LINEAR ALGEBRA AND RETRO GAMING

INTRODUCTION

Without math we would have no computer games. If you strip a computer down to its most basic form what is it? The answer is a calculator. Computers compute numbers, so it shouldn't be surprising that without math we would have no computer graphics. Objects and images in computers games are merely points draw together with lines that operate in space. The use of linear algebra is what allows interactions to occur in a computer. Once linear algebra has been translated into a format that the computer recognizes it can bring visuals to life.

2D Vector Graphics

"Vector graphics refers to representing images by mathematical descriptions of geometric objects, rather than by a collection of pixels on the screen (raster graphics)."

Vector Graphics vs. Raster Graphics

Vector and raster graphics are the two main forms of two-dimensional computer imagery. Raster graphics, or "bitmaps," are generally represented as a rectangular grid of pixels with each pixel being a varying level of color. This systematic collection of pixels, when rendered, forms a 2D image such as in digital photographs.

Vector graphics, however, use mathematical descriptions or expressions to represent images. Unlike raster images, vector graphics render images through the use of points, lines, and other geometrical primitives. At the core of any vector image are "control points," which have defined positions on the XY coordinate plane. These points connect lines or curves and "fill" information that make up a vector image.

The mathematical nature of vector graphics allow vector images to be resolution independent unlike raster-based images in which resolution is dependent on the amount of pixels that the image contains. Vector-based image files are also much smaller in size than raster images, which need more storage space as resolution increases. As the resolution increases in a raster image, so does the number of pixels that need to be saved. Vector graphics, however, can be scaled without any loss in resolution.

Vector Graphics and Linear Algebra Case Study - Retro Gaming

Computer graphics have come a long way since the early days of computers, especially in the gaming industry. Today's video games often make use of ultra-realistic and highly complex 3D computer graphics to immerse the player in a game's world and setting. Game developers of the late 1970s and early 80s, however, did not have access to the powerful computing devices that we have today and had to come up with different ways to push computer graphics technology. One such method was using vector-based graphics.

METHODS

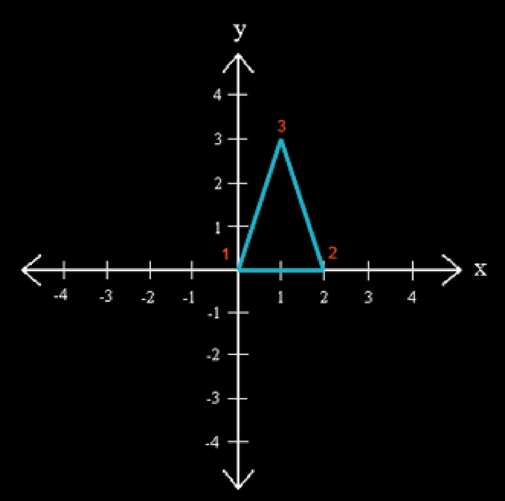
Matrix of vertices

In a game utilizing vector graphics, an object can be constructed using a set of points, or vertices. The coordinates of these points are stored in a "data" matrix such as in the figure below.

Vector data matrix and its equivalent graphical representation on the xy-plane.

Vertex:

$$\begin{array}{r c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{r} \text{x-coordinate} \\ \text{y-coordinate} \end{array} & \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 3 \end{bmatrix} \end{array}$$



Matrix Multiplication

This basic triangular shape can now, for example, represent a game's spaceship. However, a game with a stationary spaceship and nothing else isn't much fun. So, naturally there needs to be a way move the spaceship around the screen or monitor. Fortunately, this can be accomplished with a creative use of a basic concept in linear algebra, matrix multiplication.

Matrix multiplication allows us to perform transformations (rotation and scaling) and translations (position) on our objects.

This can be performed by using the **Row-Column Rule for Computing AB**.

<u>Definition</u>: If the product *AB* is defined, then the entry in row *i* and column *j* of *AB* is the sum of the products of corresponding entries from row *i* of *A* and column *j* of *B*. If $(AB)_{ij}$ denotes the (i, j)-entry in AB, and if A is an m x n matrix, then

$$(AB)_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj}$$

However, before we can perform any transformations and translations, our objects must use homogeneous coordinates since it is not possible to do a 2D translation on a 2D point using matrix multiplication.

