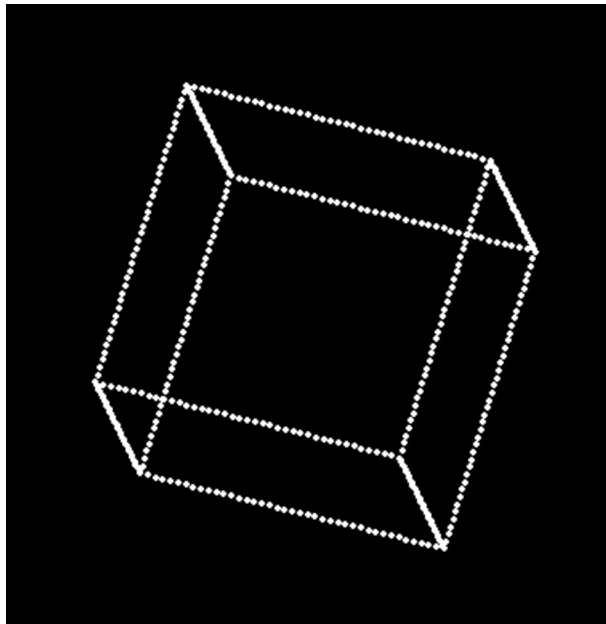Analysis H Project:
3D Graphics Generation with Matrices

Mihir Rao

April 4, 2021

Overview

For our Analysis H project, Steve and I decided to dive into the realm of computer graphics and matrices' applications in this field. Specifically, we wrote code that generates 3D points, projects these 3D points onto a 2D surface(the computer screen), and transforms these points to display an animation(using matrices). We both were fascinated by the numerous applications of matrices and thought this will be a fun project to work on. After doing further research, it also turns out that Renderman, Pixar's animation software, is heavily based on matrices and linear algebra; so this was the perfect time for us to dip our toes in the fundamentals of this field. This document serves to explain how we came up with the math of this code, challenges we faced, and the ultimate solutions to those challenges. The next section is a brief introduction of how we approached the code and following that is the actual math behind it. Enjoy!

The Math

There are many aspects of the code that we had to come up with in order to display an animated 3D object on a 2D screen. Some of these aspects rely more on math than others and this document will go over those functions in this code.

Coordinate Setup & Matrices

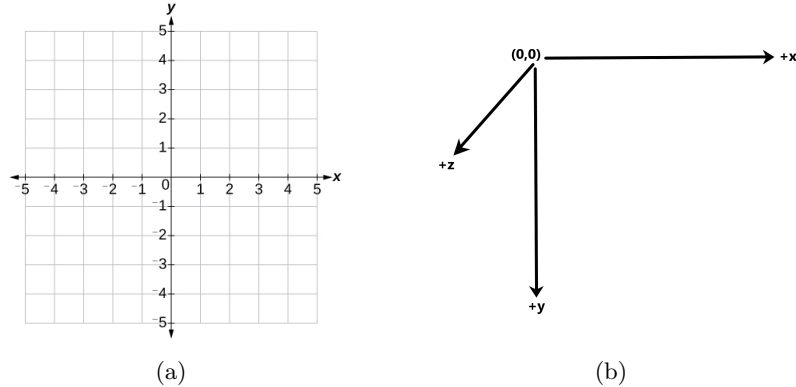(a)   (b)

Figure 1: Coordinates

In the figures above, we see (a): our regular cartesian coordinate system where (0,0) is at the center of the screen and we have 4 quadrants. However, in figure (b), we see a generic computer coordinate system–where (only the 4th quadrant is included making the origin the top left of the screen. Though it may seem more complex to display animations at a point other than the origin, computer screen layouts actually make this problem easier(with a little bit more math of course) since we no longer have to deal with negative values. So why is this important? Well, if we wanted to display an object at the center of the screen, or in fact, any point other than the origin, we'd have to come up with some sort of way to cleverly use matrices and transformations. We'll start of with our generic 2 x 2 rotation matrix.

