particular object-oriented architecture. The other animation paper presents an efficient and numerically stable technique for transforming an arbitrary modeling matrix into a fixed sequence of parametric transformations which yield the same matrix when composed. The result is that hierarchical, articulated models may be described by the human modeler or animator with any convenient sequence of transformations at each node, and the animation system will still be able to perform parametrically smooth motion interpolation.

The fast rendering paper describes a system built to allow quick modification of object surface description and lighting. I use a space/time tradeoff to capitalize on the constant geometry in a scene undergoing adjustment. The result is a system that allows relatively fast, iterative modification of the appearance of an image.

The texture mapping paper offers a refinement to the sum table technique. I show that the fixed, rectangular filter footprint used by sum tables can lead to oversampling artifacts. I offer a method which detects when oversampling is likely to occur, and another method for iteratively refining the texture estimate until it satisfies an error bound based on the oversampled area.

Together, these six papers represent a collection of algorithms designed to enhance synthetic images and animations and reduce the time required for their creation.

Acknowledgements

I give me pleasure to thank my advisor and chairman, Dr. Frederick P. Brooks, Jr., for helping me make this dissertation a reality. Dr. Brooks applied a sure touch at critical moments in the development of much of this dissertation, and I hope that some of his clear, direct style has influenced my own.

I also owe thanks to the other members of my committee. Dr. Steven Pizer helped me understand some of the implications of matrix decomposition. Dr. Steven Weiss offered steady encouragement and help when preparing this document for the graduate school. Dr. Henry Fuchs was always enthusiastically supportive of new ideas, and helped keep my imagination stirring. Dr. Turner Whitted taught me through words and example the precision and depth of thought that characterize a dissertation, and then made sure my work reached those standards.

The papers in this thesis were written mostly as a result of independent research while I studied at UNC-Chapel Hill. I never would have had the freedom to pursue this work without the help and support of two research directors, who tolerated and even encouraged many of my offbeat ideas over the years. For this and more, I thank Dr. Henry Fuchs and Dr. Frederick P. Brooks, Jr.

I would like to thank Lakshmi Dasari, who put up with my keeping a lot of strange hours when several of these projects were hatching. Lastly, I want to thank my parents. I owe them much, for they gave me a love of knowledge and creativity, and the strength to pursue both.

Table of Contents

Introduction	1
Space Subdivision for Fast Ray Tracing.....	8
Adaptive Precision in Texture Mapping.....	17
Supporting Animation in Rendering Systems.....	28
Template Parameterization for 3d Pose Interpolation.....	35
Late Binding Images	50
Spacetime Ray Tracing for Animation.....	74
Summary	86

Introduction

This dissertation contains six papers written over the course of five years. Each paper addresses a topic within the field of computer graphics; the emphasis is on efficient realistic image synthesis and animation. My goal has been to develop algorithms to enhance the use of computers to construct realistic images and animations of three-dimensional scenes.

In this introductory chapter I will discuss the problems addressed by each paper. To place the papers historically, I will consider the central issue of each paper and review the state of the art relevant to that issue at the time the research was begun. Successive chapters present the papers. A critique and discussion of each paper appears as the last chapter.

Space Subdivision for Fast Ray Tracing

IEEE Computer Graphics & Applications, vol. 10, no. 4, October 1984

The central issue addressed by this paper is the reduction of rendering time of a ray-traced image. The paper implicitly assumes that the database may be described by a collection of objects frozen in time.

The emergence of ray tracing as a popular and powerful tool for image synthesis began with [Whitted80]. That paper presented a recursive scheme for estimating the light incident upon and emitted from various surfaces.

The primary expense in the algorithm as presented there is determining which object is the first along a particular ray's path. An exhaustive approach intersects every ray with every object, and then searches among all intersected objects for the intersection nearest to the ray origin. This approach can quickly grow very expensive. Consider a situation where every intersection spawns s shadow rays (one to each of s light sources), one reflection ray, and one transparency ray. If we follow these rays for g generations, then r , the total number of rays traced per screen sample will be $r = 1 + \sum_{i=1}^g (s+2) 2^{i-1}$. If we

test each of these r rays against b objects, then we will have to compute rb intersections between rays and objects for every screen sample. As the expression for rb shows, there are many ray-object intersections in the exhaustive approach.

The biggest objection one may raise to this technique is that many of the ray-object intersections are “clearly” superfluous - if an object of small spatial extent is far from the path of the ray it should never be tested. There is no check in the exhaustive algorithm for this sort of optimization.

In December 1983, when this work was begun, existing approaches to ray tracing acceleration used nested hierarchies of bounding volumes [Rubin&Whitted80]. My perception in 1983 was that ray tracing was considered a laboratory oddity – the power of ray tracing to produce sophisticated images was accepted, but many people found it prohibitively expensive for commercial production or other routine rendering. I felt that in order to bring ray tracing into the realm of everyday rendering techniques, we needed a fast, automatic algorithm to find the first intersection between a ray and the objects in the database. I found that a spatial data structure provided such a scheme, and reported the results in *Space Subdivision for Fast Ray Tracing*, one of the first papers to explicitly describe a software algorithm to accelerate ray tracing.

The paper proposes building an auxiliary octree data structure on the space of the objects being rendered. Rays are traced by following their progress through this data structure; only objects in cells entered by the ray are examined for intersection. Typically many ray-object intersections may be avoided in this way. As a result, the total cost for generating an image is reduced to a fraction of the exhaustive approach.

Adaptive Precision in Texture Mapping

Computer Graphics, vol. 20, no. 4, Proc. Siggraph '86, August 1986

The central issue addressed by this paper is how to efficiently render a texture-mapped image using the sum-table technique, while avoiding the oversampling artifacts to which sum tables are prone.

When this work was begun in February 1986, the two most popular mechanisms for efficient texture mapping were the *mip-map* [Williams83] and the *sum table* [Crow84]. Both algorithms estimated the texture on a surface seen by a pixel by using a space-invariant box filter derived from the texture-space image of the pixel.

A mip-map is a pyramid of images of decreasing size. Each level in the pyramid contains a low-pass filtered version of the level below, saved at some lower resolution than the lower level. The bottom-most level is the original texture image; the top-most level is a single sample representing an "average" value for the entire texture. In practice, texture sampling is limited to a square region, derived from the texture-space image of the pixel by a heuristic rule [Heckbert86].

A sum table is a single table, equal in width and height to the original texture, but typically many bits deeper. In a typical sum table, each element contains the sum of all original texture samples below and to the left of that element. Four table accesses, three additions and a division can yield the box-filtered average value of the texture within any rectangle. In practice, the rectangle is usually chosen to be the smallest oriented rectangle enclosing the texture-space image of the pixel.

Because the sum-table technique can average rectangles while mip-maps are limited to squares, I felt that sum tables were the best efficient texturing method available when I began work on a new rendering system in early 1986. After generating some images I noticed that although my texturing was acceptable, there were distinct artifacts in regions of high complexity. In particular, the textures became blurry more quickly than I expected.

I studied the problem and realized that sum tables as used at the time involved two important assumptions: a box filter was a good filter for texture sampling, and the bounding rectangle of a pixel's texture-space image was a good approximation of that image. I felt that although a box filter was not ideal, it was acceptable for most work, and probably not the source of the artifacts I saw. But the second assumption would lead to oversampling of the texture function, which could well result in blurry textures. I decided to try to reduce or eliminate this oversampling.

In the paper I examine the errors introduced by estimating the texture within the bounding box of the pixel's texture space image (instead of the image itself). I show that this approximation can introduce measurable error when estimating the average texture within the pixel, and thereby cause visible artifacts. To alleviate the problem, I proposed using another table to hold the local variance of the texture. When variance is low, the bounding-box approximation is accepted. But when texture variance is high, an iterative, recursive algorithm is called to refine the region of texture space sampled for that pixel.

The essence of the idea is that standard sum tables can provide the average value within any oriented rectangle in texture space. To find the average value in a more general region, I find the average within the region's bounding rectangle and then remove smaller rectangles that lie outside the region but within the bound.

Results show that the algorithm indeed works harder than sum tables only where the texture is complex, that the extra effort is proportional to the local complexity of the texture, and the images generated by this method are superior to those produced with standard sum table techniques.

Supporting Animation in Rendering Systems

CHI+GI Workshop on Rendering Algorithms & Systems

Toronto, April 1987

This paper shows how to build an animation system that can support interactive animation design, high-quality rendering, and a database of complex objects.

I present an object-oriented architecture that meets the criteria. The animation database is controlled by a manager that is responsive to a small set of messages, which specify the parameters that describe that object's characteristics and motion. Animation specification and rendering are each accomplished by communicating with the animation manager.

I also propose a technique for implementing object-oriented importance sampling, particularly well-suited to the rendering architecture described in the paper.

Template Parameterization for 3d Pose Interpolation

A popular and common feature of many animation systems allows a user to begin the process of character animation by designing an articulated model described by a hierarchical tree structure. Keyframes may be explicitly or implicitly built by specifying a set of transformation parameters at each node of the tree. Interpolation of keyframes is then reduced to interpolation of the parameters.

In this article I show that two conditions must be met for this approach to succeed: the trees at the two keytimes must have the same topology, and the transformations within corresponding nodes must contain the same transformations in the same order. The first condition is easy to satisfy; the latter is not. When I started to build a new animation system in June 1987, I knew of no existing systems that could reliably convert an arbitrary sequence of parametric modelling transformations into a fixed sequence (or *template*) whose parameters could then be interpolated. The result of this limitation was that

modelers were prohibited from using skew or differential scaling in their original model descriptions. I didn't want to be bound by these restrictions, particularly in a system of my own creation, so I set out to find a way to convert arbitrary transformation specifications into a fixed template.

In the paper I propose the use of the Singular Value Decomposition algorithm to decompose a 4-by-4 modelling matrix into a sequence of mirror, rotation, scale, and translation matrices. The resulting sequence of matrices may then be parametrically interpolated to produce in-between poses. Since SVD can decompose any modeling matrix, the model designer is free to use any sequence of any transformations when preparing model descriptions for animation.

Late Binding Images

Submitted to *IEEE Computer Graphics & Applications*

Many images require hand-tuning of light sources and surface parameters to meet technical and aesthetic demands. Light source color and placement, object color, transparency, and reflection co-efficients interact visually in complex ways; changing one feature often necessitates changes to the others.

In January 1987 the medical applications group felt a need for a new rendering system. I realized that in the course of preparing a medical study, many of our images needed to be iteratively adjusted by the user, to find the proper balance of shading, transparency, and lighting. At the time, each change to these parameters required running the entire rendering system again, even though the viewing direction remained unchanged. I felt that we could save time by performing the expensive, view-dependent scan-conversion step once and saving the result in a file. Starting with that file, we could then iteratively adjust the lighting and surface properties of the rendered objects until the image was acceptable.

By 1987 several techniques had been published for separating the steps of scan conversion and surface shading [Bass81], [Whitted81]. This allowed users to quickly change the appearance of objects, while paying only once for the relatively high cost of scan conversion.

In this paper I describe such a rendering system which I built for medical applications at UNC-CH. The main points of the paper are a discussion of the implementation issues that arose when planning and programming the system, and a group algebra which unifies the system and provides a concise representation of its actions.

Spacetime Ray Tracing for Animation

IEEE Computer Graphics & Applications, vol. 8, no. 3, March 1988

This work was started in February of 1987. I wanted to produce an animated film, and I wanted to use ray tracing for rendering. At that time, the only algorithms for speeding up ray tracing were directed to the generation of single, static images. I could have applied such methods individually to each frame, but I wanted to include motion blur in my film. The only ways I could see to use existing techniques for motion-blurred animation seemed inelegant and difficult.

I suspected that there was a clean algorithm for efficiently producing a piece of animation with ray tracing. I also wanted to capitalize on our knowledge of the motion path of each object to further speed up the rendering process. So I investigated algorithms that would use time coherence and motion path information to accelerate ray tracing for animation.

The paper begins with an assessment of the two ray tracing acceleration techniques most popular at the time: bounding volumes and space subdivision. The analysis showed that the strengths and weaknesses of these two approaches were complementary. This suggested a hybrid algorithm, which combined features of both techniques. The result was a nested hierarchy of disjoint bounding volumes, whose density followed that of the database (any consistent measure of density could be used).

Extended into a four-dimensional spacetime, this hybrid algorithm provided a framework in which individual rays of an animation could capitalize on the known motion path of each object in the database. An efficient bounding volume structure was built in spacetime that contained each object over the course of the animation. Use of this four-dimensional information was shown to reduce several important statistics in the ray tracing algorithm for rendering a complete animation, including the total number of ray/object intersections required.

References

- [Bass81] Bass, Daniel H., "Using the Video Lookup Table for Reflectivity Calculations: Specific Techniques and Graphic Results", *Computer Graphics and Image Processing*, vol. 17, no. 3, Nov. 1981, 249-261

[Crow84] Crow, F., "Summed-Area Tables for Texture Mapping", *Computer Graphics*, vol. 18, no. 3, July 1984

[Heckbert86] Heckbert, Paul, "A Survey of Texture Mapping", *IEEE Computer Graphics and Applications*, vol. 12, no. 11, November 1986

[Rubin&Whitted80] Rubin, S., and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics*, vol. 14, no. 3, Proceedings of Siggraph '80, July 1980

[Williams83] Williams L., "Pyramidal Parametrics", *Computer Graphics*, vol. 18, no. 3, July 1984

[Whitted80] Whitted, T., "An Improved Illumination Model for Shaded Display," *Comm. ACM*, vol. 23, no. 6, June 1980

[Whitted81] Whitted, Turner, and David M. Weimer, "A Software Test-Bed for the Development of 3-D Raster Graphics Systems", *Computer Graphics* vol. 15, no. 3, Proceedings of Siggraph '81

Space Subdivision for Fast Ray Tracing

IEEE Computer Graphics & Applications
vol. 4, no. 10, October 1984

© 1984 IEEE. Reprinted with permission, from
IEEE Computer Graphics and Applications,
Vol. 4, No. 10, pp. 15-22, October 1984.

Speed up ray-tracing techniques by reducing the number of time-consuming object-ray intersection calculations that have to be made. You'll be able to handle large databases considerably faster.

Space Subdivision for Fast Ray Tracing

Andrew S. Glassner

University of North Carolina at Chapel Hill

The most powerful general image synthesis method used today is referred to generically as ray tracing. Ray tracing was first described by Appel¹ and later by Bouknight and Kelley² and Kay.³ The algorithm used by most ray-tracing programs is described by Whitted.⁴ This paradigm is attractive because of its very elegant implementation and the wide range of natural phenomena it models.

Although ray tracing as it stands is not the final word in image synthesis, it is probably the most realistic technique we have today. This realism is further enhanced by the technique of distributed ray tracing described by Cook, Porter, and Carpenter.⁵ Unfortunately, ray tracing is also very slow. Ray-tracing algorithms are famous for the large amounts of computer time they consume to create even one picture of moderate complexity. It is this slowness that prevents more people from using the powerful ray-tracing methods.

Previous work in speeding up the picture-generation process has concentrated on screen-space solutions and hardware solutions. Roth⁶ has described a method for examining a rough rendering of a scene and investigating those areas of the screen where additional work seems to be necessary. Ullner⁷ describes hardware solutions that consist of multiple microprocessors in various configurations, with each processor handling a subset of either rays or objects. Both of these approaches use the basic ray-tracing algorithm as described by Whitted and attempt to draw pictures faster by either running the algorithm in parallel or running it less often for a complete picture.

A different approach toward speeding up the process is explored in this article: we decrease the time required by the algorithm to render a given pixel. To do this, we first

need to determine what are the most time-consuming portions of the algorithm.

Whitted reports that ray-object intersections can require over 95 percent of the total picture-generation time. A synopsis of the ray-tracing technique with a qualitative breakdown of where time is spent is also given in Glassner.⁸ Kajiya⁹ has shown, with a simple skeleton of the ray-tracing process, that these intersections comprise an "inner loop" of the algorithm. He demonstrates that each ray must be checked against each object in the scene so that the number of intersection calculations is linear with respect to the product of the number of rays traced and the number of objects in the entire picture. Doubling the number of objects in a scene (about) doubles the rendering time; doubling both the objects and the rays takes four times longer to render the image.

Recent work has concentrated on the ray-object intersection problem for various classes of objects (Kajiya^{10,11} and Hanrahan¹²). These algorithms show that the intersection operation can require any amount of floating-point operations—from just a few to many thousands.

If we want to reduce the time spent on ray-object intersections, we have at least two choices. We can speed up the intersection process itself, possibly with specialized hardware. Alternately, we can reduce the number of ray-object intersections that must be made to fully trace a given ray; this is the approach followed in this article.

Overview of the new algorithm

The new algorithm is based on a simple observation. To make this observation, let us divide the space in a three-dimensional scene into small compartments, keeping a list of all the objects that reside in each of these compart-

ments. We can then speed up the ray-tracing process in the following way.

Start a ray and determine in which compartment it originated. Follow the ray and compare it against only the objects it hits in that compartment. If one or more objects in the compartment are pierced, find the closest pierced object and return its color as the value of the ray. We are then finished tracing that ray, for we have found the first object the ray hit. If the ray does not hit an object in this compartment, project the ray into the next compartment and repeat the process.

If each compartment contains a small number of objects, we can process that compartment quickly. If we're lucky and find right away that the ray has hit an object, we have only a small number of object intersections to process. If we're very unlucky and find the ray has hit nothing until we hit the world sphere (Kajiya⁸), we are still better off because we probably have checked fewer objects than there are in the entire scene. Therefore, unless the overhead of getting from compartment to compartment is very high, we will always save time relative to intersecting every object in the entire database.

Fortunately, a very good scheme for breaking up space into such compartments is available. This octree technique is described extensively in Jackins and Tanimoto¹³ and Meagher.¹⁴ An octree structure allows us to dynamically subdivide space into cubes of decreasing volume until each cube (called a voxel) contains less than a maximum number of objects. Octrees are normally used to define the shapes of objects that are difficult to model with primitive surfaces. In that context, each cell of the tree is either occupied by that object, or it is empty. Each occupied cube may contain some information about color, density, or some other attribute of the object, but the cube itself is considered to be either fully filled by the object or empty of it.

Here, we use each cell of the octree to hold a list, not a piece of an object. The list describes all the objects in the scene that have a piece of their surface in that cell.

Usually when we synthesize images we are interested only in the surfaces of the objects in our scene. The assumption is that the inside of a transparent or translucent object is either empty or else described by other, independently defined objects. For example, when we test a ray against a sphere, we care only about those points on

the sphere where the ray pierces the sphere's surface. It's unimportant to know if a given point on the ray is inside or outside the sphere. Thus, for this algorithm, we subdivide space into an octree, associating a given voxel with only those objects whose surfaces pass through the volume of the voxel. See Figure 1.

The next two sections of this article present the techniques central to the new algorithm. The first section describes the process of building and maintaining the octree and a technique for obtaining fast access to any node. The second section describes the mechanism for finding the next node intersected by a ray when it has hit nothing in the current node.

Octree building and storage

The arguments for using octrees as the spatial compartments mentioned above are that octrees are well studied and understood and that they allow dynamic spatial resolution. Volumes with high object complexity can be recursively subdivided into smaller and smaller volumes, generating new nodes in the tree for only these new volumes.

When a ray fails to hit any objects in a given node, it must move on to another node in space. As we will see in the next section, the algorithm works by finding a point guaranteed to be in the next node encountered by the ray and then determining the particular node containing that point. In this section we address the process of finding the node.

A very economical octree storage technique has been described by Gargantini.¹⁵ We use a slight variation here to speed up the time required to find a given node.

The parent node (which just encloses the world sphere) is labeled node 1. When we subdivide a node, it passes its name as a prefix to all its children, which are numbered 1 through 8, as shown in Figure 2. Thus, the eight children of the parent node are nodes 11 through 18. The children of node 13 are nodes 131 through 138, and so on. Now we need a way to address a node of a given name.

If we subdivide the parent node twice, we find the largest node name possible is 188. Clearly, we don't want to allocate 188 nodes when we start the program; for example, we might find that nodes 131-138 never need to be created. The dynamic resolution of the octree scheme suggests a dynamic allocation of memory, creating a new node only when we need it. But then we return to the problem of finding a given node. If we just ask the operating system for a chunk of memory to be used when it's time to create (say) node 173, then how do we find node 173 later on?

There are two extremes in the continuum of answers to this question. At one extreme we could create a table with an entry for every possible node name that contains that node's address. This possibility would require vast amounts of storage (more than for a straightforward eight-pointers-at-a-node scheme!), but it would also have the advantage of extreme speed in finding the address of a node with a given name. At the other extreme, we could create a large linked list of all the nodes in the octree, which we would have to scan from the beginning each time

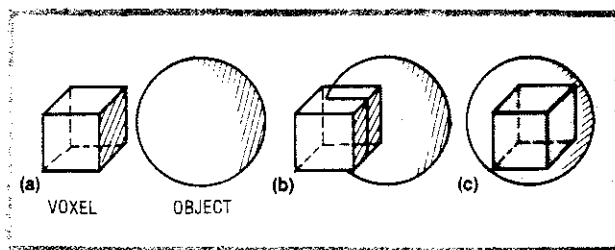


Figure 1. Space subdivision. An object is considered associated with a voxel if and only if some of the object's surface exists within the voxel. (a) shows an object not associated with the voxel; (b) shows an object that is associated with the voxel; and (c) illustrates the voxel within an object (the object is not associated with the voxel).

we want to locate a given node. This provision would have the advantage of requiring very little memory beyond that needed for the nodes themselves, but it would slow the operation to search the list each time we must look for a particular node.

An attractive compromise is to mix the two ideas using a hashing scheme. We can hash the name of a node into some small number and then follow a linked list of all nodes that hash into that number starting at a given point in a table. By changing the size of the table, we can pick any point in the continuum described above. Thus we trade speed for memory consumption and vice versa. A very simple hashing function, which merely returns the node name modulo the table size, seems to work fine.

Here we see the difference between the original numbering scheme proposed by Gargantini and the one used here. Gargantini suggested numbering the nodes 0 through 7, which had the advantage of assigning an octal number to each node. However, consider the case of subdividing node 0: one of the nodes created would have the name 00. To a computer, the number 00 is the same as the number 0, and we would have no way of differentiating the two. Similarly, 005 would be the same as 05, and so on. A solution to this problem would be to keep the name of each node as a character string. This would keep node 0 different from 000, but the string representation is bulkier than an integer, as well as slower in comparing it against another of its own type.

The modification presented here is to number the children from 1 to 8. Numbering the nodes this way loses the octal purity of the original scheme, but it allows us to name the nodes with numbers instead of character strings. Thus, node 1 could never be confused with node 111, and similarly node 15 is distinct from node 1115.

We can then find the name of a node containing a point (x,y,z) with the scheme presented in Figure 3.

Once we have a node name, we must search through the appropriate linked list for its entry and associated object list. Clearly the fewer nodes there are to be searched through, the faster (on the average) we will find the node we're looking for. We can use another observation to reduce the number of nodes stored as entries in the table/linked list structure by a factor of eight.

Each time we subdivide a node (because it contains too many objects, or more precisely, too many surfaces), we create all eight children at once. When we want to allocate memory for these eight children, we can ask the memory allocator for one large block of memory big enough to hold all eight nodes. We then use the first eighth for the first child node, the second eighth for the second child node, and so on. Now we need to store only the first child in the hash table/linked list structure. The other children are easily found by adding the right number of node lengths to the first node's address; i.e., add one node length for node 2, add two node lengths for node 3, and so on. This scheme is illustrated in Figure 4.

As we subdivide nodes, we keep a record of the smallest node created anywhere in space. This record can just be the length of the side of the smallest node; we will see why we want this information when we look at the algorithm for moving the ray from voxel to voxel.

Let's now look at the structure of an octree node. It consists of four members: a name, a subdivision flag, center and size data, and an object-list pointer. The name is an integer that is the name of this node. The subdivision flag is set if this node has been subdivided. The center and size information may be omitted to conserve memory space and derived on the fly from just the node name (this is another time-space trade-off). The object-list pointer points to the start of a list of integers in a dynamically

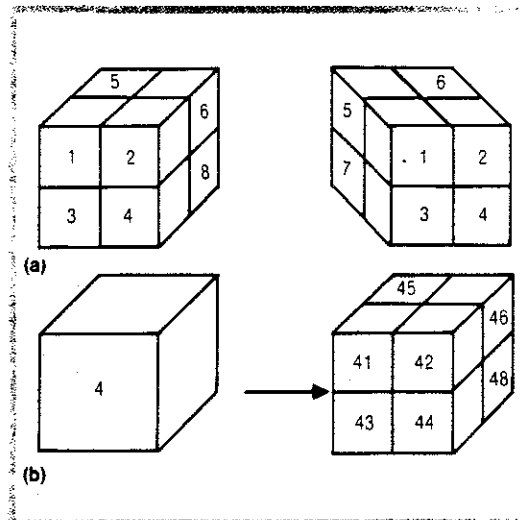


Figure 2. Space subdivision. (a) Subnodes are labeled 1-8; (b) a parent voxel passes its name as a prefix to its children.

LISTING 1

```

findnode(x,y,z) {
  node = 1;
  WHILE ( node_subdivided is TRUE ) {
    IF ( x > node_center_x )
      IF ( y > node_center_y )
        IF ( z > node_center_z )
          node = (node*10) + 6;
        ELSE node = (node*10) + 2;
      ELSE
        IF ( z > node_center_z )
          node = (node*10) + 8;
        ELSE node = (node*10) + 4;
    ELSE
      IF ( y > node_center_y )
        IF ( z > node_center_z )
          node = (node*10) + 5;
        ELSE node = (node*10) + 1;
      ELSE
        IF ( z > node_center_z )
          node = (node*10) + 7;
        ELSE node = (node*10) + 3;
  }
  RETURN ( node )
}

```

Figure 3. Node-finding scheme.

allocated array. The integer indicated by this pointer is the number of the first object in this node. Subsequent integers continue to represent other objects, until some illegal object number (say -1) is encountered, signalling the end of the object list for this node.

Now we know how to generate the octree so we can easily and quickly find a node of interest knowing only a point in the node. Let's now look at the process of deciding whether or not to subdivide a given node as we build the tree.

What we're interested in doing now is looking at the list of objects that have surfaces that pass through the parent node of the node under consideration. We will include each of these objects in the list of objects for this child if its surface also passes through the child's volume. When we have done this for each child, we can consider how many objects are contained in each child. If any child has too many objects (and we have room to create new nodes), we may then subdivide each overfull node recursively.

The algorithm used to determine whether an object's surface passes through a voxel treats convex objects (particularly spheres) with more efficiency than arbitrary objects.

In general, we intersect the object with each of the six planes that bound the voxel. Should any of these points of intersection lie within the square region of the plane that is the side of the voxel, the object is kept. Otherwise, some point within the object must be examined. If that interior

point is within the voxel, the object is kept; otherwise, it is discarded. A very efficient formulation of this algorithm for the special case of polygons is found in Sutherland and Hodgman.¹⁶

Movement to the next voxel

Two important facts guide us in designing the algorithm to get to the next voxel. First, because the space is dynamically resolved when we build the octree, we don't know how large (or small) any voxel in space is with the exception of the current one. The second fact is that the movement operation must be accomplished as fast as possible. Certainly, the movement must be minimally fast enough that we don't lose the time we save by cutting down ray-object intersections by giving that time to voxel-movement operations.

The general idea behind the voxel-movement algorithm is to find a point that is guaranteed to be in the next voxel, whatever its size. This point is then used to derive a voxel name (and its associated size and object list) according to the schemes presented in the previous section.

In the following, the term current node refers to the node that has yielded no intersections; it is the node we are leaving for greener pastures. We will refer to points on the ray being traced with the parameter t . The value of t increases as we move away from the origin, where t has the value 0.

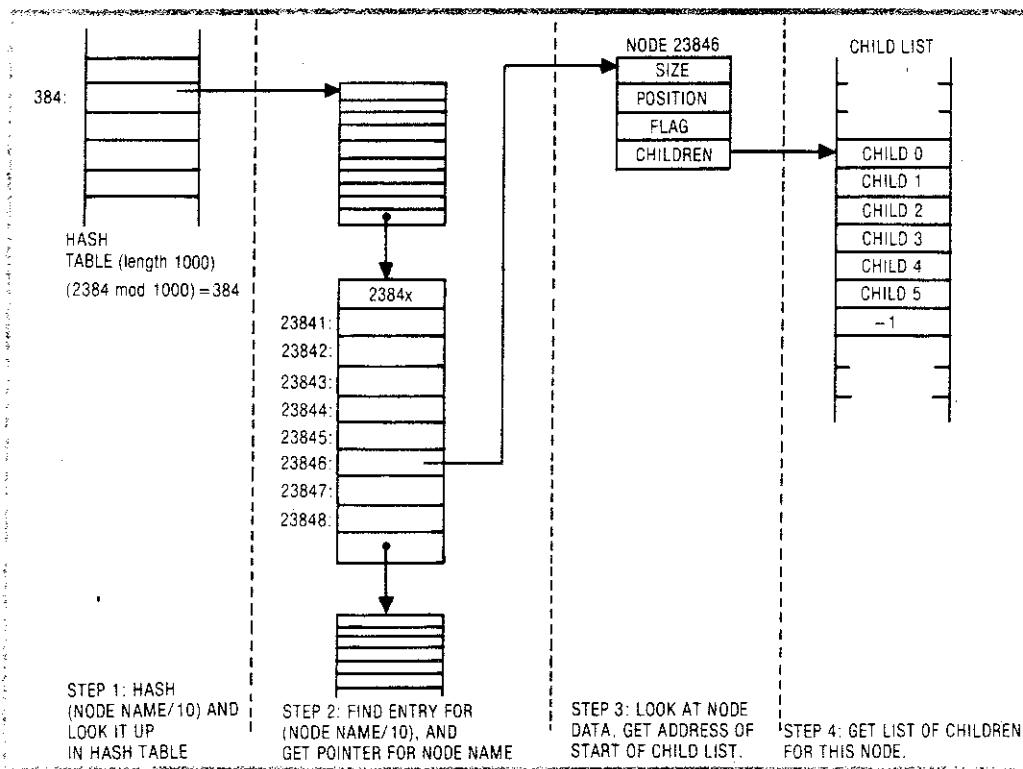


Figure 4. Sample hash table/linked list. Here, we want information for node 23846.

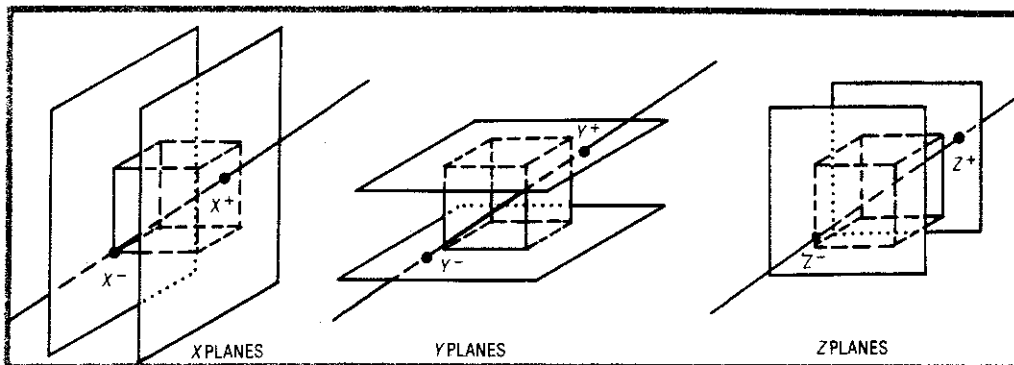


Figure 5. To find the endpoints of the ray segment within a voxel we first intersect that ray with the six bounding planes of the voxel, noting the ray's t value at each intersection.

We know that the voxel we want to examine next will contain points on the ray with t values greater than the ray may attain in the current node. Thus the first step is to find the largest value of t the ray may assume while still in the current node.

Let's designate this value of t as $t+$. We can find $t+$ by intersecting the ray with the six planes that bound the current voxel.* Two of these intersections give us bounds on t parallel to the x axis, two others parallel to the y axis, and the remaining two parallel to the z axis. We can find t values for all six of these points as shown in Figure 5. Each plane is parallel to two of the three coordinate axes, a fact that simplifies its plane equation considerably. It is inexpensive to intersect a ray with one of these "simple" planes because it costs only one subtraction and one divide operation per plane. Note that the points describing the intersections of the ray with the planes of the voxel may lie far outside the volume of the voxel itself. But certainly some values of t will hold for all three ranges: these are exactly the values of t inside the voxel. The intersection of the three ranges of t yields those values of t that the ray may assume while it is inside the box. The value of $t+$ is the value of the upper end of this range of t values, as illustrated in Figure 6.

The resolution of space in the next voxel to be encountered cannot be any finer than the finest resolution we reached when we built the octree. Now we see the reason we kept a record of the minimum-sized voxel when we built the tree. Let us call the length of the side of this smallest voxel $Minlen$.

Figure 7 illustrates that we can find the next voxel by merely moving perpendicularly to the face of the voxel that contains $t+$. If $t+$ is on an edge, we must travel perpendicularly to both faces sharing the edge, and similarly we must travel in three directions if $t+$ is on a corner of the voxel. These movement operations are trivial to compute and perform because each is perpendicular to a coordinate axis. We are guaranteed not to move outside the next voxel if we limit our movement to less than the

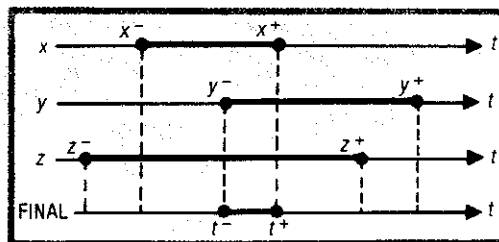


Figure 6. To find the values of t that a ray may assume within a voxel, we find the intersection of the three ranges determined in Figure 5.

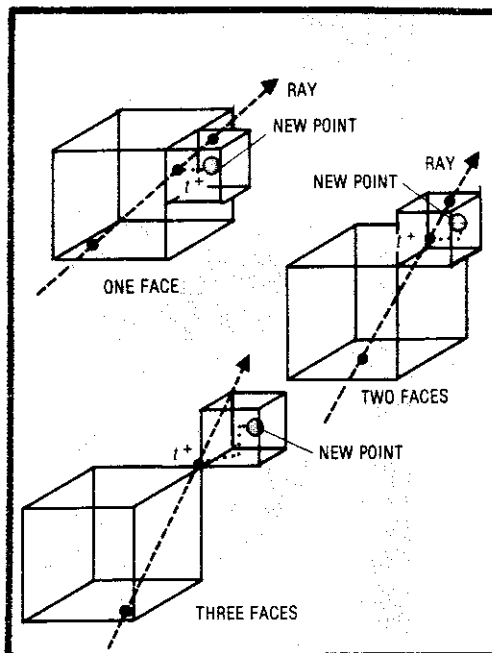


Figure 7. To find the next voxel in a ray's path, we find a point guaranteed to be in that voxel. We find that point by moving the distance $Minlen/2$ perpendicular to each face in which $t+$ lies.

* It is sufficient to intersect the ray with only four planes, but I suggest the additional code necessary to determine which four outweigh the advantages of eliminating two intersections.

Table 1. Timing statistics for old and new ray-tracing algorithms.

