Understanding Rotations

Jim Van Verth
Senior Engine Programmer, Insomniac Games
jim@essentialmath.com

Introductions. Name a little misleading, as truly understanding rotations would require a deep understanding of group theory, which I honestly neither have, nor have time to present. So a better name might be

Understanding Rotation Formats

Jim Van Verth
Senior Engine Programmer, Insomniac Games
jim@essentialmath.com

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012

Which isn't say I won't be covering other aspects of rotation, it's just that that will be the primary focus of this talk.

Intro

- Discuss various rotation reps
  - Angle (2D), Euler angles/Axis-angle (3D)
  - Matrix (2D & 3D)
  - Complex numbers (2D), Quaternion (3D)

The order here is an attempt to compare similar formats across 2D and 3D.

Intro

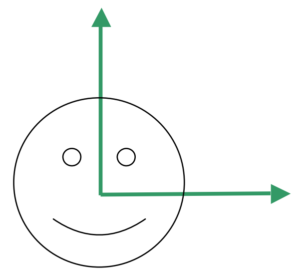
- Issues to consider
  - # elements
  - Concatenation
  - Interpolation
  - Rotation

Intro

- Interpolating not as simple as position, but more important
- E.g. camera control
  - Store orientations for camera, interpolate
- E.g. character animation
  - Body location stored as point
  - Joints stored as rotations

Intro

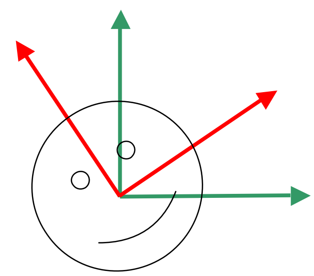
- Orientation relative to reference frame



On the previous slide, I mentioned orientation and rotation. Throughout the talk I may use them interchangeably, and I want to make sure that the distinction between them is clear. Orientation refers to where the axes of the reference frame (or coordinate system) lie.

Intro

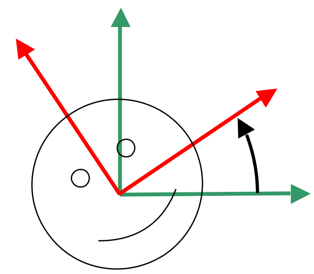
- Orientation relative to reference frame



Those axes are relative to a fixed reference frame, marked in green in this diagram.

Intro

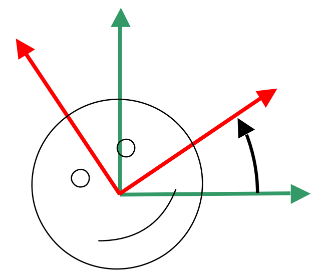
- Orientation relative to reference frame
- Rotation changes object from one orientation to another



Rotation is the operation that takes us from one orientation to another one, represented here by the black arrow.

Intro

- Orientation relative to reference frame
- Rotation changes object from one orientation to another
- Hence, represent orientation as a rotation



So, it is possible to represent orientation as a rotation from the reference frame, which is what we usually do.

Ideal Rotation Format

- Represent degrees of freedom with minimum number of values
- Allow concatenations of rotations
- Math should be simple and efficient
  - concatenation
  - interpolation
  - rotation

So what do we look for in an ideal rotation format?

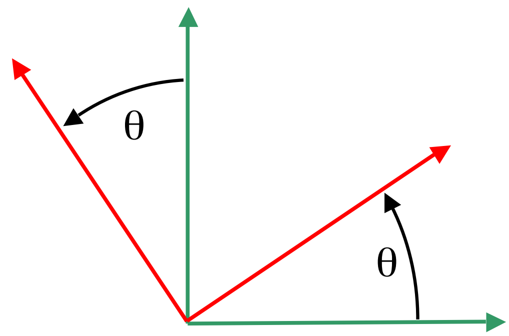
Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

Topics

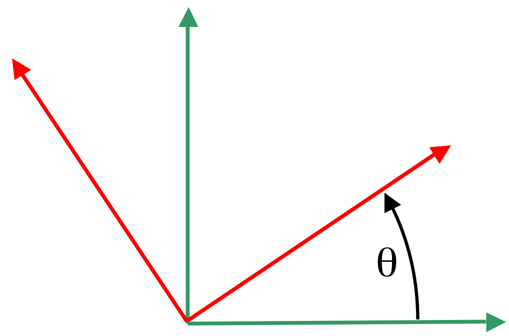
- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

2D Angle



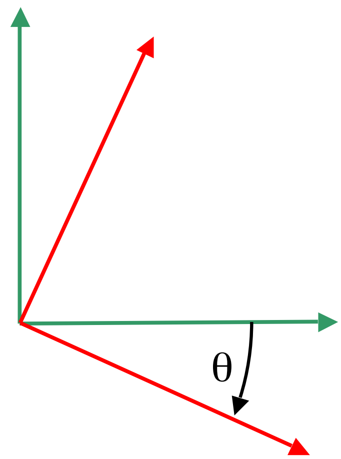
The simplest rotation format is just the angle between the original coordinate axes and the new ones. It's the same for x and y axes, so…

2D Angle



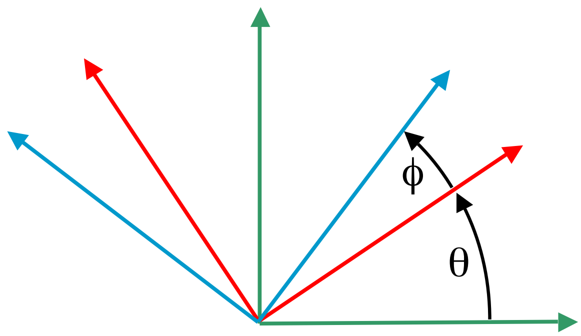
To simplify things I'll just use the angle between the old and new x axes.

2D Angle



This angle can also be negative, btw.

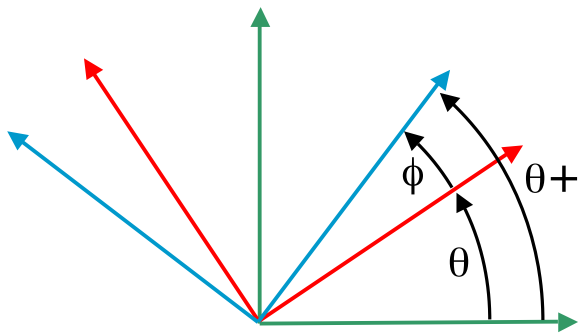
2D Angle: Concatenation



Concatenation is very simple. If we rotate by an angle theta and then an angle phi…

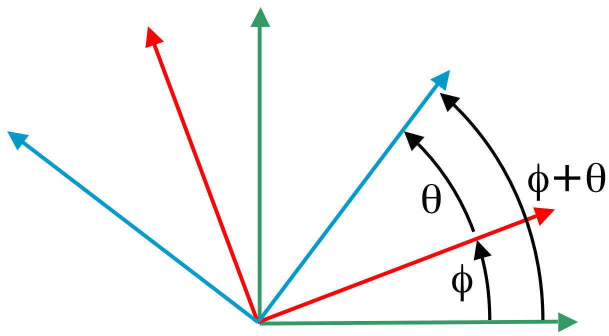
GAME DEVELOPERS CONFERENCE® 2012 MARCH 5-9, 2012 WWW.GDCONF.COM

