

## main goal

- goal: interpolation of rotations for animation
- side goal: nice arcball interface
- representation: quaternions instead of matrices
  - glorified axis-angle representation

## Interpolation of rotations

- given two frames  $\vec{w}^t R_a$  and  $\vec{w}^t R_b$
- desired: a frame that has its orientation  $\alpha$  of the way from a to b.
  - if one lerps the matrix entries, one does not get a rotation
  - if one somehow factors the rotation into XYZ rotation and lerps on these 3 scalars one gets odd answer

## interp as power

- Given the starting and ending rotations  $R_a$ , and  $R_b$ .
- define the “transition” rotation that takes from “a” to “b” as

$$R_b R_a^{-1}$$

- interpolate between “a” and “b” by starting with “a” and composing with it a “powered” version of the transition using

$$\vec{w}^t R_o = \vec{w}^t (R_b R_a^{-1})^\alpha R_a$$

## power of rotation

- a rotation can be interpreted as a right handed (ccw) rotation of  $-\pi < \theta < \pi$  radians about some axis.
- the axis of rotation is described by a unit norm 3-coordinate vector  $\hat{\mathbf{a}}$
- we can power the rotation by multiplying  $\theta$  by  $\alpha$ .
- it is annoying to get the axis angle from a matrix
- so lets store this data
- but if we store the axis and angle, how to do we compute with these?

## quaternion

- associate the rotation with the four tuple

$$\hat{\mathbf{Q}} = \begin{bmatrix} \cos(\frac{\theta}{2}) \\ \hat{\mathbf{a}} \sin(\frac{\theta}{2}) \end{bmatrix}$$

- where  $\hat{\mathbf{a}}$  is a unit coordinate vector describing the axis of rotation
  - wrt whatever the left of rule tells us from context
- this is called a unit quaternion
  - the sum of squares is 1
- the 1/2 will help us to easily calculate the composition of rotations
- note that we get a unique quaternion for  $\theta$  in  $-2\pi < \theta < 2\pi$ 
  - this will come back soon to complicate matters

## examples

- If  $\theta = 0$ , then this identity rotation is represented by the quaternion

$$\hat{\mathbf{Q}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- the rotation of 180 degrees,  $\theta = \pi$  radians about  $\hat{\mathbf{a}}$  (called a flip), the quaternion is

$$\hat{\mathbf{Q}} = \begin{bmatrix} 0 \\ \hat{\mathbf{a}} \end{bmatrix}$$

## The Power Operator

- define the power operator to represent scaling the rotation amount.

$$\begin{bmatrix} \cos(\frac{\theta}{2}) \\ \hat{\mathbf{a}} \sin(\frac{\theta}{2}) \end{bmatrix}^\alpha = \begin{bmatrix} \cos(\frac{\alpha\theta}{2}) \\ \hat{\mathbf{a}} \sin(\frac{\alpha\theta}{2}) \end{bmatrix}$$

- As  $\alpha$  goes from 0 to 1,  $\hat{\mathbf{Q}}^\alpha$  this describes a series of rotations starting from the non-rotation ending at the complete rotation  $\hat{\mathbf{Q}}$ .
- nb:  $\theta = \text{atan2}(\sin(\theta), \cos(\theta))$ .

## Antipodal Quaternions

- given a rotation of  $\theta \in \{0..\pi\}$  about  $\hat{\mathbf{a}}$ 
  - short rotation ccw.

$$\hat{\mathbf{Q}} = \begin{bmatrix} \cos(\frac{\theta}{2}) \\ \hat{\mathbf{a}} \sin(\frac{\theta}{2}) \end{bmatrix}$$

- we can instead rotate about  $\theta - 2\pi \in \{-2\pi..-\pi\}$ 
  - long rotation cw
  - this is the same “rotation”
- its quaternion is

$$\begin{bmatrix} \cos(\frac{\theta-2\pi}{2}) \\ \hat{\mathbf{a}} \sin(\frac{\theta-2\pi}{2}) \end{bmatrix} = \begin{bmatrix} \cos(\frac{\theta}{2} - \pi) \\ \hat{\mathbf{a}} \sin(\frac{\theta}{2} - \pi) \end{bmatrix} = \begin{bmatrix} -\cos(\frac{\theta}{2}) \\ -\hat{\mathbf{a}} \sin(\frac{\theta}{2}) \end{bmatrix} = -\hat{\mathbf{Q}}$$

- so the antipodal unit quaternions  $\hat{\mathbf{Q}}$  and  $-\hat{\mathbf{Q}}$  represent the exact same rotation
- but the first one powers ccw the short way, and the other powers cw the long way.
- first one has positive  $w$  coordinate.

