

UNIVERSITY OF OXFORD

Hand Gesture Recognition via Leap Motion Sensor

Authors:
Jahangir IQBAL

Supervisor:
Dr. Irina VOICULESCU

*A thesis submitted in fulfillment of the requirements
for the degree of Masters in Computer Science
in the*

Department of Computer Science

August 29, 2017

Declaration of Authorship

I, Jahangir IQBAL, declare that this thesis titled, “Hand Gesture Recognition via Leap Motion Sensor” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

University of Oxford

Abstract

Faculty Name
Department of Computer Science

Masters in Computer Science

Hand Gesture Recognition via Leap Motion Sensor

by Jahangir IQBAL

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vii
1 Rotation of the Hand UI Model	1
1.1 Description of Problem	1
1.2 JavaFx Coordinate System vs Leap Motion Coordinate System	2
1.3 Unexpected Leap Motion API Results	4
1.3.1 Pitch, Roll, Yaw in Leap API	4
1.3.2 Negative Zeros and Flipped Axes	5
1.4 Simplified Hand Model	7
1.5 Composite Rotational Transformations	8

List of Figures

1.1	Hand in Flat Orientation	1
1.2	Hand with Yaw Rotation	1
1.3	Hand with Roll Rotation	1
1.4	Hand in Weird Handshake Position	2
1.5	Hand Fixed to Vertical Orientation	2
1.6	Leap Motion Coordinate System	3
1.7	Left vs Right Hand Coordinate System	3
1.8	JavaFX Coordinate System	3
1.9	Pitch	5
1.10	Yaw	5
1.11	Roll	5
1.12	Pitch() Debugging	6
1.13	Roll Debugging	6
1.14	getPitch() Function	7
1.15	UIHandSuperSimple Hand	8
1.16	fixRotations() Function	9
1.17	After First Yaw	9
1.18	After Pitch	10
1.19	After Final Yaw	10
1.20	matrixRotateNode() Function	11
1.21	Single Rotation Matrices	11
1.22	Composite Rotational Matrix	12
1.23	Rotational Angle and Axis Formulae	12

For/Dedicated to/To my...

Chapter 1

Rotation of the Hand UI Model

In this chapter the rotation of the Hand UI model will be discussed.

1.1 Description of Problem

The UI model of the hand is built from the Leap Motion sensor data as discussed in the chapter. This data represents the hand as it is displayed in reality above the Leap Motion controller device. Originally the representation of the hand was built from this hand without any further modifications. However, to demonstrate why this is sometimes not the ideal situation, see Figure 1.1. It shows the hand as it is displayed in reality; it is in parallel plane above the device.

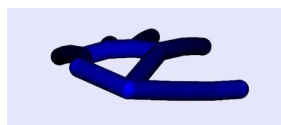


FIGURE 1.1: This shows the user's hand in a flat orientation; as determined by the Leap Motion sensor.

It should be noted how difficult it is to see the fingers and thumb. They are only barely visible. Therefore the user would be forced to force his or her hand into a vertical position by straining their wrist just so they can see the gesture they are performing on the screen. Figures 1.2 and 1.3 show other orientations the hand can take; these figures show the hand after a yaw rotation and after a roll rotation respectively. Again, note the slight difficulty in figuring out the thumb and fingers positions and orientations when the hand is rolled around the z-axis. The hand that is shown in the yaw rotation in Figure 1.2 is able to be seen a little clearly because the wrist was strained upwards.

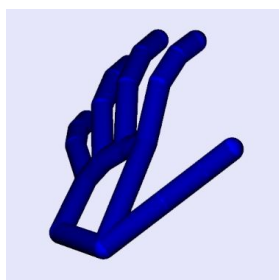


FIGURE 1.2: The user's hand after the certain yaw rotation around the y-axis.

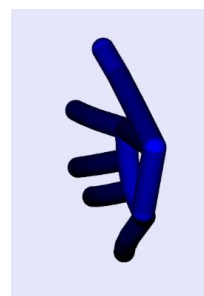


FIGURE 1.3: The user's hand after the certain roll rotation around the z-axis.

