Stereoscopic Ray-Tracing

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

Contact: Stephen J. Adelson
Phone: home: 404-299-6012
office: 404-894-5550
fax: 404-853-9378
e-mail: steve1@cc.gatech.edu

Abstract

In this paper, we describe a method to create an approximate ray-traced stereoscopic pair by transforming a fully ray-traced left-eye view into an inferred right-eye view. Performance results from evaluating several random scenes are given which indicate that the second view in a stereoscopic image can be computed with as little as 5% of the effort required to fully ray-trace the first view. We also discuss worst case performance of our algorithm, and demonstrate that our technique will always be at least as efficient as two passes of a standard ray-tracer.

Key Words:
Display algorithms, Three-Dimensional Graphics, Stereoscopic

Stereoscopic Ray-Tracing

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

Introduction

Ray-tracing, while widely-used to produce realistic scenes, is notorious for the amount of computation required. If we desire to create a stereoscopic ray-traced image, we must generate two different perspective views of the same scene, potentially doubling the required work. It is true, however, that while the left and right-eye views must be different in order to produce a three-dimensional image, they are at the same time identical in many respects. This similarity between the views makes it more efficient to generate the views simultaneously (or to infer the second of the pair from the precomputed first) than to treat them as two totally separate images (Adelson et al. 1991).

In this paper we describe an algorithm that simultaneously generates both the left and right-eye views of a ray-traced stereoscopic image by inferring the right-eye view from the fully ray-traced left-eye view. The choice of left-first generation was an arbitrary one on our part, and the following discussion can apply to right-first generation by making insignificant changes to the algorithm, all of which should be obvious to the reader.

We have chosen to measure computational savings in the text of the paper by comparing the simultaneous algorithm with the time beyond that needed to compute a single view. Using the single-image algorithm, if we can complete an operation on a scene in time t, then the upper bound on completing that operation simultaneously on two slightly different views is approximately $2t$.. The lower bound is t (i.e., the algorithm can generate two views as efficiently as one). If we view the time t to compute one eye

view as our benchmark and the increase in time beyond *t* as the time needed to compute the second view, then the decrease in computing time for the second view will range from 0% (total computing time for the second view = *t*) to 100% (total computing time for the second view = *0*).

Differences and Similarities Between Stereoscopic Views

A standard single view perspective projection is to project the three-dimensional scene onto the x-y plane with a center of projection at (0, 0, -d), as in figure 1. Given a point P = (x_p,y_p,z_p), the projected point is (x_s, y_s) where

$$x_s = x_p d/(z_p+d) \qquad (1)$$

and

$$y_s = y_p d/(z_p+d). \qquad (2)$$

For stereoscopic images we require two differently projected views of the image. Each of the observer's eyes is presented with a view of a scene from different (usually horizontally displaced) positions. For our purposes we will derive these views from two different centers of projection, but other methods are possible (Baker 1987; Sandin et al. 1989; Hodges 1992).

Assume a left-handed coordinate system with viewplane located at z=0 and two centers of projection: one for the left-eye view (LCoP) located at (-e/2, 0, -d) and another for the right-eye view (RCoP) located at (e/2, 0, -d) as shown in figure 2. A point P = (x_p, y_p, z_p) projected to LCoP has projection plane coordinates $P_{sl} = (x_{sl}, y_{sl})$ where

$$x_{sl} = (x_p d - z_p e/2)/(d + z_p) \qquad (3)$$

and

$$y_{sl} = y_p d/(d + z_p). \qquad (4)$$

The same point projected to RCoP has projection plane coordinates $P_{sr} = (x_{sr}, y_{sr})$ where

$$x_{sr} = (x_p d + z_p e/2)/(d + z_p) \qquad (5)$$

and

$$y_{sr} = y_p d/(d + z_p). \quad (6)$$

Assuming that we have already computed the left-eye position, the projected location of the right-eye position has the same y coordinate and x coordinate:

$$x_{sr} = x_{sl} + e(z_p / (d + z_p)). \quad (7)$$

In other words, the point will move horizontally between the views by a distance dependent on the depth **z** of the point to be projected, the distance **d** from the viewing position to the projection plane, and the distance **e** between the two viewing positions.

Previous Work

In a paper on generating stereoscopic animation, a system was created by Papathomas, Schiavone, and Julesz (1987) to generate frame pairs mathematically using the disparity, or separation, of the viewpoints. Their system was shown to be efficient; however, the data used in the program were only point sources so no consideration was given to the problems of occlusion and eye-point dependent shading that occur in stereoscopic ray-tracing. Other non-ray-tracing algorithms that exploit the similarities between the left and right-eye views to develop more efficient stereoscopic algorithms have been developed for polygon fill, clipping, and hidden surface elimination (Adelson et al. 1991).

Stereoscopic image generation may be viewed as a special case of the generation of animation frames. When generating ray-traced animation, we occasionally wish to create a sequence of frames in which the viewed objects are fixed and the viewpoint changes. Equivalently, the viewpoint may be fixed in space and direction, and the objects in the scene parade past the viewpoint at a single velocity and direction. It then becomes possible to take advantage of temporal coherence between frames to generate frames which are not exact, but approximate enough to serve as substitutes for completely ray-traced images. A stereo pair may be thought of as two frames of such an animation when the camera position has been shifted slightly in the horizontal direction. Since the data is fixed,

we know that the difference in the number of visible objects between frames (or individual views of a stereo pair) will come from two sources: objects entering and leaving the field of view, and occlusion, both inter-object and self-occlusion. The occlusion occurs in a predictable manner, and it is possible to use this to increase the speed of generating new frames (Hubschman and Zucker 1982).

Badt (1988) described a method for using *reprojection* to create a visible surface ray-traced scene in which the viewpoint has moved some small distance. Reprojection involves moving the pixels in one view to their inferred position in the second and cleaning up the image by recalculating only those pixels whose value is unknown or in question after the viewpoint has translated. Badt's algorithm used a box filter to determine which pixels required recalculation. Badt showed that his algorithm could save a significant amount of effort when creating the next frame; in his tests, almost 62% of the reprojected pixels were judged to be acceptable and did not require ray-tracing. Since a stereo pair is equivalent to two images in which the viewpoint moves a small horizontal distance, it follows that the same method could be used to create stereo pairs. This observation was first implemented by Ezell and Hodges (1990). One view, the left, was completely ray-traced. The second view was inferred from the previous view by reprojection, and a 25-pixel box filter was used to determine which pixels needed to be recalculated. Their algorithm resulted in a 50 to 75% reduction in the number of pixels that had to be ray-traced for the right-eye image.

By a more careful analysis of the situation specific to stereoscopic images, we have eliminated the filter and developed a more exact method for predicting the problem pixels after reprojection. Our current method eliminates the need to ray-trace up to 93% of the pixels in the right-eye view of a stereoscopic image (Devarajan and McAllister 1991; Adelson and Hodges 1992). Since then, we have extended the technique to include reflective and refractive objects. After reviewing our previous findings, we will present the results of that research.

