1986

A One Step collision Detection Method for Computer Graphics Programs

Gregory T. Eichner
University of Central Florida

A ONE STEP COLLISION DETECTION METHOD FOR COMPUTER GRAPHICS PROGRAMS

BY

GREGORY THOMAS EICHNER
B.S.E., University of Central Florida, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in Engineering
in the Graduate Studies Program of the College of Engineering
University of Central Florida
Orlando, Florida

Fall Term
1986

ABSTRACT

The topic of this thesis is a collision detection algorithm for use in computer programs dealing in three-dimensional graphics. Collision detection is usually accomplished by breaking the movement into small steps and checking if a collision has occurred at each of these discrete steps. This method is a very time-intensive way to detect for collisions and therefore, inefficient for a system which typically moves in large increments. For this kind of system, a method could be developed which checks once for collisions without dividing the move into multiple small increments. The subject of this paper is an algorithm, developed for use in a computer program, that will allow the user to make large movements of the objects and check for collisions quickly and efficiently.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

When a manufacturer purchases a robot arm for his business, he needs to set up the movement pattern for the robot to follow. Because of the risk of possible damage to the robot arm and its surroundings, it is usually not desirable to perform test movements with the robot arm. A computer program can be developed which will be able to display the arm and its surroundings. The manufacturer would then use the program to perform a series of test movements. When the test movements are satisfactory for the application, the computer will send the indicated movements to the robot arm. The movement pattern would then be fine-tuned on the actual arm with a greatly reduced chance of damaging the arm or hitting anything in its surroundings.

Since the movements of the robot arm are usually in relatively large increments, it should not be necessary to divide the movement into many small steps and check for collisions at each step. A more desirable system would check for any collisions along the path of the movement in one step. The system should flag any collisions and not allow these moves. It should allow the user to store the moves

performed and edit the steps as desired. The system must also allow the input, display and editing of the robot arm description and the description of its surroundings. The system should operate with a minimum of user knowledge of computers and a maximum of user friendliness. It should be able to display the scene so that the user has a visual image of the current positions of all objects. This enables the user to easily determine the next movement to be performed.

The topic of this paper is a computer program which performs the test movements of the robot arm within its surroundings. The paper will discuss some general methods of defining three-dimensional images and performing object rotations and translations. The main discussion in this paper will be a method of detecting collisions between the moving part of the robot arm and its environment. One way in which this can be done is to move the objects in small, discrete steps. It is then determined at each step if a collision has occurred. This method of collision detection is easy to perform but very time intensive. Therefore another method of collision detection will be developed. This system will be more efficient when rotations are usually done in large increments. The system will detect for collisions once during each rotation. This means that the collision detection would take the same amount of time

whether the movement is very small or very large.

In this paper, only the case of rotating objects will be discussed. Translating objects, objects moving in straight lines, will not be considered in the following discussions. However, it should be noted that the same theory applied to rotating objects here can be applied to translating objects also with very little modification.

DATA STORAGE METHODS

In three-dimensional graphics applications there are two main ways to store data: the Point-Line method and the Shape-Location method, as presented in Interactive Computer Graphics Data Structures, Algorithms, and Languages. The method used most often in new applications is the Shape-Location method. This is because it is a more natural way for a human to define an object and because it usually takes less memory to define relatively simple objects. A brief discussion of the two methods follows, describing why the older Point-Line method was used.

The Shape-Location definition of an object consists a list of shape identifications, boolean operators, location indicators, and size multipliers. The shape identifier indicates which shape primitive is to be used (sphere, cube, octohedron, etc.). The boolean operator tells whether this primitive is to be added to or subtracted from the current shape to obtain the desired figure. The location indicator tells where, in the figure, the center of the shape is to be placed. The size multiplier tells how big the primitive is (the circle's radius, the length of one side on the cube,

etc.). This creates a very easy-to-use figure definition scheme. For example, an open box would consist of two cube primitives. One cube, the base figure of one unit by one unit in size, with a second cube, a little bit smaller on each side, taken out of it (boolean operator = not). This means that only two shape indicators, two boolean operators, two location indicators, and two size multipliers need to be stored.

The Point-Line definition of an object consists of the list of points, lines, and sides. The list of points are all of the points on the figure which are needed to describe the object in three-dimensional space. The list of lines indicate which points are connected together with straight lines. The list of sides show which lines are the edge of a side of the object. Although this method makes it harder to create objects, it provides a much more precise definition of the objects, than if the data itself were viewed. Using the same example as above: the open box would require eight point definitions, twelve line definitions, and five plane definitions.

The Point-Line method is much more time consuming, and more memory consuming than the Shape-Location method. It is more difficult to define objects using the Point-Line method.

An advantage to the Point-Line method is that the definition of the entire figure is always known. In order to facilitate the collision detection, which is the thrust of this thesis, the Point-Line method of object definition is used. The ability to know where each and every line in the figure is without having to generate the line made the collision detection routine much simpler.

Point-Line Method

Points

1	(-1, 1, 1)	5	(-1,-1, 1)
2	(-1, 1,-1)	6	(-1,-1,-1)
3	(1, 1,-1)	7	(1,-1,-1)
4	(1, 1, 1)	8	(1,-1, 1)

Lines

1	Point 1 -> 2	7	Point 2 -> 6
2	Point 2 -> 3	8	Point 6 -> 7
3	Point 3 -> 4	9	Point 3 -> 7
4	Point 4 -> 1	10	Point 7 -> 8
5	Point 1 -> 5	11	Point 4 -> 8
6	Point 5 -> 6	12	Point 8 -> 5

Sides

1	Lines 1,5,6,7	4	Lines 4,11,12,5
2	Lines 2,7,8,9	5	Lines 6,8,10,12
3	Lines 3,9,10,11		

Shape-Location Method

Shape	Operator	Location	Size
Cube	and	(0,0,0)	2
Pillar	not	(0,0,0.05)	1.9,1.95

This shows the amount of data needed to describe an open box using both methods.

Figure 1. Comparison of different data storage methods.

MATRIX MANIPULATIONS

Any point in three-dimensional space can be represented by the X, Y, and Z coordinates of the point. The coordinates of the point can be modified, using standard geometric identities, so that a translation or rotation of the point with respect to one of the primary axis' can be simulated. If the point is represented as a 1x3 array, then the modification needed to perform the transformation is a simple matrix multiplication with the second matrix defined as shown in the following figures. The derivation of these matrices can be found in any book, such as Mathematical Elements for Computer Graphics and Fundamentals of Interactive Computer Graphics containing the basics for three-dimensional graphics.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix}$$

Figure 2. Transformation matrix for rotations about the X axis.

$$\begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}$$

Figure 3. Transformation matrix for rotations about the Y axis.

$$\begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 4. Transformation matrix for rotations about the Z axis.

These matrices are given in the non-homogeneous form. The homogeneous form of these matrices is 4x4. Elements (1,1) through (3,3) are the same as in the non-homogeneous coordinates. The extra elements in the homogeneous coordinates are all zeros except for the (4,4) element, which is one. To use the homogeneous coordinate system, the points must be represented as having four coordinates [X Y Z 1]. When the matrices are then multiplied the result will be a [X Y Z 1] for each point. The last element is a scaling factor. Since the scaling factor here will always be one, it can be ignored.

ROTATION ABOUT AN ARBITRARY AXIS

Given the matrices in the previous section it is possible to determine a single matrix which will result in the equivalent transformation as a rotation about an arbitrary axis. In order to perform such a rotation about some arbitrary axis in space, the following steps must be executed:

STEP 1: Translate one end of the axis to the origin of the primary coordinate system [0 0 0].

STEP 2: Rotate the axis to a position where it rests exactly on the X-axis.

STEP 3: Rotate the point around the X-axis the given number of degrees.

STEP 4: Reverse step 2, rotate the axis back to its relative angle with the X-axis.

STEP 5: Reverse step 1, translate the end of the axis back to its original position.

These steps are discussed in detail in Appendix B.

COLLISION DETECTION

Collision detection is the process of determining whether two objects, at least one of which is in motion, will collide with one another. There are multiple ways to detect for collisions. The simplest way is to move one of the objects in very small steps, stopping at each to check for a possible collision. This method is easy to program and works well when the system moves in multiple small increments (e.g., any real time graphics application which requires animation).

This method becomes very time consuming when the objects in the system move in large increments and the system is not real time. It seemed to the author that the collision detection could be done once for each movement of an object. The movement would not have to be broken into many small steps and therefore would not recursively call a collision checking routine many times per rotation. This means that the collision detection would take the same amount of time whether the movement is very small or very large.

The purpose of the test program is to test the movement of a robot arm in its environment. The motion performed by a

robot arm is almost always a rotation about the axis for the moving joint. The following discussion only considers objects rotating about an arbitrary axis. The idea of checking for collisions once per movement will also work on translating objects, however that will not be discussed in this paper.

When an object is rotated around any arbitrary axis, the axis of rotation, it describes a volume in space through which it travels (it will be assumed that only one object can move at a time). If the volume described, the volume of rotation, contains any piece of a non-moving object, then the objects will collide when the rotation is actually performed so the rotation is considered illegal. In order to check for collisions between objects in one step, the volume must be completely defined. Each non-moving object can then be checked to see if any part of it intrudes into the volume of rotation.

A computer deals in numerical values and equations. The equations which describe the volume of rotation can become extremely complicated. Therefore, the rotating object must be divided into simpler pieces. The simpler pieces used in this collision detection scheme are lines. Each line in the rotating object describes a surface as it rotates about the axis of rotation. These volumes have simpler equations

which can easily be determined by the computer. Any line rotating about another line will create a conic section (e.g., a right cone). There are four possible conic sections which can be created in this situation: a plane (plane, Figure 5), a right circular cylinder (cylinder, Figure 6), a right circular cone (cone, Figure 7), and a hyperboloid of one sheet (conic, Figure 8).

In order to simplify all further discussion, it is assumed that the axis of rotation is always the X-Axis. This can be simulated by finding the matrix which will rotate the Axis of Rotation to the X-Axis and rotating all points by this matrix before they are used in any calculations.