2D Angle: Concatenation



We can represent this by a single angle theta plus phi

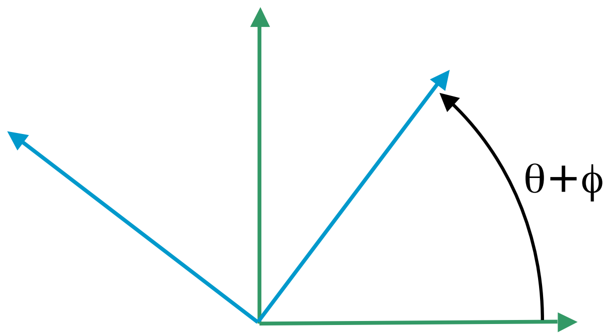
2D Angle: Concatenation



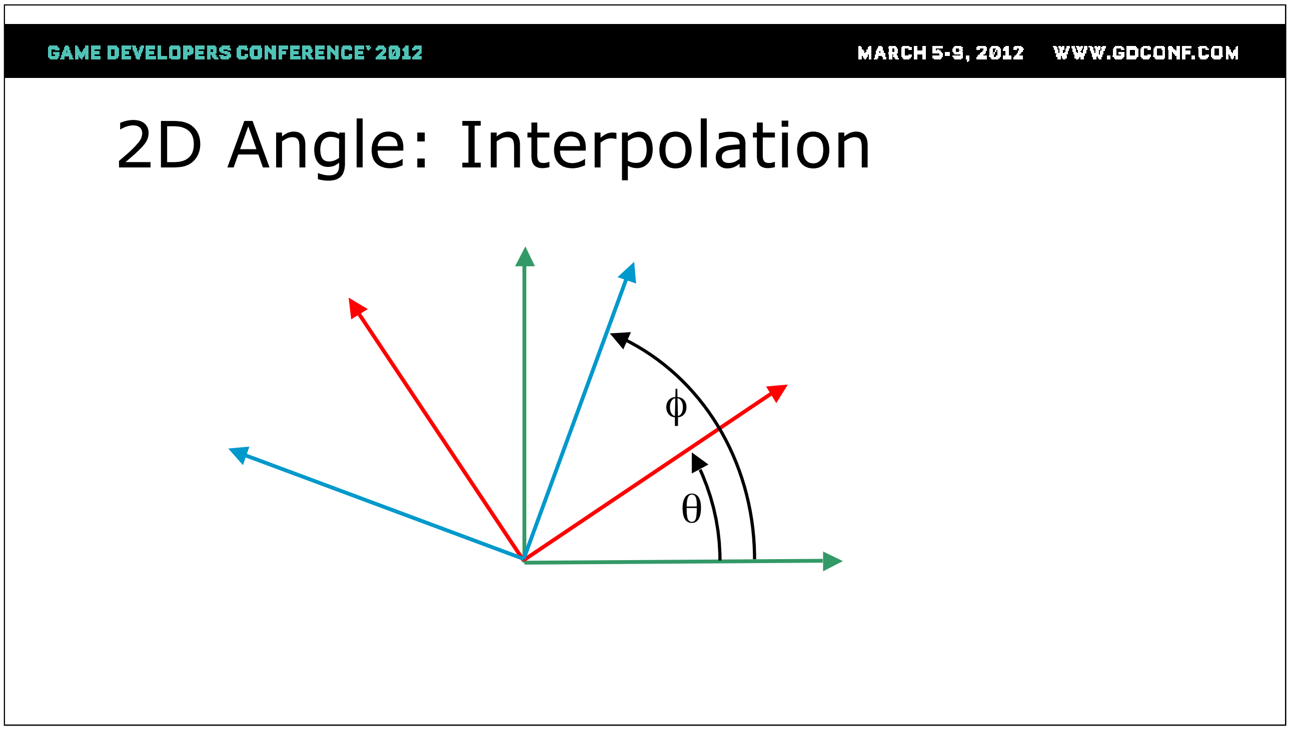
Note: 2D rotation is commutative

Note that because addition is commutative, this concatenation is commutative, so rotating by phi first and then theta we get the same result.

2D Angle: Concatenation

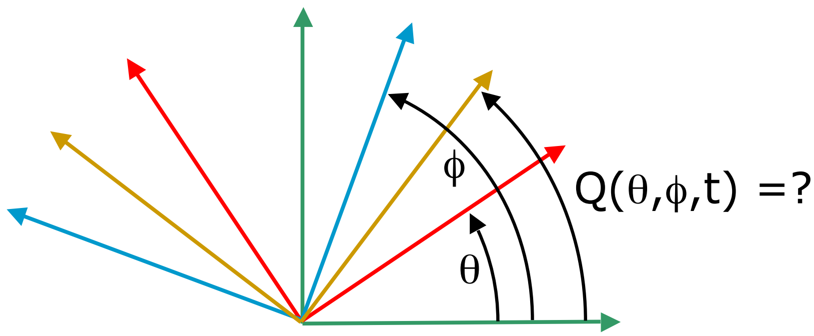


And so we have the final rotation.



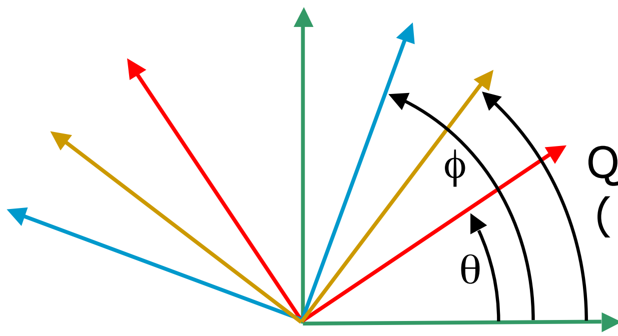
Blending between angles is just about as simple, but there are some gotchas to be aware of. So suppose we have a rotation theta and a rotation phi (a different phi than the previous one, in this case)…

2D Angle: Interpolation



And we want to find a rotation between them, using an interpolation factor t that varies from 0 to 1.

2D Angle: Interpolation



$Q(\theta,\phi,t) =$
$(1-t)\theta+t\phi$

This is pretty simple, we can just do a linear interpolation between theta and phi. This formula should seem familiar after Squirrel's talk.

2D Angle: Interpolation

What if $\theta = 30°$ & $\phi = 390°$?

- Expect always same angle
- But $(1-t)\theta + t\phi$ will vary from 30° to 390°

However, as mentioned, there are gotchas. Suppose we have angles of 30 degrees and 390 degrees. These are the same rotation, but if we do a straight linear interpolation, we'll end up with angles between 30 and 390, when we'd expect to not rotate at all.

2D Angle: Interpolation

- Problem One: angles not well-formed
- Infinite # of values can represent one rotation: 30º = 390º = -330º
- Can constrain to [0,360) or [0, 2π)

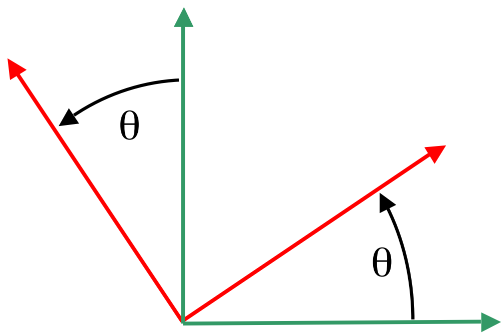
So that's one problem with angles: you can have an infinite number of values that represent one rotation. The simplest solution here is to just constrain the angles to a range, 0 to 360 or 0 to 2 pi if you're using radians.

2D Angle: Rotation

- Idea: vector/point coordinates relative to coordinate frame
- Change in frame gives change of coordinates

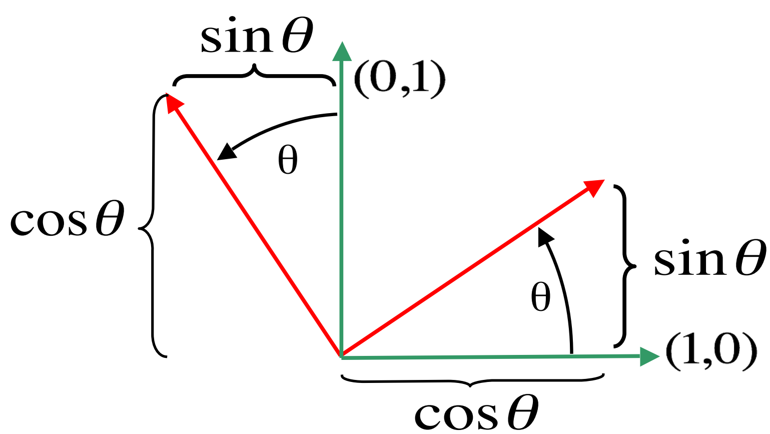
How about rotation. Here things get a little more complicated, but not too bad. As the slide says, the coordinates that we use for both vectors and points are relative to the coordinate frame we're using. So if we track how the frame changes, we can compute the new coordinates. It's all part of the magic of vector spaces… or affine spaces in this case.

