Calculus III Applied to Computer Graphics

Vector Graphics for Three-Space Perspective and Transformation

Presented for Calculus III Honors Research Project at Arizona State University[*]

December 5th, 2014

Erick Ramirez Cordero[**]

Symbols	
W = Origin of world coordinate system (3D)	S = Origin of screen coordinate system (2D)
x_W = x-axis for world coordinates	X_S = x-axis of screen coordinates
y_W = y-axis for world coordinates	Y_S = y-axis of screen coordinates
z_W = z-axis for world coordinates	Z_S = z-axis of screen coordinates
E = Origin of eye coordinate system (3D)	d = distance between E and S
x_E = x-axis for eye coordinates	ρ = distance between E and W
y_E = y-axis for eye coordinates	V = viewing transformation matrix
z_E = z-axis for eye coordinates	R = rotation matrix about an arbitrary axis

[*] Mr. Brian England as Supervisor

[**] Undergraduate Honors Student, Computer Science (BS)

I.Introduction and History of Computer Graphics

One of the earliest forms of computer graphics were vector graphics, images and models created by implementing mathematical formulas based on vectors. Compared to bitmap graphics (or raster graphics), vector graphics allowed greater detail and quality that would be lost normally by resizing and manipulating bitmap graphics. Some of the earliest uses of computer graphics include the Semi-Automatic Ground Environment (SAGE) Air Defense System, an early computer game called *Tennis for Two*, and the movie *Tron*. Modern applications of vectored graphics are implemented through coding such as Scalable Vector Graphics (SVG) or through other methods in Java, C, C++, etc.

A. Semi-Automatic Ground Environment (SAGE) Air Defense System

One of earliest uses of vectors in computer graphics was in the SAGE project.SAGE (Semi-Automatic Ground Environment) was a United States Air Force project started in 1951 and implemented in 1963 as an air defense system capable of detecting aircraft. In SAGE, vectors were used as displays to detect aircraft direction and speed relative to geographical locations.

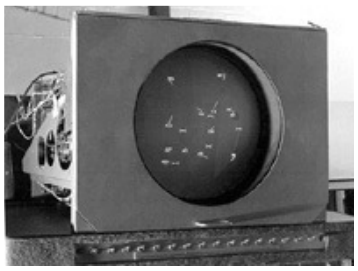

Figure 1.1. An image of SAGE displaying vectors indicating the locations of aircraft.

During the SAGE project, the light pen was also invented for user input. By using vectors in order to keep track of the moving pen, the computer could plot the points it crosses and adjust accordingly. The creation of the light pen would later give way to its applications in computer animation and CAD processes.[1]

B. *Tennis for Two*

One of the predecessors to video games was William Higinbotham's *Tennis for Two*.*Tennis for Two* is a 2D tennis game played with a button and a dial for trajectory on an oscilloscope screen created in Brookhaven National Laboratory. Introduced in October 18th, 1958, this game predates commercialized video games such as *Pong*.

Figure 1.2. 50th Anniversary Recreation of *Tennis for Two*

Higinbotham's original inspiration for creating this game came from him reading the instruction manual for the oscilloscope, which described various examples such as a bullet and bouncing ball. After first introducing his game at a visitors' event, Higinbotham reflected that "it might liven up the place to have a game that people could play, and which would convey the message that our scientific endeavors have relevance for society."[2]

C. *Tron*

One of the first major uses of computer graphic images (CGI) in cinema was *Tron* in 1982. Walt Disney had partnered with various computer graphics companies such as Information International Inc. (Triple-I) and the Mathematic Applications Group Inc.(MAGI) in order to create various objects and backgrounds such as the light cycle and The Grid, the virtual world of the movie.

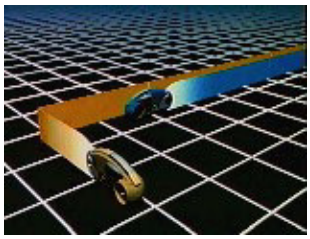

Figure 1.3. Light Cycle sequence created by Triple-I for *Tron*

While most of the computer graphics were backgrounds, the movie contained a scene of pure CGI for 15 minutes, a feat unheard of at the time. Since the concept of computer graphics was still new in cinema, the styles and skill sets of the companies were widely different and thus implemented differently. For instance, Triple-I used a combination of basic shapes in order to

animate them and create action sequences that did not require great amounts of detail, known as SynthaVision. MAGI, on the other hand, would trace a schematic onto a tablet to recreate as a series of polygons for a detailed 3D model. While not as fluid as Triple-I, MAGI's process allowed for greater depth to be created in their objects and backgrounds and would later be refined into the standard process utilized in cinema in the present day. While not a great success financially, Tron paved the way for greater use of CGIin cinema, television, and other forms of media. [3]