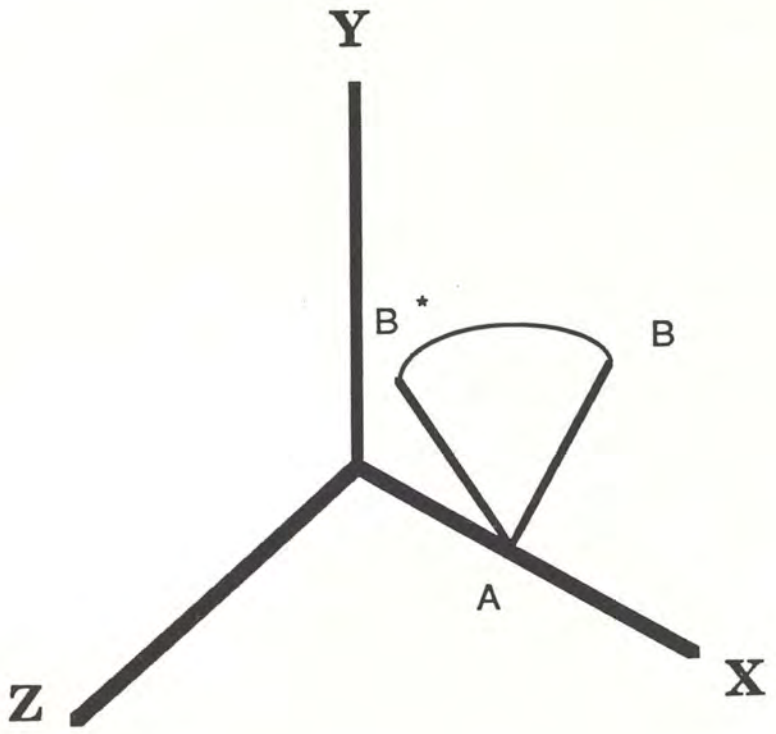

Figure 5. Rotation of a line creating a plane.

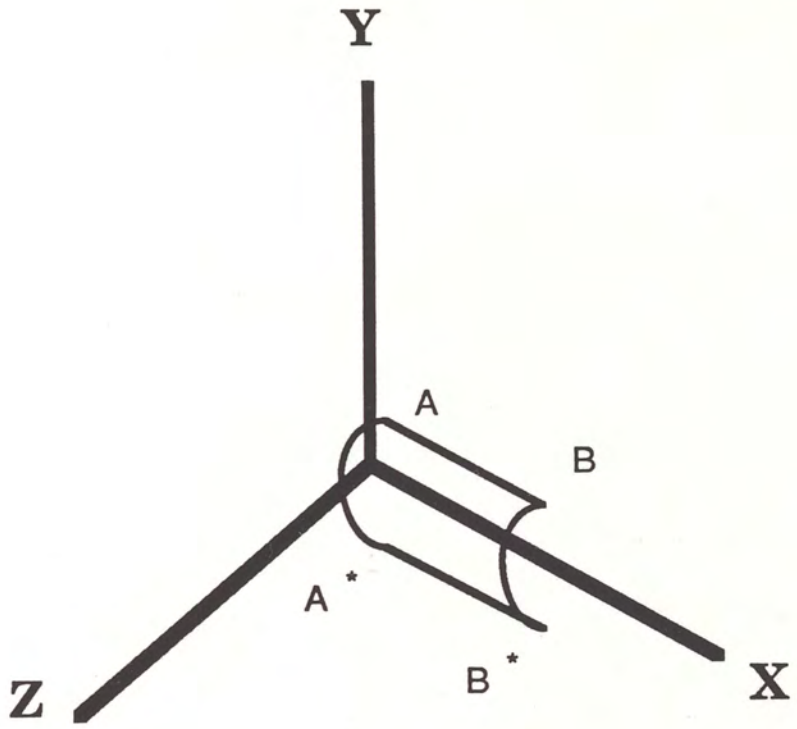

Figure 6. Rotation of line creating a cylinder.

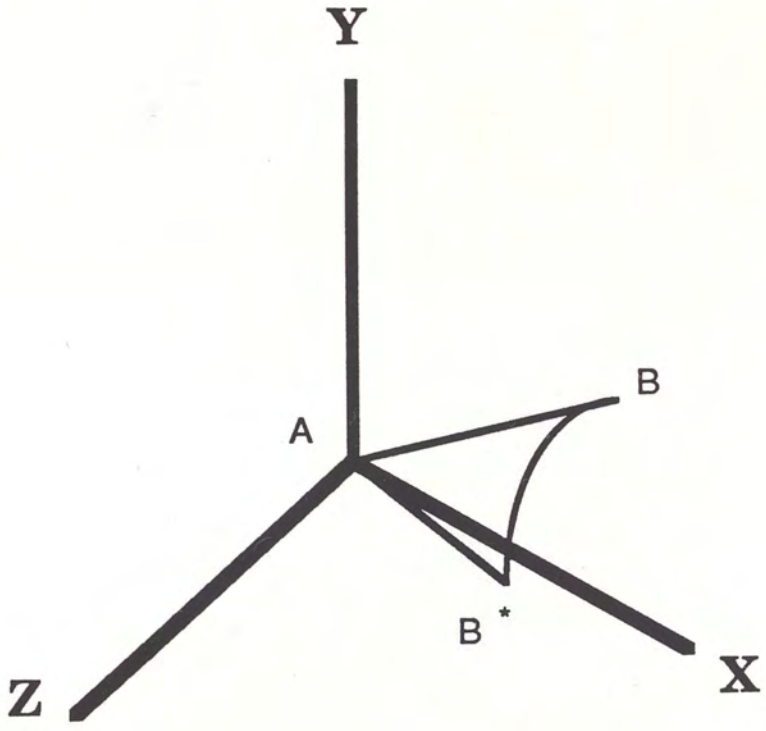

Figure 7. Rotation of a line creating a cone.

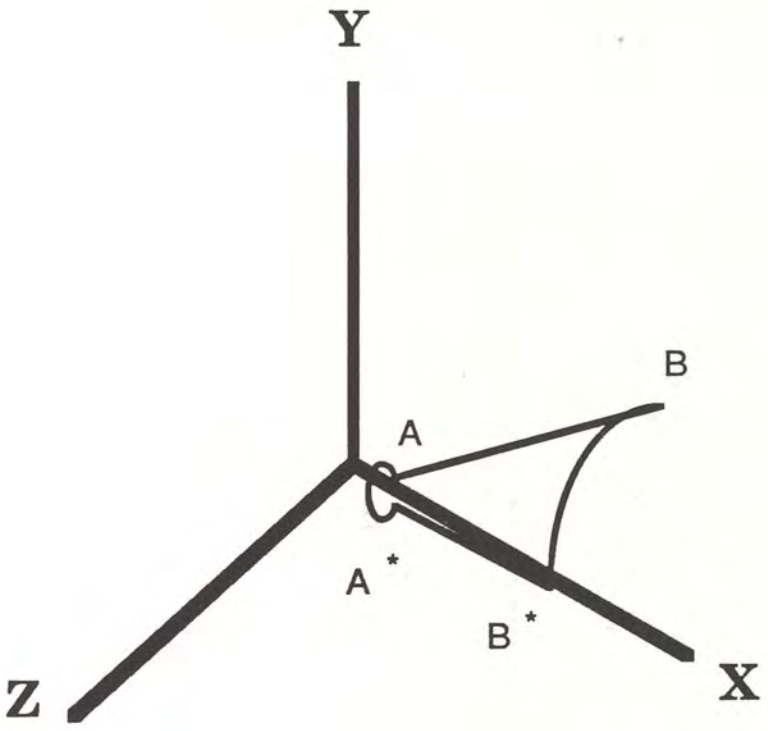

Figure 8. Rotation of line creating a conic.

A plane is the conic section generated when the line being rotated and the Axis of Rotation are perpendicular to each other. The general equation of a plane oriented along the X-Axis, as seen in Calculus and Analytic Geometry is:

$$b*Y + c*Z = d$$

Where b and c are constants and d is the X intercept.

Every line in all of the non-moving objects can then be checked to see if it intrudes into the plane so determined. If any line intrudes all further checking can be skipped because the rotation cannot be performed without a collision.

The edges of the plane generated are the unrotated line, the rotated line, an outside arc, and an inside arc. The arcs are pieces of circles. The equation for a circle in three dimensions is difficult to use in the rest of the algorithm. Therefore, the inside and outside edges of the plane are treated as part of a sphere.

Each non-moving line is checked to see if it pierces the outer sphere. If the line does not pierce the sphere, then no interference is possible. If it does pierce the sphere, then the Critical Point must be found. The Critical Point is the point where the line pierces the plane (Figure 9). The Critical Point must be within the limits of the line. If it is not within the limits, then no interference is possible. If it is within the limits, then interference is only possible if the Critical Point is within the limits of the plane.

Next, the Critical Line must be found. The Critical Line is the shortest line from the Critical Point to the X-Axis (because this is the center if the sphere, Figure 10). Given this line, interference will only occur if this line crosses the inner sphere's arc in the plane. If the line does not cross the inner sphere's arc then no iterference occurs. The Critical Point referred to in the pseudo-code is the point where the Critical Line crosses the outer sphere.

The final step in the collision detection for this line is to find the Chord (Figure 10). The Chord is the straight line connecting the two ends of the arc of the inner sphere. If the Chord and the Critical Line cross then collision only occurs if the angle of rotation is less than 180 degrees. If the Chord and the Critical Line do not cross then collision only occurs if the angle of rotation is more than 180 degrees. If the angle of rotation is exactly 180 degrees then the rotation must be broken up into two 90 degree steps. This is because the chord will be identical to the diameter of the circle and any Critical Line will always touch the center point of the circle and ,therfore, the Critical Line which indicates interference.

Appendix C shows the pseudo-code used for writing the sample program created with this paper.

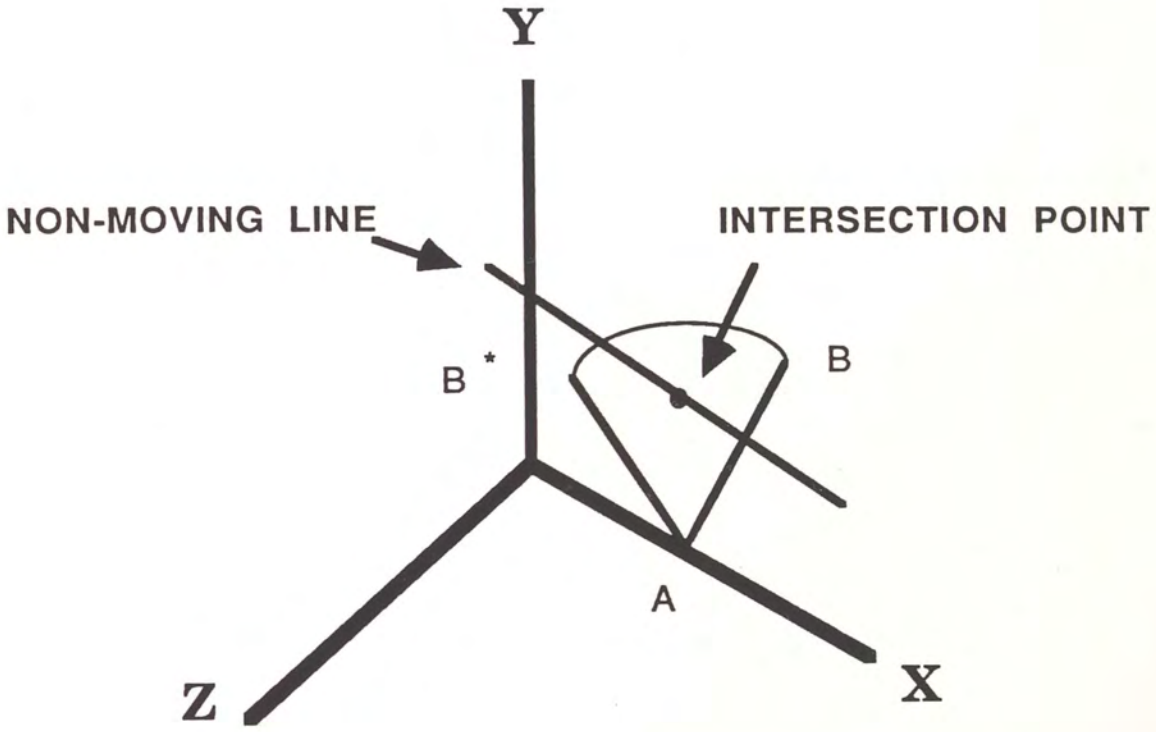

Figure 9. Critical Point in a plane.