Finally all of these different orientation can of course overlap with each other to create a complex orientation of the hand as shown in Figure 1.4, which shows the user's left hand in a weird handshake sort of position while being rotated to the right and tilted upwards.

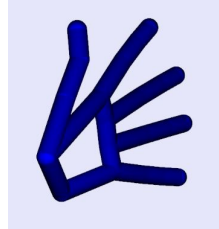


FIGURE 1.4: This shows the user's hand in a flat orientation; as determined by the Leap Motion sensor.

This chapter will explain how these different possible orientations the user's hand might be in while they are performing gestures shown on the screen are accounted for and "undone". Doing this results in the user's hand to be displayed in a set vertical orientation on the screen despite the different ways the user may have oriented their hand above the device in reality. The final resulting hand for all of the figures seen previously will be as is shown in Figure 1.5 after the composite rotation have transformed it to a vertical position.



FIGURE 1.5: This shows what the user's hand will look like after all of the possible rotational transforms have been undone and the hand has been fixed to a vertical orientation.

This feature of the application makes it easier for the user to see what his or her hand is doing without requiring them to always contain their hand in a specific orientation. The goal of this application is to measure the accuracy with which a user is able to complete certain gestures, not the orientation of their hand. In fact, the algorithms that are used to grade the correctness of the user's attempted gesture do not take the user's hand orientation into account. They focus on the fingers bones and their relative orientations to each other. At the beginning of this project, one of the ideas discussed was to build some sort of rig which would be used to place the user's hand in a set orientation. However because of what will be explained in this chapter such a rig is no longer necessary.

1.2 JavaFx Coordinate System vs Leap Motion Coordinate System

The Leap Motion controller has a different coordinate system from JavaFX. This distinction is very important to get out of the way as the rest of the chapter will rely on

such an understanding. The coordinate system for the Leap Motion device is shown in Figure 1.6. Note the placement of the green LED light in the Leap Motion device as shows the orientation of the otherwise symmetrical picture.

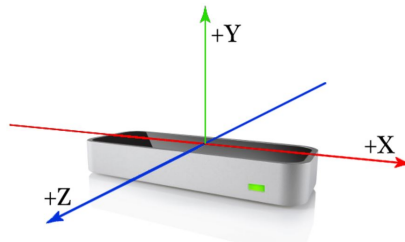


FIGURE 1.6: The coordinate system that is used by the Leap Motion sensor. Data collected will based upon these axes.

This coordinate system is referred to as a right-handed coordinate system because of the easy way in which it can be represented by the first three fingers of the right hand as shown in Figure 1.7.

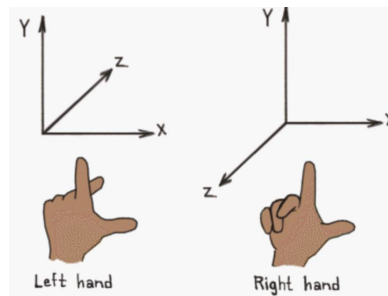


FIGURE 1.7: Left and right handed coordinate systems and how they can be shown via fingers.

In contrast, the traditional coordinate system which is used in computer science and which is what JavaFX also follows is shown in Figure 1.8. The JavaFx coordinate system is not left handed or right handed as far as we can tell from the Figures shown.

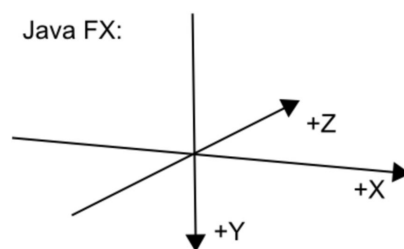


FIGURE 1.8: The coordinate system used in JavaFX. Setting transforms on objects will be based on this coordinate system.

Understanding these somewhat subtle difference and being able to switch from one context to the other was very important when considering the various rotational transforms that needed to be applied to the Hand UI model in order to straighten it to a vertical position. For example, one of the things that caused me some headaches sometimes when I was fixing and debugging my code was the fact that 3D rotations are measured in a very specific way. When it is said that some object is rotated

around the y-axis by 90 degrees, this means that if we imagine ourselves to be an observer standing on the y-axis and looking down the negative direction of this y-axis, then we would observe the object to rotate counter-clockwise about the y-axis. Therefore, it becomes very clear why it is significant to have clear understanding of the coordinate system that is being used for a particular section of code. Leap Motion data is returned with the default settings of the Leap Motion coordinate system. This data must be appropriately modified when the code using it is based upon the JavaFX coordinate system. Likewise composite rotations around certain axis might become jumbled up if one is not careful.

