motivation

- for animation, we will want to interpolate between frames in a natural way.
- we will study quaternions as alternative to rot mats

$$R = \begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix}$$

- later we will add back in the translations.

interpolation setup

- desired object frame for "time=0": $\vec{\mathbf{o}}_0^t = \vec{\mathbf{w}}^t R_0$
- desired object frame for "time=1": $\vec{\mathbf{o}}_1^t = \vec{\mathbf{w}}^t R_1$,
- we wish to find a sequence of rhon frames $\vec{\mathbf{o}}_\alpha^t$, for $\alpha \in [0..1]$, that naturally goes from $\vec{\mathbf{o}}_0^t$ to $\vec{\mathbf{o}}_1^t$.

bad ideas 1

- lin interp of matrices $R_\alpha := (1-\alpha)R_0 + (\alpha)R_1$ and then set $\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t R_\alpha$.
- each basis vector simply moves along a straight line.
- In this case, the intermediate R_α are not rotation matrices
- see fig

bad idea 2

- factor both R_0 and R_1 into 3, so-called, Euler angles
- These three scalar values could each be linearly interpolated using α. and used to generate intermediate rotations
- not natural,
- not invariant to choice of world frame
- this property is called left invariance
- see figure
- we need an intrinsic geometric operation (describable independent of coordinates or world frame)

lets get organized

- it must be the case (simply since $\vec{\mathbf{o}}_1^t$ is a frame) that $\vec{\mathbf{o}}_1^t = \vec{\mathbf{w}}^t Z R_0$ for some unique matrix Z.
- so this gives us $\vec{\mathbf{w}}^t R_1 = \vec{\mathbf{w}}^t Z R_0$
- so $Z = R_1 R_0^{-1}$ and must be a rotation matrix
- which we will call the transition matrix
- "do Z to $\vec{\mathbf{o}}_0^t$ wrt $\vec{\mathbf{w}}^t$, will map $\vec{\mathbf{o}}_0^t \Rightarrow \vec{\mathbf{o}}_1^t$
- the transformation is uniquely determined by $\vec{\mathbf{o}}_0^t$ and $\vec{\mathbf{o}}_1^t$.
  - (though the matrix Z itself depends on $\vec{\mathbf{w}}^t$)

what we want

- This Z matrix can, as any rotation matrix, be thought of as a rotation of some θ degrees about *some* axis $[k_x, k_y, k_z]^t$,
- suppose we had a power operator: $(R_1 R_0^{-1})^\alpha$

- which gave us a rotation about $[k_x, k_y, k_z]^t$ by $\alpha\theta$ degrees instead.

- then we could set

$$R_\alpha := (R_1 R_0^{-1})^\alpha R_0$$

and set

$$\begin{aligned} \vec{\mathbf{o}}_\alpha^t &= \vec{\mathbf{w}}^t R_\alpha \\ &= \vec{\mathbf{w}}^t (R_1 R_0^{-1})^\alpha R_0 \end{aligned}$$

result

- this is a sequence of frames obtained by more and more rot about a singe axis
  - read right to left
- correct start and finish: $\vec{\mathbf{w}}^t (R_1 R_0^{-1})^0 R_0 = \vec{\mathbf{w}}^t R_0 = \vec{\mathbf{o}}_0$, and $\vec{\mathbf{w}}^t (R_1 R_0^{-1})^1 R_0 = \vec{\mathbf{w}}^t R_1 = \vec{\mathbf{o}}_1$

aside: Cycles

- $R_1 R_0^{-1}$ matrix can be thought of as a rotation of some $\theta + n2\pi$ degrees for any positive or negative integer n, around some fixed axis.
- not relevant for linear tform on vectors, but is relevant for interpolation
- the natural choice is to choose n such that $|\theta + n2\pi|$ is minimal.
- you can also negate the axis and all of the rotations, choose the minimal rotation, and you will get the same sequence of frames

left invariance

- so we have a specific seqence of frames obtained by rotations about a specific axis
- this sequence choice depends only on $\vec{\mathbf{o}}_0^t$ and $\vec{\mathbf{o}}_1^t$.
  - Not any choice of world frame.
- so we have the desired left invariance.

hard part

- hard part: factor $R_1 R_0^{-1}$, into its axis/angle form.
- main quat idea: is to keep track of the axis and angle at all times but in a way that allows our manipulations
- this will allow us to do this interpolation
- it also could help in general with avoiding numerical drift away from RBTs.

the representation

- a quaternion is 4 tuple with operations
- written

$$\begin{bmatrix} w \\ \hat{\mathbf{c}} \end{bmatrix}$$

where w is a scalar and $\hat{\mathbf{c}}$ is a coordinate 3-vector.