	CHECKER-BOARDS AND BALLS	SPIRALS	RECURSIVE PYRAMID	GEODESIC CUBE	SINC FUNCTION	GREAT CIRCLES
NUMBER OF OBJECTS	53	401	1.025	1.536	3.656	7.681
NUMBER OF RAYS TRACED	884.413	532.036	352.322	597.245	448.177	466.524
CHILDREN PER VOXEL	8	20	30	8	30	25
NUMBER OF VOXELS	105	169	473	2.889	2.897	3.009
OLD NUMBER OF INTERSECTIONS	46.830.111	212.713.658	320.825.000	915.439.104	1.636.286.600 (ESTIMATE)	3.553.061.000 (ESTIMATE)
NEW NUMBER OF INTERSECTIONS	6.149.864	13.789.597	9.008.077	9.848.255	10.615.831	17.298.843
AVERAGE INTERSECTIONS PER RAY	6.9	25.9	25.6	16.5	23.7	37.1
OLD TIME (HR:MIN)	8.22	8.53	17:41	42:12	≈ 63:00 (ESTIMATE)	≈ 141:00 (ESTIMATE)
OCTREE BUILD TIME (HR:MIN)	0:04	0:02	0:02	0:08	0:12	0:21
NEW TIME (HR:MIN)	2.23	0:40	2:25	2:49	3:22	4:44

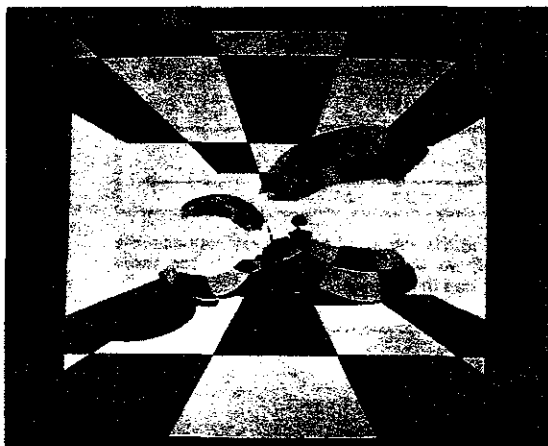


Figure 8. Two perfectly reflecting, intersecting spheres sit between a pair of checkerboards.

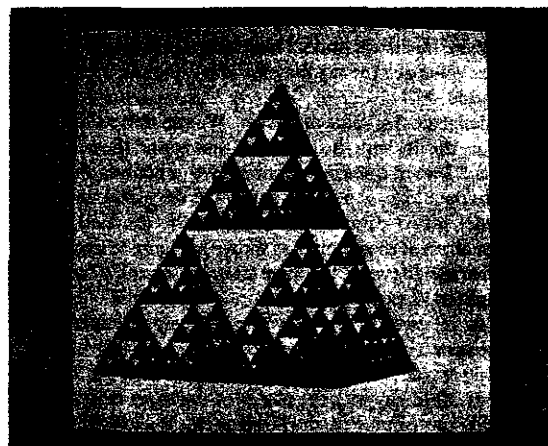


Figure 10. A procedurally generated model, similar to a kite designed by Alexander Graham Bell.



Figure 9. Two interweaving spirals of spheres. Note the shadows on the distant balls.

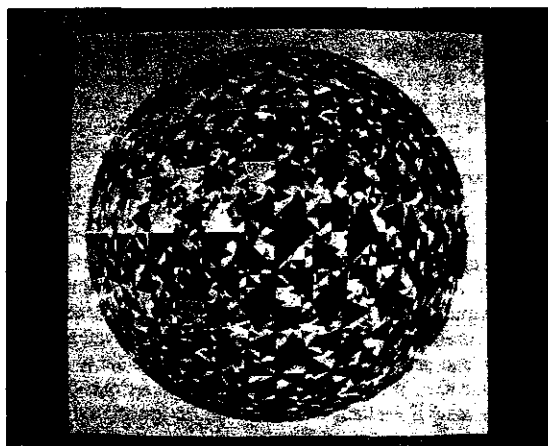


Figure 11. Two different (4, 4, 3), tilings of a geodesically projected cube share the surface of a sphere.

minimum length of the side of that box. That minimum length is Minlen , as stored when we built the octree.

Thus, if we travel some fraction of Minlen (say $\text{Minlen}/2$) from $t+$ in each necessary direction, perpendicularly to the faces of the current voxel, we have a point within the next voxel encountered by the ray.

As long as we know that the point we finally end up with is within this smallest possible voxel, we're guaranteed that it is within the smallest voxel on the other side whatever the resolution over there might be. For if the resolution isn't fine enough to have created this smallest voxel, it is certainly within the volume of the voxel that would have been its parent, or the parent of that voxel, and so on.

Timing and sample pictures

The timing figures in Table 1 are based on statistics gathered from runtime profilers and timers. The timing statistics are used with code written in C, executed under the Unix operating system, and run on a Vax-11/780.

All the measurements were made running the same code. The old technique measurements were made using the new technique and just one huge voxel containing everything; thus there was a very slight amount of additional overhead (less than 0.01 percent). The overall execution time for the new algorithm is the sum of the octree-creation time and the image synthesis time. Of course, once the octree is built, multiple points of view can be generated without performing another setup. Note that due to the nature of naive ray-tracing techniques, the order in which the database is created (and thus the order in which the objects are intersected) can heavily influence the number of intersections necessary to render a complete picture with the traditional method.

The code runs the reflection model introduced by Cook and Torrance.¹⁷ All color calculations are produced on a 16-wavelength visible light spectrum and are converted first to the CIE color coordinates XYZ and then to monitor RGB values when the final picture is displayed.

Figures 8-13 show ray-traced pictures of increasing complexity. Each scene was illuminated with ICI standard illuminant A. The light yellow, orange, and dark purple objects possess the spectral characteristics of a desaturated-yellow gladiolus petal, bright orange gladiolus petal, and wine-colored gladiolus petal, respectively. The red objects reflect as red felt, and the green objects are leaf green. All these colors are found in Evans.¹⁸ The blue backgrounds are different shades of Carolina Blue, the usual color of the sky in Chapel Hill.

Figure 8 demonstrates reflection and shadowing in a standard ray-tracing test environment. Figures 9 and 10 show shadowing from large numbers of spheres and polygons.

Figure 11 was made by subdividing a cube and then projecting it onto the surface of a sphere.¹⁹ Each resulting patch of the sphere was tiled with one of two patterns made of four triangles.

Figure 12 shows several thousand spheres following the function $\sin(x)/x$, or sinc, rotated about the y axis.

Figure 13 is also a projected, subdivided cube. In this case, the tiling consisted of an over-and-under pattern,

which was rotated and colored appropriately for each level of subdivision. The result is a set of bands that surround the sphere.

Note that the octree needs to be created only once per static database. Thus we need only make the octree once to produce multiple pictures from different points of view. Another point to note is that most machines have a restriction on the number of digits we can store in an integer. If a node is heavily populated and has used all the digit resolution the machine can afford, the node name can be split into low and high fields with a separate integer for each. This step requires extra work, but shrewd programming can keep the extra computing down to only those nodes with expanded names.

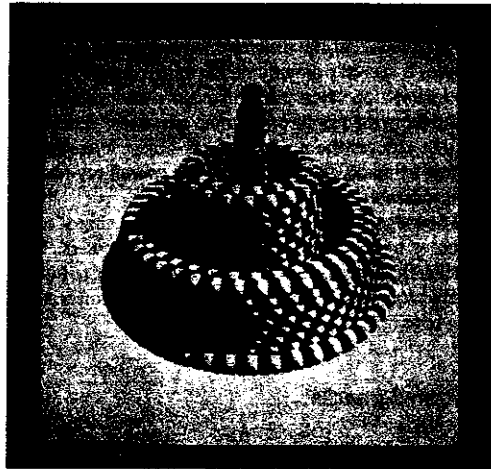


Figure 12. A large number of spheres follow the function $\sin(x)/x$ for several half-periods.

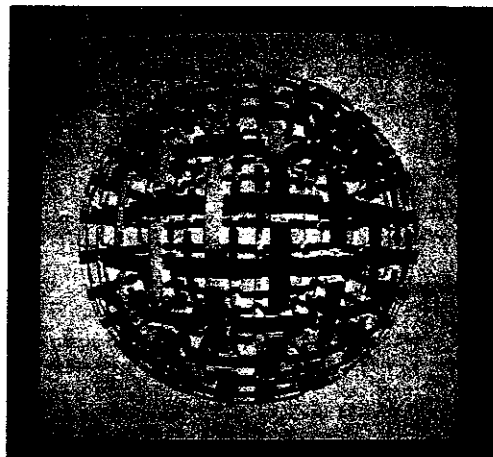


Figure 13. A single overlap pattern recursively applied to a subdivided cube of frequency 3 and then projected onto a sphere.

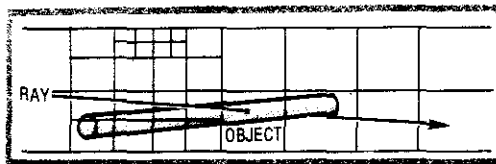


Figure 14. It is possible that a given ray may be tested against the same object several times.

In contrast to the naive techniques, it is possible that a particular ray may have to be tested against a single object several times. This uncommon event is pictured in Figure 14.

Conclusions

We have seen that the infamous slowness of ray-tracing techniques is caused primarily by the time required for ray-object intersection calculations. We have also seen a new way of tracing the ray through small subsets of space at a speed that reduces the number of ray-object intersections that must be made, thereby cutting the overall ray-tracing time considerably.

This new algorithm makes possible the ray tracing of complex scenes by medium- and small-scale computers. It is hoped that this will enable the power of ray tracing to be embraced by more people, helping them generate pictures at the leading edge of computer graphics. ■

Acknowledgments

I am grateful to Professor Frederic Way III of Case Western Reserve University who provided the opportunity and freedom to pursue this work. Thanks also go to Arch Robison and Ben Pope for helping me with algorithms to determine if particular objects were within arbitrary voxels.

Critical reviews of this paper were provided by Mark Boenke and Clayton Elwell, and editorial help was received from Ed Levinson. The recursive pyramid picture is very similar to an image produced by Alan Norton, who consented to its use as test data and an example image. Jim Weythman and Linda Laird of Bell Communications Research provided the opportunity and facilities to prepare the final text, and Tom Duff of Bell Labs helped produce early versions of the pictures in record time.

References

1. A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *AFIPS Conf. Proc.*, Vol. 32, 1968, pp. 37-45.
2. W. K. Bouknight and K. C. Kelley, "An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources," *AFIPS Conf. Proc.* Vol. 36, 1970, pp. 1-10.
3. D. S. Kay, "Transparency, Refraction, and Ray Tracing for Computer Synthesized Images," master's thesis, Cornell University, Ithaca, N.Y., Jan. 1979.
4. T. Whitted, "An Improved Illumination Model for Shaded Display," *Comm. ACM*, Vol. 23, No. 6, June 1980, pp. 343-349.
5. Robert L. Cook, Thomas Porter, and Loren Carpenter, "Distributed Ray Tracing," *Computer Graphics (Proc. Siggraph)*, Vol. 18, No. 3, pp. 137-145.
6. S. D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, Vol. 18, 1982.
7. M. K. Ullner, "Parallel Machines for Computer Graphics," PhD thesis, California Institute of Technology, Pasadena, Calif., 1983.
8. A. Glassner, *Computer Graphics User's Guide*, Howard W. Sams & Co., Indianapolis, 1984.
9. J. T. Kajiya, "Siggraph 83 Tutorial on Ray Tracing," *Proc. Siggraph*, Course 10 notes, 1983.
10. J. T. Kajiya, "Ray Tracing Parametric Patches," *Computer Graphics (Proc. Siggraph)*, Vol. 16, No. 3, 1982, pp. 255.
11. J. T. Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Computer Graphics (Proc. Siggraph)*, Vol. 17, No. 3, 1983, pp. 91-99.
12. P. Hanrahan, "Ray Tracing Algebraic Surfaces," *Computer Graphics (Proc. Siggraph)*, Vol. 17, No. 3, 1983, pp. 83-89.
13. C. L. Jackins and S. L. Tanimoto, "Octrees and Their Use in Representing Three-Dimensional Objects," *Computer Graphics and Image Processing*, Vol. 14, No. 3, p. 249-270.
14. D. Meagher, "Geometric Modelling Using Octree Encoding," *Computer Graphics and Image Processing*, Vol. 19, No. 2, 1982, pp. 129-147.
15. I. Gargantini, "Linear Octrees for Fast Processing of Three-Dimensional Objects," *Computer Graphics and Image Processing*, Vol. 20, No. 4, 1982, pp. 265-274.
16. I. E. Sutherland and G. W. Hodgman, "Reentrant Polygon Clipping," *Comm. ACM*, Vol. 17, No. 1, Jan. 1974, pp. 32-42.
17. R. L. Cook and K. E. Torrance, "A Reflection Model for Computer Graphics," *ACM Trans. Graphics*, Vol. 1, No. 1, 1982, pp. 7-24.
18. R. Evans, *An Introduction to Color*, John Wiley & Sons, New York, 1948.
19. R. Buckminster Fuller, *Synergetics: Explorations in the Geometry of Thinking*, MacMillan, New York, 1975.



Andrew S. Glassner is a graduate student with the Department of Computer Science at the University of North Carolina at Chapel Hill and a consultant in computer graphics for Bell Communications Research. He has worked on graphics at the New York Institute of Technology's Computer Graphics Lab, the IBM Thomas J. Watson Research Center, and Bell Communications Research. His research interests include raster-based computer graphics, novel input/output devices, digital sound synthesis, topology, language design, and the creative use of computers.

Glassner is the author of *Computer Graphics User's Guide*, a tutorial of computer graphics techniques for artists and other nonprogrammers. He received the BS in computer engineering from Case Western Reserve University and is a member of the ACM.

Questions about this article may be directed to the author at the University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC 27514.

Adaptive Precision in Texture Mapping

Computer Graphics (Proceedings of Siggraph '86)
vol. 20, no. 4, pp. 297-306, August 1986

Adaptive Precision in Texture Mapping

Andrew Glassner

Department of Computer Science
The University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27514

Abstract

We introduce an adaptive, iterative technique for obtaining texture samples of arbitrary precision when synthesizing a computer-generated image. The technique is an improvement on the sum table texturing method. To motivate the technique we analyze the error properties of the sum-table method. Based on that analysis we propose using a combination of tables independently or together to obtain a better estimate, and analyze the error properties of such methods. We then propose a new technique for obtaining texture samples whose accuracy is a function of the texture and the image. As part of this technique we propose the use of an auxiliary table which contains local estimates of the texture variance. We show how the iteration of a given sample may be controlled by values from this table. We then analyze the error in this method, and present images which demonstrate the improvement.

General Terms: Algorithms, Graphics

Keywords: Textures, Sum Tables, Iteration, Adaptive Refinement, Variance

CR Categories: I.3.3 Picture/Image Generation; I.3.7 Three-Dimensional Graphics and Realism

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-196-2 86 008 0297 \$00.75

1.0 Introduction

Synthetic texturing was first introduced by Catmull [Catm75]. Since that time there has been considerable interest in the correct and efficient application of texture to surfaces.

A popular use of texturing has been to apply color detail to surfaces. In this sense, textures have been used to simulate painted images [Blin76]. Another use of textures has been to simulate shape detail that would be inefficient or difficult to model directly, either through normal perturbation [Blin78], or displacement mapping [Cook84].

Some techniques for generating textures that have been discussed in the literature are stored look-up tables [Blin76], procedural routines [Gard84], and multi-dimensional methods [Peal85]. Texture has also been incorporated into synthesized images as a post-process, either to enhance the understanding of shapes [Schw83], or to generate special effects [Perl85].

Many image synthesis systems build an image by rendering pieces of surface (such as a polygon) one at a time. The pieces may be combined with previous pieces as they are rendered in a Z-buffer [Suth74] or an A-buffer [Carp84]. Alternatively, entire images may be merged after rendering [Port84], [Duff85]. In all of these schemes, the texturing process is usually one of *inverse mapping* [Feib80].

The problem of texturing may be expressed in many ways, with varying degrees of theoretical and practical considerations. A popular model which is theoretically incomplete but often visually acceptable is to assign a texture value to a pixel based on the average value of the texture within the surface region seen by that pixel. This is the model we use in this paper.

In general, a pixel may be considered to be a small window looking onto a surface. When texture is applied to that surface, the inverse viewing transform is invoked to find the projection of the pixel onto the surface (in practice, we usually project only the four corners of the rectangular pixel). We now want to know where those surface points sit in the texture. Often, the axes of two-dimensional textures are referred to as (u, v) . These four (u, v) pairs (one for each pixel corner) then describe a quadrilateral in that table. If we are rendering a very warped surface, it is possible that this quadrilateral will not be convex; in such a case we usually use the convex hull for the rest of the process. We then find an average texture value inside that quadrilateral. This average value is returned to the renderer as the average particular property of that surface seen from that pixel.



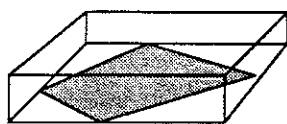
In [Will83] Williams described the *mip map*, which pre-computed averages of square regions of texture at a variety of different resolutions. In [Crow84] Crow described the *sum table*, which can provide the average value in any rectangle oriented parallel to the texture axes. The sum table is usually used to find the average texture value in the smallest oriented rectangle enclosing the mapped pixel. The texture value returned by a sum table is usually more accurate than that from a mip map, due to its ability to sample a region that more tightly encloses the texture-space image of the pixel. Sum tables have been studied in the field of probability theory as *joint cumulative probability distribution functions* on two variables [Ross76].

Both mip maps and sum tables provide a great speedup over direct averaging for every pixel, especially when the texture area covered by the pixel is large. Although sum tables are superior to mip maps, they can still present artifacts. In particular, texture outside the pixel but within the enclosing rectangle is included in the average. It is possible that the area inside the pixel is very small compared to its bounding rectangle; thus texture from outside the pixel may dominate the final average. If this extraneous texture contributes substantially to the final average, the texture value applied to a pixel may be substantially wrong.

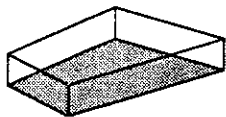
The texture-sampling problem can be expressed mathematically by writing the average value g as a nonlinear convolution of a filter kernel h with a texture function f [Andr77]. If we knew the correct filter to apply to the texture for a given sample, we could simply convolve the texture and the filter to obtain:

$$g_{\text{region}} = \iint h(x - \xi, y - \eta) f(\xi, \eta) d\xi d\eta$$

The assumption behind sum tables is that the filter h can be approximated by a unit-height filter which is 1 inside the bounding box of the texture-space image of the pixel, and 0 outside of that box (see Figure 1a).



(a) The filter used in rectangular sum tables has unit height over the bounding box of the pixel's texture-space image.



(b) We suggest a better filter would have unit height only over the texture-space pixel itself.

Figure 1

In this paper we present an improvement on the sum table technique which allows us to compensate for errors that arise from the inclusion of texture which lies outside the pixel. We will give methods to construct a filter which is 1 only inside the transformed pixel, and 0 everywhere else (see Figure 1b). Our conjecture is that this filter will provide superior results over the standard sum table bounding box filter. We present methods for obtaining this improved filter to different degrees of precision. Our technique is also iterative and adaptive, allowing us to perform only as much extra work as the image requires.

2.0 Terminology

When we refer to *texture space*, we mean that coordinates are to be interpreted as positions in the texture function. If the function is two dimensional, we call the axes u and v . When we transform a screen pixel into a corresponding quadrilateral in texture space, we call the new quadrilateral the pixel's *texture-space image*. The convex hull of this quadrilateral we will call the *inverse-mapped pixel*, or for convenience simply the *mapped pixel*. The four points that comprise a mapped pixel may form a quadrilateral, triangle, line, or single point. For simplicity, we will call the shape formed by a mapped pixel a *general quadrilateral*.

For a given region R in texture space, we will designate its area by R_a , the sum of all its values (its integral) by R_Σ , and its average value R_Σ/R_a as R_v .

We will sometimes illustrate texture operations by determining a color for a pixel, but the texture may actually be supplying any surface parameter. When we do speak of color from textures, we imply that three texture tables are accessed simultaneously (holding the red, green, and blue components of the texture color).

3.0 Fixed Polygon Approximation

When we build a sum table, each entry receives the summation of all the values in the original texture within some fixed region, oriented with reference to that entry. The traditional region used in a sum table is a rectangle. In a rectangular sum table each table entry contains the sum of the texture values between its corresponding position in the texture and the texture origin. We may extend the utility of the sum table by integrating under other shapes. The sum table is valuable because of its ability to provide the average value under a fixed region of variable size and position. However, the orientation and shape of the region must remain fixed throughout the table. Thus, we may quickly find the average value within any fixed region with a sum table, but each change in the desired shape or orientation of the region will require a new table. We will call the integration region provided by a sum table that table's *fundamental region*. Note that the region we integrate under to build a sum table is of the same shape as the fundamental region provided by the table.

The values returned by a sum table may be composed with one another to create an average value for a region with a shape other than the table's fundamental region. This may be achieved with simple linear combinations of the values returned by the table and the areas of the queried regions. Figure 2 shows an example of finding the average value in the region bounded by a letter E in a sum table with a rectangular fundamental region. The desired region is E , the enclosing rectangle is R , and the extra spaces are A , B , and C . We can express E_v , the average value in E , as

$$E_v = \frac{E_\Sigma}{E_a} = \frac{R_\Sigma - A_\Sigma - B_\Sigma - C_\Sigma}{R_a - A_a - B_a - C_a}$$

which may be generalized as

$$E_v = \frac{R_\Sigma - \sum_{i=1}^n (\text{region}_i)_\Sigma}{R_a - \sum_{i=1}^n (\text{region}_i)_a}$$

We will investigate a variety of techniques for finding the average value in regions other than a table's fundamental region. To compare these techniques it is helpful to have a measure of how much error may exist in the final value. To compare these different techniques we use the *relative error* measure:

$$\epsilon_{relative} = \left| \frac{\text{desired value} - \text{obtained value}}{\text{desired value}} \right|$$

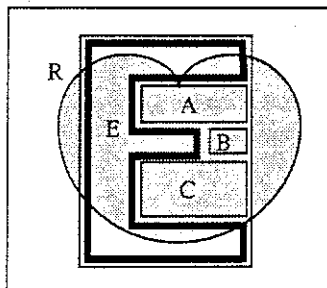
It is a bit more difficult to decide what we ought to measure. It would be nice to include the texture data itself in our comparison of texture estimation schemes. However, the only aspect of the different techniques that remains unchanged over different textures is the area averaged by that technique for a given mapped pixel. Thus, our measured values will be the area we want in our final region (whose contents are averaged to obtain a final value), and the area of the region we actually get from each technique.

Let us first analyze the area errors from the rectangular sum table. Figure 3a shows a screen pixel which has mapped into a diamond in texture space. The bounding box encloses twice as much area as the interior of the diamond. Let us call the side of the bounding box L . Then the length of one side of the diamond is $L\sqrt{2}/2$, so the diamond's total area is $L^2/2$. The relative area error in this case is

$$\epsilon_{rectangle-table} = \left| \frac{L^2/2 - L^2}{L^2/2} \right| = 1$$

One solution to this problem is to augment the rectangular sum table with a *diamond* sum table. This is simply a rectangular sum table built at a 45° angle relative to the standard rectangular sum table. When this combination is presented with a mapped pixel, based on the geometry of the pixel we determine which of the two tables to use for the texture estimate. Consider this combination of sum tables applied to Figure 3b, which shows a rectangle canted at a 22.5° angle to the table sides. We want the area inside the rectangle; this is the area of the bounding box minus the four outer triangles:

$$\begin{aligned} \text{desired area} &= L^2 - 4 \left[\frac{1}{2} \left(\frac{3}{4}L \right) \times \frac{1}{4}L \right] = \frac{5}{8}L^2 \\ \epsilon_{diamond-table} &= \left| \frac{(5/8)L^2 - L^2}{(5/8)L^2} \right| = \frac{3}{5} \end{aligned}$$



We can easily combine elements from a sum table to find the average of a shape other than the table's fundamental region.

Figure 2

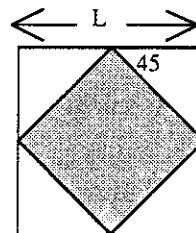
For comparison with the other techniques analyzed in this paper, let us find the relative error of the worst cases for these tables. Figure 3c shows the worst case for the rectangular table (a thin quadrilateral at 45°), and Figure 3d shows the worst case for the combined tables (a thin quadrilateral at 22.5°). Under the combined tables, if the mapped pixel's bounding box is not square, we call the shorter side L and the longer side nL . The respective errors are:

$$\epsilon_{rectangle} = \left| \frac{L - L^2}{L} \right| = |1 - L|$$

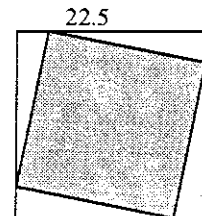
$$\epsilon_{rectangle-or-diamond} = \left| \frac{nL - nL^2}{nL} \right| = |1 - L|$$

Note that the errors in the worst case are the same, so the addition of the diamond table hasn't really earned us anything in general.

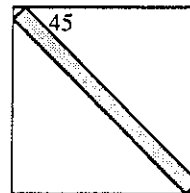
We have just seen one approach to improving estimates provided by a rectangular sum table: maintain a table with a different fundamental region and use it where the rectangular table's estimates would be at their worst. Another way to improve a texture estimate is to remove regions of texture we don't want included in our sample. We are not limited to a rectangular fundamental region for these subtracted regions. If we maintain a second table with a different shape, we may find it easier to remove unwanted areas.



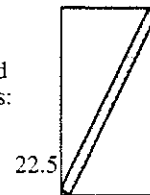
(a) A bad case for rectangular sum tables.



(b) A bad case for combined rectangular and diamond tables.



(c) A worst case for rectangular sum tables: a degenerate quadrilateral at 45 degrees.



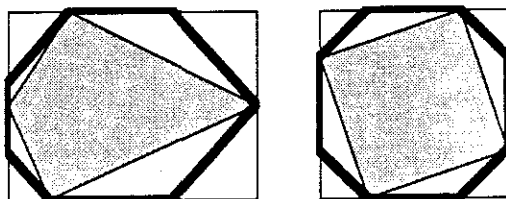
(d) A worst case for combined rectangular and diamond tables: a degenerate quadrilateral at 22.5 degrees.

Figure 3

Let us choose triangles as the fundamental region for such an auxiliary sum table. Recalling the above statements on sum tables, we may pick any fixed shape of triangle we like, but we may only have one such shape per table. Let's choose 45° right triangles, with the sides adjacent to the right angle lined up parallel to the sides of the sum table. It may appear that we need four sum tables, one for each orientation of the right angle. However, we can get by with just two triangle tables and the rectangle table. The trick is that when we want a triangle we don't have we can find its bounding box and subtract the triangle we do have. The ability to remove these triangular regions allows us to draw a generalized octagon around a mapped pixel, and obtain the average within this octagon, as shown in Figure 4a.

The worst case for the rectangle-plus-triangles scheme is when the sides of the mapped pixel make 22.5° angles with the sides of the bounding box. Figure 4b demonstrates the degenerate form of this worst case, whose error may be expressed as:

$$\epsilon_{\text{triangle}} = \left| \frac{nL - (nL^2 - 2(\frac{1}{2}L^2))}{nL} \right| = \left| 1 - \frac{1}{2}L \right|$$



(a) A general case.

(b) A worst case: all triangles are 45-45-90.

Mapped pixels approximated by 45 degree octagons.

Figure 4

This error is still linear in L , but grows half as quickly as for the combined rectangle-or-diamond table error.

We might want to improve our texture estimate further by iteratively removing more triangles of smaller size from the unwanted region. However, we would need to know when to stop. We would also like to be able to stop as soon as practical; that is, remove no more regions than the image and the texture require to provide acceptable results. In the following section we examine a method to provide a stopping point for such an iteration.

We should also mention that one can interpolate to sub-table values in the sum table, using techniques such as bilinear interpolation. Indeed, this interpolation is required to generate alias-free images. It does not save us from the kinds of oversampling errors mentioned above, however, unless carried to an extreme (as briefly discussed in Section 5).

4.0 Estimation of Local Texture Complexity

We have seen that we can improve the texture estimate (or at least the area sampled) by iteratively removing extraneous regions from the first approximation made with the bounding box. However, we noted that a stopping point is required that will enable us to stop iterating when the sampled value is sufficiently accurate for that pixel.

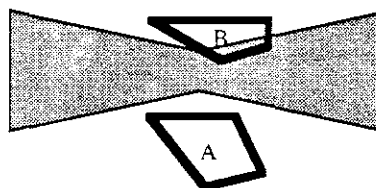
To achieve this goal we create a new table: the *variance table*. The variance table contains a local estimate of the variance of the texture at each texture position in the table. For a color texture, we can estimate local variance by looking at the 3×3 neighborhood around each texture entry, finding the mean color $(\bar{r}, \bar{g}, \bar{b})$, and computing:

$$\text{est. variance} = \frac{\sum_{i=1}^8 [(r_i - \bar{r})^2 + (g_i - \bar{g})^2 + (b_i - \bar{b})^2]}{8}$$

To use the variance table, first convert it into a rectangular sum table. When a pixel is mapped into image space, we estimate the variance in that pixel before computing the texture value. We estimate the average variance by finding the total variance inside the bounding box of the mapped pixel and dividing by the area of the bounding box.

Using this technique, we can find an estimate of the average amount of high-frequency information inside the mapped pixel, and use that information to control how much work we need to do to get a good texture estimate. Figure 5 shows a sample texture (before conversion into a rectangular sum table), along with some sample mapped pixels. Note that mapped pixel A is in a region with no local variance; the average value inside the bounding box is exactly the same as the average value inside the mapped pixel. In this case we should do no more work than that involved in looking up the bounding box. However, mapped pixel B is in a very busy area. We would like a very careful estimate of the area inside the mapped pixel in this case.

In the next section we will show how to use the estimated variance to control the accuracy of the sampled pixel.



We can approximate A coarsely, but we will want a very careful estimate for B.

Figure 5

5.0 Adaptive Polygon Approximation

In the texturing operation we desire an estimate of the average value within a general quadrilateral. One way to derive this estimate is to approximate a non-rectangular shape with rectangles. There are at least two ways to do this: additive and subtractive synthesis. We will briefly discuss additive synthesis, and then focus on subtractive synthesis for the remainder of this paper.

The image of the pixel in texture space is usually some form of quadrilateral. This quadrilateral could be scan-converted in texture space, creating a set of spans defined by one constant co-ordinate and a pair of the other co-ordinates defining the endpoints. Each such "scanline" can be looked up in the sum table. This would require one sum table access for as many lines as one cares to generate. If the quadrilateral is enclosed in a box with height L , this would require L table lookups. Alternatively, one may approximate the actual region with a set of smaller rectangles. Let us use K rectangles and apply them to the worst case (Figure 4c). Each rectangle would be L/K high by nL/K wide. Thus, the error would be

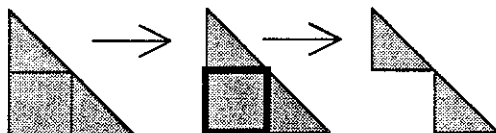
$$e_{\text{additive-synthesis}} = \left| \frac{nL - K \left(\frac{L}{K} \times \frac{nL}{K} \right)}{nL} \right| = \left| 1 - \frac{L}{K} \right| \quad (1)$$

Let us now look at subtractive synthesis. We will call the area within the quadrilateral representing a mapped pixel the *internal region*, while the total area outside the quadrilateral but within the bounding box is the *external region*. We may obtain an estimate of the average value in a general quadrilateral by first estimating the average value in its bounding box, and then removing rectangles from the exterior region. We call each removed rectangle a *bite*.

To maximize the benefit of removing bites from the exterior region we should insure that we remove the largest possible bite remaining at each step. We must also be able to identify this largest bite quickly and efficiently, since it is an operation we may perform many times for every pixel.

Bite identification is a two-step process. The first step partitions the exterior region into a set of geometric primitives, or *fragments*. The second step finds the largest available rectangular region and removes it from the set. The first step is performed once per pixel, while the second step is executed once each time we want to refine our texture estimate for a given pixel.

We chose rectangles and right-angled, table-aligned triangles for the fragments. These shapes are attractive because the area of their largest bite is easy to compute, and their extents require the storage of only four co-ordinates. The largest bite in a rectangle is the entire rectangle, and it may be stored by just its two diagonal corner points. The largest bite in a triangle is the rectangle with one corner at the right angle and the other at the midpoint of the hypotenuse, and it may be stored by its right-angle vertex and two side lengths. When a rectangle is removed it is simply deleted from the set. When a triangle is removed it is deleted from the set, and the two smaller remaining triangles are added, as shown in Figure 6.



When we take a bite out of a triangle, we remove the largest rectangle it encloses. Two smaller triangles remain.

Figure 6

We have developed an iterative technique that partitions the exterior region into rectangles and triangles. At several points in this approach we need to find the orientation of a point with respect to a line. We can find which side of the line the given point is on by examining the sign of the line equation when solved for the point. We can compute this efficiently for a point A and a line from $P0$ to $P1$ by finding the sign of

$$d = (P1_x - P0_x)(A_y - P0_y) + (P1_y - P0_y)(P0_x - A_x)$$

We first tag each point of the quadrilateral with a bit field indicating whether it lies on each of the four edges or its bounding box (points on a corner are marked by both edges). We look for three special cases before proceeding farther. Special case 1 holds if no points are corner points; then we must have the case illustrated in Figure 7. Special case 2 holds if all points are corner points; then the quadrilateral is a rectangle or a single point, and we have one of the two cases illustrated in Figure 8 (either way there is no exterior region to be partitioned). Special case 3 holds if we only have corners on a diagonal, and the other points lie along this line. We check for this case by first looking at all the corners; if we only have diagonally opposite corners we then find the sign of the distance of the other two points from that line. If the sign of both distances is 0, then they all lie along a line, and the partitioning is as illustrated in Figure 9.

If all 4 corners of the mapped pixel are on the edges of the bounding box, then the pixel must have this form: a right triangle in each corner of the bounding box.

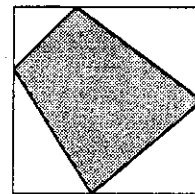
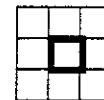


Figure 7

If all four corners of the pixel are on the corners of the bounding box, there are only these two situations:

(a) when the mapped pixel is a rectangle with non-zero area.



(b) holds when the mapped pixel degenerates into a single texture point.



Figure 8

When the mapped pixel is a degenerate line across the diagonal of its bounding box we partition the bounding box into two triangular regions of equal size, indicated A and B.

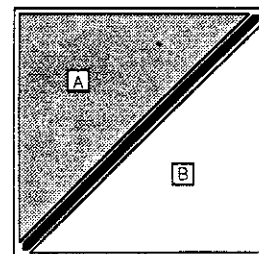
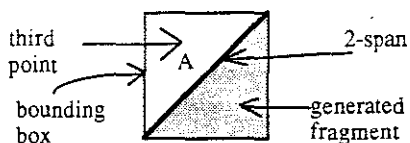


Figure 9



If the 2-span forms a diagonal of the bounding box we examine a third point (A); if possible this point is chosen to lie off the line formed by the 2-span. Based on the direction of the 2-span and the position of A, we can tell which triangle to create as a fragment.

