Simultaneous Generation of Stereoscopic Views

Stephen J. Adelson[1], Jeffrey B. Bentley[2], In Seok Chong[3], Larry F. Hodges[1] and Joseph Winograd[1]

1. College of Computing, Georgia Institute of Technology

2. Racal-Milgo, Atlanta, Georgia

3. Dept. of Electrical Engineering & Computer Science, Northwestern University

Abstract

Currently almost all computer graphic stereoscopic images are generated by doubling the work required to create a single image. In this paper we derive and analyze algorithms for simultaneous generation of the two views necessary for a stereoscopic image. We begin with a discussion of the similarities of the two perspective views of a stereo pair. Following this, several graphics algorithms that have been optimized from known single-view methods are described and performance results obtained from testing the new stereo algorithms against the originals are presented.

Categories and Subject Descriptors: I.3.2 **[Computer Graphics]**: Graphics Systems--*stereoscopic display system, display algorithms*; I.3.3 **[Computer Graphics]**: Picture/Image Generation--*display algorithms*; I.3.6 **[Computer Graphics]**: Methodology and Techniques--*visual communication, visual perception*; I.3.7 [**Computer Graphics]**: Three-Dimensional Graphics and Realism--*stereoscopic graphics, three-dimensional display, visual perception.*

General Terms: Projections, three-dimensional, stereoscopic.

Additional Key Words and Phrases: stereoscopic.

Simultaneous Generation of Stereoscopic Views

Stephen J. Adelson[1], Jeffrey B. Bentley[2], In Seok Chong[3], Larry F. Hodges[1] and Joseph Winograd[1]

1. College of Computing, Georgia Institute of Technology

2. Racal-Milgo, Atlanta, Georgia

3. Dept. of Electrical Engineering & Computer Science, Northwestern University

Introduction

Stereoscopic computer graphics, the generation and display of stereoscopic computer-generated images, has become an important and growing area of computer graphics. The introduction of improved stereoscopic workstations that use liquid crystal shutters and 120 Hz. refresh rates, and the usefulness of stereoscopic images in scientific visualization have boosted stereoscopic computer graphics from a curiosity to a useful tool. During the past five years the area of stereoscopic computer graphics that has received the most interest has been the hardware technology for display of stereoscopic images. Almost no attention has been paid to the development of graphics algorithms specific to the generation of the left and right-eye perspective views of a stereoscopic image. This lack of attention is peculiar in 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 much more efficient to generate the views simultaneously than to treat them as two totally separate images. In this paper we examine several algorithms to measure the computational savings of creating the views simultaneously.

Performance Measurements

Our purpose is to develop algorithms that simultaneously operate on two different images of a scene by extending standard single-image algorithms. Using standard algorithms, 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 operating on two images can be done as efficiently as one image). There are at least three different measures that we could use to present our results in this paper. If we choose as our benchmark the total time need to compute the left and right-eye view separately, then the decrease in computing time would range from 0% (total computing time = $2t$) to 50% (total computing time = t). If, however, 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 would range from 0% (total computing time for the second view = *t*) to 100% (total computing time for the second view = *0*). A third alternative is to measure the speedup, *S*, where *S* = (total time to separately compute the left and right-eye view) / (total time to simultaneously compute the left and right-eye view). Maximum possible speedup is 2; minimum possible speedup is 1.

We have chosen to measure computational savings in the text of the paper by comparing the simultaneous algorithm with the total time needed to separately compute both the left and right-eye views (maximum decrease in computing time = 50%). Table 1 gives all three measures for each algorithm we tested.

Differences and Similarities Between The Views

A standard computer graphics 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 shown in figure 1. Given a point $P = (x_p,y_p,z_p)$, the projected point is (x_s, y_s) where $x_s = x_p/(1 + z_p/d)$ and $y_s = y_p/(1 + z_p/d)$.[7] For stereoscopic images we require two differently projected views of the image. The views may be generated by using two horizontally displaced centers of projection, or by a single center of projection for which the data for the left and right-eye views are different[1,9,13]. We will base our projected views on two different centers of projection and show that this approach can also be implemented with a single center of projection and similarity transformations of the data. It has been shown elsewhere that modifying the data by rotations for use with a single center of projection produces several unpleasant artifacts including vertical parallax and image warping[2,8,14].