2D Angle: Rotation

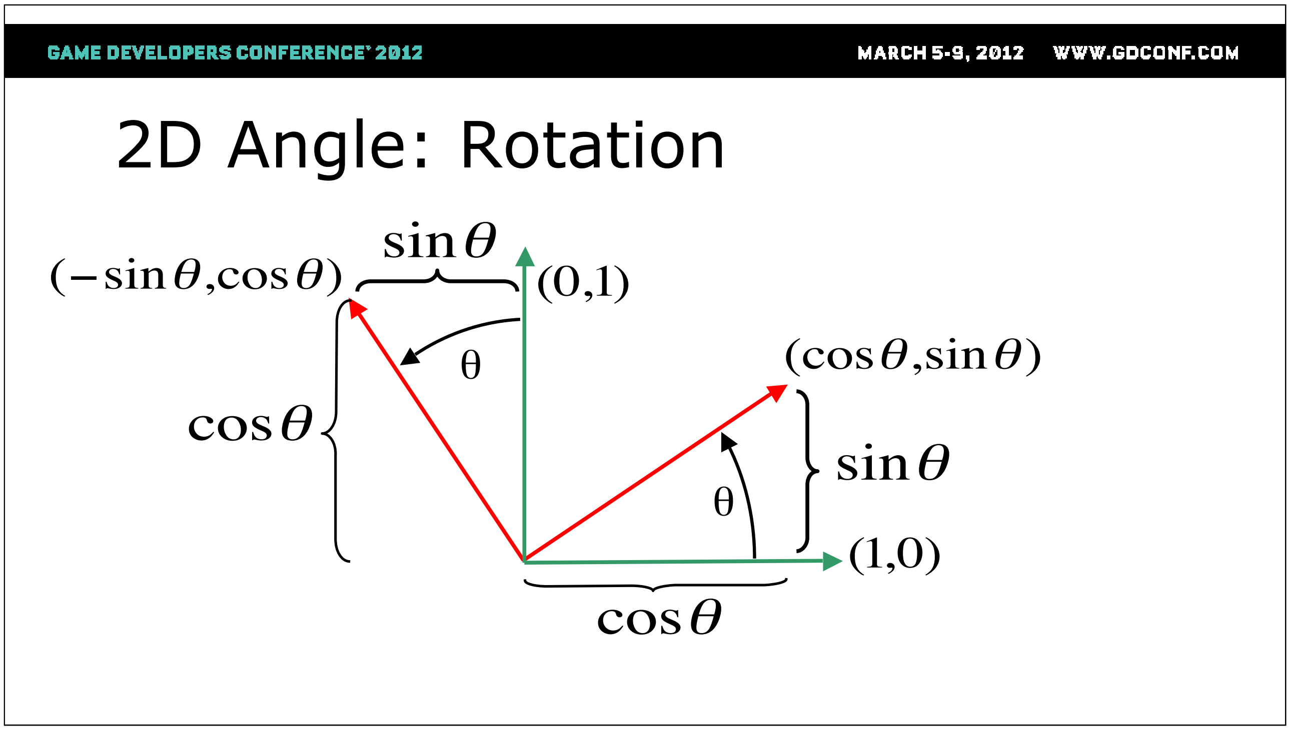


So, returning to our original diagram, with both angles in this case.

2D Angle: Rotation

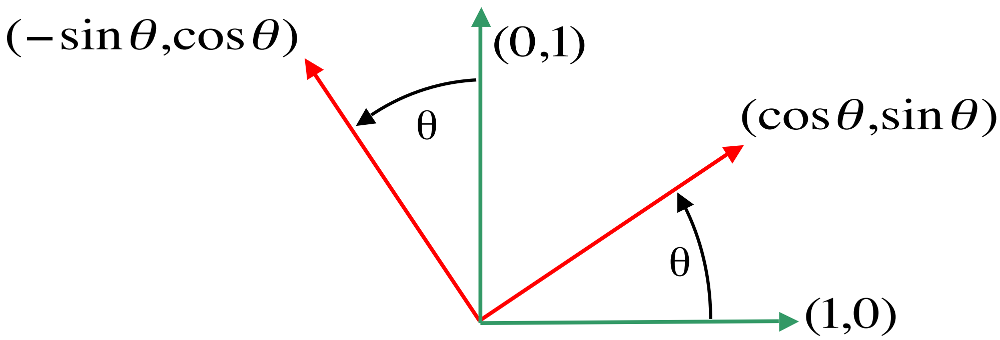


The original axes have coordinates (1,0) for the x axes and (0,1) for the y axes. Their length is one, so by trigonometry, we can easily compute the coordinates of the new axes relative to the new ones, namely (show) cos theta here and sin theta here. And the same for the y axes.



So our new coordinates are cos theta, sin theta for the x axis and -sin theta cos theta for the y-axis.

2D Angle: Rotation



Simplifying, just to make it a little more clear.

2D Angle: Rotation

- Point in old frame

$$(x,y) = x(1,0) + y(0,1)$$

- Point in new frame

$$\begin{aligned} R(x,y,\theta) &= x(\cos\theta,\sin\theta) + y(-\sin\theta,\cos\theta) \\ &= (x\cos\theta, x\sin\theta) + (-y\sin\theta, y\cos\theta) \\ &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta) \end{aligned}$$

So as I mentioned, the coordinates that we use are relative to our current frame. So for a point x, y, this just means that we take x and multiply it by (1,0) and take y and multiply it by (0,1). That gives us x,y as we expect. For the new frame, we just take our original x, y and multiply by the new axes. So that’s x times cos theta, sin theta, and y times -sin theta cos theta, which simplifies to this final result for our rotation equation.

2D Angle: Rotation

- So rotation of vector (x,y)

$$R(x,y,\theta) = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$$

- Problem two: have to calc sin and cos to rotate

So we derived our rotation formula, but as we can see, in order to compute this we'll have to compute a sin and cos, which is not always the fastest operation.

2D Angle: Summary

- Compact (1 value)
- Concat easy (add)
- Interpolation doable
- Rotation not ideal
- Be careful of infinite values

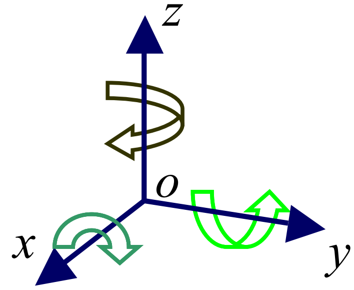
Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

So that's angles in 2D. Now we're going to look at formats that use angles for 3D rotation. We'll begin with Euler angles.

Euler Angles

- Three ordered rotations around orthogonal axes



- Could be world or local axes
- Order important! 3D non-commutative

Demo

Euler angles are just like single 2D angle, except that instead of rotating around a single (implied) axis, we're rotating around 3 different axes. This follows from Euler's theorem that all 3D rotations can be represented by three ordered rotations, hence the name.

Euler Angles vs. Fixed Angles

- Euler angle - rotates around local axes
- Fixed angle - rotates around world axes
- Rotations are reversed
  - x-y-z Euler angles == z-y-x fixed angles

Often there are differences in terminology for these -- some people like to refer to Euler angles as those rotate only around the local axes of the object, while they refer to rotations around the world axes as fixed angles. They behave similarly -- to get from one to the other you just reverse the rotation order. But often times you'll just see both kinds referred to as Euler angles, so just be aware of which axes you're rotating around.

Euler Angle Issues

- No easy concatenation of rotations
- Still has interpolation problems
- Can lead to gimbal lock

Euler angles, despite being compact, have some serious problems that make it undesirable as a general rotation format. First, our easy addition of angles goes out the window with Euler angles. Secondly, our interpolation problems are even worse. Finally, when axes align after a series of rotations, we can end up with something called gimbal lock, where we lose one degree of freedom. Let’s look at these problems in turn.

