Object Space Temporal Coherence for Ray Tracing

David A. Jevans
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California, USA 95014

(jevans@apple.com)

Abstract

A method is presented for exploiting object space temporal coherence to speed up ray tracing of animation sequences where the camera remains static. The object space is subdivided with a hierarchical voxel grid structure. Each voxel keeps a list of the rays that pass through it when the first frame of a sequence is rendered. To render a successive frame, only rays that passed through voxels in which an object has moved are retraced. The method speeds up ray tracing of a test animation sequence by nearly a factor of four.

The method is easily adapted to work with any spatial subdivision technique. The memory requirements of the method are low.

Keywords: Ray Tracing, Frame Coherence, Space Subdivision.

1 Introduction

Ray tracing [Whitted 80] is an elegant technique for synthesizing realistic images. Numerous acceleration methods have been developed to reduce the computational expense of ray tracing, including spatial subdivision [Glassner 84] [Fujimoto 86], hierarchical object extents [Rubin 80] [Kay 86], clustering and sweeping methods [Amanatides 84] [Heckbert 84] [Shinya 87], and ray coherence [Joy 86]. These methods are effective when rendering a single image, but do not make use of the temporal coherence found in animation sequences.

The traditional way to render an animation sequence is to render each frame one at a time, ignoring any object space temporal coherence that may exist between frames. Object space temporal coherence manifests itself as objects, such as floors or walls, that move slowly or remain static throughout the course of an animation.

The aim of the research presented in this paper is to take advantage of object space coherence to speed up ray tracing of animation sequences. To be useful for high quality rendering, the method must produce images that are indistinguishable from those rendered with a traditional one frame at a time approach.

2 Previous Work

Algorithms for exploiting temporal coherence operate either in image space or object space. Image space algorithms reduce the time to render an animation sequence by rendering a subset of the pixels in a frame, and estimating the value of the unrendered pixels. Due to their sampling nature, image space algorithms may generate "incorrect" frames - frames that differ from those rendered with a traditional one frame at a time approach. Object space algorithms use information about the 3D object space, and how it changes between frames, to reduce the amount of computation to render an animation sequence.

2.1 Image Space Temporal Coherence

Badt [Badt 88] proposed a method that reduces the number of rays traced during an animation by tracing the first frame of a sequence normally, then rendering successive frames by retracing only a small random sampling of pixels for each frame. If a retraced pixel's color differs from its color in the preceding frame, a flood fill algorithm that floods both in screen space and in time is used to correct its color. Flooded pixels are retraced for preceding and succeeding frames to determine their correct colors. The flood filling reduces, but does not eliminate, the possibility of incorrect pixel colors. This method requires that the object space description for every frame of the animation sequence be available at all times during the rendering.

Chapman [Chapman 90] developed another image space algorithm that traces fewer rays than Badt's method, but is potentially less accurate. The algorithm renders every kth frame of a sequence, where $k >= 1$. The pixel colors of frame n and frame $n+k$ are compared. If a pixel's color is different in the two frames, then it is retraced at frame $n+k/2$. This process is repeated recursively, resulting in a binary search that determines the frame in which the pixel's color changed. The drawback of this algorithm is that if k is chosen to be large, high frequency changes, such as those caused by fast moving objects, will be lost.

2.2 Object Space Temporal Coherence with a Moving Camera

Hubschman [Hubschman 82] presented a method for exploiting object space temporal coherence when rendering sequences where only the camera moves. The first frame of a sequence is preprocessed to determine object visibility, and successive frames are generated by determining which objects have changed their visibility status. While the technique creates "correct" images, it does not work when objects move during the animation.

2.3 4D Ray Tracing

Spacetime ray tracing [Glassner 88] accelerates ray tracing of animation sequences through the use of hierarchies of 4D bounding volumes that encompass objects as they move through space and time. Rather than building a hierarchy of 3D bounding volumes for each frame in the animation, a hierarchy of 4D bounding volumes is created once for the animation sequence. Rays are represented as a 3D direction vector and a fourth component, their position in time. Rays are traced by testing them for intersection with the 4D bounding volumes in the scene. Only objects that lie within the 4D bounding volumes that are intersected by a ray need to be tested for intersection with it.

