Methods of Rotation in Computer Graphics
Math 22A Final Project

Madeleine Hung and Jasai Martinez

December 2022

Contents

1 Cover Letter

We are both taking CS50, so we were really interested in looking into an intersection of Computer Science and Linear Algebra. We realized that in CS50 we had only scratched the surface of what goes on behind the scenes in a computer to display 3D images on a 2D screen.

After meeting with Dusty, he told us to narrow down our focus to fewer topics, so that we could explore our topic in more depth. So, we decided to focus on the parts of our project that interested us, which was comparing different methods of rotation. We also received feedback peer reviews to write clearer definitions for some important terms such as Euler angles, so we added more precision through definitions from textbooks. Amy also recommended we use more images to help visualize the content, so we incorporated more visual examples as well. Our peer reviewers also suggested we should focus on more specific examples, so we focused on providing at least one example for every definition or method we described.

After our second meeting with Dusty, he instructed us to focus and condense our introduction and work on our transitions to increase the overall flow of the paper. We cut two sections about perspective because the paper was getting very long, and focused more on including three methods of rotation. He helped us with some formatting issues (using periods at the end of definitions, formatting with math symbols). He also helped us differentiate between definitions and theorems, so we renamed some of our definitions as theorems that we proved, and we renamed some theorems as notes or definitions. We also added more to the Euler angles section, including more graphics to help with visualization and understanding.

In working on our project, we divided up the work. I (Mattie) worked more on the earlier stages of the project planning and research, writing the cover letter, introduction, the 3×3 matrix section, editing and designing figures. Meanwhile, Jasai worked more on the later sections researching and writing up Euler angles and Quaternions, the bibliography, and working more on formatting in Latex.

We're very excited for you to read our paper! We hope you enjoy it and learn something about Linear Algebra and computer graphics!

2 Introduction

The purpose of our project is to explore different methods of rotation for use in computer graphics. Rotation is very important in computer graphics because it allows us to move objects in 3D space. This is part of what makes video games and animation seem more realistic, it seems like objects are existing and interacting with a 3D environment even though they only exist on our 2D computer screen.

It is important to understand the different methods of rotation because each might encounter its own difficulties in computer graphics. Computers have limited memory and limited storage, so it is important to be as efficient as possible.

The simplest way to rotate coordinates is using a 3×3 rotation matrix, which will allow us to rotate our object around any of the x, y, or z axis. But we don't want to be limited to rotating around just the traditional x, y, or z axis; often it seems better to rotate an object around its own axis.

But how can we derive a formula for a rotation matrix that will rotate an object? How can we perform multiple rotations? How can we rotate an object around its own axes, and not just the traditional x, y, and z axes? And how can we maximize efficiency to perform many rotations for computer graphics?

We hope to answer these questions by exploring three different methods of rotation: 3×3 rotation matrix, Euler angles, and quaternions.

3 Rotation with the 3 × 3 matrix

The simplest way to rotate an object in $\mathbb{R}^3$ is by using a 3×3 matrix that rotate an object about an axis. Before we look at rotation in $\mathbb{R}^3$, let us recall the rotation matrix in $\mathbb{R}^2$.

Theorem 1. *Let (x, y), be a point in $\mathbb{R}^2$, then the matrix $\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$ rotates (x, y) counterclockwise by angle θ.*

Suppose we want to rotate a point (x, y) counterclockwise to a new point (x', y') as shown below.

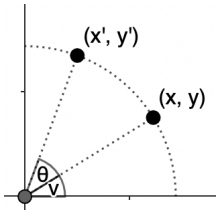

Figure 1: Rotation in $\mathbb{R}^2$

If we express these points in polar form, we find

$$\begin{aligned} x &= rcosv \\ y &= rsinv \\ x' &= rcos(v + \theta) \\ y' &= rsin(v + \theta). \end{aligned} \tag{1}$$

Then we can expand using trigonometric identities to find

$$\begin{aligned} x' &= r(cosvcos\theta - sinvsin\theta) \\ y' &= r(sinvcos\theta + cosvsin\theta). \end{aligned} \tag{2}$$