- a rotation of θ degrees about a unit length axis $\hat{\mathbf{k}}$, is represented as

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

- oddity: the division by 2 will be needed to make the operations work out as needed.

antipodes

- Note that a rotation of $-\theta$ degrees about the axis $-\hat{\mathbf{k}}$ gives us the same quaternion.
- A rotation of $\theta + 4\pi$ degrees about an axis $\hat{\mathbf{k}}$ also gives us the same quaternion.
- a rotation of $\theta + 2\pi$ degrees about an axis $\hat{\mathbf{k}}$, which in fact is the same linear transfomation, gives us the negated quaternion
- so antipodes represent the same rotation *transformation*
  - but heads up regarding cycles and power

unit norm quats == rotations

- squared norm is sum of 4 squares.
- Any quaternion of the form

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

  has a unit norm
- Conversely, any such *unit* norm quaternion can be interpreted (along with its negation) as a unique rotation matrix.

examples

- identity rotation:

$$\begin{bmatrix} 1 \\ \hat{\mathbf{0}} \end{bmatrix}, \begin{bmatrix} -1 \\ \hat{\mathbf{0}} \end{bmatrix},$$

- flip rotation

$$\begin{bmatrix} 0 \\ \hat{\mathbf{k}} \end{bmatrix}, \begin{bmatrix} 0 \\ -\hat{\mathbf{k}} \end{bmatrix},$$

Operations

- quat * quat multiply

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

- where $\cdot$ and $\times$ are the dot and cross product on 3 dimensional coordinate vectors.
- correctly models rot matrix * rot matrix multiplication!
- unit quat multiplicative inverse

$$\begin{bmatrix} \cos(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2})\hat{\mathbf{k}} \end{bmatrix}^{-1} = \begin{bmatrix} \cos(\frac{\theta}{2}) \\ -\sin(\frac{\theta}{2})\hat{\mathbf{k}} \end{bmatrix}$$

quat vector multiply setup

- start with 4-coordinate vector $\mathbf{c} = [\hat{\mathbf{c}}, 1]^t$,
- left multiply it by a 4 by 4 rotation matrix R, to get

$$\mathbf{c}' = R\mathbf{c}$$

- with result of form $\mathbf{c}' = [\hat{\mathbf{c}}', 1]^t$.

quat vector multiply setup

- let R be represented with the unit norm quaternion

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

- use cvec3 $\hat{\mathbf{c}}$ to create the non unit norm quaternion

$$\begin{bmatrix} 0 \\ \hat{\mathbf{c}} \end{bmatrix}$$

- perform the following triple quaternion multiplication:

$$\begin{bmatrix} \cos(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2})\hat{\mathbf{k}} \end{bmatrix} \begin{bmatrix} 0 \\ \hat{\mathbf{c}} \end{bmatrix} \begin{bmatrix} \cos(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2})\hat{\mathbf{k}} \end{bmatrix}^{-1}$$

- tada: result is of form

$$\begin{bmatrix} 0 \\ \hat{\mathbf{c}}' \end{bmatrix}$$

- we will write this in code as: quat * cvec = cvec
- moral: quaternions encode the axis, and let us do ops.

Power

- lets define the power operator acting on a unit quaternion

$$\begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

- first extract the unit axis $\hat{\mathbf{k}}$ by normalizing the three last entries of the quaternion.
- this gives us

$$\begin{bmatrix} w \\ \beta\hat{\mathbf{k}} \end{bmatrix}$$

  - with $w^2 + \beta^2 = 1$ (on unit circle).
- Next, extract θ using the `atan2` function in C++.
- atan(β, w) returns a unique $\phi \in [-\pi..\pi]$ such that $\sin(\phi) = \beta$ and $\cos(\phi) = w$.
- this gives us

$$\begin{bmatrix} \cos(\phi) \\ \sin(\phi)\hat{\mathbf{k}} \end{bmatrix}$$

power II

- so we get a unique value $\theta/2 \in [-\pi..\pi]$, and thus a unique $\theta \in [-2\pi..2\pi]$.
- this gives us

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

- define

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

  As α goes from from 0 to 1, we get a series of rotations with angles going between 0 and θ.

power antipodes and cycle

- but what if the transition quaterion $\begin{bmatrix} \cos(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2})\hat{\mathbf{k}} \end{bmatrix}$
- ..represents a θ of more than 180 degrees:
- in particular, if $\cos(\frac{\theta}{2}) < 0$, then $|\theta| \in [\pi..2\pi]$,
  - so $\alpha\theta$ would go more than 180 degrees which we don't want during interpolation
- in this case suppose we had swapped to the antipode before calling power
- then $\cos(\frac{\theta}{2}) > 0$, we get $\theta/2 \in [-\pi/2..\pi/2]$
  - and thus $\theta \in [-\pi..\pi]$.
- so when we interpolate, before calling the power operator, we first check the sign of the first coordinate, and conditionally negate the quaternion.
- we call this the conditional negation operator *cn*.

to interpolate

- to interpolate between two frames related to world frame by R_0 and R_1,
- and suppose that these two matrices correspond to the two quaternions

$$\begin{bmatrix} \cos(\frac{\theta_0}{2}) \\ \sin(\frac{\theta_0}{2})\hat{\mathbf{k}}_0 \end{bmatrix}, \begin{bmatrix} \cos(\frac{\theta_1}{2}) \\ \sin(\frac{\theta_1}{2})\hat{\mathbf{k}}_1 \end{bmatrix}$$

- we output

$$\left(cn\left(\begin{bmatrix} \cos(\frac{\theta_1}{2}) \\ \sin(\frac{\theta_1}{2})\hat{\mathbf{k}}_1 \end{bmatrix} \begin{bmatrix} \cos(\frac{\theta_0}{2}) \\ \sin(\frac{\theta_0}{2})\hat{\mathbf{k}}_0 \end{bmatrix}^{-1} \right) \right)^{\alpha} \begin{bmatrix} \cos(\frac{\theta_0}{2}) \\ \sin(\frac{\theta_0}{2})\hat{\mathbf{k}}_0 \end{bmatrix}$$

slerping

- this is called called *spherical linear interpolation* or just *slerping* since it happens to match moving on a great circle in R^4.
  - see book for more discussion
- an even easier hack is to do 4D lerp and renormalization
  - useful for n-way blends.

