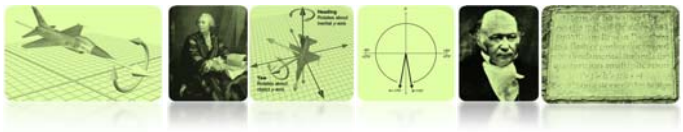


Chapter *8*

Rotation in Three Dimensions

Fletcher Dunn
Valve Software

Ian Parberry
University of North Texas

What You'll See in This Chapter

This chapter tackles the difficult problem of describing the *orientation* of an object in 3D, and the closely related concepts of *rotation* and *angular displacement*. It is divided into seven sections.

- Section 8.1 discusses the subtle differences between terms like *orientation*, *direction*, and *angular displacement*.
- Section 8.2 describes how to express orientation using a matrix
- Section 8.3 describes how to express angular displacement using Euler angles.
- Section 8.4 describes the axis-angle and exponential map forms.
- Section 8.5 describes how to express angular displacement using a quaternion.
- Section 8.6 compares and contrasts the different methods.
- Section 8.7 explains how to convert an orientation from one form to another.

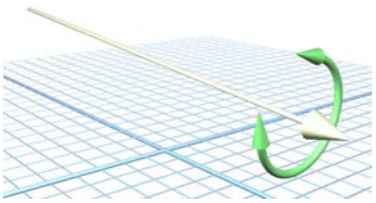
Word Cloud

Section 8.1:

What Exactly is "Orientation"?

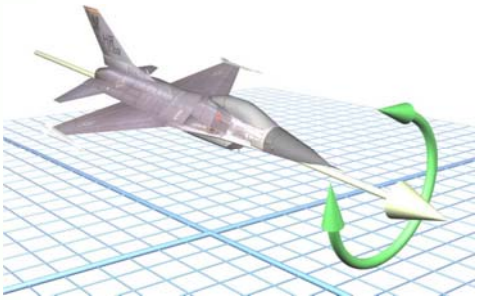
Orientation

- What is orientation? More than direction.
- A vector specifies direction but it can also be twisted.



This is Important Because

Twisting an object changes its orientation.



Some People Get Confused

- Because you can specify a direction using two angles (polar coordinates)
- Specifying an orientation requires 3 angles.
- Or at least 3 numbers whichever way you represent it.
- To add to the confusion, there are 3 popular ways to represent orientation.

What is Angular Displacement?

- Orientation can't be given in absolute terms.
- Just as a position is a translation from some known point, an orientation is a rotation from some known reference orientation (often called the *identity* or *home* orientation).
- The amount of rotation is known as an *angular displacement*.

How to Represent Orientation

1. Matrices
2. Euler angles
3. Quaternions

Section 8.2:

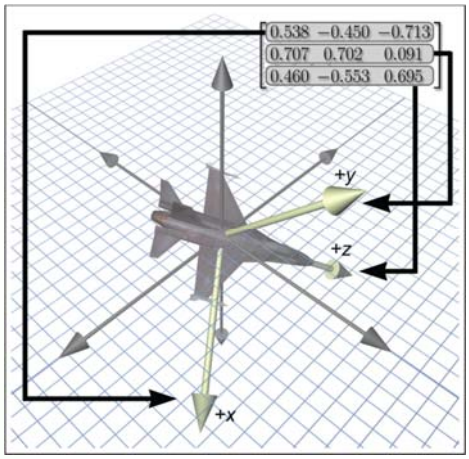
Matrix Form

Matrix Form

- List the relative orientation of two coordinate spaces by listing the transformation matrix that takes one space to another.
- For example: from object space to upright space.
- Transform back by using the inverse matrix.

Example

- We've seen how a matrix can be used to transform points from one coordinate space to another.
- In the figure on the next slide, the matrix in the upper right hand corner can be used to rotate points from the object space of the jet into upright space.
- We've pulled out the rows of this matrix to emphasize their direct relationship to the coordinates for the jet's body axes.
- The rotation matrix contains the object axes expressed in upright space.
- Simultaneously, it is a rotation matrix. We can multiply row vectors by this matrix to transform those vectors from object space coordinates to upright space coordinates.



To Transpose or Not to Transpose? That is the Question

- A legitimate question to ask is, why does the matrix contain the object space axes expressed using upright space coordinates? Why not the upright space axes expressed in object space coordinates?
- Another way to phrase this is: why did we choose to give a rotation matrix that transformed vectors from object space to upright space? Why not from upright space to object space?
- From a math perspective, this question is redundant. Because rotation matrices are orthogonal, their inverse is the same as their transpose. (Recall from Chapter 6.)
- Thus the decision is entirely a cosmetic one.

A Coder's Lament

- But practically speaking, in our opinion, it is quite important.
- At issue is whether you can write code that is intuitive to read and works the first time, or if it requires a lot of work to decipher, or a knowledge of conventions which are not stated since they are obvious to everyone but you.
- Our opinions come from watching programmers grapple with rotation matrices. We don't expect that everyone will agree with us, but we hope that every reader will at least appreciate the value in considering these issues.

It's Just a Matrix, Right?

- Certainly every good math library will have a 3 x 3 matrix class that can represent any arbitrary transformation, which is to say that it makes no assumptions about the value of the matrix elements. (Or perhaps it is a 4 x 4 matrix that can do projection, or a 4 x 3 that can do translation but not projection – those distinctions are not important here.)
- For a matrix like this, the operations inherently are in terms of some input coordinate space and an output coordinate space.
- This is just implicit in the idea of matrix multiplication.
- But if you need to go from output to input, then you need to obtain the inverse of the matrix.

But…

- It is a common practice to use the generic transformation matrix class to describe the orientation of an object.
- In this case, rotation is treated just like any other transformation. The interface remains in terms of a source and destination space.
- It is our experience that the following two matrix operations are by far the most commonly used:
  1. Take an object space vector and express it in upright coordinates.
  2. Take an upright space vector and express it in object coordinates.

Which Matrix Should We Use?

- Notice that we need to be able to go in both directions. We have no reason to believe that either direction is significantly more common that the other.
- But more importantly, the very nature of the operations and the way programmers naturally think about the operations is in terms of *object space* and *upright space* (or some other equivalent terminology, such as *parent space* and *child space*).
- We do not think of them in terms of a source space and a destination space. It is in this context that we wish to consider the question posed at the beginning of this section: which matrix should we use?

Orientation in a State Variable

- First, we should back up a bit and remind ourselves of the mathematically moot but yet conceptually important distinction between *orientation* and *angular displacement*.
- If your purpose is to create a matrix that does a specific angular displacement (such as "rotate 30° about the *x*-axis), then the two operations listed 2 slides ago are not really the ones you probably have in your head, and using a generic transform matrix with its implied direction of transformation is no problem, and so this discussion does not apply.
- Instead, suppose that the orientation of some object is stored as a state variable.

We're All Coders, Right?

- Let's assume that we adopt the common policy and store orientation using the generic transformation matrix.
- We'll arbitrarily pick a convention that multiplication by this matrix will transform from object to upright space.
- If we have a vector in upright space and we need to express it in object space coordinates, we must multiply this vector by the inverse of the matrix.
- Now let's see how our policy affects the code that is written and read hundreds of times by average game programmers.

In Code

- Rotate some vector from object space to upright space is translated into code as multiplication by the matrix.
- Rotate a vector from upright space to object space is translated into code as multiplication by the inverse of the matrix. (Actually, multiplication by the transpose, since rotation matrices are orthogonal, but that is not the point here.)

On Coding Style

- Notice that the code does not match one-to-one with the high-level intentions of the programmer.
- Furthermore, to read or write this code requires that you know what the conventions are.
- It is our opinion that this coding style is a contributing factor to the difficulty that beginning programmers have in learning how to use matrices.

Just a Matrix?

- An equivalent practice is to not use a class at all, and just declare a matrix with something like **float** R [3][3] .
- This style of code will force every user to remember what the conventions are every time they use the matrix.
- From our experience, they are usually not documented, since it was *obvious* to the author of this code how it was supposed to work.
- This invariably results in fiddling and random transposing by the uninitiated, who are forced to reverse engineer the arbitrary conventions through trial and error.

A Rotation Matrix Class

- An alternative way to code is to have a special 3 x 3 matrix class that is used exclusively for rotations.
- It assumes, as an invariant, that the matrix is orthogonal, meaning it only contains rotation. (We also would probably assume that the matrix does not contain a reflection, even though that is possible in an orthogonal matrix.)
- With these assumptions in place, we are now free to perform rotations using the matrix at a higher level of abstraction.
- Our interface functions match exactly the high-level intentions of the programmer.

Some Justification

- Furthermore, we have removed the confusing linear algebra details having to do with row vectors versus column vectors, which space is on the left or right, and which way is the regular way and which is the inverse, etc.
- Or rather, we have confined such details to the class internals, the person implementing the class certainly needs to pick a convention (and hopefully document it).
- In fact, in this specialized matrix class, the operations of *multiply a vector* and *invert this matrix* really are not that useful.
- We advocate keeping this dedicated matrix class confined to operations that couch things in terms of *upright space* and *object space*, rather than *multiply a vector*.

If Frank Sinatra Was a Coder, Would He Do It His Way?

- Of course, there are those who find it perfectly obvious which way the regular multiplication should go and which should be the transpose, and they don't understand why others can't keep it straight.
- They might say, "You just use row vectors and regular multiplication rotates from object to upright, because that's the rotation that's needed the most frequently. What could be more obvious?"
- The problem is that there are those who feel column vectors are better, or that the forward direction should go from upright to object space.
- If these conventions are so obvious, then why doesn't everybody agree on what they should be?
- In other words, coding for diversity is a good thing.

It Shouldn't Matter

- So, back to the question posed earlier: Which matrix should we use?
- Our answer is "It shouldn't matter."
- By that we mean there is a way to design your matrix code in such a way that it can be used without knowing what choice was made.

What's in a (Function) Name?

- As far as writing real C++ code goes, this is purely a cosmetic change.
- For example, perhaps we just replace the function name multiply () with objectToUpright(), and likewise we replace multiplyByTranspose() with uprightToObject().
- Our argument is that the version of the code with descriptive, named coordinate spaces is easier to read and write.
- The function names multiply() and multiplyByTranspose() have no descriptive value (they could be replaced with doTheThing() and doTheOtherThing() and no information is lost).
- To use these functions requires you to know what "the thing" is and what "the other thing" is. But objectToUpright() and uprightToObject() are descriptive and self-contained.

Back to the Math

- Enough on how our conventions encourage a robust coding style.
- You will probably just ignore us and get into trouble, but we tried.

Direction Cosines Matrix

- A *direction cosines matrix* is the same thing a a rotation matrix, but the term refers to a special way to interpret (or construct) the matrix.
- The term *direction cosines* refers to the fact that each element in a rotation matrix is the dot product of a cardinal axis in one space with a cardinal axis in the other space.
- For example, the center element m_{22} in a 3 x 3 matrix gives the dot product that the *y*-axis in one space makes with the *y*-axis in the other space.

Direction Cosines Matrix

- More generally, let's say that the basis vectors of a coordinate space are the mutually orthogonal unit vectors $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, while a second coordinate space with the same origin has as its basis a different (but also orthonormal) basis $\hat{\mathbf{p}}'$, $\hat{\mathbf{q}}'$, $\hat{\mathbf{r}}'$.
- The rotation matrix that rotates row vectors from the first space to the second is the matrix of direction cosines (dot products) of each pair of basis vectors, as follows:

Direction Cosines Matrix

$$\mathbf{v}\begin{bmatrix} \hat{\mathbf{p}}\cdot\hat{\mathbf{p}}' & \hat{\mathbf{q}}\cdot\hat{\mathbf{p}}' & \hat{\mathbf{r}}\cdot\hat{\mathbf{p}}' \\ \hat{\mathbf{p}}\cdot\hat{\mathbf{q}}' & \hat{\mathbf{q}}\cdot\hat{\mathbf{q}}' & \hat{\mathbf{r}}\cdot\hat{\mathbf{q}}' \\ \hat{\mathbf{p}}\cdot\hat{\mathbf{r}}' & \hat{\mathbf{q}}\cdot\hat{\mathbf{r}}' & \hat{\mathbf{r}}\cdot\hat{\mathbf{r}}' \end{bmatrix} = \mathbf{v}'.$$

These axes can be interpreted as geometric rather than algebraic entities, so it really does not matter what coordinates are used to describe the axes (provided we use the same coordinate space to describe all of them), the rotation matrix will be the same.

