Virtual Trackball

Objectives

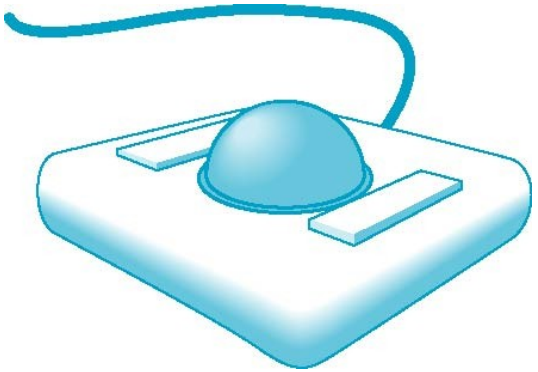
- This is an optional lecture that
  - Introduces the use of graphical (virtual) devices that can be created using OpenGL
  - Makes use of transformations
  - Leads to reusable code that will be helpful later

Interfaces to 3D applications

- Use areas of the screen
- Rotation(left mouse button, only two axes)
  - motion to the right or left will cause rotation about the x-axis
  - motion to the up or down will cause rotation about the x-axis
  - The distance from the center can control the speed of rotation
- Translation
  - right mouse button: **translate the object right to left and up to down**
  - middle mouse button: **translation in the z-direction**

Physical Trackball

- The trackball is an “upside down” mouse

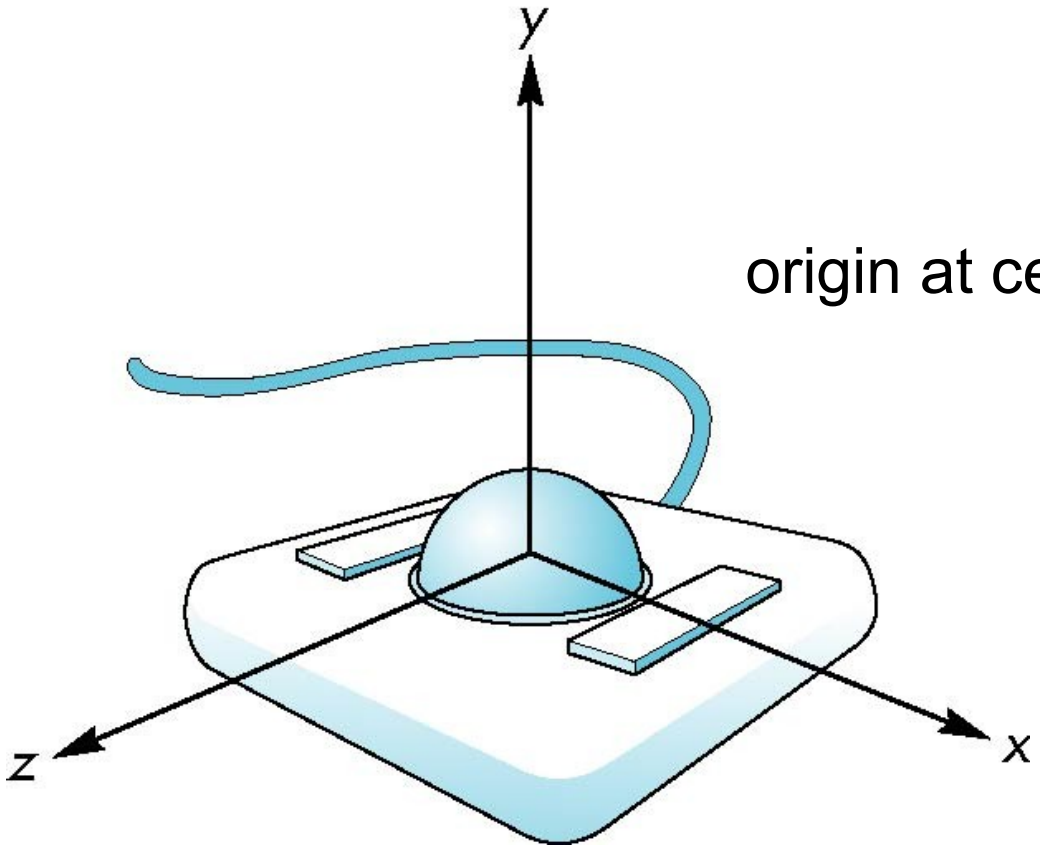


- If there is little friction between the ball and the rollers, we can give the ball a push and it will keep rolling yielding continuous changes
- Two possible modes of operation
  - Continuous pushing or tracking hand motion
  - Spinning

