CSE 528: Computer Graphics

Trackball Navigation

Klaus Mueller

Computer Science Department

Stony Brook University

Some material from Han-Wei Shen, Ohio State University and Steve Rottenberg, UCSD

Rotation Matrices Revisited

To build a general rotation matrix, we just multiply a sequence of rotation matrices together, for example:

$$\mathbf{R}_x \cdot \mathbf{R}_y \cdot \mathbf{R}_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & s_x \\ 0 & -s_x & c_x \end{bmatrix} \cdot \begin{bmatrix} c_y & 0 & -s_y \\ 0 & 1 & 0 \\ s_y & 0 & c_y \end{bmatrix} \cdot \begin{bmatrix} c_z & s_z & 0 \\ -s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix}$$

This matrix multiplication is not commutative

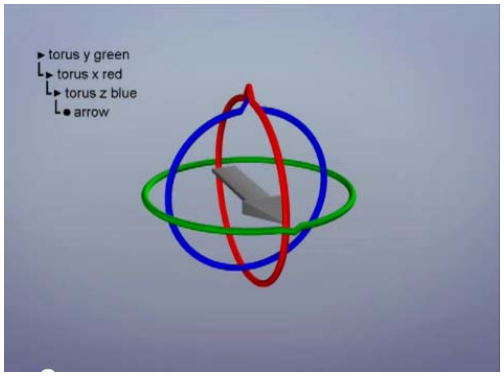
- the order of operations is important
- the multiplication is also a hierarchy: R_x [R_y [R_z object]]]
- this means R_x affects both R_y and R_z, and R_y affects R_z
- typically we rotate once per axis in some order

The rotation angles about x, y, z are also called *Euler Angles*

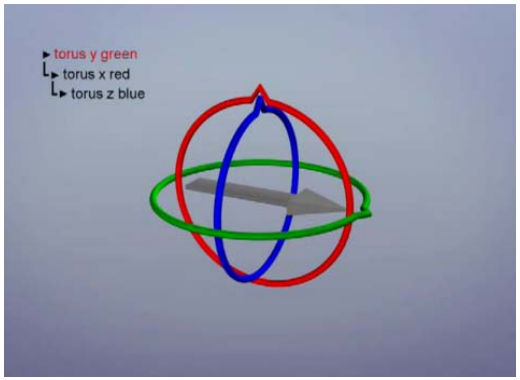
Gimbal Lock

The hierarchical rotation order causes problems

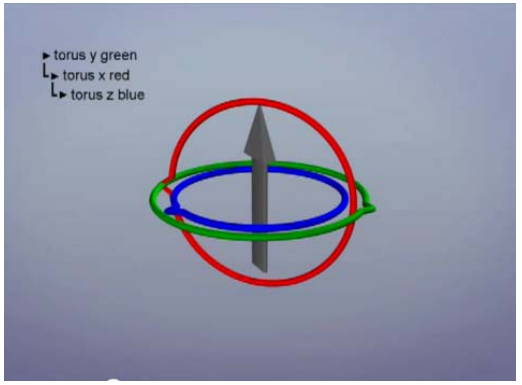
- consider R_z [R_x [R_y arrow]]] with 3 corresponding gimbals (axes) x, y, z



starting position

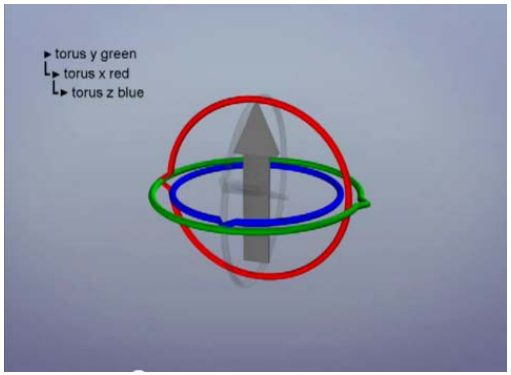


R_y rotates axis x and z as well



R_x rotates axis z into the already evaluated axis y → gimbal lock

R_z can no longer tilt arrow up and down → loss of one degree of freedom



see also
http://www.youtube.com/watch?v=zc8b2Jo7mno
and
http://www.anticz.com/eularqua.htm