1.3 Unexpected Leap Motion API Results

In the course of working on this feature, which was explained in first section of this chapter, I realized that I was being returned results from the Leap Motion API that I was not expecting. I tested and debugged further into these issues until I realized some interesting things about the Leap Motion API which helped me to implement the vertical rotation of the user's hand successfully. In this section, I will go over some of the unexpected results that I obtained when I was learning more about the Leap Motion API's. I created a special Java class to help me in this endeavor which contains code samples with some simple yet illustrative scenarios which show how the Leap Motion API works.

I first noticed these problems when I was working user hand data, however, I soon realized these issues were more so with how the Vector is defined in the Leap Motion API. Therefore, the examples shown in this section will primarily be using vectors to illustrate the issues encountered, but of course these issues then extend to the collected hand data because of the nature of how Hand objects are defined in Leap Motion.

1.3.1 Pitch, Roll, Yaw in Leap API

Leap Motion's Java API includes a Vector class that contains several useful operations one might need to perform. For example, it includes three functions that return the pitch, roll and yaw angles for the vector. However, right away we need to understand that these angles which are returned are with respect to certain axes. The `pitch()` is an instance method for a vector object that will return the angle between this vector's projection onto the y-z plane and the negative z-axis. Figure 1.9 shows this. Yaw is defined to be the angle between the negative z-axis and the projection of the vector onto the x-z plane (Figure 1.10).

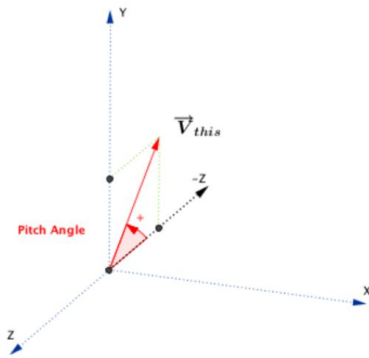


FIGURE 1.9: Pitch angle.

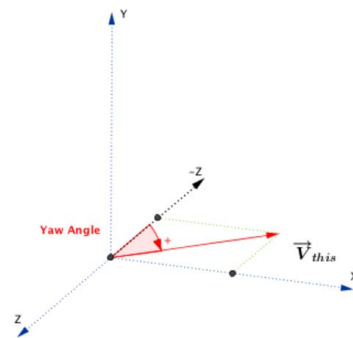


FIGURE 1.10: Yaw angle.

roll() function is defined to be gotten by projecting the vector in question onto the x-y plane and taking the angle between this projection and the positive y-axis (which as we know is defined to be pointing upwards in the Leap Motion coordinate system and downwards in JavaFX). See Figure 1.11 to get a visual understanding of how the roll angle is defined for a given vector.

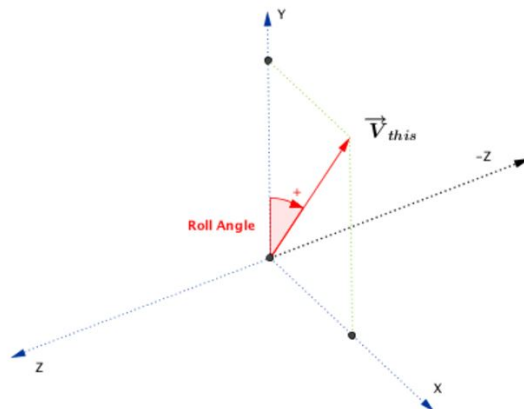


FIGURE 1.11: Roll angle as defined in Leap Motion documentation.