A Trackball from a Mouse

- Problem: we want to get the two behavior modes from a mouse
- We would also like the mouse to emulate a frictionless (ideal) trackball
- Solve in two steps
  - Map trackball position to mouse position
  - Use GLUT to obtain the proper modes

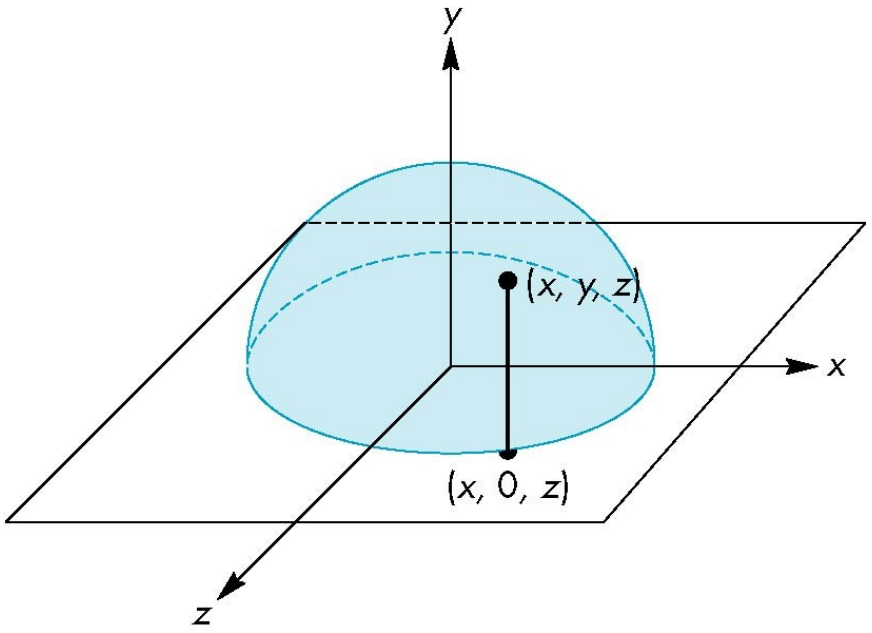
Trackball Frame



origin at center of ball

Projection of Trackball Position

- We can relate position on trackball to position on a normalized mouse pad by projecting orthogonally onto pad



Reversing Projection

- Because both the pad and the upper hemisphere of the ball are two-dimensional surfaces, we can reverse the projection
- A point (x,z) on the mouse pad corresponds to the point (x,y,z) on the upper hemisphere where