The main source of efficiency in this algorithm is that fewer bounding volumes are created for an animation sequence than with a traditional 3D bounding volume approach. This accelerates both the creation of bounding volumes and reduces the number of ray/volume intersection tests required to render a sequence. Motion blur by jittering rays in time is facilitated since the entire animation sequence is available to the renderer at each frame.

One drawback to this approach is that it requires an entire animation sequence to be resident in memory during rendering. The method is also not amenable to a voxel-based spatial subdivision approach. Thirdly, it does not reduce the number of rays that need to be traced at each frame.

Chapman, Calvert, and Dill [Chapman 91] developed a similar algorithm for using hierarchies of bounding volumes of animated objects. The difference with their approach is that objects inside the bounding volumes represent their motions as translation and rotation vectors. The ray/object intersection calculation is extended to encompass intersection with a moving object and to compute all intersections of a ray with a moving object. These intersections are sorted by time and distance along the ray. From this information the colors for a ray are calculated for the entire sequence, and each ray is traced only once for a given sequence.

Disadvantages to this technique are the complexity of intersection calculations and that it may not be readily extensible to handle motion that cannot be represented as simple translation and rotation vectors. Furthermore, the object bounding volumes may become large if an object moves significantly during the animation sequence, reducing the effectiveness of the hierarchical bounding volume approach.

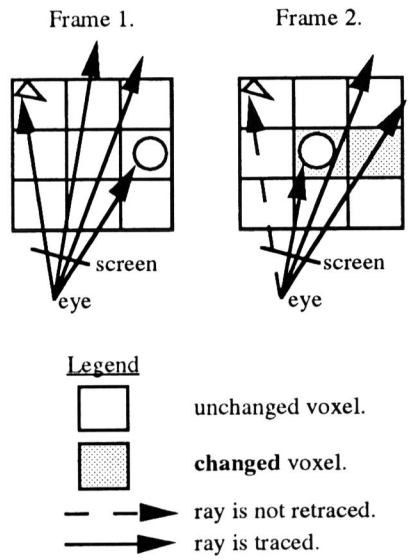


Figure 1.

2.4 Object Space Temporal Coherence with a Static Camera

When the camera remains static during an animation sequence, the color of a pixel can change from one frame to another only if an object that is visible to the pixel has changed, or if an object has moved to become visible to the pixel. An A-buffer scan conversion renderer [Carpenter 84] can make use of this by keeping a list, throughout the animation, of the surface fragments that lie under each pixel. If an object moves between frames, it is deleted from the fragment lists of all pixels. The moved object is then rescanned into the frame buffer in its new position, and is added to the fragment lists of the pixels into which it scans. Pixels whose fragment lists have changed are then re-evaluated to determine their new color values.

Séquin presented a ray tracing algorithm that stores the ray tree at every pixel so that surface attributes of visible objects can be changed without having to retrace the image [Séquin 89]. The method fails when objects change their positions, since it cannot determine if the visible surface for a ray has changed.

Murakami and Hirota [Murakami 90] extended the algorithm to handle animated objects by subdividing the space with a voxel grid, and keeping a list of traversed voxels for every ray in a ray tree. To render a subsequent frame, all objects that move are deleted from the voxel grid, and are reinserted in their new positions. The ray trees are then examined and only rays that traversed through voxels in which an object has moved are retraced (Figure 1). A hashing scheme for representing a ray's path through the voxel space is used to speed up the process of determining which rays to retrace.

The memory requirements of the Murakami and Hirota algorithm are large, typically on the order of tens of megabytes, and grow rapidly as image resolution increases. Computational requirements also grow as a function of image resolution because the ray tree of each pixel must be examined to determine whether the ray passed through a changed voxel. Their algorithm is also specific to uniform voxel subdivision due to its use of a voxel index hashing scheme.

