USING CACHING AND BREADTH-FIRST SEARCH TO SPEED UP RAY-TRACING

(extended abstract)

Pat Hanrahan

Abstract

Ray-tracing is an expensive image synthesis technique because many more ray-surface intersection calculations are done than are necessary to shade the visible areas of the image. This paper extends the concept of beam-tracing so that it can be coupled with caching to reduce the number of intersection tests. Two major improvements are made over existing techniques. First, the cache is organized so that cache misses are only generated when another surface is intersected, and second, the search takes place in breadth-first order so that coherent regions are completely computed before moving onto the next region.

Introduction

Ray-tracing has attracted considerable attention recently because of the super-realistic images that can be produced. Lighting and shading effects that require information about the global environment, such as shadows, reflections and refractions, can be calculated by recursively tracing rays from the surfaces they intersect [Whitted, 1980]. Distributed or stochastic ray-tracing can be used to simulate other optical effects, such as motion blur, finite-sized light sources, prismatic effects, etc., and to remove many of the artifacts due to point sampling the image [Cook, Porter and Carpenter, 1984]. Ray-casting can also be used to generate line drawings and sectioned views, and to perform the volume integrals needed for the calculation of mass properties [Roth, 1982]. Another advantage of ray-tracing is that it is conceptually elegant and easy to implement. The models that comprise the scene can be rendered if a procedure to intersect a ray with their surfaces is provided. Because of the object-oriented architecture, a ray-tracing system is easy to maintain and extend. The number of geometric primitives that can be ray-traced is quite large and continues to grow.

The major disadvantage of the standard ray-tracing algorithm is that the time needed to generate an image is equal to the number of geometric primitives *times* the size of the output image. This is because when an individual ray is being traced all the objects in the scene

need to be tested to check for an intersection. As a result there are many more ray-surface intersection calculations performed than there are rays intersecting visible surfaces. Several approaches have been attempted to reduce the number of intersection tests. Rubin and Whitted [1980] use a hierarchical tree of bounding boxes to describe a scene. Since the bounding volumes of the children lie entirely within the bounding volumes of the parent, the child volumes need only be searched if the ray intersects the parent volume. An alternative approach is to decompose space into a set of disjoint volumes. Each volume in the subdivision contains a list of those surfaces contained within it, and the subdivision is made fine enough so that the total number of objects in each volume is a small number. The search for an object intersection proceeds along the path of the ray through the subdivision. Two different subdivision methods have been reported. One method decomposes space into a rectangular array of voxels. In this case the volumes along the path of the ray can be determined using a 3-d incremental line drawing algorithm [Fujimoto and Iwata, 1985; Haeberli, 1985]. The second method decomposes space with an oct-tree [Glassner, 1984; Kaplan, 1985]. In this case, determining the next volume in the path is more complicated, but this disadvantage is offset by the fact that the oct-tree decomposition takes less space for a given level of detail.

In this paper we develop another method for speeding up the search for ray intersections by combining two methods previously reported in the literature: *beam-tracing* [Heckbert and Hanrahan, 1984] and *coherent ray-tracing* [Speer, DeRose and Barsky, 1985]. We discuss how the concept of a beam-tree can be used to characterize the coherence contained in an image. The beam-tree also suggests that the ray-surface intersections should be searched for in breadth-first order, that is, all the intersections with a given surface should be found before proceeding to the next surface. We also discuss several new methods for caching rays.

The Beam Tree

Heckbert and Hanrahan [1984] described a method to trace beams through a scene consisting of polygonal objects. This method was based on the observation that neighboring rays have essentially the same object intersection tree. This coherence can be quantified by introducing the notion of the beam-tree. In a ray-tree (as described in Whitted [1980]), the links represent rays of light and the nodes represent the surfaces that those rays intersected. Similarly, the links in a beam-tree represent beams of light, and the nodes contain a list of surfaces intersected by a beam. Each of the surfaces intersected by the beam spawns new beams corresponding to reflections, refractions and shadows.