1.3.2 Negative Zeros and Flipped Axes

During my testing, I realized that the Vector class returns some strange results when we test the pitch() function with the simple x-axis unit vector $\langle 1, 0, 0 \rangle$. One would expect that the pitch would be zero, since the projection of the x-axis unit vector onto the y-z plane is zero. However, as the debugging result in Figure 1.12 shows, that was not the case.

```

Run DebugHelper
"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...
Vector: (1, 0, 0)
pitch: 180.0
roll (corrected): 90.0
yaw: 90.0
Vector: (1, 0, -0)
pitch: 0.0
roll (corrected): 90.0
yaw: 90.0
Process finished with exit code 0

```

FIGURE 1.12: A simple example's run output that demonstrates the negative zeros and their importance in Leap Motion API.

It appears to be that a negative zero must be supplied in the z-component of the vector to get a zero pitch angle. When I found this out, I was very surprised as it was the first time I realized that there was such a thing as a negative floating point zero. However, another thing to note is that in the debug output, I have the word "corrected" next to the roll angle.

Like the pitch() angle, I noticed something that was a bit strange about the results the roll() function was returning during my debugging. It was returning 135 degrees when I was expecting it to return 45 degrees (see Figure 1.15) because the vector it is being tested on makes such a projection onto the x-y plane that the roll should be 45. I think the documentation page either has a mistake the roll() method or maybe the Vector class has a error in the way the roll method is defined.

```

Run DebugHelper
"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...
Vector: (0.05, 0.05, -0.9)
pitch: 3.18
roll (uncorrected): 135.0
roll (corrected): 45.0
yaw: 3.18
Process finished with exit code 0

```

FIGURE 1.13: The roll() angle returned for a vector needs to be "corrected".

The angle being returned is actually between the negative y-axis (Leap Motion coordinate system) and the projection. Therefore I wrote my own correctedRoll() which basically takes a given roll angle and subtracts it from 180 degrees to return a roll angle that is in accordance to how roll() is defined in Leap Motion documentation. In response to finding out about these problems, I wrote my own methods to find pitch, roll and yaw when I was dealing with hand objects. The code for one of these methods is shown in Figure 1.14.

```
1 private float getPitch(Hand h) {
2     //angle of hand direction to y-axis
3     float angleAmount = (float)
4         Math.toDegrees(h.direction().angleTo(Vector.yAxis()));
5     //switch direction of angle
6     if (h.direction().getZ() > 0.0f) {
7         return (-1.0f * angleAmount); //returning a -negative angle to "undo" the
8             positive pitch noticed in hand
9     } else {
10         //99 % of the time, pitch will be a positive angle, so this case will run
11             most of the time.
12         return angleAmount;
13     }
14 }
```

FIGURE 1.14: This function returns the pitch of the hand object as the angle the hand makes to the x-z plane.

This method does not rely on projections to the y-z plane to find out the pitch of the hand. Intuitively, when I imagined the pitch of the hand model to be to angle it is making to the x-z plane; regardless of which direction the wrist or the fingers of the hand are pointing in. The Leap pitch() function expects the direction of the hand to be relatively inline with the z-axis in order to give a valid pitch measurement. The method I defined does not have this restriction and is able to return an informative pitch angle no matter if the user has rotated his/hand around the y-axis. I calculate the yaw for a particular hand in the same fashion, except for that function I take the angle between the direction of the hand and the negative z-axis.

1.4 Simplified Hand Model

One of the lessons that this project has highlighted for me is how important it is to simplify a complex problem when finding a solution to it. Of course this sounds like a relatively obvious lesson, but I feel this project forced me to practice it many times. I would like to give a short example of this by talking a little bit about the UIHandSuperSimple class that I wrote when I was working on the problem this chapter addresses. The aptly named super simple hand model was a model of the user's hand that has just three components, a rectangular box to represent the palm, a cylinder to represent the thumb and a sphere to represent the fingers. See Figure ?? to view this masterpiece.

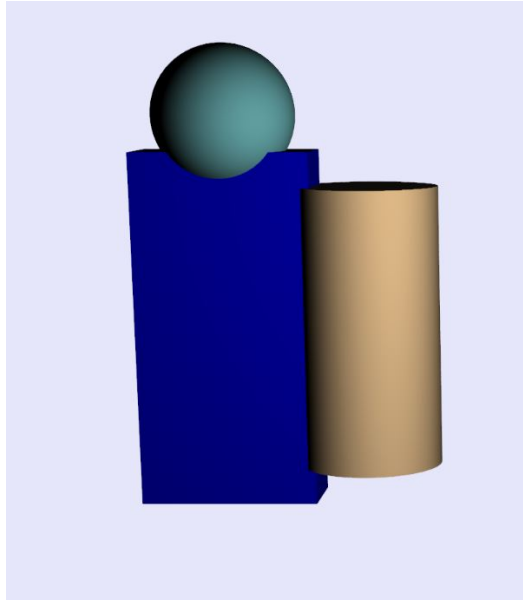


FIGURE 1.15: A very basic model of the hand.