Example

- For example, let's say that our axes are described using the first coordinate space.
- Then $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, have the trivial forms [1, 0, 0], [0, 1, 0] and [0, 0, 1], respectively.
- The basis vectors of the second space, $\hat{\mathbf{p}}'$, $\hat{\mathbf{q}}'$, $\hat{\mathbf{r}}'$ are not expressed in their own space, and thus they have arbitrary coordinates.
- When we substitute the trivial vectors $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, into the matrix on the previous slide and expand the dot products, we get:

$$\begin{bmatrix} [1,0,0]\cdot\hat{\mathbf{p}}' & [0,1,0]\cdot\hat{\mathbf{p}}' & [0,0,1]\cdot\hat{\mathbf{p}}' \\ [1,0,0]\cdot\hat{\mathbf{q}}' & [0,1,0]\cdot\hat{\mathbf{q}}' & [0,0,1]\cdot\hat{\mathbf{q}}' \\ [1,0,0]\cdot\hat{\mathbf{r}}' & [0,1,0]\cdot\hat{\mathbf{r}}' & [0,0,1]\cdot\hat{\mathbf{r}}' \end{bmatrix}$$

$$= \begin{bmatrix} p'_x & p'_y & p'_z \\ q'_x & q'_y & q'_z \\ r'_x & r'_y & r'_z \end{bmatrix}$$

$$= \begin{bmatrix} -\hat{\mathbf{p}}'- \\ -\hat{\mathbf{q}}'- \\ -\hat{\mathbf{r}}'- \end{bmatrix}$$

Conclusion

- In other words, the rows of the rotation matrix are the basis vectors of the output coordinate space, expressed using the coordinates of the input coordinate space.
- Of course, this fact is not just true for rotation matrices, it's true for all transformation matrices.
- This is the central idea of why a transformation matrix works, which was developed in Chapter 4.

The Other Case

- Now let's look at the other case.
- Instead of expressing all the basis vectors using the first coordinate space, measure them using the second coordinate space (the output space).
- This time, $\hat{\mathbf{p}}'$, $\hat{\mathbf{q}}'$, $\hat{\mathbf{r}}'$ have trivial forms, and $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, are arbitrary.
- Putting these into the direction cosines matrix produces:

$$
\begin{aligned}
&\begin{bmatrix}
\hat{\mathbf{p}} \cdot [1,0,0] & \hat{\mathbf{q}} \cdot [1,0,0] & \hat{\mathbf{r}} \cdot [1,0,0] \\
\hat{\mathbf{p}} \cdot [0,1,0] & \hat{\mathbf{q}} \cdot [0,1,0] & \hat{\mathbf{r}} \cdot [0,1,0] \\
\hat{\mathbf{p}} \cdot [0,0,1] & \hat{\mathbf{q}} \cdot [0,0,1] & \hat{\mathbf{r}} \cdot [0,0,1]
\end{bmatrix} \\
&= \begin{bmatrix}
p_x & q_x & r_x \\
p_y & q_y & r_y \\
p_z & q_z & r_z
\end{bmatrix} \\
&= \begin{bmatrix}
| & | & | \\
\hat{\mathbf{p}}^T & \hat{\mathbf{q}}^T & \hat{\mathbf{r}}^T \\
| & | & |
\end{bmatrix}
\end{aligned}
$$

Conclusion

- This says that the columns of the rotation matrix are formed from the basis vectors of the input space, expressed using the coordinates of the output space.
- This is *not* true of transformation matrices in general; it applies only to orthogonal matrices such as rotation matrices.

Advantages of Matrix Form 1

Rotation of vectors is immediately available. You can use a matrix to rotate vectors between object and upright space. No other representation of orientation allows this in order to rotate vectors, you must convert the orientation to matrix form.

Advantages of Matrix Form 2

Format used by graphics APIs.

- Graphics APIs use matrices to express orientation.
- API stands for Application Programming Interface. Basically this is the code that you use to communicate with the graphics hardware.
- When you are communicating with the API, you are going to have to express your transformations as matrices eventually.
- How you store transformations internally in your program is up to you, but if you choose another representation you are going to have to convert them into matrices at some point in the graphics pipeline.

Advantages of Matrix Form 3

Concatenation of multiple angular displacements.

- It is possible to collapse nested coordinate space relationships.
- For example, if we know the orientation of object A relative to object B, and we know the orientation of object B relative to object C, then using matrices, we can determine the orientation of object A relative to object C.
- We have encountered these concepts before, when we learned about nested coordinate spaces in Chapter 3, and then discussed how matrices could be concatenated in Chapter 5.

Advantages of Matrix Form 4

Matrix inversion.

- When an angular displacement is represented in matrix form, it is possible to compute the opposite angular displacement using matrix inversion.
- What's more, since rotation matrices are orthogonal, this computation is a trivial matter of transposing the matrix.

Disadvantages of Matrix Form 1

Matrices take more memory.

- If we need to store many orientations (for example, keyframes in an animation sequence), that extra space for nine numbers instead of three can really add up.
- Let's take a modest example. Let's say we are animating a model of a human that is broken up into 15 pieces for different body parts. Animation is accomplished strictly by controlling the orientation of each part relative to its parent part.
- Assume we are storing one orientation for each part, per frame, and our animation data is stored at a reasonable rate, say, 15hz.
- This means we will have 225 orientations per second. Using matrices and 32-bit floating point numbers, each frame will take 8,100 bytes.
- Using Euler angles (which we will meet next), the same data would only take 2700 bytes. For a mere 30 seconds of animation data, matrices would take 162K more than the same data stored using Euler angles!

Disadvantages of Matrix Form 2

Difficult for humans to use.

- Matrices are not intuitive for humans to work with directly. There are just too many numbers, and they are all from –1 to 1.
- What's more, humans naturally think about orientation in terms of angles, but a matrix is expressed using vectors.
- With practice, we can learn how to decipher the orientation from a given matrix. (Although the techniques from Chapter 4 for visualizing a matrix help a lot.)
- But still this is much more difficult than Euler angles, and going the other way is much more difficult

Disadvantages of Matrix Form 3

Matrices can be malformed.

- A matrix uses nine numbers, when only three are necessary. In other words, a matrix contains six degrees of redundancy.
- There are six constraints that must be satisfied in order for a matrix to be valid for representing an orientation.
- The rows must be unit vectors, and they must be mutually perpendicular.

How Can Matrices Get Malformed?

- We may have a matrix that contains scale, skew, reflection, or projection. Any non-orthogonal matrix is not a well-defined rotation matrix. Reflection matrices (which are orthogonal) are not valid rotation matrices either.
- We may just get bad data from an external source. For example, if we are using a physical data acquisition system, such as motion capture, there could be errors due to the capturing process. Many modeling packages are notorious for producing malformed matrices.
- We can actually create bad data due to floating point roundoff error. For example, suppose we apply a large number of incremental changes to an orientation, which could routinely happen in a game or simulation that allows a human to interactively control the orientation of an object. The large number of matrix multiplications, which are subject to limited floating point precision, can result in an ill-formed matrix. Recall our discussion of *matrix creep* in Chapter 6.

Summary of Matrix Form 1

- Matrices are a brute force method of expressing orientation: we explicitly list the basis vectors of one space in the coordinates of some different space.
- The matrix form of representing orientation is useful primarily because it allows us to rotate vectors between coordinate spaces.
- Modern graphics APIs express orientation using matrices.
- We can use matrix multiplication to collapse matrices for nested coordinate spaces into a single matrix.

Summary of Matrix Form 2

- Matrix inversion provides a mechanism for determining the opposite angular displacement.
- Matrices take two to three times as much memory as the other techniques we will learn. This can become significant when storing large numbers of orientations, such as animation data.
- The numbers in a matrix aren't intuitive for humans to work with.
- Not all matrices are valid for describing an orientation. Some matrices contain mirroring or skew. We can end up with a malformed matrix either by getting bad data from an external source, or through matrix creep.

Section 8.3:
Euler Angles

Euler Angles

- Euler angles are another common method of representing orientation.
- Euler is pronounced "oiler," not "yoolur."
- They are named for the famous mathematician who developed them, Leonhard Euler (1707 – 1783).
- (Image from Wikimedia Commons.)

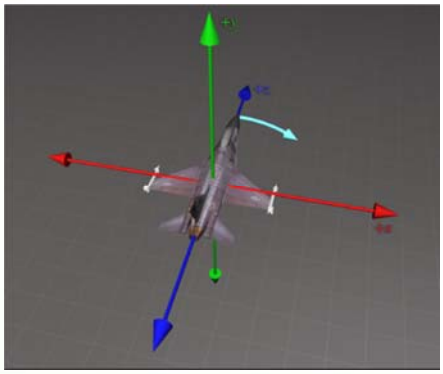
Euler Angles

- Specify orientation as a series of 3 angular displacements from upright space to object space.
- Which axes? Which order?
- Need a convention.
- Heading-pitch-bank
  - Heading: rotation about *y* axis (aka "yaw")
  - Pitch: rotation about *x* axis
  - Bank: rotation about *z* axis (aka "roll")

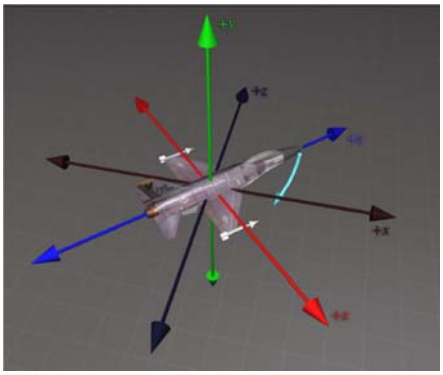
Implementing Euler Angles

- Each game object keeps track of its current heading, pitch, and bank angles (the "Euler angles").
- It also keeps track of its heading, pitch, and bank change rate.
- In each frame, calculate how much the object has changed its heading, pitch, and bank based on the amount of time since the last frame, and add this to the Euler angles.

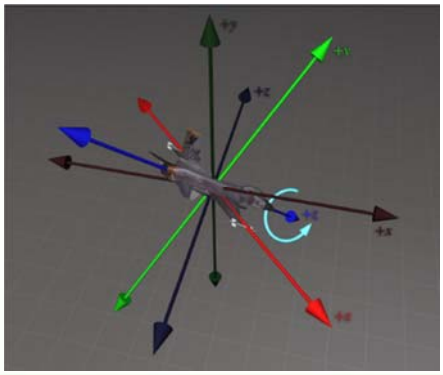
Heading



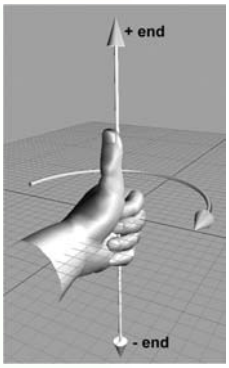
Pitch



Bank



The Sign Matters



- Use the hand rule again.
- Thumb points along positive axis of rotation.
- Fingers curl in direction of positive rotation.

The Order Matters

- Heading is first: it is relative to the upright frame of reference – that is, vertical.
- Pitch is next because it is relative to the horizon. But the *x*-axis may have been moved by the heading change. (Object *x* is no longer the same as upright *x*.)
- Bank is last. The *z*-axis may have been moved by the heading and pitch change. (Object *z* is no longer the same as upright *z*.)

What's in a Name?

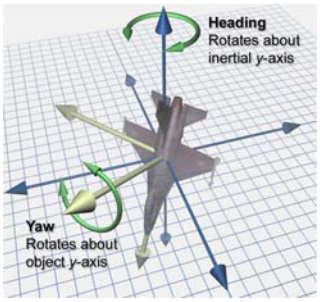
- Heading-pitch-bank is often called *yaw-pitch-roll* (at the time of writing, the latter has a Wikipedia page but the former doesn't)
- Heading is yaw, bank is roll.
- This came from the aerospace industry, where yaw in fact *doesn't* mean heading the way we interpret it.
- Other less common terms are often used.
  - *Heading* also goes by the name *azimuth*.
  - *Pitch* is also called *attitude* or *elevation*.
  - *Bank* is sometimes called *tilt*, or *twist*.

Back to Yaw-pitch-roll

- Perhaps more interesting is that fact that you will often hear these same three words listed in the opposite order: roll-pitch-yaw.
- As it turns out, there is a perfectly reasonable explanation for this backwards convention: it's the order that we actually do the rotations inside a computer!

The Fixed-Axis System

- In an Euler angle system, the axes of rotation are the body axes, which change after each rotation.
- In a *fixed-axis system*, the axes of rotation are the upright space axes.



Example

- The fixed-axis system and the Euler angle system are equivalent, provided that you take the rotations in the opposite order. Let's say we have a heading (yaw) of *h* and a pitch of *p*.
- According to the Euler angle convention:
  - We first do the heading and rotate about the object space *y*-axis by *h*.
  - Then we rotate about the object space *x*-axis by *p*.
- Using a fixed-axis scheme we get to this same ending orientation by doing the rotations in the opposite order.
  - First, we do the pitch, rotating about the upright *x*-axis by *p*.
  - Then, we perform the heading, rotating about the upright *y*-axis by *h*.