Substituting in values from above, we find

$$\begin{aligned} x' &= xcos\theta - ysin\theta \\ y' &= ycos\theta + xsin\theta. \end{aligned} \tag{3}$$

Now we can represent these results in a $2\,x\,2$ matrix to find

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

3.1 Example: Rotating around the x axis

Now, let's look at example of how we can rotate our coordinates (x, y, z) in $\mathbb{R}^3$ about the x axis. In the previous section, we derived a 2×2 rotation matrix which would rotate a point (x, y) by θ degrees, but now we want to rotate in three dimensions. We can see that in rotating (x, y, z) about the x axis, the x coordinate will stay the same. Thus, we are only changing the point (y, z) to (y', z'). So our 3×3 rotation matrix to rotate about the x axis looks very similar to our 2×2 rotation matrix.

We define a rotation of θ radians about the x axis.

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

We see that multiplying $R_x(\theta)$ and arbitrary coordinates (x, y, z) in $\mathbb{R}^3$ yield the following result.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ ycos\theta - zsin\theta \\ ysin\theta + zcos\theta \end{bmatrix}$$

We can see as discussed above that the x coordinate does not change, and that the y and z coordinates resemble the values we calculated in our derivation of the 2×2 matrix. Now we have found a rotation matrix in $\mathbb{R}^3$, and we can use this matrix to rotate a point about the x axis. Rotating about the y and z axis is very similar: for rotating about the y axis the y coordinate is held constant, while rotating about the z axis, the z coordinate is held constant.

But if we wanted to rotate an object in multiple directions, say about the x, y, and z axes, we would need to multiply the object by each matrix in succession. This process is time consuming and inefficient for computer graphics where memory and efficiency are of utmost importance. We are also still limited by only being able to rotate around the x, y, or z axis. What if we want to rotate our object about an arbitrary axis, not our traditional x, y, or z axis?

4 Rotation with Euler Angles

Rotations of objects in $\mathbb{R}^3$ consist of three independent axes, where each axis rotation accounts for a different degree of freedom. Rather than simply rotating along one axis, we can apply one rotation matrix with **Euler angles** and rotate an object around the x, y, z axes or an object's own axes.

Definition 4.1: The **Euler angles** α, β, γ, express the rotation of an object in its own coordinate system, given as individual rotations about the x, y, and z. [4, p. 356]

Euler angles are easier to visualize on the xyz-plane.

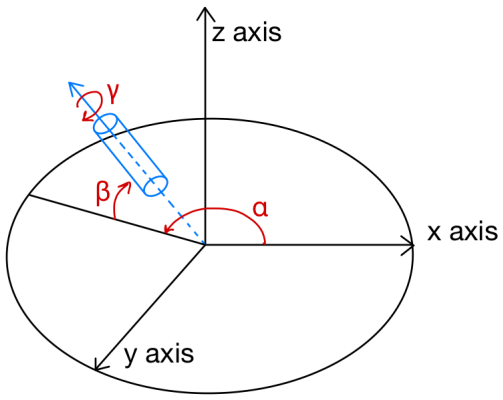

Figure 2: Euler angles (α, β, γ) shown as different angles of rotation.

Euler angles are typically denoted as α, β, γ. See how in the figure above, we can rotate in the α, β, or γ direction, independent of the x, y, and z axes. Since we want to rotate in all three directions, instead of one matrix from Section 3, we will need three matrices to represent the three directions of rotation given by our Euler angles. We define the following matrices to represent rotating in the α, β, or γ direction (Recall we derived a 3×3 rotation matrix in the previous section).