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 $x_{sl} = (x_p - z_pe/(2d))/(1 + z_p/d)$ and $y_{sl} = y_p/(1 + z_p/d)$. The same point projected to RCoP has projection plane coordinates $P_{sr} = (x_{sr}, y_{sr})$ where $x_{sr} = (x_p + z_pe/(2d))/(1 + z_p/d)$ and $y_{sr} = y_p/(1 + z_p/d)$. Note that the $y_{sl} = y_{sr}$ so that this value need only be computed once. Also note that the terms in x_{sr} and x_{sl} are of identical form and only differ by whether the terms in the numerator are added or subtracted. Assuming that we have already computed the left-eye view and have access to its component terms only one addition and one multiplication are necessary to compute the right-eye view. Assuming a pre-computed constant e/2, computing the left and right-eye projections of a point simultaneously

takes 3 additions and 5 multiplications. Computing the projections separately takes 4 additions and 8 multiplications. Assuming that an addition takes the same amount of CPU time as a multiplication gives us a conservative lower bound of a 33% decrease in CPU time. We also note that if a projected z-component ($z_p/(1 + z_p/d)$) is needed for hidden surface removal, it need only be computed once.

A different formulation of the x-components of the projected views may be derived as follows:

$$\begin{aligned} x_{sl} &= (x_p - z_p e/(2d))/(1 + z_p/d) \\ &= (x_p - z_p e/(2d) - e/2 + e/2)/(1 + z_p/d) \\ &= [(x_p + e/2) - (e/2 + z_p e/(2d))]/(1 + z_p/d) \\ &= (x_p + e/2)/(1 + z_p/d) - [(e/2)(1 + z_p/d)]/(1 + z_p/d) \\ &= (x_p + e/2)/(1 + z_p/d) - e/2 \, . \end{aligned}$$

Similarly, $x_{sr} = (x_p - e/2)/(1 + z_p/d) + e/2$. Note that this is our standard single view transformation in which the left-eye data is translated to the right by e/2 and the right-eye data is translated to the left by e/2 before the perspective projection. After the projection the data for each eye is translated back by the same amount. Since, after the perspective transformation, we are dealing with image space (i.e., pixels), the back-translation of each set of data can be most efficiently implemented by a pan to the left for the entire left-eye image and a pan to the right for the entire right-eye image.

Before the perspective projection we can view the left and right-eye data as differing only by a constant translation of the x-coordinate of each point. If we assume that a point in the right-eye data is $P_r = (x_p, y_p, z_p)$, then the corresponding point for the left-eye data is $P_l = (x_p\text{-}e, y_p, z_p)$. Viewing the left and right-eye data in this manner allows us to make several observations that will simplify the process of generating efficient algorithms for simultaneous generation of stereoscopic images. For example, if we assume that the equation of the plane for a polygon in the right-eye data is $P_r \prod_r^T = 0$ where $P_r = (x_p, y_p, z_p, 1)$ and $\prod_r = (A, B, C, D)$, then the equation of the plane for the corresponding polygon in the left-eye data must be $P_l \prod_l^T$ where $P_l = (x_p\text{-}e, y_p, z_p, 1)$ and $\prod_l = (A, B, C, Ae+D)$. It follows that a normal vector N to a polygon in the right-eye data is also a normal vector to the corresponding polygon in the left-eye data. These observations are particularly important to our hidden surface elimination algorithms.

Rendering Polygons

We will examine both a scan-line polygon fill and the Gouraud shading algorithm. If we begin with a polygon in three-space and project it to a left and right-eye center of projection as described in above, then the projected vertices of the polygons will share y-coordinates but have potentially different corresponding x-coordinates and, hence, different widths.

For a standard scan-line polygon fill algorithm the common y-coordinates allow the overhead associated with updating an active edge list and sorting the edges by y-coordinates to be shared if the polygons are filled simultaneously. We must keep separate data on the x-coordinates for each vertex and the change in x per scan-line. We tested a standard scan-line polygon fill algorithm for both the simultaneous and individual polygon fill on several polygons made up of from four to 32 edges. Since the fill is a relatively fast operation, the difference between these methods seems slight on polygons with few sides. However, we clearly see a 25% decrease in fill time for large (15 or more sided) polygons.

In a Gouraud illumination model we first define a color for each vertex of a polygon based on Lambert's law (in the form of a RGB triple, for example), an ambient term, and distance from a lightsource[6]. Each point on an edge connecting the vertices of the polygon has its color interpolated based on its distance from the vertices of the edge. Likewise, interior pixel colors are evaluated by interpolations from the endpoints of their scan-line, which are the edge points calculated previously. By computing the left and right-eye projected polygons simultaneously we save on the associated overhead administration of which scan-line and which edges are currently active. We also save on the number of interpolations needed to compute a color value for each pixel. Note that the color of a vertex is dependent on its spatial location and not the location of the viewer. Hence, a vertex will have the same color, if not the same position, in both the left and right-eye views. A polygon will have the same height (number of scan-lines between top and bottom of the polygon) in both views, so edge colors can the interpolated based on the relative y values of the endpoints; these colors will be the same on both polygons. Like the simple polygon fill, however, the width of the polygon will differ in each view because the x-values differ by $x_{sr} - x_{sl} = ez_p/(d+z_p)$. Therefore, we will have to interpolate interior pixels twice, once for each view.

