Simulating Video Game Worlds using Linear Algebra

Tanner Brown

Abstract*—*Linear Algebra, being such a broad category of mathematics, has a notably large number of applications. However, possibly the most literal example in computer science is through simulating actual 2 dimension and 3 dimensional spaces, such as flight simulators, or video games. This paper investigates several components of linear algebra and how they are used to effectively accomplish these simulations.

Index Terms— Linear Algebra, Game Development, Simulated Space

I. INTRODUCTION

THE most commonly used, and therefore the most important apsects of linear algebra that are applied in simulated spaces, are use of Vectors and Matrices.

The first section will cover vectors and their various uses in determining direction, position and distance. The first section will also explore various applications for these vectors in simulating spaces, as well as several simple formulas on how to apply them.

The second section will introduce matrices, and how they can be used both as a collection of vector data, or simply as vector graphics that can be manipulated. The focus of this section will be on linear transformations that can be applied to a matrix, and how that information can be used in various ways. One such discussion, is how rotation transformations could be applied on an aircraft to determine its place and direction in a 3D space.

This paper will conclude with an example on movement and rotation using matrices and a rotation matrix, as is done in the classic game Tetris®.

This paper primarily focuses on the theory of linear algebra when used for game development and shows how understanding the principles before programming can allow a programmer to reduce work and increase efficiency through use of simple formulas.

II. VECTORS

"It cannot be overstated how important vectors are in the game programmer's toolbox; most programmers in the industry use them every single day" [1]

A. Position Vectors

Possibly the most used aspect of linear algebra that is applied to simulating spaces is the vector. The simplest use of the vector in simulating a 3D development would be to identify the position of an entity within that environment. Using a vector without consideration for its magnitude, to identify its direction is often called by various names such as the position or coordinate vector.

Using Psuedocode, we can define a position vector:

Vector **A** = new Vector(x:1, y:1, z:0)

It would be intuitive to assume that in relation to the origin, the entity at **A** is 1 unit away in the x,y directions, and directly on the origin in the z direction.

Through simple vector operations, we can minipulate the position of this game entity in numerous ways. To use as an example, let's define a second position vector:

Vector **B** = new Vector(x: -2, y: -2, z: 0)

Let's assume our entity at position vector **A** wanted to move the position vector **B**. All we would need to do is subtract B from A to get the following vector

Vector **C** = new Vector(x: -3, y: -3, x: 0)

The result is vector **C**. Considering its magnitude, the entity at position **A** would follow this vector from its location reach it's destination at position vector **B**. Using the same logic, if the entity at vector **A** moved along the vector **C**, it would arrive at the position of vector **B**.

B. Direction

Finding the direction, a game entity is facing, or the direction of a game entity in relation to another entity is another problem

.

that is easily tackled with what are called direction vectors or unit vectors.

We will use the previously defined vectors **A**,**B**,**C** as examples. If the entity at position vector **A** wants to know the direction it needs to be facing to look at position **B**, it could just apply the direction vector of **C**. This direction vector is generally found by applying a simple formula:

$$\vec{v} = \frac{v}{|v|}$$

Where v are the vector coordinates and |v| is its magnitude.

Assume we are programming a flying simulator, and we need to determine an Airplane's direction of flight on all 3 axes. We can assign three vectors to represent the three possible directions each flight surface applies to [fig 1]. However, an aircraft tends to only travel in one direction, with only minor changes in the other two directions. We can use the 3 direction vectors apply movement in the 3 directions. However, by knowing the direction of the vectors, we can also use those values to calculate the Airplane's pitch, yaw, and roll angles. This will be revisited on the topic of linear rotation.

C. Magnitude

The third useful component to a vector that is heavily used in 3D space is magnitude. Magnitude can often be used to identify the speed at which an object moves from one place to another, or distance traveled in a direction.

Using the previously defined vectors **A**,**B**,**C** as examples, if we remove the Magnitude from vector **C**, we get a direction vector that can be used to tell the entity at vector **A** which direction to move in. Likewise, if we want to know if an entity can make it to a destination in a certain amount of time, you could calculate its magnitude and compare that with the max distance it can travel over that time.

