Exploiting Spatio-Temporal Coherence in Ray-Traced Animation Frames

by

Stephen J. Adelson and Larry F. Hodges

GIT-GVU-93-01
January 1993

Graphics, Visualization & Usability Center

Georgia Institute of Technology
Atlanta GA 30332-0280

Exploiting Spatio-Temporal Coherence in Ray-Traced Animation Frames

Stephen J. Adelson and Larry F. Hodges
Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

The majority of computer graphics images are generated as part of a sequence of animation frames. The usual approach for producing these images, however, is to render each frame in an animation as if it were a single, isolated image. We describe a technique that exploits spatial-temporal coherence between frames to speed up the generation of a ray-traced animation sequence. The concept is that the information gained when ray-tracing a particular frame in a sequence may be used to speed up the ray-tracing of other nearby frames. Based on a stereoscopic generation technique that generates the second half of a stereo pair for as little as 5% of the effort required to fully ray-trace the first half, this approach promises significant savings when ray-tracing animations with certain characteristics. Our algorithm is tested on an animation with many reflective surfaces, yet it still attains an overall 60% savings over full ray-tracing in terms of the number of rays solved.

Key Words: Display algorithms, Ray-traced animation, Spatial Coherence, Temporal Coherence.

Introduction

Ray-tracing is an attractive method of generating images because of its simplicity and its capability of producing many realistic optical effects such as refraction, reflection and shadowing. Ray-tracing's major drawback, however, is its large computational cost, due mainly to calculating ray-object intersections [Wh80]. This computational cost for complex images can run from several minutes to several hours on current workstations, depending on the machine used and the efficiency of the algorithm implementation. General speedup techniques for ray-tracing usually concentrate on improving ray-object intersection algorithms for some specific category of objects, or on eliminating unnecessary ray-object intersection tests by using bounding volumes or some type of hierarchical structuring of the data. For ray-tracing of animation sequences, a further speedup approach is to take advantage of spatio-temporal coherence from one frame to the next. The idea is that each frame is usually very similar to the frames that immediately precede

and succeed it. Therefore, the information gained when ray-tracing a particular frame in a sequence may be used to speed up the ray-tracing of other nearby frames.

In this paper we present an algorithm that exploits spatio-tempral coherence between frames to significantly decrease the rendering time of ray-traced animations produced by changing the camera's position with respect to a static scene.

Previous Work

Many attempts have been made to speed up the ray-tracing of individual frames, but most fall into only two categories. In the first, the algorithm seeks to decrease rendering time by reducing the number of intersection tests. Examples of this category include Rubin's nested bounding volumes for testing against increasing levels of detail [RW80], and space subdivision methods of Glassner's Octrees [Gl84] and Fujimoto's ARTS system [Fu86]. In the second category, time is saved by providing more efficient intersection tests. Often, this involves better bounding volumes, such as the bounding volume work by Bouville [Bo85] and Kay's work with intersections with convex hulls [KK86].

Several algorithms also attempt to use frame-to-frame coherence to speed up animation. Most of these have used object space coherence for their speed-up method. Hubschman's object visibility tests process static convex objects based on when they will become visible [HZ82]. Glassner created 4-D bounding volumes extending through both time and space to reduce both bounding volume creation and duplicated intersection tests [Gl88]. Chapman designed a method which finds the intersection of "continuous intersections" between rays and polygons through time, so that a pixel need only be retraced when the current continuous intersection ends [CC91].

Sequin stored the ray tree at each pixel so that image attributes could be changed in the image without having to cast any additional rays [SS89]. Unfortunately, the method does not work when objects move, since the algorithm cannot determine visible surface movement. This method was extended for animation by Murakami [MH90], by subdividing space into voxels and storing a list of voxels for every ray in each ray tree. Moving objects are noted in the voxels, and all rays which pass through the changed voxels are retraced. However, memory requirements for storing the lists are very large and the computational load is heavy because of the many tree traversals. Further, ray trees are completely lost if only the first ray is moved.

Jevans stored in each voxel a tag to the original generating pixel which spawned the ray passing through the voxel [Je92]. Changed voxels caused the appropriate pixels to be re-traced. Jevans' approach is limited in that it only works for static cameras and a change in a ray of any level causes the entire pixel to be retraced. Also, his voxel tags represent large blocks of pixels rather than individual pixels because of the memory constraints.

Image Space Coherence and Related Work in Stereoscopic Ray-Tracing

Badt in 1988 proposed a method of image space coherence by reusing the pixels from one frame of a diffuse-object animation to determine many of the pixels in the next frame [Ba88]. His method led to gaps and unintended pixels in the new frame. These problems were repaired to a large degree by clean-up routines, but not completely nor consistently. Ezell, realizing that a stereoscopic pair is the equivalent to two animation frames in which the objects remain motionless and the viewpoint moves some small distance, implemented Badt's method for the generation of stereoscopic ray-traced images [EH90]. However, this implementation suffered from the same limitations and image problems as Badt had in his animation frames.

We have previously recognized the causes of the image problems in Badt's method [AH92a]. Using the geometry of stereoscopic viewing, we eliminated the image problems as well as the tests needed by Badt and Ezell when attempting to fix their images. Further, we developed the method to allow full ray-tracing of stereoscopic images. We demonstrated that even though the higher levels of ray-tracing must be done twice (as they are eye-point dependent), the majority of the savings will remain [AH92b]. Given what we have learned from the specific case of stereo pairs, we return to animation frames. Our new method produces inferred ray-traced images of any scene which can be ray-traced with a guarantee that no pixel will exist out of place by more than a 1/2 pixel when rendering at a single ray per pixel, a visual error which can be attributed to undersampling. These are not approximated frames created from weighted averages of other frames (e.g. [FL90]), nor are they frames patched together from near-frame pixels values (e.g. [CL90]). Our algorithm guarantees that a color seen in a pixel will be returned by a ray passing through that pixel, but not necessarily though the center of the pixel. Further, the algorithm can be shown to increase in efficiency when more objects are added to the scene. While we produce images assuming a static object set, we believe that this method is extendible to animation with moving objects and/or light sources.