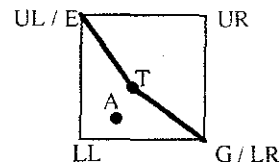
If the 2-span travels from edge to edge, then it must cut off a corner. We can deduce from the rules that created the 2-span that the rest of the polygon must lie away from that corner. We thus create the fragment triangle between the corner and the 2-span.

(a)

Figure 10

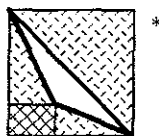
(b)

s1 = sign of distance from (E,T) to A
 s2 = sign of distance from (T,G) to A
 s3 = sign of distance from (UL,LR) to T

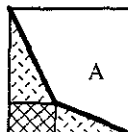


if (s3 > 0) then

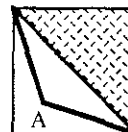
if ((s1 = 0) AND (s2 = 0)) then



else if ((s1 <= 0) AND (s2 <= 0)) then

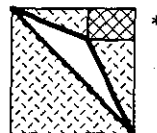


else

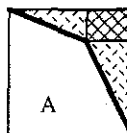


else

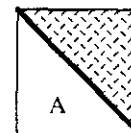
if ((s1 = 0) AND (s2 = 0) AND (s3 != 0)) then



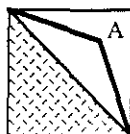
else if ((s1 > 0) AND (s2 > 0)) then if (s3 != 0) then



else



else



* An asterisk indicates this partition is complete at this step and no further spans should be processed

When we process a 3-span, we need a variety of values to help us create the fragments. We compute the sign of the distance from the middle point of the 3-span (denoted T) to each of the two corners not included in the 3-span. We also want to know if the fourth point A is on the same side of both segments of the 3-span. To this end we compute the distance from A to each of the two line segments. We then process the 3-span as shown above. The same process is followed for 3-spans on the other diagonal, with appropriate re-labelling. On each partitioning diagram we show the location of the fourth point A, determined by the algorithm, to show how the convex hull is automatically determined as we process the span.

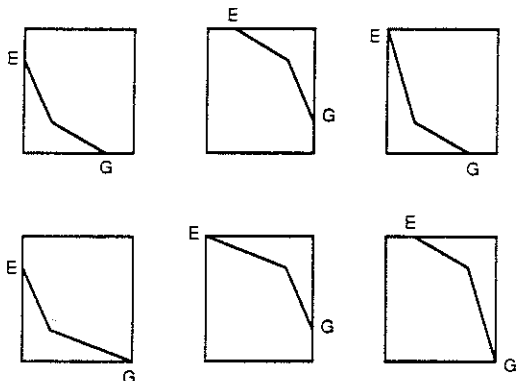
Figure 11

If none of these special cases holds, we partition the region with an iterative procedure. We start with a corner point and fix a direction to pick up the remaining points (clockwise around the original pixel works fine). We then look at the edge information for next point around the quadrilateral.

If this second point is on an edge, we call this pair of points a *2-span*. We pick one of the other two points as an auxiliary point and call it point *A*; if possible, we pick *A* to lie off the line formed by the 2-span. We then find the sign of the distance from this point to the line formed by the 2-span. If both points of the 2-span are corner points, then we create partitions as shown in Figure 10a, otherwise we create partitions as shown in Figure 10b.

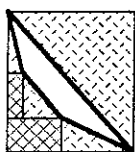
If this second point is not on an edge, then we examine the next point in turn. If this third point is on an edge, we call the trio a *3-span*, and *A* is assigned the remaining point. If the first and last points of the 3-span are corners we create partitions as shown in Figure 11, otherwise we re-label the points as shown in Figure 12 and then create the partitions shown in Figure 11.

If this third point is not on an edge, we then take the fourth point and call the quartet a *4-span*, and partition it as shown in Figure 13.

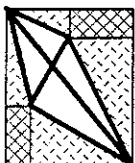


If the endpoints of a 3-span are not corners, then we label them according to these conventions and use the algorithm of Figure 11.

Figure 12



A 4-span necessarily spans opposite diagonals of the bounding box. If both of the other points are on the same side of the diagonal we partition the box into these 6 fragments.



If the two non-corner points in a 4-span are not on the same side of the diagonal, then we derive the convex hull and partition the regions outside it into 6 fragments using this scheme.

Figure 13

If the point at the end of the most recently classified span is not the same point we started with, we use last that point as the start of a new span and continue walking around the points of the quadrilateral. When we return to our starting point, we will have walked around the entire outside of the mapped pixel, partitioning the region between its convex hull and its bounding box into triangles and rectangles. The partitioning algorithm is summarized in the Appendix.

We are then prepared to remove bites from the external region, as guided by this partitioning. The process is recapitulated in Figure 14. Figure 14a shows a mapped pixel, 14b shows its decomposition into triangles and rectangles, 14c shows the removal of the first bite, and 14d shows the removal of the first six bites.

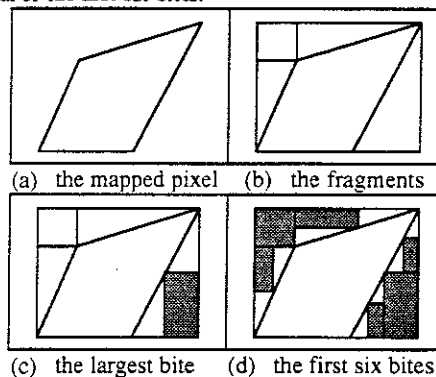


Figure 14

It is informative to compute the area error left after each step in the refinement of the estimate. A worst-case general quadrilateral consists of a line from one corner of its bounding box to its diagonal opposite. Let us label the shorter side (if there is one) as *L*, and the longer side as *nL*. Both regions around *L* have equal area; let us take the largest bite out of one of them. The area left after this bite is now

$$desired = nL^2 - \left(\frac{L}{2} \times \frac{nL}{2}\right) = \frac{3nL^2}{4}$$

Thus the relative error is

$$\epsilon_{one-bite} = \left| \frac{nL - (3nL^2/4)}{nL} \right| = \left| 1 - \frac{3}{4}L \right|$$

Similar reasoning for other numbers of bites leads us to a piecewise-linear approximation to the curve 2^{-n} , with exact matches where *n* is an integer. We thus arrive at the general formula for the error after *k* bites:

$$\epsilon_{adaptive}(k) = \left| 1 - \left(\frac{3(2^{\lceil \log_2 k \rceil}) - k - 1}{4^{\lceil \log_2 k \rceil}} \right) L \right| \quad (2)$$

This formula gives us a relationship (albeit a little complex) between the number of rectangular bites taken from the area and the resulting relative area error. Since this analysis was carried out for the worst case, if we take enough bites to meet this error for any situation we are guaranteed a maximum bound on the relative error.

The variance table and the results of Equation 2 are used to determine a maximum bound on the number of bites we need to take from a sample. We simply use a linear relationship between the range of variance and the range of error, adjusted to err on the side of over-refining.



6.0 Implementation and Results

The implementation of the technique was written to run in either of two environments: on a VAX-11/780 running UNIX BSD4.2, or within the Adage/Ikonas RDS-3000 raster graphics engine. To this end, the code was written in *gia2* [Bish82], a dialect of C.

The implementation used to generate the pictures partitions the external regions with the iterative span classification technique. The generated rectangles and triangles (or *fragments*), are kept in a doubly-linked list. Each entry in the list contains the co-ordinates needed to describe the fragment, the area of the largest bite it contains, and a pair of forward and backward pointers. Bites are taken from the sample until no fragments with non-zero area remain, or the maximum number of bites (as determined from the average variance) have been taken.

It is most efficient to pre-allocate memory for storage of the fragment list. An upper bound on the size of the list is the maximum number of starting fragments (6) plus the maximum number of bites (because each triangle bite adds one triangle to the list). When there are no more entries left to fill in the list we simply discard the fragments we can't accommodate. The maximum number of bites can be found from the last entry in the error/bite correspondence table (discussed below). In the current implementation an upper limit of 80 bites is imposed, so the fragment list has 86 entries.

Equation 2 expresses the allowable error, in terms of the number of bites taken. We want the opposite relation, i.e. how many bites to take given a particular error. To compensate for this problem we create a table indexed by tolerable error. When we have a maximum allowable error (derived from the variance map) we scan this table for the first entry with an error value less than the allowable error. The associated number of bites is the value we use to terminate the refinement.

It is interesting to note that the three color tables and the variance table can all be stored in a single square array. We used 256×256 tables, arranged in a 512 frame buffer as a two-by-two matrix.

As a demonstration of the new technique case we present Figure 15, which shows images of a black-and-white checkerboard in perspective. In the lower-left is the image generated by standard sum tables. As the checkerboard nears the horizon the sum table image blurs into a grey band. In the lower-right is the image generated by the technique described in this paper. The black and white squares of the checkerboard near the horizon are still resolvable, especially as we sight along the diagonals. Above each checkerboard is an enlargement of the square region indicated in red.

Figure 16 is an image which was generated along with the bottom-right image of Figure 15. It shows how many bites were taken on a pixel basis. Black indicates no bites were taken; a whiter entry indicates a larger number of bites. Pure white indicates 21 bites for this image. Note that the iterative technique only executes where the standard sum table would not provide a good estimate.

Figure 17 shows the same checkerboard receding at a 45° angle to the axes of the pixels.

Table 1 summarizes the number of rectangular sum table samples taken for the two images. Recall that the division operation needed to derive an average from a sum and an area is still executed only once per pixel in the new technique.

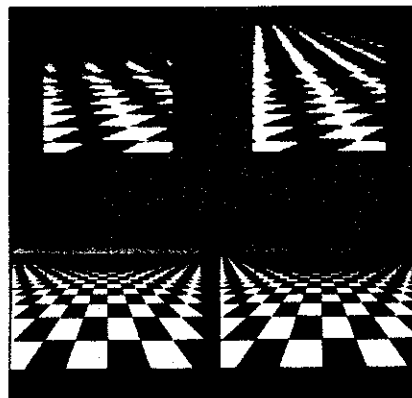


Figure 15



Figure 16

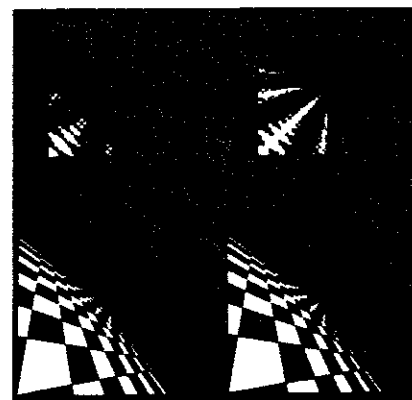


Figure 17

	Figure 15	Figure 17
Rectangle samples in standard sum table	50,176	65,536
Rectangle samples in new technique	76,942	88,774
Relative increase	1.5334	1.3545

Experimental Costs of New Technique and Sum Tables

Table 1

7.0 Discussion and Future Work

The use of the variance map to determine how many bites to take in an estimate seems to be a good approximation, but it's not ideal; in fact it can lead to estimates which are much higher than they ought to be. A better way to determine which bite to take would be to take the Fourier transform of all the possible bites at each step, and choose the bite with the maximum energy under its transform. The drawback to this scheme is clearly the high computational cost involved in taking the transforms and evaluating their energy. It would be interesting to examine techniques to get a quick estimate on these values.

It would be nice if we could remove bites from the concave portions of concave mapped pixels, rather than work with their convex hulls. It would be interesting to look for other fragment shapes that had the storage and simplicity characteristics of oriented rectangles and oriented right triangles, but could also give us a handle on approximating concave mapped pixels.

It should be noted that there is an inherent limit on the theoretical precision of this technique. As mentioned in Section 1.0, we are not performing an ideal filtering of the texture when we derive our estimate. Our first major assumption was to use rectangular, abutting "Fourier windows" to control our examined texture region. Our second assumption was to effectively sample the texture with a delta function, instead of a proper filter. These assumptions usually produce good results in synthesized images. However, after a certain point further refinement of the texture estimate by the techniques presented here will not come closer to a theoretical value. This theoretical drawback does not seem to detract from the general usefulness of the technique.

It is true that the complexity measure described in this paper is best for textures with large homogeneous areas (such as checkerboards!). Complex textures will have very high values throughout the variance map. This isn't too bad, since we will usually want very accurate estimates of complex textures. But this is another reason that a better complexity estimator than the variance would be valuable.

We have investigated another way to determine the partitions of a mapped pixel and its bounding box. We have found that there are 25 types of box-bounded convex quadrilaterals. If we can quickly determine to which of the 25 types a given mapped pixel corresponds, we can have the entire partitioning immediately. We hope to follow this line of thought farther.

The techniques described in this paper can be extended in a straightforward way to sum tables of 3 or more dimensions. In the 3d case we would remove right pyramids and right parallelepipeds from our bounding right parallelepiped to refine the texture estimate. The fragmentation code would be considerably more complex.

8.0 Acknowledgements

Several of the ideas in this paper were considered independently by Ken Perlin [Perl86]. Specifically, he investigated three-dimensional tables for the triangular tables discussed in Section 3, using two axes for u and v and the third for the free angle.

Thanks go to my advisor Henry Fuchs for his enthusiastic support. Thanks also go to my fellow students in the UNC-CH Computer Graphics Lab, whose expertise and insight contributed greatly to this work. Many of the ideas and results in this paper grew out of conversations with Greg Abram, Phil Amburn, Larry Bergman, John Gauch, Eric Grant, Jeff Hultquist, Marc Levoy, Chuck Mosher, and Lee Westover. Comments on this paper were offered by several of the above and Bobette Eckland.

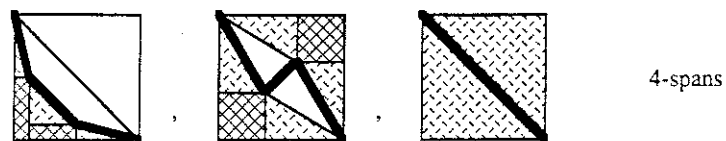
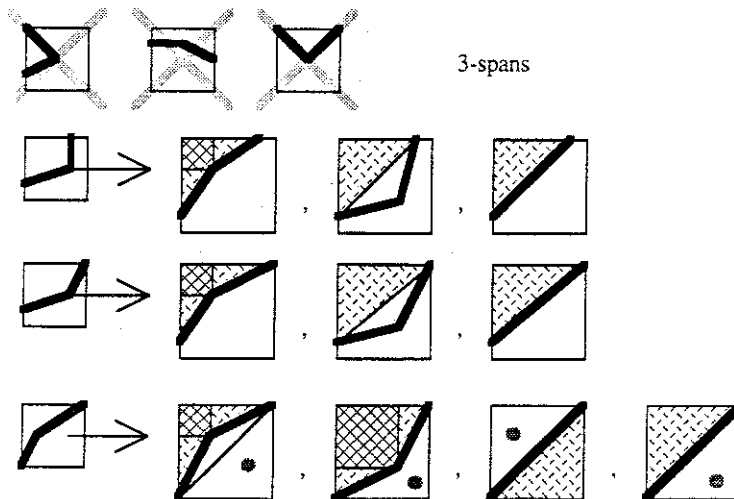
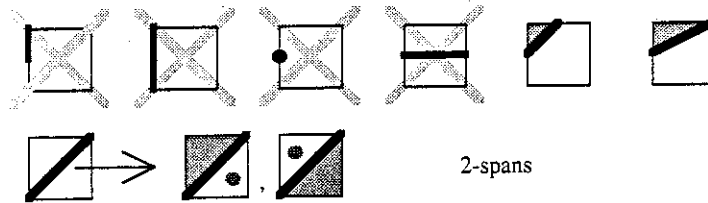
Special thanks go to Jeff and Mary Hultquist, who volunteered to assist me in the final production of this paper. The excellent layout and pasteup are entirely due to their talents and friendship.

9.0 References

- [Andr77] Andrews, H.C., and Hunt, B.R., "Digital Image Restoration," Prentice-Hall, Inc. (1977)
- [Bish82] Bishop, G., "Gary's Ikonas Assembler Version 2," UNC-CH Computer Science Department Technical Report, June, 1982
- [Blin76] Blinn, J., and Newell M.E., "Texture and Reflection in Computer Generated Images," *CACM* 19, 10 (Oct 1976)
- [Blin78] Blinn, J., "Simulation of Wrinkled Surfaces," *Computer Graphics*, vol. 12, no. 3, August 1978
- [Catm75] Catmull, E., "Computer Display of Curved Surfaces," *Proc. IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, May 1975.
- [Carp84] Carpenter, L., "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics*, vol. 18, no. 3, July 1984
- [Cook84] Cook, R., "Shade Trees," *Computer Graphics*, vol. 18, no. 3, July 1984
- [Crow84] Crow, F., "Summed-Area Tables for Texture Mapping," *Computer Graphics*, vol. 18, no. 3, July 1984
- [Duff85] Duff, T., "Compositing 3-D Rendered Images," *Computer Graphics*, vol. 19, no. 3, July 1985
- [Feib84] Feibush, Levoy, M., Cook, R., "Synthetic Texturing Using Digital Filters," *Computer Graphics*, vol. 14, no. 3 July 1980
- [Gard84] Gardner G., "Simulation of Natural Scenes Using Textured Quadric Surfaces," *Computer Graphics*, vol. 18, no. 3, July 1984
- [Peal85] Peachey D., "Solid Texturing of Complex Surfaces," *Computer Graphics*, vol. 19, no. 3, July 1985
- [Perl85] Perlin, K., "An Image Synthesizer," *Computer Graphics*, vol. 19, no. 3, July 1985
- [Per86] Perlin, K., private communication.
- [Port84] Porter, T., and Duff, T., "Compositing Digital Images," *Computer Graphics*, vol. 19, no. 3, July 1985
- [Ross76] Ross S., "A First Course in Probability," MacMillan Publishing Co, Inc. (1976)
- [Schw83] Schweitzer D., "Artificial Texturing: An Aid to Surface Visualization," *Computer Graphics*, vol. 18, no. 3, July 1984
- [Suth74] Sutherland, I., Sproull, R., Schumaker, R., "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, vol. 6, no. 1, March 1974
- [Will83] Williams L., "Pyramidal Parametrics," *Computer Graphics*, vol. 18, no. 3, July 1984

Appendix: derivation of the partitioning algorithm

The figure below shows the possible spans that may arise in a mapped pixel. When we consider edge information some spans become unrealizable; these are marked X. For example, the fifth span on the first row must have the given partitioning; if the quadrilateral had any points inside the shaded triangle the endpoints of the span would be corners. Sometimes we need another point of the polygon to decide which side of the span to partition. This point is chosen to lie off the line(s) formed by the span, if possible, and is marked ● in this diagram.



Supporting Animation in Rendering Systems

In my opinion, one of the most important issues facing the rendering system designer today is the integration of animation concepts into the rendering package. Traditional rendering systems have produced instantaneous "snapshots" of the world they imaged. Thus, each object in the database could be described by a single, static set of parameters that described the shape, orientation, surface characteristics, and other features of the object.

It is well known that motion blur enhances the apparent fluidity of animation. I believe that motion blur is sufficiently attractive that all future high-quality animation systems will provide it as a standard rendering feature.

But when we try to incorporate motion blur into standard rendering systems we find that there are severe difficulties. The crux of the problem is that time is no longer a constant throughout the rendering pipeline (see Figure 1). It is well known that any regular sampling of an unfiltered signal may lead to aliasing, but that stochastic sampling can replace this aliasing with the less objectionable artifact of noise [Cook84], [Dippe85], [Lee85], [Kajiya86]. In a stochastic sampling system, the hidden-surface resolver (usually near the end of pipeline) is now deciding at what time to sample the database, and these decisions must be supported by the object transformation software (usually near the start of the pipeline). A naïve approach would be to connect the time output from the renderer to the time input for the object transformer, which would then transform the entire database to position it at the requested time. Not only must the database transformer perform standard matrix operations, but it must also perform all of the other operations needed to support the database transformations designed by the animator, including the determination of the position of objects along control splines, and the interpolation of various shape and surface control parameters.

This is clearly a lot of work, and will involve the wasted effort of transforming many objects that don't even participate in this sample. For large databases this wasted effort will dominate the rendering time. For this reason I believe that the standard rendering pipeline is inappropriate for rendering animation with motion blur.

Other alternatives include distributed ray tracing and solving the rendering equation. In these approaches, rays at different times enter the database looking for intersections. Various algorithms may be used to prune the number of objects to be intersected, or replace

groups of objects with simpler objects, but still at some point in the process some objects must be transformed to the appropriate position, orientation, shape, and so forth to test for intersection with the ray. This is the same task performed by the object transformer in the traditional pipeline. Note that the determination of the position and shape parameters for the ray may be as complex as the animation system itself, requiring moving objects along splines, interpolating shape parameters, and so on.

But observe that this is also the exact same task performed by the animation editing system. When an animator sits in front of an interactive animation system, the animator is interacting with an animation database through a set of controls and displays. In effect, the animator specifies operations to be performed on the animation database, the results of which are then displayed. The animation system must then support all the same object transformation and distortion operations on objects that are performed when testing a ray for intersection.

Thus I propose using a single set of routines for accessing the animation database. The database itself is composed of object-oriented modules which may communicate; this is a very powerful paradigm for developing databases, both static and moving [Amburn86]. The animation system and the renderer both speak to the database only through its interface.

The animation system interacts with the database with two commands:

SetParameters(object, time, parameter-list)

Represent(object, time, representation, controls)

SetParameters associates a parameter list with an object for a given time. The parameters may describe a transformation matrix, a texture map index, a shape distortion control, a force vector, or any other information that the object may find useful. This information is stored internally by the object and may be used as traditional keyframe control, or for more complex purposes such as parameters that affect how the object animates itself (e.g. electric charge in an electromagnetic field).

Represent causes an object to represent itself in some way, usually for display. Possible representations may include vectors, polygons, patches, and so forth. The controls determine other object-related properties affecting the representation, such as level of detail.

The rendering system interacts with the database with two commands:

Intersect(object, ray, time)

Complete(object, ray, time)

Intersect determines the intersection of a given object with a given ray at a given time.

The result of such an operation is either a notification of a miss, or the ray position (e.g. the scalar s in the ray equation $\mathbf{R} = \mathbf{R}_0 + \mathbf{R}_1s$ for ray \mathbf{R}).

When the nearest intersection has been found, Complete finishes the job for the desired object. The result of complete is a description of the intersection containing all information for which the object is responsible (e.g. Complete will return surface color, reflected and refracted rays (for distributed ray tracing), surface physics co-efficients (specular, diffuse, highlight, etc.), and other shading and geometrical information).

This system places the responsibility for uniform sampling on the objects, rather than on the ray generator. For example, an object may subdivide its generic reflection hemisphere into n solid angles. Each time a reflected ray needs to be generated, the object selects one of the solid angles not chosen before. The reflected ray is generated to fall somewhere within that solid angle, and the angle is marked as used. In this way successive requests will receive reflected rays distributed over the hemisphere, no matter where on the object the intersections actually occur; see Figure 2. When all angles are used, they are all cleared and the process begins anew. I believe that this method of distributing rays is as effective as methods controlled by a ray generator. Some experiments with distributed light sources and rough surfaces have produced good images, which lends support to this view.

The composite system is shown in Figure 3. The animation database is insulated by its access routines. The animator interacts with controls that adjust and query the animation database. The rendering system contains a module which queries the database for intersection events and processes those events to form an image.

In summary, I advocate the incorporation of ray-object intersection routines into a protected animation database system. Under this scheme, as more sophisticated and involved animation and motion control techniques are presented to the animator, the rendering system remains unchanged but capable of rendering the new animation immediately.

References

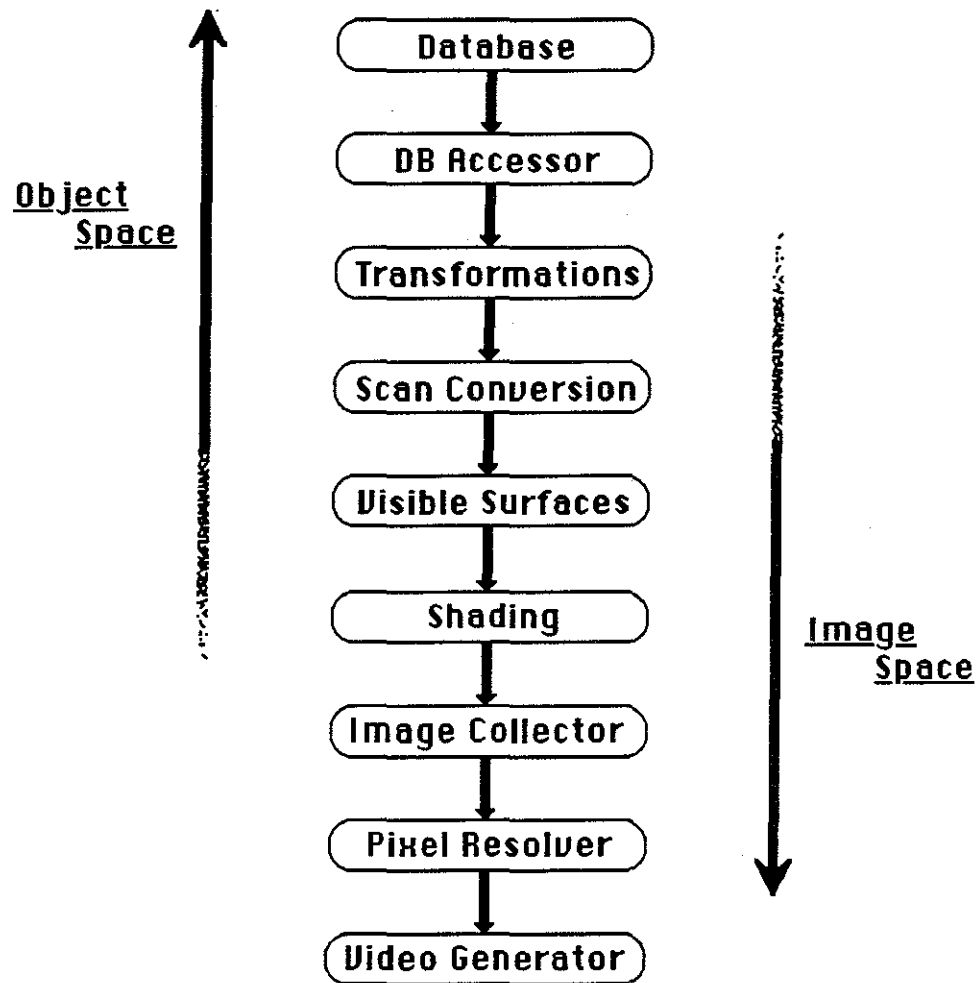
[Amburn85] Amburn, P., Grant, E., Whitted, T., "Managing geometric complexity with enhanced procedural models," *Computer Graphics* 20, 4 (August 1986)

[Cook84] Cook, R.L., Porter, T., and Carpenter, L., "Distributed ray tracing," *Computer Graphics* 18, 3 (July 1984)

[Dippe85] Dippe, M.A.Z, and Wold, E.H., "Antialiasing through stochastic sampling," *Computer Graphics* 19, 3 (July 1985)

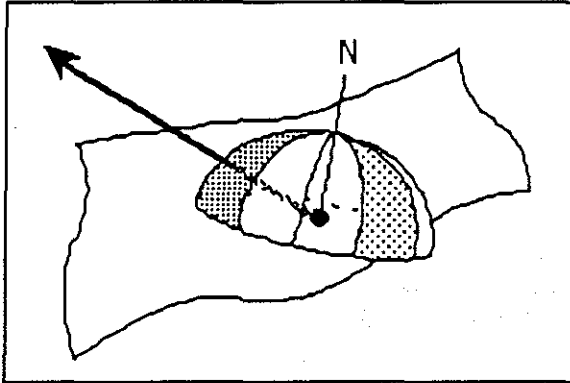
[Kajiya86] Kajiya, J.T., "The rendering equation," *Computer Graphics* 20, 4 (August 1986)

[Lee85] Lee, M.E., Redner, R.A., Uselton, S.P., "Statistically optimized sampling for distributed ray tracing," *Computer Graphics* 19, 3 (July 1985)

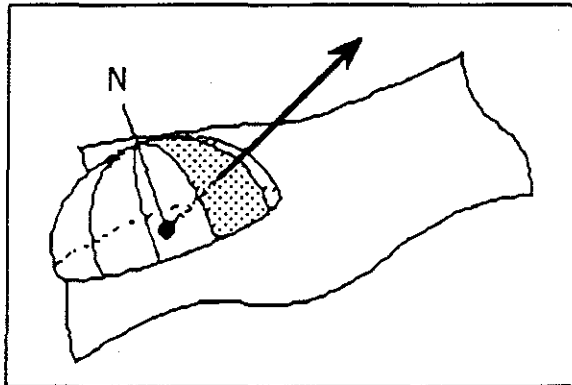


The Image Generation Pipeline

Figure 1



When a reflected ray is needed, the reflection hemisphere is searched for an unused solid angle. When such an angle is found, it is marked as used, and the reflected ray is built to pass through it. The sizes of the various solid angles and the order through which they are searched is weighted by the shape of the reflection distribution function.



The next time this object needs to generate a reflected ray, it searches its reflection hemisphere for an unused solid angle. This new angle is marked used, and the new reflected ray is constructed. Similar book-keeping is used by all objects with sampled parameter spaces. For example, an area light source subdivides itself into several smaller areas. Each time a shadow ray is generated, the light source is asked for one of these pieces. This is the location on the light source towards which the shadow ray is directed.

Figure 2

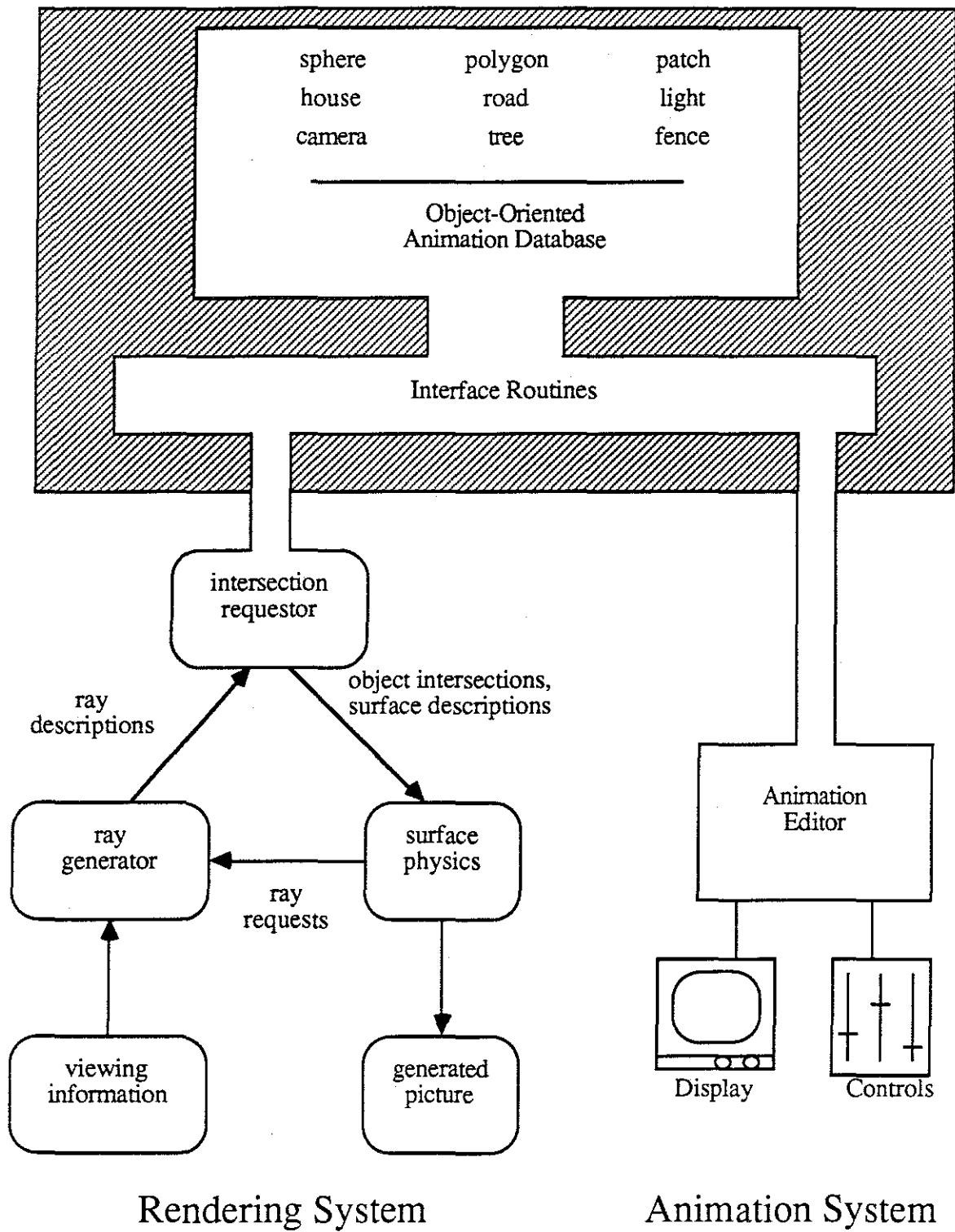


Figure 3

Template Parameterization for 3d Pose Interpolation

Abstract

Many 3-dimensional animation systems use hierarchical transformation trees to describe articulated 3-d models. These models may be animated by interpolating keyposes created (either explicitly or implicitly) by the animator at different times. Correct keypose interpolation demands that all model trees involved in the interpolation have the same topology. A second, more difficult condition for interpolating keyposes is that corresponding nodes in the transformation trees must be composed of the same transformations in the same order (called a template). We believe that a model designer should be free to specify arbitrary transformations at each node; somehow these transformations must be converted into the template form necessary for interpolation. This raises the template parameterization problem: how to find the parameters for the transformations in a given template from the composite matrix at a node. We present a new template composed of six transformations in twelve parameters, and show how it may be efficiently computed with standard numerical algorithms.

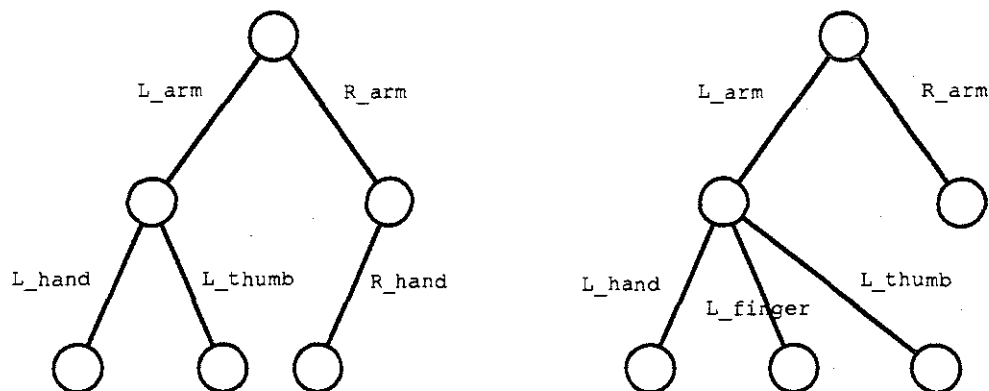
Introduction

Hierarchical modeling is a powerful and popular technique for constructing and animating complicated objects. The general idea is to build a tree of transformations and primitives, where each node in the tree represents a linear transformation, and each edge represents one or more primitive objects. The model description is recovered by

processing the tree from the root to the leaves, accumulating transformations and placing (and perhaps rendering) objects as they are encountered.