More Later

- Although we might visualize Euler angles, inside a computer when rotating vectors from upright space to object space, we will actually use a fixed-axis system.
- We'll discuss this in greater detail when we learn how to convert Euler angles to a rotation matrix.

Advantages of Euler Angles

- Easy for humans to use. Really the only option if you want to enter an orientation by hand.
- Minimal space – 3 numbers per orientation. Bottom line: if you need to store a lot of 3D rotational data in as little memory as possible, as is very common when handling animation data, Euler angles (or exponential map format – to be discussed later) are the best choices.
- Another reason to choose Euler angles when you need to save space is that the numbers you are storing are more easily compressed.
- Every set of 3 numbers makes sense – unlike matrices and quaternions.

Disadvantages of Euler Angles

- Aliasing
- Gimbal (or gymbal) lock
- Interpolation problems

Aliasing

- There are many ways to represent a single orientation, eg. Pitch down 135° is the same as heading 180°, pitch down 45°, then bank 180°.
- This is called the *aliasing* problem.
- Makes it hard to convert orientations from object to world space, eg. “Am I facing East?”

Canonical Euler Angles

- Limit heading (*y*) to $\pm 180°$
- Limit pitch (*x*) to $\pm 90°$
- Limit bank (*z*) to $\pm 180°$
- Now each orientation has a unique canonical Euler angle.
- Except for one more irritating thing.

Gimbal Lock

- Change heading (*y* axis) by 45°
- Pitch down 90°
- Now if you bank, your object space *z* axis is pointing in world space where your object space *y* axis was before you started your rotations. So any rotation about *z* could have been done as a heading change.

Canonical Euler Angles

Fix this (i.e. make canonical Euler angles unique) by insisting that if pitch is 90°, then bank must be zero. Put the bank rotation in the heading instead.

$$\begin{aligned} -180^\circ < h \leqslant 180^\circ \\ -90^\circ \leqslant p \leqslant 90^\circ \\ -180^\circ < b \leqslant 180^\circ \\ p = \pm 90^\circ \quad \Rightarrow \quad b = 0 \end{aligned}$$

Hint for Programmers

- When writing C++ that accepts Euler angle arguments, it's usually best to ensure that they work given Euler angles in any range.
- Luckily this is usually pretty easy, and frequently just automatically works without taking any extra precaution, especially if the angles are fed into trig functions.
- However, when writing code that computes or returns Euler angles, it's a good practice to try to return the canonical Euler angle triple.

More on Gimbal Lock

- It is a common misconception that, because of Gimbal lock, certain orientations cannot be described using Euler angles.
- Actually, for the purposes of describing an orientation, aliasing doesn't pose any problems.
- To be clear, any orientation in 3D can be described using Euler angles, and that representation is unique within the canonical set.
- Also, as we mentioned earlier, there is no such thing as an invalid set of Euler angles. Even if the angles are outside the usual range, we can always agree what orientation is described by the Euler angles.

Aliasing and Gimbal Lock

- What's the fuss about aliasing and Gimbal lock?
- Let's say we wish to interpolate between two orientations $\mathbf{R}_0$ and $\mathbf{R}_1$.
- That is, for a given parameter t, $0 \leq t \leq 1$, we wish to compute an intermediate orientation $\mathbf{R}(t)$ that interpolates smoothly from $\mathbf{R}_0$ to $\mathbf{R}_1$ as t varies from 0 to 1.
- This is extremely useful for character animation and camera control, for example.

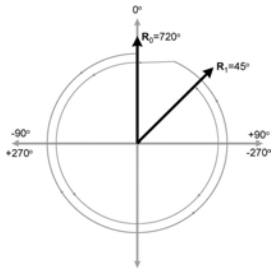
The Dangers of Lerping

The naive approach to this problem is to apply the standard linear interpolation formula (*lerp* for short) to each of the three angles independently.

$$\begin{aligned} \Delta\theta &= \theta_1 - \theta_0 \\ \theta_t &= \theta_0 + t\,\Delta\theta \end{aligned}$$

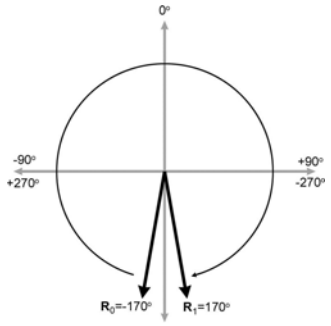
Problems

- Not good with non-canonical angles.
- For example, to interpolate from 720° to 45° in 1° increments would mean turning around twice.



What About Canonical Euler Angles?

- This can even happen with canonical Euler angles, for example, from –170° to 170°.
- It goes the "long way round".



Function wrapPi

- Define the following function:

$$\text{wrapPi}(x) = x - 360^\circ \lfloor (x + 180^\circ)/360^\circ \rfloor$$

- Using wrapPi() makes it easy to take the shortest arc when interpolating between two angles.

$$\Delta\theta = \text{wrapPi}(\theta_1 - \theta_0)$$
$$\theta_t = \theta_0 + t\,\Delta\theta$$

- Here's the code for wrapPi() (next slide):


```
float wrapPi(float theta) {

    // Check if already in range.  This is not strictly necessary,
    // but it will be a very common situation.  We don't want to
    // incur a speed hit and perhaps floating precision loss if
    // it's not necessary
    if (fabs(theta) <= PI) {

        // One revolution is 2 PI.
        const float TWOPPI = 2.0f*PI;

        // Out of range.  Determine how many "revolutions"
        // we need to add.
        float revolutions = floor((theta + PI) * (1.0f/TWOPPI));

        // Subtract it off
        theta -= revolutions*TWOPPI;
    }

    return theta;
}
```


Another Problem

- Even with these two band-aid solutions, you are left with gimbal lock.
- Solution: don't use Euler angles for interpolation. Use something else, like quaternions.

Visualizing Gimbal Lock

- If you have never experienced what Gimbal lock looks like, you may be wondering what all the fuss is about.
- Fortunately, it's easy to find an animation demonstrating the problem: just do a YouTube search for "gimbal lock." For example, http://www.youtube.com/watch?v=zc8b2Jo7mno

Summary of Euler Angles 1

- Euler angles store orientation using three angles. These angles are ordered rotations about the three object-space axes.
- The most common system of Euler angles is the heading-pitch-bank system. Heading and pitch tell which way the object is facing, with heading giving a compass reading and pitch measuring the angle of declination. Bank measures the amount of twist.
- In a fixed-axis system, the rotations occur about the upright axes rather than the moving body axes. This system is equivalent to Euler angles provided that we perform the rotations in the opposite order.
- Lots of smart people use lots of different terms for Euler angles, and they can have good reasons for using different conventions. It's best not to rely on terminology when using Euler angles. Always make sure you get a precise working definition, or you're likely to get very confused.
- In most situations, Euler angles are more intuitive for humans to work with compared to other methods of representing orientation.

Summary of Euler Angles 2

- When memory is at a premium, Euler angles use the minimum amount of data possible for storing an orientation in 3D, and Euler angles are more easily compressed than quaternions.
- There is no such thing as an invalid set of Euler angles. Any three numbers have a meaningful interpretation.
- Euler angles suffer from aliasing problems, due to the cyclic nature of rotation angles, and because the rotations are not completely independent of one another.
- Using canonical Euler angles can simplify many basic queries on Euler angles. An Euler angle triple is in the canonical set if heading and bank are in range –180° to 180° and pitch is in range –90° to 90°. What's more, if pitch is ±90°, then bank is zero.
- Gimbal lock occurs when pitch is ±90°. In this case, one degree of freedom is lost because heading and bank both rotate about the vertical axis.

Summary of Euler Angles 3

- Contrary to popular myth, *any* orientation in 3D can be represented using Euler angles, and we can agree on a unique representation for that orientation within the canonical set.
- The wrapPi function is a very handy tool that simplifies situations where we have to deal with the cyclic nature of angles. Such situation arise frequently in practice, especially in the context of Euler anglers, but at other times as well.
- Simple forms of aliasing are irritating, but there are workarounds. Gimbal lock is a more fundamental problem and no easy solution exists. Gimbal lock is a problem because the parameter space of orientation has a discontinuity. This means small changes in orientation can result in large changes in the individual angles. Interpolation between orientations using Euler angles can freak out or take a wobbly path.

Section 8.4:

Axis-Angle and Exponential Map

Euler's Rotation Theorem

- Euler's Rotation Theorem: any 3D angular displacement can be accomplished via a single rotation about a carefully chosen axis.
- That is, for every pair of orientations $\mathbf{R}_1$ and $\mathbf{R}_2$ there exists an axis $\mathbf{n}$ such that that we can get from $\mathbf{R}_1$ to $\mathbf{R}_2$ by performing a rotation about $\mathbf{n}$.
- Euler's Rotation Theorem leads to two closely-related methods for describing orientation.
- We begin with some notation. Let's say we have chosen a rotation angle θ, and an axis of rotation that passes through the origin and is parallel to the unit vector $\mathbf{n}$.

Exponential Maps

- The two values $\mathbf{n}$ and θ describe an angular displacement in *axis-angle form*.
- Alternatively, since $\mathbf{n}$ has unit length, without loss of information we can multiply it by θ, yielding the single vector $\mathbf{e} = \theta\mathbf{n}$.
- This scheme for describing rotation goes by the rather intimidating and obscure name of *exponential map*.
- The rotation angle can be deduced from the length of $\mathbf{e}$, i.e. $\theta = ||\mathbf{e}||$, and the axis is obtained by normalizing $\mathbf{e}$.
- The exponential map is not only more compact than axis-angle (3 numbers instead of 4), it elegantly avoids certain singularities and has better interpolation and differentiation properties.

Multiples of Angular Displacements

- We can directly obtain a multiple of an angular displacement.
- For example, given a rotation in axis-angle form, we can get 1/3rd of the rotation, or 2.65 times the rotation, simply by multiplying θ by this amount.
- Of course, we can do this same operation with the exponential map just as easily (by multiplying **c**).
- Quaternions can do this through exponentiation, but an inspection of the math reveals that it's really using the axis-angle format under the hood.
- Quaternions can also do a similar operation using slerp, but in a more roundabout way and without the ability for intermediate results to store rotations beyond 180°.
- We'll look at quaternions later.

Exponential Map vs Axis-Angle

- The exponential map gets more use than axis-angle.
- Its interpolation properties are nicer than Euler angles.
- Although it does have singularities (to be discussed below), they are not as troublesome as Euler angles.
- Usually when one thinks of interpolating rotations one immediately thinks of quaternions, but for some applications such as storage of animation data, the under-appreciated exponential map is a viable alternative.
- The most important and frequent use of the exponential map is not for angular displacement, but rather angular *velocity*, because the exponential map differentiates nicely (which is somewhat related to its nicer interpolation properties) and can represent multiple rotations easily.

Aliasing and Singularities

- Like Euler angles, the axis-angle and exponential map forms exhibit aliasing and singularities, although of a slightly more restricted and benign manner.
- There is an obvious singularity at the identity orientation, when θ = 0 and any axis may be used.
- Notice, however, that the exponential map nicely tucks this singularity away, since the multiplication by θ causes **e** to vanish,.
- Another trivial form of aliasing in axis-angle space can be produced by negating both θ and **n**.
- However, the exponential map dodges this issue as well, since negating both θ and **n** leaves **e** = θ**n** unchanged!

Other Aliases

- The other aliases cannot be dispatched so easily.
- As with Euler angles, adding a multiple of 360° to θ produces an angular displacement that results in the same orientation, in both the axis-angle and the exponential map.
- However, this is not always a shortcoming – for describing angular displacement, this ability to represent such extra rotation is an important and useful property.
- For example, it's quite important to be able to distinguish between rotation about the *x*-axis at a rate of 720° per second, versus rotation about the same axis at a rate of 1080° per second, even though these displacements result in the same ending orientation if applied for an integral number of seconds.
- It is not possible to capture this distinction in quaternion format.

More on the Euler Rotation Theorem.

- For any angular displacement described as a rotation matrix, there is a unique exponential map representation.
- Although more than one exponential map may produce the same rotation matrix, it is possible to take a subset of the exponential maps (those for which $||\mathbf{e}|| < 2\pi$) and form a one-to-one correspondence with the rotation matrices.
- This is the essence of the Euler Rotation Theorem.