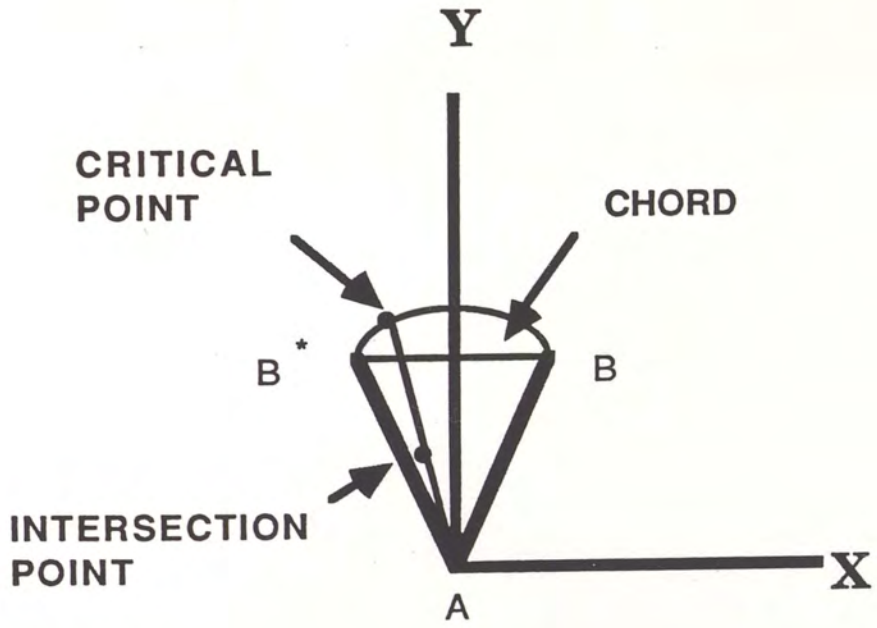

Figure 10. Chord - Critical Line check.

A cylinder is the conic section generated when a line parallel to the Axis of Rotation is rotated about it. The general equation of a cylinder oriented along the X-Axis, as seen in Calculus and Analytic Geometry is:

$$Y*Y + Z*Z = R*R$$

Where R is the radius of the cylinder.

Every line in all of the non-moving objects can then be checked to see if it intrudes into the cylinder so determined. If any line intrudes all further checking can be skipped because the rotation cannot be performed without a collision.

First the direction of the non-moving line is checked against that of the Axis of Rotation. If they are parallel, then the line can only intrude into the cylinder if it is on the edge of the cylinder (is the same distance from the axis as the moving line). If the line is on the edge of the cylinder, then the Critical Point must be found. This is the Point where the line first intrudes on the cylinder (Figure 11). The Critical Point must be within the limits of the line. If it is not within the limits, then no interference is possible. If it is within the limits, then interference is only possible if the Critical Point is within the limits of the cylinder. If the Critical Point does not exist, then

no collision occurs for this line.

If the line is not parallel to the Axis of Rotation, then the line can pierce the cylinder in up to two places. First, find the points where the line intersects the cylinder (the Intersection Point(s), Figure 11). If they do not intersect or the Intersection Point(s) are past either end of the line, then no collision occurs for this line. If the Intersection Point(s) are not within the limits of the cylinder then no collision occurs for this line. Otherwise, any Intersection Point within the limits of the cylinder becomes a Critical Point.

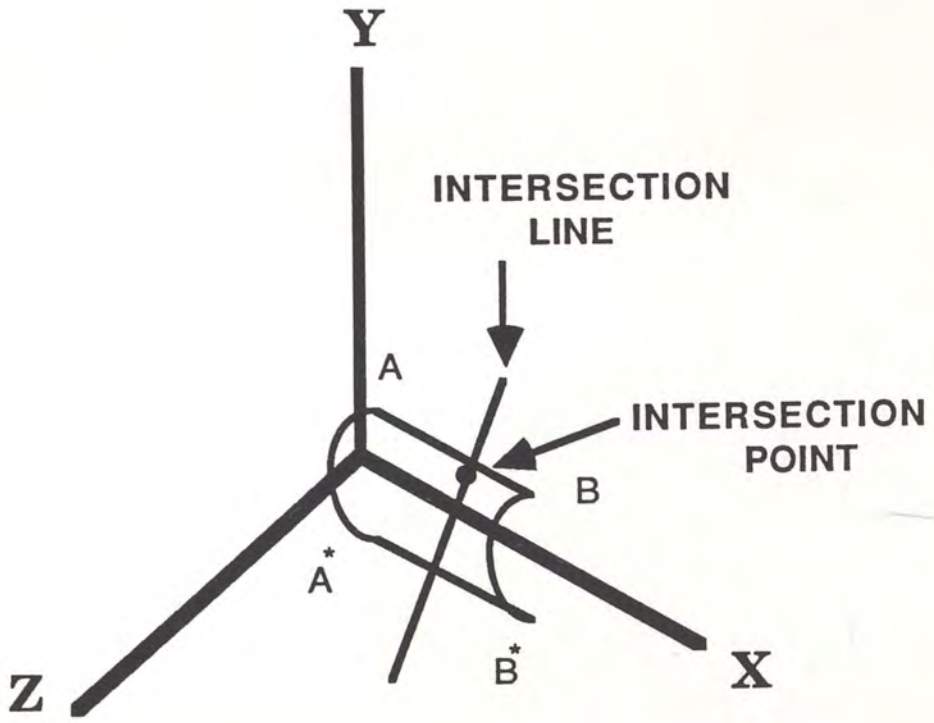

Figure 11. Critical Point in a cylinder

Next, the Critical Line must be found. The Critical Line is the shortest line from the Critical Point to the X-Axis (because this is the center if the cylinder, Figure 12). Given this line, interference will only occur if this line crosses the arc created by the rotation.

The final step in the collision detection for this line is to find the Chord. The Chord is the straight line connecting the two ends of the arc of the cylinder (Figure 13). If the Chord and the Critical Line cross, then collision only occurs if the angle of rotation is less than 180 degrees. If the Chord and the Critical Line do not cross, then collision only occurs if the angle of rotation is more than 180 degrees. If the angle of rotation is exactly 180 degrees, then the rotation must be broken up into two 90 degree steps. This is because the chord will be identical to the diameter of the circle and any Critical Line will always touch the center point of the circle and, therefore, the Critical Line which indicates interference.

Appendix C shows the pseudo-code used for writing the sample program created with this paper.

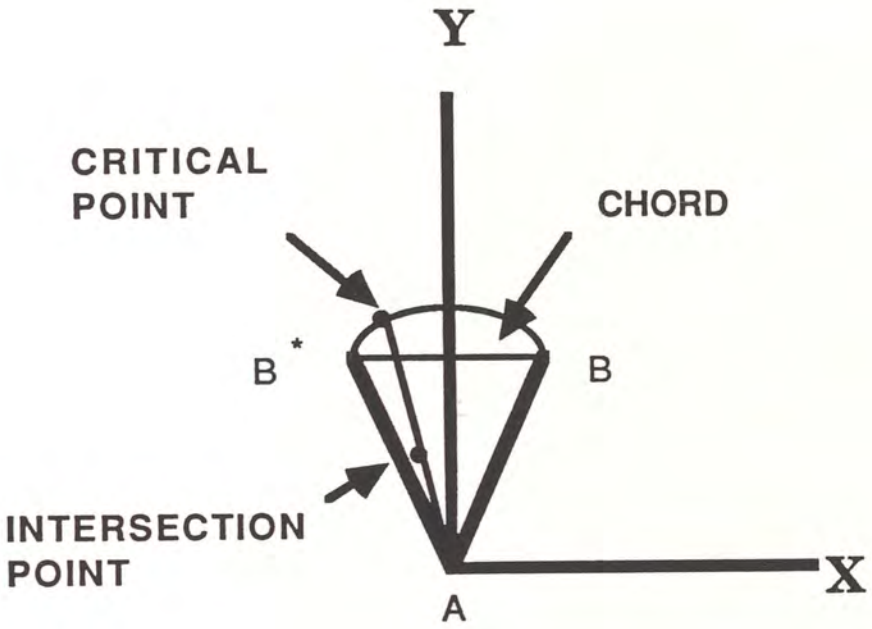

Figure 12. Chord - Critical Line check.

A cone is the conic section generated when both the line being rotated and the Axis of Rotation are coplanar, in the same plane, but the line is not parallel to the Axis of Rotation. The general equation of a cone oriented along the X-Axis, as seen in Calculus and Analytic Geometry is:

$$Y*Y + Z*Z = X*X/c$$

Where c is a constant indicating the slope of the side of the cone.

Every line in all of the non-moving objects can then be checked to see if it intrudes into the cone so determined. If any line intrudes, all further checking can be skipped because the rotation cannot be performed without a collision.

The cone case is identical to the cylinder case except: 1) the line can not be parallel to the cylinder (it must pierce the cylinder somewhere), and 2) the above equation for a cone is used in place of the cylinder equation when determining the Intersection Point and the Critical Point.

Appendix C shows the pseudo-code used for writing the sample program created with this paper.

