track and arc Balls

- how should we link mouse motion to object rotation.
- can do better than our current setup.
- want the feeling of pushing a sphere around
- want path invariance
- failure demo

setup

- we are moving an object with respect to cube-eye $\vec{a}' = \vec{w}'(O)_T(E)_R$
- The user clicks on the screen and drags the mouse. We wish to interpret this user motion as some rotation $R$ that is applied to $\vec{o}'$ with respect to $\vec{a}'$.

mental model

- imagine a sphere of some chosen radius that is centered at $\hat{o}$, the origin of $\vec{o}'$.
- user clicks on the screen at some pixel $s_1$ over the sphere in the image
  - we interpret this as the user selecting some 3D point $\hat{p}_1$ on the sphere.
- the user then moves the mouse to some other pixel $s_2$ over the sphere,
  - we interpret as a second point $\hat{p}_2$ on the sphere.
- define the unit direction vectors $\hat{v}_1, \hat{v}_2 : \text{normalize}(\vec{p}_1 - \hat{o})$ and $\text{normalize}(\vec{p}_2 - \hat{o})$ respectively.
- Define the angle $\phi = \arccos(\hat{v}_1 \cdot \hat{v}_2)$
- define the axis $\vec{k} = \text{normalize}(\hat{v}_1 \times \hat{v}_2)$.

the balls

- trackball: $R$ is the rotation of $\phi$ degrees about the axis $\vec{k}$.
- arcball: $R$ is the rotation of $2\phi$ degrees about the axis $\vec{k}$.
- could be implemented with matrices or quaternions.
- arcball is very easy with quaternions
- rotation of $2\phi$ degrees about the axis $\vec{k}$ can be represented by the quaternion
  
  $\begin{bmatrix} \cos(\phi) \\ \sin(\phi) \vec{k} \end{bmatrix} = \begin{bmatrix} \hat{v}_1 \cdot \hat{v}_2 \\ \hat{v}_1 \times \hat{v}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \hat{v}_2 \end{bmatrix} \begin{bmatrix} 0 \\ -\hat{v}_1 \end{bmatrix}$

- where $\hat{k}, \hat{v}_1$ and $\hat{v}_2$ are the coordinate 3-vectors representing the vectors $\vec{k}, \vec{v}_1$ and $\vec{v}_2$ with respect to the frame $\vec{a}'$.
- start demo

Properties

- trackball feels like the user is simply grabbing a physical point on a sphere and dragging it around.
- but $s_1$ to $s_2$, followed by $s_2$ to $s_3$ is different from moving directly from $s_1$ to $s_3$
  - $\hat{p}_1$ will be rotated to $\hat{p}_3$, but the two results can differ by some “twist” about the axis $\hat{o} - \hat{p}_3$.
- arcball: the object appears to spin twice as fast as expected.
- but is path independent
path ind proof

- If we compose two arcball rotations, corresponding to motion from \( \tilde{p}_1 \) to \( \tilde{p}_2 \) followed by motion from \( \tilde{p}_2 \) to \( \tilde{p}_3 \),
  - we have \( \vec{t} = \vec{a}B \) (for some \( B \)).
- reading from right to left, we see that our transformations are \( \vec{a}B \rightarrow \vec{a}R_1B \rightarrow \vec{a}R_2R_1B \)
  - \( \vec{a}t \) doesn’t change since we are not changing the eye frame or the origin of the object frame.
- we get for \( R_2R_1 \):
  \[
  \begin{bmatrix}
    \hat{v}_2 \cdot \hat{v}_3 \\
    \hat{v}_2 \times \hat{v}_3 
  \end{bmatrix}
  \begin{bmatrix}
    \hat{v}_1 \cdot \hat{v}_2 \\
    \hat{v}_1 \times \hat{v}_2 
  \end{bmatrix}
  \]
  - which gives us
  \[
  \begin{bmatrix}
    0 \\
    \hat{v}_3 
  \end{bmatrix}
  \begin{bmatrix}
    0 \\
    -\hat{v}_2 
  \end{bmatrix}
  \begin{bmatrix}
    0 \\
    -\hat{v}_1 
  \end{bmatrix}
  =
  \begin{bmatrix}
    0 \\
    \hat{v}_3 
  \end{bmatrix}
  \begin{bmatrix}
    0 \\
    -\hat{v}_1 
  \end{bmatrix}
  =
  \begin{bmatrix}
    \hat{v}_1 \cdot \hat{v}_3 \\
    \hat{v}_1 \times \hat{v}_3 
  \end{bmatrix}
  \]
  - which is exactly what we would have gotten had we moved directly from \( \tilde{p}_1 \) to \( \tilde{p}_3 \).

Implementation

- Trackball and Arcball can be directly implemented using either 4 by 4 matrices or quaternions to represent the transformation \( R \).
  - we will use quaternions, since we already have them
- the resulting quaternion depends only on vectors \( \hat{v} \)
  - so origin of frame is irrelevant
- so we can work in eye coordinates instead of cube-eye

getting eye coordinates

- One slightly tricky part is computing the coordinates of the point on the sphere corresponding to a selected pixel
  - this is geometric ray tracing (this is essentially ray-tracing, which we will covered later)
- hack: work in “window coordinates”.
  - x-axis is the horizontal axis of the screen, the y-axis is the vertical axis of the screen, and the z-axis is coming out of the screen.
  - think of the sphere’s center as simply sitting on the screen.
- Given the \((x, y)\) window coordinates of click the z coordinate on the sphere can be solved using \((x - c_x)^2 + (y - c_y)^2 + (z - 0)^2 - r^2 = 0, \)
  - \([c_x, c_y, 0]^t \) are the window coordinates of the center of the sphere.
  - \( r \) is the radius of the sphere measured in pixels.
  - and then normalize to get \( \hat{v} \).
- if outside of the sphere, then clamp to its silhoutte. and then normalize.
  - this can be done by just normalizing \([x - c_x, y - c_y, 0]^t \).

calculation

- need the center of the sphere
- so we give you code that transforms eye coords to screen coords.
Cvec2 getScreenSpaceCoord(const Cvec3& p,
    const Matrix4& projection,
    double frustNear, double frustFovY,
    int screenWidth, int screenHeight)

- we draw the ball using object coordinates, so we need to calculate its size in eye/object coordinates
- so we provide you with

  double getScreenToEyeScale(double z, double frustFovY, int screenHeight)

- in the ball drawer, you right multiply a scale matrix to the MVM.

**translation**

- in translation, we interpret mouse displacement (measured in pixels) to object displacement.
- may as well use the same screenToEyeScale factor so the object moves with the mouse.
- once the object is moved, or we change the eye we need to recalculate the scale
  - wait for click up.

**moving skycam**

- we will not change our roll-free skycam interface.