Virtual Trackball

Ed Angel

Professor of Computer Science, Electrical and Computer Engineering, and Media Arts

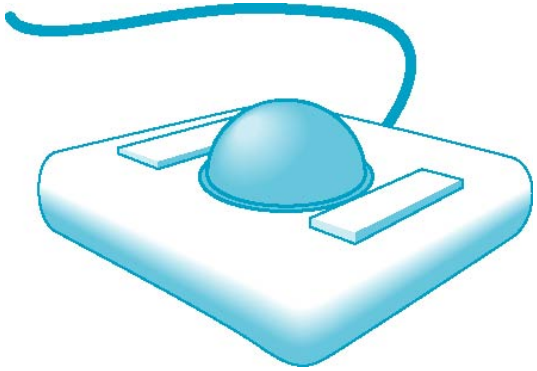
University of New Mexico

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

Physical Trackball

- The trackball is an “upside down” mouse

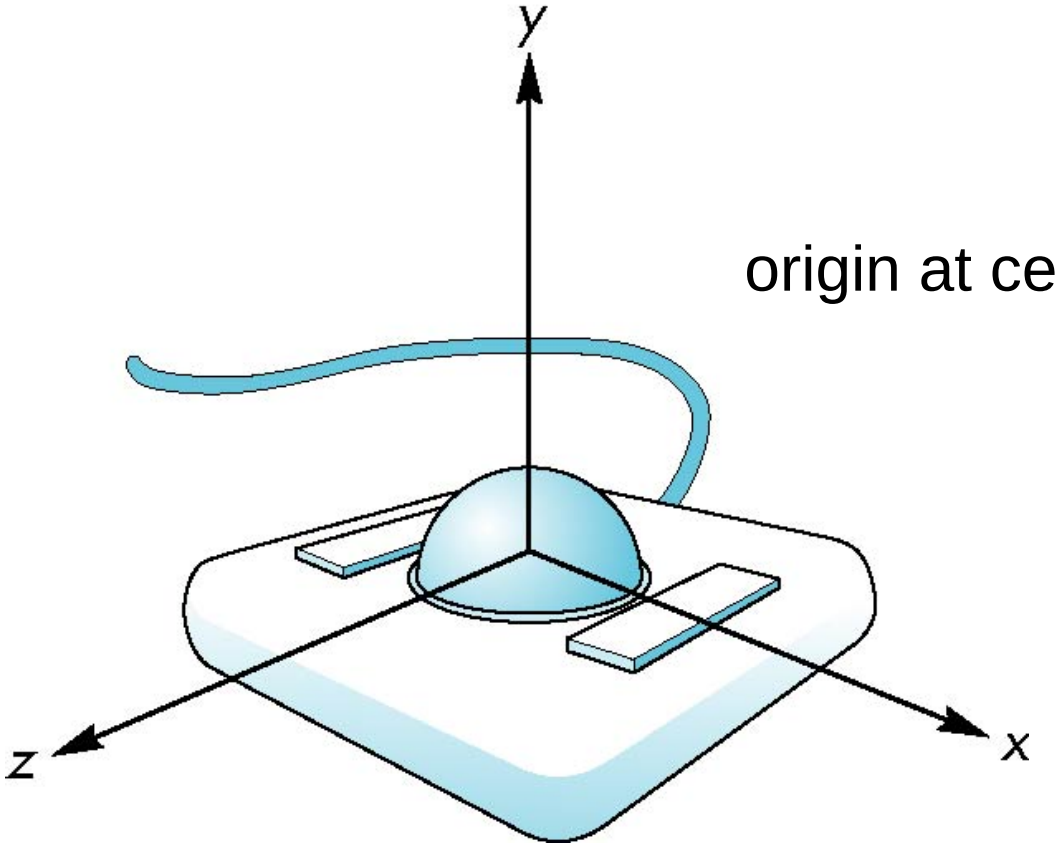


- If there is little friction between the ball and the rollers, we can give the ball a push and it will keep rolling yielding continuous changes
- Two possible modes of operation
  - Continuous pushing or tracking hand motion
  - Spinning