2.5 A New Algorithm

This paper presents an algorithm for making use of object space coherence to speed up ray tracing of animation sequences in which the camera remains static. The algorithm's memory requirements are independent of image resolution, and it is easily adapted to any spatial subdivision scheme such as uniform voxel subdivision, octree subdivision, adaptive voxel subdivision, or 5D space subdivision [Arvo 87].

3 The Algorithm

Rays are tagged with their x,y pixel index in the image frame buffer. As rays are traced through a spatially subdivided scene, each voxel keeps a record of the x,y indices of rays that pass through it. For subsequent frames, when objects inside a voxel move, the voxel notifies the frame buffer of the pixels that will be affected. Only those pixels that are affected are retraced at each frame.

3.1 The Ray Tracer

The ray tracer used to develop this algorithm utilizes an adaptive voxel subdivision scheme [Jevans 89], although any object space subdivision scheme can be adapted to use the algorithm. The object space is subdivided by a voxel grid. Each voxel maintains a list of pointers to the objects that intersect or lie within it. If the number of objects inside a voxel is larger than some threshold, the voxel is itself subdivided with a voxel grid. A set of heuristics, based on the number of objects in a voxel, is used to determine the granularity of the subdivision [Jevans 91].

Space subdivision is done on the fly when a ray first enters a voxel. This lazy evaluation technique ensures that computation and memory are not wasted subdividing areas of the object space that are not visible. To ensure that voxels are only subdivided the first time a ray enters them, they are initially marked as *not subdivided*. When a ray enters a voxel, the heuristics are used to subdivide it, and it is marked as *subdivided*. Newly created sub-voxels are marked as *not subdivided*, as they will be considered for subdivision only if rays pass through them. Voxels marked as *subdivided* are not considered for subdivision when successive rays enter. Note that if the number of objects in a voxel is small, no subdivision may occur, but it will still be marked as *subdivided*.

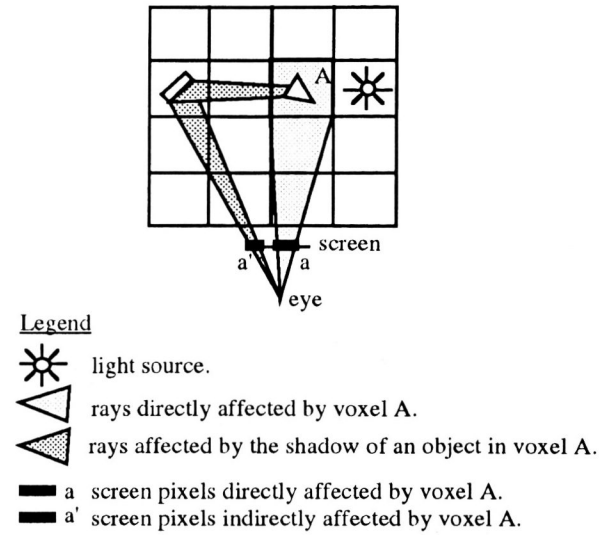


Figure 2.

3.2 Rendering the First Frame

Every pixel in the first frame of the animation sequence is ray traced. Rays are labeled with their originating pixel's x,y frame buffer index. When a ray passes through a voxel, a record of its pixel index is stored with the voxel. This information is stored for all voxels, whether they are empty or not, and whether they are leaf or interior nodes of the subdivision tree.

Each voxel has a 16 by 16 bit-table to store the x,y pixel indices of the rays that pass through it. Each bit represents a block of pixels that occupy $1/256^{th}$ of the screen area. This storage method is independent of image resolution, and only requires 32 bytes of memory per voxel. Higher resolution bit-tables can be utilized if memory usage is not a constraint. Higher resolution bit-tables provide finer granularity of the rays that will be traced at each frame, with little increase in computational overhead. The ideal resolution for the bit-tables is the resolution of the image frame buffer.

