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

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

- later we will add back in the translations.

terepolation setup

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

bad idea 1

- lin interp of matrices \( R_\alpha := (1 - \alpha) R_0 + (\alpha) R_1 \) and then set \( \vec{o}_\alpha^t := \vec{w} R_\alpha \).
- in the resulting interpolation, the basis vectors blend linearly from start to finish

\[
\vec{w} R_\alpha = \vec{w}[(1 - \alpha) R_0 + (\alpha) R_1] \\
= (1 - \alpha) \vec{w} R_0 + (\alpha) \vec{w} R_1 \\
= (1 - \alpha) \vec{o}_0^t + (\alpha) \vec{o}_1^t
\]

- the basis vectors are just linearly blended elementwise
- 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
- this turns out to give unnatural looking interpolants.
- the interpolation turns out to vary based on to choice of world frame
  - this bad property is called “non left invariance” (see fig)
- we would like an intrinsic geometric operation (describable independent of coordinates or world frame)
- note: for a 2 Euler dof (roll free) camera, this is a reasonable solution.

lets back up

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

uniqueness and cycles

- 1) a rotation of $-\theta$ degrees around $-\vec{k}$ will give us the same $Z$ and the same interpolation.
- 2) a rotation of $\theta + n2\pi$ degrees for any positive or negative integer $n$, around a fixed axis $\vec{k}$ gives us the same $Z$ but a different interpolation.
- a natural choice for the interpolation is to the value of $n$ so that that $|\theta + n2\pi|$ is minimal (ie. is a value in $[-\pi...\pi]$).
- also, if we want, we can, without changing the interpolation, choose the sign of the axis so that that $\theta + n2\pi$ is positive (ie. is a value in $[0...\pi]$).

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 $\vec{w}^t R_0 \rightarrow \vec{w}^t (R_1 R_0^{-1}) R_0$
- let us define the matrix $Z := (R_1 R_0^{-1})$
- so $Z$ must be the unique rotation matrix, such that “doing $Z$ wrt $\vec{w}$” exactly expresses $Z$.
- the $Z$ matrix is “built” using the (unit) “axis coordinates” $\hat{k}$ that are the world coordinates of $\vec{k}$.
- suppose we could compute a “power” operator $Z^\alpha$ that represents the powered rotation about $\hat{k}$.
- then as $\alpha$ goes from 0..1, the sequence $\vec{w}^t Z^\alpha R_0$, 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 $R_1 R_0^{-1}$, into its axis coordinates/angle form.
- main quaternion idea: keep track of the axis and angle at all times but in a way that allows our manipulations (composition and application to points)
- such a rep. 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.
- we will first study the quaternion representation for rotations, use it in an explicity RBT data structure, and use it to implement the arcball interface.
- at the end, we will come back to the interpolation details.

the representation

- a quaternion is 4 tuple with operations (to be defined)
- 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 represnted by a unit length coordinate 3-vector $\hat{k}$, and $\theta$ is any angle.
- to represent the rotation of $\theta$ degrees about $\hat{k}$, we will use the quaternion:

$$\begin{bmatrix} \cos(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2})\hat{k} \end{bmatrix}$$
• 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 $\hat{k}$ is the same rotation, and gives us the same quaternion. good.
• A rotation of $\theta + 4\pi$ degrees about an axis $\hat{k}$ is the same rotation and also gives us the same quaternion. good.
• A rotation of $\theta + 2\pi$ degrees about an axis $\hat{k}$, which in fact is the same rotation (as a linear transformation), gives us the negated quaternion
• so “antipodal” quaternions represent the same rotation transformation
  – we will soon think of one being a “short” $0 \leq |\theta| \leq \pi$ rotation, and the other being a “long” $\pi \leq |\theta| \leq 2\pi$ rotation.

extreme examples

• $\theta = 0$:
  \[
  \begin{bmatrix}
  1 \\
  0 
  \end{bmatrix}
  \]

• $\theta = 2\pi$:
  \[
  \begin{bmatrix}
  -1 \\
  0 
  \end{bmatrix}
  \]

• both represent the identity rotation transformation

• $\theta = \pi$
  \[
  \begin{bmatrix}
  0 \\
  \hat{k} 
  \end{bmatrix}
  \]

• $\theta = -\pi$
  \[
  \begin{bmatrix}
  0 \\
  -\hat{k} 
  \end{bmatrix}
  \]

• both represent the same flip rotation transformation.

unit norm quats $\mapsto$ rotations

• squared norm is sum of 4 squares.
• Any quaternion of the form
  \[
  \begin{bmatrix}
  \cos(\frac{\theta}{2}) \\
  \sin(\frac{\theta}{2})\hat{k} 
  \end{bmatrix}
  \]
  has a unit norm (ponder this)
• Conversely, as we will see later any unit norm quaternion can be interpreted as above with a $\hat{k}$ and $\theta$
• the interpretation is unique up to additions of $4\pi$, as well as simultaneous negation of $\hat{k}$ and $\theta$.
• using an appropriate multiple of $4\pi$, we can get a unique $\theta \in (-2\pi...2\pi]$.
• then using an appropriate negation of $\hat{k}$ and $\theta$ we can get a unique $\theta \in [0..2\pi]$.
• this might still be a “long” rotation. we will get back to this when we get back to interpolation.