A Special Case of Animation: Stereoscopic Ray-Tracing

In stereoscopic rendering, there must be two centers of projection so that a different perspective view is produced for each eye. Figure 1 illustrates that the two viewing positions are separated parallel to the X axis by a distance **e** and that they are both a distance **d** from the projection plane [Ho92]. Stereoscopic ray-tracing is really a special case of generalized animation, with the small exception that the viewing window remains fixed in three-space while in animation it is set at a certain distance and orientation to the viewpoint. We will therefore first look at certain issues in stereo before extending them to the full animation case. The following is a summary of work which can be found in [AH92b].

Given the particulars of the stereoscopic viewing geometry, a ray intersection at (x_p, y_p, z_p) projecting to position (X, Y) in the left-eye view will appear on the same scan-line or *reproject* to the position ($X + \mathbf{e} * z_p / [\mathbf{d}+z_p]$, Y)

in the right eye view. Reprojected pixels have the same diffuse color, shadowing, three-dimensional intersection, normal, and texture as the original pixel. While reflected components, refractive components, and highlights will differ, first level ray-tracing savings amount to 50-95% in most images. In the same way, we wish to determine where a pixel from one frame of an animation will appear in the next frame. Unfortunately, we cannot merely reproject and ray-trace the remaining positions.

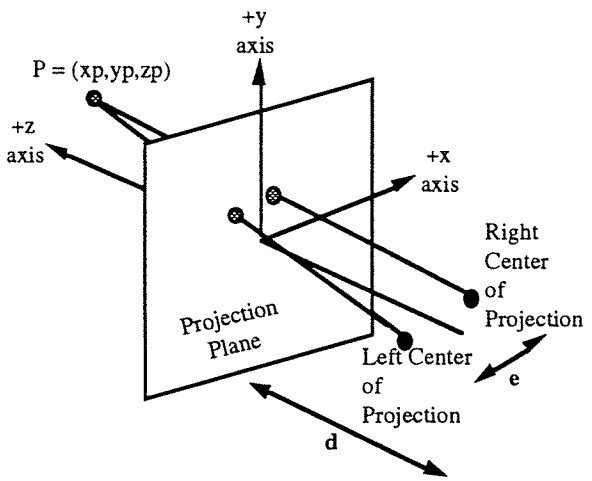

Figure 1. The Stereoscopic Viewing Geometry

When pixels reproject in the stereoscopic geometry, one of four things can happen in the second view. First, a pixel position may have exactly one pixel value reproject into it. These are called *good pixels* and we may use their color and position values directly. Positions which receive no reprojections are called *missed pixels* and must be fully ray-traced. *Overlapped pixels* are those in which two or more previous pixels values reproject to the same pixel position. We have shown that by processing pixels consecutively in either direction on the scan-line we can always extract the closest reprojection. Finally, pixels which were consecutive on the original scan-line may reproject with a gap between them, allowing other pixels to be viewed through the gap even though they should be obscured. These *bad pixels* can only be eliminated by clearing the values of all positions between two pixels which reproject with a positive gap between them (i.e. if **newx** is the reprojection function, and pixels x and x+1 reproject such that **newx**(x+1) - **newx**(x) > 1, we clear positions **newx**(x) + 1 through **newx**(x+1) - 1).

Generalizing Reprojection

As in stereoscopic ray-tracing, it is possible to make reprojection equations for a general movement in three-space. Unfortunately, these movements are far more computationally intensive and they do not take into account rotations, which must be provided separately. Instead, it is more efficient to preserve the three-dimensional intersections between rays and objects and project them to the new viewing position.

Characteristics of the Animation

Our technique will create efficient frames of any view which can be ray-traced. We are not limited, for example, to diffuse polygons. While the savings of our technique will increase with the complexity of the image and a preponderance of diffuse objects, we can still achieve a large degree of savings with reflective and refractive objects. However, to take advantage of the spatio-temporal coherence in a similar manner to stereoscopy, our animation needs to have three primary characteristics:

(1) The ray-tracing method is point-sample oriented. "Pure" ray-tracing involves following lines through the scene, and these lines are considered infinitesimally thin. Since we are projecting pixels based upon their three-dimensional intersection, we must have these point samples to allow reprojection. Methods such as beam tracing [HH84] or cone tracing [Am84] are therefore forbidden in the first level of ray-tracing. Anti-aliasing can still be accomplished by a point-source oriented method such as adaptive super-sampling, and other methods of ray-tracing may be used for the higher-order rays involved with reflection and refraction.

(2) The light source(s) do not change position, intensity, or color. The limitation allows us to reuse the diffuse color of an object from frame to frame as well as to reuse the shadow ray. Lifting the limitation would require recasting the shadow rays and recalculating the diffuse color. This by no means eliminates all savings, as we shall show (in estimation) later.

(3) Objects are static. There are two problems with allowing objects to move. First, their shadows will also move, requiring us to recast the shadow ray. Second, our method operates under the assumption that we can see all movement from the viewpoint. New or obscured objects may appear, but only because of a movement of the visible scene. If an object which was hidden previously became visible because of its own movement, our technique would have no way of recognizing the new object. We illustrate this in figure 2.

The initial frame of the animation must be completely ray-traced. For each pixel in this first frame, we save the three-dimensional intersection point, the normal vector, the diffuse color, a bit for each light source representing whether the intersection is in shadow, and an id-tag to the object on which the intersection occurs. We are now ready to infer new frames.