When a ray enters a voxel, the bit in the voxel's bit-table that corresponds to the ray's x,y index is set. The voxel's bit-table may represent disjoint areas of the screen. This occurs when an object is visible to both primary viewing rays and to secondary rays, such as shadow or reflection rays, in another part of the screen (Figure 2).

3.3 Subsequent Frames

For each subsequent frame, the object space database and the subdivision structure need updating to reflect changes that have occurred since the previous frame. The entire subdivision tree is traversed, and every voxel is marked as **unchanged**. Objects that change from the previous frame are reinserted into the voxel subdivision tree, and the voxels that they affect are marked as **changed**.

If an object is deleted from the scene, the voxels in which it lay are marked as **changed**, and any references to the object are deleted from these voxels. If an object is added to the scene, the voxels in which it now lies are marked as **changed**, and references to it are added to those voxels. If an object moved or changed shape or surface attributes, both the voxels in which it lay and the voxels to which it moved are marked as **changed**, and references are added and deleted as appropriate. When marking a leaf node voxel as **changed**, the voxels above it in the hierarchy are marked as **touched**.

As long as the number of objects that change is fewer than the number of objects that remain static, the time to resort the changed objects into the subdivision structure is less than to completely rebuild the structure. This speedup is not significant, however, as the total subdivision time of adaptive subdivision algorithms is typically on the order of a few percent of the total rendering time [Jevans 89].

3.3.1 Examine the Voxel Space

The next step is to examine the voxel space to determine which pixels need retracing. A 16 by 16 bit-table representing the frame buffer is created and every bit is initialized to zero. Starting at the top level of the subdivision tree and working down, every voxel is examined. If a leaf voxel is marked as **changed**, the frame buffer bit-table is or-ed with the voxel's bit table. After the entire voxel space has been examined, the bits that are set in the frame buffer bit-table indicate which pixel blocks must be retraced.

All records of the rays that are about to be retraced must be deleted from the voxel bit-tables in case the retraced rays do not pass through those voxels in the next frame. This is accomplished by examining the voxel space a second time and clearing all bits in the voxel bit-tables that are set in the frame buffer bit-table.

The frame can now be generated. Pixels that correspond to the bits that are set in the frame buffer bit-table are retraced. All other pixels retain their color values from the previous frame.

3.3.2 Resubdivision

If the number of objects in a voxel changes significantly from one frame to another, it may be advantageous to resubdivide it. This can be determined during the examination of the voxel space. If a voxel is marked **touched** or **changed**, and the number of objects inside it has changed significantly or gone to zero, its child voxels are recursively deleted, and it is marked as **changed** and *not subdivided*. It will be examined for resubdivision on the fly when rays are being retraced.

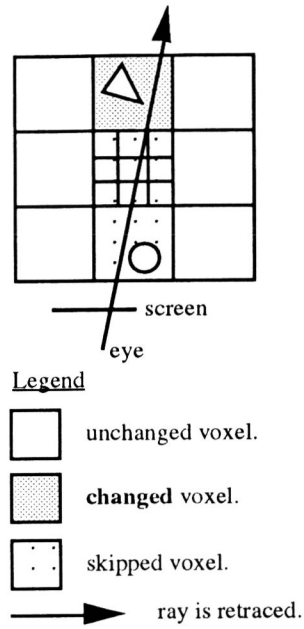


Figure 3.

3.3.3 Ray Traversal Optimization

An optimization to the traversal of viewing rays through the voxel grid can be made by storing the distance along each ray from its origin to its intersection with the visible surface. When a viewing ray is retraced, the voxel traversal algorithm can treat any non-changed voxels as empty if they are closer to the eye than this distance. Neither ray-object intersection calculations nor traversal of sub-grids need be performed for voxels that are not marked as **changed** (Figure 3).

4 Analysis of Animation

Since this algorithm requires that the camera remain static during an animation sequence, it is of interest to know if such sequences constitute a significant portion of computer animation. Table 1 presents statistics for the duration of time that the camera remains static for several well known animations. The timings in Table 1 are approximate however, because they do not account for cuts in the camera's point of view, nor for camera holds, which can be rendered as a single frame and replicated during filming.