Solution: Quaternions

Invented by W.R. Hamilton in1843 as an extension to the complex numbers

Used in computer graphics since 1985

Quaternions:

- provide an alternative method to specify rotations
- are most useful to us as a means of representing orientations
- can avoid the gimbal lock problem
- allow unique, smooth and continuous rotation interpolations

A quaternion has 4 components $\mathbf{q} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}$

- they are often written as the combination of a scalar value s and a vector value v

$$\mathbf{q} = \langle s, \mathbf{v} \rangle \quad \text{where} \quad \begin{aligned} s &= q_0 \\ \mathbf{v} &= \begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix} \end{aligned}$$

Quaternions (Imaginary Space)

Quaternions are actually an extension to complex numbers

- of the 4 components, one is a ‘real’ scalar number, and the other 3 form a vector in imaginary ijk space!

$$\mathbf{q} = q_0 + iq_1 + jq_2 + kq_3$$

$$i^2 = j^2 = k^2 = ijk = -1$$

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

Basic Operations

Conjugate: $\mathbf{q}^* = \langle -s, \mathbf{v} \rangle$

Norm: $q \cdot q^* = q_1^* \cdot q_1 + q_2^* \cdot q_2 + q_3^* \cdot q_3 + q_4^* \cdot q_4$

Inverse: $q^{-1} = (1/\text{Norm}(q)) \cdot q^*$

Identity i = (**0**,1)

Unit Quaternions and Rotations

For convenience, we will use only unit length quaternions, as they will be sufficient for our purposes and make things a little easier

$$|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = 1$$

A quaternion can represent a rotation by an angle θ around a unit axis **a**:

$$\mathbf{q} = \left[\cos\frac{\theta}{2} \quad a_x \sin\frac{\theta}{2} \quad a_y \sin\frac{\theta}{2} \quad a_z \sin\frac{\theta}{2} \right]$$

or

$$\mathbf{q} = \left\langle \cos\frac{\theta}{2}, \mathbf{a}\sin\frac{\theta}{2} \right\rangle$$

- if **a** is unit length, then **q** will be also

Quaternions as Rotations

$$|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

$$= \sqrt{\cos^2\frac{\theta}{2} + a_x^2\sin^2\frac{\theta}{2} + a_y^2\sin^2\frac{\theta}{2} + a_z^2\sin^2\frac{\theta}{2}} = \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}\left(a_x^2 + a_y^2 + a_z^2\right)}$$

$$= \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}|\mathbf{a}|^2} = \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}} = \sqrt{1} = 1$$

To convert a quaternion to a rotation matrix:

$$\begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_0q_3 & 2q_1q_3-2q_0q_2 \\ 2q_1q_2-2q_0q_3 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_0q_1 \\ 2q_1q_3+2q_0q_2 & 2q_2q_3-2q_0q_1 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

There is also a method to convert the other way

Quaternions as Rotations

Concatenation is easy – just multiply all the quaternions q_1, q_2, q_3, …. together

$$q_3 \cdot q_2 \cdot q_1 \cdot P \cdot q_1^{-1} \cdot q_2^{-1} \cdot q_3^{-1}$$

There is a one-to-one mapping between a quaternion rotation and 4x4 rotation matrix

From the vector-angle form:

- rotate about the unit vector u by angle θ:

  b = (cos(θ/2), u sin(θ/2))

A point p in 3D space is represented by the quaternion
P = (0, p)

The rotated point p' is represented by the quaternion
P' = b * P * b-1

Quaternion Dot Products

The dot product of two quaternions works in the same way as the dot product of two vectors:

$$\mathbf{p} \cdot \mathbf{q} = p_0 q_0 + p_1 q_1 + p_2 q_2 + p_3 q_3 = \left|\mathbf{p}\right|\left|\mathbf{q}\right| \cos\varphi$$