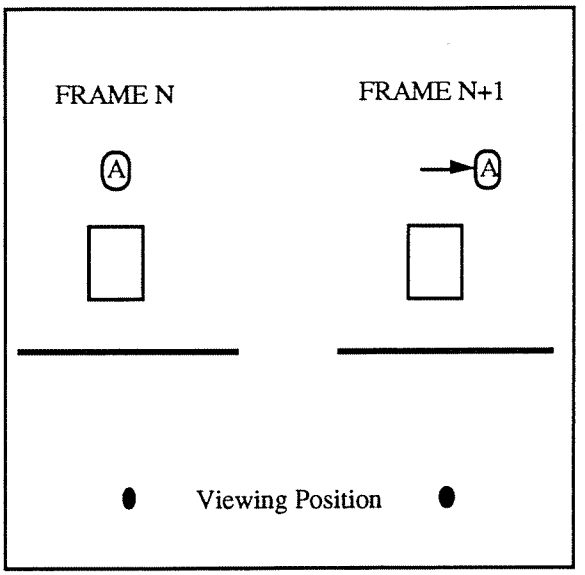

Figure 2. The algorithm can only process known pixels from frame N when creating frame N+1. If an object moves on its own, those pixels are not recognized.

The Buzzbuffer and Overlapped Pixels in Inferred Animation Frames

The camera in the new frame is given to be **d** units from the projection plane. We create a four by four matrix which translates the new viewing position to (0, 0, -**d**) and rotates the axes to match the orthogonal world axes. Each previous pixel projection is transformed by the matrix to a three-dimensional point (x_t, y_t, z_t), and projected to the new viewing position by the equations

$$X = x_t * \mathbf{d} / (\mathbf{d} + z_t),$$
$$Y = y_t * \mathbf{d} / (\mathbf{d} + z_t).$$

We would, of course, retain the $\mathbf{d} / (\mathbf{d} + z_t)$ term from the X calculation to save one addition and multiplication in the Y calculation. Unlike the translation equations in stereo, we have no simple way of knowing how far a pixel has moved on the projection plane, and unless we are performing motion blur on the image we do not need this information. However, close objects will appear to move more from frame to frame than distant objects when the camera movement is a translation, as the reader will recognize from his own experience as a moving viewpoint.

As in stereoscopic ray-tracing, we have four possibilities for pixels. The good and missed pixels are, respectively, used directly and fully ray-traced. The solutions to the overlapped and bad pixel problems were solved in stereo by taking advantage of the stereoscopic geometry and the pixel processing order. In animation, we no longer have these conveniences.

Since pixels may move in any direction because of a camera movement or rotation, we require a full-size (window sized) data structure to hold the new pixels. We call this data structure a Buzzbuffer, after the first image we rendered using this technique, an image of the Georgia Tech mascot.

The overlapped pixel problem, in which more than one pixel value from the previous frame would be viewed through the same pixel in the current frame, is solved by the Buzzbuffer. Like a Z buffer, the Buzzbuffer uses the depth of the transformed pixels to select the closest intersection projected to a pixel. In addition, the Buzzbuffer also holds the intersection point, normal vector, color information, shadow bits, and id tag to the intersected object (id_tag). This data will be used in calculating highlights, reflection, and refraction, as well as being written to disk for use in creating the next frame.

Solving the Generalized Bad Pixel Problem

Although we have access to all the pixels from the previous frame, we do not know what intersections may have been rejected because they were not the closest hit to the eyepoint. It is possible when the pixels are projected to the new viewpoint that some of these "lost" intersections should be seen; instead, portions of the image which should now be obstructed are visible (Figure 3).

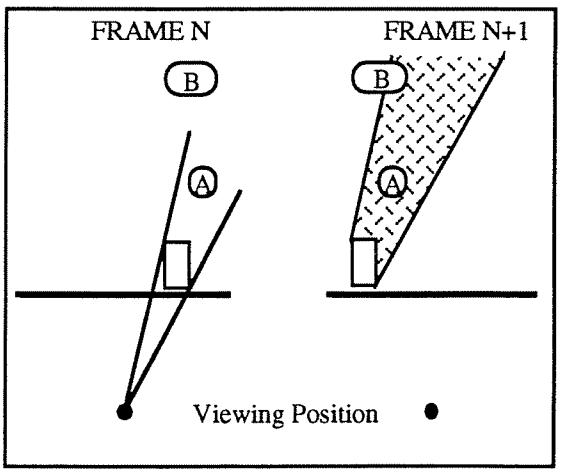

Figure 3. Anything hidden in frame N, such as object A, could be visible in frame N+1 and obscure known objects, like object B. We need to recognize the possibility of unknown objects in the shaded area of frame N+1.

The solution to this problem requires accounting for these intersections. In stereoscopic ray-tracing, the pixel positions where the lost intersections might appear were easily found because positive direction gaps would open in the image. In the general animation movement, these gaps might occur in any direction, so we are required to perform more calculations.

Anything along a line from the previous viewing position through the intersection (which lies behind the intersection itself) could potentially block more distant but already rendered pixels. We may consider the far endpoint of this line to be at some distance which we define or, if all the data is not relatively close to the viewing position, infinity. In either case we can calculate the projected position of this endpoint, and represent a line on the Buzzbuffer of possibly obscured pixels. We approximate this line by rounding both the intersection projection and endpoint projection to the nearest pixel and, using a Bresingham-type line stepping algorithm, test the pixels' Z values against the intersection of the three-dimensional line and planes through the pixel edges. Note that even if a pixel from a previous frame projects outside the new frame, it is still possible that it was obscuring an object which remains in the image. The algorithm is as follows:

Initialize Buzzbuffer by setting all z values = HUGE and id_tag to -2 (all our objects have positive tags)
Project old pixel to Buzzbuffer at (begx, begy), with depth **miny**.
If (begx, begy) is within the range of the Buzzbuffer and is the closest hit
    Buzzbuffer(begx, begy).z = **miny**
    Buzzbuffer(begx, begy).data = pixel information (i.e. intersection, normal, color, shadow, id_tag).
Calculate to where the endpoint of a line **L** from the old viewing position through the three-dimensional intersection would project from the new viewing position. Call this point (endx, endy).
Let **C** be the line defined by the points (begx, begy) and (endx, endy).
Extract the portion of **C** which lies within the Buzzbuffer. Call this **C1**
For each point (midx, midy) of **C1** excluding (begx, begy)
    If Buzzbuffer(midx, midy).z < HUGE
        Calculate planes through the new viewpoint and the sides of pixel (midx, midy).
        Calculate depth of the intersection between these planes and the line **L**.
        Let **int_z** be the maximum z value of all intersections
        If Buzzbuffer(midx, midy).z > **int_z**
            Buzzbuffer(midx, midy).z = **int_z**
            Buzzbuffer(midx, midy).id_tag = -2.

The last two steps make sure that nothing will project into the pixel unless it has a Z value of less than **int_z**, and if nothing reprojects to this position the -2 tag will allow complete ray-tracing of the pixel.

Note that if the rotation of the viewpoint is such that the transformed viewing volume planes still have positive z components, we may change the step
    If Buzzbuffer(midx, midy).z < HUGE

to

If (Buzzbuffer(midx, midy).z < HUGE) and
(Buzzbuffer(midx, midy).z > **miny**).

Also, depending on which pixels are in **C1**, it is possible to reuse some of the ray-plane intersections.

The total time for reading in new data, transforming it, projecting it, and running this algorithm to clean up is generally minimal. In our test images it accounts for 13% of the rendering time in very simple scenes (background polygon only), and as little as 0.7% in one complex test scene (1.5 million rays).

The "Creeping Pixel" Problem.

Suppose we are rendering a square polygon against a very distant background, as in figure 4. In the next frame, we approach the polygon such that the pixels reproject as shown in the figure. The distant pixels of the background do not move. The four pixels of our polygon project outwards, leaving a gap of four pixels which are ray-traced. However, there are now gaps along the polygon edge which are not recognized by the algorithm. If uncorrected, this process continues, and incorrect pixels can migrate further into the polygon, causing creeping pixels.

Without limiting pixel processing order, we can recognize these gaps after all reprojection is finished. We search both horizontally and vertically for gaps within objects, checking all pixels inside the gap to verify that those pixels have z values less than the maximum of the two endpoints of the gap. The clean up routine is as follows:

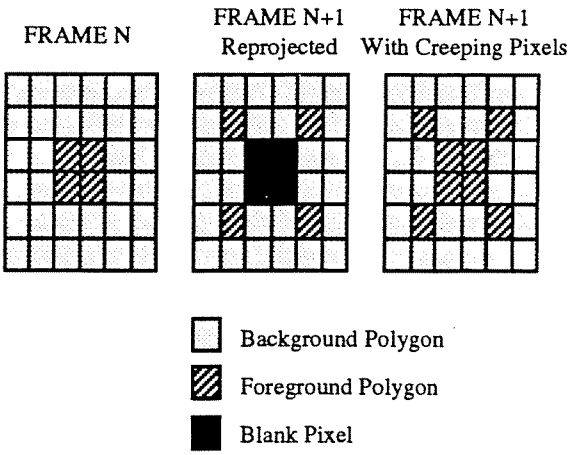

Figure 4. Creeping pixels occur when consecutive edge pixels do not reproject consecutively.

/* Assumes an N by M image and objects have positive id_tags */
/* Vertical Cleanup */
For each column **j**

Find the first positive id_tag such that the same id_tag appears later after a gap of other id_tag(s). If any id_tags = -2 before both endpoints of this gap is found, ray-cast the position.

Set **test_z** to the maximum z value of the two endpoint.

For each position **k** in the gap

If Buzzbuffer(**j**, **k**) > **test_z**

Ray-cast position (**j**, **k**)

Set Buzzbuffer(**j**, **k**) with intersection and object information

Continue cleanup from the first position in the just-checked gap.

/* Horizontal Cleanup, Ray-Tracing, and Enhancements (i.e. highlights, reflection, refraction, etc.) */

For each row **i**

Find the first positive id_tag such that the same id_tag appears later after a gap of other id_tag(s).

Enhance and write to disk any all pixels up to and including the first gap pixel.

Set **test_z** to the maximum z value of the two endpoint.

For each position **k** in the gap

If Buzzbuffer(**k**, **i**) > **test_z**

Ray-cast position (**k**, **i**)

Set Buzzbuffer(**k**, **i**) with intersection and object information

Continue cleanup from the first position in the just-checked gap.

This algorithm is $O(N^2M^2)$ in terms of comparisons, but since gaps are almost always small the ray-casting will occur only occasionally and the actual expected time of the algorithm is minimal.

The New Object Problem

In most animation, the projection plane moves along with the viewpoint, usually at a constant distance and relative orientation. The possibility arises that objects may enter the scene because of camera movement. If pixels on the edges do not move enough, portions of this new data will not be rendered. For example, suppose we had a frame containing an infinitely distant vertical plane. If we had a camera movement to the side, these pixels would not change their position, so no gaps would appear to allow newly viewed data to be intersected.

If we study the three-dimensional viewing volume from a particular position, it appears to be a pyramid with its apex at the viewing position. If the next frame of the animation is not identical, the viewing volume of the new viewing position will differ as well. It is in this area of difference that data may exist, and in order to render new objects we must be sure not to project pixels into the contested area.

For every frame we save the coefficients defining the four planes of the viewing volume. If we are generating a new frame in which data enters (or we suspect we are at such a point), we will specify a distance **k** in Z behind which (relative to the viewpoint) the new data will appear. We calculate a plane **P** parallel to the projection plane (the x-y plane) and passing through the transformed point, a simple matter since we require the orthogonal axes as camera information.