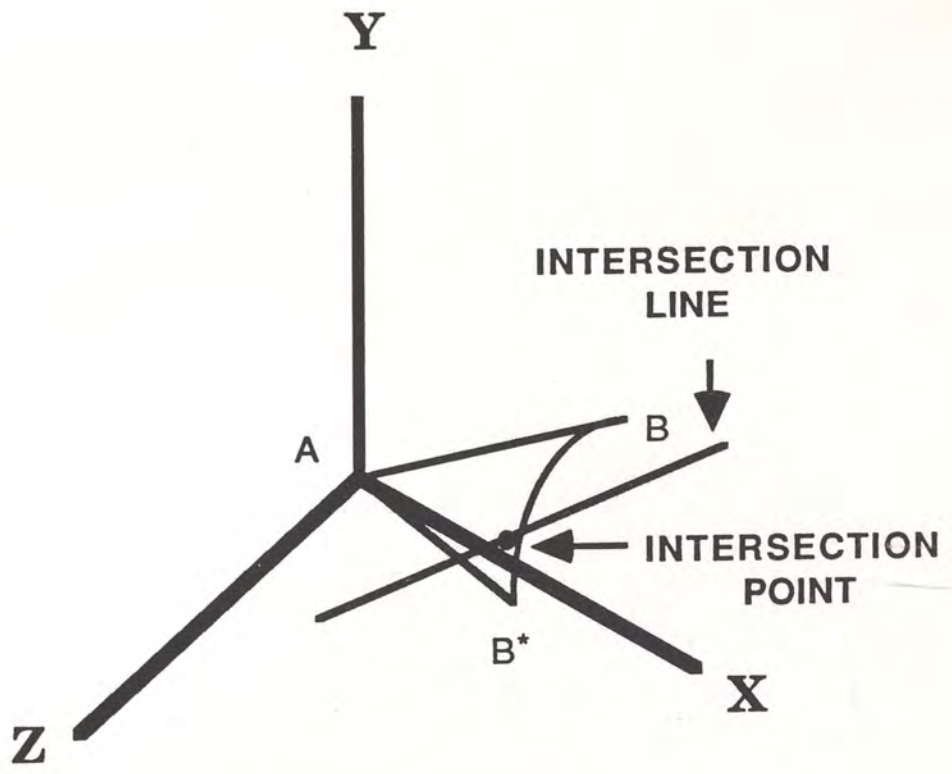

Figure 13. Critical Point in a cone.

A conic is the conic section generated when the line being rotated and the Axis of Rotation are not coplanar. The general equation for a conic oriented along the X-axis, as seen in Calculus and Analytic Geometry is:

$$Y*Y + Z*Z = (X - a)*(X - a)/c + R*R$$

Where a is the coordinate along the X-axis where the radius of the conic is a minimum, c is a constant indicating the slope of the side of the cone, and R is the minimum radius, such that when a = 0 and the minimum radius is zero (R = 0), the equation reduces to the cone equation given previously.

Every line in all of the non-moving objects can then be checked to see if it intrudes into the conic so determined. If any line intrudes, all further checking can be skipped because the rotation cannot be performed without a collision.

However the collision detection for this case does not need to be done this way. The logic and coding can be simplified even more if the conic is broken up into two sections. If the conic is cut into two pieces dividing the pieces at X = a, then two conic sections are created (Figure 14). By rotating the line about the Axis of Rotation every point on the line keeps a constant distance from the the axis. This means that each of the conic sections created

by the cut look like a cone around the Axis of Rotation. Both of these cones can use the same logic which checks for collisions in cones. Some minor transformation of the data for the rotating line is required to use the same logic. This, however, is much more efficient, in a computer program, than writing another complete collision detection routine for this case.

Appendix C shows the pseudo-code used for writing the sample program created with this paper.

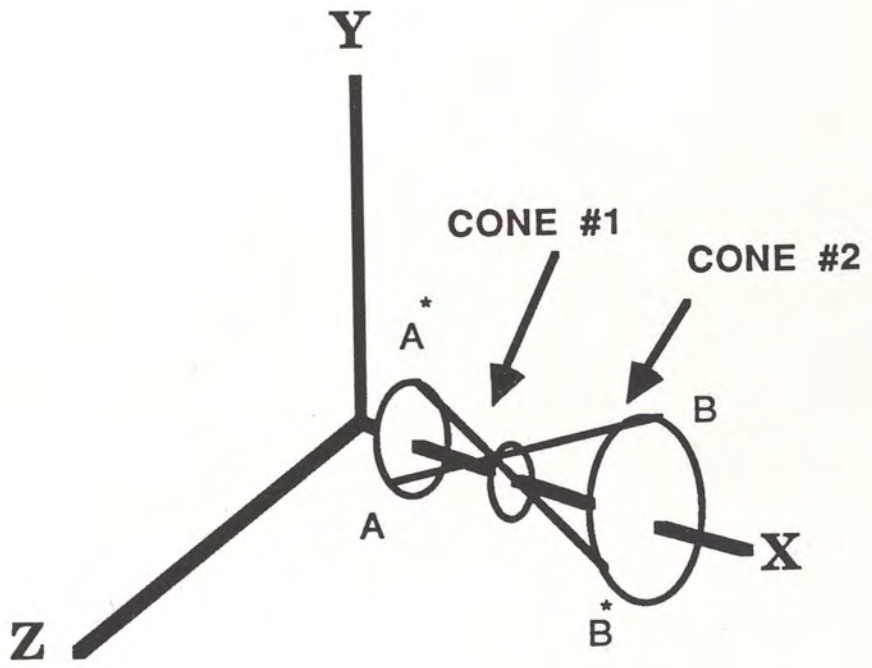

Figure 14. Dividing a conic into two cones.

CONCLUSIONS

This paper discusses the one-step collision detection algorithm in much detail. The theory behind the algorithm is also discussed. In this section the one-step method will be compared with the the multi-step method. The time comparison contains very crude estimates since a program using the multi-step method could not be found to use for comparison. However, the results do indicate the advantage of the one-step method.

A multi-step collision detection program works in discrete steps. It will move the rotating object a small distance, then check to see if any objects are currently intersecting one another. The steps are a mathematical approximation of the actual path taken by the rotating object. If the steps were used to generate an equation of the rotating objects path, the function would be discontinuous. No mater how small the steps are made there is a possibility that this method will not find a collision because it occurs in-between two steps. Also, this method presents a problem in determining the size of the step. Assumptions must be

made regarding the objects in the scene so that an algorithm to determine the size of the step can be created. As the step gets larger the likeliehood of missing a collision gets larger. As the step gets smaller the number of steps and therefore the number of collision checks needed to perform a given rotation gets larger. This causes a compromise to be made between speed and accuracy.

A one-step collision detection method does not approximate the path travelled by the rotating object. The equation of the actual path is calculated. There is no possibility of missing a collision because it occurs in-between steps. There is no trade- off between speed and accuracy. The algorithm takes the same amount of time to check for collisions on a one degree rotation or a one hundred degree rotation.

In order to compare the two methods a test case must be developed. Assume that the scene contains two cubes, one of which is rotating and one of which is stationary. Each cube is described by eight points, twelve lines, and six planes.

For each of the six moving planes, the multi-step algorithm must: 1) calculate the equation of the moving plane; and, 2) for each non-moving plane calculate the equation of the non-moving plane and check for collisions between the planes. This must be repeated for each step.

For each of the six moving planes, the one-step algorithm must: 1) calculate the equation of the moving plane at its final resting place; and, 2) for each non-moving plane, calculate the equation of the non-moving plane and check for collisions between the planes. This must be done once per rotation. For each of the twelve moving lines, the one-step algorithm must: 1) calculate the equation of the surface defined by the moving line; and, 2) for each non-moving line check if the line intrudes into the given surface.

The multi-step algorithm must calculate the equation of forty-two planes (the six moving planes and the six non-moving planes once per moving plane: 6 + 6*6) and perform thirty-six collision checks, once per step. The one-step algorithm must perform the exact same calculations for the final resting place of the rotating object and it must calculate twelve surfaces then perform 144 additional collision checks (each of the twelve non-moving lines once per moving line).

Assume that the calculation of the equation of a plane, the calculation of the equation of a surface, and a collision check all take the same amount of time. While the one-step method performs 222 calculations, the multi-step method performs seventy-eight calculations per step. After only

three steps the multi-step method has performed 234 calculations. Even assuming that the sometimes more difficult calculations in the one-step method take three times as long, the fixed number of these calculations make this method quickly become more efficient than the multi-step method.

This paper discusses only the case of objects rotating about an axis. The translating objects case is not considered. This is because the equations of the surfaces created by rotating objects can be much more difficult to calculate than those created by translating objects. Using the same theory as the rotating object case, as discussed in this paper, the translating object case becomes very simple to check for collisions in. Any translating line will always define a plane. This plane must then be checked against all non-moving planes for possible collisions. This is very similar to the Plane Check in the rotating object case except that while the plane in the rotating object case is wedge-shaped with rounded ends, the plane in the translating object case is a parallelogram. This results in much simpler equations and calculations.

Although this collision detection method worked, some assumptions were made in the algorithm. First, it was assumed that only one object could be rotated at a time.

Second, it was assumed that a collision will occur only if a line from a non-moving object pierces a side of the volume described by the rotating plane or if the two planes intersect when the the rotating object is at its final resting place. Since, in the system developed, the definition of an object is dynamically determined (the rotating object could be the hand on the first rotation, the hand and forearm on the second rotation, and the whole arm on the third rotation), the first assuption was deemed non-restrictive. The second assumption does not consider the case where a very small second object is completely contained in the volume described by the rotating plane (Figure 20). If similarly sized objects are used, this condition would almost never occur. Therefore a third assumption, that all objects be similarly sized, was needed. Another step could be included to check for this condition after all other collision checks have been completed but that will not be covered in this paper.

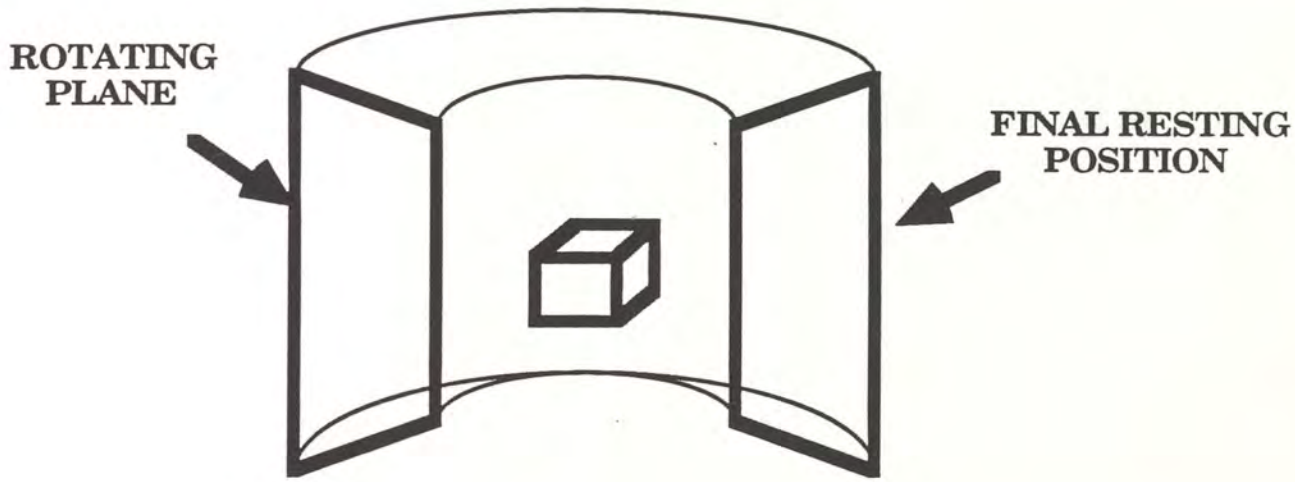

Figure 15. Undetected collision case.

