motivation

- for animation, we will want to interpolate between frames in a natural way.
- for now, we want to also improve our rotation interface
- 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.

recall this beast

- one can specify rotation about an axis represented by a unit norm 3-coordinate vector \( \hat{k} := [k_x, k_y, k_z]^T \) and a rotation angle \( \theta \) using the matrix
  
  \[
  \begin{bmatrix}
  k_x^2v + c & k_xk_yv - k_zs & k_xk_zv + k_y s & 0 \\
  k_yk_xv + k_zs & k_y^2v + c & k_yk_zv - k_x s & 0 \\
  k_zk_xv - k_y s & k_zk_yv + k_x s & k_z^2v + c & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

- where \( v \equiv 1 - c \)
- the (geometric) axis of rotation \( \hat{k} \) is determined by how the matrix is placed into an expression, using the “left of” rule

interpolation setup

- desired object frame for “time=0”: \( \vec{\omega}_0 = \vec{w}^T R_0 \)
- desired object frame for “time=1”: \( \vec{\omega}_1 = \vec{w}^T R_1 \)
- we wish to find a sequence of rhon frames \( \vec{\omega}_\alpha \), for \( \alpha \in [0..1] \), that naturally goes from \( \vec{\omega}_0 \) to \( \vec{\omega}_1 \).

bad ideas 1

- lin interp of matrices \( R_\alpha := (1 - \alpha)R_0 + (\alpha)R_1 \) and then set \( \vec{\omega}_\alpha = \vec{w}^T R_\alpha \).
- in the result, each basis vector simply moves along a straight line.
  - since \( R_\alpha \mathbf{e} = (1 - \alpha)R_0 \mathbf{e} + (\alpha)R_1 \mathbf{e} =: (1 - \alpha)\mathbf{c}_0 + (\alpha)\mathbf{c}_1 \)
- In this case, the intermediate \( R_\alpha \) 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 \( \alpha \). 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)
  - note: for a 2 Euler dof camera, this is a reasonable solution.
lets back up

• given two frames, there must be a unique affine transformation $\mathcal{Z}$ that maps $\vec{o}'_0$ to $\vec{o}'_1$.
  
  – here $\mathcal{Z}$ is a map, not a matrix, (so no need to specify the wrt frame).
• since $\vec{o}'_0$ and $\vec{o}'_1$ are both rhon frames with the same origin, $\mathcal{Z}$ must be a rotation.
• a rotation can always be described using a fixed axis $\vec{k}$ and a rotation amount $\theta$.
  
  – this is essentially unique
• let us define $\mathcal{Z}^\alpha$ to be a rotation about the same $\vec{k}$, but by $\alpha\theta$ degrees instead.
• applying $\mathcal{Z}^\alpha$ to $\vec{o}'_0$, with $\alpha \in [0..1]$, this gives us a natural interpolating sequence $\vec{o}'_{\alpha}$.
  
  – this construction is left invariant, and essentially unique.

uniqueness and cycles

• actually, $\mathcal{Z}$ can be thought of as a rotation of some $\theta + n2\pi$ degrees for any positive or negative integer $n$, around a fixed axis $\vec{k}$
  
  – 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.
  
  – this might be a negative amount.
• actually, $\mathcal{Z}$ can also be thought of as a rotation of $-\theta - n2\pi$ degrees around $-\vec{k}$
  
  – but choosing the minimal rotation will get the same sequence of frames

using world frame

• lets work this out with our world frame + matrix representation
• the mapping $\vec{o}'_0 \Rightarrow \vec{o}'_1$ can be written as $\vec{w}'R_0 \Rightarrow \vec{w}'(R_1R_0^{-1})R_0$
• let us define $\mathcal{Z} := (R_1R_0^{-1})$
• so $\mathcal{Z}$ must be the unique rotation matrix, such that “doing $\mathcal{Z}$ wrt $\vec{w}'$” exactly expresses $\mathcal{Z}$.
• the $\mathcal{Z}$ matrix is based on a coordinate axis $\hat{k}$.
• suppose we could compute a “power” operator $\mathcal{Z}^\alpha$ that represents a scaled rotation about $\hat{k}$.
• then as $\alpha$ goes from $0..1$, the sequence $\vec{w}'Z^\alpha R_0$, goes from $\vec{o}'_0$ to $\vec{o}'_1$ by rotating $\vec{o}'_0$ more and more about a single axis
  