## flip

- if we do the whole thing again but start with  $\theta \in \{\pi..2\pi\}$ 
  - long way ccw
- the second way is short way cw
- the second quat will have positive  $w$  coordinate.
- moral: if you have a quaternion that you want to power, you probably want the short way.
- to get the short way you pick either the quaternion, or its antipode, depending on which of those has positive  $w$ .

## quat to mat

- we can turn them into matrices to create the rotational part of the  $O_i$  and  $S$  matrices
- from our earlier rotation notes, we know how to obtain a rot matrix from a quat  $[w, x, y, z]^t$

$$\begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy & 0 \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx & 0 \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## multiplication of Quaternions

- given 2 unit quats  $\mathbf{Q}_1, \mathbf{Q}_2$ , representing two rotation matrices  $R_1, R_2$  i can obtain a quat representing  $R_1 R_2$  using the following formula (called quaternion multiplication)

$$\mathbf{Q}_1 \mathbf{Q}_2 = \begin{bmatrix} w_1 \\ \mathbf{v}_1 \end{bmatrix} \begin{bmatrix} w_2 \\ \mathbf{v}_2 \end{bmatrix} = \begin{bmatrix} (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2) \\ (w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \end{bmatrix}$$

- also, the inverse of a quaternion is obtained by negating its  $x, y, z$  values.

$$\begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}^{-1} = \begin{bmatrix} w \\ -x \\ -y \\ -z \end{bmatrix}$$

## applying rotation to a coordinate vector

- we could take a unit quaternion  $\hat{\mathbf{Q}}$ , turn it into a matrix  $R$  and then do matrix vector multiplication  $\mathbf{c}' = R\mathbf{c}$
- if we want, it is more efficient to do the following sequence of quaternion multiplies.

$$\begin{bmatrix} 0 \\ \mathbf{c}' \end{bmatrix} = \hat{\mathbf{Q}} \begin{bmatrix} 0 \\ \mathbf{c} \end{bmatrix} \hat{\mathbf{Q}}^{-1}$$

– don't worry that we have some non-unit quaternions here.

## in OpenGL

- for each object/skycam, we will store not a matrix, but an “RBT” data type
- the R part will be a quaternion, instead of a matrix
- when we need to interp a rotation, we have access to the quaternion.
- for rendering, we convert to matrix and give it to gl.

## RBT data type

- details

```
class Rbt
{
    Vector3 t_;
    Quaternion r_;
    ...
    Matrix4 getMatrix() const{
        return Matrix4::makeTranslation(t_)
            * r_.getMatrix();
    }
}
```

## rbt ops

- recall with matrices we needed to do things like  $O'_i = AQA^{-1}O_i$
- now we need to do this where  $O, O', A, Q$  are rbts
- so we need to be able to compose and invert rbts

## RBT composition

- suppose i compose two rbts, what is the rotation and translation of the result?
- lets first thing of rbts as matrices

$$\begin{aligned} & \left( \begin{bmatrix} 1 & t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix} \right) \left( \begin{bmatrix} 1 & t_2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} \right) = \\ & \begin{bmatrix} 1 & t_1 \\ 0 & 1 \end{bmatrix} \left( \begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & t_2 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} = \\ & \begin{bmatrix} 1 & t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_1 t_2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} = \\ & \begin{bmatrix} 1 & t_1 \\ 0 & 1 \end{bmatrix} \left( \begin{bmatrix} 1 & r_1 t_2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} = \\ & \left( \begin{bmatrix} 1 & t_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & r_1 t_2 \\ 0 & 1 \end{bmatrix} \right) \left( \begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} \right) = \\ & \begin{bmatrix} 1 & t_1 + r_1 t_2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 r_2 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned}$$

- now lets think of rbts as vec,quat pairs.
- $r_1 r_2$  will be implemented as quat quat mul
- $r_1 t_2$  will be implmented as quat vec mul
- $t_1 + (r_1 t_2)$  will be implmented as vec vec add.