The system was to be developed on a micro-computer in order to increase its potential number of users. Although this is still deemed possible, it was only partially accomplished. The system was written in Pascal and implemented on an IBM PC/XT with a 640K memory. In order to simplify development the system was divided into four parts: 1) object data manipulation, 2) object drawing and rotation, 3) scene data manipulation, and 4) scene drawing with object rotation and collision detection. Each of these parts was

successfully implemented in a separate program. All but the scene drawing program were completed. However, when a third case on the collision detection algorithm, the Conic Check, was being written problems developed. The author could no longer compile the program due to memory restrictions. The memory restrictions were not because of insufficient memory but a restriction on the compiler, Turbo Pascal, that the executable image of any program be 32K or less. Because of this restriction many desirable options, such as hidden line removal, were omitted from the system to allow for the maximum amount of room for the collision detection algorithm. It is the authors opinion that the system could be completely implemented on a micro-computer if a less restrictive compiler was found.

In conclusion, a system of programs was developed which would check for collisons between objects without breaking the movement of the objects up into many small steps. The program was developed on a micro-computer to provide for a larger number of users. Finally, the program was effective and timely in its calculated results.

APPENDIX A

DEFINITIONS

Figure

A figure is a specific item (i.e., a chair). It is made up of one or more objects (the seat, the back, and the legs) which are put together to create the desired figure.

Hidden-Line Removal

Hidden-line removal is the programing for removing those lines normally invisible to the viewer.

Homogeneous Coordinates

Homogeneous coordinates is a convienient notation for solving the translation of coordinates in threedimensional space, if the system is treated as a fourdimensional problem. The coordinates in the four-dimensional system are referred to as homogeneous coordinates. Any point (X, Y, Z) can be translated to homogeneous coordinates simply by making the fourth element 1; (X, Y, Z, 1).

Line

A line is the straight line connecting two points in three-dimensional space.

Line Equation

The line equation is the slope-intercept equation for a line:

$$Y = m*X + b.$$

Object

An object is a single primitive from which a figure is created (i.e., the back of the chair or a joint of a robot arm).

Matrices

A matrix is a mathematical format for representing points in three-dimensional space. The matrix combined with the homogeneous coordinate system provides a convenient format for storing, displaying, and manipulating three-dimensional point data.

Menu

A menu is a list of options from which the user must choose the next action to be performed.

Normal Vector

The normal vector is the vector which is perpendicular to the indicated plane. It is also used to indicate the orientation of a plane for hidden-surface removal routines.

Plane Equation

The plane equation can be written two ways:

1) Point-normal form

$$N*P + C = 0$$

N => Normal vector for the plane.

P => Any point on the plane, variable.

C => This is a known point on the plane.

2) Intercept form

$$a*X + b*Y + c*Z + d = 0$$

a, b, c, & d => constants.

The conversion from one form to the other is simple and the a, b, c in the intercept form becomes the X, Y, Z coordinates of the normal vector in the point-normal form.

Plane

A plane is the side of an object, whose edges are a series lines.

Point

A point is any point in three-dimensional space defined by an X, Y, and Z coordinates.

Polygon

A polygon is a two-dimensional, multi-sided figure. All planes in an object are polygons.

Scene

A scene is a group of figures placed in a common picture to create a setting. A scene is built from multiple objects. Even though the objects are put together to create figures, this is done in the same step where the scene is built. It is not a separate step.

Transformations

Transformations are the mathematical operations performed on the point data to determine the new coordinates after a rotation or translation is performed.

Wire Frame

Wire frame drawing refers to a scene which has all of the lines in the scene displayed (i.e. the hidden lines are not removed).

APPENDIX B

ROTATION ABOUT AN ARBITRARY AXIS

Given the 3x3 matrices from the matrix manipulation section it is possible to determine a single matrix which will result in the equivalent transformation as a rotation about an arbitrary axis. In order to perform such a rotation about some arbitrary axis in space, the following steps must be executed:

STEP 1: Translate one end of the axis to the origin of the primary coordinate system [0 0 0].

STEP 2: Rotate the axis to a position where it rests exactly on the X-axis.

STEP 3: Rotate the point around the X-axis the given number of degrees.

STEP 4: Reverse step 2, rotate the axis back to its relative angle with the X-axis.

STEP 5: Reverse step 1, translate the end of the axis back to its original position.

Most of the steps refer to the "Axis of Rotation." The transformations described are only performed on the set of points to be rotated. The steps are described in this way so that the reasoning behind the indicated transformation is easy to understand.

Throughout the rest of the discussion on ROTATION ABOUT AN ARBITRARY AXIS, the end points of the "Axis of Rotation" will be referred to as:

$$A = [X_a \quad Y_a \quad Z_a]$$
$$B = [X_b \quad Y_b \quad Z_b]$$

Also the angle of rotation about the arbitrary axis will be referred to as θ.

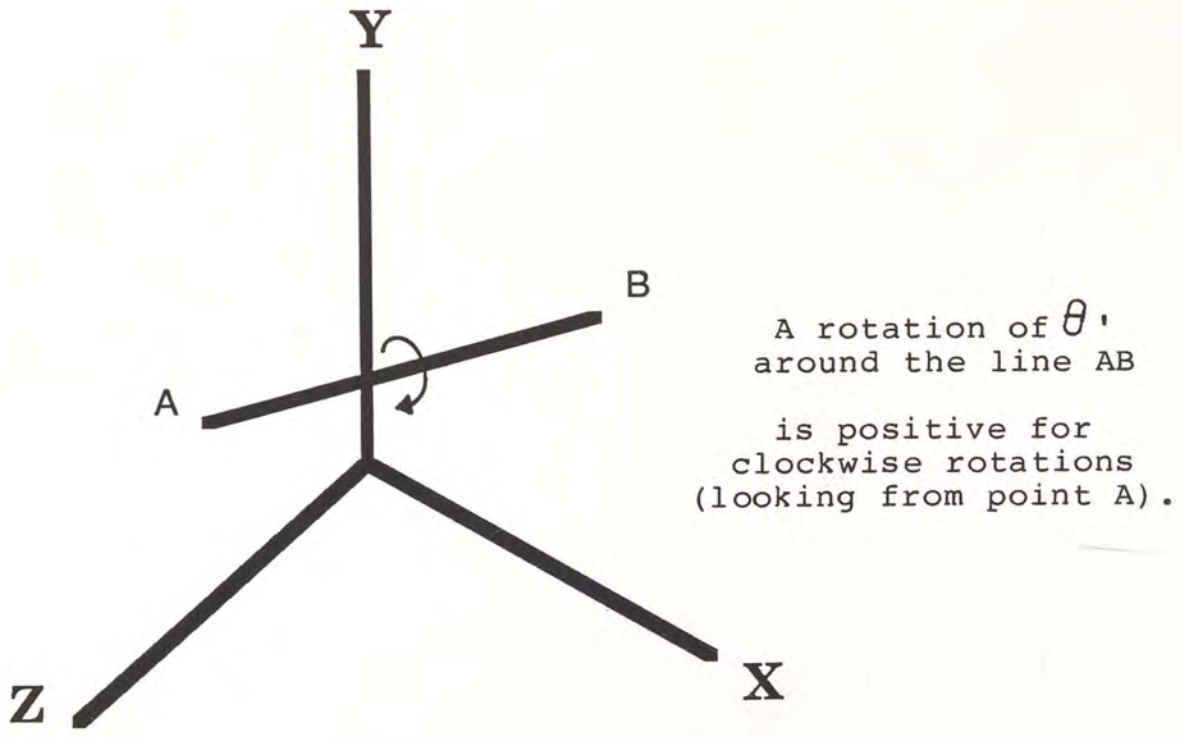

Figure 16. Rotation about an arbitrary axis.

In order to perform the translation of the points to the primary origin the following transformation matrix must be used:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -X_a & -Y_a & -Z_a & 1 \end{bmatrix}$$

If the homogeneous point matrix is multiplied by this matrix, then the result for each individual point is:

$$[X-X_a \quad Y-Y_a \quad Z-Z_a \quad 1]$$

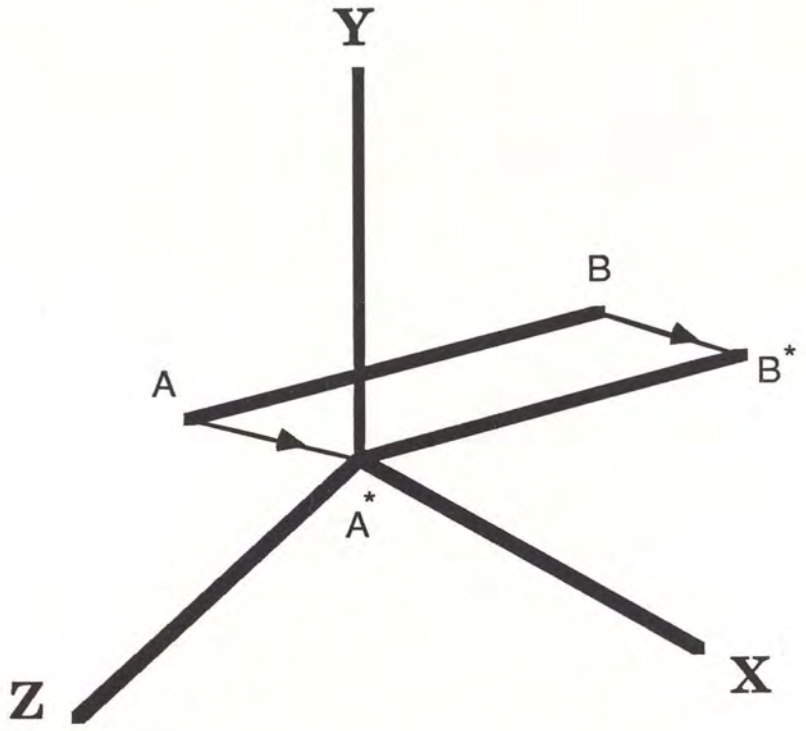

Figure 17. Translation of the arbitrary axis.