  – this exactly implements what we were looking for

hard part

• hard part: factor $R_1R_0^{-1}$, 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.
• it will also make an arcball interface very easy.

the representation

• a quaternion is 4 tuple with operations
written
\[
\begin{bmatrix}
w \\
c
\end{bmatrix}
\]

where \( w \) is a scalar and \( c \) is a coordinate 3-vector.

- suppose an axis \( \vec{k} \) is represented by a unit length coordinate 3-vector \( \hat{k} \)
- a rotation of \( \theta \) degrees about \( \hat{k} \), is represented as
  \[
  \begin{bmatrix}
  \cos\left(\frac{\theta}{2}\right) \\
  \sin\left(\frac{\theta}{2}\right)\hat{k}
  \end{bmatrix}
  \]

- 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 \(-\hat{k}\) gives us the same quaternion.
- A rotation of \(\theta + 4\pi\) degrees about an axis \(\hat{k}\) also gives us the same quaternion.
- a rotation of \(\theta + 2\pi\) degrees about an axis \(\hat{k}\), which in fact is the same linear transformation, gives us the negated quaternion
- so antipodal quaternions represent the same rotation transformation
  - but heads up regarding cycles and power

examples

- \( \theta = 0 \):
  \[
  \begin{bmatrix}
  1 \\
  \hat{0}
  \end{bmatrix}
  \]

- \( \theta = 2\pi \):
  \[
  \begin{bmatrix}
  -1 \\
  \hat{0}
  \end{bmatrix}
  \]

- both represent the identity rotation
- \( \theta = \pi \)
  \[
  \begin{bmatrix}
  0 \\
  \hat{k}
  \end{bmatrix}
  \]

- \( \theta = -\pi \)
  \[
  \begin{bmatrix}
  0 \\
  -\hat{k}
  \end{bmatrix}
  \]

- both represent the same flip rotation.

unit norm quats == rotations

- squared norm is sum of 4 squares.
- Any quaternion of the form
  \[
  \begin{bmatrix}
  \cos\left(\frac{\theta}{2}\right) \\
  \sin\left(\frac{\theta}{2}\right)\hat{k}
  \end{bmatrix}
  \]

has a unit norm
Conversely, as we will see next any unit norm quaternion can be interpreted as above with a $\hat{k}$ and $\theta$

- this construction also implicitly shows that the interpretation is unique up to additions of $4\pi$, as well as negation of $\hat{k}$ and $\theta$.

**Extract**

- lets see how to do this factoring on a unit quaternion

\[
\begin{bmatrix}
w \\
x \\
y \\
z
\end{bmatrix}
\]

- recall that $||\beta\hat{k}|| = \beta||\hat{k}||$.
- set $\beta$ to be the norm of $[x, y, z]^t$.
- extract the unit axis $\hat{k}$ by normalizing $[x, y, z]^t$.
- this gives us a positive $\beta$ and a $\hat{k}$ so that

\[
\begin{bmatrix}
w \\
\beta \hat{k}
\end{bmatrix} = 
\begin{bmatrix}
w \\
x \\
y \\
z
\end{bmatrix}
\]

- with $w^2 + \beta^2 = 1$ (on unit circle).
- $\beta^2$ is the sum of squares of $[x, y, z]$.

