Mason Howard

March 24, 2014

Dr. Nelson

Linear Algebra

Application of Linear Algebra to Computer Graphics

Computer graphics use linear algebra. Nearly everything that happens in the field, uses linear algebra in some way. From specifying color or defining models and their motion, to procedurally generating graphics, computer graphics utilize linear algebra in nearly every aspect of the field. The open source graphics library, OpenGL, will be an excellent source of examples.

Color is typically represented as a 3-tuple whose values range from 0 to 1 (OpenGL). While this does not quite define a vector space, colors are still vectors, implying various transformations. Typically, colors are made of a combination of red, green, and blue. The set $S = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ spans the set of all colors, where the first element is red, the second is green, and the third is blue. By performing operations on the matrix made from those vectors and multiplying the resultant matrix against the original color, intricate color operations can be performed (Gortler). This has the added bonus of enabling the application of the operations all at once, separating the calculation of the change and commit of the change (Gortler). This is not the only instance of vectors in computer graphics. Model definition uses vectors, as well (Lengyel).

The foundation of models in OpenGL, and all other graphics engines, is the vertex. Depending on whether or not the graphics being displayed is 2D or 3D, a vertex is a 2-tuple or 3D vector. The values of the vector components range from

-1 to 1. Three vertices define a triangle. Four coplanar vertices define a quadrilateral, or quad. Using enough triangles, any polygon can be made. If the triangles are all coplanar, the polygon is a face. Each of the individual triangles are faces, too. The collection of the vertices making up the triangles is called a model. In order to place the model in the world space, a transformation has to take place, one that maintains the relationships of the vertices. There are three kinds of transformations: translation, scaling, and rotation.

Translating a set of vertices can be thought of picking them up and setting them down somewhere else, arranged how one found them. The naive way of translating some vector $v = (x, y z)$ would be $v' = (x + x_1, y + y_1, z + z_1)$. The better way to do it would be to use a transformation matrix. With a transformation matrix, M, one could quickly apply transformations to many vertices through the equation $M \cdot v^* = r^*$, where $v^* = (x, y, z, 1)$, r^* is the transformed vector with a 1 on the end, and M is (OpenGL):

$$\begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This allows for translations to be combined with other kinds of transformations, such as scaling. Scaling a vector involves much the same methods. However, using scalars c_1, c_2, and c_3, for scaling along the x, y and z axes, respectively, the matrix is (OpenGL):

$$\begin{bmatrix} c_1 & 0 & 0 & 0 \\ 0 & c_2 & 0 & 0 \\ 0 & 0 & c_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This can be done with a 3 x 3 matrix, but then it would be incompatible with the translation matrix. Scaling a vector stretches or compresses it along the x, y, and z axes. So far, transformations have been relatively simple. Rotation is not simple.

Unlike the last two matrices, rotation does not have simple matrix. Instead, it has three relatively simple matrices defining rotation about x, y or z axes, and a

matrix defining rotation about an arbitrary axis. While one can arrive at an arbitrary axis through a combination of rotations about the standard axes, that is extremely inefficient from a programming perspective. Frequently, rotation about an arbitrary axis is needed. Rotation about an axis is θ.

X-axis

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

Y-axis

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

Z-axis

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

Rotation about an arbitrary axis *A*, where *A* is a unit vector pointing in the direction of the axis, as derived by Lengyel in his book:

$$\begin{bmatrix} (\cos(\theta)+(1-\cos(\theta)))A_x^2 & (1-\cos(\theta))A_xA_y-\sin(\theta)A_z & (1-\cos(\theta))A_xA_z+\sin(\theta)A_y & 0 \\ (1-\cos(\theta))A_xA_y+\sin(\theta)A_z & (\cos(\theta)+(1-\cos(\theta)))A_y^2 & (1-\cos(\theta))A_yA_z-\sin(\theta)A_x & 0 \\ (1-\cos(\theta))A_xA_z-\sin(\theta)A_y & (1-\cos(\theta))A_yA_z+\sin(\theta)A_x & (\cos(\theta)+(1-\cos(\theta)))A_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Constructing transformations in such a way allows for whole transformation to be computed only once and then applied to as many vertices as needed through simple matrix multiplication (OpenGL, Lengyel).

In most applications, models are loaded from a file. However, there are cases where it is possible to, instead, have the application generate the model. That process is called procedural generation. A simple example would be a program making a cylinder. It would create circles using triangles on the bottom and top of the cylinder, then create the sides of it using the outer corners of the triangles, and hold the whole construction in memory, releasing it upon program exit. In this way, one can define models as *n*-tuples of their properties, where *n* is the number of properties it takes to define it. In the example of the cylinder, the

minimum properties would be radius and height, with level of detail and center also available to be used. In this fashion, models can become vector spaces. Procedurally generating graphics was previously looked down on because processors were not fast enough, making the program appear sluggish. But with (relatively) recent advances in technology, CPU's have gained enough speed to take the load with relative ease, with little notice to the human eye. In practice, people would see no noticeable difference between procedurally generating graphics and loading them from a file. However, it gives an ease of manipulation that is extremely useful.

The field of computer graphics uses linear algebra in conjunction with modern programming practices. Although computing graphics uses more areas of math than linear algebra, it is nonetheless an essential part of the field. The foundations of computer graphics a built from it: from specifying color or defining models and their motion, to procedurally generating graphics, computer graphics is absolutely an application of linear algebra.