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{bmatrix} R_y(\beta) = \begin{bmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{bmatrix} R_z(\gamma) = \begin{bmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now we want to combine the three matrices above into one matrix. So, we multiply all three matrices to yield the following matrix:

$$R_{xyz}(\alpha, \beta, \gamma) = R_x(\alpha) R_y(\beta) R_z(\gamma)$$

$$= \begin{bmatrix} cos\beta \cos\gamma & -cos\beta \sin\gamma & sin\beta \\ cos\alpha \sin\gamma + cos\gamma \sin\alpha \sin\beta & cos\alpha \cos\gamma - sin\alpha \sin\beta \sin\gamma & -cos\beta \sin\alpha \\ sin\alpha \sin\gamma - cos\alpha \cos\gamma \sin\beta & cos\gamma sin\alpha + cos\alpha sin\beta \sin\gamma & cos\alpha \cos\beta \end{bmatrix}.$$

Our matrix now gives us an efficient way to describe a rotation using three precise angles. It is also important to note that the order with Euler angles is important because each rotation will have a cumulative effect on each other. $R_{xyz}(\alpha, \beta, \gamma)$ represents a rotation beginning with the z-axis and ending with the x-axis. Changing the order of multiplying matrices will result in a different rotation matrix.

Note! Euler angles are **NOT** always commutative.

Proof. Let $R_x(\alpha)$ and $R_y(\beta)$ be two matrices representing α, β Euler matrices, respectively.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{bmatrix} R_y(\beta) = \begin{bmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{bmatrix}$$

Then it follows that:

$$R_x(\alpha)R_y(\beta) = \begin{bmatrix} cos\beta & 0 & sin\beta \\ sin\alpha \sin\beta & cos\alpha & -sin\alpha \cos\beta \\ -sin\alpha \cos\alpha & sin\alpha & cos\alpha \cos\beta \end{bmatrix}$$

$$R_y(\beta)R_x(\alpha) = \begin{bmatrix} cos\beta & sin\alpha \sin\beta & sin\beta \cos\alpha \\ 0 & cos\alpha & -sin\alpha \\ -sin\beta & sin\alpha \cos\beta & cos\alpha \cos\beta \end{bmatrix}.$$

By inspection, $R_x(\alpha)R_y(\beta) \neq R_y(\beta)R_x(\alpha)$. □

4.1 Example: Rotating a matrix with Euler Angles

Now let's look at an example where we use Euler angles to rotate an object. Suppose we want to rotate only the β angle by 90°. Our resulting matrix will leave us with:

$$R_{xyz}(\alpha,\, 90°,\, \gamma) = \begin{bmatrix} 0 & 0 & 1 \\ sin(\alpha+\gamma) & cos(\alpha+\gamma) & 0 \\ -cos(\alpha+\gamma) & sin(\alpha+\gamma) & 0 \end{bmatrix}.$$

But now we have run into a problem! Our α and γ rotations have lost independence, and we can see that changing α or changing γ will rotate an object in the same direction. We went from having three degrees of freedom to two degrees of freedom, effectively allowing only two directions of rotation rather than three.

4.2 Gimbal Lock

This particular outcome of using Euler angles is known as Gimbal Lock. This is a BIG problem that will cause complications and ruin future rotations once gimbal lock occurs.

Definition 4.2: Suppose the rotation axes α, β, and γ, start orthogonal to each other. **Gimbal lock** occurs when a single axis is rotated by 90° which leads to two axes existing in the same plane and a degree of freedom is lost. [4, p. 357-358]

Example 4.2: A 90° rotation of β causes α and γ to become parallel. Now α and γ rotations will lead to the same result – locking α and γ orientations.

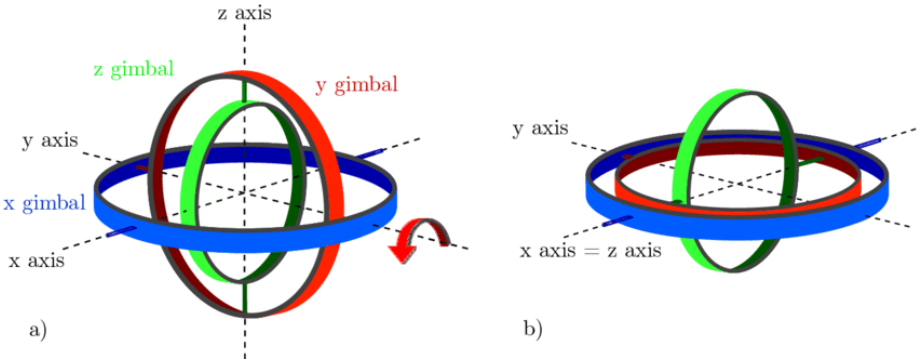

Figure 3:
An orthogonal Euler angle orientation(a) turning into Gimbal lock after a 90° rotation about the y-axis(b). [5]

Gimbal Lock can be difficult to understand without a clear visual representation. Figure 3 utilizes rings to represent the different axes of rotation. Notice how prior to any rotation, a 3D object has three degrees of freedom (the ability to rotate α, β, γ). Recall that Euler angles cumulatively affect each other, so once the β angle is used to rotate the object by 90° about the y-axis, we have also changed the orientation of the other axes. We can see in Figure 3, the x-axis and z-axis are parallel and "locked" together.