Since we are calculating the edge colors once, but the interior pixels twice, the more edge pixels and fewer interior pixels a polygon has, the more computing time will be saved by using the stereo algorithm once rather than the original algorithm twice. The single algorithm requires four interpolations per edge pixel to render each view (in each view we must calculate x and the RGB triple) and three for each interior pixel (the RGB). While the interior pixel generation

remains the same when we apply the stereoscopic algorithm, the edge pixels require only five interpolations to compute *both* views (x for each view and an RGB value which is applicable to both views). On average, then, we have 4 interpolations per edge pixel for each view using the single algorithm and 2.5 per view for the simultaneous algorithm.

One measure of performance of this algorithm is by the decrease in the total number of interpolations, since such calculations dominate a Gouraud shaded polygon fill. As expected, greater savings appear when we have a polygon that is relatively "taller" than "wide". For example, a polygon shaped as a flat triangle containing 22 edge pixels and 211 interior pixels requires 1442 interpolations if rendered as a stereo pair by applying a standard Gouraud shading algorithm twice and 1376 interpolations using the stereo Gouraud shading algorithm. The saving in interpolations is approximately 4.5%. On the other hand, a tower-shaped polygon containing 58 edge pixels and 105 interior pixels gives a performance of 862 interpolations in the standard algorithm and 775 in the stereo version, for a savings of about 10%. In the theoretical best case, a two-pixel-wide polygon, we would save 1 - 2.5/4.0, or 37.5% on the number of interpolations required. We also get improved performance on small polygons since the ratio of edge pixels to interior pixels increases inversely to the ratio of polygon size relative to pixel size.

Line and Polygon Clipping

Data clipping to display window boundaries is an important part of generating complete images. The Liang-Barsky algorithms for line clipping and polygon clipping produce excellent results for single images, and therefore are good candidates to modify for use with stereo pairs[10,11]. We will first look at our experience with the line clipping algorithm.

Line Clipping

The Liang-Barsky line clipping algorithm considers each of the four window borders as an infinite line that divides the image space into two half-planes. Each line segment to be clipped is mapped into a parametric representation in such a way that as t varies from 0 to 1 the line segment is traced out by separate parametric equations for x and y. The intersection of the parametric equations with each window border indicates where the line segment lies in relation to that boundary.

A systematic analysis of each of those intersections yields an algorithm in which the t values of the segment are clipped successively against each of the four borders. Each intersection yields either a trivial rejection of the line segment (the entire segment lies outside the border), no

change to the line segment (the entire segment lies inside the border), or a successively shorter portion of the line segment (namely, that portion of the segment which lies inside the border), as shown in figure 3. If the four clips are completed with no trivial rejection of the segment, the remaining interval of the parameter t lies inside the clip window, and the original line's coordinates can be changed to match that interval.

To optimize this algorithm for stereoscopic images, we consider the parametric equation for y. As noted in previously, stereo pairs have the property that the y value of a projected point will be identical in both the left and right-eye views. We therefore know that the y coordinates of the endpoints of a line will be identical and, hence, the parametric equation for y values will be identical for both views. Clipping the line in either view against the top and bottom borders will yield the same segments for both views. Unfortunately, this similarity does not hold for the x coordinate, so we must still test the remaining borders (left and right) independently for each view.

We wrote a clipping procedure based on the Liang-Barsky algorithm, and because of stereo optimization, we were able to reduce our invocations from eight to six. Testing with randomized data confirmed the expected 25% improvement in computational time.

Polygon Clipping

The Liang-Barsky polygon clipping algorithm is similar to the line clipping algorithm in that most of the functional computations are performed on parametric functions. For each line segment of a polygon, the parametric equations of the extended line segment are intersected with each of the window borders. The algorithm differs from line clipping fundamentally, however. It computes the intersection of the line segments with each border as a first task of the procedure, and then sorts the t values of the four intersection points to determine the path that the extended line takes through the entire window region. The order in which it enters and exits the semi-planes defined by each window border determines whether to clip the segment, leave it as it is, or alter the course of the output polygon to follow the window borders.

The initial values of the t intersection of the left and right-eye views with top and bottom borders can be performed simultaneously by following the same logic put forth in the optimization of the line clipping algorithm. Those mathematical similarities disappear, however, when the t values of the intersection points are sorted, because two images with the same values for the top and bottom intersections (but different values for the left and right intersections) could yield a different sequence of intersections for each of the two views. This possibility is illustrated in

figure 4, in which two different projections of the same line intersect the window borders differently. The left segment intersects the bottom border (t int 1) and then the top border (t int 2). This case would result in a trivial clip of the segment. The right line, in contrast, intersects the bottom border (t int 1a) and then the left window border (t int 2a). The right-eye view is not a rejection case, and is treated as an entirely different case by the polygon clipping algorithm.