Homogeneous Coordinates

Each point (x, y) in R2 can be identified with the point (x, y, 1) on the plane in R3 that lies one unit above the xy-plane. We say that (x, y) has homogeneous coordinates (x, y, 1). Homogeneous coordinates for points are not added or multiplied by scalars, but they can be transformed via multiplication by 3 x 3 matrices.

So, to make use of homogeneous coordinates, we must convert the coordinates in our spaceship matrix. This is accomplished by adding an additional "dummy" coordinate to each vertex and setting its value to one.

The spaceship matrix with homogeneous coordinates should now look like the figure shown to the right.

Vertex:

$$\begin{array}{r c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{r} \text{x-coordinate} \\ \text{y-coordinate} \\ \text{w-coordinate (dummy coordinate)} \end{array} & \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 3 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

Translation

Now that our spaceship matrix uses homogeneous coordinates, we can, for example, make the ship move forward by performing a translation via matrix multiplication.

This translation is done by multiplying a "movement" matrix and the ship matrix with the product being the new location of the spaceship.

The movement matrix consists of a slightly modified 3 x 3 identity matrix that contains x and y coordinates that represent the number of units in each direction that each point in the ship matrix will be "moved" by.

In the sample movement matrix shown on the right, r represents the number of units along the x-axis that each point will be moved by, and s represents the number of units along the y-axis that each point will be moved by.

$$\begin{bmatrix} 1 & 0 & r \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix}$$

Typically, positive r-values will result in movement to the right and negative values will result in movement to the left. Positive s-values will result in movement upward and negative values will result in movement downward.

As a result, any translation of the ship matrix will have the form shown below:

$$\begin{bmatrix} 1 & 0 & r \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} r & 2+r & 1+r \\ s & s & 3+s \\ 1 & 1 & 1 \end{bmatrix}$$

For example, to move the spaceship forward (up) 2 units, we need to translate by (0, 2) as shown below:

Vertex:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 2 & 1 \\ 2 & 2 & 5 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

```
// TRANSLATION

// Declaration of variables
int x = 0, y = 2, total = 0, count = 0;

// Creation of the matrices
int[][] currentPosition = {{0, 2, 1},
                           {0, 0, 3},
                           {1, 1, 1}};
int[][] movement = {{0, 2, 1},
                    {0, 0, 3},
                    {1, 1, 1}};

// Matrix multiplication
for(int k = 0; k < 3; k++) {
   for(int i = 0; i < 3; i++) {
      total = 0;
      for(int j = 0; j < 3; j++) {
         int num = movement[i][j] *
             currentPosition[j][k];
         total = total + num;
      }
      currentPosition[i][k] = total;
    }
}
```