The angle between two quaternions in 4D space is half the angle one would need to rotate from one orientation to the other in 3D space

Quaternion Multiplication

We can perform multiplication on quaternions if we expand them into their complex number form

$$\mathbf{q} = q_0 + iq_1 + jq_2 + kq_3$$

If **q** represents a rotation and **q'** represents a rotation, then **qq'** represents **q** rotated by **q'**

This follows very similar rules as matrix multiplication (I.e., non-commutative)

$$\mathbf{qq}' = \left(q_0 + iq_1 + jq_2 + kq_3\right)\left(q_0' + iq_1' + jq_2' + kq_3'\right)$$
$$= \left\langle ss' - \mathbf{v}\cdot\mathbf{v}', s\mathbf{v}' + s'\mathbf{v} + \mathbf{v}\times\mathbf{v}\right\rangle$$

Two unit quaternions multiplied together will result in another unit quaternion

Linear Interpolation

If we want to do a linear interpolation between two points **a** and **b** in normal space

$$\text{Lerp}(t,\mathbf{a},\mathbf{b}) = (1-t)\mathbf{a} + (t)\mathbf{b}$$

where t ranges from 0 to 1

Note that the Lerp operation can be thought of as a weighted average (convex)

- we could also write it in it's additive blend form:

$$\text{Lerp}(t,\mathbf{a},\mathbf{b}) = \mathbf{a} + t(\mathbf{b}-\mathbf{a})$$

Spherical Linear Interpolation

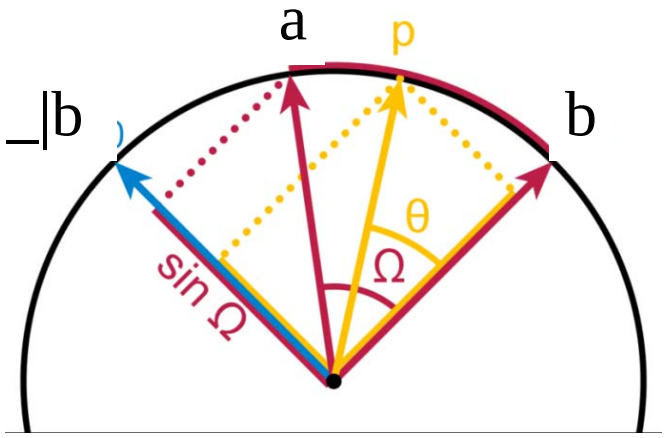
If we want to interpolate between two points on a sphere (or hypersphere), we don't just want to Lerp between them

- instead, we will travel across the surface of the sphere by following a 'great arc'

We define the spherical linear interpolation of two unit vectors as:

$$Slerp(t,\mathbf{a},\mathbf{b}) = \frac{\sin\left((1-t)\theta\right)}{\sin\theta}\mathbf{a} + \frac{\sin\left(t\theta\right)}{\sin\theta}\mathbf{b}$$

$$where: \theta = \cos^{-1}\left(\mathbf{a}\cdot\mathbf{b}\right)$$



Note, **a** and **b** can also be quaternions

- then slerp interpolation will ensure constant angular speed

Quaternion Interpolation

Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space

What is the difference between:

$$\text{Slerp}(t,\mathbf{a},\mathbf{b}) \quad \text{and} \quad \text{Slerp}(t,-\mathbf{a},\mathbf{b}) \ ?$$

