Important Math for CAD Designers

CAD programs are just complex implementations of the concepts behind linear algebra and calculus when you boil it all down. Maybe we would want to include some really nice and responsive UI elements as well.

Everything can be defined as a collection of points. A line can be defined as two points, a curve can be defined by a collection of points that have a accompanied knot and weight vectors with them (See more on that in the spline section), and a mesh is a collection of points and a set of associated edges between two points (or a collection of points and lines, which are also just points). Basically, everything is points and sometimes scalar values that go with it.

The real magic is in the definitions and functions that turn those collections of points into objects and complex surfaces. Before we get there, we need to have a solid understanding about what we can do to a point or collection of points from a mathematical stand point. For this we will be borrowing some concepts from linear algebra.

The Point Vector

Ground Work

While we can argue all day about the semantics, a point and a vector represent the same thing if they are describing the same dimensional space. Because we are primarily working in 3D space, I am going to restrict my definitions to 3 dimensions. When describing a coordinate in 3d space we use the nomenclature X, Y, and Z.

If you are a rational human being, you would use the X and Y coordinates to designate lateral and longitudinal space respectively and Z to designate the vertical height.

A point can be described as a vector of a specified length anchored at the origin. Therefore a vector that starts at the origin and has properties of [1.0, 2.0, 3.0] also describes a point a the location [1.0, 2.0, 3.0].

Therefore Point = Vector in this situation which gives us the PointVector.

Wiith that out of the way, we can get into the important things.

Vector Addition

This one is fairly straight forward. You simply break the vector into components, and each component and reassemble into the resultant vector.

$$< a_1, b_1, c_1 > + < a_2, b_2, c_2 > = < a_1 + a_2, b_1 + b_2, c_1 + c_2 >$$

Scalar Multiplication

Scalar multiplication is just about as simple as the addition. It is simply multiplying each component of the vector by a single factor. This can be thought as scaling the vector by that factor as the name might suggest.

$$s* < a, b, c > = < s * a, s * b, s * c >$$

Dot Product

The dot product is a little be more abstract than the previous operations we just learned about. The dot product is an operation that takes in two vectors and returns a scalar value. This value is meant to tell us the angle between the two vectors that we input into the function. Another interpretation of the result of a dot product is that it shows how much of vector A has a component of vector B.

$$a \cdot b = \|\vec{a}\| * \|\vec{b}\| * cos(\theta)$$

An equivalent algebraic definition can be stated as follows.

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z$$

Cross Product

The cross product is used to find a vector that is perpindicular to the two input vectors. This is often used in a cad software when calculating the normal vector to a surface or curve for example. The details of why and how are not super important here, but the implementation is written below.

$$\vec{a} \times \vec{b} = \hat{x}(a_y b_z - a_z b_y) - \hat{y}(a_x b_z - a_z b_x) + \hat{z}(a_x b_y - a_y b_x)$$

Linear Transformations

Ground work

Linear transformations are simply a method in which we can "transform" one vector into a different vector. They come in many different forms with different ideologies behind them, but they can all be applied using the same math given different setups.

The first thing we need to do is come up with a convention for how we are going to represent our 3 dimensions of coordinates. We are gonig to make a 4 dimensional array to represent our 3 dimensional coordinate system. Why, you might ask? Well we'll get there in a second, but the TLDR is that it makes the math more efficient and universal in the following steps. This vector will look something like this.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Now, we need to construct a way to do something to this vector to get an output vector. We are going to use what is called a transformation matrix to do this. A transformation matrix is a 4×4 matrix that can be filled with any value and is intended to be multiplied with our point matrix to get a result out.

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ n & p & q & r \end{pmatrix}$$

Now, this might look like a ton of values to keep track of, but we are going to find that there are a set number of arrangements of these values that we need to manipulate our output the way we want to.

Now we need to multiply these values together to get out output vector, the transformed point.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} u \\ v \\ w \\ 1 \end{pmatrix}$$

Matrix multiplication is a little bit different tha normal multiplication, but the underlying principle is still the same. I will write down the basic structure of how it works so that you can implement it in your code, but not go into detail about why it works this way.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a*x+b*y+c*z+d*1 \\ e*x+f*y+g*z+h*1 \\ i*x+j*y+k*+m*1 \\ 0*x+0*y+0*z+1*1 \end{pmatrix} = \begin{pmatrix} u \\ v \\ w \\ 1 \end{pmatrix}$$

The Identity Matrix

To get a better understanding for what is going on here and why this process works, I like to start by running through one example of the math with the identity matrix.

The identity matrix is actually a matrix that just has all 1's down the diagonal and I believe is required to be a square matrix (num_rows = num_columns).

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now, the reason it gets its name as such is because when you multiply any matrix against it, you get the original matrix back out, as I will demonstrate here.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1*x+0*y+0*z+0*1 \\ 0*x+1*y+0*z+0*1 \\ 0*x+0*y+1*z+0*1 \\ 0*x+0*y+0*z+1*1 \end{pmatrix} = \begin{pmatrix} x*1 \\ y*1 \\ z*1 \\ 1*1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

If you take the time to evaluate this you will get the vector you put into it, right back out again!

The first time someone showed this to me I thought that was kind of astounding. We managed to do so much work to get absolutely nothing back out of it. But there is something staring right at you and is only a small jump away, which we will reveal next.