Concatenating Rotations

- Suppose $\mathbf{e}_1$ and $\mathbf{e}_2$ are rotations in exponential map format.
- The result of performing the rotations in sequence, for example $\mathbf{e}_1$ then $\mathbf{e}_2$, is not the same as performing the rotation $\mathbf{e}_1 + \mathbf{e}_2$, because vector addition is commutative, but three-space rotations are not.
- For example, suppose that $\mathbf{e}_1$ is a 90° downward pitch rotation, and $\mathbf{e}_2$ is a 90° Eastward heading rotation, that is, $\mathbf{e}_1 = [90°, 0, 0]$, and $\mathbf{e}_2 = [0, 90°, 0]$.
- Performing $\mathbf{e}_1$ followed by $\mathbf{e}_2$, we would end up looking downward with our head pointing East, but doing them in the opposite order, we end up on our ear facing East.

Concatenating Small Rotations

- However, if the angles were much smaller, say 2° instead of 90° then the ending orientations would be closer.
- As we reduce the magnitude of the rotation angles, the importance of the order decreases, and at the extreme, for infinitesimal rotations, the order is completely irrelevant.
- Hence for infinitesimal rotations, which are useful for describing angular velocity, exponential maps can be added vectorially,

Terminology Disclaimer

- Alternative names for these concepts abound. We have tried to choose the most standard names we could, but it was difficult to find any consensus in the literature.
- Some authors use the term *axis-angle* to describe both axis-angle and exponential map and don't really distinguish between them.
- Even more confusing is the use of the term *Euler axis* to refer to either form (but not to Euler angles!).
- *Rotation vector* is another term you might see attached to what we are calling *exponential map*.
- Finally, the term *exponential map*, in the broader context of Lie algebra, from whence the term originates, actually refers to an operation (a map) rather than a quantity.

Section 8.5:

Quaternions

Quaternions

- Quaternions were invented by the Irish mathematician Sir William Rowan Hamilton in 1843.
- They have been the cause of much puzzlement to students since then.
- (Image from Wikimedia Commons.)

When Quaternions Were Invented

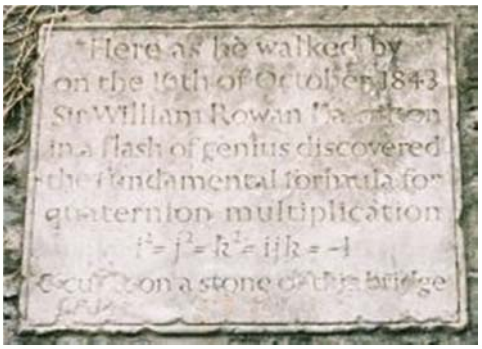
- On the 16th of October, 1843, Hamilton left the Dunsink Observatory in Dublin (where he lived and was serving as Director), and walked along the Royal Canal.
- His destination was the Royal Irish Academy at 19 Dawson Street in the center of Dublin.
- (Image from Wikimedia Commons.)

Nerd Alert!

On his way, as he was passing over Broom (Brougham/Broome) Bridge, he discovered the fundamental quaternion formula:

$$i^2 = j^2 = k^2 = ijk = -1.$$

He was so excited about this that he scratched the formula onto the bridge with his penknife. No trace can be found today, but in 1958 a plaque was erected on the site.



(Image from Wikimedia Commons.)

Quaternions Version 1: Complex Number Notation

This notation is just like complex numbers, but with three imaginary parts, *i, j, k*.

$$q = w + xi + yj + zk.$$

Here's how the imaginary parts interact:

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = k,\ jk = i,\ ki = j$$

$$ji = -k,\ kj = -i,\ ik = -j$$

This is what Hamilton scratched on the bridge.

Quaternions Version 2: Vector-Scalar Notation

Instead of the imaginary number notation:

$$w + xi + yj + zk$$

use vector-scalar notation:

$$\mathbf{q} = [\, w\ \mathbf{v}\,],$$

where *w* is a scalar and **v** a vector. Alternatively:

$$\mathbf{q} = [\, w\ (\, x\ y\ z\,)\,]$$

(it's the same *w, x, y, z*).

Quaternions Version 3: 4D Space

- Hamilton himself thought of quaternions as 4D vectors [*w, x, y, z*].
- However, quaternions are **not** the same as vectors in homogenous 4D space [*x, y, z, w*].
- In particular, his *w* is **not** the same as the *w* in homogenous 4D space.

What was Hamilton Thinking?

- He was looking for a way to extend complex numbers from 2D into 3D.
- That's the normal complex numbers you've probably met before, c = *x* + *yi* where

$$i^2 = -1$$

- What's the link between complex numbers and 2D geometry?

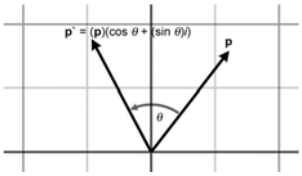
2D and Complex Numbers

- Instead of thinking of a point (*x, y*) in 2D space as two real numbers, think of it as a single complex number *x* + *yi*.
- A rotation by angle θ can be **also** be represented as a complex number

$$\cos\theta + i \sin\theta$$

- Complex numbers are doing double duty here as both points and rotations.

Complex Multiplication



A rotation (represented as a complex number) can be applied to a point (also represented as a complex number) using multiplication of complex numbers.

$$(x + yi)\ (\cos\theta + i \sin\theta)$$
$$= (x \cos\theta - y \sin\theta) + i\ (x \sin\theta + y \cos\theta)$$

What's Going On Here?

Remember the 2D rotation matrix. The imaginary part captures the negative sign on the sine (so to speak).

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

point (x,y) rotation

$$(x + yi)\ (\cos\theta + i \sin\theta)$$
$$= (x \cos\theta - y \sin\theta) + i\ (x \sin\theta + y \cos\theta)$$

rotated point (x',y')

What Do Imaginary Numbers Buy Us?

The imaginary part seems to have two roles.

1. It allows us to "jam together" the *x* and *y* part from a vector [*x, y*] into a single number *x* + *yi*. The imaginary part *i* keeps the *x* and *y* parts separate.
2. It gets us the sign change on the sin θ that we saw in rotation matrices.

So What Was He Thinking?

- Hamilton wanted to extend this to 3D rotations.
- His intuition told him that he needed 2 imaginary parts *i* and *j*. That makes sense, 1 imaginary part for 2D, and 2 imaginary parts for 3D.
- He got fixated on that.
- For some reason his mental block cleared on the bridge: 3 imaginary parts is the way to go.

In the Back of His Mind

- It was known as far back as Euler that axial rotations are closed under composition.
- What does this mean?
- If an object rotates around two or more axes simultaneously, the result is a rotation about a single axis.

Points as Quaternions

- Represent 3D point (*x y z*) as the quaternion
  p = [0 (*x y z*)].
- Or equivalently in complex number notation:
  xi + yj + zk.
- Games generally don't implement the 0 value for *w*.

Rotations as Quaternions

- The scalar part "kind of" represents the angle. The vector part "kind of" represents the axis.
- Rotation by an angle θ around unit axis **n** is represented by the quaternion:

$$\mathbf{Q} = [\cos \theta/2 \quad \sin \theta/2\, \mathbf{n}]$$
$$= [\cos \theta/2 \quad \sin \theta/2\, (\mathbf{n}_x\, \mathbf{n}_y\, \mathbf{n}_z)]$$

Quaternion Negation

- If **q** = [w (x y z)] = [w **v**] is a quaternion, define its negation to be:

$$-\mathbf{q} = -[w\ (x\ y\ z)] = [-w\ (-x\ -y\ -z)]$$
$$= -[w\ \mathbf{v}] = [-w\ -\mathbf{v}].$$

- Surprisingly, **q** = –**q**. The quaternions **q** and –**q** describe the same angular displacement. Any angular displacement in 3D has exactly two distinct representations in quaternion format, and they are negatives of each other.
- It's not too difficult to see why. If we add 360° to θ, it doesn't change the angular displacement represented by **q**, but it negates all four components of **q**.

Quaternion Magnitude

The magnitude of a quaternion is

$$\|\mathbf{q}\| = \|[w \quad (x \quad y \quad z)]\| = \sqrt{w^2 + x^2 + y^2 + z^2}$$
$$= \|[w \quad \mathbf{v}]\| = \sqrt{w^2 + \|\mathbf{v}\|^2}$$

Rotation Quaternion Magnitude

The magnitude of a rotation quaternion is 1.

$$\begin{aligned}
\|\mathbf{q}\| &= \|[w \quad \mathbf{v}]\| = \sqrt{w^2 + \|\mathbf{v}\|^2} \\
&= \sqrt{\cos^2(\theta/2) + (\sin(\theta/2)\|\hat{\mathbf{n}}\|)^2} && \text{(Substituting using } \theta \text{ and } \hat{\mathbf{n}}\text{)} \\
&= \sqrt{\cos^2(\theta/2) + \sin^2(\theta/2)\|\hat{\mathbf{n}}\|^2} \\
&= \sqrt{\cos^2(\theta/2) + \sin^2(\theta/2)\ 1} && (\hat{\mathbf{n}} \text{ is a unit vector.}) \\
&= \sqrt{1} && (\sin^2 x + \cos^2 x = 1) \\
&= 1
\end{aligned}$$

Conjugate and Inverse

The *conjugate* of a quaternion is obtained by reversing the vector part.

$$\mathbf{q}^* = [\, w\ \mathbf{v}\,]^* = [w\ -\mathbf{v}\,].$$

The inverse of a quaternion is its conjugate divided by its magnitude.

$$\mathbf{q}^{-1} = \mathbf{q}^*/||\,\mathbf{q}\,||.$$

For unit quaternions, in particular rotation quaternions, conjugate is the same as inverse

$$[\, w\ \mathbf{v}\,]^* = [\, w\ \mathbf{v}\,]^{-1}.$$

The inverse of a rotation quaternion is a rotation in the opposite direction.

Quaternion Multiplication

$$\begin{aligned}
\mathbf{q}_1\mathbf{q}_2 &= [w_1 \quad (x_1 \quad y_1 \quad z_1)][w_2 \quad (x_2 \quad y_2 \quad z_2)] \\
&= \begin{bmatrix} w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 \\ \begin{pmatrix} w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2 \\ w_1y_2 + y_1w_2 + z_1x_2 - x_1z_2 \\ w_1z_2 + z_1w_2 + x_1y_2 - y_1x_2 \end{pmatrix} \end{bmatrix} \\
&= [w_1 \quad \mathbf{v}_1][w_2 \quad \mathbf{v}_2] \\
&= [w_1w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 \quad w_1\mathbf{v}_2 + w_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2]
\end{aligned}$$

Facts About Quaternion Multiplication

- It is associative but not commutative.

$$\mathbf{q}(\mathbf{rs}) = (\mathbf{qr})\mathbf{s} \qquad \mathbf{qr} \neq \mathbf{rq}$$

- Multiplicative identity is $[1\ \mathbf{0}] = [1\ (\ 0\ 0\ 0\ )]$
- $||\mathbf{qr}|| = ||\mathbf{q}||\ ||\mathbf{r}||$, so unit quaternions are closed under multiplication.
- $(\mathbf{ab})^{-1} = \mathbf{b}^{-1}\mathbf{a}^{-1}$, and in general (just like matrices)

$$(\mathbf{q}_1\mathbf{q}_2\ldots\mathbf{q}_{n-1}\mathbf{q}_n)^{-1} = \mathbf{q}_n^{-1}\mathbf{q}_{n-1}^{-1}\ldots\mathbf{q}_2^{-1}\mathbf{q}_1^{-1}$$

Quaternion Inverse Again

Assume $[\ w\ \mathbf{v}\ ]$ is a unit quaternion (same kind of argument holds for general quaternions). Then,

$$[\ w\ \mathbf{v}\ ]\ [\ w\ \mathbf{v}\ ]^{-1}$$
$$= [\ w\ \mathbf{v}\ ]\ [\ w\ -\mathbf{v}\ ]$$
$$= [\ w^2 - \mathbf{v}.(-\mathbf{v}) \quad \mathbf{v} \times (-\mathbf{v}) + w\mathbf{v} - w\mathbf{v}\ ]$$
$$= [w^2 + x^2 + y^2 + z^2 \quad \mathbf{0}\ ]$$
$$= [\ 1\ \mathbf{0}\ ] \quad \text{(because it's a unit quaternion)}$$

Applying Rotations to Points

- To apply a rotation quaternion **q** to a 3D point *p*, use quaternion multiplication (thinking of *p* for a moment as the quaternion $[1\ \ p]$):

$$\mathbf{q}p\mathbf{q}^{-1}.$$

- Examine what happens when multiple rotations are applied to a vector. Rotate the vector **p** by the quaternion **a**, and then rotate that result by another quaternion **b**.

$$\mathbf{p}' = \mathbf{b}(\mathbf{apa}^{-1})\mathbf{b}^{-1}$$
$$= (\mathbf{ba})\mathbf{p}(\mathbf{a}^{-1}\mathbf{b}^{-1})$$
$$= (\mathbf{ba})\mathbf{p}(\mathbf{ba})^{-1}.$$

Concatenating Rotations

- Notice that rotating by **a** and then by **b** is equivalent to performing a single rotation by the quaternion product **ba**.
- Thus, quaternion multiplication can be used to concatenate multiple rotations, just like matrix multiplication.
- We say "just like matrix multiplication," but in fact there is a slightly irritating difference.
  - With matrix multiplication, our preference to use row vectors puts the vectors on the left, resulting in the nice property that concatenated rotations read left-to-right in the order of transformation.
  - With quaternions, we don't have this flexibility: concatenation of multiple rotations will always read from right to left.