The occurrence of this kind of difference between the two views and the order of their intersections with the window borders necessitates the computation of separate intersection sequences for the two segments, and rules out the simultaneous operation of the clipping process for the majority of the algorithm.

It appears that the room for optimization of this algorithm, excepting several computations at the front end, is minimal. The timing of the execution of the modified version of the algorithm showed a negligible (less than 1%) decrease in execution time as compared to computing each view separately.

Hidden Surface Elimination

Hidden surface elimination has received much attention in the graphics community. The effective identification and removal of the parts of an image that are not visible from the viewing position is considered important for efficient computation of realistic images. Numerous algorithms exist for application to single images, but we are aware of only one published account of a hidden surface algorithm optimized for use with stereoscopic images[12]. We examined two possible methods for hidden surface elimination, a combination BSP tree / Z-buffer algorithm, and a scan-line algorithm. We also developed a stereoscopic backface removal algorithm that is used as a preprocessing step for each approach.

Stereoscopic Back-Face Removal

The first step in our hidden surface elimination algorithm is to remove back-faces of objects, those sides which cannot be seen from the viewing position because of the three-dimensional orientation an object. There are several different variations of the back-face removal algorithm. The back-face removal algorithm in Hearn & Baker specifies how to identify the back-faces of a convex object based on the planar equation of a polygon and the viewing position[7]. If we assume a convex object with the equation of the plane $Ax + By + Cz + D = 0$ calculated so that the normal vector, $N = [A, B, C]$, to the polygon points out of its front-face, then a simple object space data method for identifying the back-faces is to calculate $Ax_V + By_V + Cy_V + D_V$ where

(x_v, y_v, z_v) is the viewing position. Any polygon plane for which $Ax_v + By_v + Cy_v + D_v < 0$ must be a back-face and should not be rendered.

Since the determination of a back-face depends on the viewing position, we must consider each polygon relative to both the left and right-eye viewing positions. Equivalently we may consider one viewing position relative to the left-eye and right-eye data. The possibilities are:

1. The polygon is a back-face relative to both viewing positions.
2. The polygon is a front-face relative to both viewing positions.
3. The polygon is a back-face relative to the right-eye viewing position and a front-face relative to the left-eye viewing position.
4. The polygon is a front-face relative to the right-eye viewing position and a back-face relative to the left-eye viewing position.

Assume that any point, P_l, in a left-eye view has coordinates P_r - (e,0,0). Then, given the viewing position (x_v, y_v, z_v), the stereoscopic backface removal algorithm is:

```
If Ae > 0 Then Begin
    If Axv+Byv+Cyv+Dv+Ae < 0 Then
      the polygon is a back-face for both viewing positions.
    Else If Axv+Byv+Cyv+Dv < 0 Then
      the polygon is a back-face for the right-eye viewing position
    Else
      the polygon is not a back-face for either viewing position
    End
Else Begin
    If Axv+Byv+Cyv+Dv +Ae > 0 Then
      the polygon is not a back-face for either viewing position
    Else If Axv+Byv+Cyv+Dv > 0 Then
      the polygon is a back-face for the left-eye viewing position
    Else
      the polygon is a back-face for both viewing positions.
    End
```

If we consider an operation count then it takes 2 x {3 multiplications + 3 additions + 1 comparison} = 6 multiplications + 6 additions + 2 comparisons to determine backfaces if we

compute the images separately. In the worst case it takes 4 multiplications + 4 additions + 3 comparisons for the simultaneous determination of backfaces. The improvement in running time is approximately 33%.

Combining Z-Buffers and BSP-trees

One possible method for producing stereo images with hidden surface elimination is to use two z-buffers[3], one containing the depth data for each view. While many computer graphics workstations today have a hardware z-buffer, we are not aware of any systems that provide two z-buffers. Implementing two software z-buffers often exceeds available fixed primary storage or causes excessive paging (with resulting performance degradation) with virtual memory.

Binary Space Partition-trees (BSP-trees) suggest another option for stereo pair generation[5]. A BSP-tree orders planar objects in a scene by creating a binary tree. Each node in the tree represents one object whose plane divides its subtrees' spaces; all objects in one subtree are from the same side of the root object's plane. Objects that are split by the plane of the root are replaced by two new objects and one is placed in each subtree. The root of such a tree and which branch of the subtree corresponds to which half-plane may be chosen randomly, although heuristics exist for choosing the best root. We traverse a BSP-tree in a manner such that we visit nodes corresponding to the farthest objects from the viewing position first by traversing the subtree containing the farther objects and work forward to the nearest object.