Code

- `Quat` is four-tuple of reals.
- we provide qq multiplication `(q1 * q2)`
- `inv(q)`.
- `(q * c)`,
  - copy over the 0/1 fourth coordinate

- `MakeRotation` functions
- later on (asst 5) you will code the power operator: `pow(q,alpha)`. and `slerp(q0,q1,alpha)`.
  - Remember the conditional negation

Rbt data structure

- lets now build a data structure to represent an rbt
- recall

$$\begin{array}{ccc} & A & = & TR \\ \begin{bmatrix} r & t \\ 0 & 1 \end{bmatrix} & = & \begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix} \end{array}$$

- our class

```
class RigTForm{
  Cvec3 t;
  Quat r;
};
```

rbt Interpolation

- given two frames, $\vec{\mathbf{o}}_0^t = \vec{\mathbf{w}}^t O_0$ $\vec{\mathbf{o}}_1^t = \vec{\mathbf{w}}^t O_1$
- Given two RBTs
  - we will write it as matrices $O_0 = (O_0)_T (O_0)_R$ and $O_1 = (O_1)_T (O_1)_R$, but implement in our rigTform data type.
- interpolate between them by: linearly interpolating the two translation to get: T_α,
- slerp between the rotation quaternions to obtain the rotation R_α,
- set the interpolated RBT O_α to be $T_\alpha R_\alpha$.
- set $\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t O_\alpha$.

behavior

- origin of the frame travels in a straight line with constant velocity,
  - read right to left
- the vector basis of the frame rotates with constant angular velocity about a fixed axis.
- physically natural if origin is at center of mass.
- has intrinsic description, so it is left invariant
- note: origin plays special role. if use different object frames for same geometry, we get different interpolation
  - not right invariant (see fig)

code

- change `skyRbt` and `objectRbt[]` to be `RigTform` data type instead of `Matrix4`.
- in fact almost

all

of the C++ Matrix4's should get replaced!

- we provide `RigTForm makeXRotation(const double ang)`

- you provide code for the product of a `RigTForm A` and a `Cvec4 c`, to return `A.r * c + Cvec4(A.t,0)`.
  - what if c has 0 fourth coordinate, then no translation should be done!

rbt * rbt

- let us look at the product of two such rigid body transforms.

$$\begin{bmatrix} i & t_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} i & t_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 & r_1t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} i & t_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & r_1t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} i & t_1 + r_1t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1r_2 & 0 \\ 0 & 1 \end{bmatrix}$$

- the result is a new rigid transform with translation $t_1 + r_1t_2$ and rotation r_1r_2.
  - use this to code up the * op.
  - mind the Cvec3s (the t's) and Cvec4s (needed for q*v).

inv operator

- likewise for inverse

$$\left(\begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix}\right)^{-1} =$$
$$\begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix}^{-1}\begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix}^{-1} =$$
$$\begin{bmatrix} r^{-1} & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & -t \\ 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} r^{-1} & -r^{-1}t \\ 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} i & -r^{-1}t \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r^{-1} & 0 \\ 0 & 1 \end{bmatrix}$$

- the result is a new rigid body transform with translation $-r^{-1}t$ and rotation r^{-1}.

more code

- in GLSL, you will still use its matrix data type.
- the only Matrix4s that will survive are the projMatrix, the MVM and the NMVM, which get sent to your shaders.
- also, when we need to do object scaling, we cannot capture this in an RigTform, so this will also be a Matrix4 used in creating the MVM.
- to communicate with the vertex shader using 4 by 4 matrices, we need a procedure `Matrix4 makeRotation(quat q)` which turns quat into a 4 by 4 rotation matrix.
- Then, the matrix for a rigid body transform can be computed as

```
matrix4 rigTFormToMatrix(const RigTform& rbt){
  matrix4 T = makeTranslation(rbt.getTranslation());
  matrix4 R = quatToMatrix(rbt.getRotation());
  return  T * R;
}
```

- Thus, our drawing code starts with

```
Matrix4 MVM = rigTFormToMatrix(inv(eyeRbt) * objRbt);
\\ can right multiply scales here
Matrix4 NMVM = normalMatrix(MVM);
sendModelViewNormalMatrix(curSS, MVM,NMVM);
```

- we will not need any code that takes a `Matrix4` and converts it to a `Quat`.
- scale will still represented by a `Matrix4`. (more later)