CS380: Introduction to Computer Graphics
Quaternions & Track-/Arc-ball
Chapter 7 & 8

Min H. Kim
KAIST School of Computing

QUIZZES

Quaternion I

SUMMARY

Quaternion (rotation)

- We use the quaternion representation for interpolating *rotation* only (not translation).
- We cannot interpolate a rotation matrix *R* in $\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t R_\alpha$
- Interpolating the three scalar in the XYZ Euler angles is not a good solution for natural movement (only left invariant, i.e., if the object frame changes, the rotation changes).

$$\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t R_\alpha$$

Quaternion (rotation)

- The quaternion rotation itself allows us to interpolate the rotation angle (left and right invariant). $\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t (R_1 R_0^{-1})^\alpha R_0$
- Quaternion rotation of angle θ : $\begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{k}} \end{bmatrix}$
- Quaternion multiplication:

$$\begin{bmatrix} \omega_1 \\ \hat{\mathbf{c}}_1 \end{bmatrix}\begin{bmatrix} \omega_2 \\ \hat{\mathbf{c}}_2 \end{bmatrix} = \begin{bmatrix} (\omega_1\omega_2 - \hat{\mathbf{c}}_1 \cdot \hat{\mathbf{c}}_2) \\ (\omega_1\hat{\mathbf{c}}_2 + \omega_2\hat{\mathbf{c}}_1 + \hat{\mathbf{c}}_1 \times \hat{\mathbf{c}}_2) \end{bmatrix}$$

Quaternion (rotation)

- Unit quaternion multiplication

$$\begin{bmatrix} 0 \\ \hat{\mathbf{c}}_1 \end{bmatrix}\begin{bmatrix} 0 \\ \hat{\mathbf{c}}_2 \end{bmatrix} = \begin{bmatrix} -\hat{\mathbf{c}}_1 \cdot \hat{\mathbf{c}}_2 \\ \hat{\mathbf{c}}_1 \times \hat{\mathbf{c}}_2 \end{bmatrix}$$

$$\begin{bmatrix} \hat{\mathbf{k}}_1 \cdot \hat{\mathbf{k}}_2 \\ \hat{\mathbf{k}}_1 \times \hat{\mathbf{k}}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \hat{\mathbf{k}}_2 \end{bmatrix}\begin{bmatrix} 0 \\ -\hat{\mathbf{k}}_1 \end{bmatrix}$$

- Inverse quaternion

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

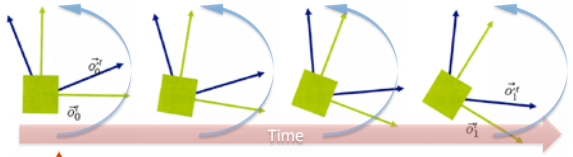
Quaternion (rotation)

- To interpolate between two frames related to world frame by R_0 and R_1

$\vec{\mathbf{o}}_0^t = \vec{\mathbf{w}}^t R_0$ *when* $\alpha = 0$

$\vec{\mathbf{o}}_1^t = \vec{\mathbf{w}}^t R_1$ *when* $\alpha = 1$

$$\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t (R_1 R_0^{-1})^\alpha R_0$$

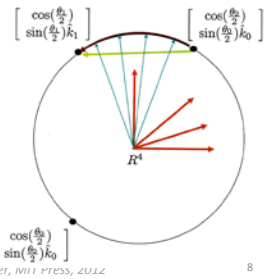


$$\left(cn\left(\begin{bmatrix} \cos\left(\frac{\theta_1}{2}\right) \\ \sin\left(\frac{\theta_1}{2}\right)\hat{\mathbf{k}}_1 \end{bmatrix}\begin{bmatrix} \cos\left(\frac{\theta_0}{2}\right) \\ \sin\left(\frac{\theta_0}{2}\right)\hat{\mathbf{k}}_0 \end{bmatrix}^{-1} \right) \right)^\alpha \begin{bmatrix}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

Slerping

- This is called *spherical linear interpolation* or just *slerping* since it happens to match moving on a great circle in $\mathbb{R}^4$

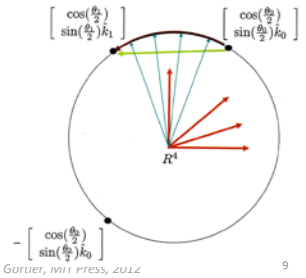
$$\frac{\sin[(1-\alpha)\Omega]}{\sin(\Omega)}\begin{bmatrix} \cos\left(\frac{\theta_0}{2}\right) \\ \sin\left(\frac{\theta_0}{2}\right)\hat{\mathbf{k}}_0 \end{bmatrix} + \frac{\sin[\alpha\Omega]}{\sin(\Omega)}\begin{bmatrix} \cos\left(\frac{\theta_1}{2}\right) \\ \sin\left(\frac{\theta_1}{2}\right)\hat{\mathbf{k}}_1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



Lerping

- An even easier hack is to do 4D lerp and renormalization (make vectors' length as 1.0)
- Both left and right invariant.
- More efficient approximation than slerp.
- Useful for blending *n* different rotations.