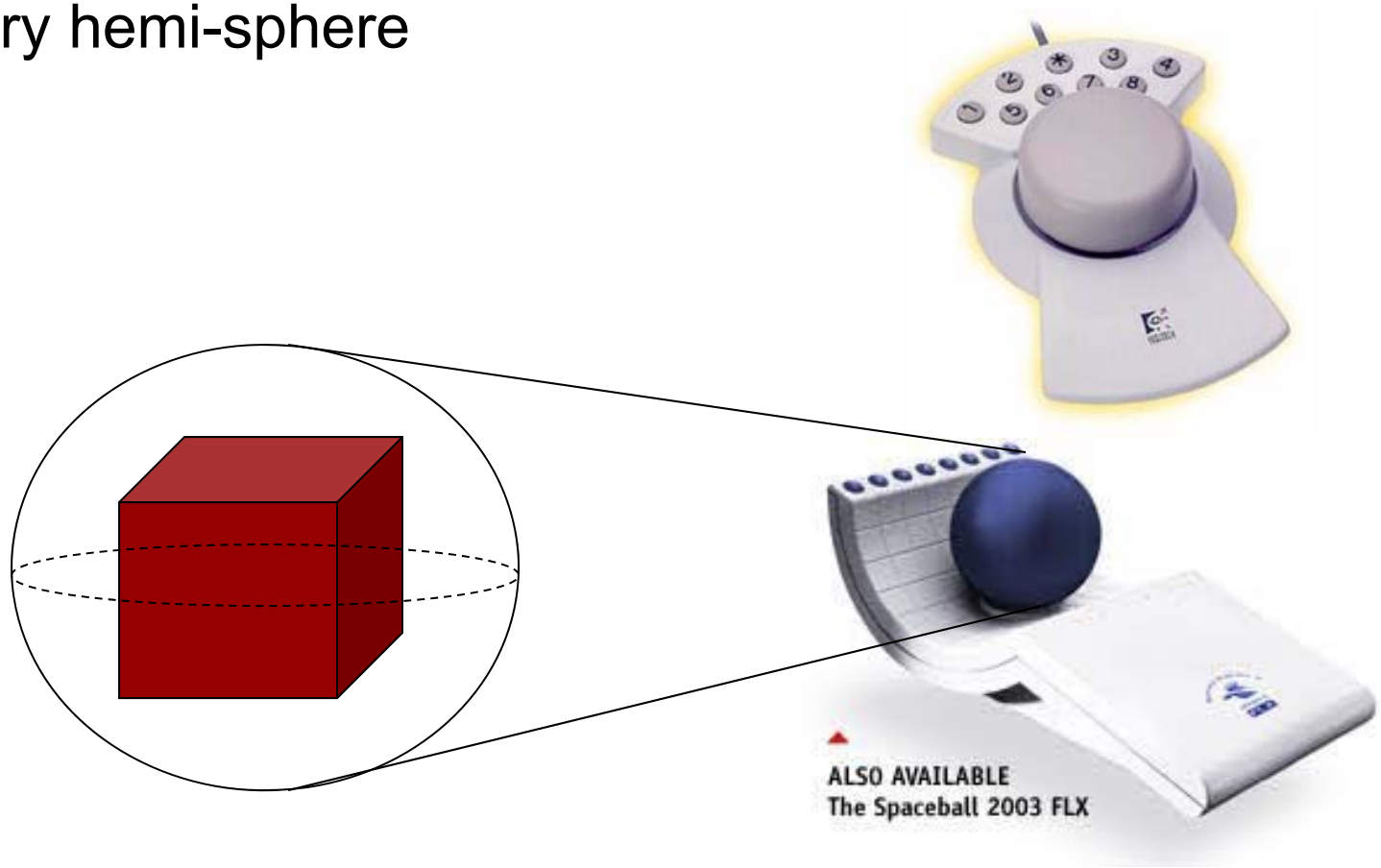
One of these will travel less than 90 degrees while the other will travel more than 90 degrees across the sphere

This corresponds to rotating the ‘short way’ or the ‘long way’

Usually, we want to take the short way, so we negate one of them if their dot product is < 0