Euler Angle Concatenation

- Can't just add or multiply components
- Best way:
  - Convert to matrices
  - Multiply matrices
  - Extract euler angles from resulting matrix
- Not cheap

Euler Angle Interpolation

- Example:
  - Halfway between (0, 90, 0) & (90, 45, 90)
  - Lerp directly, get (45, 67.5, 45)
  - Desired result is (90, 22.5, 90)
- Can use Hermite curves to interpolate
  - Assumes you have correct tangents
- AFAIK, slerp not even possible

Gimbal Lock

- Euler/fixed angles even less well-formed
- Different values can give same rotation
- Example with z-y-x fixed angles:
  - (90, 90, 90) = (0, 90, 0)
- Why? Rotation of 90° around y aligns x and z axes
- Rotation around z cancels x rotation

Demo

Euler Angles

- Good for interface
- Not so good for in-engine

So in summary: Euler angles -- avoid them!

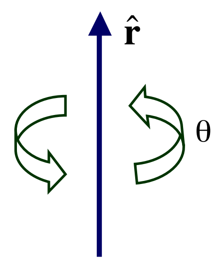
Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

So let’s look at another 3D angle format and see if that works better for us: axis-angle.

Axis and Angle

- Specify vector, rotate ccw around it
- Can interpolate, messy to concatenate



Euler also proved that any 3D rotation can be represented as a rotation around an arbitrary axis. So axis-angle is just as it sounds -- we specify an axis and how much we're going to rotate around it, in a counterclockwise direction (right-hand rule). I'm not going to spend a lot of time on axis-angle as it has it's own brand of problems. Interpolation is pretty simple -- you can just blend the axis and angle separately and get a reasonable result. However, concatenation is much the same as Euler angles -- you have to convert to a matrix (or another format, which we'll get to) -- concatenate, then convert back. In my opinion, it's just not worth it.

Axis and Angle

- Rotation

$$\mathrm{R}(\mathbf{p},\hat{\mathbf{r}},\theta) = \cos\theta \cdot \mathbf{p} + (1-\cos\theta)(\mathbf{p} \bullet \hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}} \times \mathbf{p})$$

However, it is convenient at times to be able to rotate something by an axis-angle representation, so here's the formula for that. As you can see, this is not the simplest operation either.

Axis and Angle

- More of a transitional format
- I.e. convert to axis-angle, manipulate angle or axis, convert back

We’ll see an example of this with 3D matrices in a bit.

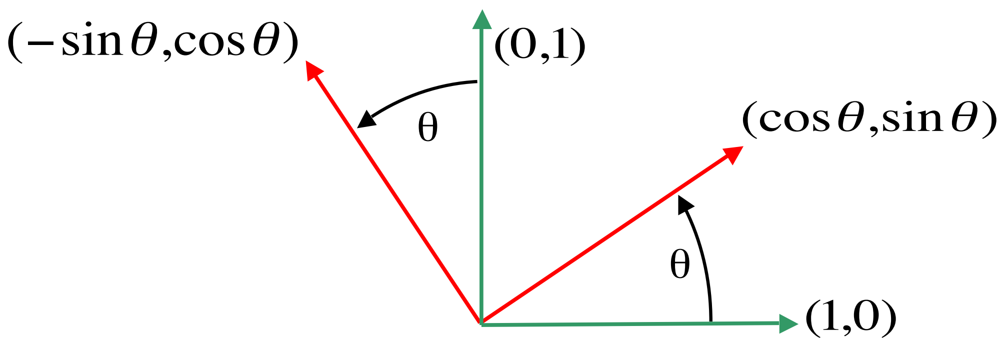
Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

Ok, now we're going to bounce back to 2D and consider a much nicer and (hopefully) familiar format, the matrix.

2D Matrix

- Recall



So going back to our original axis diagram, recall that our original axes change coordinates to these value.

2D Matrix

- Idea: Bake new frame in matrix and multiply by vector to rotate
- Matrix represents transformation

The idea of a matrix is simple: we bake this new frame in the matrix, and then matrix multiplication will do the coordinate transformation for us. By the way, if you understand this -- and I am going to go a bit fast on this, so I apologize -- but if you understand it, you can handle any transformation you need to compute. If you want one area of linear algebra to study that will help you be successful in computer graphics or even physics, this is it.

2D: Matrix

- Change in frame

$$(1,0) \Rightarrow (\cos\theta, \sin\theta)$$
$$(0,1) \Rightarrow (-\sin\theta, \cos\theta)$$

- Rotation matrix

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

(assumes row vectors)

So in the standard case, where we're working with Euclidean frames, we don't need to do anything special, just drop the new frame in. Assuming that we're using row vectors, that is, our multiplication order is from left to right, then we're going to insert our new frame in as the rows of the rotation matrix.

2D: Matrix

- Change in frame

$$(1,0) \Rightarrow (\boxed{\cos\theta, \sin\theta})$$
$$(0,1) \Rightarrow (-\sin\theta, \cos\theta)$$

- Rotation matrix

$$\begin{bmatrix} \boxed{\cos\theta \quad \sin\theta} \\ -\sin\theta \quad \cos\theta \end{bmatrix}$$

(assumes row vectors)

Just to make it more clear, our first row is the same as our new x-axis…

2D: Matrix

- Change in frame

$$(1,0) \Rightarrow (\boxed{\cos\theta, \sin\theta})$$

$$(0,1) \Rightarrow (\boxed{-\sin\theta, \cos\theta})$$

- Rotation matrix

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

(assumes row vectors)

And the second row is the same as the new y-axis.

2D Matrix: Rotation

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

And multiplying it out, we get the same result as before from our angle formula.

2D Matrix: Concatenation

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} \cos(\theta+\phi) & \sin(\theta+\phi) \\ -\sin(\theta+\phi) & \cos(\theta+\phi) \end{bmatrix}$$

Concatenation also uses multiplication, but this time we're multiplying two rotation matrices together. After multiplying and using some trigonometric identities to simply, we can see that we get the result we expect: the angle in the new matrix is just the sum of the original two angles. Note again that the multiplication order doesn't matter here because we're doing 2D rotation. That won't be the case when we get to 3D.

2D Matrix: Interpolation

- Lerp values:

$$0.5\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}+0.5\begin{pmatrix}0 & -1\\1 & 0\end{pmatrix}=\begin{pmatrix}0.5 & -0.5\\0.5 & 0.5\end{pmatrix}$$

- Result isn’t a rotation matrix!
- Need Gram-Schmidt orthonormalization

So rotating a vector and concatenating rotations are fairly nice. What about interpolation. Well, here things start to fall apart. If we take these two rotation matrices: one with no rotation and the other a rotation of negative 90 degrees, and try to do a linear interpolation between them, we get a bad result. The resulting row vectors are not unit length, so this is not a rotation matrix. Now, we can do Gram-Schmidt orthonormalization to solve this problem, but it doesn’t solve all of our problems.

2D: Matrix

- Lerp values:

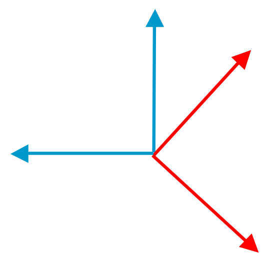
$$0.5\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} + 0.5\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

- Not even a valid affine transformation…

For example, interpolating from a rotation of negative 90 degrees to a rotation of positive 90 degrees, gives us an extremely bad matrix. So what can we do about this?

2D Matrix: Interpolation

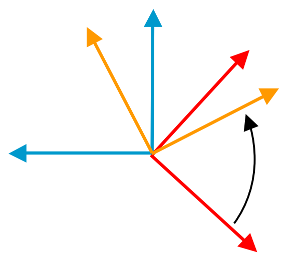
- Example



Let's take a look at what's going on here, by examining where the axis vectors go. So here are the frames for two possible rotations, the red being about a rotation of -45 degrees, the blue being a rotation of about positive 90 degrees.