Operations
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.

- magic trick number 1:
  - suppose we have the product of rotation matrices: \( R_3 = R_1R_2 \)
  - suppose we model our input rotations using unit quaternions \( q_1 \) and \( q_2 \). (there are two choices for each)
  - then the quaternion product \( q_1q_2 \) will be one of the two quaternions representing \( R_3 \).

**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
- so this represents the inverse rotation

**quat vector multiply setup**

- magic trick number 2
- start with arbitrary 3-coordinate vector \( c \), representing a point or vector.
- left multiply it by a 4 by 4 rotation matrix \( R \), to get

\[
\begin{bmatrix}
  c' \\
  0/1
\end{bmatrix} = R \begin{bmatrix}
  c \\
  0/1
\end{bmatrix}
\]

**quat vector multiply setup**

- let \( R \) be represented with the unit norm quaternion \( q \) (there are two choices)
- use cvec3 \( c \) to create the non unit norm quaternion

\[
\begin{bmatrix}
  0 \\
  c
\end{bmatrix}
\]

- perform the following triple quaternion multiplication:

\[
q \begin{bmatrix}
  0 \\
  c
\end{bmatrix} q^{-1}
\]

- tada: result is of form

\[
\begin{bmatrix}
  0 \\
  c'
\end{bmatrix}
\]

- and we might write this \( qc = c' \)
- in our Quat class, we will give you: quat * cvec3 = cvec3

**Rbt data structure**

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

\[
\begin{bmatrix}
  A \\
  r & t
\end{bmatrix} = \begin{bmatrix}
  i & t \\
  0 & 1
\end{bmatrix} \begin{bmatrix}
  r & 0 \\
  0 & 1
\end{bmatrix}
\]
• we can encode this information in the following data type

```cpp
class RigTForm{
    Cvec3 t;
    Quat q;
};
```

• this data will be always interpreted in the TR order above.

**rbt * Cvec4**

• you will write code for the product of a **RigTForm A** and a **Cvec4** c, (where the last entry is 0/1).

• only translate if the fourth coordinate is 1.
  
  – note: don’t use a hard equality constraint on reals.

• copy over the fourth coordinate from input to output.

**rbt * rbt**

• let us look at the product of two such rigid body transforms. (we can just apply matrix multiplication on the block form!)

\[
\begin{bmatrix}
i & t_1 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r_1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
i & t_2 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r_2 & 0 \\
0 & 1
\end{bmatrix} =
\begin{bmatrix}
i & t_3 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r_1 & r_1 t_2 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r_2 & 0 \\
0 & 1
\end{bmatrix} =
\begin{bmatrix}
i & t_1 + r_1 t_2 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r_1 r_2 & 0 \\
0 & 1
\end{bmatrix}
\]

• the result is a new rigid transform with translation \(t_1 + r_1 t_2\) and rotation \(r_1 r_2\).

  – use this to code up the \(*\) op.

**inv operator**

• likewise for inverse

\[
\begin{bmatrix}
i & t \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r & 0 \\
0 & 1
\end{bmatrix}^{-1} =
\begin{bmatrix}
r & 0 \\
0 & 1
\end{bmatrix}^{-1}
\begin{bmatrix}
i & t \\
0 & 1
\end{bmatrix}^{-1} =
\begin{bmatrix}
r^{-1} & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
i & -t \\
0 & 1
\end{bmatrix} =
\begin{bmatrix}
r^{-1} & -(r^{-1} t) \\
0 & 1
\end{bmatrix} =
\begin{bmatrix}
i & -(r^{-1} t) \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
r^{-1} & 0 \\
0 & 1
\end{bmatrix}
\]

• 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 \texttt{Quat makeXRotation(const double ang)}

\textbf{more code}

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

• the only \texttt{Matrix4s} that will survive in your c++ code are the \texttt{projMatrix}, \texttt{the MVM} and \texttt{the NMVM}, which get sent to your shaders.

• also, when we need to do object scaling, we cannot capture this in an \texttt{RigTform}, so this will also be a \texttt{Matrix4} used in creating the \texttt{MVM}.

• to communicate with the vertex shader using 4 by 4 matrices, we provid a procedure \texttt{Matrix4 quatToMatrix(quat q)} which turns \texttt{quat} into a 4 by 4 rotation matrix.

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

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

• Thus, our drawing code starts with

\begin{verbatim}
Matrix4 MVM = rigTFormToMatrix(inv(eyeRbt) * objRbt);
\\ can right multiply scales here
Matrix4 NMVM = normalMatrix(MVM);
sendModelViewNormalMatrix(curSS, MVM,NMVM);
\end{verbatim}

• we will not need any code that takes a \texttt{Matrix4} and converts it to a \texttt{Quat}.

• scale will still represented by a \texttt{Matrix4}. (more later)