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

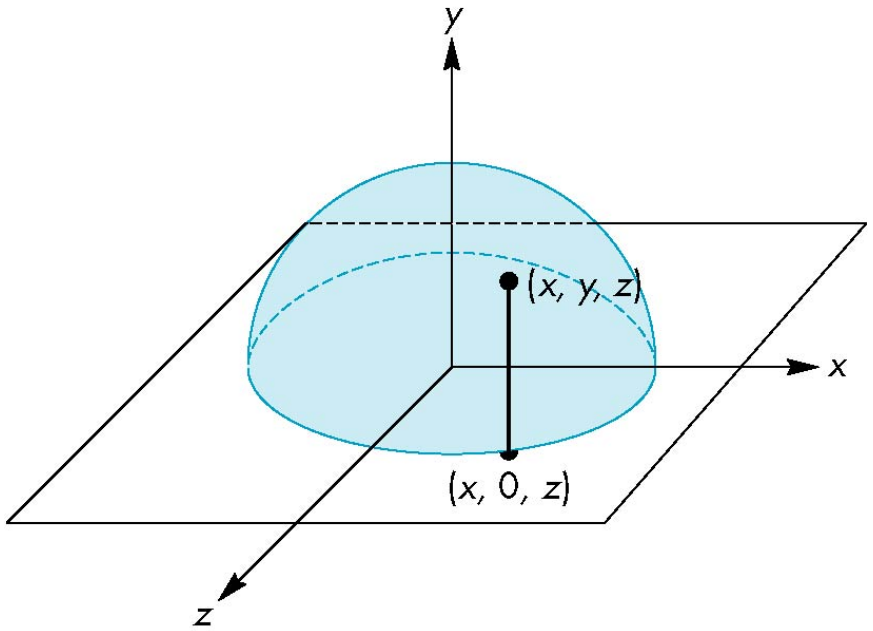
Trackball Frame



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

$$y = \sqrt{r^2 - x^2 - z^2} \qquad \text{if } r \geq |x| \geq 0,\ r \geq |z| \geq 0$$

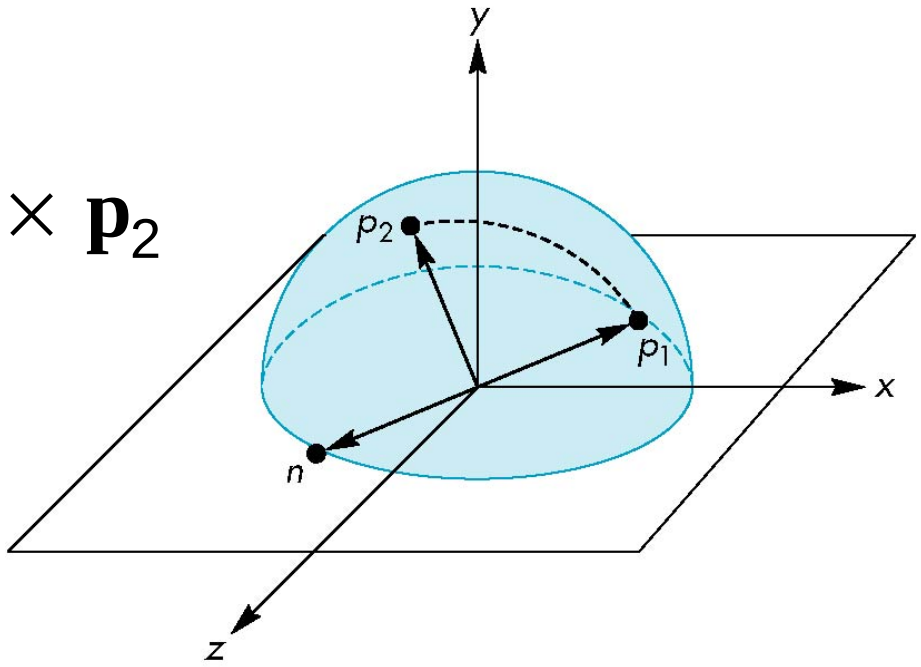
Computing Rotations

- 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
- **`redrawContinue`**: if true, idle function posts a redisplay
- **`trackballMove`**: if true, update rotation matrix

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

```
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;
    }
}
```

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