Scale Transformation

For all of us who have used any kind of computer graphics or CAD software, we are quite familiar with the idea of scaling an object. Really it is the most intuitive transformation, as we have a pretty decent concept of what it means to make something twice as large or twice as small.

To scale something with a transformation matrix, you smiply replace the coefficients in the identity matrix for the first 3 columns with a coefficient that describes the scale you want to scale to and voila, you have a scale transformation matrix.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2*x+0*y+0*z+0*1 \\ 0*x+2*y+0*z+0*1 \\ 0*x+0*y+2*z+0*1 \\ 0*x+0*y+0*z+1*1 \end{pmatrix} = \begin{pmatrix} x*2 \\ y*2 \\ z*2 \\ 1*1 \end{pmatrix} = \begin{pmatrix} 2x \\ 2y \\ 2z \\ 1 \end{pmatrix}$$

Note, that if you wanted to scale in just one dimension, you could simply just scale in the x or y or z direction, without effecting the others. There are a few tricks for constructing this into a usable interface for the end user, but for now we will just leave it at this.

Translation Transformation

Often times we will want to move a point from one location to another. As we discovered in the previous section, you can accomplish this with simple vector addition, however for the sake of keeping our transformation system homogenous we can also set this up as a transformation matrix and use the same matrix multiplication as before.

Let's say we want to move our point 5 units in the x, y and z directions. We can construct a transformation matrix like the one below to do that.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1*x+0*y+0*z+5*1 \\ 0*x+1*y+0*z+5*1 \\ 0*x+0*y+1*z+5*1 \\ 0*x+0*y+0*z+1*1 \end{pmatrix} = \begin{pmatrix} x+5 \\ y+5 \\ z+5 \\ 1*1 \end{pmatrix} = \begin{pmatrix} x+5 \\ y+5 \\ z+5 \\ 1 \end{pmatrix}$$

Rotation Transformation

Often times you will want to describe the location of a point by rotatinig about an axis. This can also be described using a transormationi matrix. We will do this via the use of polar coordinates. For not we will only consider how to do this about the z-axis, but this transformation could be generalized to work about an arbitrary axis.

A quick refresher on polar coordinates. Basically, you can define any point in 2D space by addressing the location by an angle about a circle θ from 0 to 2π and r which is the distance from the origin. The x coordinate will be defined as the cosine of the angle and the y coordinate will be defined as the sin of the angle.

$$x = r * cos(\theta)$$

$$y = r * sin(\theta)$$

We can extend this definition into our transformation matrix. For now we will just define this for rotating about the z-axis.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x*cos(\theta)+y*-sin(\theta)+z*0+1*0 \\ x*sin(\theta)+y*cos(\theta)+z*0+1*0 \\ x*0+y*0+z*1+1*0 \\ x*0+y*0+z*0+1*1 \end{pmatrix} = \begin{pmatrix} x*cos(\theta)-y*si \\ x*sin(\theta)+y*co \\ z \\ 1 \end{pmatrix}$$

Reflection Transformation

Reflection or mirror transformations are also a pretty common way to want to manipulate an object in a CAD software. This one is actually technically the same as scaling an object in a single dimension, but just with the added constraint of keeping the factor the same with an opposite sign.

Let's say you wanted to reflect your object about the x-axis. This is basically like scaling the x dimension by a factor of -1 about the origin. That looks like this in a transformation matrix.

$$\begin{pmatrix}x\\y\\z\\1\end{pmatrix}\cdot\begin{pmatrix}-1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}=\begin{pmatrix}-1*x+0*y+0*z+0*1\\0*x+1*y+0*z+0*1\\0*x+0*y+1*z+0*1\\0*x+0*y+0*z+1*1\end{pmatrix}=\begin{pmatrix}x*-1\\y*1\\z*1\\1*1\end{pmatrix}=\begin{pmatrix}-x\\y\\z\\1\end{pmatrix}$$

Projection Transformation

Something that comes in handy often when working in 3D space in a CAD software is the concept of a projection onto a surface. A projection is basically collapsing a dimension of the coordinate system to get a result that could be defined by a coordinate system with one less dimension. This is often useful when you want to make 2d drawings out of you 3D objects.

Let's say we want to project our point onto the xy-plane. To do this all we need to do is to remove any z component to our point vector. This once again can be generalized to be able to project on an arbitrary plane, but for now we will keep it simple.

$$\begin{pmatrix}x\\y\\z\\1\end{pmatrix}\cdot\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&1\end{pmatrix}=\begin{pmatrix}1*x+0*y+0*z+0*1\\0*x+1*y+0*z+0*1\\0*x+0*y+0*z+0*1\\0*x+0*y+0*z+1*1\end{pmatrix}=\begin{pmatrix}x*1\\y*1\\z*0\\1*1\end{pmatrix}=\begin{pmatrix}x\\y\\0\\1\end{pmatrix}$$

Generalized forms

***Note that this part of the document is still under construction.

The examples from the previous section were primarily foor the purpose of comprehension of the techniques. What we want to implement requires much more generalized forms of these things if we want to give the user tools that are easy to use at an arbitrary point in space, with an arbitrary construction plane, from an arbitrary point of reference, etc.

Translation

Scale

Rotation

Projection

Reflection