Rotating the translated axis of rotation to the primary X-axis can be done in two steps. First rotate the translated axis to the XZ plane, then rotate the axis to the X-axis.

In the following discussion the end points of the translated axis of rotation will be refered to as A^* and B^*. The angle between the translated axis and the XZ plane will be represented by alpha (α) and the angle between the rotated axis and the X-axis will be represented by beta (β).

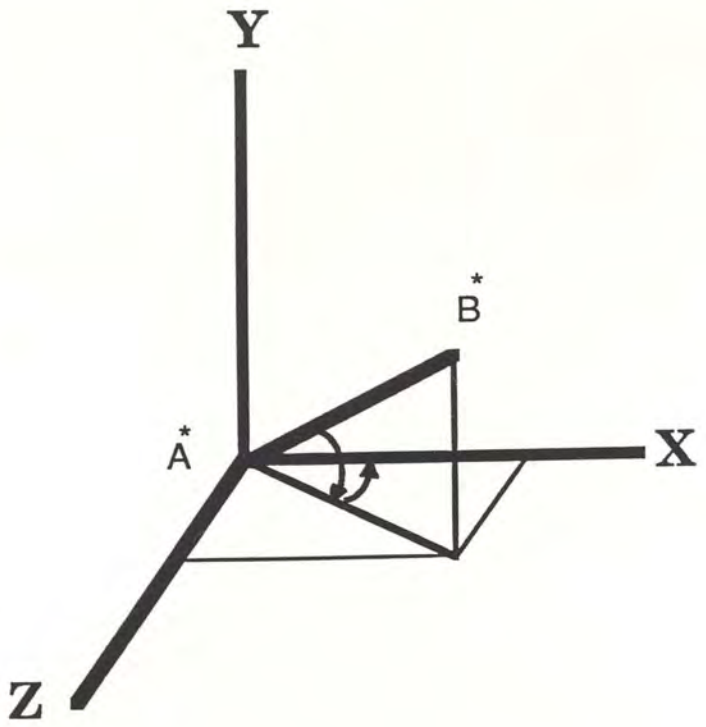

Figure 18. Rotation of the Axis of Rotation.

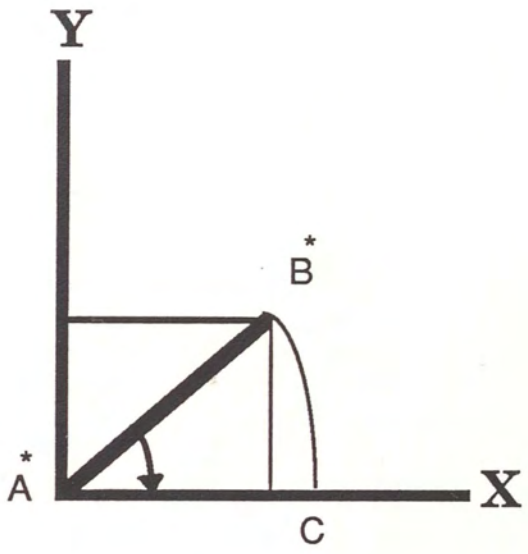

Figure 19. Rotation to the XZ plane.

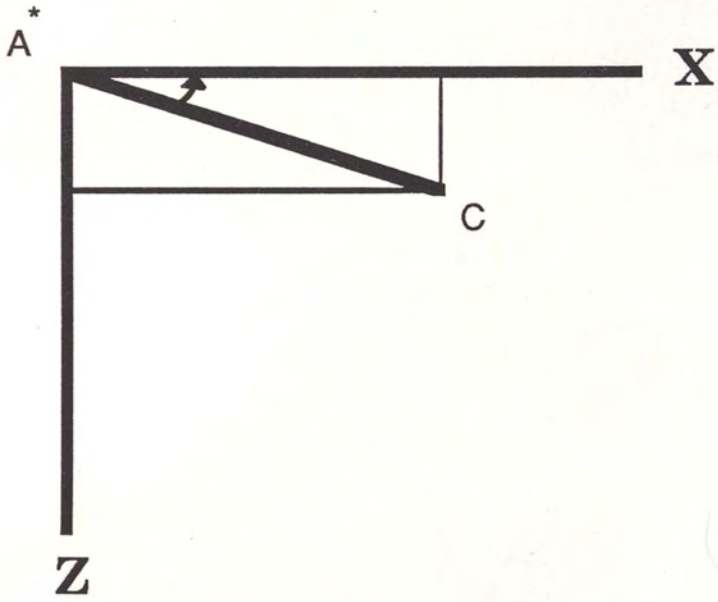

Figure 20. Rotation to the X axis.

For a negative rotation around the Z-Axis, the transformation matrix is:

$$\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For a rotation around the Y-Axis, the transformation matrix is:

$$\begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multipling these two matrices together yields:

$$\begin{bmatrix} \cos\alpha\cos\beta & -\sin\alpha & -\cos\alpha\sin\beta & 0 \\ \sin\alpha\cos\beta & \cos\alpha & -\sin\alpha\sin\beta & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To rotate the points about the primary X-axis, they must be multiplied by the following transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$

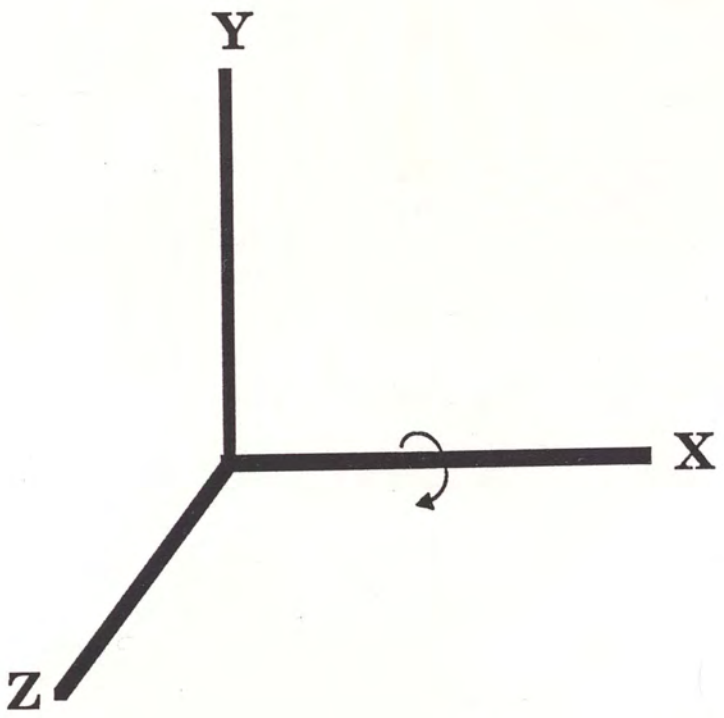

Figure 21. Rotation around the X axis.

Rotating the axis back to the "translated axis" position can be done in two steps. These steps are identical to those performed in STEP 2, but they must be performed in the opposite order.

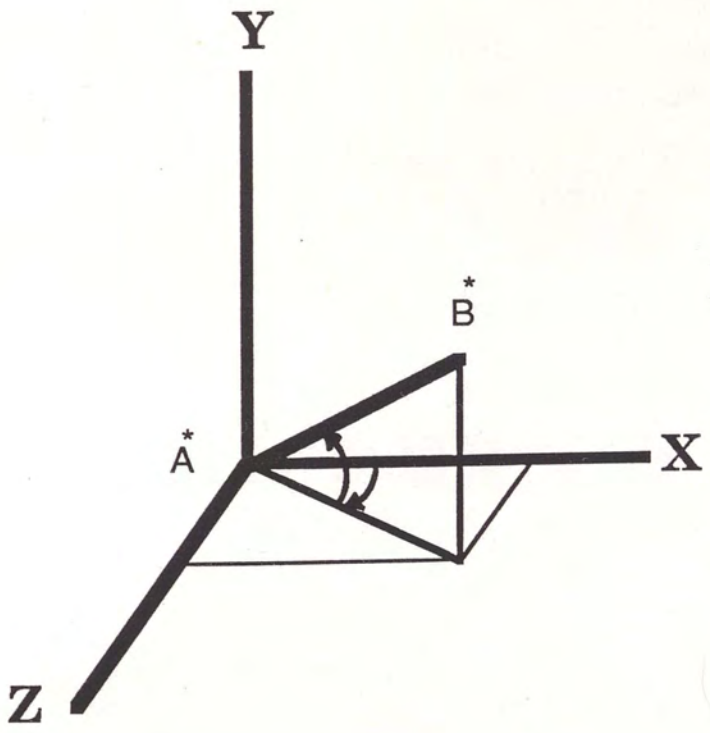

Figure 22. Rotation from the X axis.

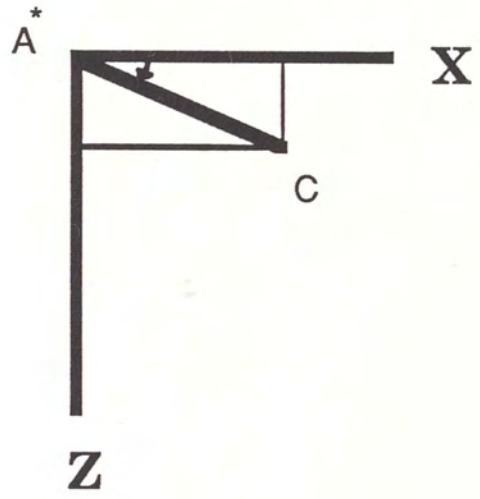

Figure 23. Rotation to the XZ plane.

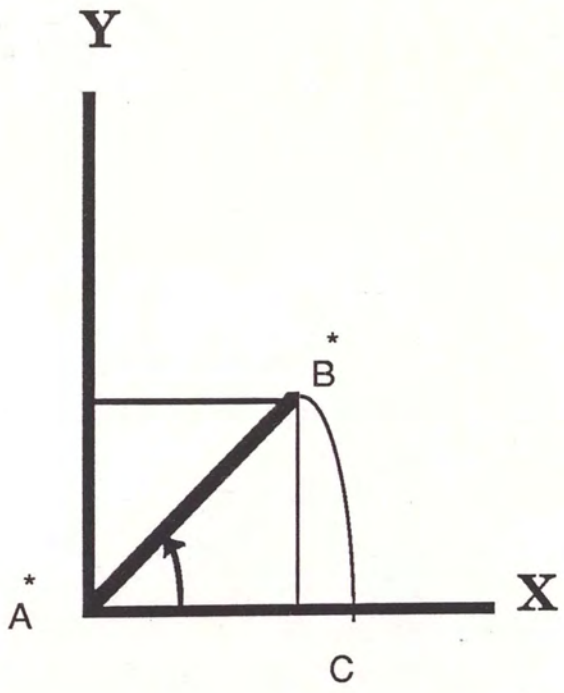

Figure 24. Rotation from the XZ plane.