Stereoscopic Visible Surface Ray-Tracing

The differences between the left and right-eye views in a stereoscopic image are due to different surfaces which are visible to each eye and to eyepoint-dependent optical phenomena of those surfaces such as reflection and refraction. In this section we consider only the visible surface problem.

When a left-eye view is reprojected to a right-eye view, there are three types of errors in right-eye view pixels which may require them to be completely recalculated. We call these errors the *missing pixel problem*, the *overlapped pixel problem*, and the *bad pixel problem*.

The missing pixel problem occurs when a pixel in the right-eye view has no value reprojected into it; such pixels are easily discovered and we ray-trace through them to eliminate the problem.

When pixels are reprojected between views, they move according to equation *(7)*. This distance is dependent upon the z value of the pixel in the original image. It is possible, however, that the z values of two adjacent pixels in the left-eye view are such that the second will be reprojected further than the first, leaving a gap of 1 to e-1 pixels. Further, it is possible that other pixels may have reprojected into this gap; these gap-filling pixel values may or may not belong in the right-eye view. It is these questionable pixels that cause the bad pixel problem.

In the overlapped pixel problem, more than one pixel in the originally generated view is reprojected into the same pixel in the second view. To properly render the scene, we must determine which reprojected pixel value to utilize.

Eliminating the Overlapped Pixel Problem

We know from our derivation of equations *(4)* and *(6)* that a reprojected pixel will not change its y value. We can therefore calculate both views of the image simultaneously, scan-line by scan-line.

The overlapped pixel problem occurs because more than one pixel in the first image is mapped to the same pixel in the second image. However, since we are able to create this image a scan-line at a time, a small data structure can be created to hold the points reprojected from the left image to the right. In its simplest form, this structure need only have an entry for each pixel in a scan-line with each entry containing a Boolean flag representing a reprojection to the corresponding pixel and a value for its color. Using this data structure, we can overwrite reprojected pixel values, resulting in the correct value at the end of processing.

Consider the ray from the RCoP through a pixel PIX_1 at $(x_1, y, 0)$ in figure 3. The equation of the z position of this ray is

$$z = -d + T * d = d * (T - 1) \qquad (8)$$

where $T > 0$ means that the ray may intersect an object of the image. Any object that intersects this ray in image space will appear in PIX_1 in the right-eye view.

Consider another ray from the LCoP through PIX_2 at $(x_2, y, 0)$. This ray will intersect the previous ray when the x values of the two rays coincide:

$$x_l = -e/2 + T(x_2 + e/2), \quad x_r = e/2 + T(x_1 - e/2)$$
$$T = e / (x_2 - x_1 + e) \qquad (9)$$

As illustrated in figure 3, a point on the surface of an object will appear in PIX_2 in the left-eye view and in PIX_1 in the right-eye view if and only if there is an object surface at $T = e / (x_2 - x_1 + e)$, $T > 0$.

Suppose we have another pixel, PIX_3, at $(x_3, y, 0)$, $x_2 < x_3$. A ray from the LCoP will intersect our ray from RCoP through PIX_1 at

$$T_0 = e / (x_3 - x_1 + e). \qquad (10)$$

If there is an object surface at this point, it will appear in PIX_3 in the left-eye view and PIX_1 in the right-eye view.

We now are faced with the overlapped pixel problem. However, note that the denominator of equation *(9)*, $x_2 - x_1 + e$, is smaller than that of equation *(10)*, $x_3 - x_1 + e$. Hence, $T_0 < T$ and the object surface at T_0 is closer to the viewing plane than the surface at T. Therefore, if we process pixels in a left to right fashion and overwrite the scan-line record as we go, we will always end up with the correct object reprojected to the pixels in the right-eye view.

Eliminating the Bad Pixel Problem

The bad pixel problem is illustrated in figures 4a and 4b. In both figures, the two leftmost pixels in the left view reproject into the right view such that there is a gap between them. The question we consider in the bad pixel problem is whether there is a true gap, or if we face an artifact of changing occlusion. In figure 4a, we can see that there is indeed a true gap and the farther object A should be reprojected into this area. In figure 4b, however, there is an intervening object that would block A were the right-eye view fully ray-traced. This problem was previously handled by Badt and Ezell and Hodges by using a filter to test each pixel. It was hoped that the bad pixels would not totally fill the gap and surrounding missed pixels will disclose the location of improper pixels (Badt 1988; Ezell and Hodges 1990). Unfortunately, it is fairly easy to imagine a case where improper pixels are missed, or where they are caught but the filter causes many good pixels to be re-traced.

This problem can be solved exactly for stereoscopic images by clearing any previously projected values in intervening pixels when two adjacent pixels in the left-eye view, x and x+1, are reprojected to the right-eye view such that **newx**(x+1) - **newx**(x) > 1, where **newx** is the reprojection equation *(7)*. Since pixels are often small relative to the objects they are displaying, the case of figure 4b seems much more likely than the case of 4a. Also, we must check the reprojected position of the final pixel on each scan-line. If the final pixel has a negative z-value (i.e. it lies in front of the screen), **newx** reprojects the pixel to the left of the original position. We must then clear all the pixels on the remainder of the scan-line to solve the bad pixel problem. Note, however, that a pixel in such a position would appear to be

a part of an object cut off by the frame, which appears physically behind the object. The probability that such a pixel will be placed in a stereoscopic scene is therefore minimal.

Certainly, it is possible that our solution eliminates several good pixels, but the possibility of waste when displaying spheres or relatively smooth polygonalized surfaces appears slight. We shall see later that the number of pixels reset in this process, on average, is small. In any case, it is absolutely impossible for bad pixels to exist in the right-eye view when using this method.

The Visible Surface Algorithm

We now define the visible surface algorithm for creating stereo pairs of ray-traced images. The pseudo-code below is also valid for full ray-tracing, but the color calculation in that case may involve ray-tracing far more rays than the single shadow ray necessary for visible surface ray-tracing.

Assume we are creating both views in tandem or have access to pixel information of one image in left to right fashion for each scan-line. For M by M resolution, we need a scan-line record line_rec of length M containing the following information for each pixel j, $0 \leq j < M$:

HIT : flag which is TRUE if there is a reprojection to pixel j, FALSE otherwise

COLOR : the color for the right-eye view of pixel j.

When we are creating images in tandem, COLOR can be computed as we go. Otherwise, we must also have access to information which will allow us to create a color for the right-eye view; this would include light-point position, specular reflection coefficients, intersection point, intersection normal, diffuse object color, and other such information. The following pseudo-code algorithm assumes a tandem creation scheme with overwrite reprojection, and no positive z objects against the right edge of the image:

For each scan-line:

OLDX = -1
For each pixel i in the left-eye image, $0 \le i < M$:
Ray-trace through pixel i to find intersection at (i_X, i_Y, i_Z) or miss (i_Z = infinity)
Calculate color for left-eye view
Let j = **newx**(i) {i + e(i_Z / [i_Z + d])}
If j < M:
Calculate color for right-eye view
Let line_rec[j].HIT = TRUE, line_rec[j].COLOR = right-eye color
else j = M
If j - OLDX > 1 then
For each pixel k in the right-eye image, OLDX < k < j
Let line_rec[k].HIT = FALSE
Let OLDX = j

We may proceed with the same algorithm using a pre-computed left-eye image as long as the data required to reproject each pixel and calculate the right-eye view color is available.