We could use two passes through a BSP-tree of the data to create the two views. Creating both views simultaneously with one pass through the BSP-tree is not possible since the traversal through the tree might well be different for different eyepoints. As shown in figure 5, given three polygons with A chosen as the root, a BSP-tree traversal for the right-eye view would be B-A-C while a left-eye view traversal would be C-A-B. As a compromise, we decided to create a method combining the z-buffer with a BSP tree, requiring only one z-buffer and one pass through the BSP tree. To be more precise, during a traversal of the BSP-tree relative to the right-eye position to render the right-eye image polygons front-to-back we simultaneously render the left-eye view using the z-buffer.

Creating a BSP-tree usually increases the original number of polygons by less than a factor of two, so the increase in the number of objects to be rendered is not significant[5]. It is true, however, that a proper BSP-tree traversal, back to front relative to the right-eye view, is potentially the worst possible z-buffer traversal for the other view. Nonetheless, we find significant savings by using this combination. On scenes containing from 144 to 960 polygons,

we found a consistent savings of 17% as compared to two passes through a BSP-tree. While it is certain that our algorithm is slower than using two hardware z-buffers, memory constraints make this an infeasible alternative to the BSP-tree/z-buffer combination.

Scan-line algorithms

Another method for implementing hidden surface elimination involves using scan-line algorithms, a summary of which appears in Sutherland[15]. As described in previously, we need only calculate the y and z coordinates of a point once for both views which provides a significant savings in number of computations. The algorithm makes a list of all polygon edges sorted by their y value; examines each scan-line, creating a list of the active (intersecting) edges ordered by y value; calculates the intersections of the active edges with the scan-line and orders them by their x value; and renders the closest intersecting face for each pixel of the scan-line.

One problem with this method is that we cannot take advantage of stereo attributes beyond the list of active sides, which is sorted by y value. Again we encounter the difficulty that we must project the x-intersections for the scan-line intersections, which of course will differ between views. Our savings for a stereoscopic algorithm, then, arises exclusively from the creation of one list of y-sorted edges for both views and the simultaneous projection of the y and z coordinates. On the other hand, since we are required to calculate scan-line/edge intersections and we also know that the color of a point does not depend on its projected position, we can implement Gouraud shading with little additional overhead.

We have found that the savings provided by the scan-line algorithm is on the order of 12%, somewhat smaller than the BSP/Z-buffer for the reasons cited above. However, our current implementation model does not include provisions for penetration of sides or any of the many kinds of coherence, for example those described by Crocker or Sutherland[4,15]. We feel that with a more complex model the savings would be even more significant. Additionally, this figure does not take into account the savings from the improved Gouraud shading. If we also add this efficiency, the overall improvement is approximately 17%.

Performance of a Simple Stereo Renderer

To test the combined performance of the simultaneous versions of the algorithms described in the previous sections we compared a simple mono rendering program with a corresponding stereo image rendering program. Each version implemented perspective projection, scan-line hidden surface removal and a Gouraud illumination model. Stereoscopic images were generated of a village scene containing eight buildings comprising approximately 1100 polygons. The

improvement in rendering time of the stereoscopic image by the stereo renderer compared to the mono rendering program was 18.8%.

Table 1 gives the specific ranges of improvement for each algorithm. We also tested a simple rendering program that combined several of the algorithms for a 18.8% overall improvement when rendering our village scene. It is important to note that the optimizations of the algorithms never hindered computational time, a fact that implies that even the above operations that did not perform up to expectations might well improve if a different algorithm is used.

Conclusions

We have examined several rendering algorithms and modified them to simultaneously generate the left and right-eye views of a stereoscopic image. When compared to generation of the left and right-eye views as if they were two totally separate images, we obtained decreases in rendering times of from <1% to 37.5%. Percent improvement depended on both the data and the algorithms used.

Acknowledgments

All algorithms in this paper were implemented in C under Unix™ on a SGI 4D superworkstation. This work was supported by the President's Fellowship Program and the Georgia Tech Computer Graphics / Scientific Visualization Laboratory.

References

1. Baker, J. "Generating Images for a Time-Multiplexed Stereoscopic Computer Graphics System," SPIE Proc. 761: True 3D Imaging Techniques and Display Technologies (January 1987), 44-52.

2. Butts, D.R.W. and McAllister, D.F., "Implementation of True 3-D Cursors in Computer Graphics," SPIE Proc. 902: Three-Dimensional Imaging and Remote Sensing Imaging (January 1988), 74-84.

3. Catmull, E., "Computer Display of Curved Surfaces," Procedures of the IEEE Conference on Computer Graphics Pattern Recognition and Data Structures, (May 1975), 11.

4. Crocker, G.A., "Invisibility Coherence for Faster Scan-line Hidden Surface Algorithms," ACM Computer Graphics, 17, 3 (July 1983) 73-82.

5. Fuchs, H., Abram, G.D., Grant, E.D., "Near Real-Time Shaded Display of Rigid Objects," ACM Computer Graphics, 17, 3 (July 1983), 65-72.