2D Matrix: Interpolation

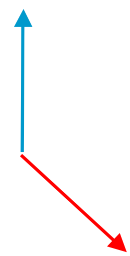
- Example



And suppose we want to interpolate between them.

2D Matrix: Interpolation

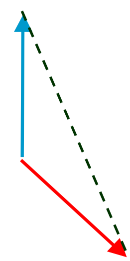
- Look at just x-axis



To simplify things, let's just look at the x-axis.

2D Matrix: Interpolation

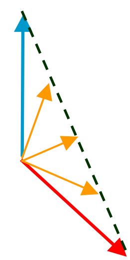
- Lerp



If we linearly interpolate between the two x-axes, that's basically just drawing a line from vector tip to vector tip…

2D Matrix: Interpolation

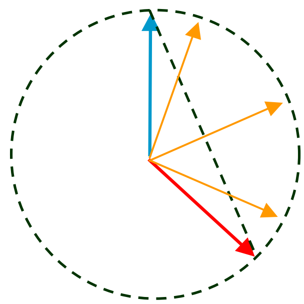
- Lerp



And picking points along the line. Here I've spaced them out at t values of 1/4, 1/2, and 3/4. Note that they are clearly shorter than our original vectors, so they're no longer unit length. Now, we could do our orthonormalization process, which would make these vectors unit length again.

2D Matrix: Interpolation

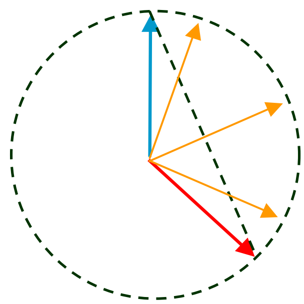
- Lerp, extended to unit length



And here we see the result of that. However, now we have another problem.

2D Matrix: Interpolation

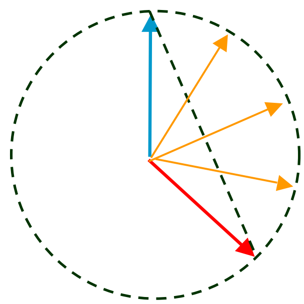
- But equal time != equal spacing



Note that along the line, the vectors are equally spaced, but along the rotation arc they're not. What we'd really like is that as we move in time, using our interpolant t, that our rotation would move equally as well.

2D Matrix: Interpolation

- Subdivide arc, not line



- Spherical linear interpolation, or slerp

So here's a diagram showing that -- note that now the arc of rotation is now subdivided equally. This is called spherical linear interpolation, or (as Ken Shoemake says, because it's fun): slerp.

2D Matrix: Interpolation

- Idea: compare position operations to orientation

- Be careful of order! (important for 3D)

$$x + y \Rightarrow xy$$

$$x - y \Rightarrow xy^{-1}$$

$$ax \Rightarrow x^{a}$$

So how can we compute slerp? One way to think about this -- and for any mathematicians in the audience this is admittedly not a formal proof, but perfectly appropriate -- we can take the operations we use for linear interpolation and take them up one level to get the appropriate operations for rotation matrices. Then we can use this to convert our linear interpolation formula to a spherical linear interpolation formula. So where we would add two angles, we multiply two matrices. Where we would subtract one angle from another, we multiply by the matrix inverse. And where we would scale an angle, we instead take the rotation matrix to the same power.

2D Matrix: Interpolation

- Apply to lerp

$$(\mathbf{x}_1 - \mathbf{x}_0)t + \mathbf{x}_0$$

- Gives slerp formula

$$(\mathbf{M}_1\mathbf{M}_0^{-1})^t\mathbf{M}_0$$

$$\boxed{\begin{aligned} x + y &\Rightarrow xy \\ x - y &\Rightarrow xy^{-1} \\ ax &\Rightarrow x^a \end{aligned}}$$

Apply this to our lerp formula, we get the following slerp formula. And as I mentioned on the previous slide, this order is important -- while any order is reasonable for 2D rotations because (all together now) they’re commutative, this is not the same for 3D rotations. However, both of these slides do bring up a question.

2D Matrix: Interpolation

- But what is $\mathbf{M}^t$?
- General: Taylor series
- 2D rotation simpler:

$$(\mathbf{M}_\theta)^t = \mathbf{M}_{t\theta}$$

$$\boxed{(\mathbf{M}_1\mathbf{M}_0^{-1})^t\mathbf{M}_0}$$

What is M to the t? For general matrices, this is just a function, and you can compute an approximation by using a Taylor series expansion (Gino will say more about Taylor series in the next talk). However, in our case we’re only considering rotation matrices, so the answer is much simpler. All you need to do is pull the angle out of the matrix, multiply it by t, and generate a new matrix for that angle.

2D Matrix: Interpolation

- Process:
  - Compute

    $$\mathbf{M} = \mathbf{M}_1\mathbf{M}_0^{-1}$$

  - Then

    $$\theta = \operatorname{atan2}(\mathbf{M}_{0,1}, \mathbf{M}_{0,0})$$

  - Finally

    $$\mathbf{M}_{t\theta}\mathbf{M}_0$$

$$\boxed{(\mathbf{M}_1\mathbf{M}_0^{-1})^t\,\mathbf{M}_0}$$

So the process is just this. Note that M0,1 is sin theta, and m0,0 is cos theta, so we can take the arc tangent to get the correct angle.

2D Matrix: Interpolation

- An alternative (only for 2D):
  - Lerp the first row
  - Renormalize
  - Rotate 90 degrees to get the second row
  - Build new matrix
  - But need to correct for time (discuss later)

2D Matrix: Interpolation

- Blend multiple matrices, e.g. skinning
  - Maya gives you weights, just lerp
  - Can use De Castlejau’s Algorithm w/slerp
  - Alternative: dual quaternions

2D Matrix: Recap

- Rotation: fast
- Concatenation: fast
- Lerp/slerp: unwieldy
- Also: 4 values to represent 1 d.o.f.

Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

3D: Matrix

- Much the same as 2D matrix
  - Map transformed axes, store as rows of matrix
  - Rotate via vector-matrix mult
  - Concatenate via matrix-matrix multiplication (but no longer commutative)

3D Matrix: Interpolation

- Lerp same problems as before
  - 9 values to interpolate
  - don't interpolate well
- Slerp even harder to compute

$$(\mathbf{M}_{(\hat{\mathbf{r}},\theta)})^t = \mathbf{M}_{(\hat{\mathbf{r}},t\theta)}$$

3D Matrix: Summary

- Workhorse of 3D graphics
- Great for rotation and concatentation (especially w/vector processors)
- Inconvenient for interpolation

Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

2D: Complex Numbers

- Review:

$$a + b\mathbf{i}$$

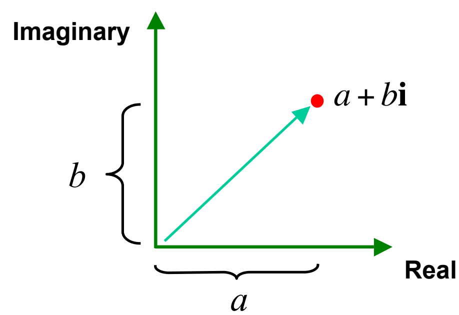
where

$$\mathbf{i} = \sqrt{-1}$$

- But ignore that “imaginary” crap

2D: Complex Numbers

- First important bit



2D: Complex Numbers

- Second important bit

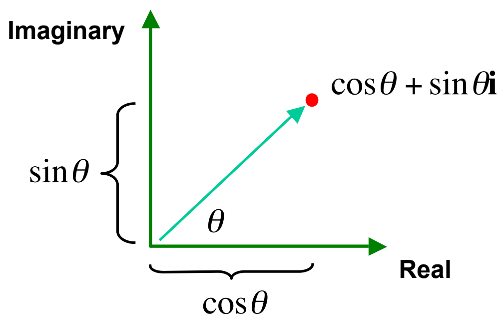
$$(a + b\mathbf{i})(c + d\mathbf{i}) = (ac - bd) + (bc + ad)\mathbf{i}$$