We first calculate which portion of **P** will be within the previous viewing volume as defined by the four planes previously saved. This area of **P** is clipped to a rectangular polygon corresponding to the current viewing volume at a depth **k**. We are left with a polygon of no more than eight vertices. This polygon is scan converted, with all interior positions and edges given a Buzzbuffer Z value of HUGE, external points given a -1 to represent possible new data.

Generally, we would do this scan-conversion as a pre-processing step. However, there may be cases in which the value of **k** is uncertain. It is possible to scan-convert after the projection of

the previous pixels (but before the clean-up for the creeping pixel problem), using (for example) the minimum Z of the projected points for the value of **k**. In this case, all exterior points of the scan-conversion would be cleared of any projected pixels.

We also note that rotations of the camera only will automatically move pixels, so this solution need only concern us when data enters a frame because of a movement of the camera.

Performance Tests

To test the algorithm, we generated a short (600 frame) animation consisting of 755 polygons and 41 quadric surfaces with a single light source. Of these surfaces, 578 of the polygons and all 41 quadrics were partially reflective; additionally, four of the quadrics were also partially refractive. The animation was rendered with one light source at 640 by 480 resolution and one ray per pixel. Camera movement included rotations about all three axes. The image displays a large degree of interreflection, from which we attain no reduction in rendering time. We therefore call the savings generated here an informal lower bound.

There are two ways of measuring performance on this algorithm. The first is in the number of total rays saved. This gives an indication of how much time we save, but it does not take into account the overhead of the reprojection and clean-up routines. If instead we take as our metric the actual time required to generate frames, we are also dealing with the implementation method as well as the load on the rendering machine at any given moment. We will therefore indicate both measures of savings, and claim that the actual savings is somewhere in the range between the two.

We fully ray-traced 85 frames of the animation to get an approximate idea of the number of rays and time involved with completely ray-tracing the animation. We have placed this data in tables 1 and 2, along with the information gathered from inferring the animation. Frame 0 (seen in figure 5) is completely ray-traced to serve as our base data frame. It would have been more efficient to fully ray-trace frame 300, which would allow us to infer frames towards both ends of the animation simultaneously. However, frame 0 contains nothing except the ground and sky planes, meaning that the "meat" of the animation must be inferred by our algorithm. For demonstration purposes, we feel that this method best illustrates the abilities of the method.

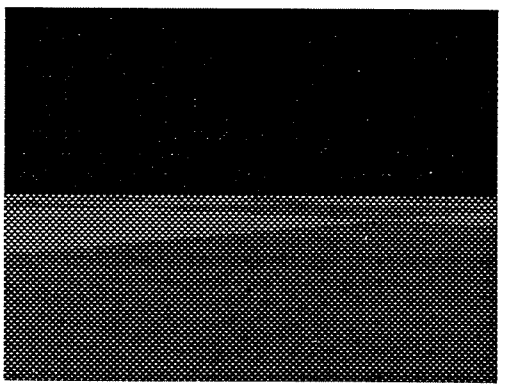

Figure 5. Frame 0 of the animation. Two infinite planes.

For analysis, we separate the rest of the animation into two sections, frames 1-24 and 25-599. The first 25 frames' data as seen in

table 1 represents ray-tracing against two infinite planes, both of which have no reflective or refractive components. Only the ground plane has highlights and shadowed pixels. Although the algorithm saved many of the rays - almost 88% - the overhead of reprojection and clean up routines were such that we actually had a net loss in rendering time of 15%. We are not very concerned with this result, however, since just about any ray-tracing speed-up technique would perform poorly on such trivial frames.

The data for the last 574 frames is in table 2. These frames contain buildings with reflective surfaces and one Yellowjacket, also reflective. Since these reflective rays are eye-point dependent, they must all be rendered at each frame. Yet despite an average of over 196,000 secondary rays per frame, we save over 60% of the total number of rays cast. Just as before, overhead time eats into the actual rendering time savings. Nonetheless, even ignoring machine load effects on rendering time, we saved over 53% over a traditional ray-tracer in rendering time.

Many algorithms which purport to speed animation generation time use diffuse images for testing. We have refrained from doing so to show the power of our technique. It is a simple matter to count rays, however, and we have found that were this a strictly diffuse animation, we would save almost 80% of the ray casting by using our method. Even if we assume the same amount of time for overhead, a diffuse image would save over 70% in rendering time.

Image Quality

No quantity of ray savings is significant if the animation frames produced are not of quality.. In figures 6-8, we have inferred frame 150, the same frame fully ray-traced, and the difference image between the two with enhanced pixel values for easier viewing. We have done the same in figures 9-11 for frame 450.

Visually, there is little difference between the inferred frames and the fully ray-traced frames. The difference images reveal that the disparity occurs on edge boundaries. This exists because the boundaries in the inferred frames have reprojected from other frames and, while they can be seen if we send a ray somewhere through that pixel, do not quite match up with the rays sent through the center of the pixels in the fully ray-traced view.

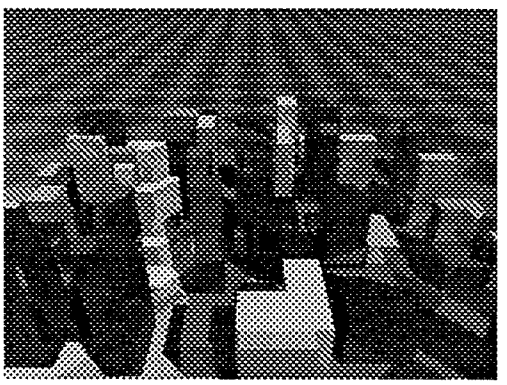

Figure 6. Inferred frame 150.