One great application of magnitude is to identify the places of cars in a race [figure 3] . In this example, we can create vectors between the cars **A** and **B** and the goal **G**, we can use the magnitudes of the resulting vectors **AG** and **BG** to determine their place in the race. Sorting the magnitudes of the vectors of all cars in relation to the goal G will then give us a list of all race cars and their order in the race.

$$\left|\left|\overrightarrow{AG}\right|\right| = \sqrt{\overrightarrow{AG}.x^2 + \overrightarrow{AG}.y^2 + \overrightarrow{AG}.z^2}$$

The above formula is how we find the magnitude of a vector. However, in the racing example above, if the actual distance from each car to the goal isn't needed, computer resources can be saved by simply not calculating the square root of each calculation. Let's define this distance as D

$$D = \overrightarrow{AG}.x^2 + \overrightarrow{AG}.y^2 + \overrightarrow{AG}.z^2$$

By not calculating the square root value of each vehicle's magnitude we don't get the actual magnitude of each vector. However, the values can still be used to compare against eachother, and by not calculating the square values of each vehicle's magnitude, we have potentially saved the computer's resources[1].

The example above is one that is worth noting as a reason why understanding Linear Algebra is so important for Video Game development. By understanding Linear Algebra and how it's functions work, an experienced programmer can modify these formulas in ways that allow them to optimize the computer's performance, while still getting the desired result.

III. MATRICES

"Through the shortcuts derived from it's theorems, computations required to calculate every single vertex in a 3D graphic becomes much easier, allowing for significantly less costly operations." [2]

Matrices, like vectors, play a crutial role in Video Game development, especially in 3D space. One crucial aspect to Linear Algebra in 3D game development are 3D/Vector graphics.

Due to the large amount of resources available on vector graphics, as well as the broad scope of the topic, it will only be mentioned briefly and not as the focus of this section.

This section will discuss the types of linear transformations that can be applied on a matrix, and why those are specifically useful for game development.

A. Scaling and Translation Transformations

Scaling is a type of linear transformation that can be used to adjust the length of a vector or matrix in a direction.

One simple example of this being performed on a vector would be if an entity in a racing game aquired a boost. The speed of the boost could then be used to scale the vehicle's direction vector to increase it's speed by a factor.

In 3D space however, a scaling matrix transformation can be used to adjust the scale of a group of vectors by a common factor. For example, if the game's camera was to zoom in or out, you could scale all vectors by the camera's zoom level to ensure their distance in relation to eachother remained constant.

This concept of scaling vectors in relation to eachother is used in vector graphics to properly scale an image dynamically. This is very powerful because it allows for

[1] This formula is so commonly used, that some processors and programming languages compute the formula faster with the square root in place than without.

images or graphics to display at virtually any size display or 3D space with very little computation required.

Translation transformations on matrices work like scaling transformations, except computations are done through adding instead of multiplying. This transformation allows for unified movement of a collection of vectors, a collection of entities, a single vector graphic containing multiple vector graphics, etc. Using the camera example, transformations could be applied to move the positions of the entities on the screen as the camera pans across the simulated space.

B. Rotation Transformations

Rotation transformations are slightly more complicated and invove trigonometry. The primary use of linear rotations in 3D rendering is orienting an obejct model, defined in a fixed orientation around its own local coordinate system.

Going back to the Airplane example [figure 1], we would use translation to position the Airplane within its world, and rotation would be used to simulate its pitch, yaw and roll [figure 2]. A linear rotation along its fuselage (body) of the aircraft will calculate it's roll. A linear rotation that points the nose up or down will adjust the pitch of the aircraft, and rotating the nose left or right will calculate its raw rotation.

Figure 1 shows two separate ways these calculations can be made. The first option is through extrinsic rotation, which is a rotation along a fixed coordinate system. In this case, the aircraft rotations in relation to it's surrounding environment. This would make the most sense in a flight simulator that would benefit from tracking the aircraft's relation to the ground. This would allow forces of gravity and drag (wind resistance), which are an important part of properly simulating flight, to be applied easily to the aircraft.

The second option, intrinsic rotation, applies rotation in relation to the current orientation of the object. This sort of rotation would be useful in a space or underwater type simulation. This is due to the fact there is less, or potentially no need to calculate forces such like gravity, or a fixed point to keep the model related to, such as the ground. When relation to a plane such as the ground or gravity isn't necessary, rotation would be most effective when applied in relation to itsself.