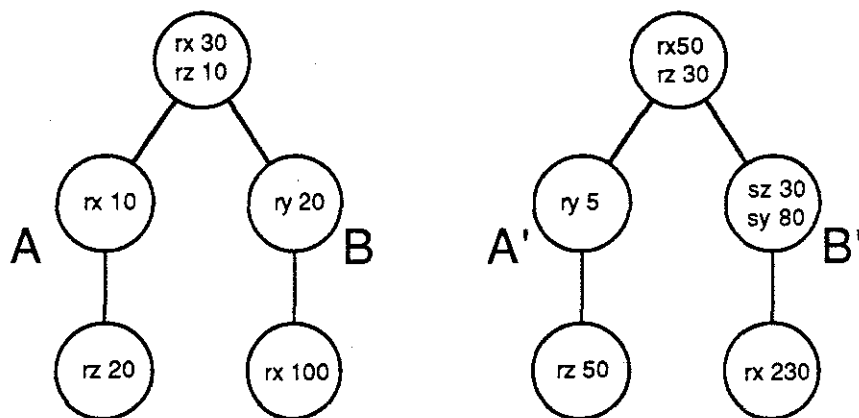
This method may be used to construct a powerful modeling language [Duff84]. In this context, the designer prepares a text script which contains the transformations and primitives that describe the model. The modeling language itself imposes no restrictions on the structure of the tree described by such a script; it is perfectly possible that some nodes have many branches while others have none. Similarly, the transformations at the nodes are usually unrestricted; any number of transformations may be specified in any order. The transformations are simply accumulated into a current modeling matrix that is used for all objects placed at that node. That matrix is also propagated to all child nodes.

This hierarchical structure is ideal for parametric animation. In such a system the designer builds a sequence of instances of the modeling tree either explicitly (in a keypose system, such as [Stern83]), or implicitly (in an event-driven system, such as [Gomez84]). For simplicity, we will assume in this paper that the keyposes are available explicitly. At moments for which a keypose was specified, the pose completely describes the object. At all other moments in the animation the object is described by a transformation tree interpolated from these keys. In order to build an interpolated tree, all trees involved in the interpolation (2 trees for linear interpolation, 3 for quadratic, and so on) must have the same topological structure, as shown in Figure 1a. This is usually not a severe problem for articulated figures such as humanoid shapes and mechanical systems of rigid parts, since the topology of these structures doesn't change much over time. When the tree structure of a model must change over time, the modeling language may include constructions that allow the user to describe the time and form of the topological change.



1a

These two trees cannot be interpolated since they have different topologies



1b

Both trees have the same topologies, but nodes A-A' and B-B' cannot be interpolated directly as given, since their transformation sequences are different.

Figure 1

A more difficult restriction to meet, however, is that the topological structure of the transformations at corresponding nodes must also be the same, as shown in Figure 1b. Parametric transformation interpolation requires that we decompose the matrix into a sequence of primitive modeling transformations, each a function of one or more parameters. If we attempt to avoid this decomposition and instead interpolate the elements of the composite transformation matrix at each node we get very poor results: the in-between matrices are shears of the key matrices, and objects are distorted by even a simple rotation.

A particular, fixed set of transformations in a node structure is called a *template*. In a template we only allow the “primitive transformations” of translation, rotation, shear, and uniform scaling (the last is a special case of shear). For notational convenience, we extend the term “primitive” to include the 3-d composite matrices combining similar, commutative elements (for example, a single transformation that applies translations to all 3 axes is also a “primitive” matrix of 3 parameters).

We refer to the problem of converting an arbitrary modeling matrix into a fixed sequence of parametric primitive transformations the *template parameterization problem*. In the next section we summarize previous work on this problem, and then in following sections we present our new solution.

Previous Work

Consider the following transformation template B, used in the bbop animation system [Stern83]:

(B) Translate_XYZ (tx, ty, tz)
 Rotate_XYZ (rx, ry, rz)
 Scale_XYZ (sx, sy, sz)

where Translate_XYZ(tx, ty, tz) means the three transformations Translate_X(tx), Translate_Y(ty), and Translate_Z(tz) in that order. Rotate_XYZ and Scale_XYZ have similar meanings. So B consists of 3 transformations in 9 parameters. The parameters which determine template B may be found by symbolically composing these 3 matrices,

and then simultaneously solving the 16 equations relating the symbolic matrix containing the nine parameters and the numerical matrix which is to be matched.

The problem of extracting the parameters for template B from a particular numerical matrix has been studied by Greene [Greene83] [Greene86]. He developed a variety of identities which show the relationships between various sets of transformations. Using these identities, we find that template B is not completely general.

One way to see this is to consider the following identity from [Greene83]:

$$\begin{aligned} \text{shear_XY (a)} &= \text{Rotate_Z (b)} \\ &\quad \text{Scale_X (c)} \\ &\quad \text{Scale_Y (d)} \\ &\quad \text{Rotate_Z (e)} \end{aligned}$$

$$\text{where } c = \frac{\tan(a) \pm \sqrt{4 + \tan^2(a)}}{2}, \quad d = \frac{1}{c}, \quad e = \tan^{-1}(c), \quad b = e - \frac{\pi}{2}$$

From this we can see that shear transformations are equivalent to differential scaling (scaling by different amounts along the principal axes) nested between rotations. Since template B contains only a single rotation node, that template cannot match a matrix that includes shear.

This is a serious restriction, since it means not only that designers cannot use shear explicitly, but they also may not scale along just one axis when modeling (since that can introduce shear if nested between rotations). This problem can be somewhat alleviated by the animation system, since after the modeling matrix has been decomposed the animator may apply differential scaling to particular keyposes. Nevertheless, we would prefer to allow the designer to build the desired model in the script, rather than relying on hand adjustment when animating. This preference becomes a necessity when scripts are generated by a simulation program, which may need to include shear and differential scaling to correctly model some deformations.

An improved approach due to Greene involves using templates of higher complexity. Not all such templates are practical; for example, Greene states that computing the parameters for the transformations in his template T6 presents “an intractably difficult algebra problem.” A “solvable” system is given by the following template G:

- (G) Perspective (90, 1, c)
- Translate_XYZ (tx, ty, tz)
- Shear_YZ (syz)
- Shear_ZY (szy)
- Shear_ZX (szx)
- Shear_XZ (sxz)
- Shear_XY (sxy)
- Shear_YX (syx)
- Scale_XYZ (sx, sy, sz)

G consists of 9 transformations in 13 parameters. Greene suggests that to find the parameters in G we again write the composite matrix symbolically and then find the solution of the 16 equations describing this matrix for the 13 unknowns corresponding to a particular numerical matrix.

Greene's approach represents one solution to the parameter extraction problem. In the next section we present a new approach, using a new template based on the techniques of matrix transformations.

Singular Value Decomposition

A well-established technique in the field of numerical linear algebra is the computation of the singular value decomposition (SVD) of an arbitrary matrix [Golub71]. SVD was developed to solve problems in linear algebra, for which more direct routines fail from numerical instability.

The use of SVD for linear algebra problems is very much like the use of the adjoint matrix in modeling problems, where the adjoint is often used where the theory requires the inverse. The adjoint of a matrix differs from the inverse by only a constant factor (the inverse of determinant of the matrix), but it always exists, even when the matrix is singular. The adjoint is also easier to compute. In cases where the scaling is critical, we can always divide the result by the determinant (if it is non-zero).

Similarly, SVD is useful for inverting linear algebra problems involving a singular matrix. Suppose we have a singular transformation A and two vectors x and b , related by $Ax=b$, and we wish to find x given A and b . The normal solution is to invert A and solve $x=A^{-1}b$. Unfortunately, if A is singular then A^{-1} doesn't exist, so this technique is not

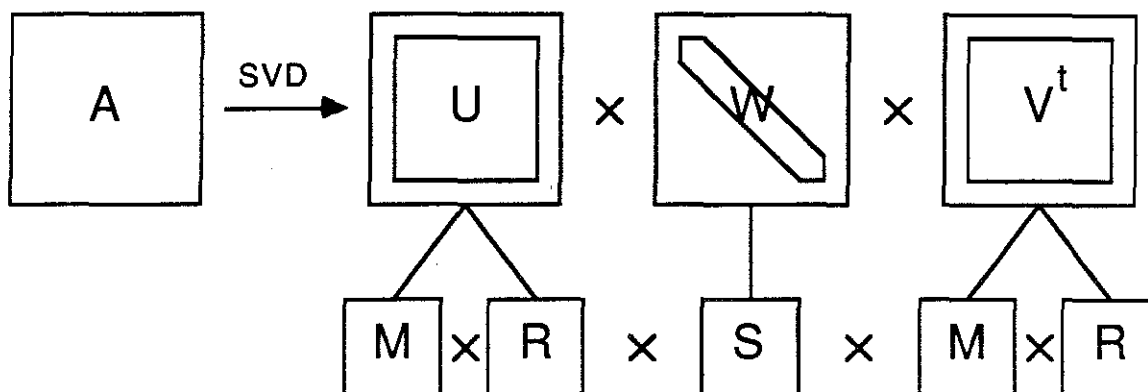
useful. But if we use the SVD technique, then we can compute a vector \mathbf{x} which is the “closest” correct answer, in the least squares sense that it minimizes the residual $\|\mathbf{Ax} - \mathbf{b}\|$.

The input to SVD is a matrix; the output is three new matrices, which when multiplied together reconstruct the original matrix. SVD is useful for our purposes because of the form of the matrices it produces. Given a square, real matrix \mathbf{A} , the singular value decomposition of \mathbf{A} results in the three real matrices \mathbf{U} , \mathbf{W} , and \mathbf{V} , such that $\mathbf{A} = \mathbf{U}\mathbf{W}\mathbf{V}^t$, where \mathbf{V}^t is the transpose of \mathbf{V} . Some properties of the three SVD-generated matrices \mathbf{U} , \mathbf{W} , and \mathbf{V} will prove to be important to us. First, the entries along the diagonal of \mathbf{W} (called the singular values of \mathbf{A}) are closely related to the eigenvalues of \mathbf{A} . Second, \mathbf{U} and \mathbf{V} are both orthonormal matrices (the columns of \mathbf{U} for which the diagonal element in the corresponding column of \mathbf{W} is nonzero form an orthonormal basis for the range of the transformation \mathbf{A} ; the columns of \mathbf{V} for which the diagonal element in the corresponding column of \mathbf{W} is zero also form an orthonormal basis, this time for the nullspace of \mathbf{A}). One reason SVD is popular is because from the singular values we can find when \mathbf{A} suffers from rank degeneracies. Isolating the particular singular values that are nearly zero can help us analyze \mathbf{A} , and perhaps modify it for some other purpose. We will not be using SVD in this analytic capability; we are only interested in using the equivalent matrix representation it generates.

A New Template

Our goal is to decompose a modeling matrix \mathbf{A} into some sequence of primitive transformations whose parameters we may interpolate. Recall that a square orthonormal matrix may be interpreted as a rotation, a mirror, an inversion, or some combination of the three. We will find it useful to replace orthonormal matrices with an equivalent “MR-pair” - two matrices, where the first (\mathbf{M}) includes any mirroring or inversion, and the second (\mathbf{R}) includes any pure rotation. Recall also that a diagonal matrix may be interpreted as representing pure scaling (i.e. no shear).

Let us consider applying SVD directly to a 4-by-4 modeling matrix \mathbf{A} . Then \mathbf{U} and \mathbf{V}^t will be 4-by-4 orthonormal matrices, and \mathbf{W} a 4-by-4 diagonal matrix. We may consider \mathbf{U} and \mathbf{V}^t to represent MR pairs in a 4-dimensional space, and \mathbf{W} to represent a 4-d scaling vector. Note that in this context the fourth row and column of each matrix is not homogeneous information; they have the same spatial interpretation as the first three rows and columns. We may thus create a sequence of the form $(\mathbf{MR})\mathbf{S}(\mathbf{MR})$, which is formed from \mathbf{A} by direct application of SVD; this is illustrated in Figure 2.



Singular Value Decomposition converts a square matrix into three new matrices: a diagonal matrix flanked by two orthonormal matrices. We may consider each orthonormal matrix to be a product of a mirror-inversion matrix and a pure rotation matrix.

Figure 2

The drawback to this form is that these 4-dimensional transformations carry little intuitive meaning for us. On the other hand, 3-dimensional transformations are well understood and easily implemented. For example, recent work for rotations has shown how to smoothly interpolate composite 3-d rotations without resorting to Euler angles [Shoemake85]. We know of no similar work which applies to 4d rotations.

Rather than develop a new body of techniques for 4d transformations, we can use SVD to generate 3-d transformations if we are willing to add one more transformation to our template. We will find it notationally convenient to introduce a "promotion" operator $P(\mathbf{B})$, which accepts as argument a 3-by-3 matrix \mathbf{B} and promotes it into a 4-by-4 matrix by substituting \mathbf{B} into the upper-left corner of a 4-by-4 identity matrix. The inverse operator $P^{-1}(\mathbf{B})$ "demotes" a 4-by-4 matrix \mathbf{B} into a 3-by-3 matrix by stripping off the bottom row and rightmost column.

To make SVD more useful we first note that no combination of primitive modeling transformations (rotation, translation, scaling, and shearing) alters the rightmost column of a composite transformation matrix; it is always $[0\ 0\ 0\ 1]^t$. We then note that A may be

written as $\mathbf{A}' \cdot \mathbf{T}$, where $\mathbf{A}' = \mathbf{P}(\mathbf{P}^{-1}(\mathbf{A}))$, and \mathbf{T} is an identity matrix augmented with the translation components. In diagram form, this is

$$\mathbf{A} = \begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ tx & ty & tz & 1 \end{pmatrix} = \begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{pmatrix} = \mathbf{A}' \cdot \mathbf{T}, \quad \mathbf{P}^{-1}(\mathbf{A}) = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

If we now run \mathbf{A}' through SVD, we get three new 3-by-3 matrices \mathbf{U} , \mathbf{W} , and \mathbf{V} , which represent a sequence of 3-d transformations in the order (MR)S(MR), that together make up \mathbf{A}' .

To convert each of the orthonormal matrices \mathbf{U} and \mathbf{V} into MR-pairs, we follow a practical procedure. For each matrix we first compute the cross product of the upper two rows. We then compare the six possible sign and position permutations of that result with the third row of the matrix. The correct mirroring operation is easily deduced from the permutation that aligns the two vectors, and is summarized in Table 1. Pre-multiplying \mathbf{U} or \mathbf{V} with the correct \mathbf{M} yields the associated pure rotation matrix. In degenerate cases, it is possible that more than one mirror matrix will align the two vectors; it is therefore important that we always examine the mirror matrices in the same order and use the first one that matches.

Given a matrix A with rows A_1 , A_2 , and A_3 , we compute $B = A_1 \times A_2$.

We then find the permutation of B that makes it the same vector as A_3 ; each permutation has an associated mirror-inversion matrix M . In this table, I_{23} means the identity matrix with rows 2 and 3 interchanged; $-I$ is the identity matrix multiplied by -1 .

<u>Permutation of B that matches A_3</u>	<u>Matrix M</u>
(b_x , b_y , b_z)	I
($-b_x$, $-b_y$, $-b_z$)	$-I$
(b_y , b_x , b_z)	I_{12}
($-b_y$, $-b_x$, $-b_z$)	$-I_{12}$
(b_x , b_z , b_y)	I_{23}
($-b_x$, $-b_z$, $-b_y$)	$-I_{23}$
(b_z , b_y , b_x)	I_{13}
($-b_z$, $-b_y$, $-b_x$)	$-I_{13}$

Table 1

If we multiply these the (MR)S(MR) matrices together, promote the result, and then compose this with the translation matrix T computed above, we recover the original matrix A . Alternatively, we may promote each of the five matrices first, and then compose the promoted matrices together with the T matrix.

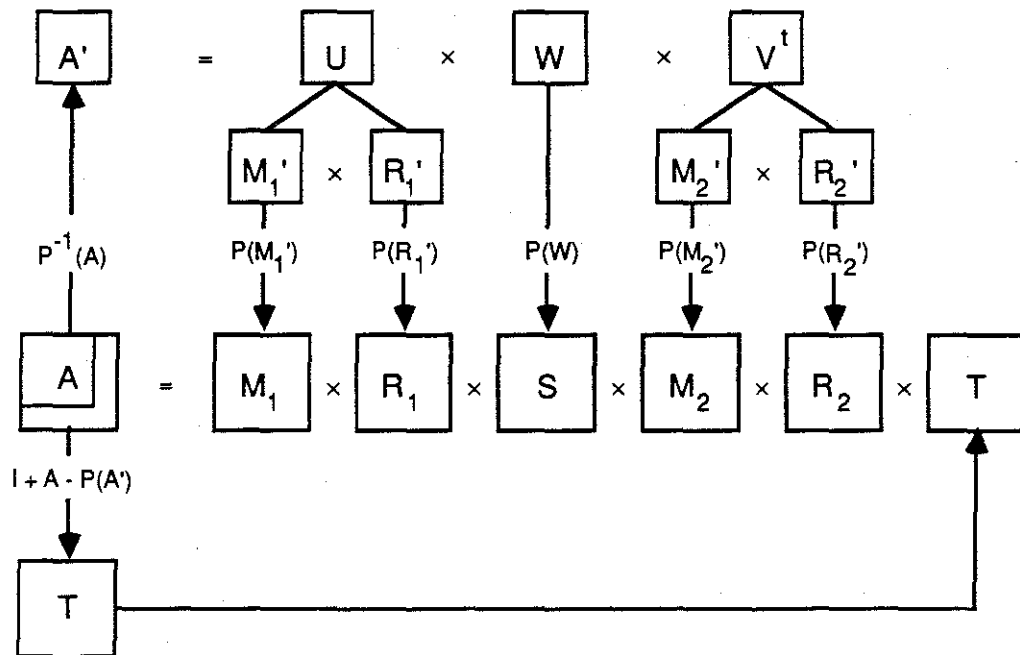
Thus SVD allows us to match the (MR)S(MR)T template:

((MR)S(MR)T) Mirror-Inversion
 Rotate_XYZ (rx, ry, rz)
 Scale_XYZ (sx, sy, sz)
 Mirror-Inversion
 Rotate_XYZ (rx, ry, rz)
 Translate_XYZ (tx, ty, tz)

which consists of six transformations in 12 parameters.

We can summarize the parameter extraction algorithm for (MR)S(MR)T with the following steps:

1. $A' = P^{-1}(A)$
2. $T = I + A - P(A')$
3. Compute $SVD(A')$, generating 3-by-3 matrices U , W , and V^t
4. Decompose $U = M_1' \cdot R_1'$
5. Decompose $V = M_2' \cdot R_2'$
6. $M_1 = P(M_1')$, $R_1 = P(R_1')$, $M_2 = P(M_2')$, $R_2 = P(R_2')$, $S = P(W)$
7. $A = M_1 \cdot R_1 \cdot S \cdot M_2 \cdot R_2 \cdot T$



Using SVD and MR pairs to match a matrix A with the (MR)S(MR)T template.

Figure 3

This technique is diagrammed in Figure 3. Note that step 5 needs the matrix V , but SVD in step 3 generates V^t . Since the promotion operator adds the right row and bottom column of the (symmetric) identity matrix, it doesn't matter if we transpose before or after the promotion, but we must not forget to transpose before working with V .

Step 7 expresses our original matrix A in the (MR)S(MR)T template. We may now convert every matrix in our tree into this form when we wish to interpolate. If we wish to interpolate Euler angles for rotation, we may extract the three angles corresponding to a given order of axes from the elements of R_1 and R_2 . Equations for the zyx order are given in [Shoemake85]; equations for other orders are easily derived. If we use quaternions for rotations, then we may extract the quaternion directly from the matrix elements in R_1 and R_2 using the equations in [Shoemake85].

To understand the motion produced by interpolating the (MR)S(MR)T template for an object we need to look at the operation of the component transformations. We can see that the first MR-pair rotates the object to align its eigenvectors with the environment's co-ordinate system, the S matrix then scales the object along its eigenvectors, the second MR pair rotates the object back into the desired orientation, and finally the T matrix positions the object in space. Thus smoothly interpolating each of these transformations will produce smooth components that together make sensible motion for the object.

Note that the two Mirror-Inversion matrices have no parameters. This embodies the natural requirement that all nodes involved in an interpolation reside in co-ordinate systems with the same handedness. Happily, since SVD seeks out the eigenvalues that make up W in decreasing order, matrices U and V will normally have the same chirality for all nodes. They may differ if an animator deliberately changes the sense of the modelling co-ordinate system between successive poses. Regardless of the interpolation scheme, such motion is sure to be somehow degenerate; perhaps the best thing to do is to allow the object to pass through itself with an inversion. Probably the best solution to this problem is to always model with consistent chirality.

Computation of SVD

The SVD algorithm is not easy to briefly summarize in detail. Consider the case of *Numerical Recipes* [Press86], an excellent 818-page book on numerical methods. In Chapter 2.9 they discuss SVD, and after a high-level summary they say, "As much as we dislike the use of black-box routines, we are going to ask you to accept this one, since it

will take us too far afield to cover its necessary background here.” And this is from a book written expressly to explain numerical algorithms! But this is not a disaster, since SVD is a popular algorithm, and actual code is available from several sources.

Fortran and Pascal programs for computing SVD are given in [Press86], and C code is available in [Press88], both of which are also available in diskette form. Computation of SVD is also provided as part of LINPACK [Dongarra79], a linear algebra package, in the routine SSVDC; pages C.122 through C.129 present the Fortran source (also available from netlib [Dongarra87] at no cost: to get the Fortran source (for SVD only) send the mail message `send ssvdc from linpack` to either `netlib@anl-mcs.arpa` on the Arpanet, or `research!netlib` on the uucp network, or use `send linpack` to get all the support code as well). An Algol listing for SVD is given in [Golub71], which also discusses the algorithm; this program is useful for study, but the algorithms from *Numerical Recipes* and LINPACK are slightly more numerically sophisticated.

We have attempted to build a “streamlined” version of SVD particularly for matching the (MR)S(MR)T template. This version was hard-coded for 3-by-3 matrices, and unrolled most of the loops for efficiency. Unfortunately, the resulting code was far more complex than that presented in the above sources, and almost no faster; it's the computations of SVD that dominate the run time, not the flow control.

For those who are inclined to understand the technique of SVD, discussions may be found in [Golub71], [Dongarra79], and [Press86], along with references to the component algorithms which work together to compute the singular value decomposition. Most of this material is rather dense due to concerns of efficiency and error in a digital computer; the algebraic manipulations at the heart of the algorithms are themselves quite clean and elegant.

Comparison

The central differences between the approach presented here and [Greene86] are the cost and stability of the parameter extractions.

The two templates are very similar in complexity, which is no surprise since they both are rather compact solutions to the problem. The (MR)S(MR)T template contains 6 transformations in 12 parameters. The G template contains 9 transformations in 13 parameters. The discrepancy in parameter count is because G includes the perspective transform, which we do not.

We wanted to compare the time required for the computation of the templates in the two approaches. Unfortunately, using Vax Macsyma [Macsyma79] we were unable to solve the system of simultaneous equations required to extract the parameters for template G; this was in fact the original motivation for this work.

Determination of the parameters for the (MR)S(MR)T template is performed by the singular value decomposition, a carefully studied and tuned algorithm which is extremely stable. Our experience with SVD has been that it runs quickly enough for modeling work. On our VAX-11/750 we can compute a 3-by-3 SVD in 850 microseconds, using the Pascal code of [Press86]. We have not tried it yet in an interactive animation environment, although we are incorporating it into an animation system under development.

Summary and Conclusions

We have presented a solution to the template parameterization problem. We have proposed the (MR)S(MR)T template which consists of 2 composite rotations, 1 composite translation, 1 composite homogeneous scale, and 2 parameter-free mirror transforms. Parameterization of this template from an arbitrary 4-by-4 modeling matrix is accomplished by trivially extracting the T matrix, executing the SVD algorithm, and separating the resulting orthonormal matrices into MR-pairs.

Our experience with this template is that its parameterization is efficient and numerically stable.

Acknowledgements

Principal thanks go to Frits Post and Wim Bronsvort. This work benefitted greatly from many discussions with Frits, who also helped get a Pascal version of SVD running on our system. Wim offered constant encouragement, and also acted as my host during my visit with the Faculty of Informatics at the Delft University of Technology. The other members of the Leerstoel Technische Toepassingen helped this guest feel at home in their friendly and supportive atmosphere. I also want to thank colleagues in the Vakgroep Informatica and the entire Facultit der Technische Wiskunde en Informatica for their assistance in arranging and supporting my visit, which made this work possible.

At the University of North Carolina at Chapel Hill I am grateful for the freedom and support provided by my advisor, Dr. Frederick P. Brooks, Jr. My understanding and

analysis of numerical issues involved in SVD benefitted from discussions with Dr. James Coggins and Dr. Steve Pizer. Thanks go to Lakshmi Dasari for help in preparation of this article. And as always, this work was influenced both in the large and small as a result of numerous discussions with my colleagues in the UNC-CH Computer Graphics Lab.

References

- [Dongarra79] J. Dongarra, C. Moler, J. Bunch, G. Stewart, "Linpack User's Guide", Society for Industrial and Applied Mathematics, Philadelphia, 1979
- [Dongarra87] J. Dongarra, E. Grosse, "Distribution of Mathematical Software via Electronic Mail," *Communications of the ACM*, 30(5), May 1987
- [Duff80] T. Duff, "The Mat Manual, Second Edition", NYIT Computer Graphics Lab, July 1980
- [Golub71] G. Golub and C. Reinsch, "Singular Value Decomposition and Least Squares Solutions", Contribution I/10 in *Linear Algebra: Volume II of Handbook for Automatic Computation*, Edited by J. Wilkinson and C. Reinsch. Springer-Verlag, 1971
- [Gomez84] J. Gomez, "Twixt: A 3-d Animation System", *Proceedings of Eurographics '84*, Elsevier Science Publishers, 1984
- [Greene83] N. Greene, "Transformation Identities", Personal communication
- [Greene86] N. Greene, "Extracting Transformation Parameters from Transformation Matrices (Extended Abstract)", Personal communication
- [MacSYMA79] "MacSYMA Reference Manual", The Mathlab Group, Laboratory for Computer Science, MIT, Cambridge, MA. 1977
- [Press86] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, "Numerical Recipes", Cambridge University Press, 1986
- [Press88] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, "Numerical Recipes in C", Cambridge University Press, 1988
- [Shoemaker85] K. Shoemaker, "Animating Rotations with Quaternion Curves", *Computer Graphics* 19(3), Proceedings of Siggraph '85, July 1985
- [Stern83] G. Stern, "Bbop - A System for 3-d Keyframe Figure Animation", Siggraph '83 Course Notes, Introduction to Computer Animation, 1983

Late Binding Images

Abstract

We have developed a high-quality Z-buffer based rendering system which allows users to quickly change the surface properties and illumination of objects in a 3d scene. We use standard scan-conversion techniques for each primitive to produce a surface description packet for each relevant pixel on the screen. When rendering is complete, these packets are depth-sorted at each pixel and stored on disk. The user then binds shading coefficients, textures, and colors to the objects, and also positions light sources to illuminate the scene. Because the expensive scan conversion step has been separated from the shading calculation, we can produce images with new surface properties and illumination more quickly than with a standard rendering pipeline. Storing all potentially visible surfaces at a pixel also enables us to support transparency.

The system is easily described with a group algebra, which supports the implementation, provides a convenient user interface, and also serves as a succinct but exact manual of operation. We have used the system extensively for almost two years and have found it to be a robust and practical tool.

Introduction

Part of our ongoing research at UNC-Chapel Hill is the application of computer graphics to medicine. In particular, we work on developing tools to assist physicians with the interpretation of radiographic images (such as those obtained from a CAT or NMR scanners), as well as planning radiation therapy (such as burning away a tumor with focused beams of radiation). Some of our tools create a 3d database of surface primitives which we then render with traditional shaded surface techniques. For these images to be of value to the physician, they must be of high quality, free of rendering artifacts, and quickly produced.

For the last few years we have created our shaded images using rendering programs that we already had running in the lab. But we recently decided that we needed to generate our final, medically useful 3d images more quickly. We felt that the best way to make pictures faster would be to speed up those portions of the image generation pipeline where users were spending most of their time. I watched users prepare several complete radiological studies, and noted which steps were the slowest.

The first step was viewpoint selection. The user loaded a small subset of the database into a program that could generate near-real-time images of scenes composed of several hundred polygons. The user then interacted with that program to select an eyepoint, gaze direction, clipping planes, and perspective information.

This viewing information was then fed into the polygon rendering system. This program produced images at a fixed 2048-by-2048 resolution (filtered down to 512-by-512 for display). Users would typically run this expensive rendering program over and over again, changing the surface parameters of the objects and the positions of the lights between runs. It turned out that although specular highlights were desirable in the final images, highlights on a transparent surface often occluded important details behind it; this meant the lights had to be moved and the image re-rendered. The degree of transparency of different objects would also have to be adjusted time and again; our images typically have several transparent objects nested within one another, and too much or too little transparency on any one surface can ruin the illusion of nesting. Colors and surface reflectivity coefficients were also adjusted, in response to the changes in lighting and transparency.

This rendering program was quite slow; it often took many hours to complete an image. When I observed users running the program repeatedly to tune colors, surface properties, and illumination parameters, it became clear that this iterative re-rendering was the slowest step in our system.

From these observations we constructed the goals for our new rendering system. We knew that our users did not change the viewing information between renderings; we wanted to exploit this fact. Given a viewing transformation, we wanted to be able to quickly change the illumination and all surface characteristics of all elements in the scene - this would be the essential new feature of our system. Additionally, the system should be capable of handling large databases (many tens of thousands of primitives), a wide variety of primitives (not just polygons), and transparency.

Although our system was motivated by the medical application discussed above, this list of criteria is independent of its medical origins, and the result was a general purpose renderer.

Our solution was to separate the scan conversion of an image from its display. Given a viewpoint, the database is scan converted, with the information produced by the scan converter placed into a disk file. To display the image, the user specifies surface attributes and light sources, and then shades the image. The user can then adjust the illumination and any surface properties and re-render the image quickly, since the very costly scan-conversion step has been eliminated from each iteration of the cycle. Because the surface and lighting attributes of an image are bound to objects after scan conversion, we have named the rendering system the Late Binding Image, or LBI, renderer.

Previous Approaches

The literature contains several previous approaches to generating images with different surface parameters without necessarily repeating the scan conversion. A common theme is to encode surface information into frame buffer pixels, and then adjust the colormap to achieve the desired effects [Shoup79].

For example, [Holmes85] describes a system where the user could interactively change the apparent direction of a light source illuminating a shaded scene.

The problem with such approaches is the quantization due to storing bulky geometric information in memory designed to hold color information. One simply runs out of bits very quickly. Additional hardware enhancements like crossbars [Ikonas82] and wide colormaps can ameliorate the problem, but they still cannot provide us with enough bits to encode all the geometric information we need, since we must see color, transparency, and localized highlights simultaneously.

[Perlin85] describes a system for exploring textures based on manipulation of pixel streams. Scenes were built from 2d digital composition, not rendering, so it would be difficult to provide for more than one surface at a pixel.

The LBI Process

In this section we give an overview of the process of building and modifying LBI images (see Figure 1). The essence of the project is that we have separated the scan conversion step from the rest of the image generation pipeline. After scan conversion, the system performs specular shading and then supersampled Z-buffer hidden surface removal, with transparency.

The first step in building an image is the selection of the viewing parameters. The user specifies an eyepoint, a viewing angle, gaze direction, an up vector, the desired size and location of the image on the screen, the size and location of a viewing window into that image, and whether or not the final image should be anti-aliased.

The viewing parameters and database of surface primitives are then given to the viewing transformation program, whose output is sent to the scan conversion program, where each primitive is scan converted using standard techniques. We must be sure that this step generates all the information we will need further down the image generation pipeline. To accomplish this the scan converter generates a list of packets for each pixel, one packet for each surface. If more than one surface is visible at a particular pixel, the system maintains all of the packets at that pixel in a list.

The Packet Structure

The first part of each surface description packet is an object tag: a small integer that specifies to which larger, composite object this primitive belongs. This tag is given to the scan converter as part of the surface description. All polygons that are part of a lung, for example, will have the same object tag; all patches that are part of the nearby heart will share a different object tag. In general, each surface to which we want to assign unique surface properties will have a unique object tag.

Let us assume for a moment that we have turned off anti-aliasing and texturing in the interests of speed. Then for a given primitive, the scan converter will only produce packets for pixels whose centers are covered. For such pixels the scan converter will produce a Z depth and a surface normal for that primitive at the center of the pixel. The tag, depth, and

normal comprise the complete packet for an image with no anti-aliasing or textures; see Figure 2a.

If anti-aliasing is enabled, we also include a coverage bitmask as in the A-buffer [Carpenter84]. The surface normal and depth are computed at the pixel center; if the primitive does not cross the pixel center we estimate the surface normal and Z depth as if it did, by extending the geometry of the surface. Figure 2b shows a packet including this bitmask. As in the A-buffer, packets which have a common tag and Z but disjoint bitmasks are merged into a single packet by OR'ing their bitmasks, and averaging their colors weighted by relative coverage.

Texture information is included by adding a texture list to the packet. The texture list is preceded by a texture count, stating how many textures are to follow. Textures are saved as the center and side lengths of the smallest box in texture space containing the projection of the associated pixel. Each individual texture is represented by a texture tag, and the coordinates of the texture space box. If there are several textures, they are simply listed one after the other, as in Figure 2c.

After all primitives have been scan converted, the packets at each pixel are depth sorted, merged if necessary, and then written to disk.

Image Generation

The next step is to shade the surfaces at each pixel. Shading requires going through each packet list at the pixel, evaluating the incident and reflected light, and computing (perhaps partially) hidden surfaces.

To compute the shade we use a slightly modified version of Phong's shading model. We specify a surface by color (RGB), transparency, diffuse reflectivity, specular reflectivity, and highlight exponent. Surfaces are defined in one of the input files to the shader, called the Surface Property Binding (SPB) file, which can specify a unique surface description for the inside and outside of every object. The SPB file contains a list of the 14 surface properties (7 each for inside and outside) for each object tag in the scan converted file, as well as a background color for the object world. If we have textures in the system, the SPB file associates each texture tag with an actual texture (by giving its file name), and a texturing operation (bump, color, transparency, diffuse or specular reflectivity, or highlight exponent). Thus, it is perfectly possible to apply a transparency map, a picture map, and a bump map to the same primitive.

The other input to the shader is a list of the (infinitely far away) light sources in the Illumination Binding (IB) file. Each light is specified by its color and direction of

illumination. As an aid to the user, the direction may be specified in either object space or screen space. The light color is modulated by a single intensity parameter, which scales the color of the light more conveniently than by adjusting the three RGB color values. The color of an ambient light source may also be specified.