Order of Multiplication

- This means that to apply a rotation quaternion **q** followed by a rotation quaternion **r**, we apply the product quaternion **qr**:

$$\mathbf{r}^{-1}(\mathbf{q}^{-1}p\ \mathbf{q})\mathbf{r} = (\mathbf{qr})^{-1}p\ (\mathbf{qr})$$

- This is cool because quaternion multiplication is done the same order as the transformations.
- Books that use column vectors often define quaternion product backwards – reversing the order of the cross product – to make the order natural for them.

Row vs. Column Vectors

- Quaternion multiplication for row vectors:

$$[\ q_w r_w - \mathbf{q}_v \mathbf{r}_v \quad \mathbf{q}_v \times \mathbf{r}_v + r_w\ \mathbf{q}_v + q_w\ \mathbf{r}_v\ ]$$

- Quaternion multiplication for column vectors:

$$[\ q_w r_w - \mathbf{q}_v \mathbf{r}_v \quad \mathbf{r}_v \times \mathbf{q}_v + r_w\ \mathbf{q}_v + q_w\ \mathbf{r}_v\ ]$$

- Some books do it in the wrong order and just put up with the inconvenience.
- Moral: *Caveat Emptor*

Quaternion Difference

- Let **a** and **b** be quaternions representing two orientations.
- The quaternion **d** that takes orientation **a** to orientation **b** is called the *quaternion difference* between **a** and **b**.
- Want **ad** = **b**. What is **d**?
- $\mathbf{d} = \mathbf{a}^{-1}\mathbf{b}$
- Why? $\mathbf{ad} = \mathbf{a}(\mathbf{a}^{-1}\mathbf{b}) = (\mathbf{a}\mathbf{a}^{-1})\mathbf{b} = [\ 1\ \mathbf{0}\ ]\ \mathbf{b} = \mathbf{b}$

Quaternion Dot Product

- Similar to vector dot product:

$$\mathbf{q}_1.\mathbf{q}_2 = [\ w_1\ \mathbf{v}_1\ ]\ .\ [\ w_2\ \mathbf{v}_2\ ] = w_1 w_2 + \mathbf{v}_1.\mathbf{v}_2$$

- If $\mathbf{v}_1 = [\ x_1\ y_1\ z_1\ ]$ and $\mathbf{v}_2 = [\ x_2\ y_2\ z_2\ ]$, then:

$$\mathbf{q}_1.\mathbf{q}_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$$

- Geometric interpretation: the larger the absolute value of $\mathbf{q}_1.\mathbf{q}_2$, the more similar their orientations are.

Quaternion Log

- As a shorthand, let $\alpha = \theta/2$. Let **n** be a unit vector. Suppose quaternion $\mathbf{q} = [\cos\alpha\ \ \mathbf{n}\sin\alpha]$
- The *logarithm* of **q**, denoted log **q**, is defined to be $[0, \alpha\mathbf{n}]$.
- Note that log **q** is not necessarily a unit quaternion.
- Also note the similarity between the *logarithm of a quaternion* and the *exponential map format*.

Quaternion Exponential

- The exponential function for quaternions is defined in the exact opposite manner.
- Suppose $\mathbf{p} = [0, \alpha\mathbf{n}]$, where **n** is a unit vector.
- The exponential of **p**, denoted exp **p**, is defined to be $[\cos\alpha\ \ \mathbf{n}\sin\alpha]$.
- Note that exp **p** is always a unit quaternion.
- Quaternion log and exponential are inverses, that is, for all quaternions **q**,

$$\exp(\log \mathbf{q}) = \log(\exp \mathbf{q}) = \mathbf{q}.$$

Multiplying a Quaternion by a Scalar

- Given a scalar *k* and a quaternion $\mathbf{q} = [w\ \mathbf{v}]$,

$$k\mathbf{q} = k[w\ \mathbf{v}] = [kw\ k\mathbf{v}].$$

- This will not usually result in a unit quaternion, which is why multiplication by a scalar is not for representing angular displacement.

Quaternion Exponentiation

- We used the term "exponential" before, now it's time for exponentiation.
- If **q** is a quaternion and *t* is a scalar, define

$$\mathbf{q}^t = \exp(t \log \mathbf{q}).$$

- As *t* varies from 0 to 1, the quaternion $\mathbf{q}^t$ varies from $[1, \mathbf{0}]$ to **q**.
- Quaternion exponentiation is useful because it allows us to extract a fraction of an angular displacement.
- For example, $\mathbf{q}^{1/3}$ is a quaternion that represents 1/3rd of the angular displacement represented by the quaternion **q**.

Facts About Quaternion Exponentiation

- Exponents t outside the range $0 \le t \le 1$ behave mostly as expected, with one major caveat.
- For example, $\mathbf{q}^2$ represents twice the angular displacement as $\mathbf{q}$.
- If $\mathbf{q}$ represents a clockwise rotation of 30° about the x-axis, then $\mathbf{q}^2$ represents a clockwise rotation of 60° about the x-axis, and $\mathbf{q}^{-1/3}$ represents a counterclockwise rotation of 10° about the x-axis.
- Notice that $\mathbf{q}^{-1}$ yields the quaternion inverse.

The Caveat

- The caveat we mentioned is this: a quaternion represents angular displacements using the shortest arc. Multiple spins cannot be represented.
- For example, if $\mathbf{q}$ represents a clockwise rotation of 30° about the x-axis, then $\mathbf{q}^8$ is not a 240° clockwise rotation about the x-axis as expected; it is a 120° counterclockwise rotation.
- Of course, rotating 240° in one direction produces the same end result as rotating 120° in the opposite direction, and this is the point: *quaternions only really capture the end result.*

In Consequence

- If further operations on this quaternion were performed, things will not behave as expected. For example, $(\mathbf{q}^8)^{1/2}$ is not $\mathbf{q}^4$, as we would intuitively expect.
- In general, many of the algebraic identities concerning exponentiation of scalars, such as $(a^s)^t = a^{st}$, do not apply to quaternions.
- In some situations, we do care about the total amount of rotation, not just the end result.
- The most important example is that of angular velocity.
- Quaternions are not the correct tool for this job; use the exponential map (or its cousin, the axis-angle format) instead.

Implementation Notes

- Here's why $\mathbf{q}^t$ interpolates from the identity quaternion to $\mathbf{q}$ as t varies from 0 to 1.
  - The log operation essentially converts the quaternion to exponential map format. (Except for a factor of 2.)
  - Then, when we perform the scalar multiplication by the exponent t, the effect is to multiply the angle by t.
  - Finally, the exp undoes what the log operation did, recalculating the new w and $\mathbf{v}$ from the exponential vector.
- Direct implementation of the formula $\mathbf{q}^t = \exp(t \log \mathbf{q})$ as an algorithm for computing $\mathbf{q}^t$ can give rise to code that is more complicated than necessary.
- Instead of working with a single exponential-map-like quantity like the formula tells us to, we will break out the axis and half-angle separately.


```
// Quaternion (input and output)
float w,x,y,z;

// Input exponent
float exponent;

// Check for the case of an identity quaternion.
// This will protect against divide by zero
if (fabs(w) < .9999f) {

    // Extract the half angle alpha (alpha = theta/2)
    float alpha = acos(w);

    // Compute new alpha value
    float newAlpha = alpha * exponent;

    // Compute new w value
    w = cos(newAlpha);

    // Compute new xyz values
    float mult = sin(newAlpha) / sin(alpha);
    x *= mult; y *= mult; z *= mult;
}
```


Notes About Code 1

The check for the identity quaternion is necessary since a value of w = 1 would cause the computation of mult to divide by zero.

```
// Quaternion (input and output)
float w,x,y,z;

// Input exponent
float exponent;

// Check for the case of an identity quaternion.
// This will protect against divide by zero
if (fabs(w) < .9999f) {

    // Extract the half angle alpha (alpha = theta/2)
    float alpha = acos(w);

    // Compute new alpha value
    float newAlpha = alpha * exponent;

    // Compute new w value
    w = cos(newAlpha);

    // Compute new xyz values
    float mult = sin(newAlpha) / sin(alpha);
    x *= mult; y *= mult; z *= mult;
}
```

Notes About Code 2

Raising an identity quaternion to any power results in the identity quaternion, so if we detect an identity quaternion on input, we simply ignore the exponent and return the original quaternion.

```
// Quaternion (input and output)
float w,x,y,z;

// Input exponent
float exponent;

// Check for the case of an identity quaternion.
// This will protect against divide by zero
if (fabs(w) < .9999f) {

    // Extract the half angle alpha (alpha = theta/2)
    float alpha = acos(w);

    // Compute new alpha value
    float newAlpha = alpha * exponent;

    // Compute new w value
    w = cos(newAlpha);

    // Compute new xyz values
    float mult = sin(newAlpha) / sin(alpha);
    x *= mult; y *= mult; z *= mult;
}
```


Notes About Code 3

We use the arccos function, which always returns a positive angle, to compute alpha. Any quaternion can be interpreted as having a positive angle of rotation, since negative rotation is the same as positive rotation about the opposite axis.

```
// Quaternion (input and output)
float w,x,y,z;

// Input exponent
float exponent;

// Check for the case of an identity quaternion.
// This will protect against divide by zero
if (fabs(w) < .9999f) {

    // Extract the half angle alpha (alpha = theta/2)
    float alpha = acos(w);

    // Compute new alpha value
    float newAlpha = alpha * exponent;

    // Compute new w value
    w = cos(newAlpha);

    // Compute new xyz values
    float mult = sin(newAlpha) / sin(alpha);
    x *= mult; y *= mult; z *= mult;
}
```


Quaternion Interpolation

- The *raison d'etre* of quaternions in games and graphics today is an operation known as *slerp*, which stands for **s**pherical **l**inear int**erp**olation.
- Slerp allows us to smoothly interpolate between two orientations while avoiding all the problems that plagued interpolation of Euler angles.
- Slerp is a ternary operator, meaning it accepts three operands, 2 quaternions and a scalar.

The Slerp Function

- The first two operands the two quaternions $\mathbf{q}_0$ and $\mathbf{q}_1$ between which we wish to interpolate.
- The third operand is the interpolation parameter, a real number t such that $0 \le t \le 1$.
- As t varies from 0 to 1, the slerp function slerp($\mathbf{q}_0$, $\mathbf{q}_1$, t) will return an orientation that interpolates from $\mathbf{q}_0$ to $\mathbf{q}_1$ by fraction t.

Scalar Linear Interpolation

To interpolate between two scalar values a_0 and a_1, use the standard linear interpolation formula:

$$\Delta a = a_1 - a_0$$

$$\text{lerp}(a_0, a_1, t) = a_0 + t.\Delta a$$

This requires three basic steps:

1. Compute the difference between the two values
2. Take a fraction of this difference
3. Take the first value and adjust it by this fraction of the difference.

Slerp By Analogy to Lerp

We use the same idea to interpolate between orientations. Recall that quaternion multiplication reads right-to-left.

1. Compute the difference between the two values. The angular displacement from $\mathbf{q}_0$ to $\mathbf{q}_1$ is given by
   $$\Delta\mathbf{q} = \mathbf{q}_1\mathbf{q}_0^{-1}.$$
2. Take a fraction of this difference using quaternion exponentiation. The fraction of the difference is given by
   $$(\Delta\mathbf{q})^t.$$
3. Take the original value and adjust it by this fraction of the difference by composing the angular displacements using quaternion multiplication, giving
   $$(\Delta\mathbf{q})^t\,\mathbf{q}_0.$$

The Slerp Equation

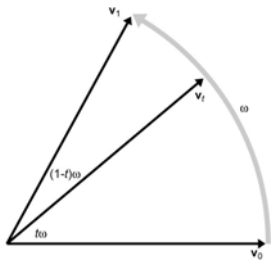
- Putting these steps together,

  $$\text{slerp}(\mathbf{q}_0, \mathbf{q}_1, t) = (\mathbf{q}_1\mathbf{q}_0^{-1})^t\,\mathbf{q}_0.$$

- This is how slerp is computed in theory. In practice, a more efficient technique is used.
- We start by interpreting the quaternions as existing in a 4D space. We are only interested in unit quaternions, which live on the surface of a 4D hypersphere.
- We interpolate around the arc that connects the two quaternions, along the surface of the 4D hypersphere.
- Hence the name *spherical linear interpolation*.