C. Tetris

The airplane example shows how powerful transformations can be at solve something that might be otherwise very complex. However, they can also be used to quickly solve something simple. Probably the simplest and most relatable example of linear transformations in game development, is through their use in the classic game Tetris.

In Tetris, each Tetromino (also known as a Block), consists of 4 squares arranged in different orders. Assuming we are programming a game of Tetris, we could define a block as such:

```
var Block = []
Block[0] = new Vector(0,0)
Block[1] = new Vector(1,0)
Block[2] = new Vector(2,0)
Block[3] = new Vector(3,0)
```

This Block, being an array of vectors (as a matrix is), would be the ***I*** Block, that is the block that is 4 squares in a row. In this case, arranged horizontally.

Moving this Block around in space would be relatively easy [figure 4]. Moving left to right, a programmer could simply increase or decrease the X-value of each Block by 1. As the Block moves down, the Y-value of each Block would increase by 1.

Rotating a block, however, is where linear algebra saves quite a bit of computation. If a programmer was to neglect to use Linear Algebra for this computation, one might have to iterate through each of the 4 blocks, determine its new position based on which direction it is rotating, and apply those changes. The programmer may have to explicitly write instructions for each block type and where they would have to end up.

```
ROTATION = {v1:[x:0 , y:-1], v2:[x:1, y:0]}
```

However, if we define a rotation Matrix like the one above, we can apply this rotation on the Bock arra. It's mathematicall formula would be as follows:

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0123 \\ 0000 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & -2 & -3 \end{bmatrix}$$

By simply multiplying each number in the Block array by the right rotation matrix, each square in the Block was given the proper position vector to align it vertically. This same logic can then be applied for each individual block or applied in the opposite direction for a counter clockwise rotation[2].

IV. Conclusion

Given the examples above, it is clear how important vectors, matrices and linear transformations are to simulating 3D space. By understanding these functions, and how they can be used, a programmer can find ways to drastically reduce computation time and enhance performance of the simulated space.

[2] This method, known as the Super Rotation System (SRS) is considered the currently endorsed rotation method by the Tetris Company [4].

V. References

[1] S. Madhav, "Game Programming Algorithms and Techniques," Pearson Education, Inc., 2014, p. Ch 3.

[2] B. Will, *3D Transformations,* 2013.

[3] N. Hall, "Scalars and Vectors," 05 May 2015. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/vectors.html.

[4] 2009 Tetris Design Guideline, Blue Planet Software, Inc., 2009.