Another variation

Since we know that the last reprojection to a pixel when processing left to right is the correct reprojection, it follows that we could process right to left and allow the first reprojection to a pixel to be the only reprojection. During processing, we generate the reprojection of a pixel as before, but only set line_rec[**newx**(x)].COLOR if line_rec[**newx**(x)].HIT = FALSE. If any two consecutive pixels x and x-1 in the left-eye view are reprojected to the right-eye view such that **newx**(x) - **newx**(x-1) > 1, then line_rec.HIT of all pixels in the right-eye view between the two reprojected values are set to TRUE. This will take care of the bad pixel problem.

It would seem that this method would perform better than the previously described overwrite method, but there is one difference that makes them very close in performance: we must guarantee that the scan-line record is cleared before the processing of a new scan-line begins. In the overwrite method, every record element is overwritten to the correct value or

blanked by the solution to the bad pixel problem. The new method relies on comparisons with the current values of the scan-line records. This resetting takes enough time that comparison of the two methods shows no significant difference in performance.

Limitation of the Algorithm

The main limitation of the algorithm is that, unlike Badt's original algorithm, we cannot process the pixels in any randomized order. We must have access to the pixels in either a left to right or right to left manner consistently, although the scan-lines could be processed in any order and in parallel, if desired. However, it is this limitation that provides the ability to dispense with the filter notion and other tests which appear partially because of the random order processing.

Performance of Stereoscopic Visible Surface Ray-Tracing

Our algorithm was implemented in a ray-tracer and used to test several scenes with diffuse objects and specular highlights, the full results and analysis of which appear in Adelson and Hodges (1992). All images consisted of 20 ellipsoids (40% spheres) of random size, positioning, and color. Every image was of 512 by 512 resolution, containing 262144 pixels. One image from this group appears in figure 5.

On average, only 18521 pixels in the right-eye view, or 7.07%, needed to be re-traced. Many of these pixels were those on the left side of the image which were not visible in the left-eye view. As an example of the significance of the left edge, if the first column of pixels contained nothing (hit at infinity), each one would be moved $e(z_p/[d + z_p])$, z_p very large, or e, pixels in the right-eye view. This means that in a N by N resolution scene, there could be as many as eN pixels to re-trace because of the left edge alone. For the values of e and N as tested, this corresponds to 11520 pixels, or 4.39% of the image. Of course, if there was a planar object at z = 0 then the pixels would be moved e(0/[d + 0]) or 0 pixels. In practice, we have estimated that about half of the pixels needing to be re-traced were in the left-edge of the screen.

Figure 6 displays the good / bad pixel image for the scene in figure 5. Pixels which did not require re-tracing appear in green; those that did, in

red. In addition to the large left edge gap, note the crescent shaped areas requiring rays. These shapes are caused by isodepth reprojections of pixels, which on ellipsoids are curved. Each isodepth quantum moves by an amount dictated by equation *(7)*. The larger the depth, the farther to the right the pixels move. This is to be expected when we consider that by moving the viewpoint to the right to create the second half of the stereoscopic pair, we can see more of the right side of objects than we could in the left-eye view. Hence, gaps in the image which we must fill.

Several of the scenes were tested by running just the left-view and doubling that time to estimate the time required to ray-trace both views. This time averaged 504 seconds. Since the stereoscopic ray-tracer generated both views in an average 291 seconds, the algorithm saves over 84% of the computational time to produce the second view. To try and calculate the correctness of the scenes being generated, one reprojected and re-traced right-eye view was compared pixel by pixel to the completely ray-traced right-eye view. 14.6% of the pixels did not hold identical values, but they were only incorrect by a very small amount: an average distance of 2.64, or 0.60%, where distance is measured as linear distance in RGB (256 x 256 x 256) space. Visually, the scene as rendered by the reprojection stereoscopic algorithm is identical to a scene rendered by two passes of a standard ray-tracer.

Extending to Multiple Rays per Pixel

We now examine the possibility of multiple rays per pixel. Multiple rays are desirable because they allow a better representation of the image seen in a pixel and provide anti-aliasing. If we decide to have N rays per pixel, the color of the pixel will be some weighted average of all the rays through the pixels. To test multiple rays, the positioning of these N rays must be chosen. For our first test, there were 9 rays distributed evenly through the pixel, one on each corner, one in the middle of each side, and one in the center. Special data structures were created so that the top row of rays in a pixel could be re-used as the bottom row on the next scan-line and the right column of pixels could be re-used as the left column of the adjacent pixel. Since this re-use of five rays can be provided for nearly every pixel, it was expected that time would only be increased by a factor of four.

The first question which arises when using multiple rays per pixel is how to reproject a left view pixel. More precisely, what value should be used for the intersection z value when any of 9 sub-pixels may have hits? The obvious answer is to average all z values, but this method does not work when not all the rays hit an object. If only z values of rays that hit an object are averaged, the color of a close object (which should be reprojected by a short distance) will be moved too far if many of the rays hit an object much farther away. We decided to allow only the middle ray to act as an *aiming ray* for the entire pixel color.

If we simply reproject based on the aiming ray, a ghost outline will appear to the right of every object in the image. This ghost image occurs when the aiming ray of a pixel falls to infinity, but some of the sub-pixels still hit an object; when the colors are averaged and the pixel reprojected, a dim object-colored pixel will be rendered. The problem is simple to correct by changing the basic algorithm slightly: when two adjacent pixels are separated after reprojection by more than one pixel, all pixels from the one after the first through the second are eliminated. This results in only slightly higher re-tracing costs.

We ran the 30 scenes used with the single-ray method through the multiple ray-tracer. An average scene using nine rays per pixel takes 1075 seconds to render, or 3.69 times the speed of the single-ray per pixel method. (The time is less than the predicted four times of the single ray-tracer because the coding allows the work to be performed without requiring four times the subroutine calls, a significant element on our graphics workstations.) The average scene also contained 20557 missed pixels, or 9.93% more than the single ray (7.84% of the entire image), all of which are a result of the extra blanked pixel as explained above. Rendering the scenes individually for both views required 1780 seconds, implying a speed up of almost 80% by the stereoscopic algorithm. Again, one right view was compared pixel by pixel with a fully ray-traced right view. 15.6% of the pixels were judged to be off, but still by a very small amount: 3.18, or 0.72%.

Adding Reflection and Refraction

Inferring pixel values when reflection and refraction are involved is not as straight forward. The reason for this complication is that whereas in the reprojection equation *(7)* we need only the z-value of a point to calculate the new position of a pixel, calculating reflection requires the use of the normal to render the image. For refraction we must also know the normal, shape of the object, normal on the far side, and the index of refraction of the material! Except in a few very special cases, we find that we cannot use the reprojection algorithm for anything beyond the first level of ray-tracing.

Yet having just that first level subject to reprojection can save significant computation in most scenes. Since images rarely consist of a preponderance of reflective and/or refractive objects (with the exception of the occasional demonstration image), we will observe that we can use the reprojection algorithm and still achieve significant computational speedup.