D. Scalable Vector Graphics (SVG)

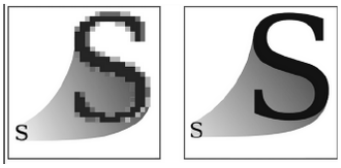

Figure 1.4. Comparison between Bitmap (Left) and Vector (Right) Graphics

Vectored images are created through mathematical equations and expressions, as opposed to Bitmap graphics, to allow proper scaling for a clear image at any size.One of the major forms of programming vector images is the usage of Scalable Vector Graphics (SVG). Other forms of vector coding/drawing such as the Vector Markup Language (VML) were also created, but SVG is the most commonly used by browsers and multimedia software. [4]

Scalable Vector Graphics is an Extensible Markup Language (XML) used in order to create vector images by either defining all the elements of the graphic (i.e. lines, shapes, colors) and/or modifying raster/bitmap images. Compared to other languages, SVG differs greatly depending on the User's process and goals. For instance, SVG images can include other images, text, can be interactive and dynamic through scripting, and can be utilized for other purposes. [5]

II.Three-Space Perspective

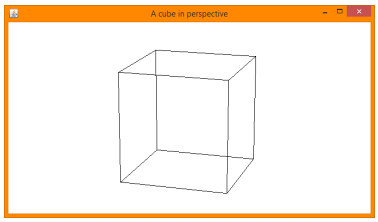

Figure 2.1. A Cube in 3D Perspective

Three-dimensional perspective in computer graphics is created in two major steps: a viewing transformation and a perspective transformation.In the viewing transformation, the program translates the original coordinates, referred to throughout this report as the "world coordinates", into a set of "eye coordinates": a second set of 3D coordinates representing the cube on the other side of the screen. In the perspective transformation, the program takes the both the world coordinates and the eye coordinates created in the first step and creates vectors intersecting a plane that represents the screen of the computer, known as the viewing plane. By doing so the 3D coordinates are translated into a set of 2D coordinates plotted and connected on the screen to create a wire-frame model of a cube. These 2D coordinates are referred to as "screen coordinates" as they are the points on the viewing plane intersecting the vectors.[6]

A. Viewing Transformation

The viewing matrix utilized for creating the eye coordinates is composed of three steps in itself: creating the origin for the eye coordinate system, rotating the coordinate system around the z_W-axis, and rotating the coordinate system around the x_W-axis. In order to rotate the world coordinate system, points are converted from Cartesian coordinates to spherical coordinates:

$$x_W = \rho\sin(\varphi)\cos(\theta) \qquad y_W = \rho\sin\varphi\sin\theta \qquad z_W = \rho\cos\varphi$$

After converting, the origin is shifted from the world coordinate system to the location where the eye coordinate grid will be created.The coordinate system is then rotated about the z_W-axis by

the angle $-\left(\theta+\frac{\pi}{2}\right)$, creating the negative z-axis. Finally, the coordinate system is rotated about the x_W-axis by $-\varphi$, as opposed to just φ in order to achieve the proper rotation.

$$T=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\-x_E&-y_E&-z_E&1\end{bmatrix}\quad R_z=\begin{bmatrix}-\sin\theta&-\cos\theta&0&0\\\cos\theta&-\sin\theta&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\quad R_x=\begin{bmatrix}1&0&0&0\\0&\cos\varphi&-\sin\varphi&0\\0&\sin\varphi&\cos\varphi&0\\0&0&0&1\end{bmatrix}$$

The three matrices T, R_z, and $R_x$can be multiplied into a single viewing matrix V that can be multiplied with the matrix holding the 3D coordinates. By doing so, the set of eye coordinates are created for use in the perspective transformation. On a side note, this transformation creates a negative z-axis in order to retain a right-handed coordinate system.[6]

$$V=TR_zR_x=\begin{bmatrix}-\sin\theta&-\cos\varphi\cos\theta&\sin\varphi\cos\theta&0\\\cos\theta&-\cos\varphi\cos\theta&\sin\varphi\sin\theta&0\\0&\sin\varphi&\cos\varphi&0\\0&0&-\rho&1\end{bmatrix}$$