For the films analyzed in Table 1, static camera sequences account for a significant portion of the animation. Naturally there are degenerate cases, such as fly-by sequences, where static camera sequences are few or nonexistent. However, for animations that include static camera sequences, a method that accelerates the ray tracing of such sequences can have a significant impact on the overall rendering time.

Animation	Running time	Static camera time	% static camera
Luxo Jr	131 sec	131 sec	100%
Red's Dream	320 sec	259 sec	81%
Tin Toy	451 sec	418 sec	93%
Pencil Test	253 sec	231 sec	91%
The Audition	309 sec	240 sec	78%

Table 1.

5 Results

A sequence from the Apple Computer, Inc. animation "The Audition", shown at SIGGRAPH '90, was used to test the object space coherence algorithm. In this sequence a weight is dropped onto the see-saw, launching Eric the worm into the air. The motions of Eric, the see-saw, and the weight are derived from a dynamic simulation.

The sequence is 351 frames in length. The scene consists of 6000 polygons and 4 light sources. All frames were rendered at 640 by 480 resolution, with one ray per pixel, on a Silicon Graphics Personal Iris 4D/25 workstation.

The sequence was rendered one frame at a time with a ray tracer that utilizes an adaptive voxel subdivision technique. The number of pixels traced and the CPU time required to render the sequence are listed in Table 2 under the heading **Traditional Algorithm**. The sequence was then rerendered with the identical ray tracer, modified to use the object space coherence algorithm described in this paper. The CPU time, number of rays, and the ratios of these numbers compared to the traditional frame by frame approach are listed in Table 2 under the heading **Coherence Algorithm**.

Figure 4 shows several frames of the animation sequence illustrating only the pixels that were retraced by the coherence algorithm. Note that all the pixels of frame 0 are rendered by both the traditional and coherence algorithms.

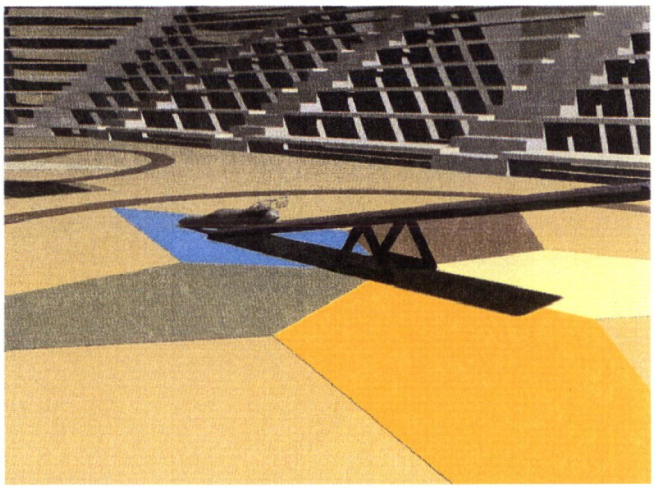
5.1 Discussion

Examining the **Entire Sequence** row in Table 2 illustrates that over the course of the animation sequence the coherence algorithm rendered only 19.35% of the rays that were traced by the traditional algorithm, and required only 26.72% of the CPU time used by the traditional algorithm, yielding a speedup of nearly a factor of four.

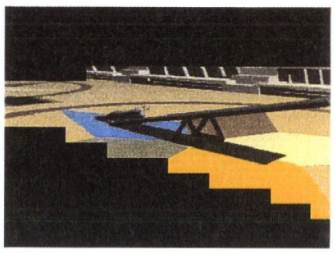
The discrepancy between the percentage of rays traced (19.35%) and percentage of CPU time (26.72%) required to render the sequence with the coherence algorithm is due to two factors. First is the overhead incurred by the coherence algorithm in building and maintaining bit-tables in each voxel and of collecting them at the beginning of each frame to determine the pixels that must be retraced. This overhead is apparent in Table 2 in the row that gives the statistics on the rendering of frame 0. The unmodified ray tracer requires 666 CPU seconds to render the frame whereas the frame coherence algorithm increases the rendering time