6. Gouraud, H., "Continuous Shading of Curved Surfaces," IEEE Transactions on Computers, C-20, 6 (June 1971), 623-629.

7. Hearn, D. and Baker, M.P. Computer Graphics, Prentice Hall, Inc., Englewood Cliffs, New Jersey (1986), 261-262.

8. Hodges, L.F. and McAllister, D.F. "Rotation Algorithm Artifacts in Stereoscopic Images. Optical Engineering 29,8 (August 1990), 973-976.

9. Hodges, L.F., Johnson, P. and DeHoff, R.J., "Stereoscopic Computer Graphics," J. of Theoretical Graphics and Computing, 1,1 (December 1988), 1-12.

10. Liang, Y.D. and Barsky, B.A., "An Analysis and Algorithm for Polygon Clipping," Communications of the ACM, 26, 11 (November 1983), 868-877.

11. Liang, Y.D. and Barsky, B.A., "A New Concept and Method for Line Clipping," ACM Transactions on Graphics, 3, 1 (January 1984), 1-22.

12. Papathomas, T.V., Schiavone, J.A., and Julesz, B.,"Stereo Animation for Very Large Data Bases: Case Study - Meteorology," IEEE CG&A, 7, 9 (September 1987), 18-27.

13. Sandin, D.J., et al., "Computer-Generated Barrier-Strip Autostereogram," SPIE Proc. 1083: Non-Holographic True 3D Display Technology (January 1989), 65-75.

14. Saunders, B.G., "Stereoscopic Drawing by Computer--Is It Orthoscopic?" Applied Optics 7,8 (August 1968), 1499-1503.

15. Sutherland, L.E., Sproull, R.F., and Schumacker, R.A., "A Characterization of Ten Hidden Surface Algorithms," ACM Computing Surveys, 6, 1 (March 1974), 1-55.

Table 1. Percent savings in rendering time for simultaneous generation of the left and right-eye views of stereoscopic images when compared to separate generation of the left and right-eye views for various algorithms.

	Savings by Measuring Method		
Algorithm	***I***	***II***	***III***
Perspective Projection	33%	67%	1.67
Flat Shaded Polygon	<1% - 25%	<1% - 50%	1 - 1.5
Gouraud Shaded Polygon	4.5% - 37.5%	9% - 75%	1.1 - 1.75
Liang-Barsky Line Clipping	25%	50%	1.5
Liang-Barsky Polygon Clipping	<1%	<1%	1
Backface Removal	33%	67%	1.67
BSP/Z-Buffer compared to BSP	17%	33%	1.33
Scan-Line Hidden Surface	12%	24%	1.24
Simple Renderer (Village Scene)	1.38	18.8%	37.6%

Measure I: Measure is the total time need to compute the left and right-eye view separately. The possible decrease in computing time ranges from 0% (no decrease) to 50% (two views as efficiently as one view).

Measure II: Measure is the decrease in time required to compute the second eye view as compared to the time needed to compute the first eye view. The decrease in computing time for the second view would range from 0% (no decrease) to 100% (two views as efficiently as one view).

Measure III: A third alternative is to measure the speedup, S, where S = (total time to separately compute the left and right-eye view) / (total time to simultaneously compute the left and right-eye view). Maximum possible speedup is 2 (two views as efficiently as one view); minimum possible speedup is 1 (no speedup).