Our α and γ rotations were supposed to lead to different directions of rotation and different orientations of our object, but now because of gimbal lock, rotating by α or γ will cause the same rotation. We can now only rotate in two directions and we have effectively lost a degree of freedom! In computer graphics this is a HUGE problem because now it limits our ability to rotate an object freely in the xyz-plane. Thus, Euler angles are often not the first choice in computer graphics because getting around Gimbal Lock is difficult and time consuming.

If Euler angles have these fault cases, then what method is truly best for 3D rotations?

5 Rotation with Quaternions

Sir William Rowan Hamilton luckily created a system in which vectors similar to matrices could be used algebraically to represent geometric shapes. **Quaternions** are an additional method for showing rotations in $\mathbb{R}^3$.

Definition 5.1: A **quaternion** is a 4D extension from complex numbers represented with the equation:
$q = w + xi + yj + zk; w, x, y, z \in \mathbb{R}$ and $i^2 = j^2 = k^2 = ijk = -1$. [3, p. 48]

It is extremely important to note the different components of a quaternion. Quaternions have a scalar part, w, and the three unit vector $[x, y, z]$, along the i, j, k axes. Our x, y, z components will represent the axis where the rotation occurs, and the w component is the degree of rotation. The quaternion equation can also be reduced to its scalar and vector components as: $q = (s, v)$ where $(s, v) = s + v_x i + v_y j + v_z k$

Quaternions are often the optimal form of axis rotations in 3D because they avoid many of the problems that can arise with 3×3 matrices and Euler angles. Quaternions are also easier to visualize in practice. Let's shift from the algebraic understanding of quaternions to implementing and visualizing quaternions.

5.1 Visualizing Quaternions

Let's use a simple sphere to understand a simple 3D object on the xyz plane.

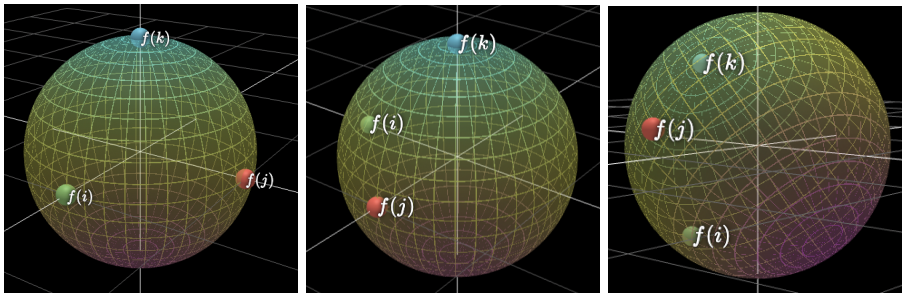

Figure 4: Sphere through quaternion rotations on the ijk-plane. [1]

We can see Figure 4 starts on the ijk-plane. The first image can be represented as the quaternion $Q_a = 1 + 0i + 0j + 0k$. Q_a which has yet to be rotated about any axis. Now applying a rotation using quaternions, our second image can be represented as the quaternion: $Q_b = 0.71 + 0i + 0j - 0.71k$ (the 0.71 might feel random, but recall that $cos(45°) = 0.71$). Notice how we can easily represent this 90° rotation with four numbers. Quaternions can even go beyond simple 90° rotations. Figure 4's final orientation can be represented as the quaternion: $Q_c = 0.63 + 0.38i + 0.23j - 0.63k$.