Experiments in stereoscopic reflection and refraction

We changed the rendering algorithm so that when any ray-tracing levels beyond the first were needed the algorithm produced two rays, one for each eye. This departure reflects the idea that our technique cannot be used to render reflection and refraction in the generalized case. We took a 125 ellipsoid image (with Phong highlighting) and tested the savings on the image at one ray per pixel with no reflection or refraction, with 10% of the objects made refractive, 10% of the objects made reflective, and with both. These images appear in figures 7 and 8. While the placement of the objects affects the exact savings (a refractive object seen through another seen through a third involves far more rays than a diffuse object seen in a single reflective object), the results are indicative of the level of savings achievable. These results are summarized in table 1.

As we expected, the algorithm savings dropped from 94.6% in the all-diffuse image to 89% in the reflective image (one extra ray per reflective object hit) to 80% in the refractive image (two extra rays per refractive hit) and finally down to 72% in the combined image. Certainly a fall from the diffuse image, but the majority of the savings remain.

We also tested several scenes containing 10% of each type object at the 100 and 1000 object level. These results appear in table 2. While the savings drop to as little as 68%, it should be noted that this is not an average case. The 1000 object scenes were randomly built out of objects of the same size as the fewer object scenes. Hence, the 1000 object scenes were very crowded, and the relatively large objects made it likely for inter-object reflection and refraction to occur. Therefore, we feel that the 68% savings is a informal lower bound for saving on average scenes.

Reprojection Values - a New Problem

When we reproject a point in the left-eye view into the right-eye view, we use integer values in the line record to correspond to integer pixel locations. This means that a point may reproject to a position in the right view which is off from its true reprojection by as much as half of a pixel. We have shown previously that for diffuse objects this difference is effectively insignificant (Adelson and Hodges 1992). However, when we add reflection and refraction the color becomes more important.

We compared the inferred right-eye image from figure 8 pixel by pixel to one completely ray-traced. We found that the pixels which did not match were off in color by a distance of 6.6 units in RGB space. We calculated in our 1992 paper that a scene containing only diffuse objects would have pixels off by an average of 2.69. As illustrated in figure 9, most of this difference occurs because of refraction. Since most of this difference is small, we have enhanced the pixels so that RGB values of 0-10 are mapped to the range 0-255. We clearly see in this image that most of the worst pixels are in the refractive objects. This occurs because of the first level of ray-tracing being off by up to half a pixel. What we see refracted in a pixel is correct (the algorithm makes two rays for refraction), but the location of the pixel may be off.

While these differences are relatively minor, binocular rivalry can make small objects seen through or reflected in other objects look indistinct and even non-stereo (Hebber and McAllister 1991; Levelt 1968). The brain, in trying to fuse the image of a reflective or refractive object, finds that one

of the images is slightly off and refuses to merge the smaller objects into a stereoscopic image. One possible way to fix this problem is to make sharper images by using more rays.

Reflection and Refraction with Multiple Rays Per Pixel

As described previously, we formerly reprojected with multiple rays per pixel by using an aiming ray through the center of the pixel. While this technique worked sufficiently well for diffuse objects, other problems appear when we attempt to use the method with more complex phenomena. Using an aiming ray means that individual pixels of lines and curves in the x-z plane which projected nearly vertically in the left eye view and were anti-aliased by a multiple ray per pixel technique would project different distances in the right-eye view because of their depths and make aliasing re-appear. This aliasing is acceptable in smooth curved objects like ellipsoids, but it is highly objectionable in straight-edged objects and grids such as our background checkerboard. And, of course, this does little to alleviate the problem of indistinct reflected and refractive objects.

Another method for generating multiple rays which avoids this problem is to multiply the rows and columns by n and filter each n by n block into a single pixel. The ray through each sub-pixel is jittered randomly from the center of the sub-pixel to alleviate aliasing which may occur even at this level. It is then a simple matter to treat each row to the reprojection algorithm. We submit a stereo pair rendered with this technique: figure 10 shows 80 reflective spheres. This method effectively solves both the re-appearing aliasing and the indistinct higher-level rays problem. We would expect the execution time and number of pixels which must be re-traced to be increased by approximately a factor of n^2. In fact, as table 2 shows, this is exactly what happens.

Using the n by n method, we require $O(n^2)$ primary rays per pixel for each view (a value very close to $0.55 * n^2$). This number grows geometrically with the number of rays per pixel. Since this is such a high cost to pay for improved image quality, it would be far better if we efficiently convert the adaptive super-sampling ray-tracing technique for stereoscopic rendering (Whitted 1980). In standard adaptive super-sampling, we first trace

through the four corners of a pixel. If the colors of the four points do not match within some tolerance, we sub-divide the pixel into four equally-sized sub-pixels and repeat the process (reusing the previously traced positions and colors) until we are within the set tolerance or else have reached a level beyond which little improvement is made. Each sub-pixel is averaged together using an area-dependent weighting function until one color is returned.

To reproduce this type of ray-tracing in stereo, we need a method of preserving rays between views. For this we will make an n by n data structure representing a single pixel as illustrated in figure 11, where n will determine the number of levels of adaptivity we will allow. If we wish to allow a maximum of k levels of adaptivity, $n = 2^k + 1$. At first, we ray-trace through positions (1,1), (n,1), (1,n) and (n,n). If the color values are not within tolerance, we subdivide into four pixels of size $2^{k-1} + 1$ and ray trace through positions (n/2,1), (1,n/2), (n/2, n/2), (n,n/2), and (n/2,n). Ray-tracing only what is needed, we then are able to use our reprojection technique on each of the n rows. We also note that, because of the regular spacing, we are able to reuse the bottom row and left column of each pixel in the left view and the bottom row in the right view.

We analyzed the results from a maximum of two and three levels of super-adaptive stereoscopic ray-tracing on each of the images presented previously as well as a complex image of 1000 ellipsoids (figure 12) and the Georgia Tech mascot "Buzz" (figure 13), a partially reflective and refractive image. This is approximately equivalent to the level of quality generated by 9 and 25 equally-spaced rays per pixel. The results appear in Table 3. In all cases, our technique generated a savings of at least 50% over a standard super-sampling method.

A Discussion of Theoretical Worst Cases

Ray-tracing is computationally intensive because of the necessity of finding intersections with the objects in the scene to be rendered. Over a decade of research has resulted in many techniques to streamline this operation, but in complex scenes intersections are still a far greater computational problem than generating the color for the intersected object. For example, Whitted stated that his program spent 75% of its time on ray-

surface intersections for simple scenes and more that 95% for complex scenes - the type we are concerned with for calculating worst cases (1980). Therefore, in the following discussion we will limit ourselves to the calculating the savings in number of rays generated and ignore both the savings in reusing the object color and the overhead of reprojecting positions of objects from one view to the next.