B. Perspective Transformation

The 2D coordinate system that will be used as the viewing plane is represented by $z_E=-d$, where d is the distance between the eye coordinate origin E and the origin S. These 2D coordinates about the origin S shall be referred to as "screen coordinates". The x_E- and y_E- axis of the eye coordinate system are parallel to the X_S - and Y_S - axis of the viewing plane, and when comparing the vectors connecting the two coordinate systems, the following comparisons are drawn:

$$\frac{X_S}{d}=\frac{x_E}{-z_E}\qquad X_S=-d\left(\frac{x_E}{z_E}\right)\qquad Y_S=-d\left(\frac{y_E}{z_E}\right)$$

Using these equations, the screen coordinates to be connected and rendered on screen are derived.[6]

C.A Cube in Perspective and Vector Representation

CubePers.java is an example program in Leen Ammeraal's and Kang Zhang's textbook *Computer Graphics for Java Programmers*. In CubePers.java, a cube is drawn in perspective through Java by utilizing a viewing transformation and perspective transformation on a set of 3D coordinates representing the cube. The program then displays the cube in a GUI for the User to see as a wire-frame model.

Vectors and matrices in this program, along with the other programs discussed, are created by using arrays and data types for storing coordinatesin order to calculate them more efficiently in code. The classes Point2D and Point3D serve as a customized data types for 2-D Coordinates and 3-D coordinates respectively. By creating arrays with Point2D and Point3D as data types, the result creates matrices for the separate points as opposed to the linear format of generic arrays. The custom array can then be treated as a matrix in calculating new coordinates, drawing lines to form the object, and rotating the object it composes.[6]

```
class Point2D
{
      float x, y;
      Point2D(float x, float y)
      {
            this.x = x;
            this.y = y;
      }
}
```

Figure 2.2. A code snippet of the class Point2D, where 2D coordinates are stored for use in arrays.

III.Rotations in Three-Space

The type of translation utilized by the programs created through this research is a rotation about an arbitrary axis predetermined in code. Rotation occurs in a program before the object goes through the viewing and perspective transformations by multiplying the world coordinates with the rotation matrix. The rotation matrix itself is complex as it is created by multiplying

multiple other matrices relating to a rotation α about vector v extending from the origin. The arbitrary axis and points to be shifted about the axis are translated to this vector v with spherical coordinates and shifted back to the original world coordinate grid with their rotation intact. To create the rotation matrix R, the inverse rotation matrices R_z^{-1} and R_y^{-1}, the rotation matrix R_v about the vector v, and the actual rotation matrices R_y and R_z are all multiplied in that specific order.To complete the rotation matrix, the points have to be translated to the vector v before the rotation matrix is applied and returned to their new position after the rotation through the usage of a transformation matrix T. The full derivation of the rotation matrix is in Appendix A.

Instead of having the program go through the task of multiplying all of these matrices every time a rotation occurs, the coordinates are instead multiplied by the appropriate final rotation matrix equations r_{11} to r_{43}. This calculation occurs in the method rotate within the class Rota3D.[6]

```
static Point3D rotate(Point3D p)
   {
      return new Point3D
            (p.x * r11 + p.y * r21
            +
            p.z * r31 + r41,
            p.x * r12 + p.y * r22 +
            p.z * r32 + r42,
            p.x * r13 + p.y * r23 +
            p.z * r33 + r43);
   }
```

Figure 2.3. A code snippet of the method rotate, a method that multiplies a coordinate by the rotation matrix

A.Two Cubes Rotating

CubRot2.java is another example program from Ammeraal's and Zhang's textbook *Computer Graphics for Java Programmers*. By combining the ideas from rotations and three-dimensional perspectives, this program creates two cubes next to each other that rotate along their own axis.

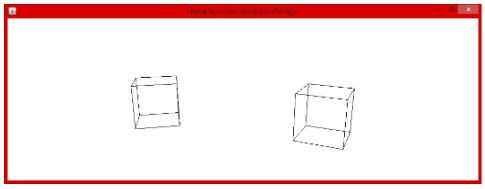

Figure 3.1. Two cubes rotating on their own arbitrary axis

The program itself is essentially a modified version of the previous example program demonstrating a cube in perspective. The modifications include an additional cube and a timer. However, the program also includes an additional class, Rota3D, which applies the rotation to the world coordinate points before the viewing transformation and perspective transformation takes place. The Point3D class in charge of storing the 3D coordinates also requires an additional constructor in order to convert and store the 3D coordinates rotated back into the compatible data type.[6]

IV.A Sword with Three Space Perspective and Rotation

By studying the programs CubePers.java and CubRot2.java, I created a program named ThreeSpaceSwordRotation.java which rotates a wire-frame model of a basic sword about an arbitrary axis for as long as the program is running.