## rbt inverse

- given  $M = TR$
- we know  $M^{-1} = R^{-1}T^{-1}$
- use the above tricks to directly express the components  $M^{-1} = T'R'$
- use the expression to directly implement rbt inversion.

## Arcball rotation interface

- interpret mouse moves as rotations
- this will give us the Q in “do Q to O wrt A”
- the “A” will be one of our special frames
- many possible ways to define the Q.
- “arcball” has the property that if one starts the mouse at point “a” moves it “b” and then to “c”, the interpreted rotation will not depend on the position of “b”
- lets look at demo

## here's how 1

- Imagine a sphere of some size in space
- centered on object (for cube-sky) or world origin (for world-sky) at  $\tilde{c}$

- When the user clicks on a pixel a ray from the eye goes out and hits some point  $\tilde{p}_0$  on the sphere.
  - if it misses, there is some closets point instead
- Compute coordinates for the vector  $\tilde{p}_0 - \tilde{c}$ , and normalize it
- we should be using coords wrt cube-sky coord
- but since we are using vectors, we can just use sky coords  $\hat{v}_0$

### here's how 2

- this ends up a bit too involved so we will use a screen space hack that approximates the above.
- we will give you code that takes in the matrix and window info, and sphere info, and tells you the radius and center of the sphere in 2d pixel coords
- imagine a hemi sphere sitting over the screen with a 3rd coordinate
- user's click gives you a 2d pixel coordinate
- pick the 3rd coordinate to lift this up to the sphere
- in this space, compute coordinates for the vector  $\tilde{p}_0 - \tilde{c}$ , and normalize it to obtain the coordinates  $\hat{v}_0$

### next click

- When the user now moves the mouse to some second point, repeat these steps to obtain  $\hat{v}_1$
- Imagine the arc along the hemisphere connecting  $\tilde{p}_0$  and  $\tilde{p}_1$ . The (non unit) axis of rotation can be computed in coordinates as  $\hat{a} = \hat{v}_0 \times \hat{v}_1$ . The angle  $\phi$  of the arc satisfies  $\cos(\phi) = \hat{v}_0 \cdot \hat{v}_1$
- Interpret this user interaction as a request to perform a rotation of  $\theta = 2\phi$  around  $\hat{a}$ .
- The unit quaternion that performs this rotation can be computed as

$$\begin{bmatrix} \cos(\frac{\theta}{2}) \\ \hat{a} \sin(\frac{\theta}{2}) \end{bmatrix} = \begin{bmatrix} \cos(\phi) \\ \hat{a} \sin(\phi) \end{bmatrix} = \begin{bmatrix} \hat{v}_0 \cdot \hat{v}_1 \\ \hat{v}_0 \times \hat{v}_1 \end{bmatrix} = \begin{bmatrix} 0 \\ \hat{v}_1 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{v}_0 \end{bmatrix}$$

### path independence

- suppose the user moves how to new sphere point, defining a new direction  $\hat{v}_2$
- lets concatenate the two interpreted rots
- since we are moving object wrt (unchanged) eye-obj c.s., this ends up as left concatenation.

$$\begin{aligned} \begin{bmatrix} \hat{v}_1 \cdot \hat{v}_2 \\ \hat{v}_1 \times \hat{v}_2 \end{bmatrix} \begin{bmatrix} \hat{v}_0 \cdot \hat{v}_1 \\ \hat{v}_0 \times \hat{v}_1 \end{bmatrix} &= \begin{bmatrix} 0 \\ \hat{v}_2 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{v}_1 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{v}_1 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{v}_0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ \hat{v}_2 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{v}_0 \end{bmatrix} \end{aligned}$$

- the intermediate request is irrelevant
- downside: the imaginary arc of  $\phi$  radians is interpreted as a request for a rotation of  $\theta = 2\phi$ .
  - moving across the hemisphere requests full 360 degree turn.

### when moving eye

- when we move the mouse “down”, we want to rotate the camera “up” so we swap the order

$$\begin{bmatrix} 0 \\ \hat{v}_0 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{v}_1 \end{bmatrix}$$

- since we are updating “E” wrt special eye-world cs, this will end up as right concatenation, and we will still get path independence.