Since we are reprojecting objects from the left view to the right, there is a single special case in which the technique will fail to render the proper image. This case occurs when the right-eye can see an object which the left eye cannot and it blocks the right eye's view of the shared stereoscopic area. As we see in figure 14, such an object would have to be situated to the right of the crosshatched region but in the right-eye's view. However, the figure is exaggerated to illustrate the case. For example, the imaging plane as used in our renderer has a width approximately 40 times the interocular distance, and the distance to the imaging plane is also of this magnitude. If such an object did exist, it would exist in the monoscopic vision area of the right eye only and as such would not detract from the stereoscopic image and would likely not be missed by a viewer. Also, note that the region exists only on the viewer side of the imaging plane. Any objects placed on the far side of the imaging plane are "safe" in that the conditions for creating the situation cannot arise. Additionally, because of physical constraints on the user, objects are never placed closer to the user than half of the distance between the viewing positions and the imaging plane (and typically not even that close to the viewing positions), so the region with which we must concern ourselves is smaller yet.

The savings generated by our stereoscopic technique derives from the our ability to reproject the primary rays and to know whether a given point is in shadow by re-using the shadow rays (assuming the object has the slightest diffuse color component - certainly true in any photo-realistic image). These savings are lost if the right eye cannot see a portion of the left eye's view. In the absolute worst case, the left and right eyes will see completely different views. This situation could arrive, for example, in the case where an object is so close to the left eye that the right eye does not see it at all, or if a "wall" were placed between the eyes. In that case, we

would be required to double our efforts to create the right-eye view. However, this would be no worse than using a standard technique twice (with the exception of the insignificant reprojection costs).

We now calculate theoretical worst cases based on the percentage of primary rays from the left-eye view which are visible in the right-eye view. In this analysis, we will use the following variables:

N the number of primary rays in the left-eye view. We will assume that this number is equivalent to the number of primary rays in a fully ray-traced right-eye view.

M the percentage of primary rays ray-traced in the right-eye view relative to N, expressed as a decimal number between 0.0 and 1.0.

S the number of light sources and, therefore, the number of shadow rays per hit.

L the maximum number of ray-tracing levels allowed before black is returned as the color.

v the number of levels we assume that reflections and refractions continue. Note that $v \leq L$.

Rendering misses: The fully ray-traced view would generate N primary rays, the inferred image MN primary rays.
Worst case savings: 100 * (1 - M)%

Rendering diffuse objects (including texture-mapped objects, bump-mapped objects, or any other non-eye-point dependent phenomena): The fully ray-traced view generates N primary rays and NS shadow rays. The inferred image reuses these rays and generates MN primary rays and MNS shadow rays.
Worst case savings: 100 * (1 - M)%

Rendering a scene of objects that are partially diffuse and partially reflective or refractive: The fully ray-traced view would generate N primary rays, S shadow rays for each object intersected, and a reflected or refracted ray for each object intersected at level less than L, for a total of N ((1+ S) v (+ 1 if $v < L$)). MN of the rays in the inferred view will also have

this number of rays, and the rest will be able to reuse the primary and shadow rays from the left-eye view. The inferred view will have $MN((1 + S)\ v\ (+\ 1 \text{ if } v < L)) + (1 - M)N((1 + S)\ v - (1 + S)\ (+1 \text{ if } v < L))$.
Worst case savings: if $v = L$,
$100 * (1 - ((1+S)LM + (1-M)((1+S)L - (1+S))) / (1+S)L) = 100 * ((1 - M)/ L)\%$

If $v < L$,
$100 * (1 - (M((1+S)v + 1) + (1 - M)((1+S)v - S)) / ((1+S)v + 1)) =$
$100 * (1 - M) * (1+S) / ((1+S)v + 1)\%$

Rendering a scene of objects that are diffuse, reflective and refractive (such as panes of glass): The fully rendered view contains N primary rays, S shadow rays for each object intersected, and two rays (one reflecting and one refracting) for each object intersected at a level less than L. Total: $N((2^v - 1)\ (1 + S)\ (+\ 2^v \text{ if } v < L))$. The inferred view will have MN with this total, and the rest will be able to reuse the primary and first level shadow rays, for $(1-M)N((2^v - 1)\ (1 + S) - (1 + S)\ (+\ 2^v \text{ if } v < L))$.
Worst case savings: if $v = L$,
$100 * (1 - (M(2^L - 1)(1 + S) + (1 -M)(2^L - 2)(1 + S)) / ((2^L - 1)(1 + S)) =$
$100 * (1 - M) * (1 / (2^L - 1))\%$.

If $v < L$,
$100* (1 - (M(2^v - 1)(1 + S) + 2^v) + (1 -M)(2^v - 1)(1 + S)- (1 + S) + 2^v)) / ((2^v - 1)\ (1 + S) + 2^v)) =$
$100 * (1 - M) * (1+ S) / ((2^v - 1)\ (1 + S) + 2^v)\%$

The following two cases are quite unusual in photo-realistic images as they are entire scenes of images without diffuse components, but we include them for completeness:

Rendering a scene of objects that are 100% reflective or refractive: The fully rendered view requires N primary rays and a reflected or refracted ray for every object intersected at level less than L, for a total of $N\ (v\ (+\ 1 \text{ if } v < L))$. The inferred view will have MN rays at $(v\ (+\ 1 \text{ if } v < L)$ and the rest at $(v - 1\ (+\ 1 \text{ if } v < L))$.

Worst case savings: if v = L,
$100 * (1 - (ML + (1 - M)(L - 1)) / L) = 100 * (1 - M) / L\%$.

If v < L,
$100 * (1 - (M(v + 1) + (1 - M)(v)) / (v + 1)) = 100 * (1 - M) * (1 - v / (v + 1))\%$.

Rendering a scene of objects that are reflective and refractive and have no diffuse component: The fully rendered view has N primary rays and two rays (one reflecting and one refracting) for each object intersected at a level less than L. Total: $N((2^v - 1$ ($+ 2^v$ if v < L). The inferred view will have MN with this total, and the rest will be able to reuse the primary ray.

Worst case savings: if v = L,
$100 * (1 - (M(2^L - 1) + (1 -M)(2^L - 2)) / (2^L - 1) = 100 * (1 - M) * (1 / (2^L - 1))\%$.

If v < L,
$100 * (1 - (M(2^{v+1} - 1) + (1 -M)(2^{v+1} - 2)) / (2^{v+1} - 1) = 100 * (1 - M) * (1 / (2^{v+1} - 1))\%$.

Assuming that L has the common value of 5, we can calculate the worst case savings in terms of M (The value of S is unimportant for calculating the worst case):

Misses	100.0 * (1 - M) %
Diffuse	100.0 * (1 - M) %
Reflective / Refractive and Diffuse	20.0 * (1 - M) %
Reflective, Refractive, and Diffuse	3.226 * (1 - M) %
Reflective / Refractive, no Diffuse	20.0 * (1 - M) %
Reflective and Refractive, no Diffuse	3.226 * (1 - M) %

Unless we are rendering the unusual case where each eye sees a completely different view, the stereoscopic technique will always be better than two passes of the non-stereo technique. In all cases the stereoscopic

rendering technique will be at least as efficient as two passes of a standard method.

The value of M will vary with the interocular distance e, the complexity of the image, and the rendering technique. Recall from reprojection equation *(7)* that the reprojection distance of a pixel is $e(z_p / (d + z_p))$, or a maximum of e pixels. It follows, then, that the larger the value of e, the larger the size of potential gaps caused by the left-hand edge of the image and our solution the bad pixel problem. Therefore, M will rise proportionally with e.