The program begins by generating the Graphical User Interface (GUI) for the object and detecting the center of the GUI. A timer also begins where every time about 20 milliseconds pass by, the angle alpha in which the rotation from the original world coordinates were located is increased by 0.01 degree.

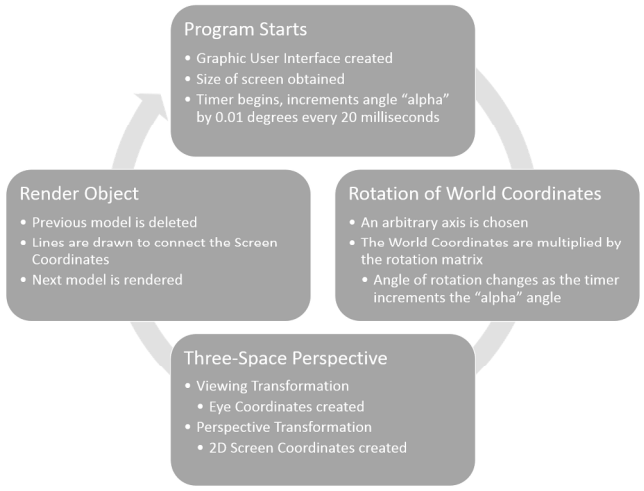

Figure 4.1. Chart representation of the code for the program rotating a sword in Three-Space

The original world coordinate matrix that composesthe object, a sword, is first rotated in the Rota3D class by triggering the method Obj.rotateSword (alpha). An arbitrary axis is formed

by two pre-chosen coordinates, along with the degree variable alpha, and is entered in the method Rota3D.initRotate (). Next, a loop from the first coordinate to the last coordinate occurs in order to have every coordinate pass through the method Rota3D.rotate (s[i]), s[i] representing each starting world coordinate within the matrix.

After being rotated, the method Obj.eyeAndScreen () occurs, causing the viewing transformation and perspective transformation. The method Obj.initPersp () defines the set of variables representing the viewing transformation matrix to be multiplied by the set of world coordinates, then the method Obj.eyeAndScreen () does the actual multiplication and utilizes the equations for perspective transformation in order to create a new set of 2D coordinates in another array vScr [i] with the Point2D class. From there the previous rendered model is erased and the new, rotated model is rendered. The program cycles through this process until the user exits the program.

V. Conclusion

Vector graphics can be applied to several applications due to their images and models coming from vector equations. Due to the images being rendered by vectors, the detail in images can be retained when resizing and manipulating them unlike bitmap graphics. Early applications of computer and vector graphics include the Semi-Automatic Ground Environment (SAGE) Air Defense System, the early computer game *Tennis for Two*, and the movie *Tron*. Modern applications of vectored graphics are implemented through coding libraries such as Scalable Vector Graphics (SVG) and other languages such as Java.

In Java, perspective in three-space is created through a viewing transformation, a viewing matrix created by rotating the original, world coordinates through the use of spherical coordinates. From there, a perspective transformation completes the 3D perspective by using

vectors and an intersecting viewing plane. Rotations in three-space also utilize spherical coordinates and can occur over a variety of axis. A rotational matrix created through a variety of individual rotations rotate the world coordinates before perspective and the model is rendered.

Through researching thedescribed subjects, as well as studying the various example programs and snippets of code analyzed in Sections II.C and III.A, I created a program that creates a basic wire-frame model of a sword, rotates the sword about an arbitrary axis, and renders the rotating model in three-space. Through the use of a timer, the program deletes the previous model and creates a new model with the next rotation every 20 milliseconds. The source code for this program, known as ThreeSpaceSwordRotation.java, can be found in Appendix B.