**angle**

- we use both the sin and cos. (using only one of them would leave us with 2 solutions).
- Next, extract $\theta$ using the *atan2* function in C++.
- $\text{atan}(\beta, w)$ returns a unique $\phi \in [-\pi..\pi]$ such that $\sin(\phi) = \beta$ and $\cos(\phi) = w$.
- this gives us $\phi$ and $\hat{k}$ so that

\[
\begin{bmatrix}
\cos(\phi) \\
\sin(\phi) \hat{k}
\end{bmatrix} = 
\begin{bmatrix}
w \\
x \\
y \\
z
\end{bmatrix}
\]

**angle 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 $\theta$ and $\hat{k}$ such that

\[
\begin{bmatrix}
\cos(\frac{\theta}{2}) \\
\sin(\frac{\theta}{2}) \hat{k}
\end{bmatrix} = 
\begin{bmatrix}
w \\
x \\
y \\
z
\end{bmatrix}
\]

- so we are done.

**power**

- define

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

4
• where $\theta$ and $\hat{k}$ were extracted uniquely, as above.
  – where $\theta \in [-2\pi..2\pi]$
• As $\alpha$ goes from 0 to 1, we get a series of rotations with angles going between 0 and $\theta$.

short quaternion

• given a quaternion on which we want to power: $
\begin{bmatrix}
\cos\left(\frac{\theta}{2}\right) \\
\sin\left(\frac{\theta}{2}\right)\hat{k}
\end{bmatrix}$
• suppose $\cos\left(\frac{\theta}{2}\right) > 0$
• this means $\theta/2 \in [-\pi/2..\pi/2]$
  – and thus $\theta \in [-\pi..\pi]$.
• so when we interpolate, we will get a sequence that spans less than 180. good.

long quaternion

• but suppose $\cos\left(\frac{\theta}{2}\right) < 0$,
• this means $|\theta/2| \in [\pi/2..\pi]$ 
  – and thus $|\theta| \in [\pi..2\pi]$.
  – so $\alpha \theta$ would go more than 180 degrees which we are not going to want during interpolation
• in this case suppose we can simply negate the quaternion, giving us a short quaternion.
• so when we interpolate, before calling the power operator, we should first check the sign of the first coordinate, 
  and conditionally negate the quaternion.
• we call this the conditional negation operator $cn$.

Operations

• magic trick number 1.
• quat * quat multiply

\[
\begin{bmatrix}
w_1 \\
c_1
\end{bmatrix}
\begin{bmatrix}
w_2 \\
c_2
\end{bmatrix} = 
\begin{bmatrix}
(w_1w_2 - c_1 \cdot c_2) \\
(w_1c_2 + w_2c_1 + c_1 \times c_2)
\end{bmatrix}
\]
• 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

\[
\begin{bmatrix}
\cos\left(\frac{\theta}{2}\right) \\
\sin\left(\frac{\theta}{2}\right)\hat{k}
\end{bmatrix}^{-1} = 
\begin{bmatrix}
\cos\left(\frac{\theta}{2}\right) \\
-\sin\left(\frac{\theta}{2}\right)\hat{k}
\end{bmatrix}
\]
• easy to verify

to interpolate

• if we want to interpolate between $\vec{w}^tR_0$ and $\vec{w}^tR_1$
• and suppose that $R_0$ and $R_1$ are modeled as $q_0$ and $q_1$.
• recall the desired interpolation frames in matrix form is $\vec{w}^t(R_1R_0^{-1})^\alpha R_0$
• so we calculate $(cn(q_1q_0^{-1}))^\alpha q_0$
• this is called slerping (see book for more).
quat vector multiply setup