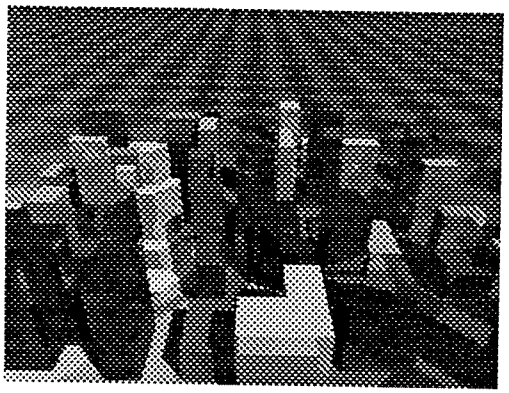

Figure 7. Fully ray-traced frame 150

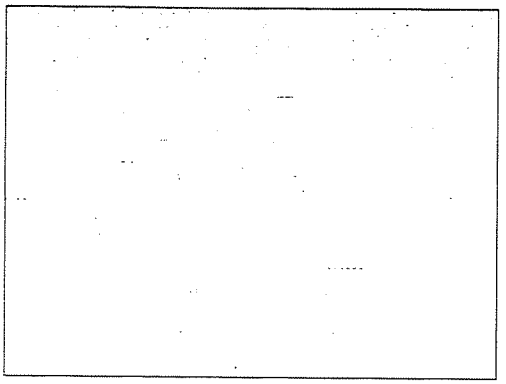

Figure 8. Difference image of figures 6 and 7. Pixel values in the range of 0-100 have been mapped to 0-255.

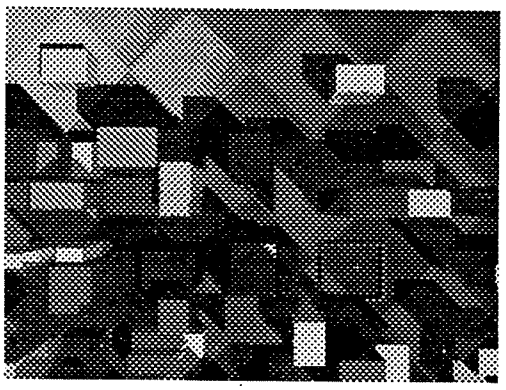

Figure 9. Inferred frame 450.

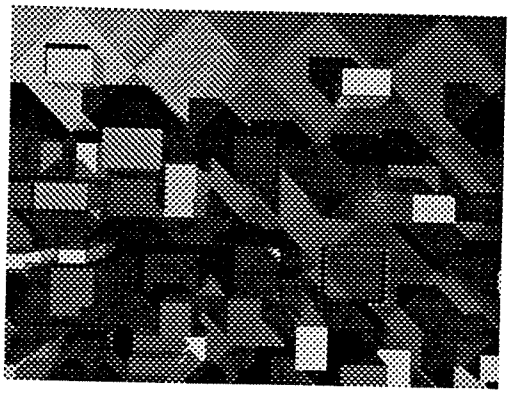

Figure 10. Fully ray-traced frame 450.

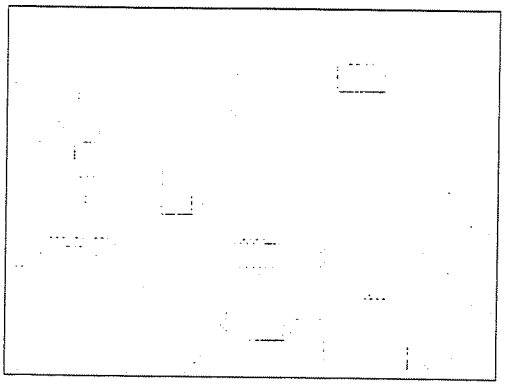

Figure 11. Difference image of figures 9 and 10. Again, pixel values have been enhanced.

Finally, we present figure 12. Inferred frame 599, inferred through 598 steps from the initial frame. We leave it to the reader to judge the quality.

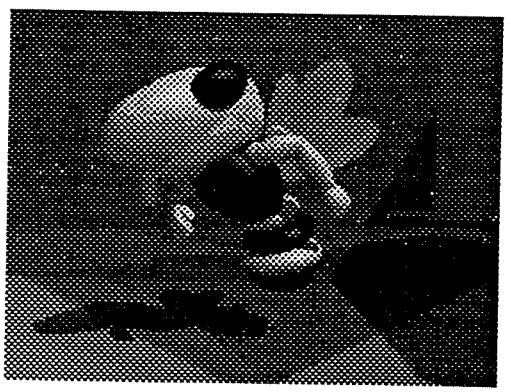

Figure 12. Inferred frame 599.

Storage Details

Our inter-frame information file requires over 17 megabytes of storage at one ray per pixel and 640 by 480 resolution. This includes at each pixel three double precision numbers for the three-dimensional intersection and normal vector, 3 bytes for the diffuse color, one byte (8 bits) to represent up to 8 shadow rays, and one integer for an object tag. This is 56 bytes per pixel, plus an additional 200 bytes for the frame representing the four planes of the previous viewing volume, the old viewing position, and the size of the image in x and y.

If storage is at a premium, the first order of business would be to reduce the double precision numbers to standard floating points. This will put the storage requirements at less than 10 megabytes. If needed, we could recalculate the diffuse color every time, saving almost another megabyte of storage.

This file need only be saved, however, until the next frame is rendered. Or to be more precise, until the information contained in the file has been reprojected.

Future Work

Anti-Aliasing

It is interesting to note that aliasing from distant objects will be less objectionable using our rendering technique than a traditional ray-tracer would be. Remember that distant objects will appear to move more slowly than closer objects. This means that a distant aliasing problem, say a checkerboard ground plane, will tend to move slowly as one piece and not change its aliasing from frame to frame. The closer objects with aliasing are often not noticed because, while their aliasing will differ between frames, they are changing position too quickly for the aliasing to register.