In Heckbert and Hanrahan [1984] the beams of light were pyramidal cones. The original beam was the viewing pyramid and since the objects in the scene were polygons, all the secondary beams also had polygonal cross-sections. In the case of reflection and shadows, and under certain assumptions, refraction, it was shown that the new beams were also pyramidal cones -- that is, they contained a single apex. These restrictions allowed an entire beam to be traced at once by using a recursive polygonal hidden surface algorithm similar to that described in Weiler and Atherton [1977].

In case of curved surfaces or of true refraction, the form of a beam changes drastically when it interacts with a surface. Therefore, it is difficult to devise an algorithm to trace all the rays contained within it simultaneously. However, as we will demonstrate, it is still possible to take advantage of the coherence of a beam. In the general case we define a beam as a set of rays that all originate from the same object (or from the same point) and all intersect the same object. Generally, rays grouped together into beams will belong to adjacent pixels, although this is not strictly required. For example, all the rays through the eye that hit the background may

be considered a single beam even though the regions comprising the background may not be connected. A beam under this definition need not be uniform, for example, it might contain caustics and other singularities. An example of a coherent beam that contains a singularity is one which passes through a lense or into a crystal ball.

Notice that the beam tree for a given image (scene plus point of view) is independent of the method used to generate it. Given the ray-trees for all the pixels in the scene, the beam-tree could be created as a post-process by recursively merging adjacent rays if they intersect the same surface. The size of the beam-tree is a natural measure of the intrinsic coherence in an image.

$$\text{coherence} = \frac{\text{average ray-tree size}}{\text{total beam-tree size}}$$

Where the average ray-tree size is the total number of nodes in the ray-tree divided by the number of pixels. If at each level of the tree all the rays could be coalesced into a single beam then the size of the beam-tree would be the same as the average ray-tree size. This would be the maximum coherence possible. If the beam-tree is any bigger, this implies that adjacent ray-trees could not be merged, resulting in a new sub-beam, and therefore, less coherence. Notice that using this definition, the coherence does not depend on the relative sizes of the different sub-beams. For example, an image with one large beam and two small beams has the same coherence as an image with three equal-sized beams.

The amount of work needed to compute an image is the sum of two factors: shading and intersection processing. The number of calculations to shade the image is proportional to the size of the image times the average size of the ray tree. The optimal number of calculations needed to compute the ray-tree at each pixel is proportional to the size of the beam-tree. Optimistically, the cost of computing each node in the beam-tree would be proportional to the number of objects in the scene. Thus, coherence allows us to decouple the complexity of the intersection phase of the calculation (which is most sensitive to the number of objects in the scene) from the shading phase of the calculation. (which is most sensitive to resolution of the image). In particular, notice that in standard ray-tracing the cost per pixel is multiplied by the number of objects, whereas using beam tracing this cost is amortized over the average number of pixels per beam. Thus, beam-tracing wins big at high resolutions.

Caching

Speer, DeRose and Barsky [1985] described a method to speed up ray-tracing which they term coherent ray-tracing. Their method is based on the same observation as contained in Heckbert and Hanrahan [1984]: that adjacent rays have a high probability of intersecting the same objects. However, instead of attempting to trace many rays simultaneously, they save the ray-tree corresponding to the previous ray and use it to guide the next intersection test. The ray-tree is intended to act as a cache. A cache hit occurs if the next ray intersects the same surface; a miss occurs if another surface is intersected. The cache is complicated by the fact that, although the same object may be hit by the next ray, another object may block the ray before it hits that object. They solved this problem by using a cache with two types of information: the last object intersected and a cylindrical region of safety. The cylinder of safety is the largest region surrounding the ray which does not contain any other surfaces. When a cache hit occurs, the ray is only tested against the last sphere and the cylinder. Thus, since only two tests need to be done, if there is a cache hit the average cost of computing an intersection per ray is constant within a beam.

We implemented this technique and upon examining caching statistics found that there were many cache misses even though the same sphere was still intersected. This is because the cylindrical region of safety is much smaller than the beam cross-section (see Figure 1). To remedy this, we devised a method which may in some situations require more work, but will cause a cache miss only if the last sphere was not intersected.