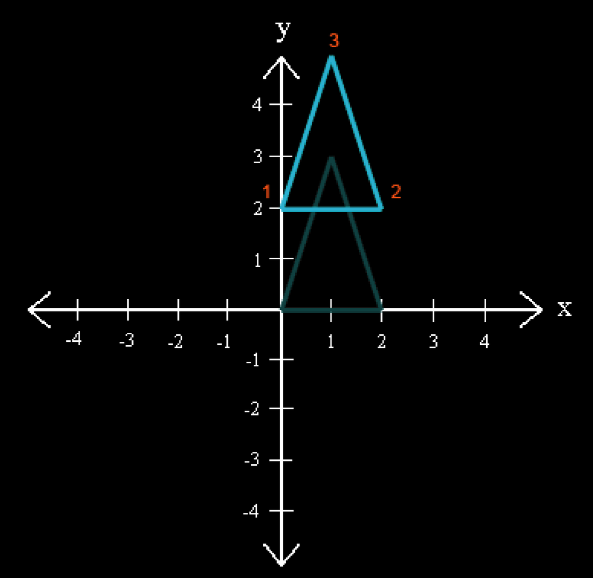


Figure 1 - Ship after translation

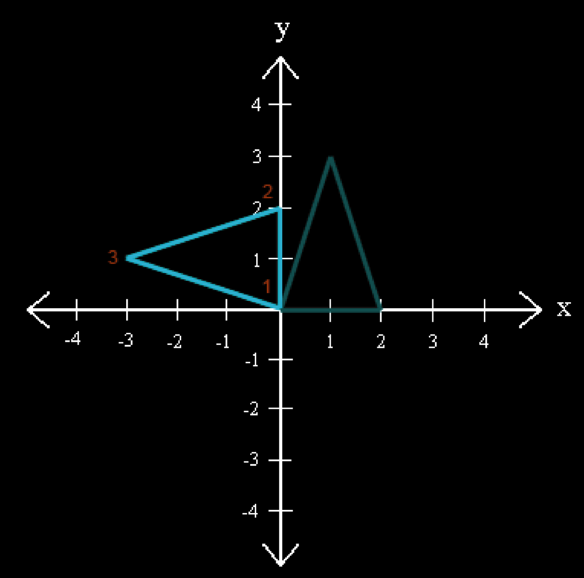


Figure 2 - Ship after rotation

Rotations

Now that the spaceship has the ability to move in 2D space, the ship needs to be able to rotate in the direction that it is traveling, which can be done by performing a transformation via matrix multiplication.

This transformation is done by multiplying a "rotation" matrix and the ship matrix with the product being the new position of the spaceship.

The rotation matrix consists of a partitioned matrix of the form shown on the right, where A is a 2 x 2 matrix that represents clockwise or counterclockwise rotation about the origin at an angle.

$$\begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}$$

Counterclockwise rotation

$$A = \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix}$$

Clockwise rotation

So, to rotate the spaceship 90 degrees to the left, we would perform the following operation:

Vertex:

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & -3 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

The product now represents the spaceship's new position after rotation and is shown graphically in Figure 2.

```
// ROTATION

// Initializing variables
double[] x = new double[3];
int arrayPosition = 0;

//Creating the matrices
int[][] rotation = {{0, 0, 0},
                    {0, 0, 0},
                    {0, 0, 1}};
int[][] currentPosition = {{0, 2, 1},
                           {0, 0, 3},
                           {1, 1, 1}};

// getting the (X, Y) for the center of the
ship
  int centerPointX =
(((currentPosition[0][1]) -
(currentPosition[0][0]))/2);
  int centerPointY =
(((currentPosition[1][2]) -
(currentPosition[1][1]))/2);

// Create the A and A inverse matrices for
the center point
  int[][] A = {{1, 0, centerPointX},
      {0, 1, centerPointY},
      {0, 0, 1}};
  int[][] AInverse = {{1, 0, -centerPointX},
        {0, 1, -centerPointY},
        {0, 0, 1}};

  // Check the direction and create the sub-
  matrix points
    if(keyPressed == LEFT_KEY)
    {
        x[1] = Math.cos(degree);
        x[2] = Math.sinh(degree);
        x[3] = Math.sin(degree);
        x[4] = Math.cos(degree);
    }
    if(keyPressed == RIGHT_KEY)
    {
        x[1] = Math.cos(degree);
        x[2] = Math.sin(degree);
        x[3] = Math.sinh(degree);
        x[4] = Math.cos(degree);
    }

  //Store the values into the matrix
  for(int i = 0; i < 2; i++)
  {
   for(int k = 0; k < 2; k++)
   {
    rotation[i][k] = (int) x[arrayPosition];
    arrayPosition++;
   }
  }
```

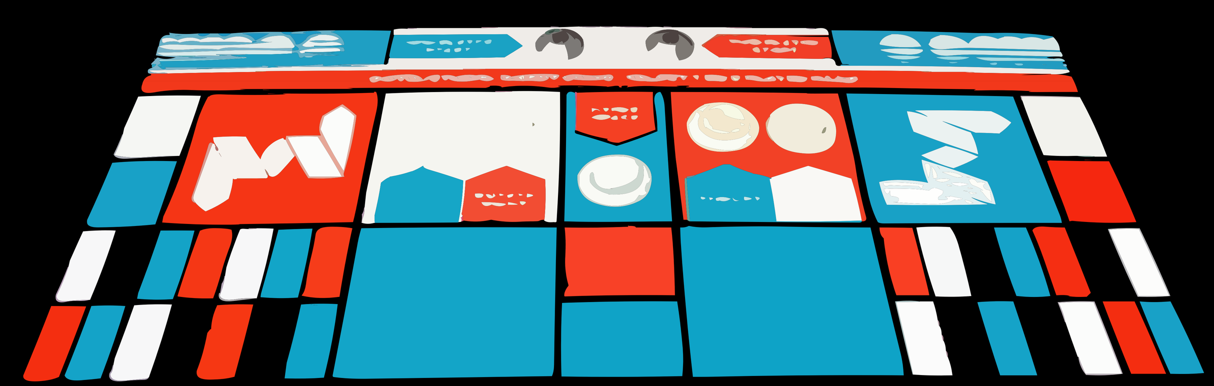
```
// ROTATIONS AND TRANSLATIONS (CONTINUED)

  // Matrix multiplication for rotation and the center point
  for(int k = 0; k < 3; k++) {
     for(int i = 0; i < 3; i++) {
        total = 0;
        for(int j = 0; j < 3; j++) {
           int num = rotation[i][j] * A[j][k];
           total = total + num;
        }
        currentRotate[i][k] = total;
     }
  }

  // Matrix multiplication for the inverse of A
  for(int k = 0; k < 3; k++) {
     for(int i = 0; i < 3; i++) {
        total = 0;
        for(int j = 0; j < 3; j++) {
           int num = currentRotate[i][j] * AInverse[j][k];
           total = total + num;
        }
        currentRotate[i][k] = total;
     }
  }

  // Matrix multiplication for the rotation applied to the current position
  for(int k = 0; k < 3; k++)  {
     for(int i = 0; i < 3; i++) {
        total = 0;
        for(int j = 0; j < 3; j++) {
           int num = currentRotate[i][j] * currentPosition[j][k];
           total = total + num;
        }
        currentPosition[i][k] = total;
     }
  }
```