- magic trick number 2
- start with arbitrary 3-coordinate vector \( \mathbf{c} \), representing a a vector.
- left multiply it by a 3 by 3 rotation matrix \( \mathbf{r} \), to get
  \[
  \mathbf{c}' = \mathbf{r}\mathbf{c}
  \]

quat vector multiply setup

- let \( \mathbf{r} \) be represented with the unit norm quaternion \( \mathbf{q} \)
- use \( \textcvec{3} \mathbf{c} \) to create the non unit norm quaternion
  \[
  \begin{bmatrix}
  0 \\
  \mathbf{c}
  \end{bmatrix}
  \]
- perform the following triple quaternion multiplication:
  \[
  \mathbf{q} \begin{bmatrix}
  0 \\
  \mathbf{c}
  \end{bmatrix} \mathbf{q}^{-1}
  \]
- tada: result is of form
  \[
  \begin{bmatrix}
  0 \\
  \mathbf{c}'
  \end{bmatrix}
  \]
- and we might write this \( \mathbf{q}\mathbf{c} = \mathbf{c}' \)
- in our \texttt{Quat} class, we will give you: \texttt{quat * cvec3 = cvec3}

Rbt data structure

- lets now build a data structure to represent an rbt
- recall
  \[
  \begin{bmatrix}
  \mathbf{r} & \mathbf{t} \\
  0 & 1
  \end{bmatrix} = \begin{bmatrix}
  \mathbf{i} & \mathbf{t} \\
  0 & 1
  \end{bmatrix} \begin{bmatrix}
  \mathbf{r} & 0 \\
  0 & 1
  \end{bmatrix}
  \]
- we can encode this information in the following data type

  ```
  class RigTForm{
    Cvec3 t;
    Quat q;
  };
  ```
- this data will be always interpreted in the \( \mathbf{TR} \) order above.

\texttt{rbt * Cvec4}

- you will write code for the product of a \texttt{RigTForm} \( \mathbf{A} \) and a \texttt{Cvec4} \( \mathbf{c} \), (where the last entry is 0/1).
- only translate if the fourth coordinate is 1.
- copy over the fourth coordinate from input to output.

\texttt{rbt * rbt}
• let us look at the product of two such rigid body transforms.

\[
\begin{bmatrix}
  i & t_1 \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_1 & 0 \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  i & t_2 \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_2 & 0 \\
  0 & 1 \\
\end{bmatrix}
= \\
\begin{bmatrix}
  i & t_1 \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_{11} & r_{12} \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_2 & 0 \\
  0 & 1 \\
\end{bmatrix}
= \\
\begin{bmatrix}
  i & t_1 \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  i & r_{1t_2} \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_1 & 0 \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_2 & 0 \\
  0 & 1 \\
\end{bmatrix}
= \\
\begin{bmatrix}
  i & t_1 + r_{1t_2} \\
  0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  r_{1r_2} & 0 \\
  0 & 1 \\
\end{bmatrix}
\]

• 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.

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 $r^{-1}$.

code

• change `skyRbt` and `objectRbt[]` to be `RigTform` data type instead of `Matrix4`.

• in fact almost all of the C++ Matrix4's will get replaced!

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

more code

• in GLSL, you will still use its matrix data type.

• the only Matrix4s that will survive in your c++ code 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 provide a procedure `Matrix4 quatToMatrix(quat q)` which turns quat into a 4 by 4 rotation matrix.

• Then, the matrix for a rigid body transform can be computed as

```c++
matrix4 rigTFormToMatrix(const RigTform& rbt){
    matrix4 T = makeTranslation(rbt.getTranslation());
    matrix4 R = quatToMatrix(rbt.getRotation());
    return T * R;
}
```
• Thus, our drawing code starts with

```cpp
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)

### rbT Interpolation

• let's get back to discussing interpolation.

• given two frames, \( \vec{o}_0 = \vec{w}_t O_0 \) \( \vec{o}_1 = \vec{w}_t O_1 \)
  
  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 it using two `RigTForm` variables.

• interpolate between them by: linearly interpolating the two translation to get: \( T_\alpha \),

• slerp between the rotation quaternions to obtain the rotation \( R_\alpha \),

• set the interpolated RBT \( O_\alpha \) to be \( T_\alpha R_\alpha \).

• set \( \vec{o}_\alpha = \vec{w}_t O_\alpha \).

### 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)