The reason this hand was important during my journey to solving this problem was because I could easily represent different orientations in it by using just simple vectors. This allowed me to quickly debug and figure out errors in the way I was approaching things. The other Hand UI model could not be set only by a single vector in such a way that the every single bone of the hand would also be appropriately set. This was because that model relies on a lot more data from the Leap Device to build the model more accurately. The simplicity of dealing with this hand model eventually also allowed me to figure out how to undo the unwanted hand rotations for the UI display that were part of the Leap Motion data.

1.5 Composite Rotational Transformations

To understand how the UI hand model is rotated to a vertical position, we will consider the starting hand position to be the "weird handshake position" as is shown in Figure 1.4, which represents the left hand in a handshake position that is slightly pitched upwards along the x-axis and with a 45 degree yaw to the right around the y-axis. The method which will reset this user's hand to a vertical orientation automatically (without requiring the user to change the position of their hand at all) is shown in Figure 1.16.

```

1 private void fixRotations(Hand h) {
2     //add a yaw to perform before the matrixRotateNode sets the axis and angle.
3     float firstYaw = getFirstYaw(h); // dont need to convert to radians for
        Rotate transform java class.
4     if (this.getTransforms().size() == 0) {
5         this.getTransforms().add(new Rotate(firstYaw, new Point3D(0, 1, 0)));
6     } else {
7         this.getTransforms().set(0, new Rotate(firstYaw, new Point3D(0, 1, 0)));
8     }
9
10    //correct to fit Javafx coordinate system and convert to radians.
11    float pitch = (float) Math.toRadians(getPitch(h) * (-1.0f));
12    float finalYaw = (float) Math.toRadians(getFinalYaw(h) * (-1.0f));
13    float roll = 0; //roll is never needed
14
15    //order of operations: pitch, yaw, roll.
16    ViewMath.matrixRotateNode(this, roll, pitch, finalYaw);
17 }

```

FIGURE 1.16: This function fixes the rotations of the UI Hand model so that it will be always displayed in a vertical orientation, no matter what orientation the user's hand is in.

This method takes in a Leap Motion Hand object and basically performs three transformations on it. Firstly, it performs a yaw rotation of the amount determined by the `getFirstYaw()` method. The `getFirstYaw()` method determines what the angle of the hand's direction is to the negative z-axis; and so the first transform that will get added to the UI hand model is will rotate the hand about the y-axis. Figure 1.17 shows what the hand model will look like after this first rotational transform has been performed.

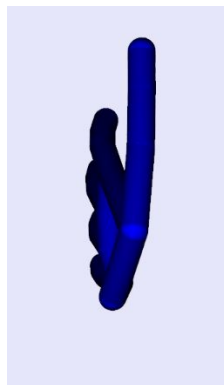


FIGURE 1.17: The hand model of Figure 1.4 after first yaw.

Secondly, the pitch angle is determined the `getPitch()` to see how much to rotate the hand upward to be inline with the y-axis. Also the finalYaw angle is determined using the `getFinalYaw()` method which will return an angle representing how much the hand model has to be rotated around the y-axis again when the hand is pointing upwards in the positive y-direction in the JavaFX coordinate system. There is no roll angle that is needed for putting the hand in a vertical position. The pitch and finalYaw angle (along with a 0 value for the roll angle) are passed into a helper function called `matrixRotateNode()`, shown in Figure 1.20. This method computes

the composite rotational matrix based upon the angles that are passed into it and performs the correct composite rotation around the calculated axis of rotation on the JavaFX Node object that is passed into as its first argument. If for a moment we consider the hand model originally started in the weird handshake position as stated above, then after the first yaw, and pitch have been performed, the hand model will be resting with its fingers pointing straight up and its palm facing the x-axis. See Figure 1.18

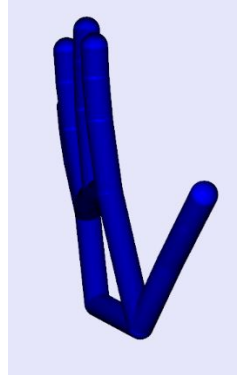


FIGURE 1.18: The hand model of Figure 1.4 after first yaw and pitch have been performed.