- Another way to think of it

$$(a, b) \cdot (c, d) = (ac - bd, bc + ad)$$

2D: Complex Numbers

- Suppose: restrict to unit length



- Also written as $\cos\theta + \mathbf{i}\sin\theta = e^{\mathrm{i}\theta}$

2D: Complex Numbers

- Multiply general complex number by unit one

$$(x + y\mathbf{i})(\cos\theta + \sin\theta\mathbf{i}) =$$
$$(x\cos\theta - y\sin\theta) + (x\sin\theta + y\cos\theta)\mathbf{i}$$

- Look familiar?
- Gives us rotation

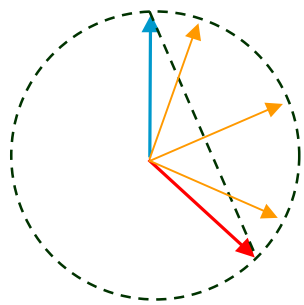
2D: Complex Numbers

- Concatenation

$$(\cos\theta + \sin\theta\mathbf{i})(\cos\phi + \sin\phi\mathbf{i}) =$$
$$(\cos(\theta + \phi)) + (\sin(\theta + \phi))\mathbf{i}$$

2D: Complex Interpolation

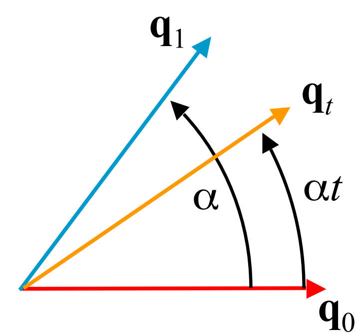
- Lerp similar, but can normalize (nlerp)



Lerping our complex numbers is much like matrixes, except in this case each arrow represents an entire complex number instead of just the x-axis of a matrix. So rather than doing the full orthonormalization process we can just perform our linear interpolation and then just do one normalization operation. This is often called nlerp. That said, the same problems still remain with non-equal subdivision of our rotation arc, so let's look at slerp again.

2D: Complex Interpolation

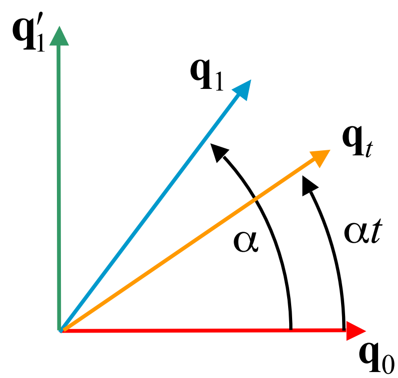
- Slerp
  - Want to find $\mathbf{q}_t$



In generating a formula for slerp with complex numbers we can take a different approach than with matrices. Suppose we have two complex numbers q0 and q1 and we want to blend between them. The angle between them is alpha, and we want to find the complex number that's alpha t between the two.

2D: Complex Interpolation

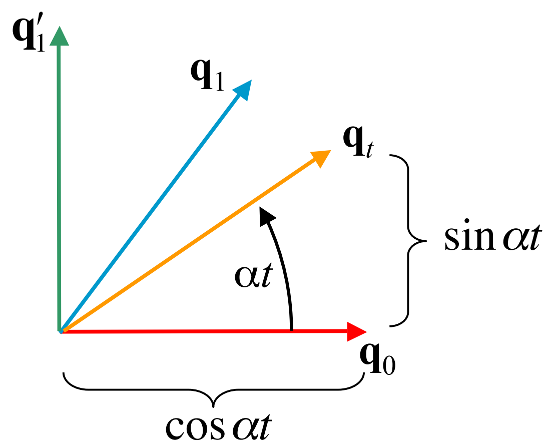
- Slerp
  - Create basis



Suppose we can find a perpendicular to q0 based on q1 -- we'll just call that q1′. That gives us a coordinate frame..

2D: Complex Interpolation

- Slerp
  - Generate coords

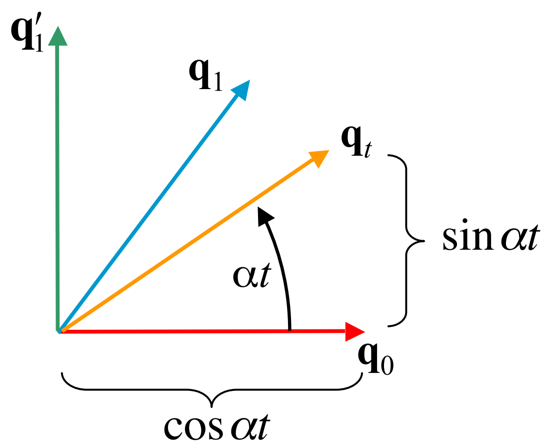


And we can use this frame to generate coordinates for our new q_t. As before, the distance along the q0 axis is just cos alpha t, and the distance along the q1' axis is sin alpha t.

Complex Number Interpolation

- Slerp
  - Then

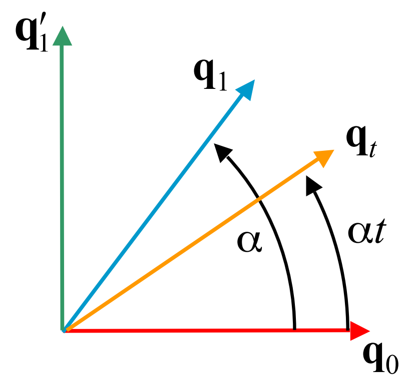
$$\mathbf{q}_t = \cos\alpha t\,\mathbf{q}_0 + \sin\alpha t\,\mathbf{q}_1'$$



So for an arbitrary q0 and q1′, our slerped complex number is this.

2D: Complex Interpolation

- Finding $\mathbf{q}_1'$
  - In 2D can do $\mathbf{q}_{0\perp}$



That leaves one open question: how to we compute this q1'? Well, in 2D we can just rotate q0 90 degrees to get the perpendicular.

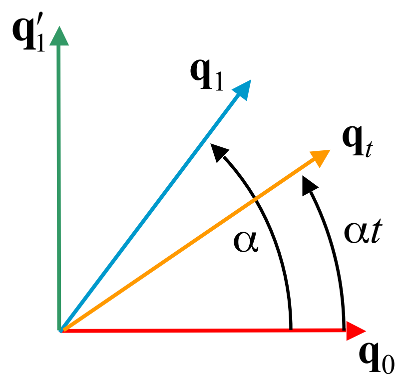
2D: Complex Interpolation

- Finding $\mathbf{q}_1'$
  - In general,

$$\mathbf{q}_1' = \frac{\mathbf{q}_1 - (\mathbf{q}_0 \bullet \mathbf{q}_1)\mathbf{q}_0}{\left\| \mathbf{q}_1 - (\mathbf{q}_0 \bullet \mathbf{q}_1)\mathbf{q}_0 \right\|}$$

  - Simplifies to

$$\mathbf{q}_1' = \frac{\mathbf{q}_1 - \cos\alpha\,\mathbf{q}_0}{\sin^2\alpha}$$



But let’s consider the general case -- this will be useful when we get to quaternions. We can compute this by projecting q1 onto q0, subtracting the result from q1, and then normalizing. This is just one step in Gramm-Schmidt orthonormalization. For the case of our unit complex numbers (or any unit vector, for that matter), this just simplifies to this.