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



Chapter 7

QUATERNIONS II

Putting back the translation

- Lets now build a data structure to represent an RBT
- Recall: RBT data structure $A = TR$

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

- Our class

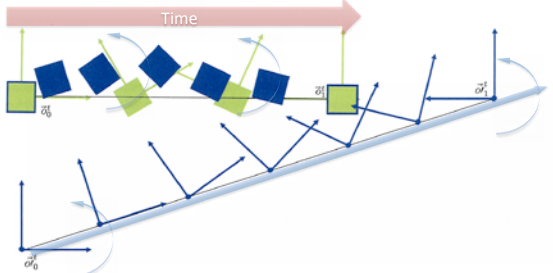
```
Class RigTForm{
    Cvec3 t;
    Quat r;
};
```


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 translations to get: T_α
- Slerp between the rotation quaternions to obtain the rotation R_α
- Set the interpolation RBT O_α to be $T_\alpha R_\alpha$
- Set $\vec{\mathbf{o}}_\alpha^t = \vec{\mathbf{w}}^t O_\alpha$

RBT Interpolation Behavior

- Origin of $\vec{\mathbf{o}}^t$ travels in a straight line with constant velocity,
- The vector basis of $\vec{\mathbf{o}}^t$ rotates with constant angular velocity about a fixed axis.
- Physically natural if origin is at center of mass



RBT Interpolation Behavior

- Even though the quaternion rotation is left and right invariant, the quaternion rotation + object translation is left invariant.
- The translation of the origin plays special role.
- If we use different object frames for same geometry, we get different interpolations
  - Not right invariant

Code (assignment)

- Change skyRbt and objectRbt[] to be RigTform data type instead of Matrix4
- In fact, almost all of the C++ Matrix4s 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.

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

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

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

$$\begin{bmatrix} i & t_1 + r_1 t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 r_2 & 0 \\ 0 & 1 \end{bmatrix} \quad A = TR$$

RBT * RBT

- 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 * operator.
  - Mind the Cvec3s (the t's) and Cvec4s (needed for q*v).

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

Inverse operator

- Likewise for inverse

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

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

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

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

$A = TR$

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 an 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 quaternions into a 4-by-4 rotation matrix.

More code

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

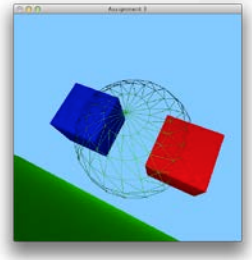
```
Matrix4 rigTFormToMatrix(const RigTform& rbt){
    matrix4 T = makeTranslation(rbt.getTranslation());
    matrix4 R = quatToMatrix(rbt.getRotation());
    return T * R;
}
```