To shade a pixel, we start with the nearest packet, and search in increasing Z for a primitive that is fully opaque and covers the pixel. We search until we find such a primitive or reach the end of the list (an opaque, fully covered pixel of the background color is assumed at the end of each list). We then work backwards, packet by packet, returning to the head of the list. At each step we first shade the current primitive (this includes texturing). If we are anti-aliasing with bitmaps, we scale the color contribution of this pixel by the coverage of its bitmap. We then adjust the color by the primitive's transparency factor and add in any light passing through the surface from behind. We then step back to the next nearest packet in the list, incrementally accumulating the final color for the pixel.

The Algebra of the LBI Group

The LBI rendering system may be usefully described with a simple group algebra. The advantages of writing this algebra are threefold: first, the requirements of the algebra guided our selection of data structures for implementation of the system; second, the algebra provides a simple and consistent user interface for describing operations on LBI files; third, the algebra provides an exact, unambiguous description of what user actions will do and what the result will be. In effect, the algebra is a complete and precise user's manual for the file composition part of the system.

The elements of the algebraic group are the LBI files themselves (the files produced by the scan conversion process). These files contain a depth-sorted list of packets at each pixel. Operations on LBI files cause operations on their lists, so we will focus our attention in this section on these lists and the individual packets they contain. Nevertheless, it is the files themselves that are the group elements.

The group operation is file addition. When two files are added, the sorted lists at each pixel are merged and the result is sorted by Z. If two individual data structures are identical except for Z depths of opposite sign, they cancel and both data structures are eliminated from the sum. A data structure with a Z depth of 0 would be exactly in the image plane, and we must make some arbitrary decision about how to handle it. We will see in a moment that a good choice is to simply ignore such data structures: they never appear in the

result of an LBI operation. If any pixels in the file have no list associated with them, we define that empty list to be equivalent to a data structure with every field equal to 0.

We now see why we chose to ignore data structures with a depth of 0: they are our identity element. More precisely, an LBI file with every pixel containing a data structure at a Z depth of 0 (or, equivalently, no pixel lists at all) is the identity, since adding it to any LBI file leaves that file unchanged.

Since two packets that are identical except for opposite depths (one positive, one negative, of equal magnitude) cancel and leave behind nothing (or an empty list, equal to the identity), two such data structures are inverses of each other. Thus the inverse of a complete LBI file is another LBI file with the same contents, except every packet has a Z value of opposite sign.

Clearly the sum of two LBI files results in another LBI file: it may only contain a list of packets at each pixel. Thus file addition is a binary operation, and the set of LBI files is closed under file addition.

Because the contents of individual packets in a pixel list are never altered (they may be entirely deleted, but the values are never changed), imprecisions of computer arithmetic do not affect the associativity of LBI operations. If Z values are stored as exactly representable integers then all comparisons are associative; thus file addition is also associative. Incidentally, the act of binding and displaying an LBI file is considered an interpretation of that file, and does not enter into the group description.

We have shown that the set of LBI files under file addition forms a group: every element has an inverse, the group has an identity, it is closed, and it is associative. It is interesting to note that file addition is also commutative.

This algebra gives us a simple user interface, a clean description of the system, and an unexpected, powerful operation: object subtraction. Let's say that several LBI files have been added together into one, very large file containing many objects. Upon rendering, the user decides that one or more objects are not needed in this particular image.

One way to remove those objects is to make them fully transparent. The drawback here is that the specification of scene attributes followed by rendering is an iterative process repeated many times. Each time the image is bound and displayed those transparent objects will still need to be processed when computing shading and visibility, costing some time. If this time is large then we will have some significant overhead in repeatedly waiting for the processing of invisible objects! The image will be correct, but slow.

A faster alternative is to subtract the unwanted objects from the LBI file. The user simply adds to the composite file the inverses of the original LBI files of the unwanted objects. The result is a new composite LBI file different from the first only in lacking the

undesired objects. Figure 3 illustrates another situation where the LBI algebra proves useful.

From this discussion we can see one useful way to use the system. A user scan converts all objects that might be in the final image, placing each object into a separate LBI file. These files are then added and subtracted to build a composite LBI file containing only the desired objects. This approach of adding and subtracting scan-converted (but unshaded) object may be considered a kind of 3-dimensional "matting" operation [Porter84], [Duff85]. We thus also have the unusual "un-matting" operation supported by the subtraction operator.

An issue on the border between theory and practice is repeated elements: what is the result when two identical LBI files are added together? There are many solutions that preserve the group properties. We chose to accumulate both files, in exactly the same way as if they were different. Thus if the user takes a composite file, adds in some new file three times, and subtracts it twice, the result still contains one copy of the new file.

Since the operations of image addition and subtraction are associative, the user is freed from any concern about the order of the addition and subtraction operations used to build an image.

The implementor may augment and enhance the LBI algebra with other operators, applied immediately to a file given as an argument. For example, such operators may implement windowing and clipping (applied to intensity, depth, and the screen image), or the simulation of fog (depth cuing).

Implementation

In this section we will describe the practical choices we made in our implementation of this system. The choices were influenced by the nature of our computing environment and the needs of our users. In fact, we have written two complete systems. The earlier system does not include anti-aliasing or textures, and has a primitive user interface. This system has been in regular use for almost two years, and has proven robust as our standard renderer. Our more recent research system supports anti-aliasing, textures, and a more complete algebra as described above.

Our main rendering hardware consists of a VAX-11/780 with 6 megabytes of processor memory. The VAX runs UNIX 4.2 BSD, which incorporates virtual memory. We also have an Ikonas/Adage RDS-3000 bit-slice microprogrammable graphics engine and frame

buffer. The Ikonas is supported by the gia2 language [Bishop82] and various local libraries and debugging aids [Glassner86].

Scan Conversion

We wrote a new polygon scan converter specifically to produce LBI files in a fast and efficient manner. An important consideration was the effective allocation of storage for the packet lists at each pixel. The simplest technique would be to allocate a 2d array equal to the size of our picture times the size of a packet, and place the first element in each list into this array, as in Figure 4a. Subsequent packets in the list would be dynamically allocated. The list would be singly- or doubly-linked to ease the depth sorting, though those links would be discarded when the writing the file to disk.

Unfortunately, this straightforward approach is very inefficient. The VAX page size is 512 bytes. A typical picture in our environment is 512-by-512 pixels. We know that polygons can arrive in any order, starting at an arbitrary scanline on the screen. Thus, even if our 2d array consisted solely of (4-byte) packet pointers, in a simple 2d array each time we move to a new scanline we would get a page fault, which would cost us time.

A better organization is to arrange the virtual frame buffer into a 2d array of smaller 2d arrays of pointers, each 16 by 8, as in Figure 4b. Since this smaller array contains 128 entries, and each pointer consumes 4 bytes, exactly one of these smaller arrays fits on each page. Thus, if a polygon is enclosed within a bounding box of 45 pixels wide by 45 pixels high, we will at worst have 18 page faults, instead of 45. This is illustrated in Figure 5. We have observed typically about an 8-fold reduction in page faults when rendering many small polygons.

Each entry in this 2d virtual frame buffer is a pointer to a data structure created at the same time that the pointer is allocated. We do use forward and backward-pointing links to ease the job of sorting the list when we create the output file. When we need to create additional pointers and storage for new packets, we create entire 16-by-8 blocks at a time. We try to keep all the packets for a given polygon within the same block, which can sometimes cause small holes in our packet list. The scan conversion is faster due to this technique, but memory consumption is slightly increased (typically about 4-6% for our images). This statistic is kept down by searching for an appropriately sized, existing hole for a polygon before allocating new memory for it.

File Format and Shading

The LBI file itself begins with a header, which provides information about the file for the system and the user. The header contains text fields containing creation time and date, and any descriptive text the user wishes to associate with the file. There then appear fields containing the image origin and window size, the smallest and largest Z values in the file, the smallest and largest object tags in the file, and flags indicating the presence of anti-aliasing and texture information.

Each of the individual data structures uses 8 bits (1 byte) for the object tag. This limits us to a maximum of 256 individually bindable objects in any LBI file. In our environment, this is a reasonable upper bound.

We store the Z depth as a 16 bit integer (recall the identity value $Z=0$).

Surface normals are used to compute shading. The diffuse shading from a collection of infinite light sources may be looked up in a table indexed by two of the (normalized) surface normal components [Greene86]. The illumination color from the table is then scaled by the diffuse reflection surface color of the object. Specular shading is handled with another set of tables, one for each light source. All of these tables are built by the binder, which knows both the eye position and the light directions. The values in the specular tables are exponentiated appropriately for each surface when we shade.

Our light tables are 64 by 64, and provide good shading if the values are linearly interpolated. Experimentation with real images has convinced us that we routinely produce images for which 32 steps of interpolation is insufficient to prevent contours and Mach banding. We thus compute the two light table addresses (derived from the normal components) to 12 bits (6 bits each of integer and fraction), but store these addresses in the least significant bits of a pair of bytes; this keeps us nicely aligned on byte boundaries. In an efficiency hack, we encode the sign of the Z component of the surface normal in the high-order bit of the Y address.

Anti-aliasing requires adding a coverage bitmap to every data structure. We have found that we get good results with a bitmap that is 8 bits wide by 4 bits high (as in the A-buffer), requiring 4 additional bytes per data structure.

Texture information is more complex. Our research system uses a variety of system-dependent hacks to reduce storage. In general, each primitive will require one byte for texture count. Then each texture needs a map tag (1 byte), U and V indices (2 bytes each), and texture sample width and height values (2 bytes each), for a total of 9 bytes per texture.

Thus the full LBI file contains 12 bytes per primitive per pixel, plus another 9 bytes for each texture applied to that primitive at that pixel. Without anti-aliasing or texturing, each LBI packet contains 7 bytes.

The actual scan conversion of surface primitives into LBI files is performed by a C program on the VAX. Binding is performed by Ikonas microcode.

Results

Figures 6, 7, 8, and 9 shows various models rendered with the LBI system. Each model was scan converted once, and then shaded with a variety of surface parameters. Figure 6 shows several heads seen from above, built from radiographic data. Because the CAT scans did not continue all the way to the top of the skull, the top of the brain is missing; this makes the brain appear as a donut when seen from above. The rows show the head and brain with different colors and reflectivities, the columns show different amounts of transparency.

Figure 7 shows four icosahedra, tessellated with a variety of tiles.

Figure 8 was built from NMR scans of the pelvis of a 59 year-old man. The main visible bones are the pelvic girdle and the spine. At the lower left and right the tops of the femurs are shown where they fit into the pelvis. The bladder is yellow, the prostate is purple, and the rectum is colored red. The bladder should have a rounder shape; it has been distorted by cancer. This study shows a possible treatment of the bladder by radiation therapy. The dark shape surrounding the internal organs is a 90% isodosage surface. The radiation has been shaped to also irradiate the lymph system, above the bladder but not rendered in this image.

Figure 9 shows two views of a section of a male chest containing a stomach, spine, esophagus, lung, skin, bones, and 16 different radiation isodosages computed for treating a tumor on the esophagus. In each of these images one of the isodosage contours has been made opaque, and the others have been left transparent.

The LBI rendering system has been in daily use since May 1986. In that time it has produced many images useful in the medical imaging research we pursue with our radiotherapy colleagues. The system has also been used to produce several animations.

The production of an animation of a rotating pelvis with internal bones and organs identified an interesting use of the LBI system. The script for the film (each frame of which was rendered at 512-by-512 pixels) specified a spinning pelvis, displaying just the bones. Then various organs would fade in, appear for several cycles of rotation, and then

fade out, to be replaced by other collections of organs. Thus, viewers could see the relationships of various body parts with other parts by viewing a succession of small collections of the objects. The plan was to scan convert the spinning pelvis once with all the objects. Then the different parts of the film could be generated by making the unwanted parts partially transparent (when fading in and out), or by subtracting them entirely from the file (when invisible). Thus the entire film could be generated by one set of scan conversions, and then many bindings. Unfortunately, because of the size of the rendered image the data files grew beyond the virtual memory limits of our machines, so each frame was individually scan-converted and rendered.

By reducing the image size and thus virtual memory consumption, this approach was used successfully to create the film shown in Figure 10. In this film we see 4 different views of the same chest as in Figure 9. Reading from left-to-right, top-down, we see increasing levels of radiation dosage converging on the esophagus (the three beams are most clearly visible in the upper-right image of each set). This film was made from only 4 LBI files, one for each view. By keeping the image size to 256-by-256, we were able to hold all of the internal structure and radiation contours in virtual memory simultaneously. Each of these files was then bound 16 times, each time with different surface parameters, to produce the animation.

It is interesting to note the performance of the LBI system for these frames. Our old rendering system required an average of 105 minutes of system time to produce each individual image (about 58,000 polygons each). The LBI system scan-converted each of the 4 views in about 17.2 minutes, and produced a bound image in about 1.1 minutes.

Thus, creation of the 64 images by the old system would have required $64 * 105 = 6720$ minutes, or about 4 days and 14 hours. The LBI system required $(4 * 17.2) + (64 * 1.1) = 139.2$ minutes, or just over 2 hours and 20 minutes, which shows dramatic improvement. Note that these are system times; our clock times can be found by multiplying by 3.2, compensating for the performance degradation on our heavily time-shared machine.

Encouraged by the success of this "one-scan, many-binds" approach, we would like to modify the scan converter so that the entire image need not reside in virtual memory at once. Instead, full (or nearly-full) packet blocks will be written to disk and their memory freed. This will require some pre-sorting of the polygons after the viewing transformations, but will enable us to generate images with many more primitives at high resolutions, limited only by disk capacity.

Discussion

An interesting comparison between the LBI technique and the radiosity approach to image synthesis [Immel86] was suggested by Marc Levoy [Levoy86]. Radiosity effectively pre-computes the shading in a 3d scene, but calculates visibility on the fly. This allows it to generate multiple viewpoints of a fixed model quickly. Our technique pre-computes visibility, but calculates shading on the fly, allowing the fast selection of shading parameters. The two techniques effectively span a wide range of applications, each focusing on one end of a well-defined spectrum.

Our final medical images are now produced in far less time than required by our more traditional image rendering tools. Each medical image must be carefully (and usually iteratively) tuned to emphasize important structures and guarantee that all objects are easily differentiable. The fast generation of images with adjusted surface properties and illumination has provided an effective solution to this otherwise time-intensive problem.

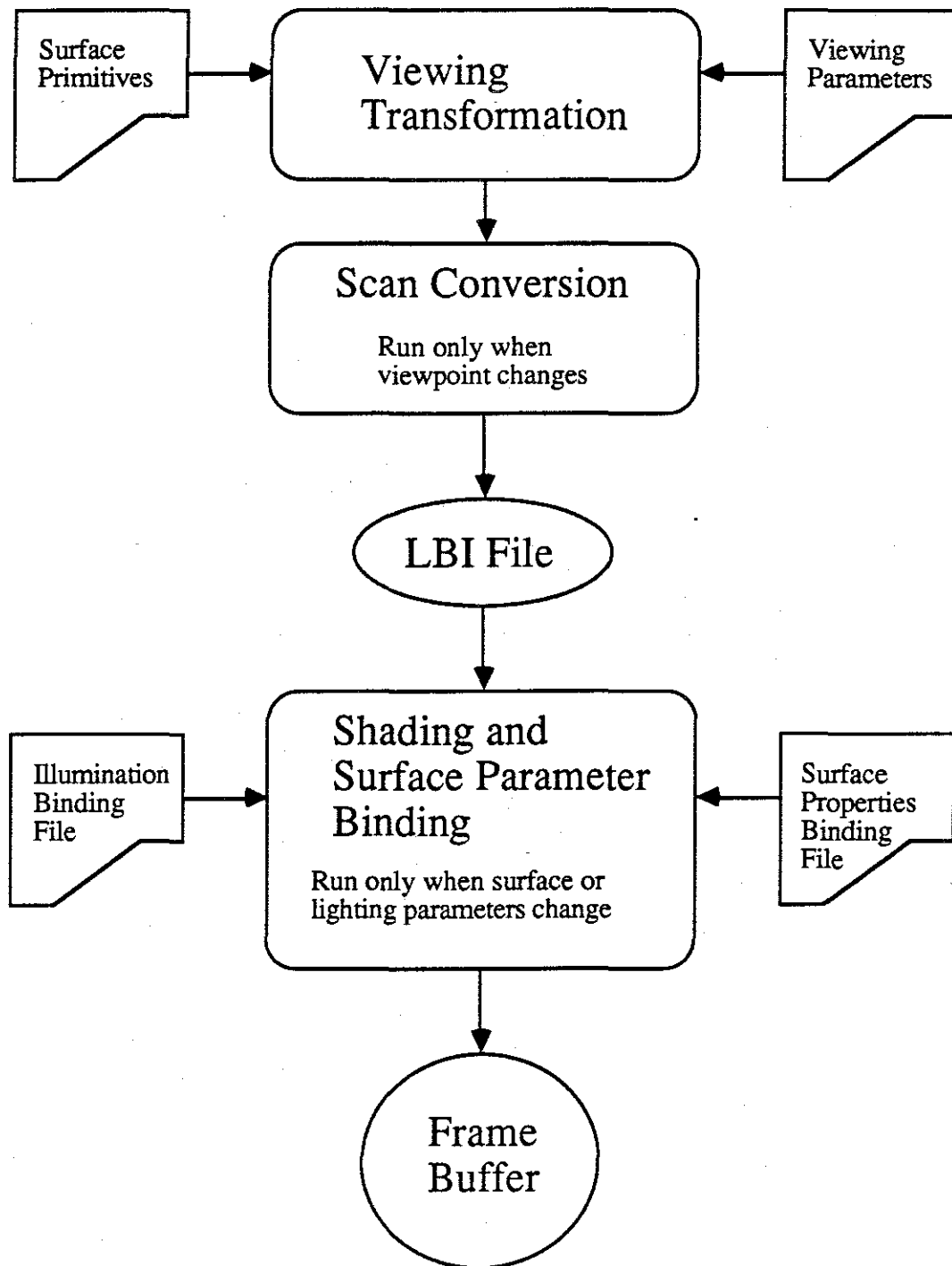
The LBI rendering system has shown itself to be robust, useful, pleasant to use, and fast. In addition to medical imaging, we have used the system for many standard computer graphics projects, including the display of curved surfaces and solid modelling.

Acknowledgements

My thanks go principally to Drs. Henry Fuchs and Stephen Pizer, who supervised this work. An early version of this paper was reviewed by John Gauch, Marc Levoy, Chuck Mosher, and Lee Westover, whom I thank for their insightful and constructive comments. Many persons in the Department of Radiation Therapy were helpful in this paper and this project. Figure 9 was generated by Chuck Mosher and Kevin Novins. Figures 10 and 11 were generated by Andy Skinner. Thanks also go to our medical colleagues Drs. Chaney, Rosenman, and Sherouse for their encouragement and support while this system was being designed and built. I would also like to thank the anonymous reviewers for their helpful comments. This work was performed under the Med3d program, supported by NIH Grant R01-CA39060.

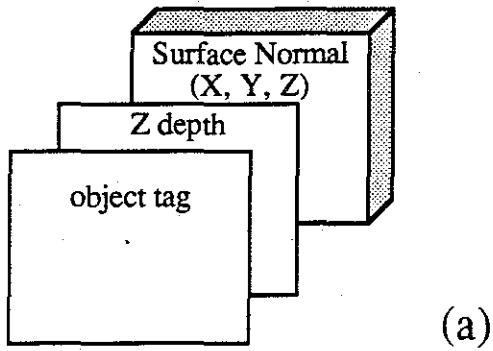
References

- [Bergman86] Bergman, L., Fuchs, H., Grant, E., Spach, S., "Image Rendering by Adaptive Refinement", *Computer Graphics* **20**, 4, August 1986
- [Bishop82] Bishop, G., "Gary's Ikonas Assembler Version 2", UNC-CH Computer Science Department Technical Report, June 1982
- [Carpenter84] Carpenter, L., "The A-buffer, an Antialiased Hidden Surface Method", *Computer Graphics* **18**, 3, July 1984
- [Crow77] Crow, F., "The Aliasing Problem in Computer-generated Shaded Images", *Communications of the ACM*, **20**, 11, November 1977
- [Duff86] Duff, T., "Compositing 3-D Rendered Images" *Computer Graphics* **19**, 3, July 1985
- [Glassner86] Glassner, A., "Ikonas Utilities Package, Release 1.2," University of North Carolina at Chapel Hill, Chapel Hill, NC.
- [Greene86] Greene, N., "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics & Applications*, **6**, 11, November 1986.
- [Ikonas82] RDS-3000 Programming Reference Manual, Adage Inc., Billerica, MA. June 1982.
- [Holmes85] Holmes, D., "Three dimensional Depth Perception Enhancement by Dynamic Lighting," Master's Thesis, Department of Computer Science, UNC-Chapel Hill, 1975
- [Immel86] Immel, D., Cohen, M., and Greenberg, D., "A Radiosity Method for Non-Diffuse Environments," *Computer Graphics* **20**, 4, August 1986
- [Levoy86] Levoy, M., private communication, December 1986
- [Newman79] Newman, W., and Sproull, R., "Principles of Interactive Computer Graphics, 2nd edition", McGraw Hill 1979
- [Perlin85] Perlin, K., "An Image Synthesizer," *Computer Graphics* **19**, 3, July 1985
- [Porter84] Porter, T., and Duff, T., "Compositing Digital Images", *Computer Graphics* **18**, 3, July 1984
- [Shoup79] Shoup, R., "Color Table Animation", *Computer Graphics* **13**, 2, August 1979

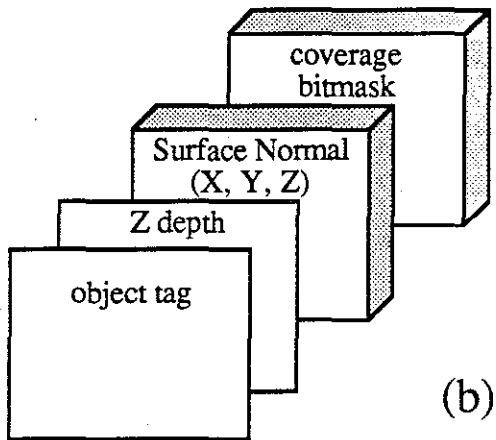


The LBI Renderer Pipeline

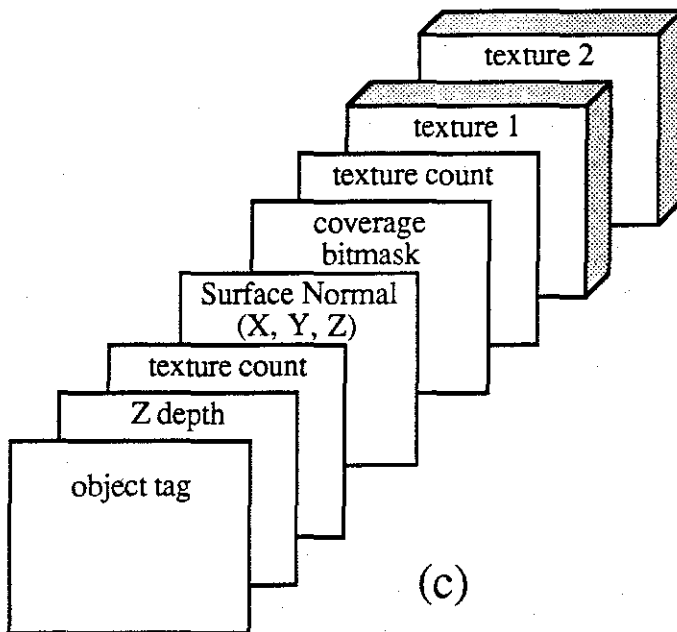
Figure 1



A simple
LBI packet.



A packet with a
bitmask for
anti-aliasing.



A packet with
bitmask and
texture
information

Figure 2

These examples show the construction of two composite LBI files in a medical environment. We assume that a scan conversion program has produced LBI-format files called skin, bones, heart, lungs, and stomach. The first image we want to produce will contain the skin, bones, heart, and lungs:

```
file1 = skin + bones + heart + lungs
```

When we first build the second image we specify that we want the skin, bones, heart, and stomach. One approach is to build a new composite file from these three smaller files:

```
file2 = skin + bones + heart + stomach
```

A better approach is to take the first composite file, subtract the lungs, and add the stomach:

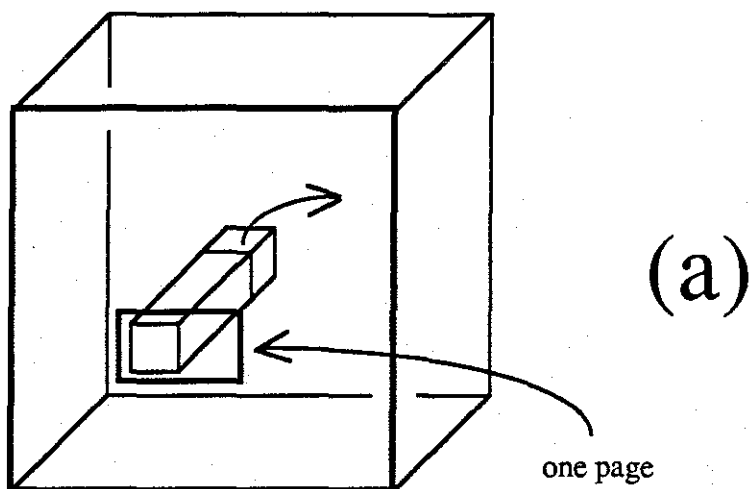
```
file2 = file1 - lungs + stomach
```

Let's now say that we want to see the tumor unobscured by the lungs: no lung data closer to us than $Z=3000$ should be rendered. We also want the skin to appear only above and below the lungs (scanlines 100 and 350, respectively). File windowing commands can help us accomplish this goal

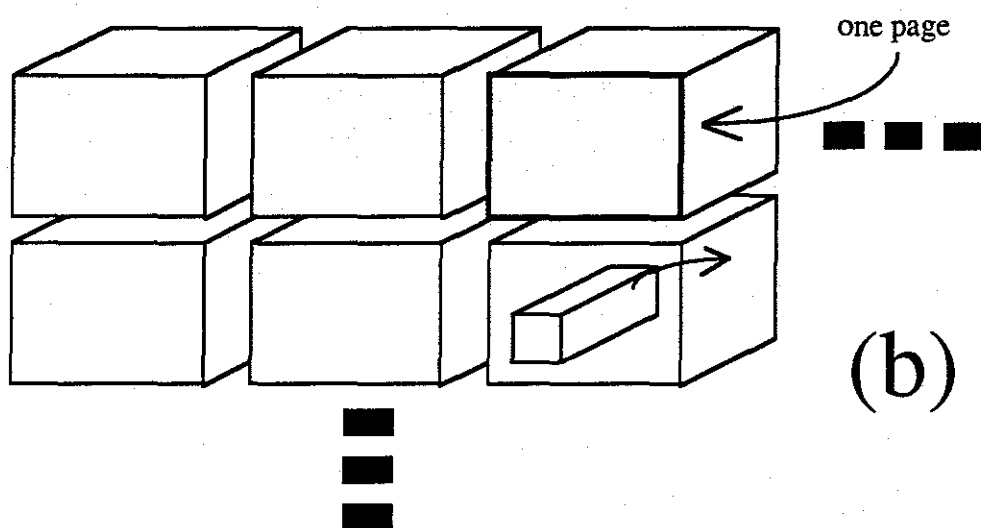
```
file3 = bones + tumor + hitherClip(lungs,3000) +  
topClip(skin, 100) + bottomClip(skin, 350):
```

Examples of LBI Commands

Figure 3

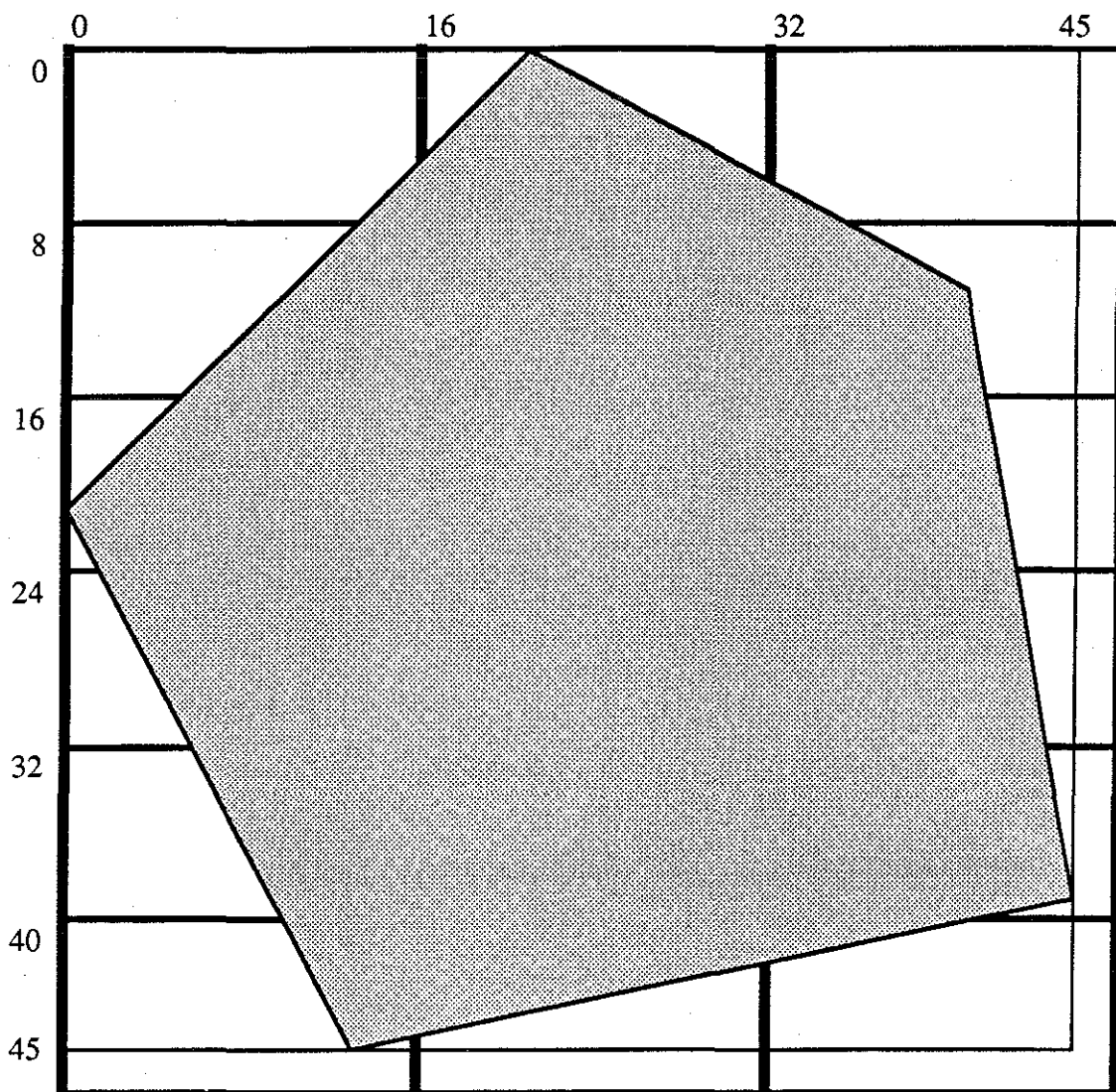


Here is a virtual screen that contains packets. The screen is a rectangular array large enough to hold one packet and a pointer to the next packet at that pixel, which is located elsewhere in memory. Because very few of these packets can fit in a page, polygons which cover several pixels horizontally will page fault on every new scanline.



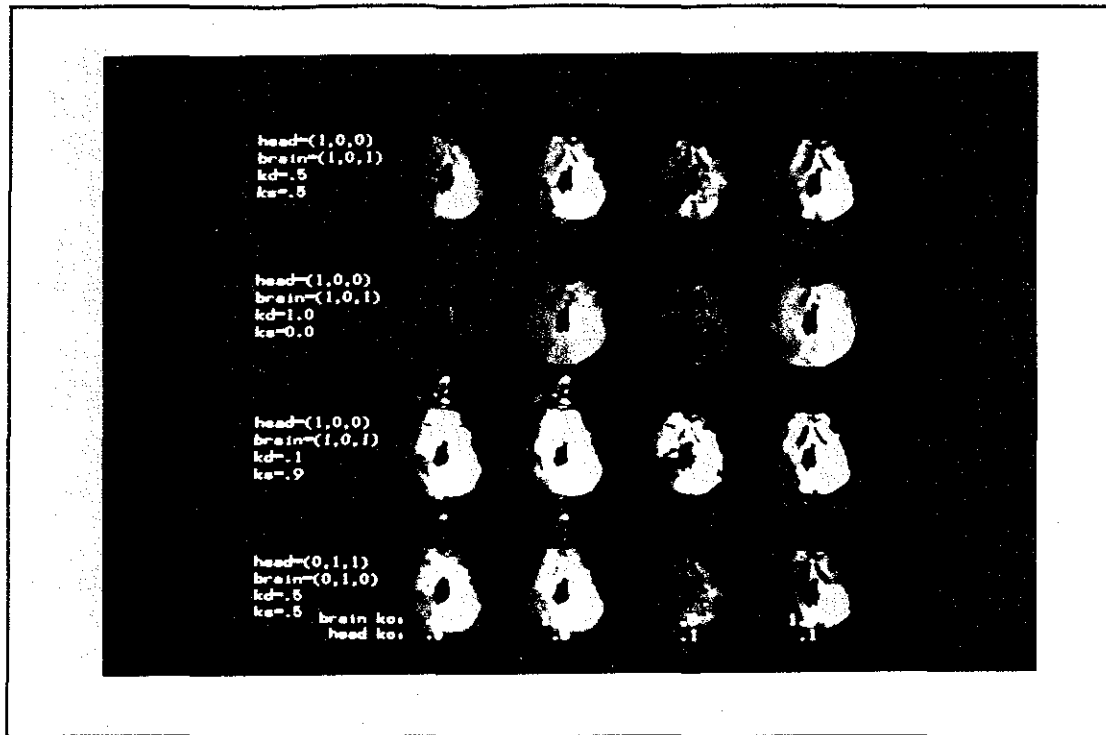
Here we've allocated the screen as blocks of pointers. Each block fits onto one page. When a primitive is scan converted, transitions from one scanline to another in the same block don't cause page faults. The actual packet records are allocated in a similar manner.

Figure 4



A polygon with a 45-by-45 bounding box, cornered on the origin of a page. In this example, we get 18 pages faults when finding the pointers for the packet lists (18 blocks of pointers are accessed). If we allocated the memory as a single large 2d array, we'd get 45 faults, one for each scanline.