But let us suppose that we desire anti-aliasing in our animation frames. Any point-related anti-aliasing technique will suffice, and we have developed a limited adaptive super-sampling method which will function for our renderer. We subdivide the Buzzbuffer by a factor of $2^{L-1} + 1$, where **L** is the maximum number levels we wish our frames to super-sample. A subpixel of the Buzzbuffer is shown in figure 13. After anti-aliasing in a given frame, we save all traced positions, and reproject all of them in the next frame. The creeping pixel cleanup routine must be changed so that only positions which have had reprojections are considered. No ray-tracing of intermediate pixels is done until sampling time.

Just as we guarantee reprojections within 1/2 pixel at one ray per pixel, this method will always reproject within 1/2 of the subpixel size involved. For example, at three levels of potential super-sampling, reprojections will fall within 1/8 of a pixel of their actual position. This will significantly reduce the error apparent in the difference images above.

1	3	2	3	1
3	3	3	3	3
2	3	2	3	2
3	3	3	3	3
1	3	2	3	1

Figure 13. Pixel-sized data structure to allow adaptive super-sampling.
This 5 by 5 structure would allow up to three levels of sampling. The numbers in the sub-pixels represent the level of sampling at which the sub-pixel would be ray-traced.

Despite the improved image quality, we will often want to limit sampling to as few levels as necessary. The more levels of super-sampling we allow, the more storage we will require for the frame information, the larger the Buzzbuffer will have to be, and the greater the chance that reprojecting pixels will be wasted, as in figure 14.

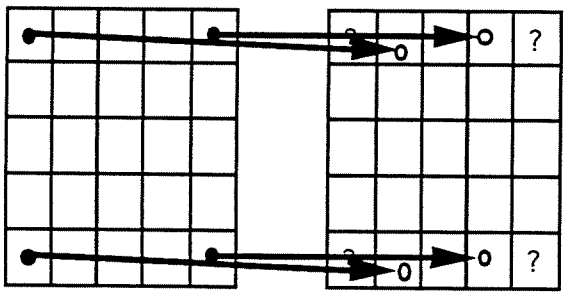

Figure 14. Reprojection doesn't always mean savings!
Here, four sub-pixels reproject into the inferred frame such that savings occur only if we go to the third level of sampling in the pixel.

We provided this technique in our studies of stereoscopic ray-tracing [AH92b]. From those results, we estimate that the savings will decrease by about 10% because of wasted reprojections and additional overhead of larger data structures.

Multiple Light Sources

We rendered our images with one light source. If multiple light sources exist, we will have a bit representing the shadows of each. However, we have previously shown that if the light sources are static, then the savings are unchanged [AH92b]. The pixels which must be retraced require more computations with multiple light sources, and therefore the percentage of rays and color combination calculations saved remains the same.

Variable Light Sources

If light sources are static but change intensity or color, we will need to recalculate the diffuse color for pixels which are not in shadow from the changing light source. Since we already save the object tag, intersection, and normal, there is only slightly more work involved with recalculating the color of the object.

Moving Objects

As we have stated previously, our method only works properly when we are assured that no objects change their position. If only light sources move, we may use the algorithm as is but recast the appropriate shadow rays. For our animation, the savings in frames 25-599 would still be over 30% if we were casting such rays, and 41% if we ignore non-diffuse rays. Multiple light sources will continue to degrade

the saving as the number of sources grows, but even with five moving light sources our animation would achieve 12.5% savings overall.

It is often the case that we want objects to move independently. We have not implemented such an algorithm, but we suggest the following, based upon Jevans [Je92]. Object space is divided into voxels by some space partitioning method. For a new frame, we calculate which voxels will have changed because of an object movement. These voxels are set with a single bit. After the reprojection of previous frame pixels and before the creeping pixel cleanup, we test every reprojected pixel in the Buzzbuffer. For each pixel, if a line between the intersection and the new viewing position passes through a set voxel, the position is cleared to be fully ray-traced later. If the pixel survives this test, we check a line between the intersection and each light source. If a set voxel is intersected, we may keep the position, but we need to recalculate the diffuse color and recast that shadow ray.

More efficient, but at the cost of higher storage, would be to save the parametric **t** for each light source rather than a bit to represent the shadow on the intersection. In that case, the shadow line need only be checked as far as the voxel at distance t. If the light source itself moved, we would need to recast the ray to that light source for all pixels, although we may keep the diffuse color if we find that a shadowed position remained shadowed.

While it is outside the scope of this paper, it would seem that the method outlined above might be an improvement over Jevans' algorithm for animations with few or no higher-order rays.

Parallelization of Frame Generation

The creation of animation frames using this technique is a linear process; we have to finish creating the previous frame before we can begin the next frame. It may be desirous in long animations to render frames on different machines. Obviously, camera cuts are natural points of breaking the animation. Otherwise, we suggest that if an appropriate anti-aliasing technique is used that every kth frame be fully ray-traced (k an arbitrarily large odd integer, but based on the number of machines available and their relative speed), and infer the $\lfloor k/2 \rfloor$ frames on either side of the fully ray-traced image.

If we are not using anti-aliasing, the aliasing artifacts will not match at the join points, so the matching points should be placed in sections of fast camera movement to reduce their notability.

Conclusions

Decreasing animation frame generation time by exploiting spatio-temporal coherence with a moving camera position has been largely ignored because of the general perception that all information is lost when the camera moves. We have shown not only that we can save data from the first level of ray-tracing, but that we can also save the shadow ray information. We are not aware of any other system that can create inferred exact frames and save the shadow rays without saving the entire ray tree at each pixel. This represents an important advantage when storage facilities are not large. While we lose

some savings to the eye-point dependent reflection and refraction rays, it should be noted that most of the world is not reflective and refractive but predominately diffuse.