However, for a given renderer, e does not often change. A more important influence is the complexity of the image. The more crowded an image is, the less likely we are to have large differences between z-values of consecutive pixels in the left-eye view, so gaps needing to be filled in the right-eye view will be smaller. For example, we find that M falls from 0.15 in the relatively sparse 20 ellipsoid image to 0.05 in the extremely crowded 1000 ellipsoid example.

We would also like to stress that although the ellipsoids we have rendered are simplistic images, the results are still valid for more complex figures. Consider Table 4, which shows the reprojection distances for ranges of z, given the values of d and e as used in our renderer. As long as a given object or polygon stays within a particular reprojection quantum, the complexity of the object is irrelevant. Two consecutive pixels in the left-eye view will remain consecutive in the inferred view, because they will be reprojected by the same distance. On the other hand, gaps will open when objects cross quantum lines in the positive direction; hence the crescent shape of the re-traced pixels in our images. Planar objects would have vertically-oriented re-trace areas. The objects which could potentially create problems are those which cross quantum lines several times in both directions, causing large amounts of self-occlusion and thereby expanding the bad pixel problem. Fortunately, such objects are often comprised of many small polygons which tend to stay in the same quantum, and we believe that we can safely ignore their existence when analyzing the general savings of the algorithm.

Finally, the method of rendering will affect M. In the single-ray per pixel ray-tracing, we never see a value of M greater than 0.18. In the adaptive super-sampler, however, the n by n data structure opens the possibility of rays from the original view reprojecting into portions of the inferred view which may not be used when sampling the view, as we can see in figure 15. As a result, the more possible levels of super-sampling we allow, the higher M will be. Surprisingly, this is one case where the images with higher frequency will improve the value of M, since this will require more rays in the original view and therefore lower the chance that no rays will reproject where we need them in the inferred view data structure. In testing all of our images for two and three possible levels of super-sampling, the value of M never rose above 0.3, generally staying in the 0.2 - 0.25 range. The image with the lowest frequency range, the 20 ellipsoid image, was tested on a 9 by 9 data structure, allowing a possibility of four levels of super-adaptivity. Even in this case where we might expect little savings, M remained at 0.338, giving an overall savings of 66% over a standard adaptive super-sampler with four levels.

Conclusions

We have shown that the reprojection technique is a valuable tool for creating a ray-traced stereo pair, saving up to 95% of the effort of creating a second image of a stereo pair. It is limited by the necessity of a pixel processing order as well as the accuracy of color in the reprojected pixels, whose location may vary by as much as half a pixel, but this problem is insignificant for diffuse objects and greatly rectified by a multiple rays per pixel technique using multiplied rows and columns or stereoscopic adaptive super-sampling.

It is a simple matter to intuit where the technique will become less efficient. The algorithm's strength lies in its ability to reposition the first level of ray-tracing. It follows, then, that the less reflection and refraction existent in a scene, the greater the computational savings due to the algorithm. Secondly, we must always re-trace those pixels that are ambiguous. These pixels are a result of a discrepancy in z values between two consecutive points in the left image. Discrepancies are minimized

when the scene is "busy" so that rather than one pixel hitting an object and the next falling to infinity - opening a large gap - the second pixel instead strikes another object, making a smaller gap. If we make the reasonable statement that the computations of a ray-tracer are dominated by finding intersections, we find that the overhead of reprojection, whose location can be found with two additions and two multiplications, is insignificant. Therefore, the algorithm is always as efficient, and usually much more efficient, than a standard ray-tracer.

References

Adelson SJ, Bentley JB, Chong IS, Hodges LF, and Winograd J (1991) Simultaneous generation of stereographic views. Comput Graph Forum 10 (1): 3-10

Adelson SJ and Hodges LF (1992) Visible Surface Ray-Tracing of Stereoscopic Images. Proc SE ACM: 148-157

Badt, Jr. S (1988) Two algorithms taking advantage of temporal coherence in ray tracing. Vis Comput 4 (3): 123-132

Baker J (1987) Generating images for a time-multiplexed stereoscopic computer graphics system. SPIE Proc 761: 44-52

Devarajan R and McAllister DF (1991) Stereoscopic ray tracing of implicitly defined functions. SPIE Proc 1457: 37-48

Ezell JD and Hodges LF (1990) Some preliminary results on using spatial locality to speed up ray tracing of stereoscopic images. SPIE Proc 1256: 298-306

Hebbar PD and McAllister DF (1991) Color quantization aspects in stereopsis. SPIE Proc 1457: 233-241

Hodges LF (1992) Time-multiplexed stereoscopic computer graphics. IEEE Comput Graph Appl 12 (2): 20-30

Hubschman H and Zucker S (1982) Frame-to-frame coherence and the hidden surface computation: constraints for a convex world. ACM Trans Graph 1 (2): 129-162

Levelt WJM (1968) On Binocular Rivalry. Mouton, The Hague

Papathomas TV, Schiavone JA, and Julesz B (1987) Stereo animation for very large data bases: case study - meteorology. IEEE Comput Graph Appl 7 (9): 18-27

Sandin DJ, Sandor E, Cunnaly WT, Resch M, DeFanti TA, and Brown MD (1989) Computer generated barrier-strip autostereography. SPIE Proc 1083: 1-10

Whitted, T (1980) An Improved Illumination Model for Shaded Display. Commun ACM 23 (6): 343-349.

Acknowledgements

The authors would like to recognize the anonymous reviewers who provided us with many helpful suggestions for the improvement of this paper.