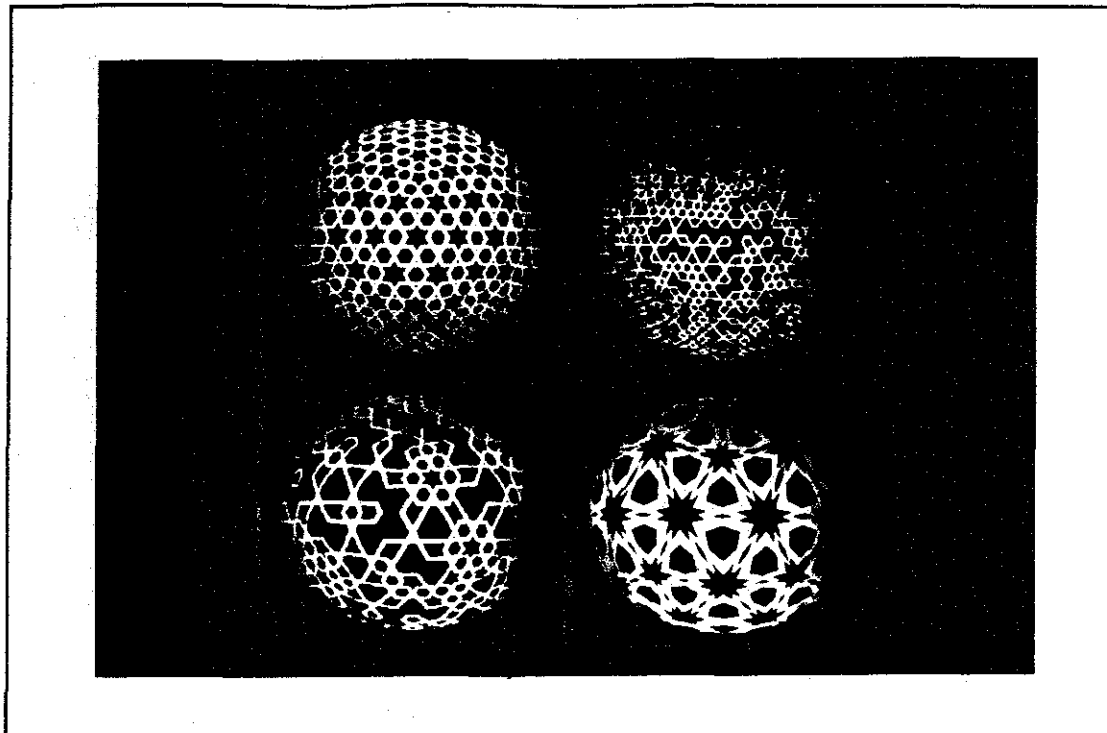
Figure 5



A head and brain, from CAT scan data. The colors, diffuse reflection, and specular reflection change across the rows. Opacity changes across the columns.

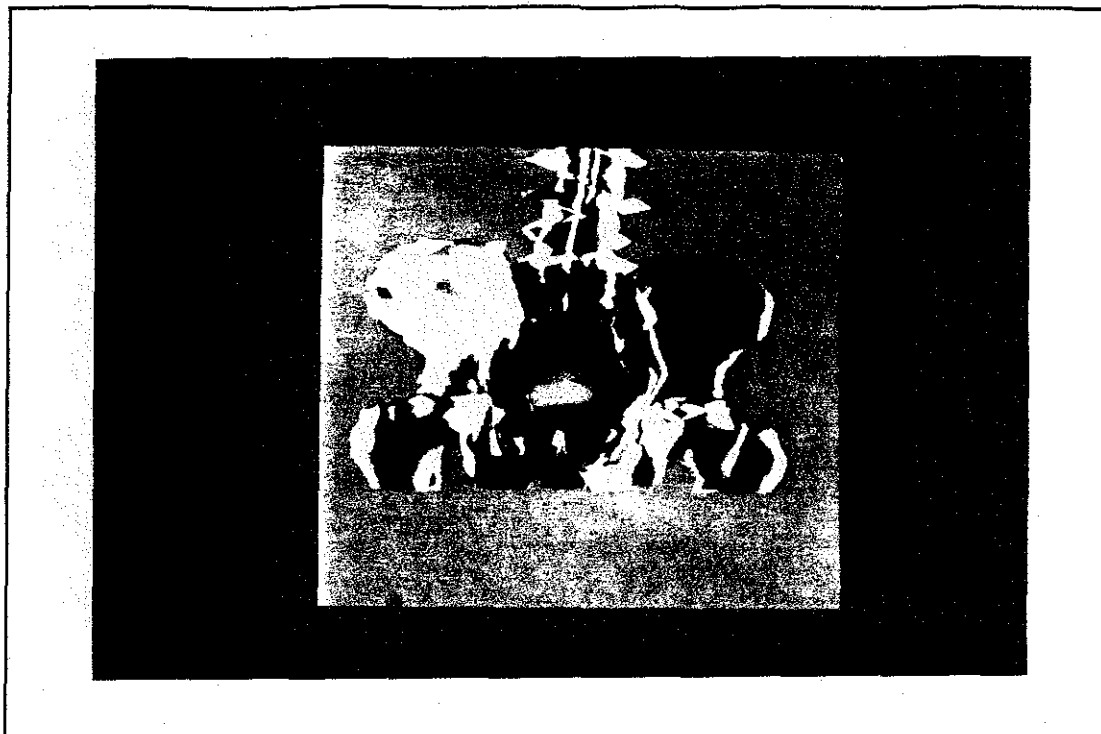
The hole in the brain is in the data, because the scanning process was stopped before the entire head had been scanned.

Figure 6



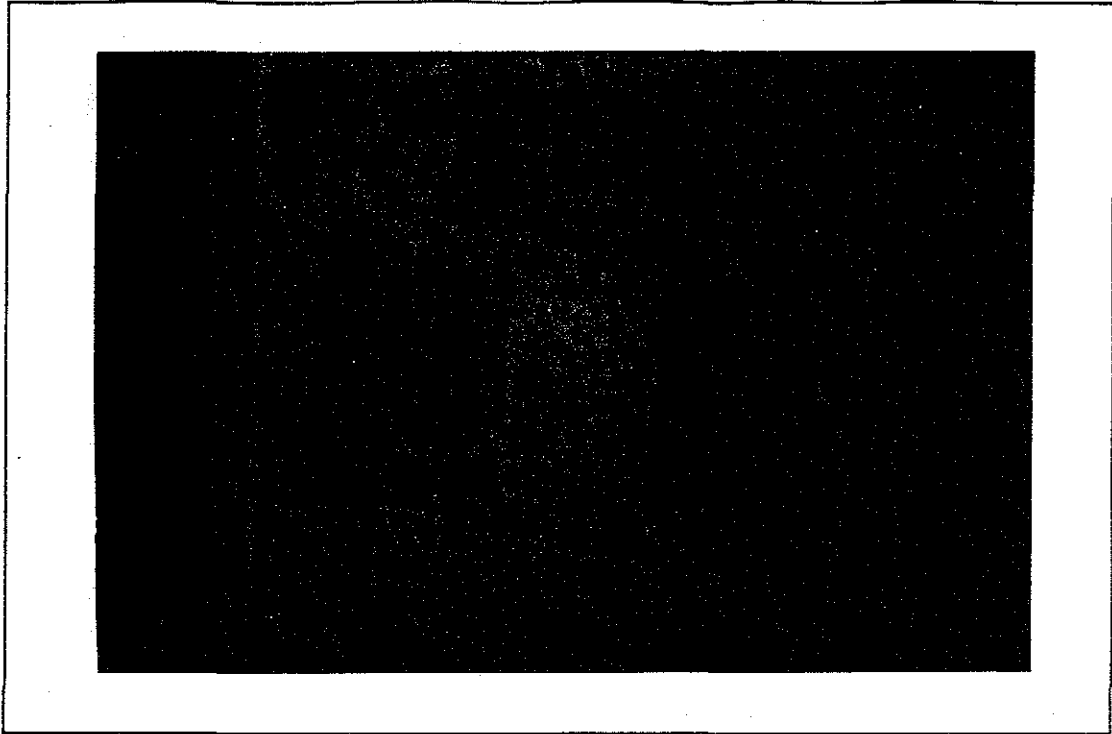
Several tiled icosahedra rendered with polygons, illuminated by colored lights.

Figure 7



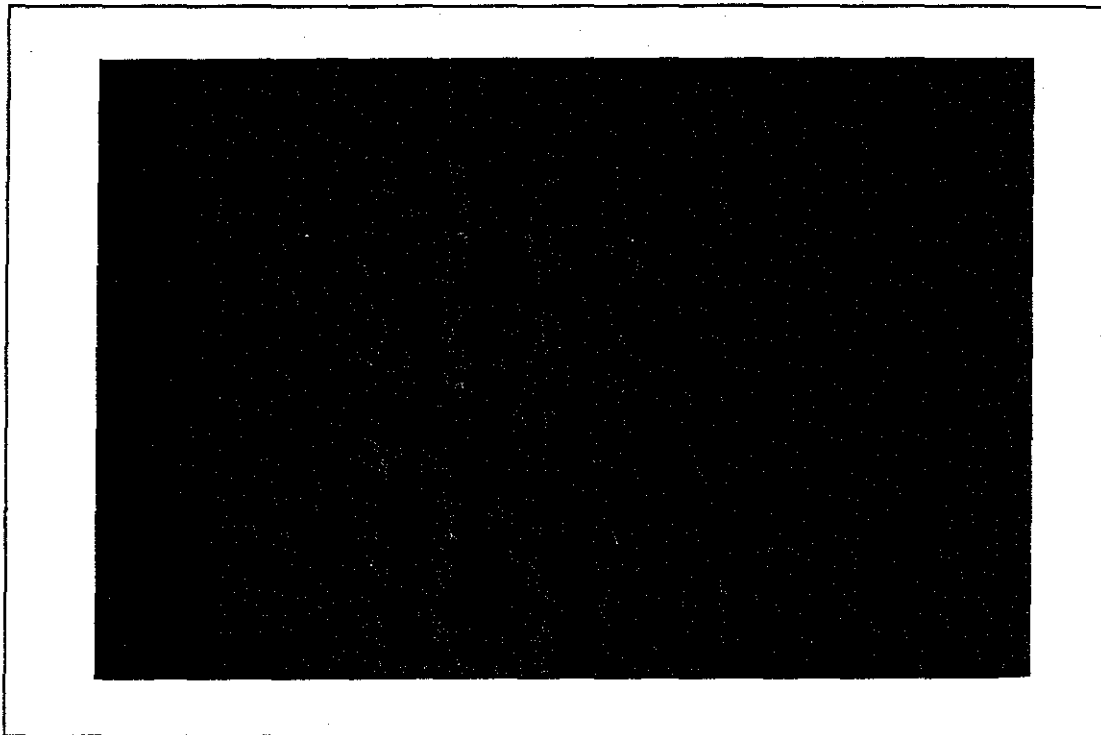
A pelvis with internal organs. The large bones are the pelvic girdle. Fitting into sockets at the bottom left and right of the girdle are the tops of the femurs. The red shape is the rectum, the purple is the prostate. The yellow bladder is cancerous, and is a candidate for radiation therapy. The darker surface surrounding the organs is a surface of 90% radiation dosage.

Figure 8



This data was generated from NMR scans of a male chest. The long blue tube is the esophagus, which leads to the stomach. This esophagus has a tumor which is to be treated with focused radiation beams. The spine is yellow. The lungs, skin, and vertebral bodies are mostly transparent. The scanned LBI file contains 16 isodosage surfaces. On the left, all surfaces but the 55% radiation surface are transparent; on the right, all but the 80% surface are transparent.

Figure 9



This 16-frame film was built from the same data as Figure 9. Each frame shows 4 different views of the chest, and is labelled with the particular isodosage surface made visible in that frame. The 4 views were each scan converted once, and then bound with a different set of surface parameters for each frame. Note that as the radiation dose increases, the surface changes from light blue to red.

Figure 10

Spacetime Ray Tracing for Animation

IEEE Computer Graphics & Applications

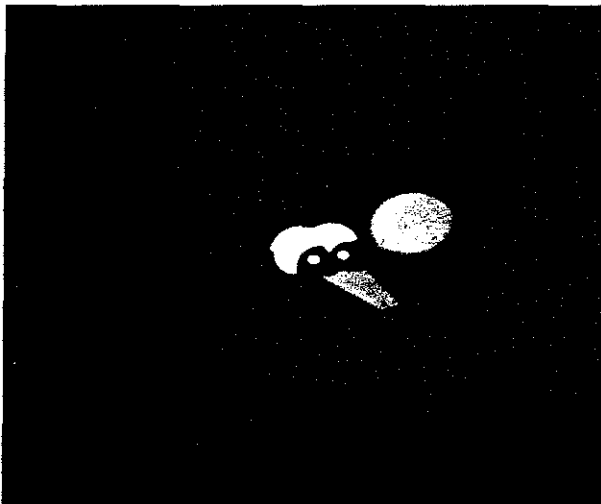
vol. 8, no. 3, March 1988

© 1988 IEEE. Reprinted with permission, from
IEEE Computer Graphics and Applications,
Vol. 8, No. 3, pp. 60-70, March 1988.

Spacetime Ray Tracing for Animation

Andrew S. Glassner

University of North Carolina at Chapel Hill



We are presenting techniques for the efficient ray tracing of animated scenes. These techniques are based on two central concepts: spacetime ray tracing and a hybrid adaptive space subdivision/bounding volume technique for generating efficient, motion-blurred hierarchies of bounding volumes.

In spacetime ray tracing, instead of rendering statically moving objects in 3D space, we render static objects in 4D spacetime. To support spacetime ray tracing, we use 4-dimensional analogues to familiar 3-dimensional ray-tracing techniques.

Our new bounding-volume hierarchy combines efficient

space subdivision and bounding volume hierarchies. The quality of the hierarchy and its ability to handle motion-blurred objects is an improvement over previous approaches because both attributes reduce the number of potential intersections that must be tested. These techniques are essential in animation because of the much higher cost of computing ray-object intersections for motion-blurred animation.

We show that it is possible to ray trace large animations more quickly with spacetime ray tracing using this hierarchy than with straightforward frame-by-frame rendering.

Ray tracing is a powerful and popular technique for image synthesis. When first introduced for computer graphics,^{1,2} ray tracing was comparable in power to scan conversion, but less attractive because of its high computational cost.

Survey

The effects of reflections, refractions, and shadows were estimated by adding recursion to the original ray-

tracing algorithm.^{3,4} Unfortunately, some notable combinations of these effects were incorrect.

Image synthesis and ray tracing

For example, if a shadow-testing ray encountered a partially transparent sphere, there was no proper single direction in which to send the ray after passing through the sphere's surface. Either this shadow was rendered as though blocked by an opaque object, or the modeler

introduced ad hoc techniques into the algorithm to handle particular situations correctly.

A solution to some of these problems was introduced in the form of distributed ray tracing.⁵ Whenever there was no single correct value for a ray parameter (such as the direction of the shadow ray discussed above), the domain of useful values was searched for an "appropriate" choice. This choice was made on the basis of the shape of the parameter space being sampled and the expected number of samples to be taken. Soft shadows and antialiasing in all dimensions were now available in a single, conceptually elegant algorithm. A technique for dynamically optimizing the number of rays cast when generating an image was presented by Lee and Usselton.⁶

The ray-tracing algorithm was theoretically unified and extended again by Kajiya.⁷ Ray tracing was formalized as a technique for solving the "rendering equation," which describes light distribution and energy balancing in an environment. This work suggested ways to include caustics and diffuse interreflections in a ray-tracing environment.

Unfortunately, a straightforward implementation of ray tracing is prohibitively expensive in computer resources and time. Finding efficient techniques to implement ray tracing is an active research area.

A brief survey of single-image rendering speedup techniques

Efforts to improve the efficiency of the technique have taken place on two major fronts: bounding volumes and space subdivision. Both of these efforts have seen investigation of important subissues: hierarchies for bounding volumes and the style of decimation for space subdivision.

A central idea behind bounding volumes is that it is often cheaper to intersect a ray with several mathematically simple objects than a single complex one. So complex objects are surrounded by simple objects (the bounding volumes), and these are recursively grouped together and enclosed within larger bounding volumes, forming a hierarchy. Rays that miss a bounding volume save a lot of work: they needn't examine any object within. Rays that do strike a bounding volume must then be intersected with everything inside the volume (which might include smaller bounding volumes). Such rays suffer the penalty of having computed the bounding volume intersection; the details of this intersection are useless except to signal that the internal objects must be tested. Bounding volume approaches to ray tracing are described in a number of works.⁸⁻¹⁴

A different approach to speeding up ray tracing is called space subdivision. The central idea here is to decimate space into a collection of disjoint simple volumes (often boxes), which are chosen so that each encloses only a small number of objects. When a ray enters a given box, it is intersected only with the objects within that

box. If no objects are hit within the box, the ray moves to the next box on its path and repeats the procedure. Several approaches that use space subdivision have been published.¹⁵⁻¹⁸

Both techniques address the issue of rendering a single image. In this article we propose combining these methods and extending them into the realm of animated sequences.

A hybrid technique combining adaptive space subdivision and bounding volumes

In this section we present a technique for the creation of efficient bounding volume hierarchies. The technique is a hybrid of adaptive space subdivision and bounding volume techniques.

The advantages of bounding volume techniques lie in their ability to easily avoid computing ray-object intersections for all objects within a bounding volume not penetrated by a particular ray. If the volume is entered, then all of its immediate children must be intersected. If the bounding volumes can overlap, then it is not sufficient simply to proceed with the nearest of these children, since the nearest bounding volume may not contain the nearest object.⁹ In this context, the biggest drawback to bounding volume techniques is that sometimes ray-object intersections are ignored; such computations (which may be very expensive for complex objects) are unnecessary. A recent paper¹⁴ presents some techniques for measuring and building a hierarchy, but the definition and construction of good hierarchies is still poorly understood.

The other popular speedup technique is space subdivision. Many space subdivision schemes use rectangular prisms (called cells) for the unit element of space. The hierarchy created by adaptive space subdivision techniques is excellent: No cells at any given level overlap, and cells are dense only where the database is dense. On the other hand, rectangular prisms can perform poorly as bounding volumes compared to sets of slabs and other techniques.

To summarize, bounding volumes offer tight bounds but poor hierarchies, while adaptive space subdivision offers poor bounds but very good hierarchies; the approaches are complementary in their strengths and weaknesses. Our technique is to use the excellent hierarchy created by space subdivision as a guide to control the structure of the tighter bounding volume hierarchy. We will now present an overview of the algorithm, and then discuss some variations.

The general theme of our approach can be summarized as constructing a bounding volume hierarchy in the order "space subdivision down, bounding volumes up." For simplicity, we will often refer to a bounding volume simply as a "bound."

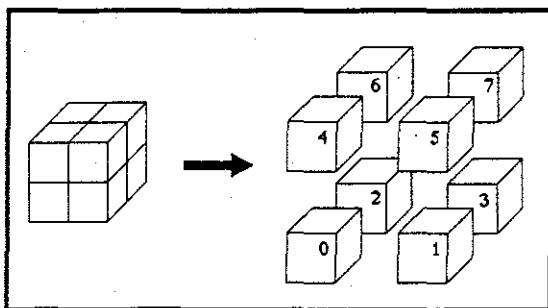


Figure 1. The subdivision of a rectangular prism into eight smaller prisms. On the left is the prism showing the locations of the cutting planes. On the right is an exploded view of the subdivided prism showing the labels of the eight smaller prisms.

We begin by finding an enclosing box for the entire database, including light sources and the eye. We then evaluate a subdivision criterion (discussed below) for that box and its contents. If we decide to subdivide, then that box is split into eight new, smaller boxes, as shown in Figure 1. It is important to note that these boxes do not overlap. We then examine each new box in turn, determining which objects within the parent box are also within each child. We then evaluate the subdivision criterion for each child box, and recursively apply the subdivision procedure for each box that must be split. The recursion terminates when no boxes need to be subdivided.

This concludes the "space subdivision down" step. We now build the bounding volume hierarchy as we return from the recursive calls made by the space subdivision process. Each node is examined, and a bounding volume is built which encloses all the objects contained within that node, *within the bounds of that node*. One way to visualize this process is to consider building a bounding volume for all objects within a node, and then clipping that volume to lie within the walls of the space subdivision box, as shown in Figure 2 (note that implementations may use a simpler and more direct method). As we work our way back up the space subdivision tree, we build bounding volumes that contain the bounds and primitives of the child boxes at each node. We note that if a cell has only one child, then we may replace that cell by its child to improve efficiency when rendering.

This completes the "bounding volume up" step. The result is a tree of bounding volumes that has both the nonoverlapping hierarchy of the space subdivision technique and the tight bounds of the bounding volume technique. Thus the new hierarchy shares the strengths of both approaches while avoiding their weaknesses. The result is that when we trace a ray, we can always examine the nearest bounding volume at all levels in the hierarchy. If we find an intersection in that volume then we

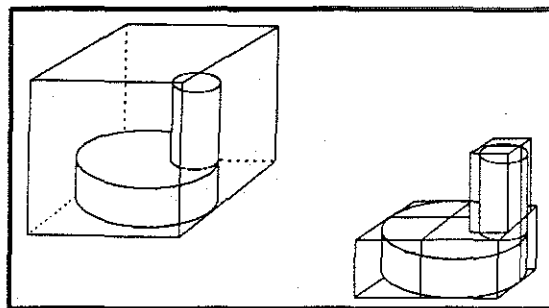


Figure 2. On the top is an object surrounded by a single bounding volume (for clarity the bounding volumes are rectangular prisms in this figure). On the bottom is the same object after the surrounding bounding volume has been subdivided. Note that the new, smaller bounding volumes are contained within, but not equal to, the smaller prisms created by the subdivision of the original bounding volume.

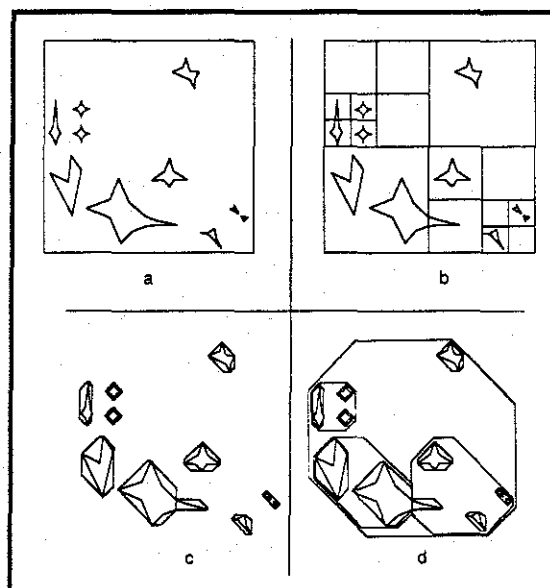


Figure 3. (a) shows a scene of 10 objects. (b) shows an adaptive space subdivision grid placed on those objects, subdividing any cell that contains more than two objects. (c) shows octagonal bounds (four slabs) placed around each primitive. (d) shows the final bounding hierarchy formed by the bounds in (c) and the subdivision tree in (b).

can immediately stop. Figure 3 summarizes the hierarchy creation process.

If no intersection is found, we then proceed to the next bounding volume, using either the bounding volume or

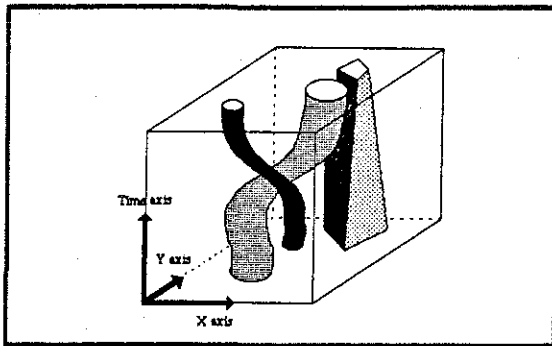


Figure 4. This diagram shows a 3D spacetime. The two space axes are labeled X and Y. Each slice of this spacetime volume parallel to the X and Y axes selects the world at a particular instant of time. Inhabitants of this world would experience the flow of time if they were moving along the time axis at a steady rate.

space subdivision structure to guide the ray propagation. It is important to note that since our bounding volumes might not completely enclose their objects, we must check that the intersection is indeed within the limits of the current bound.

There are many ways to apply adaptive space subdivision and bounding volumes; we will briefly mention some of the variations. The space subdivision cells may be axis-oriented,¹⁵ or oriented arbitrarily in space.¹³ The adaptive subdivision may be performed by equal cuts in all directions,¹⁵ in a BSP methodology,¹⁷ or with the median cut algorithm¹⁹ based on the distribution of objects in the cell. The subdivision criteria may be based on the amount of projected "void area,"¹⁰ the object count in a cell,¹⁵ or on the density ratio of the total volume enclosed by the objects to the total volume of the cell.²⁰ The last technique is useful when working with "intelligent" objects, which may represent themselves with different bounding volumes, depending on the level of the bounding hierarchy.

The bounding volume construction may use rectangular boxes,¹³ polyhedrons,¹⁰ parallel slabs,⁹ or surfaces of revolution.²¹ Because the bounds constructed at each cell are a union of the bounds of all child cells and primitive objects in that cell, the style of bound at each cell may be different, enabling one to "tune" the bounds of each object individually.

Spacetime ray tracing

The central idea in our solution to the ray tracing of animated sequences is to consider the time-varying geometry of the 3D database as a static structure in 4D spacetime.²² Since many people find it difficult to visualize 4D spaces directly, we will approach the 4D spacetime algorithm by analogy with 3D spacetime.

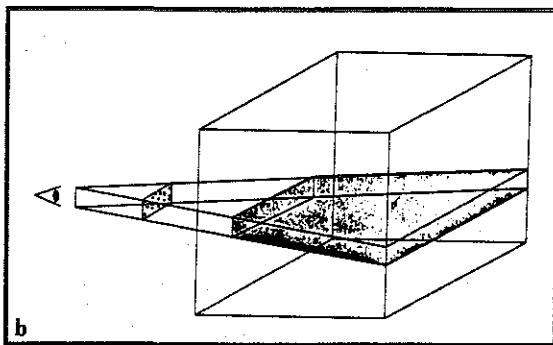
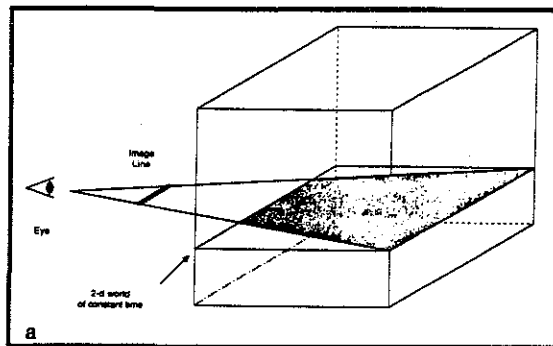


Figure 5. (a) To ray trace a frozen instant of 3D spacetime, we choose a 2D slice along the space axes. This entire slice has the same time value. We then project this 2D world onto a 1D image line, with the observer at the apex of this 2D viewing pyramid (a triangle). (b) To approximate motion blur, different samples in the image are taken at different times. This has the effect of thickening our sampling plane into a sampling volume.

3D spacetime

Three-dimensional spacetime can be thought of as a 3D space, containing a 2D space translated continuously in time.²³ Figure 4 shows a 2D world (a section of a plane), changing with time. In an animated sequence, objects will move about in this 2D space as time progresses.

Most of the rendered animation we usually produce consists of the projection of worldly 3D objects onto a 2D image plane. In a world of one less dimension (the 2D world of Figure 4), we render 2D objects onto a 1D image line, as shown in Figure 5a. Let's say we want to use ray tracing to produce a movie of this changing 2D world. Our rays that sample the 3D spacetime may start at any point (say an intersection with an object) and move in any direction. If we want to include motion blur in our movie, then these rays may also start at any time during a frame, as shown in Figure 5b.

The question now becomes one of quickly tracing the 3D ray in spacetime, intersecting it with each 2D object in its spacetime path. Each intersection of a ray and object is denoted by the three coordinates (X,Y,T). Each (X,Y) location in pure space is called a space point. Each (X,Y,T) location in spacetime is called a spacetime event.²⁴

To make our movie of the motion depicted in Figure 4, we could simply shoot rays from the eye, at various times and in various directions, into the 3D spacetime structure and try intersecting the rays with each object, searching for the first event along the ray's path. This naive approach would be very expensive computationally. Alternatively, we can adapt the bounding volume hierarchies described in the previous section. Instead of building spatial bounding volumes in 3D space, we will build spacetime bounding volumes in 3D spacetime. As long as we know the 3D spacetime structure, we can rename one of the axes in the 3D space algorithm as time. When we subdivide along the time axis, we are actually now subdividing the amount of time for which this bounding volume encloses its child objects.

One way to see this is to envision Figure 1 as bounding volume for a 2D object in 3D spacetime. With this interpretation, nodes 0 through 3 now contain the first half of the time interval, and nodes 4 through 7 contain the second half (visualize the time axis as moving from the bottom to the top of the page). Now we can restrict our intersection tests only to those objects that occupy the same region of space and time that the ray is sampling.

Subdivision in higher dimensions

So far we have looked at a 3D spacetime containing a 2D world, rendered onto a 1D image line. The techniques discussed above extend easily into a 4D spacetime of three spatial dimensions plus time. Higher dimensions are also straightforward, and may be useful in situations where objects change along dimensions that are being sampled other than just space and time, such as wavelength.

Animation in 4D spacetime

When we pierce a spacetime bounding volume with a 4D ray, we don't yet actually have the ray/object intersection event. Since collections of objects and other bounding volumes may reside within a single bounding volume, we must look into the volume and test the ray against its contents. Because objects may move in complicated ways over time, we feel that 4D ray tracing is best handled by an object-oriented environment, which allows intelligent objects to perform their own intersections. After describing such an environment, we will describe how to achieve the same function (though with greater effort) from data-driven animation in a proce-

dural environment, such as a traditional keyframe animation system.

Bounding volumes and intersection events from intelligent objects

In our technique the bounding volumes are created by the objects themselves, in response to requests by the hierarchy construction preprocessor.²⁵ Requests consist of asking an object for its enclosing spacetime volume within some region of spacetime. Many objects can easily respond with one of the bounding volumes discussed earlier.

An advantage of this intelligent object approach (such as described by Amburn, et al.²⁶) is that objects can determine their own most efficient representations. For example, a group of stars may represent itself by a single bounding volume when the subdivision begins. When the bounding volume requests enclose smaller spacetime volumes, the star group may improve its representation by describing itself as several smaller clusters, returning several bounding volumes instead of one. Another advantage is the simplicity of the program itself, and the ease of adding new objects. Objects are also able to respond to requests to intersect themselves with a particular 4D ray, returning the first such event along the ray if one exists.

If the application environment of the ray tracer does not support such intelligent objects, then the work of building bounding volumes and finding intersection events must be made by the animation manager. For example, a keyframe animation system would need to construct the bounding volumes for objects in given ranges of space and time according to the interpolation techniques it used to build the animation. When building a particular bounding volume, such a system needs to examine the object carefully throughout the time duration of the request to which it is responding. In a complicated animation system objects may move and change in complicated ways; one must be careful to insure that each bound completely encloses the object for the entire time interval. Similar care must also be taken when determining intersection events.

Summary of the 4D spacetime algorithm

The creation of a piece of animation begins with a preprocessing step. This step recursively builds an adaptive space subdivision tree on the static 4D spacetime structure of the moving objects. When we return back up the tree, we build bounding volumes that enclose the objects at each cell.

To render this structure we fire 4D rays into the bounding volume hierarchy. Because of the nonoverlapping nature of the hierarchy, we are guaranteed that we may choose the nearest bounding volume at every level. If we strike an object in this nearest volume, then we need not also test other, further volumes.

Motion blur due to camera motion is naturally accommodated by using different starting times and positions for the primary rays.

The sources of efficiency

The spacetime rendering algorithm is efficient for animation for the same reason that space subdivision and bounding volumes are efficient for single frames. Consider that in ray tracing, a ray must find its nearest object: This requires searching the entire database. Space subdivision sorts the database almost completely in a preprocessing step. Now each ray need only search the objects in a given volume. Any bounding volume hierarchy in fact does the same thing, although the sorting may be more complicated. With these techniques, each ray needs only to search through a small number of objects, each with a high probability of intersection. In single-frame techniques the bulk of the searching was distributed to the preprocessing sort that built the hierarchy of space enclosures.

Space subdivision and bounding volumes speed rendering by sorting the database once at the start of the frame, instead of for every ray. The technique introduced in this article speeds rendering by using a single, nearly complete spacetime sort instead of many space sorts. It performs this one sort at the start of the animation, instead of for every frame.

Another important source of efficiency is the reduction in the number of object transformation calculations that must be performed. When we shoot rays at different times to approximate motion blur, those rays intersect the objects in the database at different times. To intersect the ray and each object properly, the object must be transformed to the correct position, orientation, and shape for that time. If the object motion is complex, the transformation may include deformations and other sophisticated changes. These transformations may be very expensive to compute. Because our bounding volume hierarchy is nonoverlapping, we avoid computing intersection events along hierarchy descent paths that don't lead to the first intersection. This reduction in the number of intersections that we must compute can become significant for complex object transformations in dense regions of the database.

Implementation

The algorithm generating the hierarchy of spacetime bounding volumes requires a technique of bounding volumes and a technique of adaptive space subdivision. In our implementation, we chose slab bounding volumes⁹ and equal subdivision.¹⁵ Both algorithms are easily extended to work in spacetime instead of just space.

Spacetime rays are represented by a pair of 4D events giving the origin and direction of the ray. When render-

ing at normal scales, the time component of the direction vector may be set to 0, implying that the light ray has infinite speed. At extremely large and small scales, we may instead set the time component to a value consistent with the speed of light in the database. With suitable enhancements to the ray-tracing geometry, we can then handle relativistic effects.²⁴

In the 3D environment, a good set of bounding planes consists of the seven planes generated from the three axes and the eight octants they form.⁹ The three axes give rise to three principal planes (each containing one unique pair of axes: XY, XZ, or YZ), plus four auxiliary planes that each diagonally slice two octants.

In 4D we have four principal axes, which cut spacetime into 16 subspaces, which we call hexants. Eight of these hexants contain the first half of the time interval, while the other eight cover the latter half of the time interval. We now have four principal planes (containing XYZ, XYT, XZT, YZT), plus eight auxiliary planes that diagonally slice half of the hexants, for a total of 12 planes.

Our principal planes have normals:

$$(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)$$

Note that these slabs are required if we are using axis-oriented subdivision, since they form the walls that separate adjacent cells. The auxiliary planes have normals

$$(.5,.5,.5,.5) (.5,.5,.5,-.5) (.5,.5,-.5,.5) \\ (.5,.5,-.5,-.5)$$

$$(.5,-.5,.5,.5) (.5,-.5,.5,-.5) (.5,-.5,-.5,.5) \\ (.5,-.5,-.5,-.5)$$

Using planes formed by these normals we effectively enclose each spacetime path in a convex, bounding polyhedron formed by the intersection of 12 slabs, each composed of two parallel planes. We may add additional spacetime slabs to those above if we desire even tighter bounding volumes.

The cost of the 4D spacetime ray/slab intersection is virtually the same as for the 3D case. The difference is an extra pair of multiplies and additions once per ray per normal to compute both of the dot products of the spacetime normal with the ray origin and direction.⁹ But we consider that cost negligible, since it is amortized over the life of the ray.