	Traditional Algorithm		Coherence Algorithm			
Frame #	# Rays	CPU Time	# Rays	Ratio to Traditional	CPU Time	Ratio to Traditional
0	307,200	666 sec.	307,200	1.0	780 sec.	1.171
1	307,200	669 sec.	122,400	0.3984	418 sec.	0.6248
75	307,200	684 sec.	146,400	0.4765	528 sec.	0.7719
150	307,200	649 sec.	58,800	0.1914	209 sec.	0.3220
200	307,200	634 sec.	9,600	0.0312	44 sec.	0.0694
350	307,200	655 sec.	8,400	0.0273	43 sec.	0.0656
Entire Sequence	107,827,200	63.32 hrs.	20,866,800	0.1935	16.92 hrs.	0.2672

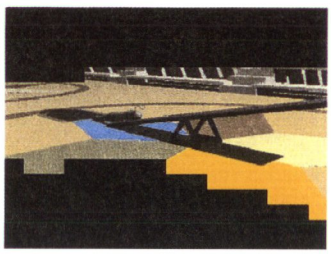
Table 2.



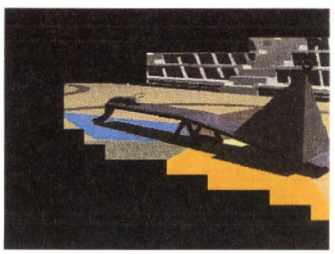
Frame 0



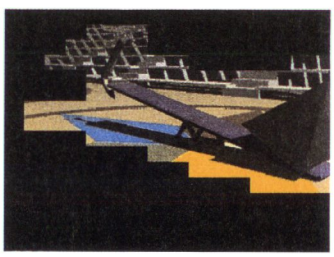
Frame 1



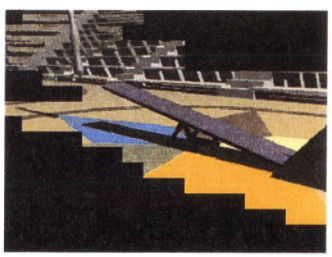
Frame 15



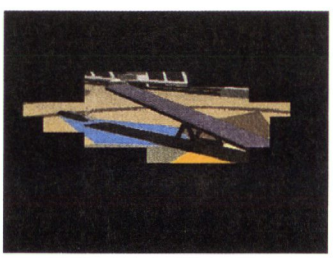
Frame 35



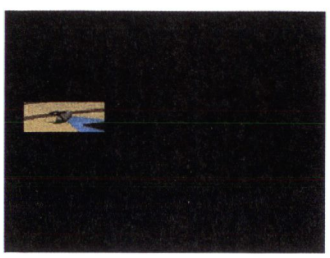
Frame 40



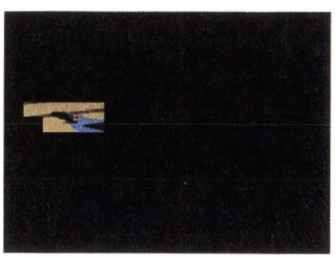
Frame 75



Frame 150



Frame 200



Frame 350

Figure 4

to 780 CPU seconds, a ratio of 1.17. This overhead is more than offset by the savings in subsequent frames.

The second source of discrepancy is due to the fact that rays are not uniform in their rendering cost. In this animation sequence, a complex object, the worm, is being retraced at each frame. The area around this complex object is more densely subdivided than the rest of the scene, requiring more traversal time per ray. The worm also has a more complex illumination model than the background model. The rays that are not retraced at each frame are typically those that intersect the background of the scene. These rays travel largely through empty voxels and intersect more simple objects, such as the tent model in this animation sequence.