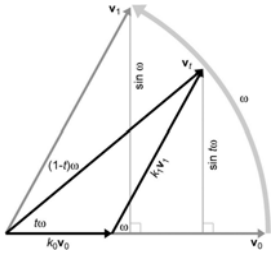
Visualize It in the Plane

- Given unit length 2D vectors $\mathbf{v}_0$ and $\mathbf{v}_1$, compute $\mathbf{v}_t$, the result of smoothly interpolating around the arc by a fraction t of the distance from $\mathbf{v}_0$ to $\mathbf{v}_1$, for some real number t such that $0 \le t \le 1$.
- If we let ω be the angle intercepted by the arc from $\mathbf{v}_0$ to $\mathbf{v}_1$, then $\mathbf{v}_t$ is the result of rotating $\mathbf{v}_0$ around this arc by an angle of $t\omega$.



Visualize It in the Plane

- Express $\mathbf{v}_t$ as a linear combination of $\mathbf{v}_0$ and $\mathbf{v}_1$.
- That is, find nonnegative constants k_0 and k_1 such that $\mathbf{v}_t = k_0\,\mathbf{v}_0 + k_1\,\mathbf{v}_1$.
- Use elementary geometry to determine the values of k_0 and k_1 as in this diagram.

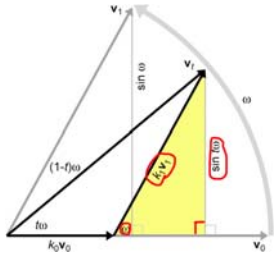


Solving for k_1

- Applying some trig to the right triangle with hypotenuse $k_1\,\mathbf{v}_1$ (recalling that $\mathbf{v}_1$ is a unit vector), we see that:

$$\sin\omega = \frac{\sin t\omega}{k_1}$$

$$k_1 = \frac{\sin t\omega}{\sin\omega}$$



Continuing

Similarly,

$$k_0 = \frac{\sin(1-t)\omega}{\sin\omega}$$

And therefore,

$$\mathbf{v}_t = k_0\mathbf{v}_0 + k_1\mathbf{v}_1 = \frac{\sin(1-t)\omega}{\sin\omega}\mathbf{v}_0 + \frac{\sin t\omega}{\sin\omega}\mathbf{v}_1$$

Generalizing to quaternion space:

$$\text{slerp}(\mathbf{q}_0, \mathbf{q}_1, t) = \frac{\sin(1-t)\omega}{\sin\omega}\mathbf{q}_0 + \frac{\sin t\omega}{\sin\omega}\mathbf{q}_1$$

Computing ω

- We need a way to compute ω, the angle between the two quaternions.
- As it turns out, the analogy from 2D vector math can be carried into quaternion space; the quaternion dot product is cos ω.

Complication 1

- There are two slight complications. First, the two quaternions **q** and –**q** represent the same orientation, but may produce different results when used as an argument to slerp.
- This problem doesn't happen in 2D or 3D because the surface of a 4D hypersphere has a different topology than Euclidian space.
- The solution is to choose the signs of $\mathbf{q}_0$ and $\mathbf{q}_1$ such that the dot product $\mathbf{q}_0 \cdot \mathbf{q}_1$ is nonnegative.
- This has the effect of always selecting the shortest rotational arc from $\mathbf{q}_0$ to $\mathbf{q}_1$.

Complication 2

- The second consideration is that if $\mathbf{q}_0$ and $\mathbf{q}_1$ are very close, then ω is very small, and thus sin ω is also very small, which will cause problems with the division.

$$\text{slerp}(\mathbf{q}_0, \mathbf{q}_1, t) = \frac{\sin(1-t)\omega}{\sin\omega}\mathbf{q}_0 + \frac{\sin t\omega}{\sin\omega}\mathbf{q}_1$$

- To avoid this, if sin ω is very small, then use simple linear interpolation instead.
- Next: some C code…


```
// The two input quaternions
float w0,x0,y0,z0;
float w1,x1,y1,z1;

// The interpolation parameter
float t;

// The output quaternion will be computed here
float w,x,y,z;

// Compute the "cosine of the angle" between the
// quaternions, using the dot product
float cosOmega = w0*w1 + x0*x1 + y0*y1 + z0*z1;

// If negative dot, negate one of the input
// quaternions, to take the shorter 4D "arc"
if (cosOmega < 0.0f) {
    w1 = -w1;
    x1 = -x1;
    y1 = -y1;
    z1 = -z1;
    cosOmega = -cosOmega;
}
```



```
// Check if they are very close together, to protect
// against divide-by-zero
float k0, k1;
if (cosOmega > 0.9999f) {

    // Very close - just use linear interpolation
    k0 = 1.0f-t; k1 = t;
} else {

    // Compute the sin of the angle using the
    // trig identity sin^2(omega) + cos^2(omega) = 1
    float sinOmega = sqrt(1.0f - cosOmega*cosOmega);

    // Compute the angle from its sine and cosine
    float omega = atan2(sinOmega, cosOmega);

    // Compute inverse of denominator, so we only have
    // to divide once
    float oneOverSinOmega = 1.0f / sinOmega;

    // Compute interpolation parameters
    k0 = sin((1.0f - t) * omega) * oneOverSinOmega;
    k1 = sin(t * omega) * oneOverSinOmega;
}

// Interpolate
w = w0*k0 + w1*k1; x = x0*k0 + x1*k1;
y = y0*k0 + y1*k1; z = z0*k0 + z1*k1;
```


Advantages of Quaternions

- **Smooth interpolation**. The interpolation provided by slerp provides smooth interpolation between orientations. No other representation method provides for smooth interpolation.
- **Fast concatenation and inversion of angular displacements.** We can concatenate a sequence of angular displacements into a single angular displacement using the quaternion cross product. The same operation using matrices involves more scalar operations, though which one is actually faster on a given architectures is not so clean-cut: SIMD vector operations can make very quick work of matrix multiplication. Quaternion conjugate provides a way to compute the opposite angular displacement very efficiently. This can be done by transposing a rotation matrix, but is not easy with Euler angles.
- **Fast conversion to and from matrix form**. As we will see later, quaternions can be converted to and from matrix form a bit faster than Euler angles.
- **Only four numbers**. Since a quaternion contains four scalar values, it is considerably more economical than a matrix, which uses nine numbers. (However, it still is 33% larger than Euler angles.)

Disadvantages of Quaternions

However, these advantages do come at some cost. Quaternions suffer from a few of the problems that affect matrices, only to a lesser degree:

- **Slightly bigger than Euler angles**. That one additional number may not seem like much, but an extra 33% can make a difference when large amounts of angular displacements are needed, for example, when storing animation data. And the values inside a quaternion are not evenly spaced from –1 to 1 so the component values do not interpolate smoothly even if the orientation does. This makes quaternions more difficult to pack into a fixed point number than Euler angles or an exponential map.
- **Can become invalid**. This can happen either through bad input data, or from accumulated floating point round off error. (We can of course normalize the quaternion to ensure that it has unit magnitude.)
- **Difficult for humans to work with**. Of the three representation methods, quaternions are the most difficult for humans to work with directly.

Section 8.6:

Comparison of Methods

Comparison of Methods 1

Rotating points between coordinate spaces (object and upright)

- **Matrix**: Possible, and often highly optimized by vector instructions.
- **Euler Angles**: Impossible (must convert to rotation matrix)
- **Exponential Map**: Impossible (must convert to rotation matrix)
- **Quaternion**: On a chalkboard, yes. Practically, in a computer, not really. You might as well convert to rotation matrix.

Concatenation of multiple rotations

- **Matrix**: Possible, and can often be highly optimized by vector instructions. Watch out for matrix creep.
- **Euler Angles**: Impossible.
- **Exponential Map**: Impossible.
- **Quaternion**: Possible. Fewer scalar operations than matrix multiplication, but might not as easy to take advantage of SIMD instructions. Watch out for error creep.

Comparison of Methods 2

Inversion of rotations

- **Matrix**: Easy and fast, using matrix transpose
- **Euler Angles**: Not easy.
- **Exponential Map**: Easy and fast, using vector negation
- **Quaternion**: Easy and fast, using quaternion conjugate

Interpolation

- **Matrix**: Extremely problematic
- **Euler Angles**: Possible, but Gimbal lock causes quirkiness
- **Exponential Map**: Possible, with some singularities, but not as troublesome as Euler angles.
- **Quaternion**: Slerp provides smooth interpolation

Comparison of Methods 3

Direct human interpretation

- **Matrix**: Difficult
- **Euler Angles**: Easiest
- **Exponential Map**: Very difficult
- **Quaternion**: Very difficult

Storing in a memory or in a file

- **Matrix**: Nine numbers
- **Euler Angles**: Three numbers that can be easily quantized
- **Exponential Map**: Three numbers that can be easily quantized
- **Quaternion**: 4 numbers that don't quantize well; can be reduced to 3 by assuming unit length and that 4^{th} component is nonnegative.

Comparison of Methods 4

Unique representation for a given rotation

- **Matrix**: Yes
- **Euler Angles**: No, due to aliasing
- **Exponential Map**: No, due to aliasing; less complicated than Euler angles.
- **Quaternion**: Exactly two distinct representations for any angular displacement, and they are negatives of each other

Possible to become invalid

- **Matrix**: Six degrees of redundancy inherent in orthogonal matrix. Matrix creep can occur.
- **Euler Angles**: Any three numbers can be interpreted unambiguously
- **Exponential Map**: Any three numbers can be interpreted unambiguously
- **Quaternion**: Error creep can occur

How to Choose Representation 1

- Euler angles are easiest for humans to work with. Using Euler angles greatly simplifies human interaction when specifying the orientation of objects in the world.
- This includes direct keyboard entry of an orientation, specifying orientations directly in the code (e.g. positioning the camera for rendering), and examination in the debugger.
- This advantage should not be underestimated. Certainly don't sacrifice ease of use in the name of optimization until you are certain that your optimization will make a difference in practice.

How to Choose Representation 2

- Matrix form must eventually be used if vector coordinate space transformations are needed.
- However, this doesn't mean you can't store the orientation using another format and then generate a rotation matrix when you need it.
- An alternative solution is to store the main copy of the orientation using Euler angles or a quaternion, but also maintain a matrix for rotations, re-computing this matrix anytime the main copy changes.

How to Choose Representation 3

- For storage of large numbers of orientations (e.g. animation data), Euler angles, exponential maps, and quaternions offer various tradeoffs.
- In general, the components of Euler angles and exponential maps quantize better than quaternions.
- It is possible to store a rotation quaternion in only three numbers; by assuming the fourth component is nonnegative and the quaternion has unit length it can be computed from the three that are stored.

How to Choose Representation 3

- Reliable quality interpolation can only be accomplished using quaternions.
- However, even if you are using a different form you can always convert to quaternions, perform the interpolation, and then convert back to the original form.
- Direct interpolation using exponential maps might be a viable alternative in some cases, since the points of singularity are at very extreme orientations and are often easily avoided in practice.

How to Choose Representation 4

For angular velocity or any other situation where extra spins over and above 360° need to be represented, use the exponential map or axis-angle.

Section 8.7:

Converting Between Representations

Conversion Algorithms

1. Euler angles to rotation matrix.
2. Rotation matrix to Euler angles.
3. Quaternion to matrix.
4. Matrix to quaternion.
5. Euler angles to quaternion.
6. Quaternion to Euler angles.

Euler Angles to Matrix 1

- Euler angles represent a transformation of inertial space axes to object space axes.
- We want a matrix to transform points.
- Of course, it matters whether we want to go from inertial to object space or the other way around.
- Let's start with object to upright space, since that's how Euler angles are defined.
- Since transforming axes is the opposite of transforming points, we need to negate the angles.

Euler Angles to Matrix 2

- Create rotation matrices for heading, pitch, and bank angles, then multiply the matrices.

$$\mathbf{M}_{object \to upright} = \mathbf{BPH}$$

- **B**, **P**, and **H** are the rotation matrices for bank, heading, and pitch, which rotate about the *z*, *x*, and *y*-axis, respectively.