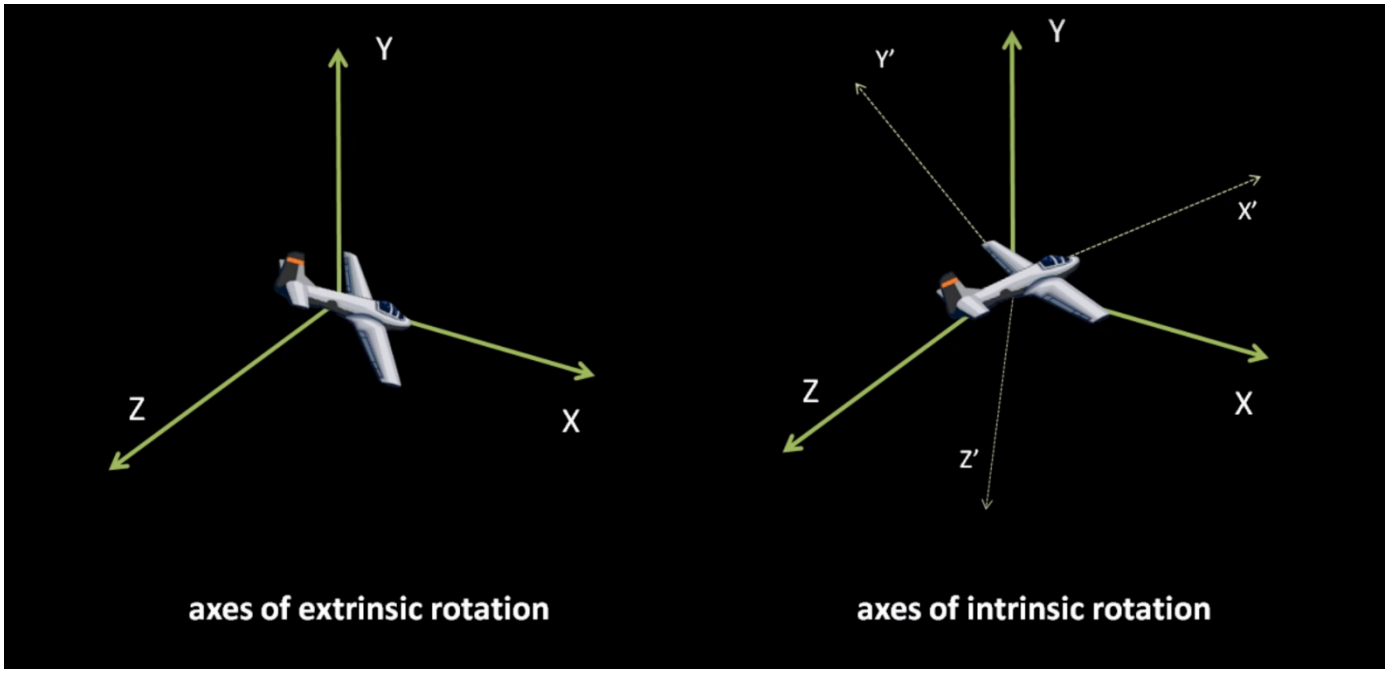


Figure 1 – Airplane

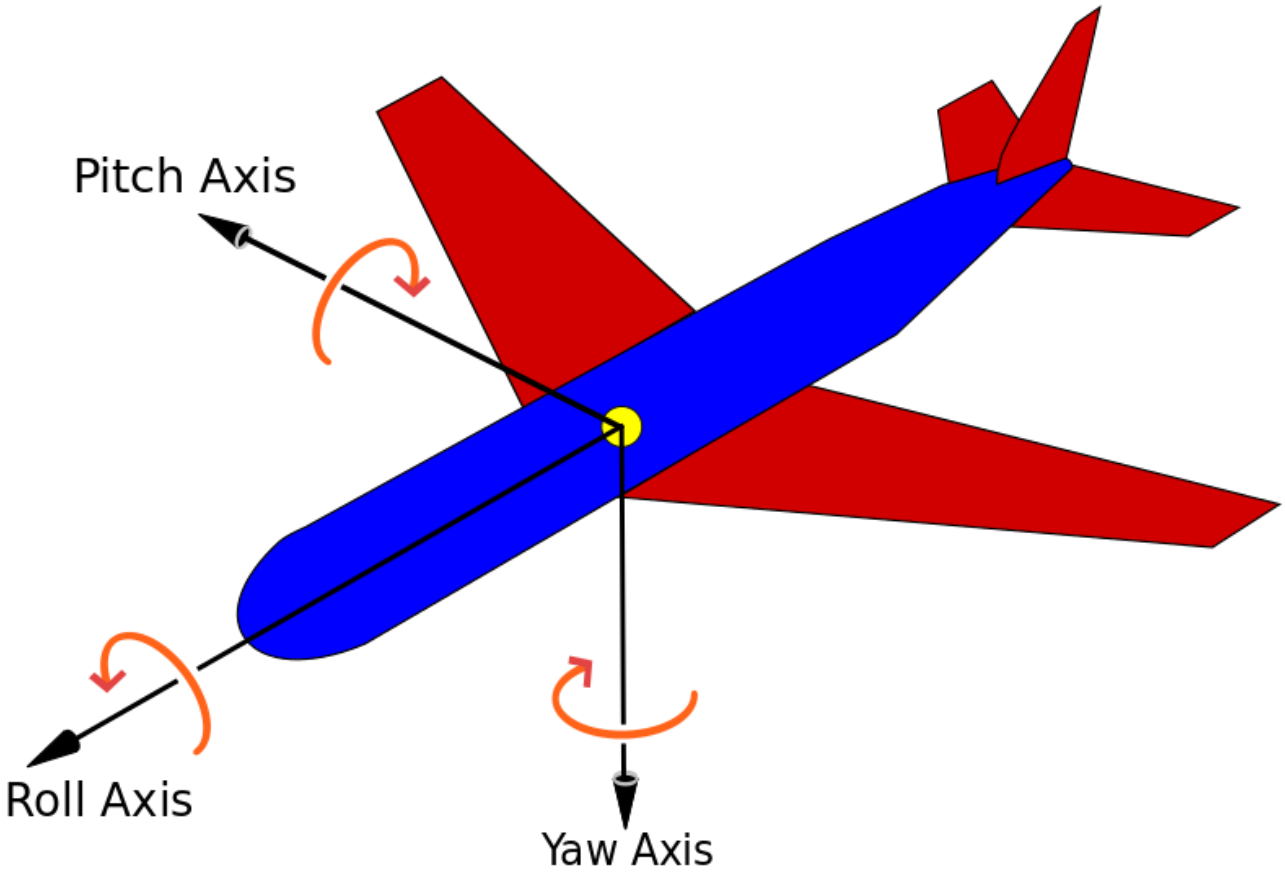
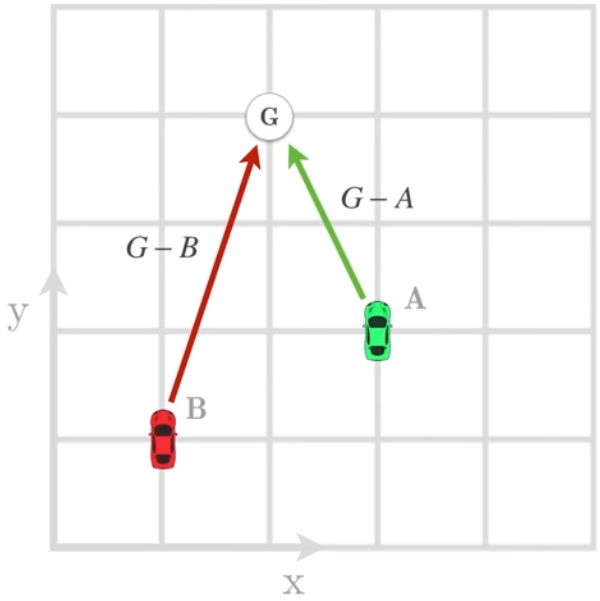