Line Drawn Figures

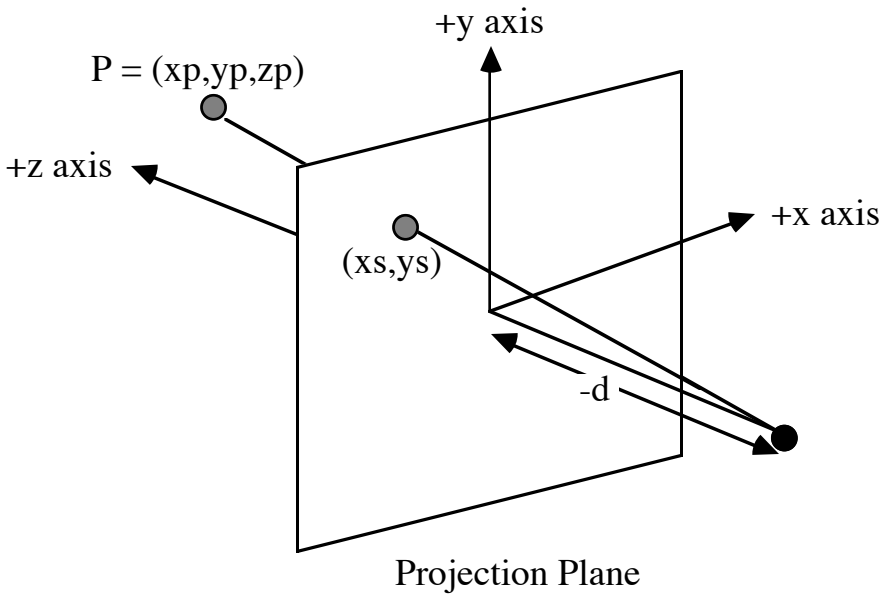

Figure 1.
Standard Perspective Projection

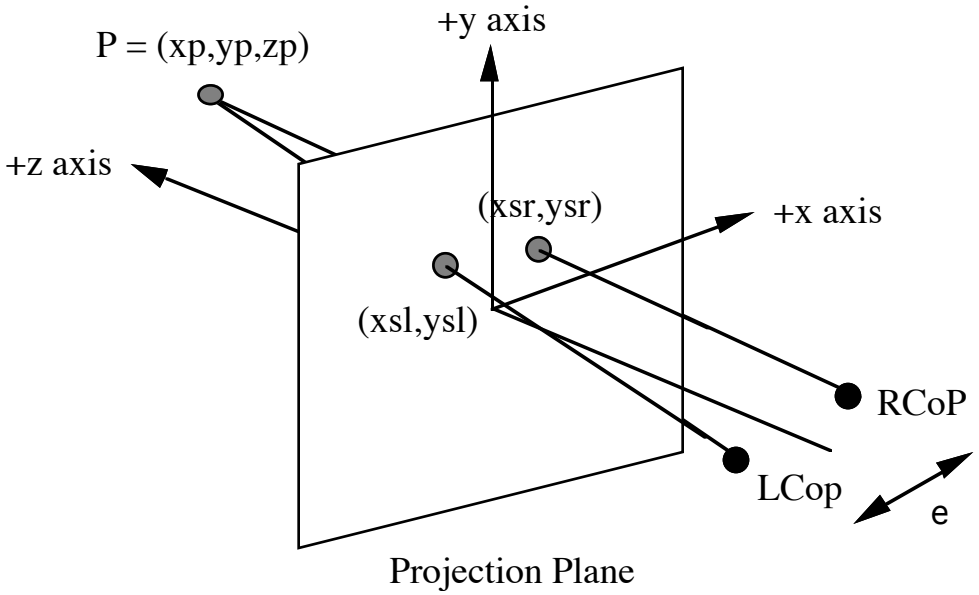

Figure 2.
Stereoscopic Perspective Projection

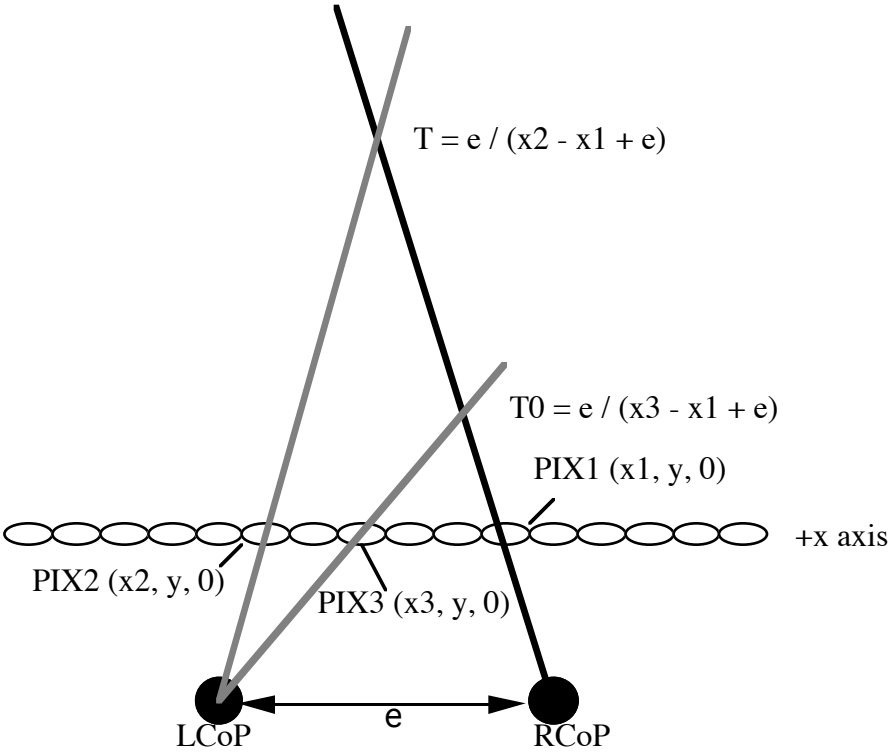

Figure 3.
Solving the Overlapped Pixel Problem

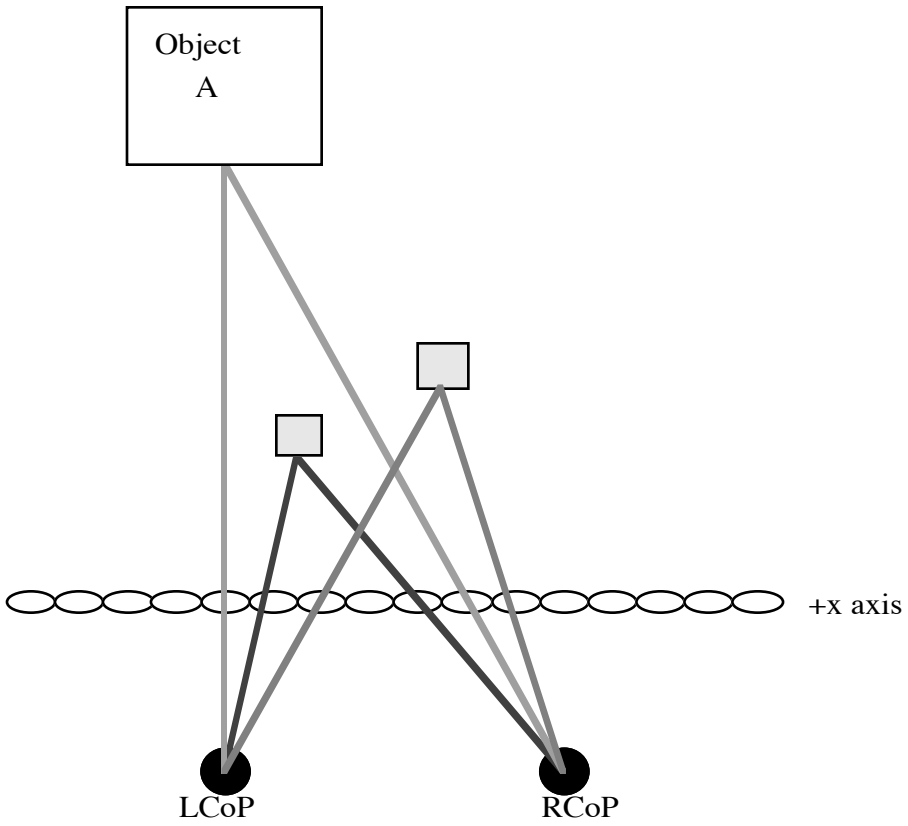

Figure 4a.
Object A should appear in right-eye view

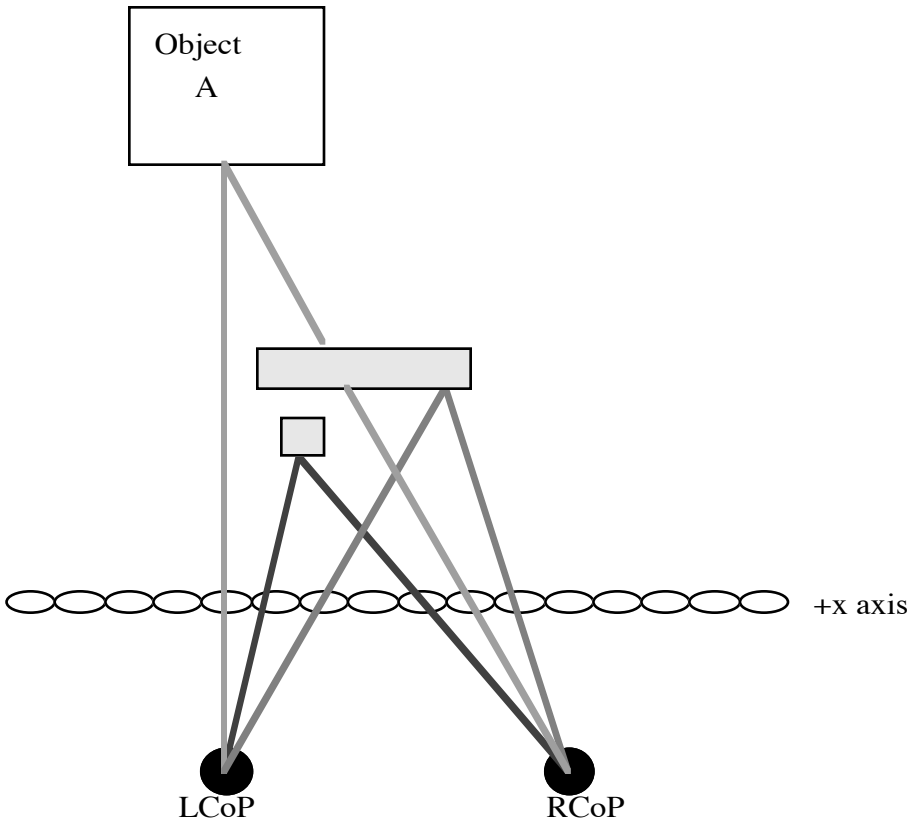

Figure 4b.