Euler Angles to Matrix 3

$$\mathbf{B} = \mathbf{R}_z(b) = \begin{bmatrix} \cos b & \sin b & 0 \\ -\sin b & \cos b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P} = \mathbf{R}_x(p) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos p & \sin p \\ 0 & -\sin p & \cos p \end{bmatrix}$$

$$\mathbf{H} = \mathbf{R}_y(h) = \begin{bmatrix} \cos h & 0 & -\sin h \\ 0 & 1 & 0 \\ \sin h & 0 & \cos h \end{bmatrix}$$

Euler Angles to Matrix 4

$$\mathbf{M}_{object \to upright} = \mathbf{BPH}$$

$$= \begin{bmatrix} \cos h \cos b + \sin h \sin p \sin b & \sin b \cos p & -\sin h \cos b + \cos h \sin p \sin b \\ -\cos h \sin b + \sin h \sin p \cos b & \cos b \cos p & \sin b \sin h + \cos h \sin p \cos b \\ \sin h \cos p & -\sin p & \cos h \cos p \end{bmatrix}$$

Euler Angles to Matrix 5

- To transform points from object to upright space, it's just the inverse of the matrix we just saw.
- And the inverse of a rotation matrix is its transpose.

$$\mathbf{M}_{upright \to object} = (\mathbf{M}_{object \to upright})^{-1}$$
$$= (\mathbf{M}_{object \to upright})^{T}$$

Matrix to Euler Angles 1

- We must know which rotation the matrix performs: either object-to-upright or upright-to-object. Here, we will develop a technique using the upright-to-object matrix.
- The process of converting an object-to-upright matrix to Euler angles is supposedly very similar.
- For any given angular displacement, there are an infinite number of Euler angle representations due to Euler angle aliasing.
- The technique presented here will always return canonical Euler angles, with heading and bank in range ±180° and pitch in range ±90°.

Matrix to Euler Angles 2

- Some matrices may be malformed, and so we must be tolerant of floating point precision errors.
- Some matrices contain transformations other than rotation, such as scale, mirroring, or skew.
- The technique described here only works on proper rotation matrices, perhaps with the usual floating point imprecision, but nothing grossly out of orthogonality.
- If it's given a non-orthogonal matrix, the results are unpredictable.
- With those considerations in mind, we set out to solve for the Euler angles from the rotation matrix equation directly.

Matrix to Euler Angles 3

$$\mathbf{M}_{object \to upright} = \begin{bmatrix} \cos h \cos b + \sin h \sin p \sin b & \sin b \cos p & -\sin h \cos b + \cos h \sin p \sin b \\ -\cos h \sin b + \sin h \sin p \cos b & \cos b \cos p & \sin b \sin h + \cos h \sin p \cos b \\ \sin h \cos p & -\sin p & \cos h \cos p \end{bmatrix}$$

Solve for p from m_{32}

$$m_{32} = -\sin p$$

$$p = \text{asin}(-m_{32})$$

Matrix to Euler Angles 4

- The C standard library function `asin()` returns a value in the range $-\pi/2$ to $\pi/2$ radians, which is –90° to +90°, exactly the range of values we wish to return for pitch.
- Now that we know p, we also know $\cos p$. Let us first assume that $\cos p \neq 0$. We can determine $\sin h$ and $\cos h$ by dividing m_{31} and m_{33} by $\cos p$, as follows:

$$m_{31} = \sin h \cos p$$

$$\therefore \sin h = m_{31} / \cos p$$

and

$$m_{33} = \cos h \cos p$$

$$\therefore \cos h = m_{33} / \cos p$$

Matrix to Euler Angles 5

Once we know the sine and cosine of an angle, we can compute the value of the angle using the SDK function atan2f().

$$\tan h = \sin h / \cos h$$

$$h = \text{atan}(\sin h / \cos h) = \text{atan2}(\sin h, \cos h)$$

This function returns an angle from $-\pi$ to π radians, which is –180° to +180°, our desired output range.

Matrix to Euler Angles 6

- `atan2`(y, x) works by taking the arctangent of y/x using the signs of the two arguments to determine the quadrant of the returned angle. Since $\cos p > 0$, the divisions do not affect the quotient and are, therefore, unnecessary.
- Thus, heading can be computed more simply by:

$$h = \texttt{atan2}(\sin h, \cos h)$$

$$= \texttt{atan2}(m_{13}/\cos p, m_{33}/\cos p)$$

$$= \texttt{atan2}(m_{13}, m_{33})$$

Matrix to Euler Angles 7

- Now we have pitch and heading. Bank is similar to heading:

$$b = \texttt{atan2}(m_{21}, m_{22})$$

- We skipped the case $\cos p = 0$.
- If $\cos p = 0$, then $p = \pm 90°$ (looking straight up or straight down).
- This is the gimbal lock case.

Matrix to Euler Angles 8

Arbitrarily assign all rotation about the vertical axis to heading and set bank equal to zero. So,

$$\cos p = 0$$

$$b = 0$$

Which means

$$\sin b = 0$$

$$\cos b = 1$$

Matrix to Euler Angles 9

$$\begin{aligned}
&\mathbf{M}_{object \to upright} \\
&= \begin{bmatrix} \cos h \cos b + \sin h \sin p \sin b & \sin b \cos p & -\sin h \cos b + \cos h \sin p \sin b \\ -\cos h \sin b + \sin h \sin p \cos b & \cos b \cos p & \sin b \sin h + \cos h \sin p \cos b \\ \sin h \cos p & -\sin p & \cos h \cos p \end{bmatrix} \\
&= \begin{bmatrix} \cos h(1) + \sin h \sin p(0) & (0)(0) & -\sin h(1) + \cos h \sin p(0) \\ -\cos h(0) + \sin h \sin p(1) & (1)(0) & (0) \sin h + \cos h \sin p(1) \\ \sin h(0) & -\sin p & \cos h(0) \end{bmatrix} \\
&= \begin{bmatrix} \cos h & 0 & -\sin h \\ \sin h \sin p & 0 & \cos h \sin p \\ 0 & -\sin p & 0 \end{bmatrix}.
\end{aligned}$$

Matrix to Euler Angles 10

```
// Assume the matrix is stored in these variables:
float m11,m12,m13;
float m21,m22,m23;
float m31,m32,m33;

// We will compute the Euler angle values in radians and store them here:
float h,p,b;

// Extract pitch from m32, being careful for domain errors with
// asin(). We could have values slightly out of range due to
// floating point arithmetic.
float sp = -m32;
if (sp <= -1.0f) {
    p = -1.570796f; // -pi/2
} else if (sp >= 1.0) {
    p = 1.570796; // pi/2
} else {
    p = asin(sp);
}
```


Matrix to Euler Angles 11

```
// Check for the Gimbal lock case, giving a slight tolerance
// for numerical imprecision
if (fabs(sp) > 0.9999f) {

    // We are looking straight up or down.
    // Slam bank to zero and just set heading
    b = 0.0f;
    h = atan2(-m13, m11);

} else {

    // Compute heading from m13 and m33
    h = atan2(m31, m33);

    // Compute bank from m21 and m22
    b = atan2(m12, m22);
}
```


Quaternion to Matrix 1

Start by reverse engineering the matrix for rotating around an arbitrary axis **n** = (n_x, n_y, n_z) from Chapter 5.

$$\begin{bmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_x n_y(1-\cos\theta)+n_z\sin\theta & n_x n_z(1-\cos\theta)-n_y\sin\theta \\ n_x n_y(1-\cos\theta)-n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_y n_z(1-\cos\theta)+n_x\sin\theta \\ n_x n_z(1-\cos\theta)+n_y\sin\theta & n_y n_z(1-\cos\theta)-n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta \end{bmatrix}$$

See if we can rearrange it so that the parts of a quaternion leap out at us.

Quaternion to Matrix 2

We're trying to manipulate the matrix so that the following values pop out at us:

$$w = \cos(\theta/2)$$

$$x = n_x \sin(\theta/2)$$

$$y = n_y \sin(\theta/2)$$

$$z = n_y \sin(\theta/2)$$

there are really only two major cases to handle: the diagonal elements and the off-diagonal elements.

Quaternion to Matrix 3

Let's start with the diagonal elements of the matrix. We'll work through m_{11} here; m_{22} and m_{33} can be solved similarly.

$$\begin{bmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_xn_y(1-\cos\theta)+n_z\sin\theta & n_xn_z(1-\cos\theta)-n_y\sin\theta \\ n_xn_y(1-\cos\theta)-n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_yn_z(1-\cos\theta)+n_x\sin\theta \\ n_xn_z(1-\cos\theta)+n_y\sin\theta & n_yn_z(1-\cos\theta)-n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta \end{bmatrix}$$

Quaternion to Matrix 4

$$\begin{aligned} m_{11} &= n_x^2(1-\cos\theta)+\cos\theta \\ &= n_x^2 - n_x^2\cos\theta + \cos\theta \\ &= 1 - 1 + n_x^2 - n_x^2\cos\theta + \cos\theta \\ &= 1 - (1 - n_x^2 + n_x^2\cos\theta - \cos\theta) \\ &= 1 - (1 - \cos\theta - n_x^2 + n_x^2\cos\theta) \\ &= 1 - (1 - n_x^2)(1 - \cos\theta) \end{aligned}$$

Quaternion to Matrix 5

- Now we need to replace the cos θ term with something that contains cos $\theta/2$ or sin $\theta/2$, since the components of a quaternion contain those terms.
- Let $\alpha = \theta/2$. We'll write one of the double-angle formulas for cosine from Section 1.4.4 in terms of α, and then substitute in θ.

$$\cos 2\alpha = 1 - 2\sin^2\alpha$$
$$\cos\theta = 1 - 2\sin^2(\theta/2)$$

Quaternion to Matrix 6

Substituting for cos θ, we have

$$\begin{aligned} m_{11} &= 1 - (1 - n_x^2)(1 - \cos\theta) \\ &= 1 - (1 - n_x^2)\left(1 - \left(1 - 2\sin^2(\theta/2)\right)\right) \\ &= 1 - (1 - n_x^2)\left(2\sin^2(\theta/2)\right). \end{aligned}$$

Quaternion to Matrix 7

Since **n** is a unit vector, $n_x^2 + n_y^2 + n_z^2 = 1$, and therefore $1 - n_x^2 = n_y^2 + n_z^2$, so:

$$\begin{aligned} m_{11} &= 1 - (1 - n_x^2)\left(2\sin^2(\theta/2)\right) \\ &= 1 - (n_y^2 + n_z^2)\left(2\sin^2(\theta/2)\right) \\ &= 1 - 2n_y^2\sin^2(\theta/2) - 2n_z^2\sin^2(\theta/2) \\ &= 1 - 2y^2 - 2z^2. \end{aligned}$$

As we said, elements m_{22} and m_{33} are derived in a similar fashion.

Quaternion to Matrix 8

Now let's look at the non-diagonal elements of the matrix; they are easier than the diagonal elements. We'll use m_{12} as an example.

$$\begin{bmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_xn_y(1-\cos\theta)+n_z\sin\theta & n_xn_z(1-\cos\theta)-n_y\sin\theta \\ n_xn_y(1-\cos\theta)-n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_yn_z(1-\cos\theta)+n_x\sin\theta \\ n_xn_z(1-\cos\theta)+n_y\sin\theta & n_yn_z(1-\cos\theta)-n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta \end{bmatrix}$$

We'll need the reverse of the double-angle formula for sine.

$$\sin 2\alpha = 2\sin\alpha\cos\alpha$$
$$\sin\theta = 2\sin(\theta/2)\cos(\theta/2)$$

Quaternion to Matrix 9

$$\begin{aligned} m_{12} &= n_x n_y (1 - \cos\theta) + n_z \sin\theta \\ &= n_x n_y \left(1 - \left(1 - 2\sin^2(\theta/2)\right)\right) + n_z \left(2\sin(\theta/2)\cos(\theta/2)\right) \\ &= n_x n_y \left(2\sin^2(\theta/2)\right) + 2n_z \sin(\theta/2)\cos(\theta/2) \\ &= 2\left(n_x \sin(\theta/2)\right)\left(n_y \sin(\theta/2)\right) + 2\cos(\theta/2)\left(n_z \sin(\theta/2)\right) \\ &= 2xy + 2wz \end{aligned}$$

As we said, the other nondiagonal elements are derived in a similar fashion.

Quaternion to Matrix 10

Finally, we present the complete rotation matrix constructed from a quaternion:

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

Other variations can be found in other sources. For example $m_{11} = -1 + 2w^2 + 2z^2$ also works, since $w^2 + x^2 + y^2 + z^2 = 1$.

Matrix to Quaternion 1

To extract a quaternion from a rotation matrix, reverse engineer the matrix from the last slide.