Figure 2 – Pitch , Yaw, Roll Axis of rotation



$$\sqrt{x^2 + y^2}$$

IN THE LEAD

A

Figure 3 – Calculating race positions

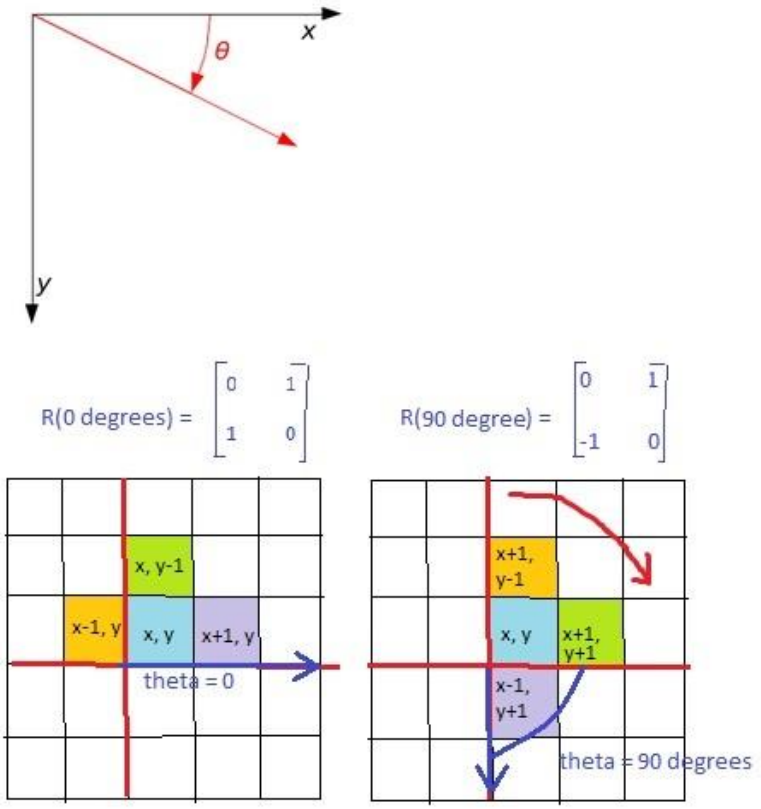


Figure 4 – Tetris Movement and Rotation Example