For a negative rotation around the Y-Axis, the transformation matrix is:

$$\begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For a rotation around the Z-Axis, the transformation matrix is:

$$\begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multipling these two matrices together yields:

$$\begin{bmatrix} \cos\alpha\cos\beta & \sin\alpha\cos\beta & \sin\beta & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ -\sin\alpha\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In order to perform the translation of the points from the primary origin back to its original position, the following transformation matrix must be used:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ X_a & Y_a & Z_a & 1 \end{bmatrix}$$

If the homogeneous point matrix is multiplied by this matrix, then the result for each individual point is:

$$[X+X_a \quad Y+Y_a \quad Z+Z_a \quad 1]$$

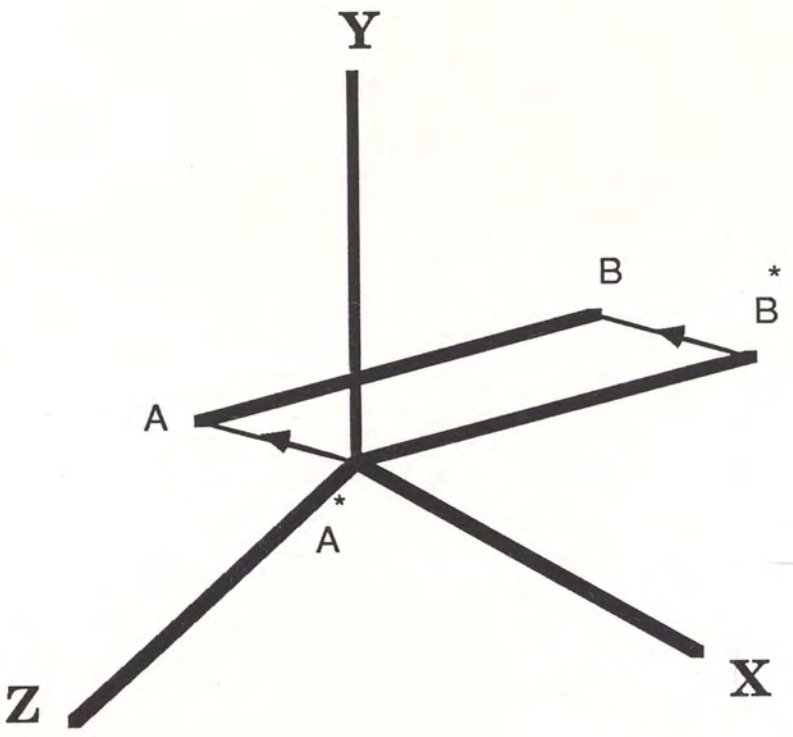

Figure 25. Translation back to the arbitrary axis.

Multipling the five matrices together yields the following matrix:

$$\begin{bmatrix} T_{11} & T_{21} & T_{31} & 0 \\ T_{12} & T_{22} & T_{33} & 0 \\ T_{13} & T_{23} & T_{33} & 0 \\ T_{14} & T_{24} & T_{34} & 1 \end{bmatrix}$$

where

$$T_{11} = \cos\alpha*\cos\alpha*\cos\beta*\cos\beta - \cos\alpha*\sin\alpha*\sin\beta*\sin\theta + \sin\alpha*\sin\alpha*\cos\theta + \cos\alpha*\cos\alpha*\sin\beta*\sin\beta*\cos\theta + \cos\alpha*\sin\alpha*\sin\beta*\sin\theta$$

$$T_{12} = \cos\alpha*\sin\alpha*\cos\beta*\cos\beta - \sin\alpha*\sin\alpha*\sin\beta*\sin\theta - \cos\alpha*\sin\alpha*\cos\theta + \cos\alpha*\sin\alpha*\sin\beta*\sin\beta*\cos\theta - \cos\alpha*\cos\alpha*\sin\beta*\sin\theta$$

$$T_{13} = \cos\alpha*\cos\beta*\sin\beta + \sin\alpha*\cos\beta*\sin\theta - \cos\alpha*\cos\beta*\sin\beta*\cos\theta$$

$$T_{14} = X_a - X_a T_{11} * - Y_a T_{12} - Z_a T_{13}$$

$$T_{21} = \cos\alpha*\sin\alpha*\cos\beta*\cos\beta + \cos\alpha*\cos\alpha*\sin\beta*\sin\theta - \cos\alpha*\sin\alpha*\cos\theta + \cos\alpha*\sin\alpha*\sin\beta*\sin\beta*\cos\theta + \sin\alpha*\sin\alpha*\sin\beta*\sin\theta$$

$$T_{22} = \sin\alpha*\sin\alpha*\cos\beta*\cos\beta + \cos\alpha*\sin\alpha*\sin\beta*\sin\theta + \cos\alpha*\cos\alpha*\cos\theta + \sin\alpha*\sin\alpha*\sin\beta*\sin\beta*\cos\theta - \sin\alpha*\cos\alpha*\sin\beta*\sin\theta$$

$$T_{23} = \sin\alpha*\cos\beta*\sin\beta - \cos\alpha*\cos\beta*\sin\theta - \sin\alpha*\cos\beta*\sin\beta*\cos\theta$$

$$T_{24} = -X_a T_{21} + Y_a - Y_a T_{22} * - Z_a T_{23}$$

$$T_{31} = \cos\alpha*\cos\beta*\sin\beta - \cos\alpha*\cos\beta*\sin\beta*\cos\theta - \sin\alpha*\cos\beta*\sin\theta$$

$$T_{32} = \sin\alpha*\cos\beta*\sin\beta - \sin\alpha*\cos\beta*\sin\beta*\cos\theta + \cos\alpha*\cos\beta*\sin\theta$$

$$T_{33} = \sin\beta*\sin\beta + \cos\beta*\cos\beta*\cos\theta$$

$$T_{34} = X_a T_{31} * - Y_a T_{32} + Z_a - Z_a T_{33}$$

APPENDIX C

COLLISION DETECTION ALGORITHM IN PSEUDO CODE

```
procedure COLLISION_DETECT
--
--      Pseudo-code for one step collision detection.
--
--  To simplify the mathematics involved in the entire
--  routine,  it was assumed that the axis of rotation
--  is the X-axis.  This can be done by rotating the
--  Axis of Rotation to the X-axis.  Then using the
--  matrix which performs this transformation to rotate
--  every point during the rest of the process.
--
--  Determine the matrix which would rotate the Axis of
--  Rotation to the X-axis (the Collision Transform
--  matrix).
--
find Collision_Transform;
```

Figure 26. Pseudo-code for Collision_Detection.

```
--
--  Multiply the Axis of Rotation by the Collision
--  Transform matrix.  (Rotate it to the X-Axis.)
--
Axis := Axis X Collision_Transform;
for every Moving_Object do
    for every Line in the Moving_Object do
--
--  Store the starting and ending positions of the line
--  (the before and after rotation positions).
--
      get Start_Moving_Line; get End_Moving_Line;
--
--  If the line and the Axis of Rotation are mutually
--  perpendicular, then the rotating line will generate
--  a plane around the Axis of Rotation.
--
    if (Start_Moving_Line and Axis are perpendicular) then
      goto PLANE_CHECK;
    else
```

Figure 26. Pseudo-code for Collision_Detection (cont).

```
--
--  If the line and the Axis of Rotation (now the same
--  as the X-axis) are parallel, then the rotating line
--  generates a cylinder around the Axis of Rotation.
--
      if (Start_Moving_Line and Axis are parallel) then
        goto CYLINDER_CHECK;
      else
--
--  If the line and the Axis of Rotation are coplanar,
--  in the same plane then then rotating line will
--  generate a cone around the Axis of Rotation.
--
        if (Start_Moving_Line and Axis are coplanar) then
           goto CONE_CHECK;
        else
           goto CONIC_CHECK;
        endif;
      endif;
    endif;
   next Line in this Moving_Object;
```

Figure 26. Pseudo-code for Collision_Detection (cont).

```
--
--  After  every line in this moving object has been
--  checked for possible collisions, the ending position
--  of each plane in the moving object must be checked
--  for possible collisions.
--
  goto FINAL_POSITION_CHECK;
next Moving_Object;
end COLLISION_DETECT
```

Figure 26. Pseudo-code for Collision_Detection (cont).

```
procedure PLANE_CHECK
--
--  Pseudo-code for collision detection during the
--  "plane case" as described earlier.
--
--  Determine the equation of the plane which the line
--  and the Axis of Rotation are in.
--
find Plane;
--
--  Determine the equation for both the inner and outer
--  sphere defined by the rotating line.
--
find Inner_Sphere; find Outer_Sphere;
for every Non_Moving_Object do
  for every Line in the Non_Moving_Object do
    if (Line does not intersect Outer_Sphere) then
      no interference possible with this Line;
      next Line;
    endif;
```

Figure 27. Pseudo-code for Plane_Check.

```
--
--  Determine if the intersection of the line and the
--  plane is a line, a point, or does not exist.
--
    if (Line is parallel to Plane) then
      if (Line_Start_Point is not in Plane) then
        the Line does not intersect the plane;
        next Line;
      else
        the Line and Plane intersect at Intersect_Line;
        Dist_1 := distance from Center_of_Sphere to
                  Start_Intersect_Line;
        Dist_2 := distance from Center_of_Sphere to
                  End_Intersect_Line;
        if (Dist_1 > Outer_Radius) and
            (Dist_2 > Outer_Radius) then
          the Line does not intersect the plane;
          next Line;
      endif;
    else
      the Line and Plane intersect at Intersect Point;
      Intersect_Line:= line from Center_of_Sphere to
                       Intersect_Point;
    endif
```

Figure 27. Pseudo-code for Plane_Check (cont).

```
--
--  The intersect line is a line, in the plane, which
--  will cross the arc which defines the inner sphere.
--
--  To check if the intersection line crosses the arcs
--  of the inner sphere, find where the line crosses the
--  sphere boundary.
--
    the Intersection_Line and the Inner_Sphere intersect
        at Critical_Point;
    Check_Line:= line from Center_of_Sphere to
                 Critical_Point;
    if (Critcal_Point is within the ends of
        Intersection_Line) then
      Cross_Line:= line from Start_of_Start_Moving_Line
                   to Start_of_End_Moving_Line;
```