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

A couple things that are important to note before we proceed are that this matrix rotates points in 2 dimensions and on the xy plane–meaning around the z axis. Since we need to go into 3D, we need to change this matrix to work for 3D points and we can do that by using the concept of an identity matrix. Shown as the top right matrix, we can multiply the rotation matrix by this and still obtain the same rotation matrix. A 3x3 identity matrix is the just the same and we can use that concept to come up with a rotation matrix for 3D. These new matrices are shown below.

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

2

Just like we can add on another row and column to the identity matrix, to get the matrix for the new dimension, we can do the same for the rotation matrix. To prove this works, we can take any point and verify its coordinates using both matrices.

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

$$\begin{bmatrix} cos(\frac{\pi}{2}) & -sin(\frac{\pi}{2}) & 0 \\ sin(\frac{\pi}{2}) & cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} cos(\frac{\pi}{2}) \\ sin(\frac{\pi}{2}) \\ 0 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{3}$$

As we can see above, we get the correct answer of $(0, 1, 0)$ after rotating the point $(1, 0, 0)$ by $\frac{\pi}{2}$ radians counter clockwise. Similarly, we can do this to the rotation matrices of not only the z axis but also the y and x axes. Once we do this, we get the 3 rotation matrices for the x, y, and z axes.

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

Rotation Matrix X

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

Rotation Matrix Y

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

Rotation Matrix Z

(NOTE: Derivations of these matrices have been left out to keep paper length minimal). Now that we have our 3 main matrices, we can go back to the problem of a computer's coordinate system. If we were to use just these matrices themselves, our points would be rotating about the origin and the center of the cube would not stay fixed. So how might we solve this problem? One way to solve this problem is using translation matrices. First, we translate the entire cube to the origin, rotate it at the origin, and then translate the entire cube back by using the inverse of the first translation matrix. To create these translation matrices, we need to refer back to Figure 1(b). Since, once again, (0,0) is the top left of the screen, the center would be the point $(\frac{w}{2}, \frac{h}{2}, \frac{d}{2})$ where w, h, and d are the width, height and depth of the screen, respectively. Now that we know how much to translate by, we need to figure how to write these matrices.

$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \qquad\qquad \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix}$$

Above, we have a translation matrix(left) that translates by x units in the x direction and y units in the y direction, followed by its inverse(right) that translates by $-x$ units in the x direction and $-y$ units in the y direction. However, recall that to translate a 2D point, we need to go into 3D. These matrices above only translate in 2 dimensions–the x direction and the y direction–but that's not what we want.

We want to translate a 3 dimensional point and for that, we must go into 4D. The new translational matrices are shown below.

$$
\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}
\qquad\qquad
\begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Translation Matrix 1   Translation Matrix 2

We can prove that these matrices do their job by applying both of them onto the point $(\frac{w}{2}, \frac{h}{2}, \frac{d}{2})$ to see if we get the same point back. Since we want to translate to the origin first, we'll perform the transformation $T_2 \circ T_1$.

$$
\left(\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} \frac{w}{2} \\ \frac{h}{2} \\ \frac{d}{2} \end{bmatrix} = \begin{bmatrix} \frac{w}{2} \\ \frac{h}{2} \\ \frac{d}{2} \end{bmatrix}
$$

$$
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{w}{2} \\ \frac{h}{2} \\ \frac{d}{2} \end{bmatrix} = \begin{bmatrix} \frac{w}{2} \\ \frac{h}{2} \\ \frac{d}{2} \end{bmatrix}
$$

(4)