3D Rotations with Trackball

Imagine the objects are rotated along with a imaginary hemi-sphere



Virtual Trackball

Allow the user to define 3D *rotation* using mouse click in 2D windows

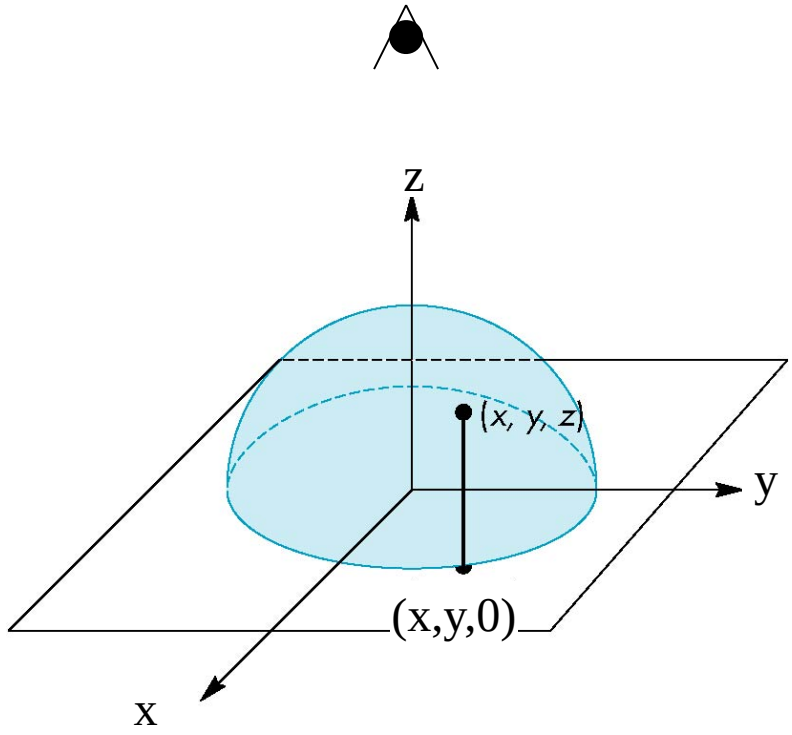
Work similarly like the hardware trackball devices

Virtual Trackball

Superimpose a hemi-sphere onto the viewport

This hemi-sphere is projected to a circle inscribed to the viewport

The mouse position is projected orthographically to this hemi-sphere



Virtual Trackball

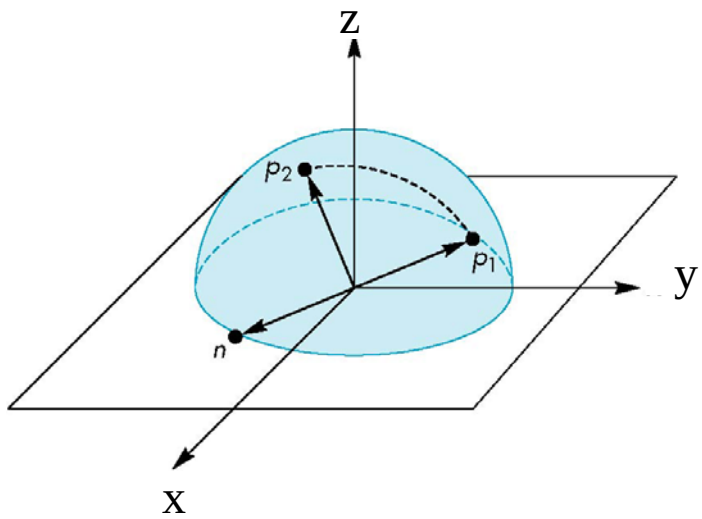
Keep track of the previous mouse position and the current position

Calculate their projection positions p_1 and p_2 to the virtual hemi-sphere

We then rotate the sphere from p_1 to p_2 by finding the proper rotation axis and angle

This rotation (in eye space!) is then applied to the object (call the rotation before you define the camera with gluLookAt())

You should also remember to accumulate the current rotation to the previous modelview matrix