Our system is implemented in the C programming language under Unix, which is not the most natural environment for object-oriented programming. We thus use indirect procedure calls and consistent methodology to achieve an object-oriented flavor in the system. For example, our objects are able to respond to messages requesting bounding volumes within a given 4D box, intersections with a given ray, and intersection comple-

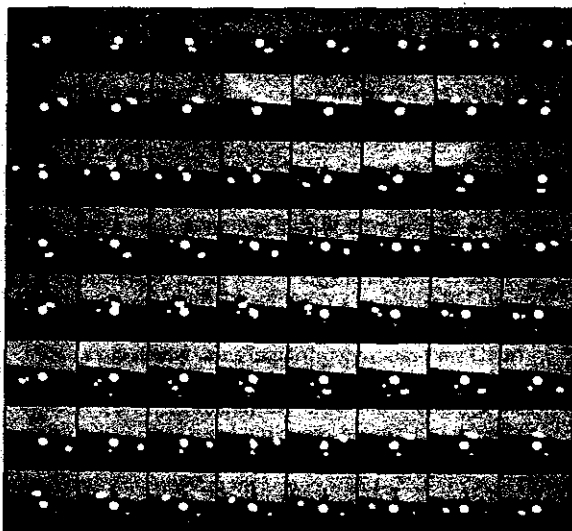


Figure 6. The upper photo shows a cyclic 64-frame animation of six small spheres ("electrons") spinning in different speeds in complicated motion around a larger central sphere (the "nucleus"). The lower photo is an enlargement of the 25th frame, showing the effect of motion blur on the small balls and their shadows.

tions (e.g., determining surface normal). Our objects also perform a variety of householding tasks such as maintaining their own motion paths, managing time-varying surface deformations and texturing, and so on.

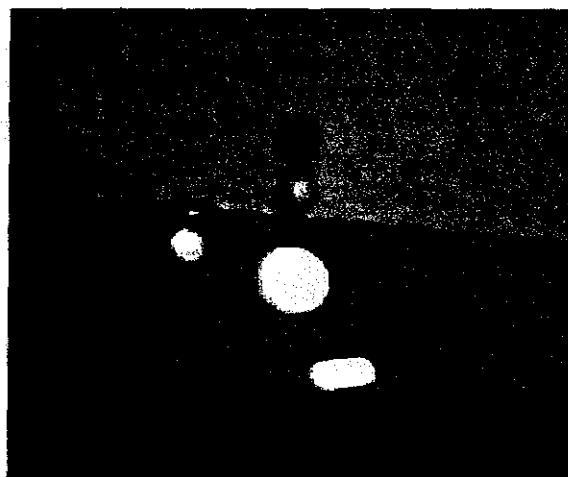
Results

Figures 6, 7, and 8 show three animations we have produced with these techniques. Because of the limitations of the print medium, we present the animations by grids of frames equally spaced in time. Read the animation grids as you would read a book: starting at the upper left, moving left to right and top to bottom.

Figure 6 shows 64 samples from a cyclic animation of an atomic model. Six spheres spin about each other and about a central nucleus in a complicated ballet. Motion blurring is evident in the faster moving balls and their shadows.

Figure 7 shows 16 samples from a cyclic animation of a group of spheres, moving on the surface of an octagonal prism. This figure was generated with four samples per pixel, distributed in space and time.

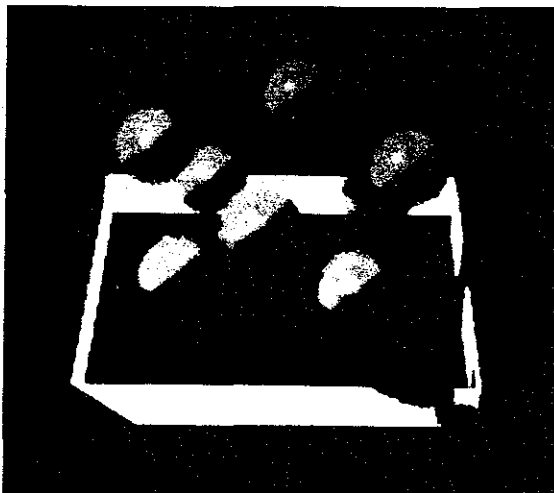
Figure 8 is a 64-frame synopsis of the short film *Dino's Lunch*.



Tables 1 and 2 summarize the statistics we have measured for Figures 6 and 8. All frames were generated at 126-by-126 pixels. Each frame of Figure 6 contains seven spheres and one polygon, and was sampled with a constant 32 eye-rays per pixel. Each frame of Figure 8 contains 47 spheres and 21 polygons, and was sampled with a constant four eye-rays per pixel. All eye-rays were distributed in the 3D spacetime volume occupied by the frame.

The columns labeled "Frame-by-frame" report the costs for the entire animation when generating purely spatial bounding volumes anew for each frame. These

Figure 7. The upper photo shows a cyclic 16-frame animation of eight spheres bounding on the surface of an octagonal prism in a small box. The lower photo is an enlargement of the 12th frame, showing the speckled pattern characteristic of low-density distributed ray tracing with motion blur (each pixel fired four rays).



bounds were taken to encompass the object for the duration of the frame, created and arranged in a hierarchy as in Kay and Kajiya.⁹ The column labeled "Spacetime" reports the equivalent costs using the techniques in this paper.

Figures 6, 7, and 8 were all computed using a hybrid subdivision criterion. At the upper levels of the tree, we subdivided until no more than three objects were in a cell. After meeting that criterion, we used a density measure: If the ratio of the volume enclosed by the objects in a cell to the volume of the cell was less than 0.3, the cell was subdivided (we used standard numerical inte-



gration techniques²⁷ to estimate the volumes). The column labeled "Spacetime to frame-by-frame ratio" contains the ratios of the animation totals for the two techniques, and is graphed in Figure 9.

Clock timings are not presented in the tables, since actual rendering times are strongly influenced by programming style and code tuning. Specifically, our code is not optimized for the algorithms in this article, since it performs many other tasks as part of a much larger system.

We have thus normalized all time measurements to an arbitrary unit time. Our unit time was the average time to render one frame of Figure 8. All measurements below the heavy horizontal line in the Tables are reported relative to this time unit. The proper statistics for comparison lie not in the elapsed time, but in the other columns, reporting the number of bounding volumes made and intersected, and the number of ray/object intersections with their accompanying expensive object transformation.

Discussion

The ratios in the comparison columns in Tables 1 and 2 are encouraging. They reveal that even in animations of modest complexity spacetime ray tracing with a hybrid hierarchy yields savings over frame-by-frame rendering.

From Table 1 we see that spacetime ray tracing was able to cut the rendering cost of Figure 6 to about 50 percent of that required by frame-by-frame techniques. Table 2 shows that spacetime ray tracing reduced the cost of Figure 8 to about 80 percent of frame-by-frame methods.

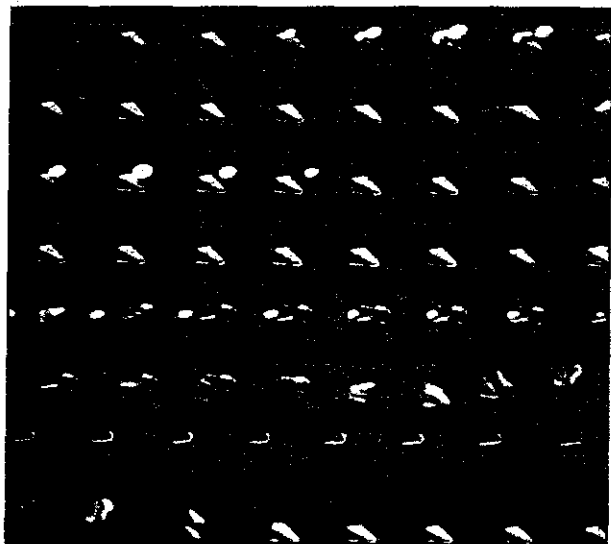


Figure 8. This is a 64-frame sample of the short film *Dino's Lunch*.

Table 1. Rendering statistics for Figure 6.

	FRAME-BY-FRAME		SPACETIME	Spacetime to Frame-by-Frame Ratio
	Frame Average	Animation Total	Animation Total	
# rays	793,637	50,792,795	50,792,794	1.0
# bounding volumes built	7.3	468	217	0.463
# ray/primitive intersections	1,825,365	116,823,428	66,030,634	0.565
# ray/bounding volume intersections	2,857,093	182,854,062	106,664,869	0.583
average primitive intersections per ray	2.3	2.3	1.3	0.565
average bounding volume intersections per ray	3.6	3.6	2.1	0.583
bounding volume hierarchy creation time	0.011	0.710	0.33	0.465
rendering time	34.8	2233	1179	0.768
total animation generation time	35.1	2234	1180	0.528

Table 2. Rendering statistics for Figure 8.

	FRAME-BY-FRAME		SPACETIME	Spacetime to Frame-by-Frame Ratio
	Frame Average	Animation Total	Animation Total	
# rays	22,748	1,455,876	1,455,876	1.0
# bounding volumes built	66	4221	331	.08
# ray/primitive intersections	73,148	4,681,506	4,201,077	0.90
# ray/bounding volume intersections	275,962	17,661,570	12,841,239	0.73
average primitive intersections per ray	3.22	3.22	2.89	0.90
average bounding volume intersections per ray	12.13	12.13	8.82	0.73
bounding volume hierarchy creation time	0.1	6.4	2.2	0.34
rendering time	1.0	64	49	0.77
total animation generation time	1.4	90.2	71.1	0.79

Since spacetime ray tracing builds bounding volumes only once at the start of the animation, we would expect that it should generate fewer bounding volumes over the course of the animation than frame-by-frame techniques. The results show that we indeed observed such an effect in both animations. The most important statistic is the

number of ray-object intersections; this figure also decreases in spacetime ray tracing, thanks to the additional information provided by the time component in the spacetime bounding volumes.

Thus even in these simple animations, spacetime ray tracing can offer us significant savings in time by reduc-

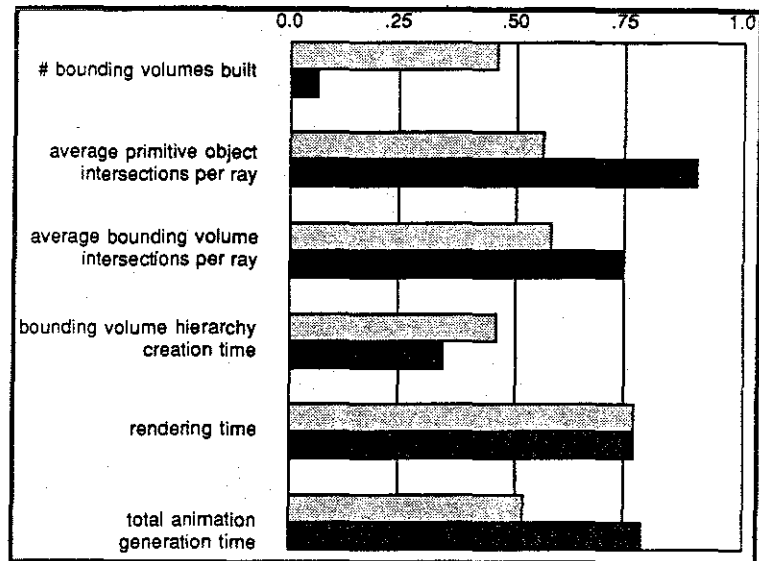


Figure 9. Percentage of work required by new technique relative to frame-by-frame techniques. Light shaded boxes are for Figure 6; dark boxes are for Figure 7.

ing the number of complex intersection operations that must be performed.

We note that several factors strongly affect these statistics, such as the distribution of objects in the scene, the complexity of each ray-object intersection, the complexity of the animation and database transformations, and the length of the animation in frames.

Consider a complex object changing in complex ways over time, such as a boiling fractal volcano with flowing lava. It can be very expensive to intersect such an object with a ray at a given time. This is because time-dependent intersections require positioning an object along its motion path, interpolation of all object description parameters, and then construction of the object itself (at least to a level sufficient to reject the ray). Without spacetime bounds all of this work will have to be performed for each object, even for rays that cannot possibly hit the object because they are in the wrong place at the wrong time. Spacetime bounds eliminate the majority of these useless intersection calculations, eliminating also their associated complex object positioning and construction operations.

We therefore expect that complex animation, involving many complex objects in sophisticated motion, will yield substantially higher savings than the examples presented here; indeed, we expect that as the animation grows more complex, the savings will become greater. This is based on the above discussions about the sources of efficiency, and by analogy to the performance of space subdivision and bounding volume algorithms. We are currently planning an elaborate animation called *Dino and the Windmill* (the sequel to *Dino's Lunch*) to test this

expectation. *Dino and the Windmill* will also include extensive movement of the lights and camera.

Future work includes lazy evaluations of the bounding hierarchy (constructed of only those bounds needed as the animation progresses). We also plan to synthesize our methods with other multidimensional ray-tracing acceleration techniques, such as that described by Arvo and Kirk.²⁸

Summary and conclusion

We have presented techniques for efficient ray tracing of animated scenes. We view the animation problem as a spacetime rendering problem. Thus, instead of rendering dynamically moving 3D objects in space, we render static 4D objects in spacetime. To trace rays in spacetime efficiently, we developed a hybrid technique of adaptive spacetime subdivision and spacetime bounding volumes, which generates an excellent hierarchy of nonoverlapping bounding volumes. The spacetime subdivision is also used during preprocessing to help intelligent objects select the most appropriate bounding volume for differently sized spacetime hypervolumes built as the subdivision progresses. We then trace 4D rays in this static spacetime to find ray-object intersection events.

We are able to ray trace a piece of animation more quickly with this spacetime algorithm and bounding hierarchy than with straightforward frame-by-frame rendering. ■

Acknowledgments

This work was developed and implemented in the Computer Graphics Lab at the University of North Carolina at Chapel Hill.

Thanks go to my advisor, Henry Fuchs, for his support of independent research. The idea of incorporating composite spacetime information into an animation system came out of discussions with Larry Bergman. Both a preliminary and a revised version of this article were carefully reviewed by a host of my fellow students in the Department of Computer Science; my thanks go to Marc Levoy, Pete Litwinowicz, Chuck Mosher, Tom Palmer, and Lee Westover. Kevin Novins of the Department of Radiology also provided insightful comments. Mark Harris and Doug Turner helped with the phrasing of important passages, and Margaret Neal helped make the article cohesive and clear. The observations and suggestions of these volunteer reviewers helped give this article structure and focus. Their friendly companionship in the Lab helped make the work a pleasure. Lakshmi Dasari provided key assistance in the production of the animations.

References

1. A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *Proc. AFIPS Conf.*, Vol. 32, 1968, pp. 37-45.
2. W. Bouknight and K. Kelley, "An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources," *Proc. AFIPS Conf.*, Vol. 36, 1970, pp. 1-10.
3. D. Kay, "Transparency, Refraction, and Ray Tracing for Computer Synthesized Images," master's thesis, Cornell University, Ithaca, NY, 1979.
4. T. Whitted, "An Improved Illumination Model for Shaded Display," *CACM*, June 1980, pp. 343-349.
5. R.L. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Computer Graphics (Proc. SIGGRAPH)*, July 1984, pp. 137-145.
6. M.E. Lee, R.A. Redner, and S.P. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing," *Computer Graphics (Proc. SIGGRAPH)*, July 1985, pp. 61-67.
7. J.T. Kajiya, "The Rendering Equation," *Computer Graphics (Proc. SIGGRAPH)*, July 1986, pp. 143-150.
8. E.A. Haines and D.P. Greenberg, "The Light Buffer: A Shadow-Testing Accelerator," *CG&A*, Sept. 1986, pp. 6-16.
9. T. Kay and J.T. Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics (Proc. SIGGRAPH)*, July 1986, pp. 269-278.
10. H. Weghorst, G. Hooper, and D. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Trans. on Graphics*, Jan. 1984, pp. 52-69.
11. J.T. Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Computer Graphics (Proc. SIGGRAPH)*, July 1982, pp. 245-254.
12. S. Roth, "Ray Casting for Modelling Solids," *Computer Graphics and Image Processing*, Vol. 18., 1982, pp. 109-144.
13. S.M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics (Proc. SIGGRAPH)*, July 1980, pp. 110-116.
14. J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *CG&A*, May 1987, pp. 14-20.
15. A.S. Glassner, "Space Subdivision for Fast Ray Tracing," *CG&A*, Oct. 1984, pp. 15-22.
16. M. Dippe and J. Swenson, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics (Proc. SIGGRAPH)*, July 1984, pp. 149-158.
17. M. Kaplan, "Space Tracing: A Constant Time Ray Tracer," *SIGGRAPH 85 Tutorial on the State of the Art in Image Synthesis*, July 1985.
18. A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *CG&A*, Apr. 1986, pp. 16-27.
19. P. Heckbert, "Color Image Quantization for Frame Buffer Display," *Computer Graphics (Proc. SIGGRAPH)*, July 1982, pp. 297-307.
20. A.S. Glassner, "Spacetime Ray Tracing for Animation," *Introduction to Ray Tracing, course notes #13 (SIGGRAPH)*, ACM, New York, 1987.
21. J.T. Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Computer Graphics (Proc. SIGGRAPH)* July 1983, pp. 91-102.
22. R. Rucker, *The Fourth Dimension*, Houghton Mifflin, Boston, 1984.
23. A. Abbott, *Flatland*, Dover Publications, Mineola, N.Y., 1952 (original copyright 1884).
24. P. Bergmann, *Introduction to the Theory of Relativity*, Dover Publications, Mineola, N.Y., 1975.
25. A.S. Glassner, "Supporting Animation in Rendering Systems, Proc. CHI+GI, Canadian Information Processing Soc., Toronto, 1987.
26. P. Amburn, E. Grant, and T. Whitted, "Managing Geometric Complexity with Enhanced Procedural Models," *Computer Graphics (Proc. SIGGRAPH)*, July 1986, pp. 189-195.
27. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes*, Cambridge University Press, N.Y., 1986.
28. J. Arvo and D. Kirk, "Fast Ray Tracing by Ray Classification," *Computer Graphics (Proc. SIGGRAPH)*, July 1987, pp. 55-64.



Andrew S. Glassner is a PhD student in Computer Science at the University of North Carolina at Chapel Hill, where he studies algorithms for image synthesis, modeling, and animation. He spent recent summers working on computer graphics at a variety of research institutions, including the Delft University of Technology, Xerox PARC, IBM TJ Watson Research Laboratory, Bell Communications Research, and the New York Institute of Technology.

Glassner writes on computer graphics for both the technical and popular literature. His book for artists, *Computer Graphics User's Guide*, has recently been translated into Japanese. He is presently working on two new books, describing procedural geometric methods for graphics programmers, and symmetry design for artists. Glassner's current research interests include image synthesis, geometrical methods for designing computer graphics algorithms, and the theory of certain knotwork patterns. He is also interested in other uses of computers for enhancing and supporting creativity, including music, multimedia displays, and interactive fiction, both verbal and visual.

Glassner can be reached at the Department of Computer Science, Sitterson Hall, Box 3175, UNC-CH, Chapel Hill, NC 27599.

Summary

Introduction

This dissertation contains six papers written over a span of five years. Each paper addresses a topic in the field of computer graphics; the emphasis is on efficient realistic image synthesis and animation.

In this concluding chapter, I will consider the content of each of the six papers individually. I will attempt to redress errors and omissions, clarify what is unclear, refine what is vague, and set them in the context of subsequent developments. These sections should be read as commentary on the papers, not revisions of the papers themselves.

Space Subdivision for Fast Ray Tracing

IEEE Computer Graphics & Applications, vol. 10, no. 4, October 1984

Space Subdivision for Fast Ray Tracing was essentially a synthesis paper: it took several existing ideas, and combined them with some new techniques to make a composite algorithm.

The thesis of the paper was that a software implementation of ray tracing could advantageously use a spatial data structure to speed the process of finding ray-object intersections. The data structure chosen was the linear octree, described in [Gargantini82]. The new techniques were a hashing scheme for fast voxel lookup and a mechanism for moving from voxel to voxel along a ray.

We will now look at the choice of the octree structure, the hashing mechanism, and the movement mechanism.

Spatial Data Structure

The choice of linear octree was explained with the following argument: in addition to being well understood, octrees have the desirable property that they can change the resolution of the subdivided space based on the properties of the objects in that space. In this paper, the density of the octree structure followed the density in the enclosed space, measured by the number as objects in some volume.

This was the extent of the defense of the selection of octrees in the paper. Although the claims are valid, there was no mention of the problems with octrees, nor consideration of alternative spatial data structures. These issues were addressed by a variety of later papers [Fujimoto85], [Kaplan85], [Kay85], [Jansen86], [Nemoto86], [Amanatides87], [Arnaldi87], [Cleary87], [Peng87], which also extended and refined some of the algorithms in this article. It now appears that arguments can be made both for adaptive and fixed spatial structures, depending on the nature of the database being rendered.

The essence of the tradeoff between the structures is in the cost for the voxel traversal mechanism, and the cost of finding the data structure corresponding to the next voxel. Typically the voxel traversal cost is more expensive for adaptive structures, since there is some built-in uncertainty about the resolution of the next voxel. The algorithm in this paper, for example, requires 6 subtractions, 6 multiplications, four comparisons, and an average of k additional multiplications, k additions, and $4k$ comparisons, where k is the average height of the octree. Uniform structures can exploit the regularity of subdivision to use simpler, very efficient traversal algorithms. For example, the 3DDDA algorithm of [Fujimoto85] requires about 4 truncations and additions per step (their paper is sparse on details, so this is an estimate from their statement "...one way to realize 3DDDA is to use two synchronized DDA's...", using the DDA algorithm in [Newman79]). This measure does not include initialization of the DDAs.

The advantage of an adaptive structure such as the octree, k-d tree, or BSP tree, is most evident in a database where the object density is heterogeneous on a large scale. Such databases include most human environments: a typical room has a large amount of empty space for people to move within, and dense, varied regions such a bookshelf or coatstand. The ability of an adaptive structure to conform to the changing spatial density of objects can translate to a faster propagation speed of the ray through the database; the ray may get from one end of the room to the other by passing through

only one or two large, mostly empty nodes. The extra work involved in moving from one cell to another may be compensated by the smaller number of cells involved.

On the other hand, many databases of very small and very large phenomena can be rather homogeneous in density, or at least composed of a simple, regularly repeating structure. An example of the former is a volume of the ocean; the latter might be a salt crystal. In such a database uniform spatial subdivision is attractive, as described by [Fujimoto85]. The cost of moving from one voxel to the next is less than for an adaptive technique, and the likelihood of striking an object in each voxel is relatively high.

The critical question is the comparison of the total cost of each algorithm. In a heterogeneous scene, adaptive techniques appear superior: the high voxel-traversal cost is offset by the small number of voxels required to pass through sparse areas. Uniform techniques would need to process many empty cells, and though each step may be faster than an adaptive step, the overall cost would be higher. In a homogeneous scene, uniform subdivision appears superior: movement is cheaper, and more likely to result in an intersection. In this case the more expensive voxel propagation cost of adaptive techniques would be a burden, since only a small number of cells are likely to be traversed until an intersection is found.

Counting objects in a small volume was a very crude method for measuring the local density of space. The reason we want to use some kind of density measure is because we want to estimate the chances of hitting an object when a ray enters a cell. Better estimates of this probability can come from finding the ratio of the sum of all object volumes compared to the cell volume, or else a ratio of the sum of all surface areas compared to the cell's surface area [Weghorst84].

Voxel Hashing Technique

The idea behind the hashing technique was to trade time for space in the storage of the octree. The mechanism was to eliminate the eight pointers at each node required to explicitly store the octree, and replace them with a number of linked lists. To find a node, one hashed its name into a small integer, and then followed a linked list of nodes associated with that integer. The claim was made that in the largest limit, each node would reside in its own list, so that access was immediate upon hashing. In the smallest limit, all nodes would reside in the same, large linked list, so that the whole list would need to be searched for each node.

To determine whether this was a good approach, we must look at just how much memory was actually saved. The result: we indeed save measurable memory, but the context in which this approach is used makes it of dubious value.

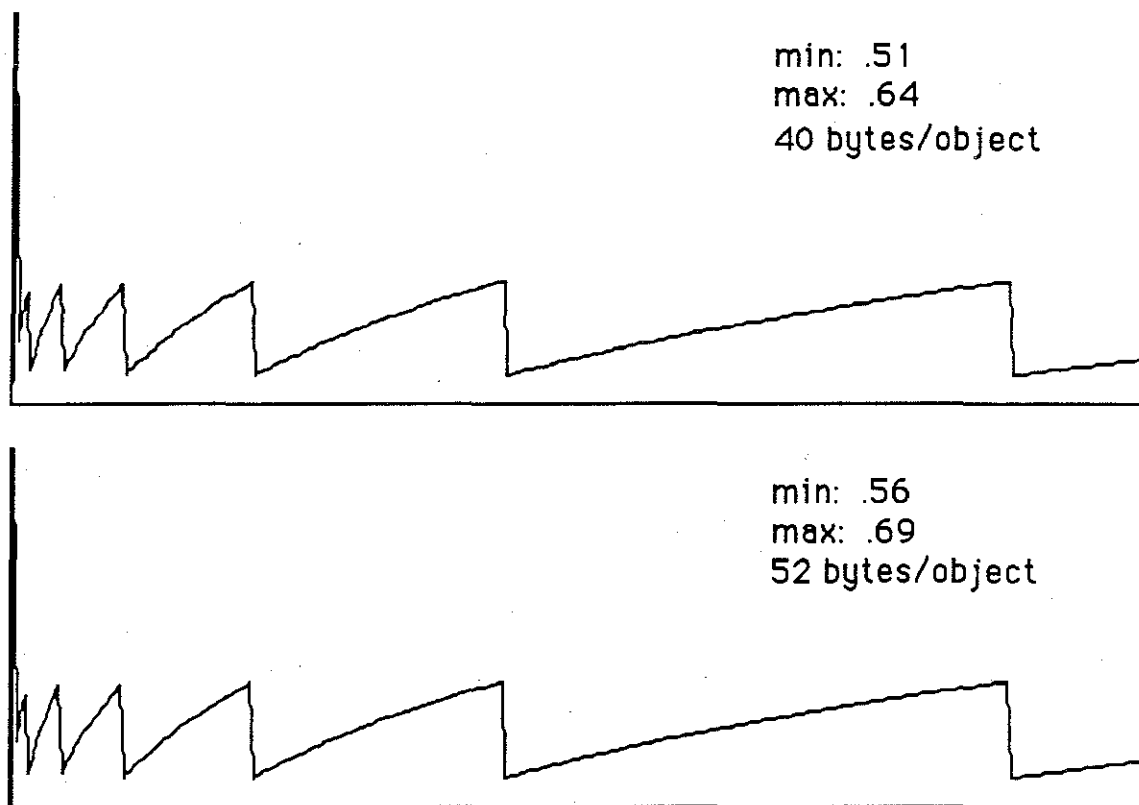
To address the first problem, consider that each node still had to contain one pointer to the next node in the linked list. Thus in a tree of k nodes, we saved $7k$ pointers. Suppose we had a scene composed of n objects. If each object is a triangle, then we will store at least 3 points in 3-space with the object, plus some surface information (let's suppose this is just a pointer to a surface description). Then assuming 4 bytes each for a floating-point number and a pointer, each triangle will cost 40 bytes, for a total of $40n$ bytes to store the database. Assume the objects are distributed uniformly in space, so the tree is as wide as possible. Then the tree will have k levels, where k is the smallest integer that satisfies $2^k \geq n$. Solving for k gives $k = \left\lceil \frac{\log n}{\log 2} \right\rceil$. The total

number of nodes that we need to store this tree is $t = \sum_{i=0}^{k-1} 2^i$ (we only sum to $k-1$ because the leaves have no child pointers).

Let us assume that each node contains a pointer to an object list, and a byte of local information, costing 5 bytes. In the hashed scheme, the cost of the complete tree would be $2^k (5+4) = 9 * 2^k$ bytes. For the complete tree, each node has 8 child pointers, for a total of $5+(8*4) = 37$ bytes; thus the tree costs $37 * 2^k$ bytes. The relative memory consumption of the hashed scheme is thus

$$s = \frac{9t + 40n}{37t + 40n}, \text{ where } t = \sum_{i=0}^{k-1} 2^i, \text{ and } k = \left\lceil \frac{\log(n)}{\log(2)} \right\rceil$$

From this, we can plot s as a function of n ., as in Figure 1 (in the Figure, n is plotted from 1 to 10000):



Relative memory costs for complete tree versus hash table
Figure 1

We can see that when we store 40 bytes/object, the ratio of hash-table to full-tree memory use ranges from about .51-.64. If we store a surface normal with each object, that takes 12 more bytes; the savings per node is less now, increasing the ratio to the range .56-.68. Clearly as each object consumes more memory, the relative savings from node reduction will continue to decrease.

We made two assumptions in the above discussion. The first was small object sizes; in the spacetime ray tracing system (the last article in this dissertation) each object consumes at least 400 bytes; some take much more. The memory savings ratio for 400 bytes/object is from .88-.94, or a savings between 6-12%. The second was assuming a complete tree; typical octrees will have many fewer nodes, and thus will expend memory on fewer pointers.

Unfortunately, we pay a rather large time penalty for these memory savings. Consider that when we wish to find the next voxel along a ray's path (using the techniques in this paper), we must descend the octree from the root, using the mechanism of Figure 3 in the text. Thus we examine each voxel along the path from

the root to the leaf of our next voxel along the ray's path. If the 8 child pointers are stored at that node, then we may simply consult the appropriate pointer to find the next child immediately. Using the hash table mechanism, we must hash the name of the child and follow the linked list to find the child. The irony is that we will descend much of the tree in the same order repeatedly (only differing below the lowest common parent of the two leaves); each time, for each node, we must go through the hash-lookup step.

The actual time consumed by this step is dependent on the hashing function and the number of lists maintained.

Is the space savings worth the time cost? Even without experimental data, we can make some observations. Virtual memory is now routinely available on most computers. With declining memory costs, the number pages that can be active in real memory at any given time is increasing. Furthermore, the very spatial locality exploited by the octree technique (successive voxels are spatially adjacent) translates into memory locality on the pages. In effect, our working set of pages corresponds into a working set of voxels. I suspect that we can hold enough complete (appropriate) voxels in memory at a time that the memory savings from the hash table is not necessary, and thus we need not pay its time penalty.

In conclusion, the hashing table does provide a savings in memory, but at a cost in running time; we probably don't need to save that memory these days, and consequently we may dispose of the technique and improve execution time. The hashing technique is only attractive when memory is more precious than running speed.

Movement Mechanism

When no intersections were found in a particular voxel, the algorithm found the next voxel with a three-step process. First, the point was found where the ray left the voxel. Second, that point was displaced away from the voxel interior by a small amount (carefully determined) perpendicular to each face which shared that point, creating a new point in another voxel. Third, the octree was descended from the root to find the voxel containing that new point.

The technique worked. The amount moved was equal to half the length of the smallest voxel side in the database, guaranteeing that the point was within the next voxel on the ray's path. The displacement simulated the propagation of the ray, and the octree descent indeed found the correct voxel.

The algorithm was slow. Faster mechanisms have since been found, such as those of [Amanatides87] and [Lathrop88]. But it was a reasonable way to proceed, and worked correctly.

Other items

Several other points in the article bear re-examination.

The algorithm that constructs the octree only considers the surfaces of objects, not their interiors. The article states, "The assumption is that the inside of a transparent or translucent object is either empty or else described by other, independently defined objects." This is not wrong, but it is incomplete. What we really are concerned with is where a ray might change its direction or color. If an object has a translucent interior of varying density, then the interior might well be associated with the enclosing object itself, rather than as a separate object. This presents no difficulty to the algorithm, since the critical information is which object is influencing the ray, and that is exactly the object that the ray is within. So objects may have sophisticated internal structure, but as long as that structure is part of the object's description, and has a well-defined boundary or surface, then we may restrict our attention only to the surfaces.

The mechanism for determining which voxels contain a given object was not discussed at all. In fact, the program used to make the pictures did a simple overlap test between the object's bounding box and the voxel; this should have been mentioned. Of course, more sophisticated tests would be useful and would probably provide for sparser trees (and therefore faster image generation).

The footnote on page 19 is in error; only three planes, not four, need to be tested to find the exit point of a ray from a voxel. My opinion in that footnote was wrong; determining which three planes lie in the forward direction of the ray can be part of an efficient movement algorithm [Amanatides87].

Lastly, the article states that a single object may straddle several cells, and thus the ray would need to intersect that object several times. It claims that this is an "uncommon" event. Perhaps that was so for the particular images in the paper, but it could easily be a common event in the general situation. Consider a dense region of space containing many objects; there is no reason to believe that the voxel walls should cleanly separate groups of objects without piercing any of them; indeed, such penetration appears likely. Multiple intersections with a single object appear to be a real drawback.

Fortunately, one may simply associate the ray parameter at the intersection point with the object, along with the ray id number. Should that ray ever query that object again, the intersection parameter may be simply retrieved from this "ray intersection cache" without re-solving the intersection problem [Hanrahan86].

The article is missing two important literature citations. [Rubin&Whitted80] presented one of the few ray-tracing acceleration techniques extant when this paper was written. [Murakami83] also discussed ray tracing in the context of a spatially-subdivided world.

Further work

Several papers have extended and improved many of the techniques described in this paper; others have explored alternatives. Other strategies for subdivision have been studied, including uniform subdivision [Fujimoto85], BSP trees [Kaplan85]; the use of k-d trees has also been explored [Hultquist87]. Voxel movement algorithms have been examined and extended by [Amanatides87] and [Fujimoto85]. Another way to find a voxel's address immediately has been proposed by [Lathrop88].

Summary

The main contribution of *Space Subdivision for Ray Tracing* was not in any of its component algorithms. Rather, it was one of the first demonstrations that a simple spatial data structure could be applied to the ray tracing problem to achieve speedups of more than an order of magnitude. Further work by others focused on other spatial data structures, including hierarchies of more general bounding volumes. This paper was most useful not for the particular algorithms employed, but for demonstrating the power of spatial data structures for accelerating ray tracing.

Adaptive Precision in Texture Mapping

Computer Graphics vol. 20, no. 4, Proc. Siggraph '86, August 1986

This paper isolated one of the approximations used in the sum-table texturing technique, and proposed a solution whose cost and effectiveness were proportional to an estimate of the error in the approximated texture.

The sum-table method is useful in many contexts, but it is most convenient to consider it in a traditional pipeline rendering system. In their job of estimating the texture appropriate for a given pixel, sum tables incorporate two fundamental approximations. First, the texture filter is approximated by a flat filter, with unit response inside its footprint, and zero response outside. Second, the image of a pixel is approximated by a rectangle oriented to the texture axes. This paper accepted the first approximation, but attempted to reduce the errors introduced by the second. Thus the final quality of the texture sample is limited by the flat filter response, however accurate its footprint.

The main assumption in the paper is that a better footprint is given by the convex hull enclosing the texture-space images of the four pixel corners. Indeed, this filter is superior to some arbitrary axis-oriented rectangle, but it is also inferior to other forms of space-variant footprints which allow overlap among pixels.

The paper begins by analyzing various additive and subtractive schemes among one or more sum tables. The quality of a texture estimate was measured by the amount of texture included in the sample, yet outside the pixel's texture-space image. This measure obviously ignores the values in the texture itself, which is a drawback. The analysis of various additive and subtractive schemes is useful and interesting. It is surprising that the availability of another sum table at a 45° angle to the first does not reduce the extraneous area sampled in the worst case.

A different error measure is then introduced, based on the local variance of the texture. Variance is a reasonable measure of image complexity, and indeed has been used to control stochastic sampling schemes [Lee85]. It was later proposed that a better measure is contrast, since that is a concept that correlates to our perception of a scene, rather than its abstract statistical properties [Mitchell87].