Points Between Function
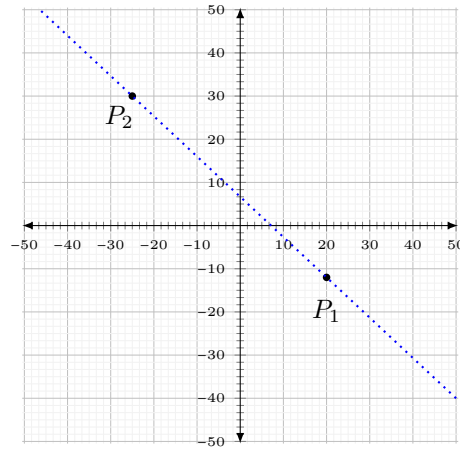
Above, we have a graph of 2 points: P_1 and P_2. Given these two points, the pointsBetween function aims to return a list of n number of points that lay on the line formed by P_1 and P_2 (as shown in blue). To do this, we'll define a few variables: Let n = the number of points we want(given), $\vec{v} = \overleftrightarrow{P_1 P_2}$ and $\vec{w} = \frac{\vec{v}}{n}$. Like stated, $\vec{w}$ is a scaled down version of $\vec{v}$ which will mainly be used to generate these points, but more on that in a second. For now, we'll calculate some of the values for these variables and then put them together to solve this problem. In the above diagram, $P_1 = (20, -12)$ and $P_2 = (-25, 30)$.

4

$$n = 5$$
$$\vec{v} = \; < x_{P2} - x_{P1}, y_{P2} - y_{P1} >$$
$$\vec{w} = \frac{\vec{v}}{5}$$

$$n = 5$$
$$\vec{v} = \; < -45, 42 >$$
$$\vec{w} = \; < \frac{-45}{5}, \frac{42}{5} >$$

Now that we have $\vec{w}$, we can repeatedly add it to P_1 and take note of all the points it takes us to. For example, if $P_1 = (0,0)$, $P_2 = (0,10)$, and $n = 10$, the value of w would be $< \frac{0}{10}, \frac{10}{10} >$ or $< 0, 1 >$. Now, we would start off at P_1 and add w giving us the point $(0,1)$. Adding it again and again, we'd get $(0,2)$, $(0,3)$, $(0,4)$,..., and $(0,10)$ which are our generated points! To tell our code when to stop adding $\vec{w}$, we can compare the newly generated point to P_2(our destination). Once a generated point $g + \vec{w} = P_2$, we know to stop. This idea is visually represented in Figure 2(b) through vectors.
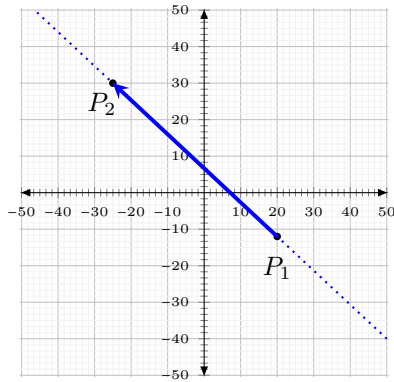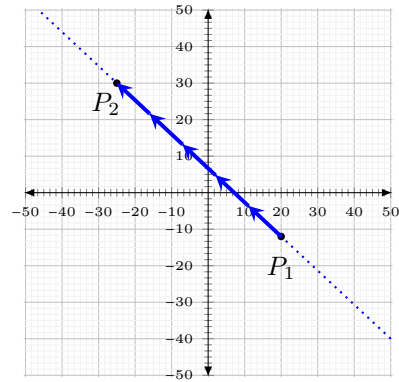

Figure 2(a)

Figure 2(b)

But hmm, this is in 2D and not 3D. So how would we generate points just like we did but in a 3 dimensional space? Well, it turns out that the process is exactly the same. The only thing that would differ is that our points and vectors would have one more parameter: z. For example, $\vec{v}$ might look like $< x, y, z >$ instead of just $< x, y >$ and $\vec{w}$ might look like $< \frac{x}{n}, \frac{y}{n}, \frac{z}{n} >$ instead of just $< \frac{x}{n}, \frac{y}{n} >$. However, we can most definitely still use the concept of scaled vectors to solve this problem.