6 Future Work

6.1 Inactive Voxel Collection

Adaptive spatial subdivision algorithms can reduce the amount of memory they require by taking advantage of ray coherence. When rendering an image, parts of the subdivision structure can be deleted if rays are no longer passing through them. This is common when rendering scanlines from top to bottom, as rays originating from scanlines near the bottom of the screen rarely pass through the same voxels as rays from higher scanlines. Voxels that are no longer active can be identified periodically during the rendering, and can be collected. This entails deleting the voxel's grid structure, and marking the voxel as *not subdivided*. If the assumption proves incorrect, and a ray passes through the voxel at a later time, the voxel will be resubdivided.

This idea can be extended to the temporal coherence algorithm by collecting areas of space that remain unchanged and untraversed for a number of frames. If an object inside a collected voxel changes, or a ray traverses the voxel, then it will be resubdivided.

6.2 Light Sources

When animating a light source, all rays that pass through the voxels in which it lies must be retraced. Since most ray tracers treat light sources as invisible if viewed directly, it is desirable to avoid retracing viewing rays that pass through a voxel in which a light source has moved (Figure 5). If a separate pixel index bit table for shadow rays is maintained in each voxel, then only ray trees that are affected by a moved light source are retraced.

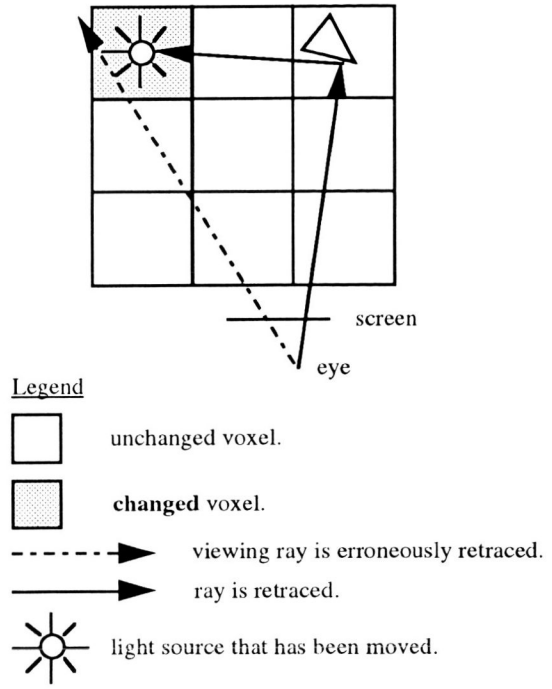


Figure 5.

6.3 Moving Camera

The algorithm presented in this paper may be extensible to sequences where the camera is moving, through the use of the reprojection technique proposed by Badt [Badt 88]. The 3D intersection points of the directly visible surfaces are projected onto the screen when the camera moves. If the camera moves only slightly, then the samples will not change in density, and a new image can be reconstructed from them. If the density of the samples changes, then the pixels will have to be retraced to avoid erroneous hidden surface results.

6.4 Backwards Ray Tracing

The object space coherence algorithm is useful, even with a moving camera, to accelerate backwards ray tracing techniques. Heckbert uses bidirectional ray tracing to calculate global illumination [Heckbert 90], and Watt uses backwards beam tracing to calculate light-water interaction [Watt 90]. Both methods could reduce the number of view independent rays required to render an animation sequence with non-moving light sources.

7 Conclusion

An algorithm has been presented for making use of object space temporal coherence when ray tracing animation sequences where the camera remains static. Only rays that pass through voxels in which objects have changed are traced at each frame. Memory use is independent of image resolution, and the algorithm is easily adapted to any spatial subdivision scheme.

8 Acknowledgements

Thanks to Gavin Miller for his help with the preparation of this manuscript and for the worm animation, George Drettakis and Michael Kass for their editorial help, Doug Turner for the ray tracer, and Steve Rubin for the circus tent. Many thanks to the entire Advanced Technology Group at Apple, without whom this work would not have been possible.

9 References

[Amanatides 84] J. Amanatides, "Ray tracing with cones," *Computer Graphics*, vol. 18, no. 3, pp. 129-135, July 1984.