Figure 27. Pseudo-code for Plane_Check (cont).

```
      if (Angle_Rotated > 180) then
        if (Cross_Line and Check_Line do not cross) then
          *** INTERFERENCE OCCURS AT CROSS_POINT***;
          return;
        endif;
      else
        if (Cross_Line and Check_Line do cross) then
          *** INTERFERENCE OCCURS AT CROSS_POINT***;
          return;
        endif;
      endif;
--
--  If the routine falls through to here, then no
--  interference occurs with this line.
--
   the Line does not intersect the plane;
   next Line in the Non_Moving_Object;
next Non_Moving_Object;
end PLANE_CHECK;
```

Figure 27. Pseudo-code for Plane_Check (cont).

```
procedure CYLINDER_CHECK
--
--  Pseudo-code for collision detection during the
--  "cylinder case" as described earlier.
--
--  Determine the equation of the cylinder which the line
--  creates while rotating about the Axis of Rotation.
--
find Cylinder;
for every Non_Moving_Object do
  for every Line in the Non_Moving_Object do
--
--  If the Line is parallel to the Axis of Rotation then
--  interference can not occur unless the distance from
--  the Axis of Rotation to the line is equal to the
--  radius of the cylinder.
--
    if (Line and Axis are parallel) then
      if (Distance_From_Line = Radius_Of_Cylinder) then
```

Figure 28. Pseudo-code for Cylinder_Check.

```
--
--  The critical line is the line trimmed to piece of
--  the line which is along the length of the cylinder.
--
        if (Critical_Line exists) then
          Critical_Line := Line trimmed to Cylinder;
        else
          no interference possible from this Line;
          next Line;
        endif;
        Critical_Point_1 := Start_Of_Critical_Line;
        Critical_Point_2 := Start_Of_Critical_Line;
      else
        no interference possible from this Line;
        next Line;
      endif;
    else
```

Figure 28. Pseudo-code for Cylinder_Check (cont).

```
--
--  If the line and the Axis of Rotation are not
--  parallel, then the line will pierce the cylinder in
--  up to two places. These are the critical points.
--
      if (Line pierces Cylinder) then
        find Critical_Point_1; find Critical_Point_2;
      else
        no interference possible from this Line;
        next Line;
      endif;
--
--  If the line pierces the cylinder within the arc
--  traced by the rotating line, then line connecting
--  the critical point and the center of the cylinder
--  and the chord of the cylinder must be checked for
--  crossing.  If the angle of rotation is greater than
--  180 degrees and the lines do not cross, then
--  interference occurs.  If the angle of rotation is
--  less than 180 degrees and the lines do cross, then
--  interference occurs.
--
```

Figure 28. Pseudo-code for Cylinder_Check (cont).

```
    for both Critical_Points do
      Check_Line:= line from Center_of_Cylinder to
                   Critical_Point;
      if (Critical_Point is within Cylinder_Limits) then
--
--  Use the points from the moving lines where X is
--  the same as the X coordinate or the Critical point
--
        Cross:= line from Point_Start_Moving_Line
                     to Point_End_Moving_Line;
        if (Angle_Rotated > 180) then
          if (Cross and Check_Line do not cross) then
            *** INTERFERENCE OCCURS AT CROSS_POINT***;
            return;
          endif;
        else
```

Figure 28. Pseudo-code for Cylinder_Check (cont).

```
          if (Cross_Line and Check_Line do cross) then
            *** INTERFERENCE OCCURS AT CROSS_POINT***;
            return;
          endif;
        endif;
      until both Critical_Points done;
    endif;
   the Line does not intersect the plane;
   next Line in the Non_Moving_Object;
next Non_Moving_Object;
end CYLINDER_CHECK;
```

Figure 28. Pseudo-code for Cylinder_Check (cont).

```
procedure CONE_CHECK
--
--  Pseudo-code for collision detection during the
--  "cone case" as described earlier.
--
--  Determine the equation of the cone which the line
--  creates while rotating about the Axis of Rotation.
--
find Cone;
for every Non_Moving_Object do
  for every Line in the Non_Moving_Object do
--
--  The line will pierce the cone in up to two places.
--  These are the critical points.
--
    if (Line pierces Cone) then
        find Critical_Point_1; find Critical_Point_2;
      else
          no interference possible from this Line;
          next Line;
        endif;
```

Figure 29. Pseudo-code for Cone_Check.

```
--
--  If the line pierces the cone within the arc traced
--  by the rotating line, then line connecting the
--  critical point and the center of the cone and the
--  chord of the cone must be checked for crossing.  If
--  the angle of rotation is greater than 180 degrees and
--  the lines do not cross, then interference occurs.
--  If the angle of rotation is less than 180 degrees
--  and the lines do cross, then interference occurs.
--
    for both Critical_Points do
      Check_Line:= line from Center_of_Cone to
                   Critical_Point;
      if (Critical_Point is within Cone_Limits) then
--
--  Use the points from the moving lines where X is
--  the same as the X coordinate or the Critical point
--
          Cross:= line from Point_Start_Moving_Line
                      to Point_End_Moving_Line;
```

Figure 29. Pseudo-code for Cone_Check (cont).

```
          if (Angle_Rotated > 180) then
            if (Cross and Check_Line do not cross) then
              *** INTERFERENCE OCCURS AT CROSS_POINT***;
              return;
            endif;
          else
            if (Cross_Line and Check_Line do cross) then
              *** INTERFERENCE OCCURS AT CROSS_POINT***;
              return;
            endif;
          endif;
        endif;
      until both Critical_Points done;
    the Line does not intersect the plane;
    next Line in the Non_Moving_Object;
next Non_Moving_Object;
end CONE_CHECK;
```

Figure 29. Pseudo-code for Cone_Check (cont).

```
procedure CONIC_CHECK
--
--  Pseudo-code for collision detection during the
--  "conic case" as described earlier.
--
--  Determine the points on the start moving line and on
--  the end moving line which are closest to the Axis of
--  Rotation.
--
find Closest_Point_1; find Closet_Point_2;
--
--  Determine the equation of the cone which the line
--  would create while rotating about the Axis of
--  Rotation.
--
for both Closest_Points do
  find Start_Moving_Line using Closest_Point;
  find End_Moving_Line using Closest_Point;
  find Cone;
  for every Non_Moving_Object do
    for every Line in the Non_Moving_Object do
```

Figure 30. Pseudo-code for Conic_Check.

```
--
--  The line will pierce the conic in two places.
--  These are the critical points.
--
        find Critical_Point_1; find Critical_Point_2;
--
--  If the line pierces the conic within the arc traced
--  by the rotating line, then line connecting the
--  critical point and the center of the cone and the
--  chord of the cone must be checked for crossing.  If
--  the angle of rotation is greater than 180 degrees and
--  the lines do not cross, then interference occurs.
--  If the angle of rotation is less than 180 degrees
--  and the lines do cross, then interference occurs.
--
```

Figure 30. Pseudo-code for Conic_Check (cont).

```
      for both Critical_Points do
        Check_Line:= line from Center_of_Conic to
                     Critical_Point;
          if (Critical_Point is within Conic_Limits) then
--
--  Use the points from the moving lines where X is
--  the same as the X coordinate or the Critical point
--
            Cross:= line from Point_Start_Moving_Line
                         to Point_End_Moving_Line;
            if (Angle_Rotated > 180) then
              if (Cross and Check_Line do not cross) then
                *** INTERFERENCE OCCURS AT CROSS_POINT***;
                return;
              endif;
            else
              if (Cross_Line and Check_Line do cross) then
                *** INTERFERENCE OCCURS AT CROSS_POINT***;
                return;
              endif;
            endif;
        endif;
      until both Critical_Points done;
```

Figure 30. Pseudo-code for Conic_Check (cont).

```
        the Line does not intersect the plane;
        next Line in the Non_Moving_Object;
    next Non_Moving_Object;
next Closest_Point;
end CONIC_CHECK;
```

Figure 30. Pseudo-code for Conic_Check (cont).

```
procedure FINAL_POSITION_CHECK
--
--  Pseudo-code for collision detection checking for
--  interference between planes at the final resting
--  place of the moving plane.
--
for every Moving_Object do
  for every Plane in the Moving_Object do
--
--  Determine the equation of the moving plane.
--
    find Moving_Plane;
    for every Non_Moving_Object do
      for every Plane in the Non_Moving_Object do
--
--  Determine the equation of the non-moving plane.
--
        find Non_Moving_Plane;
        if (Moving_Plane and Non_Moving_Plane are
            parallel) then
          if (Moving_Plane <> Non_Moving_Plane) then
            no interference possible from this Line;
            next Non_Moving_Plane;
          else
```

Figure 31. Pseudo-code for Final_Check.

```
--
--  Check every line in the non-moving plane to see if
--  any cross any of the moving plane lines.
--
          for every Line in the Non_Moving_Plane do
            for every Line in the Moving_Plane do
              if (Moving_Line crosses
                   Non_Moving_Line) then
                  *** INTERFERENCE OCCURS ***
                  return;
              endif;
            next Moving_Line;
          next Non_Moving_Line;
        endif;
      else
```

Figure 31. Pseudo-code for Final_Check (cont).

```
--
--  Determine the line of intersection between the
--  moving plane and the non-moving plane.  Then find
--  if the limits on the line are within the bounds of
--  both the moving plane and the non-moving plane.
--
          find Intersection_Line;
          limit Intersection_Line within Moving_Plane;
          if (Intersection_Line within
              Non_Moving_Plane) then
             *** INTERFERENCE OCCURS ***
             return;
          endif;
        endif;
        next Non_Moving_Plane;
        next Non_Moving_Object;
      next Moving_Plane;
    next Moving_Object;
end FINAL_POSITION_CHECK;
```

Figure 31. Pseudo-code for Final_Check (cont).

LIST OF REFERENCES

[1] Rogers, David F., and Addams, J. Allan. Mathematical Elements for Computer Graphics. New York: McGraw-Hill, 1976.

[2] Giloi, Wolfgang K. Interactive Computer Graphics Data Structures, Algoritms, Languages. Englewood Cliffs, New Jersey: Prentice-Hall Inc, 1978.

[3] Foley, James D., and Van Dam, Andries. Fundamentals of Interactive Computer Graphics. Reading, Massachusetts: Addison-Wessley Publishing Company, 1982.

[4] Vince, John. Dictionary of Computer Graphics. London: Frances Pinter (Publishers), 1984.

[5] Gilbert, Phillip. Software Design and Developement. Chicago: Science Research Associates, Inc, 1983.

[6] Crow, Franklin C. "Three-Dimensional Computer Graphics, Part I," BYTE, March 1981.

[7] Crow, Franklin C. "Three-Dimensional Computer Graphics, Part II," BYTE, April 1981.

[8] Shenk, Al. Calculus and Analytic Geometry. Santa Monica, California: Goodyear Publishing Company, Inc, 1977.

[9] Horowitz, Ellis and Sartaj Sahni. Fundamentals of Data Structures. Chicago: Computer Science Press, Inc, 1976.