Object A is blocked in the right-eye view and should <u>not</u> appear

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 11. 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.

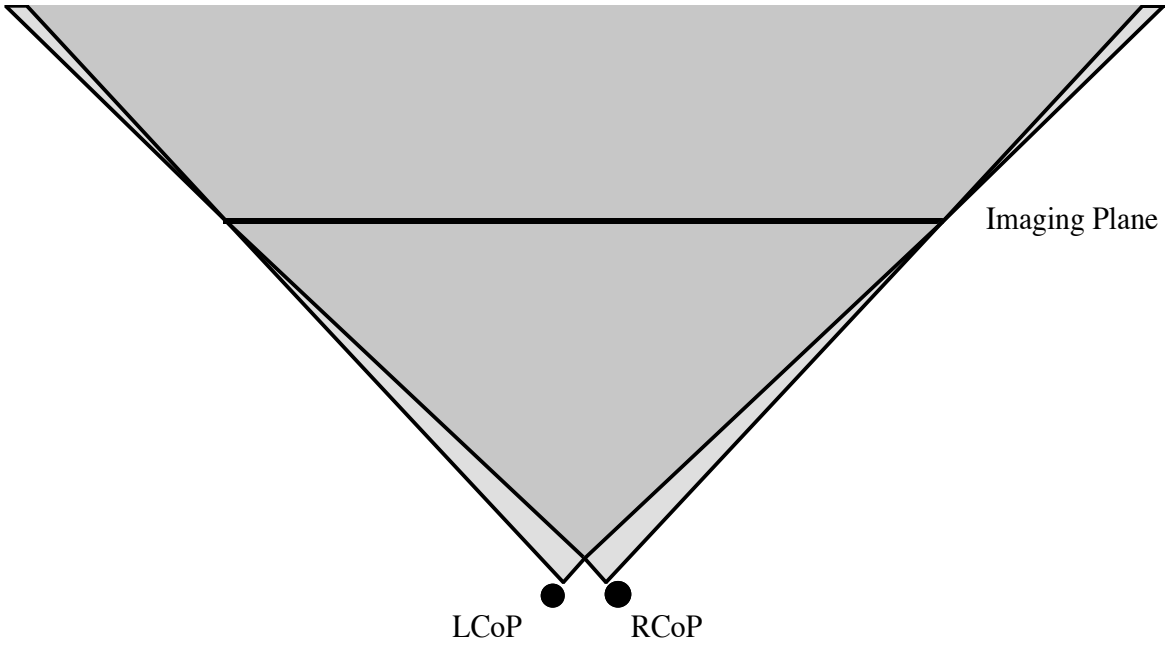

Figure 14. The case where the algorithm fails.
An object must appear the right eye's view, but not in the region shared with the left eye (crosshatched area).

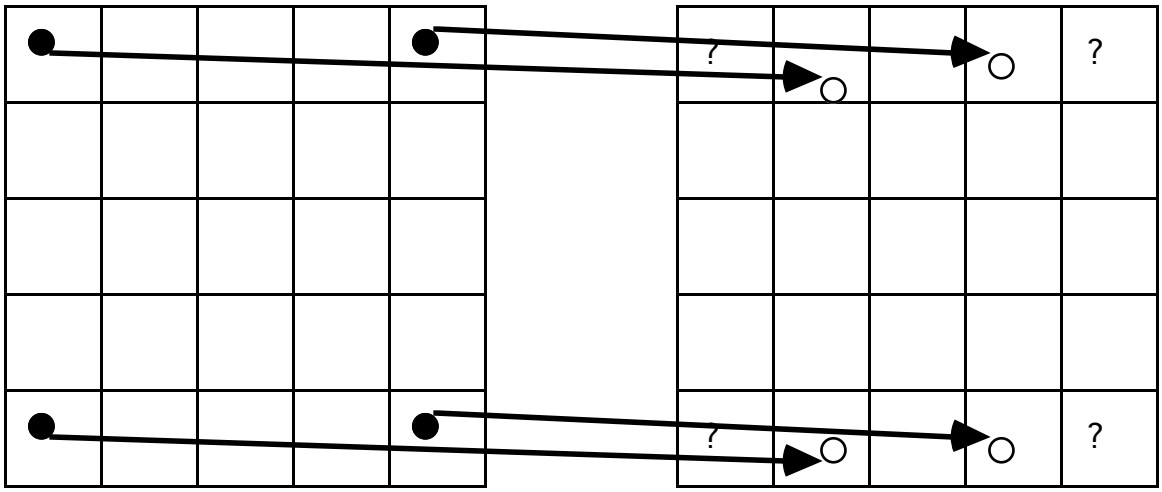

Figure 15. Reprojection doesn't always mean savings!