Figure 2 shows a situation where a ray hits one sphere and then hits a second sphere. The cache contains the last sphere hit and a list of spheres that could potentially block a ray travelling from the first to the second. Normally this list is empty or contains only a small number of spheres. Any ray originating on the surface of the first sphere that also intersects the second sphere can only intersect objects contained on this list of potential blocking spheres. A cache miss occurs if, first, the ray does not intersect the second sphere, or second, it intersects a sphere contained in the list of blocking spheres. Using this caching system all misses imply that a new object has been intersected. Another advantage of this technique is that no special ray-cylinder intersection tests are required.

There are several ways in which the list of potential blocking spheres is generated. The most common situation is where the ray originated from the eye point or is travelling to the light source. In this case the list of spheres are all those spheres contained in a cone from the point to the sphere that was intersected. The second most common situation is where a ray travels between two spheres. In this case all the spheres that lie within a cone circumscribed around the two spheres and between them are determined. Another method used to generate a list of potential blocking spheres is when a ray enters the interior of a transparent sphere and intersects itself. In this case the list of blocking spheres are all those spheres that intersect the interior of the transparent sphere. It should be mentioned that it is possible to precompute the list of potential blocking spheres for a given scene before an image is generated. However, the naive algorithm to do this is of $O(n^3)$.

This new method works well for rays that travel from sphere to sphere, from point to sphere, or from sphere to point, but does not work if a ray doesn't intersect any objects. In this case the original method due to Speer, DeRose and Barsky should be used.

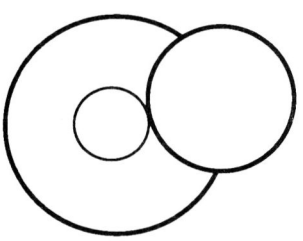

Figure 1 - This figure shows two spheres (in bold) and a cylinder of safety viewed along the axis of the ray (marked with a +). Notice that the cylinder is much smaller than the visible part of the large sphere.

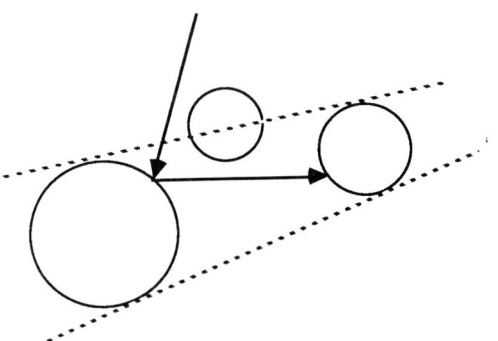

Figure 2 - This figure shows a ray reflecting off the large sphere on the left and hitting the smaller sphere on the right. Around these spheres is a cone which contains a single sphere which might potentially block another ray travelling between the same two spheres.

Breadth-first Search

Caching will work effectively if the searches are ordered in a way which maximizes the probability that a cache hit will occur. For example, if rays were randomly chosen from different pixel locations then we would expect few cache hits. Thus, the effectiveness of caching depends both on the intrinsic coherence in the scene and the search strategy employed. Fortunately, we are at liberty to reorder the search. This is analogous to the situation encounted in optimizing compilers where instruction execution order is rearranged to maximize the number of cache memory hits. The goal is to use knowledge about the general properties of the beam-tree so that searches for ray-surface intersections can be ordered in a way that maximizes the probability of cache hits.

The first important point is that the cache should be organized as a tree. There is little reason to suspect that a reflected ray will hit the same object as the refracted ray or the incident ray. Practically this means that if a cache miss occurs at a parent node then we should flush the cache of all the child nodes.

The second and major point is that the beam's cross-section is two-dimensional, not one-dimensional. Consider the simplified case where the ray-tracer is only being used to remove hidden surfaces, so that there are no reflected or transmitted rays. In the standard ray-tracer, rays are generated in scanline order. The number of cache misses per scanline is equal to the number of regions crossing that scanline. Each cache miss causes all the objects to be searched. The total number of complete searches is therefore much greater than the total number of regions. In order to achieve one complete search per region the search should continual two-dimensionally until a cache miss occurs. This is similar to the common seed fill or boundary fill algorithm used in paint systems [Smith, 1979]. This search method also works when the tree has greater depth. If we imagine the complete ray-tree as including all the rays emanating from the eye point, the region fill corresponds to a breadth-first search of this tree.

The initial reaction to breadth-first search is that since images are so large, the size of the list of rays queued would be prohibitively large. However, it is possible to organize the search in scanline order so that the list is kept to a reasonable size.