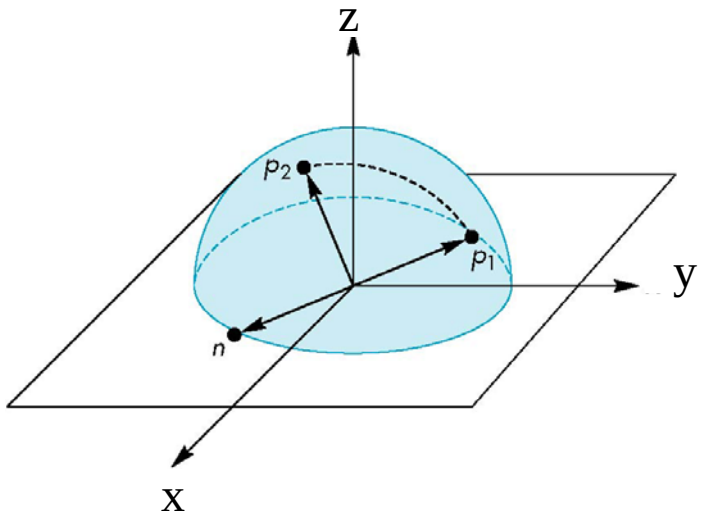
Virtual Trackball

The axis of rotation is given by the normal to the plane determined by the origin, $\mathbf{p}_1$, and $\mathbf{p}_2$

$$\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_1$$

The angle between $\mathbf{p}_1$ and $\mathbf{p}_2$ is given by

$$|\sin\theta| = \frac{|\mathbf{n}|}{|\mathbf{p}_1||\mathbf{p}_2|}$$



Virtual Trackball

How to calculate p_1 and p_2?

Assuming the mouse position is (x,y), then the sphere point P also has x and y coordinates equal to x and y

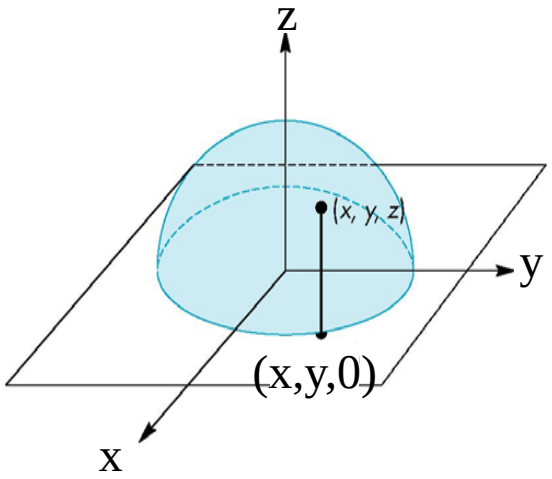
Assume the radius of the hemi-sphere is 1. So the z coordinate of P is

$$\sqrt{1 - x^2 - y^2}$$

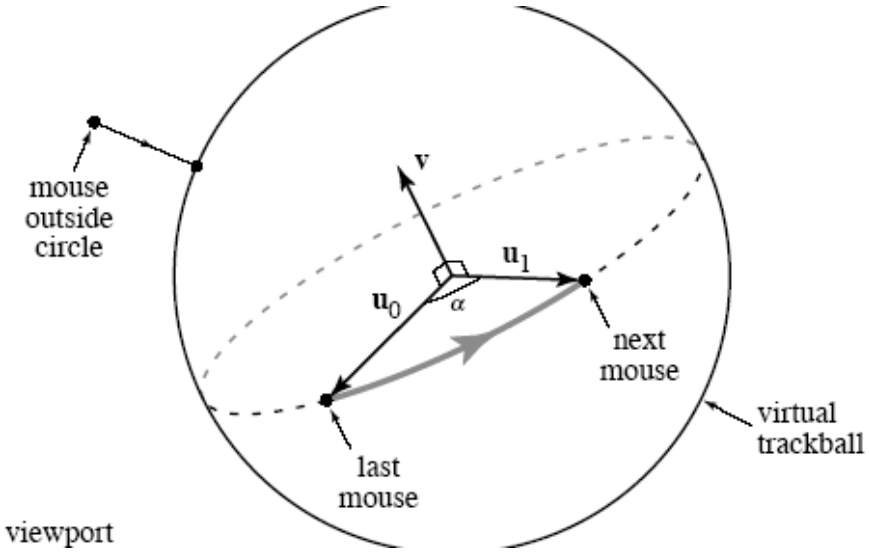
Note: normalize viewport y extend

to -1 to 1

If a point is outside the circle, project it to the nearest point on the circle (set z to 0 and renormalize (x,y))