[Arvo 87] J. Arvo and D. Kirk, "Fast ray tracing by ray classification", *Computer Graphics*, vol. 21, no. 4, pp. 55-64, July 1987.

[Badt 88] Sig Badt Jr., "Two algorithms for taking advantage of temporal coherence in ray tracing," *The Visual Computer*, no. 4, pp. 123-132, 1988.

[Carpenter 84] Loren Carpenter, "The A-buffer, an antialiased hidden surface method," *Computer Graphics*, vol. 18, no. 3, pp. 103-108, July 1984.

[Chapman 90] J. Chapman, T. W. Calvert, and J. Dill, "Exploiting temporal coherence in ray tracing," *Proceedings of Graphics Interface '90*, pp. 196-204, 1990.

[Chapman 91] J. Chapman, T. W. Calvert, and J. Dill, "Spatio-temporal coherence in ray tracing," *Proceedings of Graphics Interface '91*, pp. 101-108, June 1991.

[Chmilar 89] M. Chmilar and B. Wyvill, "A software architecture for integrated modelling and animation," *New Advances in Computer Graphics (Proceedings of Computer Graphics International '89)*, pp. 257-276, June 1989.

[Fujimoto 86] A. Fujimoto, "ARTS: accelerated ray tracing system," *IEEE CG&A*, vol. 6, no. 4, pp. 16-26, April 1986.

[Glassner 84] A. S. Glassner, "Space subdivision for fast ray tracing,"*IEEE CG&A*, vol. 4, no. 10, pp. 15-22, Oct. 1984.

[Glassner 88] A. Glassner, "Spacetime ray tracing for animation," *IEEE CG&A*, vol. 8, no. 2, pp. 60-70, March 1988.

[Heckbert 84] P. S. Heckbert and P. Hanrahan, "Beam tracing polygonal objects," *Computer Graphics*, vol. 18, no. 3, pp. 119-128, July 1984.

[Heckbert 90] P. S. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing," *Computer Graphics*, vol. 24, no. 4, pp. 145-154, August 1990.

[Hubschman 82] H. Hubschman and S. W. Zucker, "Frame to frame coherence and the hidden surface computation: constraints for a convex world," *ACM TOG*, vol. 1, no. 2, pp. 129-162, April 1982.

[Jevans 89] D. Jevans and B. Wyvill, "Adaptive voxel subdivision for ray tracing," *Proceedings of Graphics Interface '89*, pp. 164-172, June 1989.

[Jevans 91] D. Jevans, *Adaptive Voxel Subdivision for Ray Tracing*, Master's Thesis, University of Calgary, 1991.

[Joy 86] K. I. Joy and M. N. Bhetanabhotla, "Ray tracing parametric surface patches utilizing numerical techniques and ray coherence," *Computer Graphics*, vol. 20, no. 4, pp. 279-285, Aug. 1986.

[Kay 86] T. L. Kay and J. T. Kajiya, "Ray tracing complex surfaces," *Computer Graphics*, vol. 20, no. 4, pp. 269-278, Aug. 1986.

[Murakami 90] K. Murakami and K. Hirota, "Incremental Ray Tracing," *Eurographics Workshop on Photosimulation, Realism, and Physics in Computer Graphics*, pp. 15-29, June 1990.

[Rubin 80] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," *Computer Graphics*, vol. 14, no. 3, pp. 110-116, July 1980.

[Séquin 89] C. H. Séquin and E. K. Smyr, "Parameterized ray tracing," *Computer Graphics*, vol. 23, no. 3, pp. 307-314, July 1989.

[Shinya 87] M. Shinya, T. Takahashi, and S. Naito, "Principles and applications of pencil tracing," *Computer Graphics*, vol. 21, no. 4, pp. 45-54, July 1987.

[Watt 90] M. Watt, "Light-water interaction using backward beam tracing," *Computer Graphics*, vol. 24, no. 4, pp. 377-385, August 1990.

[Whitted 80] T. Whitted, "An improved illumination model for shaded display," *CACM*, vol. 23, no. 6, pp. 343-349, June 1980.