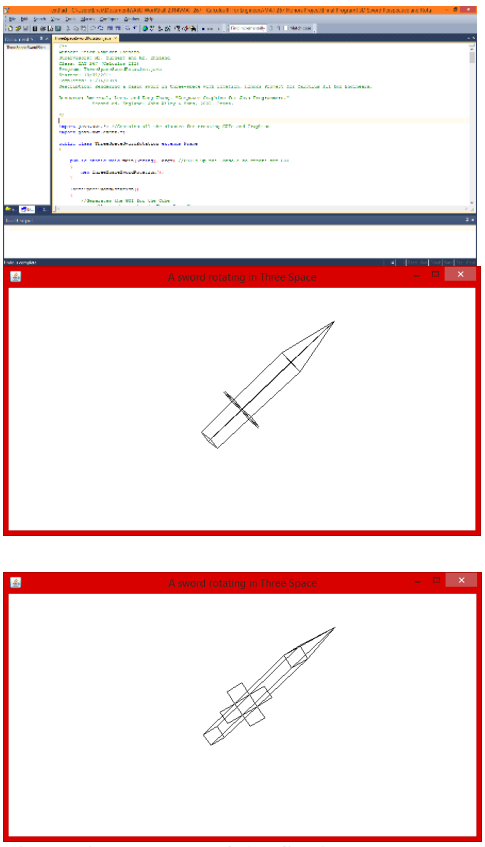

Figure 5.1. Images of the final program: A sword rotating about an arbitrary axis in Three-Space

Appendix A. Derivation of the Rotation Matrix

The actualrotation matrix is complex as the matrix is created by multiplying multiple individual rotation matrices. During the programs explained in Sections III.A and IV, a timer increments an angle α to rotate the vector v through the origin W. The inverse rotation matrices R_z^{-1} and R_y^{-1} are utilized to rotatethe world coordinates about the angles φ and θ used in the spherical coordinates, respectively, in order to get vector v to line up with the z-axis of the world coordinates. The rotation matrix R_v then occurs about the vector v lined up with the z_W axis, utilizing the angle α. The rotation matrices R_y and R_z then returns the rotated coordinates to the original world coordinate system. Since the actual elements of the rotation matrix become larger and complex due to the spherical coordinates utilized, they are represented by their placement in the matrix (11 to 43) later on in the report and in the final program's code.

$$R_z^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R_y^{-1} = \begin{bmatrix} \cos\varphi & 0 & \sin\varphi \\ 0 & 1 & 0 \\ -\sin\varphi & 0 & \cos\varphi \end{bmatrix}$$

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

$$R_y = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying the five matrices in that order result in the following rotation matrix (with an extra row and column added for later):

$$R = R_z^{-1} R_y^{-1} R_v R_y R_z = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$r_{11} = \cos^2\theta(\cos\alpha\cos^2\varphi + \sin^2\varphi) + \cos\alpha\sin^2\theta$$

$$r_{12} = \sin\alpha\cos\varphi + (1-\cos\alpha)\sin^2\varphi\cos\theta\sin\theta$$

$$r_{13} = \sin\varphi(\cos\varphi\cos\theta(1-\cos\alpha) - \sin\alpha\sin\theta)$$

$$r_{21} = \sin^2\varphi\cos\theta\sin\theta(1-\cos\alpha) - \sin\alpha\cos\varphi$$

$$r_{22} = \sin^2\theta(\cos\alpha\cos^2\varphi + \sin^2\varphi) + \cos\alpha\cos^2\theta \qquad r_{23} = \sin\varphi(\cos\varphi\sin\theta(1-\cos\alpha) + \sin\alpha\cos\theta$$

$$r_{31} = \sin\varphi(\cos\varphi\cos\theta(1-\cos\alpha) + \sin\alpha\sin\theta) \qquad r_{32} = \sin\varphi(\cos\varphi\sin\theta(1-\cos\alpha) - \sin\alpha\cos\theta)$$

$$r_{33} = \cos\alpha\sin^2\varphi + \cos^2\varphi$$

To complete the rotation matrix, an additional translation is needed from the arbitrary axis to the vector v about the origin and back to the arbitrary axis after the rotation is complete, represented by a matrix T:

$$R = T^{-1}RT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_x & -a_y & -a_z & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_x & a_y & a_z & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ r_{41} & r_{42} & r_{43} & 1 \end{bmatrix}$$

$$r_{41} = a_x - a_x r_{11} - a_y r_{21} - a_z r_{31} \qquad r_{42} = a_y - a_x r_{12} - a_y r_{22} - a_z r_{32}$$