Results

To test these ideas we implemented a simple cached ray-tracer for spheres. The code was written so it was easy to turn caching on or off. This program was run over a variety of different scenes with similar results. In the table below the scene consisted of a NxNxN array of spheres whose centers and radii were randomly jittered. This cube of spheres was then viewed from an angle and scaled so that it filled the screen. As can be seen from Table 1, caching itself sped up the ray-tracer from 2-5 times; adding breadth-first search sped it up by another factor of 2-3.

Number of spheres	3^3	4^3	5^3
Not cached	1.00	1.00	1.00
Cached	.53	.29	.20
Cached, breadth-first	.25	.15	.11

Table 1 - Timing results

Conclusions

Although these results are preliminary, this area of research look promising. Analyzing the caching statistics and comparing them to the actual coherence as measured by the beam-tree we find that the number of complete searches is still much more than the theorectical maximum. In particular the cache hit test is not very

successful when a ray had not previously hit an object. Improving this case would significantly speed up the program.

Spheres were chosen in this study because the intersection tests are easy to implement and because spheres can also be used to bound the extent of other object types. An avenue for further research is to devise cache tests for other object types. In the general case it may be worthwhile to allow different caching strategies for different objects. For example, the polygons in a convex polyhedral solid cannot occlude other polygons of that solid. Thus, these polygons cannot be on the list of potential blockers. Sometimes even in the case when a search through all the objects in the scene need be done, a cache can be used to speed this up. Considering again the case of a convex polyhedral solid, we can cache the last polygon hit. If that polygon is missed by the next ray, then we should search polygons adjacent to it first.

Caching is a very general method for speeding up computations when coherence exists. For this reason it can be used along with other methods, such as cellular decomposition, to speed up the search for ray-surface intersections. It is also likely that many other hidden surface algorithms would benefit from caching.

Finally, the idea of breadth-first search was originally motivated by the desire to build an interactive ray-tracing tool. If each ray is immediately painted onto the image after it is traced, then the details of the image will gradually be filled in. First, the a hidden surface view will be drawn, followed by reflections and shadows at greater and greater depth.

Acknowledgements

Some of this work was done while the author was at the University of Wisconsin and a member of the research staff of the Digital Equipment Corporations Systems Research Center. Thanks to Paul Haeberli for his insight and enthusiam.

References

Cook, R., Porter, T. and Carpenter, L., Distributed ray tracing, *Computer Graphics,* (SIGGRAPH '84 Proceedings) 18(3), pp. 137-145, 1984.

Fujimoto, A., and Iwata, K., Accelerated ray-tracing, *Computer Graphics*, Tokyo, 1985, pp. 1-26, 1985.

Glassner, A.S., Octree encoding to speed up ray tracing, *IEEE Trans. Comp. Graphics and Appl.,* 4(10), pp. 15-22, 1984.

Haeberli, P., (personal communication) 1985.

Heckbert, P.S. and Hanrahan, P., Beam tracing polygonal objects, *Computer Graphics* (SIGGRAPH '84 Proceedings), 18(3), pp. 119-127, 1984.

Kaplan, M., Constant-time ray tracing, Notes for State of the Art in Image Synthesis, Proc. of SIGGRAPH '85, 1985.

Roth, S.D., Ray casting for modeling solids, *Computer Graphics and Image Processing*, 18(2), pp. 109-144, 1982.

Rubin, S.M., and Whitted, T., A 3-dimensional representation for fast rendering of complex scenes, *Computer Graphics* (SIGGRAPH '80 Proceedings), 14(3), pp. 110-116, 1980.

Smith, A.R., Tint fill, *Computer Graphics* (SIGGRAPH '79 Proceedings) 13(2), pp. 276-283, 1979.

Speer, L.R., DeRose, T.D., and Barsky, B.A., A theoretical and empirical analysis of coherent ray-tracing, *Graphics Interface* '85, 1985.

Weiler, K.J., and Atherton, P.A., Hidden surface removal using polygon area sorting, *Computer Graphics* (SIGGRAPH '77 Proceedings), 11(3), 1977.

Whitted, T., An improved illumination model for shaded display, *C.A.C.M.*, 23(6), pp. 343-349, 1980.