To estimate variance, a second table was built containing the variance in the local neighborhood of each pixel; this table was then converted into sum-table format. When the texture in a pixel was to be estimated, the average variance within the pixel was found by consulting the variance sum table. Once the variance was estimated, the algorithm removed rectangular pieces from the texture estimate until the sampled area was less than a worst-case value, originally computed by formula but stored in a table. The texture estimate improved as more pieces were removed.

A better way to estimate the average variance was suggested by [Heckbert86]. He suggested building a sum table of the squares of the texture values x_i , in addition to the normal texture sum table. Variance could then be estimated by the following (for a sampling region of n texture samples):

$$\sigma^2 = \frac{\sum x_i^2}{n} - \left(\frac{\sum x_i}{n} \right)^2$$

Comments

The abstract stated that texture samples could be determined to arbitrary precision; this is wrong. Sample quality was still limited by the flat filter response, and the “ideal” footprint assumed by the algorithm (the convex hull of the mapped corners of the pixel) is not ideal. But the algorithm could respond with increasing precision up to those limits.

In Section 3.0, a distinction was made between the region summed over in a sum table, and the region which could be queried. The former was called the table's *fundamental region*. Although the idea of a fundamental region is important to the study of sum tables, in retrospect there seems no reason to distinguish between the integration region and the query region. Indeed, the paper goes on to state that both regions usually have the same shape.

The discussion in Section 7.0 repeated a confusion between the approximations in the paper. It incorrectly stated that the texture was sampled “with a delta function, instead of a proper filter”, which is both incorrect and misleading. The error is that the filter used was not a Dirac delta function (zero everywhere except for unit height at one value), but rather unit-height within the sample and zero elsewhere. The misleading part is that this filter shape is not improper, though it certainly is not ideal.

Supporting Animation in Rendering Systems

CHI+GI Workshop on Rendering Algorithms & Systems

Toronto, April 1987

This short paper has a single major message, and an interesting, though unrelated idea. The message is an advocacy of an object-oriented, distributed database for animation support; the idea is to let objects help determine efficient sampling strategies. Both ideas were implemented in my spacetime rendering system.

The starting point is an argument that the traditional rendering pipeline is not efficient for creating motion-blurred animation. The proposed alternative is a

distributed, object-oriented architecture to support both animation design and rendering. This is not the first time such an architecture has been proposed, but to my knowledge this paper is the first to supply the actual messages required to support such a scheme. The arguments in favor of code sharing were also not new, but persuasion is required to justify the difficulty of implementing such a system in a system such as Unix, which is not friendly to code-sharing. Overall, the paper proposes a very simple and straightforward system, but one I had not seen explicitly described before.

The surprise in the paper is the suggestion of object-oriented parameter space sampling schemes for stochastic ray tracing; particularly for path tracing. The idea is not fully developed here, but the essence is described. The idea was later crystallized in the spacetime ray tracing system in the form of the "deck" data structure. The final analysis has not yet been completed, but the images appear at least as good as with other approaches for efficient path tracing [Kajiya86].

In summary, this paper presents a straightforward, though complete architecture for animation support. It also introduces the barely related concept of object-oriented parameter space sampling.

Template Parameterization for 3d Pose Interpolation

This paper suggests a technique for allowing a model designer complete freedom in specifying the transformations that describe a hierarchical model destined for interactive animation. The work came about when I was building an animation system, and wished to remove some of the restrictions inherent in many other modeling systems I have seen.

The use of the Singular Value Decomposition is appropriate; the numerical stability of the published algorithms of SVD makes this an attractive approach for solving our numerically delicate problem of matrix transformation.

This paper is straightforward; its message and technique are simple and are stated in only a few pages. However, one issue is not completely resolved in the paper, and that has to do with the conformation of mirror-inversion matrices between two decomposed keys.

The mirror-inversion matrices were introduced because I wanted some way to parametrically interpolate the orthonormal matrices created by SVD. From linear algebra we know that an orthonormal matrix J may be matched by $J=MR$, where R is

a pure rotation matrix, and \mathbf{M} combines a single mirror and a single inversion. \mathbf{R} may be represented as a function of three Euler angles, or with an equivalent quaternion, which respectively provide us with either 3 or 4 components to interpolate.

But \mathbf{M} is not parametric, and therein lies the problem. Consider two matrices $\mathbf{J}_0 = \mathbf{M}_0 \mathbf{R}_0$ and $\mathbf{J}_1 = \mathbf{M}_1 \mathbf{R}_1$. We wish to create an intermediate matrix $\mathbf{J}_{.5} = \mathbf{M}_{.5} \mathbf{R}_{.5}$. Since \mathbf{R}_0 and \mathbf{R}_1 are described parametrically, we can create $\mathbf{R}_{.5}$ by interpolating the parameters at the two extremes. But without parametric descriptions for the \mathbf{M} matrices, it is unclear how to create $\mathbf{M}_{.5}$.

In the extraction of the original matrix into the $(\mathbf{M}\mathbf{R})\mathbf{S}(\mathbf{M}\mathbf{R})\mathbf{T}$ template, two \mathbf{M} -type matrices are generated. When two keys are to be interpolated, each of the in-between matrices is built from interpolated key matrices; the \mathbf{M} matrices, though, are not parametric, and cannot be interpolated. If they have the same form at both keys then all is well, since all intermediate \mathbf{M} matrices are the same. But if the endpoint \mathbf{M} matrices have different forms, then it is unclear how intermediate keys should be built.

If the modeler does not introduce inversions or mirrors into the model, then I believe that both \mathbf{M} matrices will always have the same form at successive keys, making interpolation easy. I cannot prove this now, but I have run the algorithm on several hundred randomly-composed test keys that obeyed the above rules, and I never found a pair of mis-matched \mathbf{M} matrices.

But hoping for continued success of an underanalyzed algorithm is a risky proposition. A better way to handle the problem is to remove the parameter-free \mathbf{M} matrices from the template. In the paper, the \mathbf{M} matrix is extracted by considering the cross product of the first two rows of the corresponding \mathbf{R} matrix and the third row of the composite matrix being decomposed. By working instead with columns we may match $\mathbf{J} = \mathbf{R}\mathbf{M}$. If we apply this technique to the first orthonormal matrix from SVD, we get the template $(\mathbf{R}\mathbf{M})\mathbf{S}(\mathbf{M}\mathbf{R})\mathbf{T}$, which we may re-write as $\mathbf{R}(\mathbf{M}\mathbf{S}\mathbf{M})\mathbf{R}\mathbf{T} = \mathbf{R}\mathbf{H}\mathbf{R}\mathbf{T}$. In general, \mathbf{H} will represent simultaneous shears of all 9 varieties in 3d. Since \mathbf{H} represents a composite shear matrix we may interpolate its elements directly. This technique removes the difficulty of working with the parameter-free \mathbf{M} matrices, but what kind of motion it would generate is not clear.

This straightforward paper advocates a simple solution to a common difficulty in modeling and animation systems. Test animation produced with this template has appeared qualitatively similar to that built from animation derived from templates matched by the modeler.

Late Binding Images

Submitted to *IEEE Computer Graphics & Applications*

This paper proposes a scheme for separating scan conversion from shading; the term *shading* covers hidden surface removal (with possible transparency) and surface coloring. The approach is a classic space/time tradeoff: to reduce the total rendering time, the first few steps in a classical rendering pipeline (transformation and scan conversion) are performed only once and the results stored in a large file. As long as the viewpoint remains unchanged, the contents of this file may be used directly for shading, avoiding repetition of those first few pipeline steps.

This is useful when trying to develop a complex image composed of many objects. To make an image informative and visually pleasing, typically one must carefully tune the placement and coloring of lights, surface colors and reflectivities, and object transparencies. This adjustment is often repeated many times until all these parameters are both individually appropriate and collectively harmonious.

In the paper I make the argument early that separation of scan conversion and shading is desirable; the remaining bulk of the paper describes my implementation, rather than an argument for the approach.

Previous work

Separation of scan conversion and shading is essentially a simple idea. Others have previously advocated various approaches to separating the steps of scan-conversion and shading; unfortunately many of the appropriate references do not appear in the text.

Probably the first description of the idea was given by [Crow74], who discussed hardware implementations for storing multiple objects per pixel. The "raster testbed" in [Whitted81] included provisions for storing spans of scan-converted objects. These spans could be individually adjusted with external procedures to effectively change the description of each object when constructing an image. A system is described [Atherton81] which stored several objects at each pixel, principally for determining various combinations of CSG operations on the objects. This system allowed very fast iterative rendering by avoiding repeated scan conversions.

Fast interaction of shading parameters and light sources has been implemented through colormap modification. Interactive texturing is possible by encoding the surface normal of the closest object at a pixel in the pixel's color fields, a process called *normal encoding*, discussed by [Sloan79], [Bass81], and [Heckbert88]. Interactive

adjustment of the texture is accomplished by modifying the colormap, which can be done very quickly. These latter schemes suffer from the drawbacks of very low resolution of the surface normal orientation, and lack of support for transparency. Shading in response to a moving light is described in [Holmes85].

Point sampling

In the section on Packet Structure I consider the problem of a point-sampling scheme that has some anti-aliasing information. In particular, the LBI system can store a bitmask with each packet, giving the surface coverage of that object. But the Z information is presumed to describe the depth of the surface at the pixel center. Addressing this subject, the paper says "The surface normal and depth are computed at the pixel center; if the primitive does not cross the pixel center we estimate the surface normal and Z depth as if it did, by extending the geometry of the surface." Is this justifiable?

I believe so, but the statement should have been restricted to polygons. Although the LBI renderer was developed to handle any primitives after scan-conversion, much of the scan-conversion discussion is implicitly focused on polygon rendering. The environment in which the system was developed and in which it is mostly used is a polygon environment, and some work went into handling polygons efficiently. The extension of a polygon's geometry to cover a pixel center involves only linear extension of the polygon's surface, which can be performed very accurately. In effect we are guessing where the surface would be if it extended as far as the pixel center, but since polygons are planar our guess can be excellent.

An alternate estimate of the surface depth at the pixel center is to use the Z depth of the surface at its closest approach to the pixel center. Unfortunately, this can result in a choppy, zig-zag edge, particularly where two surfaces interpenetrate. The situation can become complex and subtle when dealing with highly curved surfaces. Thus it would have been more proper to restrict the earlier statement to polygons.

Another solution might be to store Z values at each corner of the pixel, rather than the center, as in [Duff86]. Of course, this is only a matter of convention and costs no more storage, but it has the advantage of giving us four pieces of depth information per pixel rather than only one. If all four corners are not covered then we must again extend the surface to estimate the depth at those points.

Texture and shading

To handle texture the system saves “the side lengths of the smallest box in texture space containing the projection of the associated pixel.” This is to facilitate texturing with sum tables.

In the section called Image Generation I refer to a modified version of Phong’s shading equation; the difference is that we handle transparency in the manner described in the text.

Light sources are restricted to those “infinitely far away”. This is because we wish to shade using pre-computed tables, which give illumination intensity as a function of surface normal. Such tables are only useful for infinite light sources; local sources must have their position included in the shading calculation.

When discussing the tables, the article states that “...we routinely produce images for which 32 steps of interpolation is insufficient...”; this is true but not explained. A table at 64-by-64 resolution with 32 steps of interpolation gives us up to 2048 unique shades when interpolating between two antiparallel normals. We felt that this gave us a safe margin when building 512-by-512 images. However, when working with medical images users often wanted to zoom in on some structures of interest. Magnifications of 8- and 16-fold were common in our community, which resulted in normal quantization which assigned the same shade to groups of 2 or 4 pixels. These shades would form clearly visible bands around the object. Thus we elected to provide 64 steps of interpolation, and left the code amenable to changes for further resolution. The issue was critical to our timing because each additional bit of interpolation required measurable expense in the Ikonas graphics engine.

The LBI Algebra

There is some discussion of a group algebra to support the LBI system. I believed at the time, and still do, that this is a powerful part of the overall structure of the renderer. The algebra is certainly simple, but it is also powerful. Most importantly, it is a proof that the system works. I have had many experiences where a plausible argument or algorithm has a hidden flaw, only discovered after extensive work. Even mere existence of a working program is not a proof; there could still be subtle cases in which the program suddenly produces unexpected, wrong results. The group algebra proves that no such flaws exist in the system structure. The actual implementation programs may have bugs, but the intellectual structure is sound.

Implementation Details

We state that "...a Z depth of 0 would be exactly in the image plane..."; this is because of how we set up the transformation at the start of the pipeline. More precisely, we transform 3d objects so that the viewing plane is positioned at $Z=0$ in the transformed world. This is useful because we use packets at $Z=0$ as identity elements in the algebra.

The discussion of matting is correct, but incomplete. When composing two LBI files, we can invoke a 3d transformation operator, corresponding to one of the 2d operators such as *Dissolve* and *Opaque* [Porter85]. The simplest is a 3d windowing operator that restricts inclusion of the new file to within a volume defined by another file. In this spirit, we can follow the lead of the conceptual structure developed to describe the "rgba" [Porter85] and "rgbaz" [Duff86] file formats, and describe LBI files using an "onz" format (for object-tag, normal, z depth).

When discussing repeated elements, I say that there are other solutions that preserve the group properties. Another approach that would work includes a packet count field in each packet; rather than duplicating a packet we just increment the count field; subtracting a packet decrements the count. This approach does not expand the file when the same LBI file is added to itself; the approach in the paper would double the file size in such a situation.

The final sentence in the algebra section might be seen as implying that fog and depth cueing are synonymous; they are not. Fog simulation reduces contrast as a function of distance from the eye, as a result of scattering due to particulate matter in the air. Depth cueing refers to a variety of techniques for representing depth information to the viewer; fog is one such technique. In vector and point-plotting displays, fog may be inexpensively approximated if one assumes a black background; then contrast reduction can be achieved with simple intensity reduction.

Paging and polygon scan converting

The scan conversion section explicitly discusses some efficiency techniques I used for polygon rendering. Much of the discussion focuses on the notion of page allocation and page faulting; unfortunately, some imprecision in the discussion weakens the conclusions. As mentioned previously in this chapter, locality on the screen often translates into locality in memory. Thus a working set of pages can include the core representation of a piece of screen memory. The text implies that each time we change

pages we suffer a page fault; of course, this is not true - we fault only when we access memory outside of the working set. Because polygons affect locally dense regions of the screen, it's likely that two polygons adjacent in the model will be adjacent in the screen, and thus share some pages.

Despite the lack of precision in the presentation, the argument presented in the text for the allocation of rectangular blocks of screen memory per page is valid. To see this, consider the tiling process that generates the polygons in our medical system. It generates rings of polygons that surround an object. Now consider viewing an object so that one of these rings runs vertically; that is, its central axis is horizontal. Adjacent polygons will be vertically adjacent, as in Figure 2.

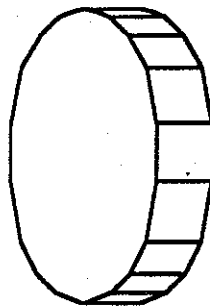


Figure 2

In this figure, the boundaries between polygons are shown as horizontal edges. Successive polygons will thus have few scanlines in common. If we store each scanline on its own page, then once we have filled the page store, each new polygon will require a new scanline, and we will indeed page fault and need to access a new page from memory.

The essence of the problem is that polygons tend not to be oriented wide and short, like scanlines, but rather in small regions of the image plane without preferred orientation. This has been noted before, and has even served as the basis for hardware design [Sproull83].

The solution advocated in the paper is similar to the hardware solutions, in that it allocates memory in small blocks with comparable side lengths (we use a ratio of 2:1 to match our page size; hardware solutions typically use 1:1). Part of the mechanism of accessing these blocks is not well described in the paper.

Memory for packets is allocated sequentially, although we allocate enough at a time to fill one of the 16-by-8 blocks. For each pixel, we maintain a pointer to the head of

the packet list, and a pointer to the last packet in the list; the latter lets us inexpensively extend the packet list. This is illustrated in Figure 3.

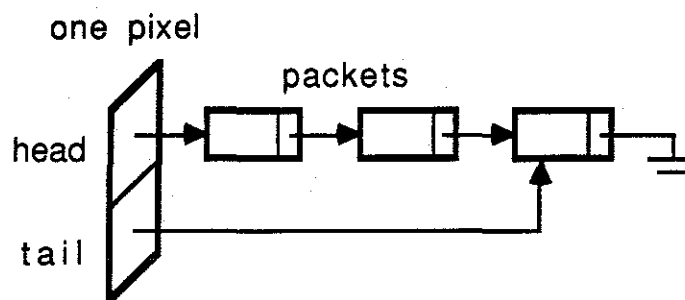


Figure 3

This is a cheap approach while scan converting, but it becomes expensive when writing the complete list out to disk. We prefer this distribution of expenses, since scan conversion is repeated many times, while disk writing happens but once.

The allocation of packets is shown in Figure 4, where for clarity we assume packets are allocated in 2-by-2 blocks, rather than the 16-by-8 blocks used in the system:

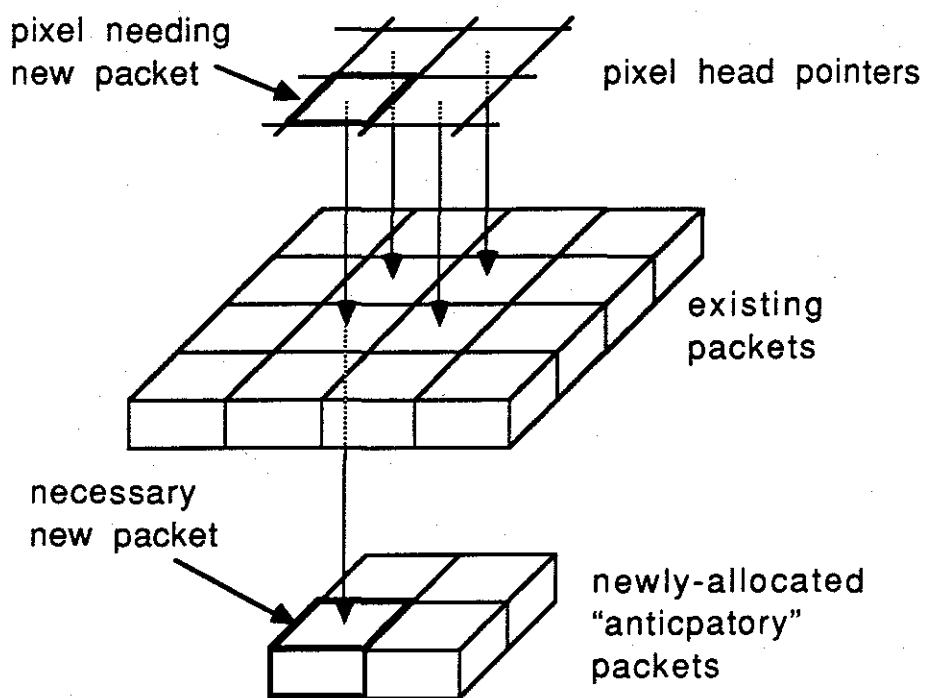


Figure 4

Here one packet was demanded to extend the list for a particular pixel; we allocated an entire block of packets, though, in anticipation of upcoming demands.

Comments

The LBI system began as an architecture, file format, and group algebra. From these components it turned into a complete rendering system that indeed sped up our image production time. The figures reported in the paper show improved rendering time for scenes that were rendered multiple times from a single point of view.

In practice, the size of our images in core (a function of the number of primitives and the resolution of the image) becomes very large very fast, and can easily overwhelm the virtual memory manager. A working solution would be to write sections of the image to disk when virtual memory begins to fill, and then clearing the packet lists. The various files would then be merged together into a single, larger LBI file after scan conversion is complete. This bears similarity to the processing and disk-writing order of the Reyes rendering system [Cook87].

The LBI paper describes the design and construction of a rendering system. As a case study it presents an interesting discussion of the tradeoffs and efficiency concerns encountered. Perhaps the paper's strongest contribution is the group algebra, which gives the system a sound theoretical footing. Defining and following this algebra gave the project focus and coherency of concept, and gave both the author and users confidence in its proper behavior in new situations.

Spacetime Ray Tracing for Animation

IEEE Computer Graphics & Applications, vol. 8, no. 3, March 1988

This is the most recent paper in this dissertation. In fact, this paper is really two papers in one: one paper on a new hierarchy structure, and another on four-dimensional spacetime ray tracing. I did not see this distinction until the first draft of the paper was finished; rather than separate it into two smaller papers, I left them combined.

This may have been a mistake, since it could give the impression that the two methods are linked in some way. In fact, the bounding volume hierarchy and the 4-d ray tracing are completely independent ideas, and either one may be used with or without the other.

Besides this major organizational point, I believe the problems in this paper are minor, and mostly in the presentation, not the algorithms.

Presentation

The survey of single-image ray tracing acceleration techniques surveyed some of the field at the time it was written. Possibly beam tracing [Heckbert84] and cone tracing [Amanatides84] should have been included; for simple databases these techniques may produce anti-aliased images more quickly than point sampling. Since the time that survey was written, pencil tracing [Shinya87] and ray classification [Arvo87] have been presented as additional speedup mechanisms.

In the section introducing the hybrid bounding volume technique I state "... the definition and construction of good hierarchies is still poorly understood." At the time of writing I believed this was generally accepted in the field. Nevertheless, it should have been more clearly labeled as an opinion.

Figure 3 is missing a horizontal line representing a bound separating the upper third of the upper-leftmost object. The corrected Figure is given here. The change is the addition of a horizontal bound on the concave, star-shaped polygon in the upper left.

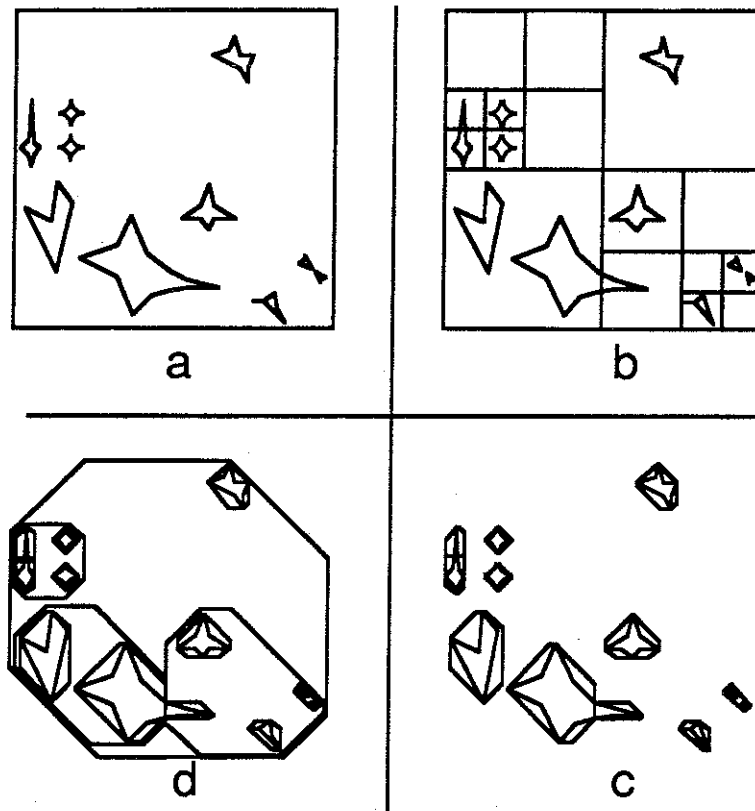


Figure 4 does not show interpenetration of objects. It would have been a good idea to show two spacetime prisms passing through one another, indicating that the techniques support objects passing through one another in space.

Both photos in Figure 7 should be flipped horizontally, exchanging left for right.

Performance

The performance figures given in Tables 1 and 2 show an increase of modest size; the new technique required about 53% and 79% of the time required by other techniques. The discussion called these “significant,” which is perhaps overstating the case.

Why are the savings so small? I believe this is simple an artifact of the test cases. The power of the spacetime algorithm comes from early rejection of intersection events with moving, deforming objects. The savings comes about because the object position and deformation need not be calculated for each event; the spacetime bounds are built up front, one time for the entire animation. The more complex the movement and deformation, the greater the savings will be from the pre-processing step that builds the bounds.

In these test cases all motion was linearly interpolated from keyframes. The only objects to undergo deformation were most of the spheres in the animation of the article's Figure 8 (titled *Dino's Lunch*), and their only change was a varying radius. In the atomic ballet, there was no deformation at all. With such simple motion and deformations, there was no chance for the rejection mechanism to display a savings. Had I created an animation with more sophisticated motion or deformations, I expect that the savings would have been much greater. To test this I have started work on a new animation called *Dino and the Windmill*, the sequel to *Dino's Lunch*.

As discussed in the paper, this expectation of increased performance is supported by experience with the space subdivision technique described in the first paper of this dissertation; in both cases, the more complex the database, the greater the savings. Simple databases will display small savings. Unfortunately for the spacetime paper, I did not build a sophisticated test case involving complex (perhaps procedural) motion and deformations.

In retrospect, it might have been wiser to hold off publication until more dramatic results had been obtained, since the algorithm appears to have acquired a reputation for yielding only modest savings. I believe the reputation is inaccurate.

Summary

These six papers represent several different approaches to different problems in computer graphics. The most important papers in terms of new ideas and algorithms are the first, *Space Subdivision for Fast Ray Tracing*; second, *Adaptive Precision in Texture Mapping*; and last, *Spacetime Ray Tracing for Animation*. Each of these papers makes either a theoretical or practical contribution to the field of realistic image generation.

Each of the six papers uses geometry as a solution technique; sometimes the geometry is in the data structures, other times it is in the flow of the algorithm. This is not very surprising, since most of the problems studied were geometric in nature.

The analyses have pointed up a lack of precision and rigor in most of the papers. The discussion in this chapter has attempted to compensate for that lack, and gives a level of analysis I would like to strive for in the future.

I feel that these papers represent useful and creative approaches to a variety of interesting problems in computer graphics; the collection has breadth and depth.

References

- [Amanatides84] Amanatides, John, "Ray Tracing with Cones", *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18, no. 3, July 1984, pp. 129-135,
- [Amanatides87] Amanatides, John, and Andrew Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", *Eurographics '87*, North-Holland, Amsterdam
- [Arnaldi87] Arnaldi, Bruno, Thierry Priol, Kadi Bouatouch, "A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes", *Visual Computer*, vol. 3, 1987,
- [Arvo87] Arvo, James, and David Kirk, "Fast Ray Tracing by Ray Classification", *Computer Graphics (SIGGRAPH '87 Proceedings)*, vol. 21, no. 4, July 1987
- [Atherton80] Atherton, Peter R., "A Method of Interactive Visualization of CAD Surface Models on a Color Video Display", *Computer Graphics* vol. 15, no. 3, Proceedings of Siggraph '81
- [Bass81] Bass, Daniel H., "Using the Video Lookup Table for Reflectivity Calculations: Specific Techniques and Graphic Results", *Computer Graphics and Image Processing*, vol. 17, no. 3, Nov. 1981, 249-261
- [Cleary87] Cleary, John G., Geoff Wyvill, "An Analysis of an Algorithm for Fast Ray-Tracing using Uniform Space Subdivision", Research Report 87/264/12, U. of Calgary, Dept. of CS, 1987
- [Cook87] Cook, Robert L., Loren Carpenter, Edwin Catmull, "The Reyes Image Rendering Architecture", *Computer Graphics (SIGGRAPH '87 Proceedings)*, vol. 21, no. 4, July 1987
- [Crow74] Crow, Franklin, "Expansions on the Frame Buffer Concept", personal communication (also presented at a Computer Science Seminar at the University of Utah, 6 February 1974)
- [Dippé84] Dippé, Mark E., John Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18, no. 3, July 1984, pp. 149-158
- [Duff86] Duff, Tom, "Compositing 3-D Rendering Images", *Computer Graphics* vol. 19, no. 3, Proceedings of Siggraph '86

- [Fujimoto85] Fujimoto, Akira, Kansei Iwata, "Accelerated Ray Tracing", *Computer Graphics: Visual Technology and Art (Proceedings of Computer Graphics Tokyo '85)*, Toshiyasu Kunii ed., Springer Verlag, Tokyo, 1985, pp. 41-65
- [Fujimoto86] Fujimoto, Akira, Takayuki Tanaka, Kansei Iwata, "ARTS: Accelerated Ray-Tracing System", *IEEE Computer Graphics and Applications*, Apr. 1986, pp. 16-26
- [Gargantini82] Gargantini, I., "Linear Octrees for Fast Processing of Three-Dimensional Objects", *Computer Graphics and Image Processing*, vol. 19, no. 2, 1982
- [Glassner84] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, vol. 4, no. 10, Oct. 1984, pp. 15-22
- [Glassner88] Glassner, Andrew S., "Spacetime Ray Tracing for Animation", *IEEE Computer Graphics and Applications*, vol. 8, no. 2, March 1988, pp. 60-70
- [Haines86] Haines, Eric A., Donald P. Greenberg, "The Light Buffer: A Ray Tracer Shadow Testing Accelerator", *IEEE Computer Graphics and Applications*, vol. 6, no. 9, Sept. 1986, pp. 6-16
- [Hanrahan86] Hanrahan, Pat, "Using Caching and Breadth-First Search to Speed Up Ray-Tracing", *Graphics Interface '86*, May 1986, pp. 56-61
- [Heckbert84] Heckbert, Paul S., Pat Hanrahan, "Beam Tracing Polygonal Objects", *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18, no. 3, July 1984, pp. 119-127
- [Heckbert86] Heckbert, Paul, "A Survey of Texture Mapping", *IEEE Computer Graphics and Applications*, vol. 12, no. 11, November 1986
- [Heckbert88] Heckbert, Paul, private communication, May 1988
- [Holmes85] Holmes, D., "Three-dimensional Depth Perception Enhancement by Dynamic Lighting", Master's Thesis, Department of Computer Science, UNC-Chapel Hill, 1985
- [Hultquist87] Hultquist, Jeff, private communication, March 1987
- [Jansen86] Jansen, Frederik, "Data Structures for Ray Tracing", L. R. A. Kessener ed., F. J. Peters ed., M. L. P. van Lierop ed., *Data Structures for Raster Graphics*, (Eurographic Seminar), New York, 1986, Springer-Verlag, pp. 57-73
- [Kajiya86] Kajiya, James T., "The Rendering Equation", *Computer Graphics (SIGGRAPH '86 Proceedings)*, vol. 20, no. 4, Aug. 1986, pp. 143-150
- [Kaplan85] Kaplan, Michael R., "Space-Tracing, A Constant Time Ray-Tracer", *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, July 1985

- [Kay86] Kay, Timothy L., James T. Kajiya, "Ray Tracing Complex Scenes", *Computer Graphics (SIGGRAPH '86 Proceedings)*, vol. 20, no. 4, Aug. 1986, pp. 269-278
- [Lathrop88] Lathrop, Olin, "Notes Regarding Ray Tracing with Octrees", *Ray Tracing News*, vol. 2, no. 1, February 1988
- [Lee85] Lee, Mark E., Richard A. Redner, Samuel P. Uzelton, "Statistically Optimized Sampling for Distributed Ray Tracing", *Computer Graphics (SIGGRAPH '85 Proceedings)*, vol. 19, no. 3, July 1985, pp. 61-67
- [Mitchell87] Mitchell, Don, "Generating Anti-aliased Images at Low Sampling Densities", *Computer Graphics* vol. 21, no. 4, Proceedings of Siggraph '87
- [Murakami83] Murakami, Kouichi, Hitoshi Matsumoto, "Ray Tracing with Octree Data Structure", *Proc. 28th Information Processing Conf.*, 1983
- [Nemoto86] Nemoto, Keiji, Takao Omachi, "An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing", *Graphics Interface '86*, May 1986, pp. 43-48
- [Newman79] Newman, William M. and Robert F. Sproull, "Principles of Interactive Computer Graphics, 2nd Edition", McGraw-Hill Book Co., 1979
- [Peng87] Peng, Q. S., "A Fast Ray Tracing Algorithm Using Space Indexing Techniques", *Eurographics '87*, North-Holland, Amsterdam
- [Porter84] Porter, Tom, and Tom Duff, "Compositing Digital Images", *Computer Graphics* vol. 18, no. 3, Proceedings of Siggraph '84
- [Rubin80] Rubin, Steven M., Turner Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes", *Computer Graphics (SIGGRAPH '80 Proceedings)*, vol. 14, no. 3, July 1980, pp. 110-116
- [Shinya87] Shinya, Mikio, Tokiichiro Takahashi, and Seiichiro Naito "Principles and Applications of Pencil Tracing", *Computer Graphics (SIGGRAPH '87 Proceedings)*, vol. 21, no. 4, July 1987
- [Sloan79] Sloan, Kenneth R., Jr., Christopher M. Brown, "Color Map Techniques", *Computer Graphics and Image Processing*, vol. 10, no. 4, Aug. 1979, 297-317
- [Sproull83] Sproull, Robert F., Ivan E. Sutherland, Alistair Thompson, and Charles Minter, "The 8 by 8 Display", *ACM Transactions on Graphics*, vol. 2, no. 1, January 1983
- [Ullner83] Ullner, Mike K., "Parallel Machines for Computer Graphics", PhD thesis, California Institute of Technology, 1983

- [Weghorst84] Weghorst, Hank, Gary Hooper, Donald P. Greenberg, "Improved Computational Methods for Ray Tracing", *ACM Trans. on Graphics*, vol. 3, no. 1, Jan. 1984, pp. 52-69
- [Whitted81] Whitted, Turner, and David M. Weimer, "A Software Test-Bed for the Development of 3-D Raster Graphics Systems", *Computer Graphics* vol. 15, no. 3, Proceedings of Siggraph '81
- [Wyvill86] Wyvill, Geoff, Toshiyasu L. Kunii, Yasuto Shirai, "Space Division for Ray Tracing in CSG", *IEEE Computer Graphics and Applications*, vol. 12, no. 4, Apr. 1986, pp. 28-34,

About the Type

This dissertation was prepared on an Apple Macintosh II using Microsoft Word 3.01. The main body of the text (and all papers not appearing as reprints) is 12 point Times. Figures were created with MacPaint 1.5 and MacDraw 1.9.5. Equations were prepared with Expressionist 1.11. Camera-ready output was generated on a LaserWriter II NTX with a resolution of 300 dots per inch using fonts from Adobe.

Space Subdivision for Fast Ray Tracing and *Spacetime Ray Tracing for Animation* were prepared on a VAX-11/780 running Unix BSD4.2. The text was entered and edited with the vi display editor, and formatting was prepared with nroff. Final typesetting for both papers was prepared by the journal. Figures for *Spacetime Ray Tracing for Animation* were printed by the journal directly from originals prepared by the author using MacDraw 1.9.5.

Adaptive Precision in Texture Mapping was also written with the VAX/Unix system, but used the T_EX typesetting language for formatting. Output was generated at 120% on an Imagen printer, and photoreduced for journal publication. Figures were created with MacDraw and inserted into the manuscript by hand during pasteup.

All computer-generated images in all papers were shot with a tripod-mounted 35mm camera. The monitor was a Tektronix 69M41, displaying the output of an Adage/Ikonas RDS-3000 graphics system. Gamma correction for all figures was performed in the colormap.

dulce est desipere in loco

(sweet it is to rest at the proper time)