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

Matrix to Quaternion 2

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

Examining the sum of the diagonal elements (known as the *trace* of the matrix) we get:

$$\begin{aligned} \mathrm{tr}(\mathbf{M}) &= m_{11} + m_{22} + m_{33} \\ &= (1-2y^2-2z^2) + (1-2x^2-2z^2) + (1-2x^2-2y^2) \\ &= 3 - 4(x^2+y^2+z^2) \\ &= 3 - 4(1-w^2) \\ &= 4w^2 - 1, \end{aligned}$$

Matrix to Quaternion 3

Therefore,

$$w = \frac{\sqrt{m_{11}+m_{22}+m_{33}+1}}{2}.$$

and similarly,

$$\begin{aligned} x &= \frac{\sqrt{m_{11}-m_{22}-m_{33}+1}}{2} \\ y &= \frac{\sqrt{-m_{11}+m_{22}-m_{33}+1}}{2} \\ z &= \frac{\sqrt{-m_{11}-m_{22}+m_{33}+1}}{2} \end{aligned}$$

Matrix to Quaternion 4

- Unfortunately, we cannot use this trick for all four components, since the square root will always yield positive results. (More accurately, we have no basis for choosing the positive or negative root.)
- However, since **q** and **–q** represent the same orientation, we can arbitrarily choose to use the nonnegative root for one of the four components and still always return a correct quaternion.
- We just can't use the above technique for all four values of the quaternion.
- Another trick is to examine the sum and difference of diagonally opposite matrix elements.

Matrix to Quaternion 5

$$m_{12} + m_{21} = (2xy + 2wz) + (2xy - 2wz) = 4xy$$
$$m_{12} - m_{21} = (2xy + 2wz) - (2xy - 2wz) = 4wz$$
$$m_{31} + m_{13} = (2xz + 2wy) + (2xz - 2wy) = 4xz$$
$$m_{31} - m_{13} = (2xz + 2wy) - (2xz - 2wy) = 4wy$$
$$m_{23} + m_{32} = (2yz + 2wx) + (2yz - 2wx) = 4yz$$
$$m_{23} - m_{32} = (2yz + 2wx) - (2yz - 2wx) = 4wx$$

Matrix to Quaternion 6

Armed with these formulas, we use a two-step strategy.

1. Solve for one of the components of the trace.
2. Plug that value into one of the equations from the previous slide to solve for the other three.

This strategy boils down to selecting a row from the table on the next slide, then solving the equations in that row from left to right.

Matrix to Quaternion 7

$$w = \frac{\sqrt{m_{11}+m_{22}+m_{33}+1}}{2} \implies x = \frac{m_{23}-m_{32}}{4w} \quad y = \frac{m_{31}-m_{13}}{4w} \quad z = \frac{m_{12}-m_{21}}{4w}$$
$$x = \frac{\sqrt{m_{11}-m_{22}-m_{33}+1}}{2} \implies w = \frac{m_{23}-m_{32}}{4x} \quad y = \frac{m_{12}+m_{21}}{4x} \quad z = \frac{m_{31}+m_{13}}{4x}$$
$$y = \frac{\sqrt{-m_{11}+m_{22}-m_{33}+1}}{2} \implies w = \frac{m_{31}-m_{13}}{4y} \quad x = \frac{m_{12}+m_{21}}{4y} \quad z = \frac{m_{23}+m_{32}}{4y}$$
$$z = \frac{\sqrt{-m_{11}-m_{22}+m_{33}+1}}{2} \implies w = \frac{m_{12}-m_{21}}{4z} \quad x = \frac{m_{31}+m_{13}}{4z} \quad y = \frac{m_{23}+m_{32}}{4z}$$

Matrix to Quaternion 8

- The only questions is, which row should we use? Which component should we solve for first?
- The simplest strategy would be to just pick one arbitrarily and always use the same procedure, but this is fraught with problems.
  - Let's say we choose to always use the top row, meaning we solve for *w* from the trace, and then for *x*, *y*, and *z* with the equations on the right side of the arrow.
  - But if *w* = 0, the divisions to follow will be undefined.
  - Even if *w* > 0, a small *w* will produce numeric instability.
- Shoemake suggests the strategy of first determining which of *w*, *x*, *y*, and *z* has the largest absolute value, computing that component using the diagonal of the matrix, and then using it to compute the other three according to the table on the previous slide.


```
// Input matrix:
float m11,m12,m13;
float m21,m22,m23;
float m31,m32,m33;

// Output quaternion
float w,x,y,z;

// Determine which of w, x, y, or z has the largest absolute value
float fourWSquaredMinus1 = m11 + m22 + m33;
float fourXSquaredMinus1 = m11 - m22 - m33;
float fourYSquaredMinus1 = m22 - m11 - m33;
float fourZSquaredMinus1 = m33 - m11 - m22;

int biggestIndex = 0;
float fourBiggestSquaredMinus1 = fourWSquaredMinus1;
```



```
if (fourXSquaredMinus1 > fourBiggestSquaredMinus1) {
    fourBiggestSquaredMinus1 = fourXSquaredMinus1;
    biggestIndex = 1;
}
if (fourYSquaredMinus1 > fourBiggestSquaredMinus1) {
    fourBiggestSquaredMinus1 = fourYSquaredMinus1;
    biggestIndex = 2;
}
if (fourZSquaredMinus1 > fourBiggestSquaredMinus1) {
    fourBiggestSquaredMinus1 = fourZSquaredMinus1;
    biggestIndex = 3;
}

// Perform square root and division
float biggestVal = sqrt(fourBiggestSquaredMinus1 + 1.0f) * 0.5f;
float mult = 0.25f / biggestVal;
```

```
// Apply table to compute quaternion values
switch (biggestIndex) {
    case 0:
        w = biggestVal;
        x = (m23 - m32) * mult;
        y = (m31 - m13) * mult;
        z = (m12 - m21) * mult;
        break;

    case 1:
        x = biggestVal;
        w = (m23 - m32) * mult;
        y = (m12 + m21) * mult;
        z = (m31 + m13) * mult;
        break;

    case 2:
        y = biggestVal;
        w = (m31 - m13) * mult;
        x = (m12 + m21) * mult;
        z = (m23 + m32) * mult;
        break;

    case 3:
        z = biggestVal;
        w = (m12 - m21) * mult;
        x = (m31 + m13) * mult;
        y = (m23 + m32) * mult;
        break;
}
```


Euler Angles to Quaternion 1

- Similar to how we converted Euler angles to a rotation matrix.
- Convert heading, pitch, and bank to the corresponding quaternions, then use quaternion multiplication to get the composite transformation.
- Just as with matrices, there are two cases to consider: one when we wish to generate an object→upright quaternion, and a second when we want the upright→object quaternion.
- Since the two are conjugates, we will only walk through the derivation for the object→upright quaternion.

Euler Angles to Quaternion 2

- Suppose we start with Euler angles *h*, *p*, and *b* for heading, pitch, and bank.
- Let **h**, **p**, and **b** be quaternions which perform the rotations about the *y*, *x*, and *z*-axes, respectively.

$$\mathbf{h} = \begin{bmatrix} \cos(h/2) \\ \begin{pmatrix} 0 \\ \sin(h/2) \\ 0 \end{pmatrix} \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} \cos(p/2) \\ \begin{pmatrix} \sin(p/2) \\ 0 \\ 0 \end{pmatrix} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} \cos(b/2) \\ \begin{pmatrix} 0 \\ 0 \\ \sin(b/2) \end{pmatrix} \end{bmatrix}$$

Euler Angles to Quaternion 3

- Now to concatenate these in the correct order. We have two sources of backwardness that cancel each other out.
- We will use fixed-axis rotations, so the order of rotations will actually be bank, then pitch, then heading.
- However, quaternion multiplication performs the rotations from right-to-left.

Euler Angles to Quaternion 4

$$\mathbf{q}_{object \to upright}(h,p,b) = \mathbf{hpb}$$

$$= \begin{bmatrix} \cos(h/2) \\ \begin{pmatrix} 0 \\ \sin(h/2) \\ 0 \end{pmatrix} \end{bmatrix} \begin{bmatrix} \cos(p/2) \\ \begin{pmatrix} \sin(p/2) \\ 0 \\ 0 \end{pmatrix} \end{bmatrix} \begin{bmatrix} \cos(b/2) \\ \begin{pmatrix} 0 \\ 0 \\ \sin(b/2) \end{pmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \cos(h/2)\cos(p/2) \\ \begin{pmatrix} \cos(h/2)\sin(p/2) \\ \sin(h/2)\cos(p/2) \\ -\sin(h/2)\sin(p/2) \end{pmatrix} \end{bmatrix} \begin{bmatrix} \cos(b/2) \\ \begin{pmatrix} 0 \\ 0 \\ \sin(b/2) \end{pmatrix} \end{bmatrix}$$

Euler Angles to Quaternion 5

$$= \begin{bmatrix} \cos(h/2)\cos(p/2)\cos(b/2) + \sin(h/2)\sin(p/2)\sin(b/2) \\ \begin{pmatrix} \cos(h/2)\sin(p/2)\cos(b/2) + \sin(h/2)\cos(p/2)\sin(b/2) \\ \sin(h/2)\cos(p/2)\cos(b/2) - \cos(h/2)\sin(p/2)\sin(b/2) \\ \cos(h/2)\cos(p/2)\sin(b/2) - \sin(h/2)\sin(p/2)\cos(b/2) \end{pmatrix} \end{bmatrix}$$

The upright→object quaternion is simply the conjugate:

$$\mathbf{q}_{upright \to object}(h,p,b) = \mathbf{q}_{object \to upright}(h,p,b)^*$$

$$= \begin{bmatrix} \cos(h/2)\cos(p/2)\cos(b/2) + \sin(h/2)\sin(p/2)\sin(b/2) \\ \begin{pmatrix} -\cos(h/2)\sin(p/2)\cos(b/2) - \sin(h/2)\cos(p/2)\sin(b/2) \\ \cos(h/2)\sin(p/2)\sin(b/2) - \sin(h/2)\cos(p/2)\cos(b/2) \\ \sin(h/2)\sin(p/2)\cos(b/2) - \cos(h/2)\cos(p/2)\sin(b/2) \end{pmatrix} \end{bmatrix}$$

Quaternion to Euler Angles 1

Similar to how we converted a rotation matrix to Euler angles. Pitch:

$$\begin{aligned} p &= \arcsin(-m_{32}) \\ &= \arcsin\left(-2(yz - wx)\right) \end{aligned}$$

Quaternion to Euler Angles 2

Heading:

$$h = \begin{cases} \begin{aligned} &\operatorname{atan2}(m_{31}, m_{33}) \\ &\quad = \operatorname{atan2}(2xz + 2wy, 1 - 2x^2 - 2y^2) \\ &\quad = \operatorname{atan2}(xz + wy, 1/2 - x^2 - y^2) \end{aligned} & \text{if } \cos p \neq 0 \\ \begin{aligned} &\operatorname{atan2}(-m_{13}, m_{11}) \\ &\quad = \operatorname{atan2}(-2xz + 2wy, 1 - 2y^2 - 2z^2) \\ &\quad = \operatorname{atan2}(-xz + wy, 1/2 - y^2 - z^2) \end{aligned} & \text{otherwise} \end{cases}$$

Quaternion to Euler Angles 3

Bank:

$$b = \begin{cases} \begin{aligned} &\operatorname{atan2}(m_{12}, m_{22}) \\ &\quad = \operatorname{atan2}(2xy + 2wz, 1 - 2x^2 - 2z^2) \\ &\quad = \operatorname{atan2}(xy + wz, 1/2 - x^2 - z^2) \end{aligned} & \text{if } \cos p \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Quaternion to Euler Angles 4

Some code next.

1. Convert an object→upright quaternion into Euler angles.
2. Convert an upright→object quaternion into Euler angles.


```
// Input quaternion
float w,x,y,z;

// Output Euler angles (radians)
float h,p,b;

// Extract sin(pitch)
float sp = -2.0f * (y*z - w*x);

// Check for Gimbal lock, giving slight tolerance
// for numerical imprecision
if (fabs(sp) > 0.9999f) {

    // Looking straight up or down
    p = 1.570796f * sp; // pi/2

    // Compute heading, slam bank to zero
    h = atan2(-x*z + w*y, 0.5f - y*y - z*z);
    b = 0.0f;

} else {

    // Compute angles
    p = asin(sp);
    h = atan2(x*z + w*y, 0.5f - x*x - y*y);
    b = atan2(x*y + w*z, 0.5f - x*x - z*z);
}
```



```
// Extract sin(pitch)
float sp = -2.0f * (y*z + w*x);

// Check for Gimbal lock, giving slight tolerance
// for numerical imprecision
if (fabs(sp) > 0.9999f) {

    // Looking straight up or down
    p = 1.570796f * sp; // pi/2

    // Compute heading, slam bank to zero
    h = atan2(-x*z - w*y, 0.5f - y*y - z*z);
    b = 0.0f;

} else {

    // Compute angles
    p = asin(sp);
    h = atan2(x*z - w*y, 0.5f - x*x - y*y);
    b = atan2(x*y - w*z, 0.5f - x*x - z*z);
}
```

That concludes Chapter 8. Next, Chapter 9:

Geometric Primitives