In our animation performance test, the algorithm saved approximately 60% of the rays, and 53% in rendering time. A similar animation with diffuse-only objects would have resulted approximately in an 80% reduction in rays cast. Our animation consisted of ellipsoids and quadric surfaces, but the technique is not limited to these objects. Any scene which can be ray-traced in a point-sampling manner - quadrics, fractals, implicit surfaces - is usable material for the method. This algorithm has potential for adding moving objects, moving light-sources, and anti-aliasing. As such, it represents an important step towards using spatio-temporal coherence as an aid in creating animation sequences.

References

[AH92a] Adelson SJ and Hodges LF Visible Surface Ray-Tracing of Stereoscopic Images. Proc SE ACM: 148-157

[AH92b] Adelson SJ and Hodges LF Stereoscopic Ray-Tracing. Georgia Tech TR GIT-GVU-92-17 To appear in The Visual Computer.

[Am84] Amanatides J Ray Tracing with Cones. Comput Graph 18 (3): 129-135.

[Ba88] Badt, Jr. S Two algorithms taking advantage of temporal coherence in ray tracing. Vis Comput 4 (3): 123-132

[Bo85] Bouville C Bounding Ellipsoids for Ray-Fractal Intersection. Comp Graph 19 (3): 45-52

[CC90] Chapman J, Calvert TW and Dill J Exploiting Temporal Coherence in Ray Tracing. Proc Graph Interface 90: 196-204

[CC91] Chapman J, Calvert TW and Dill J Spatio-Temporal Coherence in Ray Tracing. Proc Graph Interface 91: 101-108

[EH90] Ezell JD and Hodges LF Some preliminary results on using spatial locality to speed up ray tracing of stereoscopic images. SPIE Proc 1256: 298-306

[FL90] Foley TA, Lane DA and Nielson GM Towards Animating Ray-Traced Volume Visualization. Jour Vis Comput Anim 1 (1): 2-8

[Fu86] Fujimoto A ARTS: Accelerated Ray Tracing System. IEEE Comput Graph Appl 6 (4): 16-26

[Gl84] Glassner AS Space Subdivision for Fast Ray Tracing. IEEE Comput Graph Appl 4 (10): 15-22

[Gl88] Glassner AS Spacetime Ray-Tracing For Animation. IEEE Comput Graph Appl 8 (2): 60-70

[HH84] Heckbert PS and Hanrahan P Beam Tracing Polygonal Objects. Comput Graph 18 (3): 119-127.

[Ho92] Hodges LF Time-multiplexed stereoscopic computer graphics. IEEE Comput Graph Appl 12 (2): 20-30

[HZ82] Hubschman H and Zucker SW Frame to Frame Coherence and the Hidden Surface Problem: Constraints for a Convex World. ACM Trans Graph 1 (2): 129-162

[KK86] Kay TL and Kajiya JT Ray Tracing Complex Surfaces. Comput Graph 19 (3): 269-278

[Je92] Jevans DA Object Space Temporal Coherence for Ray Tracing. Proc Graph Interface 92: 176-183

[MH90] Murikami K and Hirota K Incremental Ray Tracing. Eurograph Workshop on Photosimulation, Realism, and Phys Comput Graph 23 (3): 15-29

[RW80] Rubin SM and Whitted T A 3-Dimensional Representation for Fast Rendering of Complex Scenes. Comput Graph 14 (3): 110-116

[SS89] Sequin CH and Smyr EK Parameterized Ray Tracing. Comput Graph 23 (3): 307-314

[Wh80] Whitted T An Improved Illumination Model for Shaded Display. Commun ACM 23 (6): 343-349.

Acknowledgments

"3D Buzz" was modeled by Thomas Meyer using the Wavefront render. The city model was created by Augusto Opdenbosh using AutoCAD.

Table 1

Comparison data for Frames 1-24 (Background Polygons Only)

Time is expressed in seconds per frame, rays in thousands per frame

	Fully Ray-Traced Frames	Inferred Frames
Mean Set Up Time[1]	N/A	5.6
Mean Reprojection Time[2]	N/A	34.8
Mean Total Time	261	295
σ for Mean Total Time	17	30
Mean Primary Rays	307	34
Mean Primary Shadow Rays	154	24
Mean Total Rays	461	58
Mean Primary Rays / Pixel	1.00	0.11
Mean Total Rays / Pixel	1.50	0.19
Savings in terms of rays cast		87.5%
Savings in terms of total time		-15.3%

[1] Set up time includes the time needed to create the transformation matrix and generate the bounding planes of the new viewing volume. σ is 0.5 seconds.

[2] Reprojection time includes the time to read the previous pixels, transform them, project them, and fix the bad pixel problem. σ is 1.8 seconds.

Table 2

Comparison data for Frames 25-600

Time is expressed in seconds per frame, rays in thousands per frame

	Fully Ray-Traced Frames	Inferred Frames
Mean Set Up Time[1]	N/A	6.8
Mean Reprojection Time[2]	N/A	27.7
Mean Total Time	5763	2693
Mean Primary Rays	307	62
Mean Primary Shadow Rays	291	58
Mean Secondary Rays	196	196
Mean Total Rays	795	317
Mean Primary Rays / Pixel	1.00	0.39
Mean Total Rays / Pixel	2.59	1.03
Savings in terms of rays cast		60.1%
Savings in terms of total time		53.3%
Savings in terms of primary and shadow rays cast		79.9%
Savings in time, diffuse scene (estimated)		70.2%

[1] Set up time includes the time needed to create the transformation matrix and generate the bounding planes of the new viewing volume. σ is 0.6 seconds.

[2] Reprojection time includes the time to read the previous pixels, transform them, project them, and fix the bad pixel problem. σ is 9.2 seconds.