- 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);
```

More code

- 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)



Chapter 8

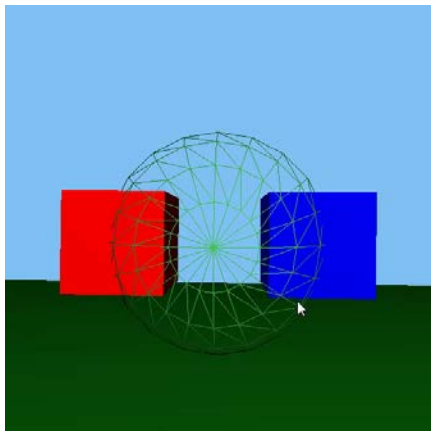
BALLS: TRACK AND ARC

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 (trackball)
- Want *path invariance* (arcball)

- Reminders:
  - Affine transform: $A_{affine} = TL$
  - Rigid body transform: $A_{RBT} = TR$

Track and arc balls

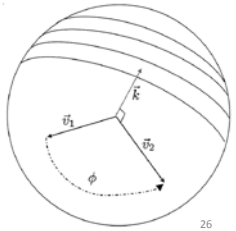


Setup

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

Mental model

- Imagine a sphere of some chosen radius that is centered at $\tilde{o}$, the origin of $\vec{\mathbf{o}}^t$.
- 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 $\tilde{p}_1$ on the sphere.
- The user then moves the mouse to some other pixel s_2 over the sphere,
  - We interpret as a second point $\tilde{p}_2$ on the sphere



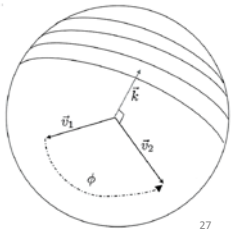
Mental model

- Define the unit direction vectors $\vec{v}_1, \vec{v}_2$: normalize $(\tilde{p}_1 - \tilde{o})$ and normalize $(\tilde{p}_2 - \tilde{o})$ respectively.
- Define the angle

$$\phi = \arccos(\vec{v}_1 \cdot \vec{v}_2)$$

- Define the axis

$$\vec{k} = normalize(\vec{v}_1 \times \vec{v}_2)$$



The balls

- Trackball: M is the rotation of ϕ degrees about the axis $\vec{k}$.
- Arcball: M is the rotation of 2ϕ degrees about the axis $\vec{k}$.
- Could be implemented with matrices or quaternions.
- Arcball is very easy with quaternions

The balls

- Rotation of 2ϕ degrees about the axis $\vec{k}$ can be represented by the quaternion.

$$\begin{bmatrix} \cos(\phi) \\ \sin(\phi)\hat{\mathbf{k}} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_2 \\ \hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \hat{\mathbf{v}}_2 \end{bmatrix} \begin{bmatrix} 0 \\ -\hat{\mathbf{v}}_1 \end{bmatrix}$$

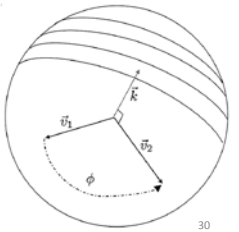
  - Where $\hat{\mathbf{k}}, \hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2$ are the coordinates 3-vectors representing the vectors $\vec{k}, \vec{v}_1, \vec{v}_2$ with respect to the frame $\vec{\mathbf{a}}^t$.

(Recap) quaternion multiplication:

$$\begin{bmatrix} \omega_1 \\ \hat{\mathbf{c}}_1 \end{bmatrix} \begin{bmatrix} \omega_2 \\ \hat{\mathbf{c}}_2 \end{bmatrix} = \begin{bmatrix} (\omega_1\omega_2 - \hat{\mathbf{c}}_1 \cdot \hat{\mathbf{c}}_2) \\ (\omega_1\hat{\mathbf{c}}_2 + \omega_2\hat{\mathbf{c}}_1 + \hat{\mathbf{c}}_1 \times \hat{\mathbf{c}}_2) \end{bmatrix}$$

Properties

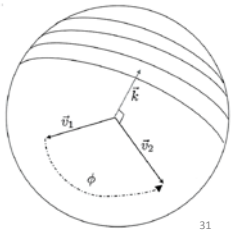
- Trackball feels like the user is simply grabbing a physical point on a sphere and dragging it around.
- But s_1 to s_2 , followed by s_2 to s_3 is different from moving directly from s_1 to s_3
  - $\tilde{p}_1$ will be rotated to $\tilde{p}_3$, but the results can differ by some 'twist' about the axis $\tilde{o} - \tilde{p}_3$.
  - This path dependence also exists in our simple rotation interface.



Properties

- Arcball: the object appears to spin twice as fast as expected.
- But <u>it is path independent</u>.

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



(Proof) Path independent

- 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 get

$$\begin{bmatrix} \hat{\mathbf{v}}_2 \cdot \hat{\mathbf{v}}_3 \\ \hat{\mathbf{v}}_2 \times \hat{\mathbf{v}}_3 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_2 \\ \hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2 \end{bmatrix}$$

  - Read right to left, global in the unchanging $\vec{\mathbf{a}}^t$ frame

$$\begin{bmatrix} 0 \\ \hat{\mathbf{v}}_3 \end{bmatrix} \begin{bmatrix} 0 \\ -\hat{\mathbf{v}}_2 \end{bmatrix} \begin{bmatrix} 0 \\ \hat{\mathbf{v}}_2 \end{bmatrix} \begin{bmatrix} 0 \\ -\hat{\mathbf{v}}_1 \end{bmatrix} = \begin{bmatrix} 0 \\ \hat{\mathbf{v}}_3 \end{bmatrix} \begin{bmatrix} 0 \\ -\hat{\mathbf{v}}_1 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_3 \\ \hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_3 \end{bmatrix}$$

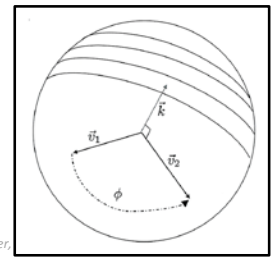
- which is exactly what we would have gotten had we moved directly from $\tilde{p}_1$ to $\tilde{p}_3$.

Implementation

- Trackball and Arcball can be directly implemented using either 4-by-4 matrices or quaternions to represent the transformation *M*.
  - We will use quaternions, since we already have them.
- The resulting quaternion depends only on vector $\hat{\mathbf{v}}$
  - So origin of frame is irrelevant
- 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 cover 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.

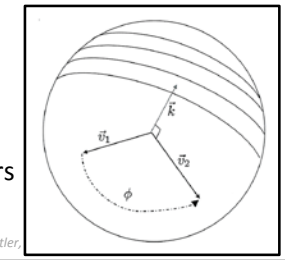


Getting eye coordinates

- Given the (x,y) window coordinates of click, the z coordinate on the sphere can be solved using

$$(x-c_x)^2+(y-c_y)^2+(z-0)^2=r^2$$

  - $[c_x, c_y, 0]^t$ are the window coordinates (in pixels) of the center of the sphere
  - *r* is the radius of the sphere measured in pixels
  - if outside of the sphere, then clamp to its boundary
  - All we need is normalized $\hat{\mathbf{v}}$, so just normalized such vectors



Calculation

- Need the center of the sphere
- So we give you code that transforms eye coordinates to screen coordinates

```
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.