2D: Complex Interpolation

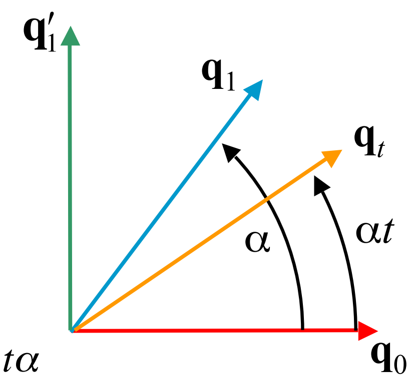
- Slerp
  - Combine

$$\mathbf{q}_t = \cos\alpha t\,\mathbf{q}_0 + \sin\alpha t\,\mathbf{q}_1'$$

$$\mathbf{q}_1' = \frac{\mathbf{q}_1 - \cos\alpha\,\mathbf{q}_0}{\sin^2\alpha}$$

  - Get

$$\mathbf{q}_t = \frac{\sin(1-t)\alpha}{\sin\alpha}\mathbf{q}_0 + \frac{\sin t\alpha}{\sin\alpha}\mathbf{q}_1$$



Combining our two formulas together we get the following,

2D: Complex Interpolation

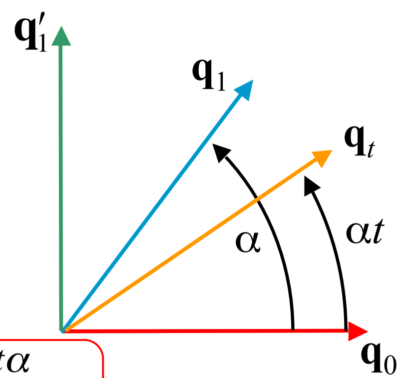
- Slerp
  - Combine

$$\mathbf{q}_t = \cos\alpha t\,\mathbf{q}_0 + \sin\alpha t\,\mathbf{q}_1'$$

$$\mathbf{q}_1' = \frac{\mathbf{q}_1 - \cos\alpha\,\mathbf{q}_0}{\sin^2\alpha}$$

  - Get

$$\boxed{\mathbf{q}_t = \frac{\sin(1-t)\alpha}{\sin\alpha}\mathbf{q}_0 + \frac{\sin t\alpha}{\sin\alpha}\mathbf{q}_1}$$



which is our final slerp formula.

2D: Complex Interpolation

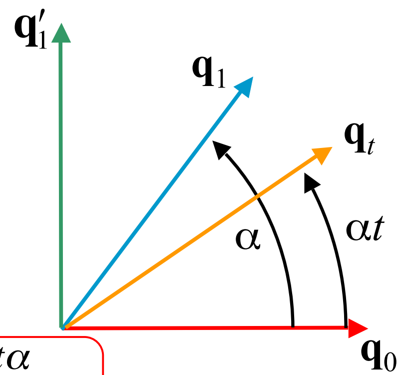
- Slerp
  - Combine

$$\mathbf{q}_t = \cos\alpha t\,\mathbf{q}_0 + \sin\alpha t\,\mathbf{q}_1'$$

$$\mathbf{q}_1' = \frac{\mathbf{q}_1 - \cos\alpha\,\mathbf{q}_0}{\sin^2\alpha}$$

  - Get

$$\boxed{\mathbf{q}_t = \frac{\sin(1-t)\alpha}{\sin\alpha}\mathbf{q}_0 + \frac{\sin t\alpha}{\sin\alpha}\mathbf{q}_1}$$



Same as: $\mathbf{q}_t = \mathbf{q}_0(\mathbf{q}_0^{-1}\mathbf{q}_1)^t$

Btw, it can be shown that this gives the same result as our other slerp formula. However, this one is more practical to compute.

2D Complex Interpolation

- Slerp not ideal

$$\mathbf{q}_t = \frac{\sin(1-t)\alpha}{\sin\alpha}\mathbf{q}_0 + \frac{\sin t\alpha}{\sin\alpha}\mathbf{q}_1$$

  - Computing α, sin α, sin α t slow
  - Numeric error as $\alpha \to 0$

Also, depending on how we calculate alpha, this can be non-commutative as well, I.e. slerping from q0 to q1 is not the same as slerping from q1 to q0 -- you end up going different ways around the circle. That said, most implementations assume that alpha is greater than 0, which will make it commutative.

Faster Slerp

- Lerp is pretty close to slerp
- Just varies in speed at middle
- Idea: can correct using simple spline to modify t (adjust speed)
- From Jon Blow's column, *Game Developer*, March 2002
- Lerp speed w/slerp precision

Demo

Faster Slerp

- In practice, we have small angles
- nlerp alone may well be good enough

Complex Numbers

- Note: complex multiplication is commutative, as is 2D rotation

Complex Numbers

- Half-angle form

$$\mathbf{q} = (\cos(\theta/2) + \sin(\theta/2)\mathbf{i})$$

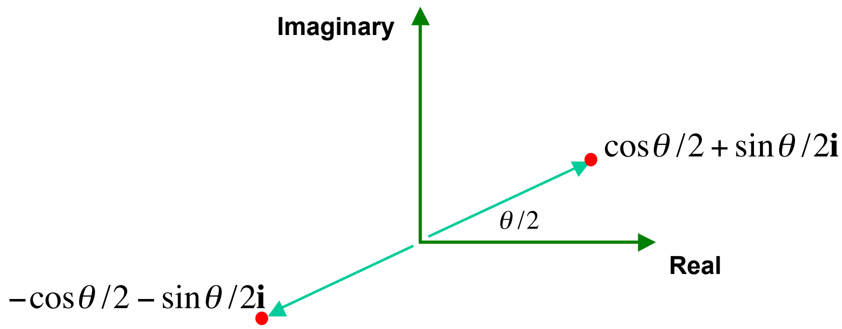
- Then rotation could be

$$\mathrm{Rot}(\mathbf{p},\theta) = \mathbf{qpq}$$

- Still unit length

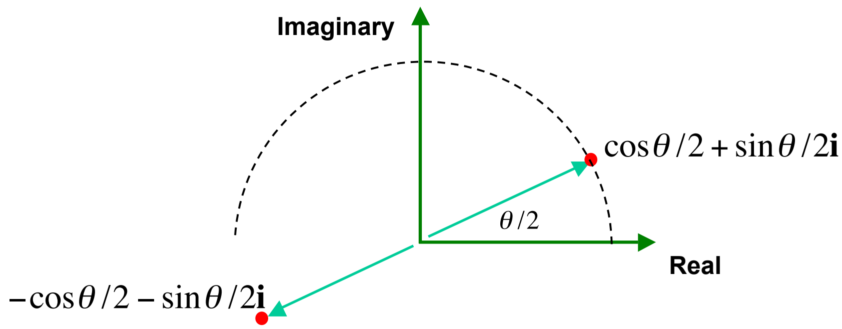
Complex Numbers: Half Angle

- Oddity: negatives apply same rotation



Complex Numbers: Half Angle

- Semi-circle rep. all rotations



Complex Numbers: Summary

- In practice not used all that often
- Not sure why -- probably because angles are simple enough

Topics

- Angle (2D)
- Euler Angles (3D)
- Axis-Angle (3D)
- Matrix (2D)
- Matrix (3D)
- Complex number (2D)
- Quaternion (3D)

What is a Quaternion?

- Created as extension to complex numbers

$$a + b\mathbf{i}$$

becomes

$$w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

- Can rep as coordinates

$$(w, x, y, z)$$

- Or scalar/vector pair

$$(w, \mathbf{v})$$

What is Rotation Quaternion?

- Unit quat is rotation representation
  - also avoids f.p. drift

Why *4* values?

- One way to think of it:
- 2D rotation ->
  - One degree of freedom
- Unit complex number ->
  - One degree of freedom
- 3D rotation ->
  - Three degrees of freedom
- Unit quaternion ->
  - Three degrees of freedom

What is Rotation Quaternion?