$$y = \sqrt{r^2 - x^2 - z^2} \qquad \text{if } x^2+z^2 <= r^2$$

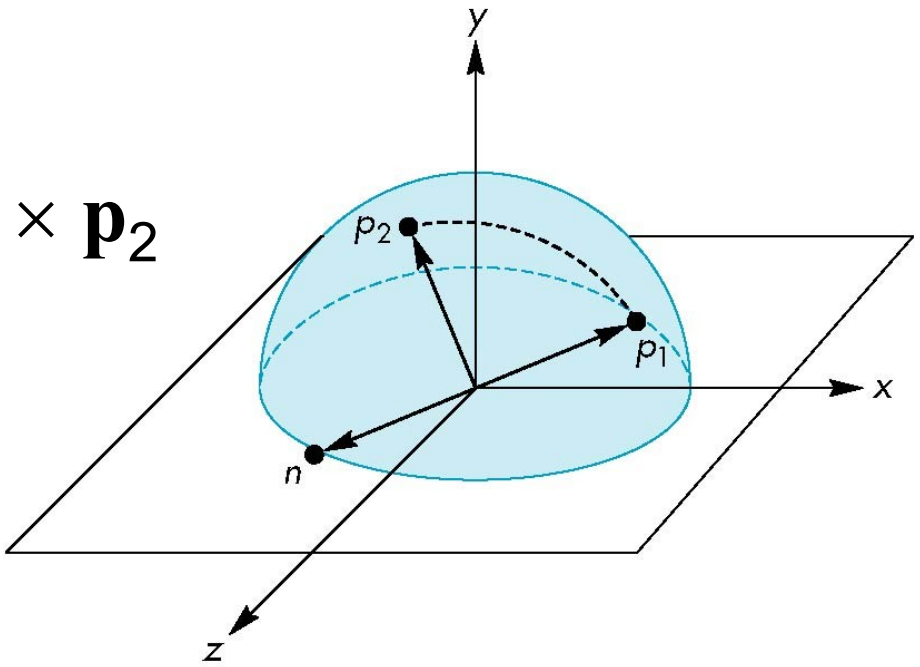
Computing Rotations

- Suppose that we have two points that were obtained from the mouse.
- We can project them up to the hemisphere to points $\mathbf{p}_1$ and $\mathbf{p}_2$
- These points determine a great circle on the sphere
- We can rotate from $\mathbf{p}_1$ to $\mathbf{p}_2$ by finding the proper axis of rotation and the angle between the points

Using the cross product

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

$$\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_2$$



Obtaining the angle

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

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

- If we move the mouse slowly or sample its position frequently, then θ will be small and we can use the approximation

$$\sin\theta \approx \theta$$

Implementing with GLUT

- We will use the idle, motion, and mouse callbacks to implement the virtual trackball
- Define actions in terms of three booleans
  - **`trackingMouse`**: if true, update trackball position
  - **`trackballMove`**: if true, update rotation matrix
  - **`redrawContinue`**: if true, idle function posts a redisplay

Example

- In this example, we use the virtual trackball to rotate the color cube we modeled earlier
- The code for the colorcube function is omitted because it is unchanged from the earlier examples

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

```
Void trackball_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;
    }
}
```

Smooth Rotation

- From a practical standpoint, we are often want to use transformations to move and reorient an object smoothly
  - Problem: find a sequence of model-view matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_n$ so that when they are applied successively to one or more objects we see a smooth transition
- For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere
  - Find the axis of rotation and angle
  - Virtual trackball (see text)

Incremental Rotation

- Consider the two approaches
  - For a sequence of rotation matrices $\mathbf{R}_0, \mathbf{R}_1, \ldots\ldots, \mathbf{R}_n$, find the Euler angles for each and use $\mathbf{R}_i = \mathbf{R}_{iz}\,\mathbf{R}_{iy}\,\mathbf{R}_{ix}$
    - Not very efficient
  - Use the final positions to determine the axis and angle of rotation, then increment only the angle
- Quaternions can be more efficient than either

Incremental Rotation

```
for(i=0, i<imax; i++)
{
    glRotatef(delta_theta, dx, dy, dz) ;
    draw_object() ;
}
```

do better if we compute the rotation matrix once and reuse it

```
GLfloat m[16];
glRotatef(dx, dy, dz, delta_theta) ;
glGetFloatv(GL_MODELVIEW_MATRIX, m);
```

```
For(i = 0, i<imax; i++)
{
    glMultMatrixf(m);
    draw_object() ;
}
```

$$\sin\theta \approx \theta,$$

$$\cos\theta \approx 1.$$

$$\mathbf{R} = \mathbf{R}_z(\psi)\mathbf{R}_y(\phi)\mathbf{R}_x(\theta)$$

$$= \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\approx \begin{bmatrix} 1 & -\psi & \phi & 0 \\ \psi & 1 & -\theta & 0 \\ -\phi & \theta & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Quaternions

- Because the rotations are on the surface of a sphere, quaternions provide an interesting and more efficient way to implement the trackball
- See code in some of the standard demos included with Mesa

Quaternions

- Extension of imaginary numbers from two to three dimensions
- Requires one real and three imaginary components $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$$

- Quaternions can express rotations on sphere smoothly and efficiently. Process:
  - Model-view matrix $\rightarrow$ quaternion
  - Carry out operations with quaternions
  - Quaternion $\rightarrow$ Model-view matrix