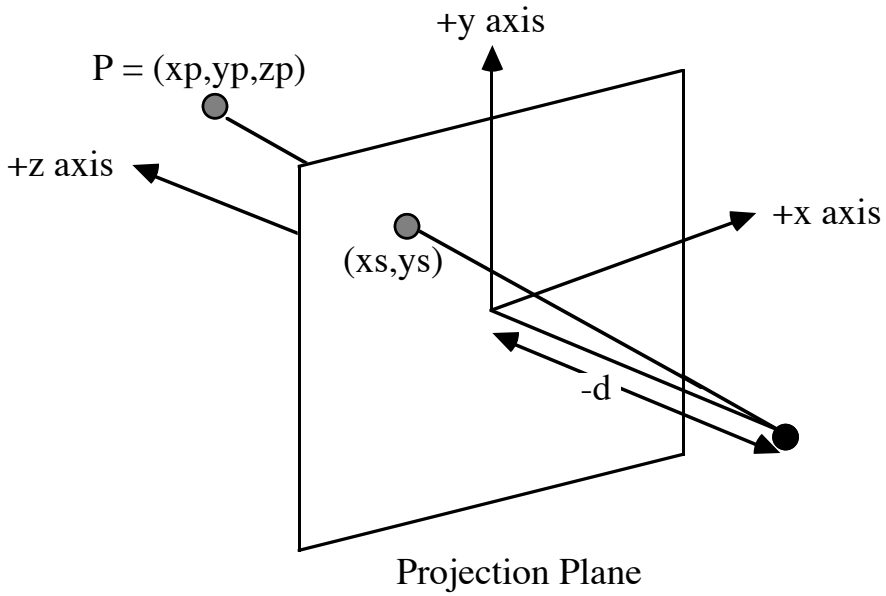

Figure 1.
Standard Perspective Projection

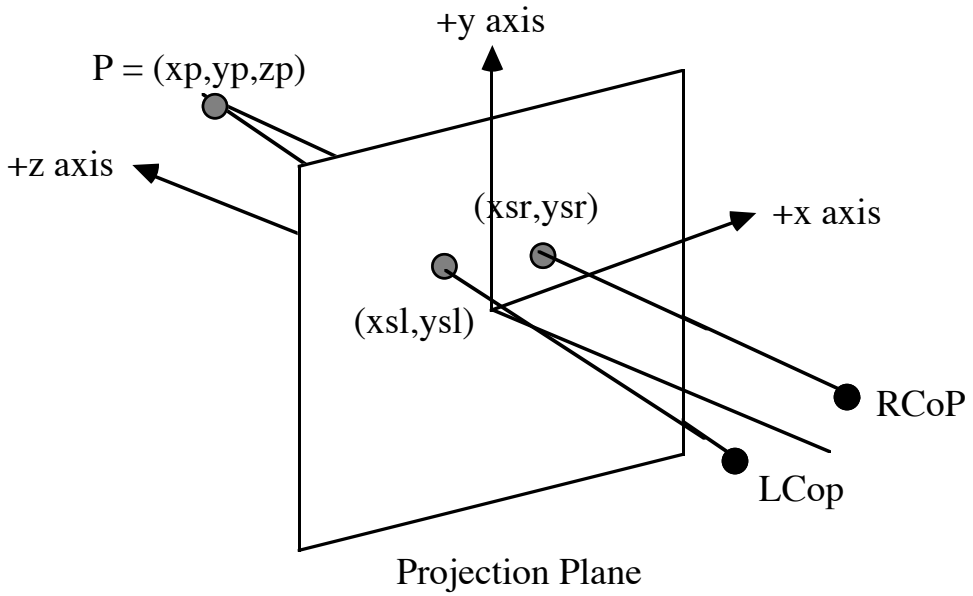

Figure 2.
Stereoscopic Perspective Projection

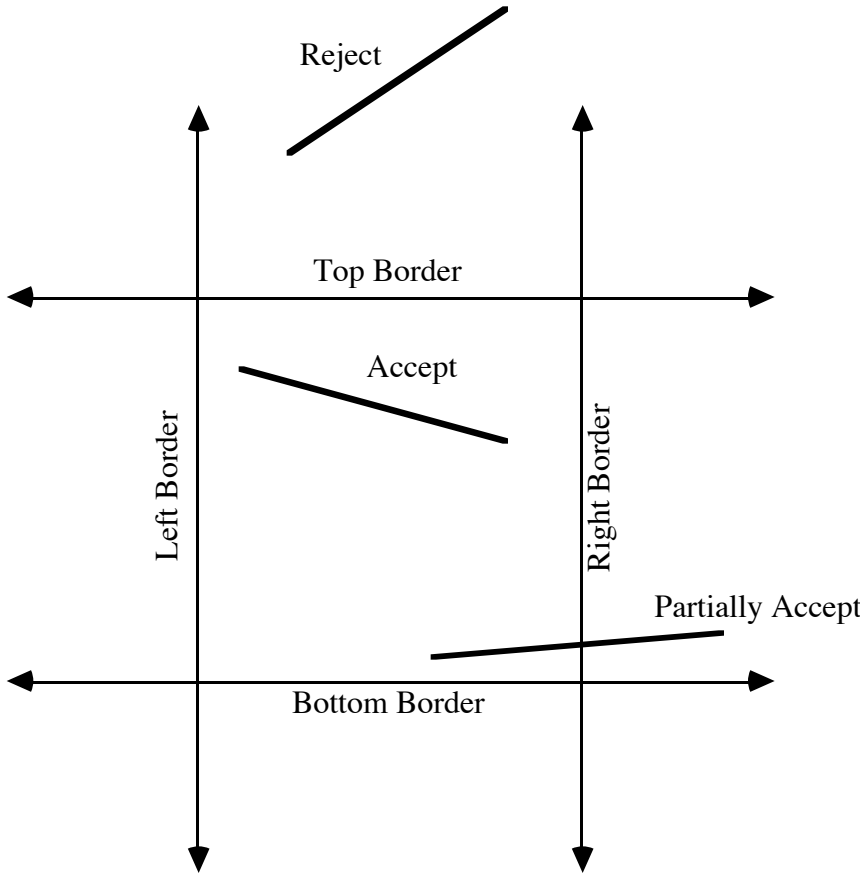

Figure 3.
Possible Operations on a Line Segment by Liang-Barsky Algorithm

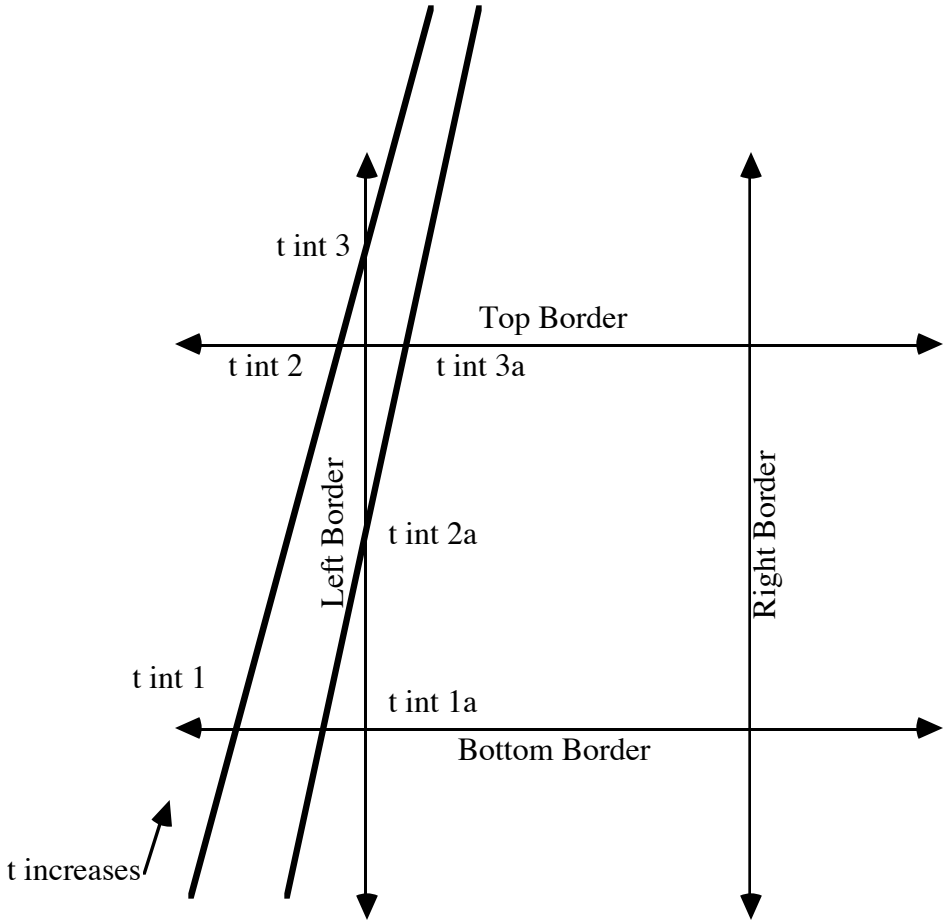

Figure 4.
Two Extended Lines & Their Parametric Intersections With The Window Borders.

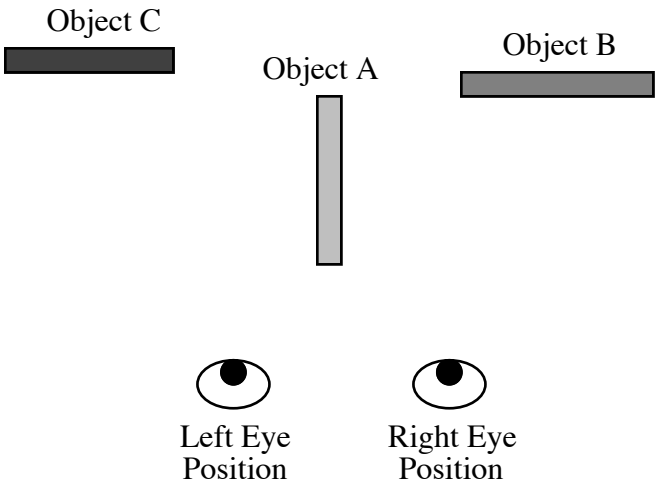

Figure 5.
Right Eye Traversal of a BSP Tree With Root A is C-A-B. Left Eye Traversal of a BSP Tree With Root A is B-A-C.

Captions for Figures

Figure 1.
Standard Perspective Projection

Figure 2.
Stereoscopic Perspective Projection

Figure 3.
Possible Operations on a Line Segment by Liang-Barsky Algorithm

Figure 4.
Two Extended Lines & Their Parametric Intersections With The Window Borders.

Figure 5.
Right Eye Traversal of a BSP Tree With Root A is C-A-B. Left Eye Traversal of a BSP Tree With Root A is B-A-C.