- Unit quat (w, x, y, z)
- w represents angle of rotation θ
  - $w = \cos(\theta/2)$
- x, y, z from <u>normalized</u> rotation axis $\hat{\mathbf{r}}$
  - $(x\ y\ z) = \mathbf{v} = \sin(\theta/2)\cdot\hat{\mathbf{r}}$
- Often write as $(w, \mathbf{v})$
- In other words, modified axis-angle

Creating Rotation Quaternion

- So for example, if want to rotate 90° around z-axis:

$$w = \cos(45^\circ) = \sqrt{2}/2$$

$$x = 0 \cdot \sin(45^\circ) = 0$$

$$y = 0 \cdot \sin(45^\circ) = 0$$

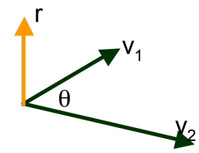
$$z = 1 \cdot \sin(45^\circ) = \sqrt{2}/2$$

$$\mathbf{q} = (\sqrt{2}/2, 0, 0, \sqrt{2}/2)$$

Creating Rotation Quaternion

- Another example
  - Have vector $\mathbf{v}_1$, want to rotate to $\mathbf{v}_2$
  - Need rotation vector $\hat{\mathbf{r}}$, angle θ

$$\mathbf{r} = \mathbf{v_1} \times \mathbf{v_2}$$
$$\theta = \operatorname{acos}(\hat{\mathbf{v}}_\mathbf{1} \bullet \hat{\mathbf{v}}_\mathbf{2})$$



  - Plug into previous formula

That's gives a particular solution. But suppose we want to generate a quaternion a little more programmatically. A case that comes up often is that we have a vector pointing in one direction, and we want to generate a quaternion that will rotate it to a new direction. One way we might think of doing this is just take the cross product to get our axis of rotation r, then take the dot product of the normalized vectors, and take the arccos of that to get the angle, and plug the result into the quaternion.

Creating Rotation Quaternion

- From Game Gems 1 (Stan Melax)
- Use trig identities to avoid arccos
  - Normalize $\mathbf{v}_1$, $\mathbf{v}_2$

$$\mathbf{r} = \hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2 \qquad s = \sqrt{2(1 + \hat{\mathbf{v}}_1 \bullet \hat{\mathbf{v}}_2)}$$

  Build quat

$$\mathbf{q} = (2s, \mathbf{r}/s)$$

  - More stable when $\mathbf{v}_1$, $\mathbf{v}_2$ near parallel

In most cases that will work, but there are some problems when v1 and v2 are pointing pretty much the same direction. Stan Melax has a great solution for this, which is to normalize v1 and v2, compute these quantities r and s, and then plug into the quaternion as follows.

Multiplication

- More complex (har) than complex
- Take $\mathbf{q}_0 = (w_{0,} \mathbf{v}_0)$ $\mathbf{q}_1 = (w_{1,} \mathbf{v}_1)$

$$\mathbf{q}_1\mathbf{q}_0 = (w_1 w_0 - \mathbf{v}_1 \bullet \mathbf{v}_0, w_1\mathbf{v}_0 + w_0\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_0)$$

- Non-commutative:

$$\mathbf{q}_1\mathbf{q}_0 \neq \mathbf{q}_0\mathbf{q}_1$$

So that provides a way to create a quaternion. Suppose we want to concatenate them. As with matrices and complex numbers, multiplication does the trick. However, in this case the multiplication operator is a little more complicated. Still, it's all simple vector math, so it isn't too bad. Note again that due to the cross product this is non-commutative.

Identity and Inverse

- Identity quaternion is $(1, 0, 0, 0)$
  - applies no rotation
  - remains at reference orientation
- $\mathbf{q}^{-1}$ is inverse
  - $\mathbf{q} \cdot \mathbf{q}^{-1}$ gives identity quaternion
- Inverse is same axis but opposite angle

Computing Inverse

- $(w,\ \mathbf{v})^{-1} = (\ \cos(\theta/2),\ \sin(\theta/2) \cdot \hat{\mathbf{r}}\ )$

$$\begin{aligned}(w,\mathbf{v})^{-1} &= (\cos(-\theta/2),\sin(-\theta/2)\hat{\mathbf{r}}) \\ &= (\cos(\theta/2),-\sin(\theta/2)\hat{\mathbf{r}}) \\ &= (w,-\mathbf{v})\end{aligned}$$

- Only true if $\mathbf{q}$ is unit
  - i.e. $\mathbf{r}$ is a unit vector

Vector Rotation

- Have vector $\mathbf{p}$, quaternion $\mathbf{q}$
- Treat $\mathbf{p}$ as quaternion $(0, \mathbf{p})$
- Rotation of $\mathbf{p}$ by $\mathbf{q}$ is $\mathbf{q}\,\mathbf{p}\,\mathbf{q}^{-1}$
- Vector $\mathbf{p}$ and unit quat $(w, \mathbf{v})$ boils down to

$$\mathbf{p}' = (1 - w^2)\mathbf{p} + 2(\mathbf{v} \bullet \mathbf{p})\mathbf{v} + 2w \cdot (\mathbf{v} \times \mathbf{p})$$

Possible to show that this formula is the same as the rotation formula for axis and angle.

Vector Rotation (cont'd)

- Why does $\mathbf{q}\ \mathbf{p}\ \mathbf{q}^{-1}$ work?
- Similar to complex w/half angle:
  - first multiply rotates halfway and into 4th dimension
  - second multiply rotates rest of the way, back into 3rd
- See references for more details

Vector Rotation (cont'd)

- Can concatenate rotation

$$\mathbf{q}_1 \cdot (\mathbf{q}_0 \cdot \mathbf{p} \cdot \mathbf{q}_0^{-1}) \cdot \mathbf{q}_1^{-1} = (\mathbf{q}_1 \cdot \mathbf{q}_0) \cdot \mathbf{p} \cdot (\mathbf{q}_1 \cdot \mathbf{q}_0)^{-1}$$

- Note multiplication order: right-to-left

Quaternion Interpolation

- As with complex numbers
  - Lerp

$$\mathbf{q}_t = (1-t)\mathbf{q}_0 + t\mathbf{q}_1$$

  - Slerp

$$\mathbf{q}_t = \frac{\sin(1-t)\alpha}{\sin\alpha}\mathbf{q}_0 + \frac{\sin t\alpha}{\sin\alpha}\mathbf{q}_1$$

Quaternion Interpolation

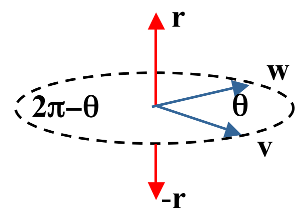
- Technique used depends on data
- Lerp generally good enough for motion capture (lots of samples)
  - Need to normalize afterwards
- Slerp only needed if data is sparse
  - Blow's method for simple interpolation
  - (Also need to normalize)

- These days, Blow says just use lerp. YMMV.

Demo

Interpolation Caveat

- $\mathbf{q}$ and $-\mathbf{q}$ rotate vector to same place
- But not quite the same rotation
- $-\mathbf{q}$ has axis $-\mathbf{r}$, with angle $2\pi-\theta$
- Causes problems with interpolation (different hemispheres)



This is due to the half-angle form of quaternions.

Interpolation Caveat

- How to test?
- If dot product of two interpolating quats is < 0, takes long route around sphere
- Solution, negate one quat, then interpolate
- Preprocess to save time

As mentioned…

Operation Wrap-Up

- Multiply to concatenate rotations
- Addition only for interpolation (don't forget to normalize)
- Be careful with scale
  - Quick rotation assumes unit quat
  - Don't do $(0.5 \cdot \mathbf{q}) \cdot \mathbf{p}$
  - Use lerp or slerp with identity quaternion

Summary

- Talked about orientation
- Formats good for internal storage
  - Angle
  - Matrices (2D or 3D)
  - Quaternions
- Formats good for UI
  - Euler angles
  - Axis-angle
- Complex numbers not really used

References

- Shoemake, Ken, "Animation Rotation with Quaternion Curves," *SIGGRAPH '85*, pp. 245-254.
- Shoemake, Ken, "Quaternion Calculus for Animation," SIGGRAPH Course Notes, *Math for SIGGRAPH*, 1989.
- Hanson, Andrew J., *Visualizing Quaternions*, Morgan Kaufman, 2006.
- Blow, Jonathan, "Hacking Quaternions," *Game Developer*, March 2002.
- Busser, Thomas, "PolySlerp: A fast and accurate polynomial approximation of spherical linear interpolation (Slerp)," *Game Developer*, February 2004.
- Van Verth, Jim, "Vector Units and Quaternions," *GDC 2002. http://www.essentialmath.com*