Trackball Visualization



Using the quaternion representation at some point in the process will enable convenient specification of the rotation matrix, apart from the other advantages

- could keep internal quaternion representation throughout the navigation and only convert to update the Modelview Matrix

Example

Example from Ed Angel's OpenGL Primer

In this example, the virtual trackball is used to rotate a color cube

The code for the colorcube function is omitted

I will not cover the following code, but I am sure you will find it useful

Initialization

```
#define bool int  /* if system does not support
                     bool type */
#define false 0
#define true 1
#define M_PI 3.14159 /* if not in math.h */

int    winWidth, winHeight;

float angle = 0.0, axis[3], trans[3];

bool   trackingMouse = false;
bool   redrawContinue = false;
bool    trackballMove = false;

float lastPos[3] = {0.0, 0.0, 0.0};
int curx, cury;
int startX, startY;
```

The Projection Step

```
voidtrackball_ptov(int x, int y, int width, int height, float v[3])
{
    float d, a;
    /* project x,y onto a hemisphere centered within width, height ,
   note z is up here*/
    v[0] = (2.0*x - width) / width;
    v[1] = (height - 2.0F*y) / height;
    d = sqrt(v[0]*v[0] + v[1]*v[1]);
    v[2] = cos((M_PI/2.0) * ((d < 1.0) ? d : 1.0));
    a = 1.0 / sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    v[0] *= a;    v[1] *= a;    v[2] *= a;
}
```

glutMotionFunc (1)

```
Void mouseMotion(int x, int y)
{
     float curPos[3],
     dx, dy, dz;
     /* compute position on hemisphere */
     trackball_ptov(x, y, winWidth, winHeight, curPos);
     if(trackingMouse)
    {
        /* compute the change in position
              on the hemisphere */
        dx = curPos[0] - lastPos[0];
        dy = curPos[1] - lastPos[1];
        dz = curPos[2] - lastPos[2];
```

glutMotionFunc (2)

```
if (dx || dy || dz)
 {
  /* compute theta and cross product */
    angle = 90.0 * sqrt(dx*dx + dy*dy + dz*dz);
    axis[0] = lastPos[1]*curPos[2] –
         lastPos[2]*curPos[1];
    axis[1] = lastPos[2]*curPos[0] –
         lastPos[0]*curPos[2];
    axis[2] = lastPos[0]*curPos[1] –
         lastPos[1]*curPos[0];
  /* update position */
    lastPos[0] = curPos[0];
    lastPos[1] = curPos[1];
    lastPos[2] = curPos[2];
  }
 }
 glutPostRedisplay();}
```

Idle and Display Callbacks

```
void spinCube()
{
   if (redrawContinue) glutPostRedisplay();
}

void display()
{    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
 if (trackballMove)
 {
    glRotatef(angle, axis[0], axis[1], axis[2]);
 }
 colorcube();
 glutSwapBuffers();
}
```

Mouse Callback

```
void mouseButton(int button, int state, int x, int y)
{
 if(button==GLUT_RIGHT_BUTTON) exit(0);

/* holding down left button
      allows user to rotate cube */
 if(button==GLUT_LEFT_BUTTON) switch(state)
 {
    case GLUT_DOWN:
        y=winHeight-y;
        startMotion( x,y);
        break;
    case GLUT_UP:
        stopMotion( x,y);
        break;
 }
}
```

Start Function

```
void startMotion(int x, int y)
{
  trackingMouse = true;
  redrawContinue = false;
  startX = x;
  startY = y;
  curx = x;
  cury = y;
  trackball_ptov(x, y, winWidth, winHeight, lastPos);
  trackballMove=true;
}
```

Stop Function

```
void stopMotion(int x, int y)
{
    trackingMouse = false;
  /* check if position has changed */
    if (startX != x || startY != y)
      redrawContinue = true;
    else
    {
      angle = 0.0;
      redrawContinue = false;
      trackballMove = false;
    }
}
```