Finally, after the final yaw has also been performed, the hand model will be resting in the final vertical orientation with its fingers pointing up and the palm facing the negative z-axis as shown in Figure 1.19



FIGURE 1.19: The hand model after it's orientation has been fixed.

The final yaw angle is determined by which direction the palm normal of the leap motion hand is facing. This palm normal can be pointing along the x-axis or the z-axis. A weighted average of the two directions is taken to determine the angle to be computed for the final yaw.

```

1 public static void matrixRotateNode(Node n, double roll, double pitch, double
    yaw) {
2     //build rotational matrix
3     double A11 = Math.cos(roll) * Math.cos(yaw);
4     ...
5     double A32 = -Math.cos(yaw) * Math.sin(pitch);
6     double A33 = Math.cos(pitch) * Math.cos(yaw);
7
8     //angle of rotation
9     double angleOfRotation = Math.acos((A11 + A22 + A33 - 1d) / 2d);
10    if (angleOfRotation != 0d) {
11        double den = 2d * Math.sin(angleOfRotation);
12        Point3D axisOfRotation = new Point3D((A32 - A23) / den, (A13 - A31) / den,
            (A21 - A12) / den);
13        //rotate node
14        n.setRotationAxis(axisOfRotation);
15        n.setRotate(Math.toDegrees(angleOfRotation));
16    }
17 }

```

FIGURE 1.20: This function takes in a JavaFX node object and three angles representing the roll, pitch and yaw angles and returns that object after it has been rotated by appropriate final angle around the computed axis of rotation.

The code shown in the `matrixRotateNode()` in the Figure 1.20 above looks very complicated. It is the result of mathematical formulas that determine how to rotate an object in 3D space. Given three angles representing the pitch, yaw and roll we can compute a final matrix of rotation from the three smaller matrices that would represent rotations about the specific x-axis, y-axis, z-axis that the pitch, yaw, and roll angles would represent respectively. The matrices that would represent these three individual rotations are shown in Figure 1.21.

$$\begin{aligned}
 \mathbf{A}_X &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{bmatrix} \\
 \mathbf{A}_Y &= \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix} \\
 \mathbf{A}_Z &= \begin{bmatrix} \cos \psi & \sin \psi & 0 \\ -\sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

FIGURE 1.21: The three matrices representing individual rotations about the x, y, and z axis.

Their composite matrix would be computed by $A =$

$$A_z * A_y * A_x$$

and the result of this matrix multiplication operation is shown in Figure ?? . The method uses this final matrix to determine the angle and axis of rotation by the

formulae shown in Figure 1.23.

$$\mathbf{A} = \begin{bmatrix} \cos \theta \cos \psi & \cos \phi \sin \psi + \sin \phi \sin \theta \cos \psi & \sin \phi \sin \psi - \cos \phi \sin \theta \cos \psi \\ -\cos \theta \sin \psi & \cos \phi \cos \psi - \sin \phi \sin \theta \sin \psi & \sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi \\ \sin \theta & -\sin \phi \cos \theta & \cos \phi \cos \theta \end{bmatrix}$$

FIGURE 1.22: The result of the matrix multiplication.

$$\begin{aligned} \theta &= \arccos \left(\frac{1}{2} [A_{11} + A_{22} + A_{33} - 1] \right) \\ e_1 &= \frac{A_{32} - A_{23}}{2 \sin \theta} \\ e_2 &= \frac{A_{13} - A_{31}}{2 \sin \theta} \\ e_3 &= \frac{A_{21} - A_{12}}{2 \sin \theta} \end{aligned}$$

FIGURE 1.23: The formulae which determine the angle of rotation and the axis of rotation $\langle e_1, e_2, e_3 \rangle$.