BY:
WALTER ALVARADO
DANIEL CATIPON
JAMES HALL
KRIS SCHALLER



RESULTS

Composite Transformations

With translations and transformations the spaceship can now move and rotate around the screen. Unfortunately, our ship doesn't rotate in the way that most people expect, which is around the center of the ship instead of around the origin.

To solve this problem, we can use "composite transformations" which combine two or more basic transformations.

So, our composite transformation to rotate around the center of the ship, (1, 1.5), should be:

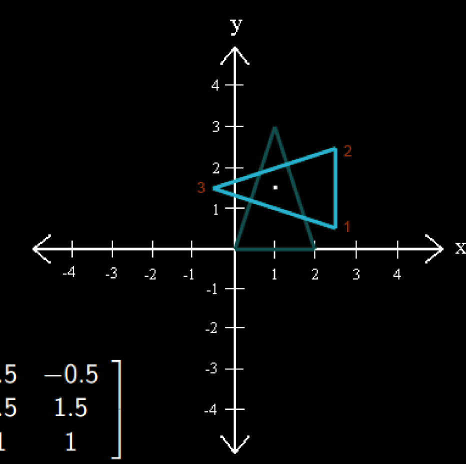
$$A^{-1}BA = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1.5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1.5 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 2.5 \\ 1 & 0 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

A represents the translation by (-1, -1.5).
B represents the rotation counterclockwise by 90 degrees.
A^{-1} represents the translation by (1, 1.5).

This composite transformation first makes the center of the ship the origin, then does the specified rotation, and finally moves the center back where it is supposed to be.

Multiplying the result with the ship matrix, M, will then result in the desired rotation around the ship's center.

$$(A^{-1}BA)M = \begin{bmatrix} 0 & -1 & 2.5 \\ 1 & 0 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2.5 & 2.5 & -0.5 \\ 0.5 & 2.5 & 1.5 \\ 1 & 1 & 1 \end{bmatrix}$$



CONCLUSION

Vector graphics is a linear algebraic way of storing and manipulating computer images. 2D vector graphics were especially suited to moving, rotating, and scaling images and objects within retro games. Through the use of some fundamental concepts of linear algebra, early game developers were able to create games that were revolutionary for its time. At the core of vector graphics based games such as the 1979 classic, Asteroids, are simple matrix multiplications, transformations, and translations. While those games of the past look quite primitive by today's standards, the techniques being used laid the foundation for future advances in computer graphics, especially in 3D computer graphics.

ACKNOWLEDGEMENTS

Weber, Rebecca. *Computer Graphics and Linear Algebra* (2007). Retrieved April 29, 2012. http://www.math.dartmouth.edu/~rweber/teaching/VectorSlides.pdf

Vector graphics, 2012. *In Wikipedia, The Free Encyclopedia*. Retrieved April 29, 2012. http://en.wikipedia.org/wiki/Vector_graphics

Lay, David C. *Linear Algebra and its Applications*. Fourth Ed. Pearson Education, 2012.

Asteroids game screenshot courtesy of Atari / digitaltrends.com.