Quaternions are extremely useful and easy to manipulate in practice because they store rotations with only four numbers in comparison to the nine values in 3 $\times$ 3 matrices. We can also make these rotations through one equation, compared to the three cumulative rotations needed with Euler angles; its flexibility with just a single equation avoids complicated issues like gimbal lock.

Now imagine a computer animation that requires a 3D object to be rotated multiple times. Quaternions have the capability of creating multiple rotations!

5.2 Quaternion Operations

Because computer graphics can get so complex, it is important to utilize a method that is efficient at performing numerous rotations of 3D objects. We can do this with quaternions by multiplying two quaternions to yield another quaternion.

But before we multiply quaternions, let's define the **cross product** which is necessary for our quaternion multiplication formula.

Definition 5.2: Given vectors $\vec{a} =< a_x, a_y, a_z >$ and $\vec{b} =< b_x, b_y, b_z >$, the **cross product of $\vec{a}$ and $\vec{b}$** is
$\begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} = i \begin{bmatrix} a_y & a_z \\ b_y & b_z \end{bmatrix} - j \begin{bmatrix} a_x & a_z \\ b_x & b_z \end{bmatrix} + k \begin{bmatrix} a_x & a_y \\ b_x & b_y \end{bmatrix}.$

Now we can define quaternion multiplication!

Theorem 2. *Let $q_1 = s_1 + v_1$ and $q_2 = s_2 + v_2$ be two different quaternions. Then $q_1 q_2 = (s_1 s_2 - v_1 \cdot v_2) + (s_1 v_2 + s_2 v_1 + v_1 \times v_2)$. [4, p. 360]*

Note! *Similar to Euler angles, quaternion multiplication is **NOT** always commutative. [3, p. 68]*

Let's now practice computing multiple rotations with quaternions!

Example 5.2: Let $q_1 = 2 + i + 3j - k$ and $q_2 = 4 - 2i - j + 3k$
$q_1 q_2 = (8 \ - v_1 \cdot v_2) + (2 < -2, -1, 3 > + 4 < 1, 3, -1 > + v_1 \times v_2)$
$q_1 q_2 = 8 - (-2 - 3 - 3) + < -4, -2, 6 > + < 4, 12, -4 > + v_1 \times v_2$
$q_1 q_2 = (8 + 8) \ + (< 0, 10, 2 > + < 1, 3, -1 > \times < -2, -1, 3 >)$
$q_1 q_2 = 16 + (< 0, 10, 2 > + < 8, -1, 5 >)$
$q_1 q_2 = 16 + < 8, 9, 7 >$
$q_1 q_2 = q_3 = 16 + 8i + 9j + 7k$

Why does it matter? Well, we have effectively converted two rotations into a single rotation. From *Example 5.1*, instead of having a computer rotate with the quaternion q_1 and then rotate with q_2 we can instead multiply both quaternions and create a single rotation $q_3 = 16 + 8i + 9j + 7k$.

Now imagine computer animations that require multiple rotations for a 3D object. Using 3×3 matrices requires us to hold 9 values, and if you need multiple rotations by 3×3 matrices, be prepared for slower run times and more memory usage! However, quaternions only need 4 values, and multiplying them for multiple rotations would solve this time and memory problem.

6 Conclusion: Putting it all together

So what is the best method for rotation in computer graphics? That depends. For simple programs with few rotations, multiplication with 3×3 matrices is perfectly adequate. In such cases, it might actually increase runtime to use quaternions or euler matrices because the latter two methods would require more calculations than necessary.

If you are making a mid level program, Euler angles might be the preferred method. But for complicated programs with many rotations such as video game design and movie animation, quaternions are the preferred method. In larger programs, although difficult to understand at first, quaternions provide the most flexibility and efficiency for large scale programming.

References

[1] Ben Eater. Visualizing quaternions, an explorable video series.

[2] David J Eck. Introduction to computer graphics, 2021.

[3] John Vince. *Quaternions for computer graphics*. Springer, 2011.

[4] A Watt and Mark Watt. Advanced animation and rendering techniques: Theory and practice, 1992.

[5] Julian Zeitlhöfler. *Nominal and observation-based attitude realization for precise orbit determination of the Jason satellites*. PhD thesis, 06 2019.