$$r_{43} = a_z - a_x r_{13} - a_y r_{23} - a_z r_{33}$$

Appendix B. Source Code for ThreeSpaceSwordRotation.java

```
/**
Author: Erick Ramirez Cordero
Supervisors: Mr. Surgent and Mr. England
Class: MAT 267 (Calculus III)
Program: ThreeSpaceSwordRotation.java
Started: 10/31/2014
Completed: 11/16/2014
Description: Rendering a basic sword in three-space with rotation. Honors Project for Calculus
III for Engineers.

Resource: Ammeraal, Leen, and Kang Zhang. "Computer Graphics for Java Programmers."
                        Second ed. England: John Wiley & Sons, 2007. Print.
*/

importjava.awt.*; //Contains all the classes for creating GUIs and Graphics
importjava.awt.event.*;

public class ThreeSpaceSwordRotation extends Frame
{
        public static void main(String[] args) //Pulls up the Console to create the GUI
        {
                newThreeSpaceSwordRotation();
        }

        ThreeSpaceSwordRotation()//Generates the GUI for the Cube
        {
                super("A sword rotating in Three Space");
                addWindowListener(new WindowAdapter()
                {public void windowClosing(WindowEvent e){System.exit(0);}});
                setLayout(new BorderLayout());

                //Creates the Cube to be displayed
                add("Center", new CvCubePers());
                Dimension dim = getToolkit().getScreenSize();
                setSize(dim.width/2, dim.height/2);
                setLocation(dim.width/4, dim.height/4);
                show();
        }
}

classCvCubePers extends Canvas implements Runnable
{
        intcenterX, centerY, w, h; //Variables for centering the objects according to the screen
        Obj obj = new Obj(); // An object for the class Obj
```

```
Image image;
Graphics gImage;

double alpha = 0; //Degree to rotate cube
Thread thr = new Thread(this);

public void run() //Continuous Rotation
{
	try
	{
		for (;;)
		{
			alpha += 0.01; //Increases the degree to rotate the cube
		repaint();
		Thread.sleep(20); //Delay between rotations in milliseconds
		}
	}
	catch (InterruptedException e){}
}

CvCubePers(){thr.start();}

public void update(Graphics g) //Method to repaint the graphic for every rotation
{
	paint(g);
}

intiX(float x)
{
	returnMath.round(centerX + x);
}

intiY(float y)
{
	returnMath.round(centerY - y);
}

void line(int i, int j)
//Draws lines based on the points calculated in the class "Obj" and the size of the screen
{
	Point2D p = obj.vScr[i], q = obj.vScr[j];
	gImage.drawLine(iX(p.x), iY(p.y), iX(q.x), iY(q.y));
}

public void paint(Graphics g)
```

```
{
        Dimension dim = getSize();
        intmaxX = dim.width - 1;
        intmaxY = dim.height - 1;
        centerX = maxX/2;
        centerY = maxY/2;

        intminMaxXY = Math.min(maxX, maxY);

        obj.d = obj.rho * minMaxXY / obj.objSize; //Definesdistance 'd' in class Obj
        obj.rotateSword(alpha); //Triggers the rotation
        obj.eyeAndScreen(); //Triggers the Viewing and Perspective Transformations

        if (w != dim.width || h != dim.height) //Failsafe for if model size exceeds screen
        {
                w = dim.width; h = dim.height;
                image = createImage(w, h);
                gImage = image.getGraphics();
        }

        gImage.clearRect(0, 0, w, h); //Erases previous rotations

        //Lines between screen coordinates to be drawn by line method

        // Horizontal edges at the bottom:
        line(0, 1);
        line(1, 2);
        line(2, 3);
        line(3, 0);

        // Horizontal edges at the top:
        line(4, 5);
        line(5, 6);
        line(6, 7);
        line(7, 4);

        // Vertical edges:
        line(0, 4);
        line(1, 5);
        line(2, 6);
        line(3, 7);

        //Tip of blade
        line(4, 8);
        line(5, 8);
        line(6, 8);
```

```
                line(7, 8);

                //Hilt of Sword
                line( 9,  0);
                line(10,  3);
                line( 9, 10);

                line(11, 1);
                line(12, 2);
                line(11,12);

                line(13, 0);
                line(14, 1);
                line(13,14);

                line(15, 3);
                line(16, 2);
                line(15,16);

                //Base of Sword
                line(17, 18);
                line(18, 19);
                line(19, 20);
                line(20, 17);

                line( 0, 17);
                line( 1, 18);
                line( 2, 19);
                line( 3, 20);

                g.drawImage(image, 0, 0, null); //Renders all of the drawn lines to form the image
        }
}

class Obj // Contains 3D object data and the Transformations
{
        float rho, theta=0.3F, phi=1.3F, d, objSize, v11, v12, v13, v21, v22, v23, v32, v33, v43;
        // Elements of viewing matrix V

        Point3D[] s;//Starting set of points
        Point3D[] w; // World coordinates in an array "w" using Point3D as a data type
        Point2D[] vScr; // Screen coordinates in an array "vScr" using Point2D as a data type

        Obj() //Default Constructor to generate 3D coordinates of cube
        {
                s = new Point3D[21]; //Starting World Coordinates
```

```
        w = new Point3D[21]; //World Coordinates after rotation
        vScr = new Point2D[21]; //Screen Coordinates

        // Middle surface:
        s[0] = new Point3D(0,0,0);
        s[1] = new Point3D(1,0,0);
        s[2] = new Point3D(1,1,0);
        s[3] = new Point3D(0,1,0);

        // Top surface:
        s[4] = new Point3D(4,4,4);
        s[5] = new Point3D(5,4,4);
        s[6] = new Point3D(5,5,4);
        s[7] = new Point3D(4,5,4);

        //Tip of Sword
        s[8] = new Point3D(7,7,7);

        //Hilt of Sword
        s[9] = new Point3D(-1,0,0);
        s[10]= new Point3D(-1,1,0);

        s[11]= new Point3D( 2,0,0);
        s[12]= new Point3D( 2,1,0);

        s[13]= new Point3D(0,-1,0);
        s[14]= new Point3D(1,-1,0);

        s[15]= new Point3D(0,2,0);
        s[16]= new Point3D(1,2,0);

        //Bottom of Sword
        s[17]= new Point3D(-3,-3,-3);
        s[18]= new Point3D(-2,-3,-3);
        s[19]= new Point3D(-2,-2,-3);
        s[20]= new Point3D(-3,-2,-3);

        objSize = (float)Math.sqrt(200F); //object size for rho when used in perspective
        rho = 5 * objSize; //Manipulates object size
}

voidrotateSword(double alpha) //Rotation of 3D Object
{
        Rota3D.initRotate(s[0], s[4], alpha); //Axis of Rotation

        for (int i=0; i<s.length; i++)
```

```
            {
                w[i] = Rota3D.rotate(s[i]);
            }
        }

        voidinitPersp() //Viewing Transformation using spherical coordinates
        {
            floatcosth = (float)Math.cos(theta);
            floatsinth = (float)Math.sin(theta);
            floatcosph = (float)Math.cos(phi);
            floatsinph = (float)Math.sin(phi);

            //Variables for the Viewing Matrix in the Viewing Transformation

            v11 = -sinth;
            v12 = -cosph * costh;
            v13 = sinph * costh;
            v21 = costh;
            v22 = -cosph * sinth;
            v23 = sinph * sinth;
            v32 = sinph;
            v33 = cosph;
            v43 = -rho;
        }
        voideyeAndScreen() //Viewing and Perspective Transformations Triggered
        {
            initPersp();

            for (int i=0; i<s.length; i++)
            {
                Point3D p = w[i];
                float x = v11 * p.x + v21 * p.y;
                float y = v12 * p.x + v22 * p.y + v32 * p.z;
                float z = v13 * p.x + v23 * p.y + v33 * p.z + v43;
                Point3D Pe = new Point3D(x, y, z); //Viewing Transformation
                vScr[i] = new Point2D(-d * Pe.x/Pe.z, -d * Pe.y/Pe.z); //Perspective

                /*
                Creates the 2D points to plot on the screen using the "v11" to "v43"
                variables defined in the class "initPersp()" and stores the
                2D points in the array "vScr"
                */
            }
        }
}
```

```
// Rota3D.java: Class used in other program files for rotations about an arbitrary axis.

class Rota3D
{
	static double r11, r12, r13, r21, r22, r23, r31, r32, r33, r41, r42, r43;

	static void initRotate(Point3D a, Point3D b, double alpha)
	{
		double v1 = b.x - a.x,
			v2 = b.y - a.y,
			v3 = b.z - a.z,
			theta = Math.atan2(v2, v1),
			phi = Math.atan2(Math.sqrt(v1 * v1 + v2 * v2), v3);

		initRotate(a, theta, phi, alpha);
	}

	static void initRotate(Point3D a, double theta, double phi, double alpha)
	{
		doublecosAlpha, sinAlpha, cosPhi, sinPhi,
			cosTheta, sinTheta, cosPhi2, sinPhi2,
			cosTheta2, sinTheta2, pi, c,
			a1 = a.x, a2 = a.y, a3 = a.z;

		cosPhi = Math.cos(phi); sinPhi = Math.sin(phi);
		cosPhi2 = cosPhi * cosPhi; sinPhi2 = sinPhi * sinPhi;
		cosTheta = Math.cos(theta);
		sinTheta = Math.sin(theta);
		cosTheta2 = cosTheta * cosTheta;
		sinTheta2 = sinTheta * sinTheta;
		cosAlpha = Math.cos(alpha);
		sinAlpha = Math.sin(alpha);

		c = 1.0 - cosAlpha;
		r11 = cosTheta2 * (cosAlpha * cosPhi2 + sinPhi2) + cosAlpha * sinTheta2;
		r12 = sinAlpha * cosPhi + c * sinPhi2 * cosTheta * sinTheta;
		r13 = sinPhi * (cosPhi * cosTheta * c - sinAlpha * sinTheta);
		r21 = sinPhi2 * cosTheta * sinTheta * c - sinAlpha * cosPhi;
		r22 = sinTheta2 * (cosAlpha * cosPhi2 + sinPhi2) + cosAlpha * cosTheta2;
		r23 = sinPhi * (cosPhi * sinTheta * c + sinAlpha * cosTheta);
		r31 = sinPhi * (cosPhi * cosTheta * c + sinAlpha * sinTheta);
		r32 = sinPhi * (cosPhi * sinTheta * c - sinAlpha * cosTheta);
		r33 = cosAlpha * sinPhi2 + cosPhi2;
		r41 = a1 - a1 * r11 - a2 * r21 - a3 * r31;
		r42 = a2 - a1 * r12 - a2 * r22 - a3 * r32;
		r43 = a3 - a1 * r13 - a2 * r23 - a3 * r33;
```

```
	}

	static Point3D rotate(Point3D p)
	{
		return new Point3D(p.x * r11 + p.y * r21 + p.z * r31 + r41,
				p.x * r12 + p.y * r22 + p.z * r32 + r42,
				p.x * r13 + p.y * r23 + p.z * r33 + r43);
	}
}

class Point2D //Used as a data type for storage of 2D coordinates
{
	float x, y;
	Point2D(float x, float y)
	{
		this.x = x;
		this.y = y;
	}
}

class Point3D //Used as a data type for storage of 3D coordinates
{
	float x, y, z;
	Point3D(float x, float y, float z)
	{
		this.x = x;
		this.y = y;
		this.z = z;
	}
	Point3D(double x, double y, double z)
	{
		this.x = (float)x;
		this.y = (float)y;
		this.z = (float)z;
	}
}
```

Works Cited

[1] "History: The SAGE Air Defense System." *MIT Lincoln Laboratory*. Lincoln Laboratory, 1 Jan. 2014. Web. 29 Sept. 2014. <https://www.ll.mit.edu/about/History/SAGEairdefensesystem.html>.

[2] "The First Video Game?" Brookhaven National Laboratory. Brookhaven National Laboratory, N.d. Web. 14 Oct. 2014. <http://www.bnl.gov/about/history/firstvideo.php>.

[3]Carlson, Wayne. "A Critical History of Computer Graphics and Animation." Ohio State University, 1 Jan. 2003. Web. 29 Sept. 2014. <https://design.osu.edu/carlson/history/lessons.html>.

[4] Dawber, Damian. *Learning Raphaël JS Vector Graphics.* Birmingham, UK: Packt, 2013. Web. <http://site.ebrary.com.ezproxy1.lib.asu.edu/lib/asulib/reader.action?docID=10714266>.

[5] "Introduction."*Introduction – SVG 1.1 (Second Edition).* World Wide Web Consortium. 16 Aug. 2011. Web. 21 Nov. 2014. <http://www.w3.org/TR/SVG/intro.html>.

[6] Ammeraal, Leen, and Kang Zhang. *Computer Graphics for Java Programmers*. Second Ed. England: John Wiley & Sons, 2007. Print.