Implementing virtual trackball

1 Overview

Our goal is to implement a natural and easy to use interface allowing to rotate the displayed 3D model. We'll do it by letting the user rotate an imaginary ball that is located right in front of his eyes (center projects to the center of the window) and is inscribed into the window (Figure 1). To simplify things, we'll assume that both the ball and the model are centered at the origin (we'll say more precisely what it means for the model later on).

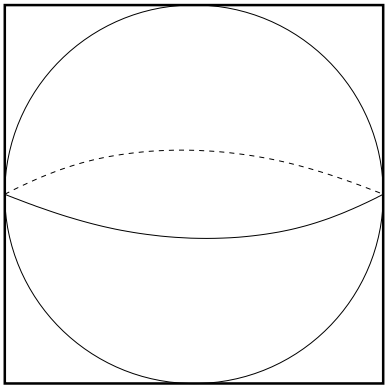

Figure 1: Trackball = imaginary ball whose center projects to the center of the window.

When the user clicks on a point on that ball and then moves the mouse with the button down, we'll make the ball (together with the displayed object) rotate so that the clicked point stays under the mouse cursor all the time.

We'll have a procedure that associates a point on the ball with every pixel of our window. For pixels that are in the projection of the ball onto the screen, this procedure will return an approximation of the intersection of the ray starting at the viewpoint through the pixel with the ball. For points that are not in the projection of the ball, the corresponding point on the ball will be the point which (again, approximately) is the closest to the ray from the viewpoint through that pixel.

When the user clicks on a pixel within our window (which, from programmer's perspective, basically means that an event handler is called), we'll calculate the corresponding point on the ball (call it A). Then, as long as the mouse is moved with the button down (such movement will generate a series of events) we'll compute the point on the ball corresponding to the pixel at which the mouse cursor is (B) and calculate the rotation $R_*(A, B)$ that takes A into B. this rotation will be applied *after* all rotations that resulted from previous user actions. Technically speaking, if R is the composite transformation defined by

all of the prior user's actions then we need to apply $R_*(A,B)R$ when we display the model. Unfortunately, this is not what OpenGL does: `glMultMatrix` multiplies the current transformation by the matrix provided by its argument on the right, not on the left. Therefore, in this case, we will maintain the R matrix in software:

- Whenever the 'mouse button up' event happens, we will calculate $R_*(A,B)$, where B is the point at which the button-up event happened and A - the point where the preceding button-down event happened and update $R := R_*(A,B) * R$ (clearly, R and $R_*(A,B)$ will be represented as either 3×3 or 4×4 matrices)

- To give immediate feedback to the user, we'll request a redraw after handling each of the mouse-moving-with-button-down events (this can be done by calling `glutPostRedisplay` function at the end of the event handler). When redrawing, we'll use $R_*(A,B) * R$ matrix as the rotation matrix. Here, A is the point where the preceding button-down event happened and B is the current location of the mouse cursor.

2 Computing $R_*(A,B)$

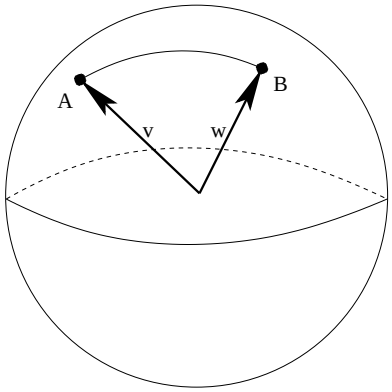

Figure 2: Two points A and B on the tranckball.

In the situation shown in Figure 2, we will compute $R_*(A,B)$ as the rotation that takes A 'directly' (i.e. along the shortest path possible) to B. To get the rotation axis, we take the cross product of the vectors from the center of the ball to A and B, $\vec{v} \times \vec{w}$. The rotation angle is simply the angle between the two vectors and can be computed using the dot product:

$$\angle(\vec{v},\vec{w}) = \mathrm{acos}(\frac{\vec{v}}{|\vec{v}|} \cdot \frac{\vec{w}}{|\vec{w}|}).$$

Having a rotation axis and an angle, how to write the rotation matrix? Say that $\vec{a} = [x, y, z]$ is a *unit* vector stretching along the rotation axis and α is the rotation angle. The rotation matrix is given by:

$$\begin{bmatrix} 1+(1-\cos(\alpha))(x^2-1) & -z\sin(\alpha)+(1-\cos(\alpha))xy & y\sin(\alpha)+(1-\cos(\alpha))xz \\ z\sin(\alpha)+(1-\cos(\alpha))xy & 1+(1-\cos(\alpha))(y^2-1) & -x\sin(\alpha)+(1-\cos(\alpha))yz \\ -y\sin(\alpha)+(1-\cos(\alpha))xz & x\sin(\alpha)+(1-\cos(\alpha))yz & 1+(1-\cos(\alpha))(z^2-1) \end{bmatrix}$$

3 Computing a point P on the ball corresponding to a pixel (x, y) in the window

Assuming that your view will be close to parallel, the trackball is inscribed in the (square) viewport and that the viewing direction is along the z-axis in the negative direction, you could treat x and y coordinates provided by the mouse events after scaling them to $[-1, 1]$ (be careful about the orientation of the screen when you scale), as the x- and y- coordinates of the point on the trackball (this is only an approximation, but will work OK). Then, compute that point as $(x, y, \sqrt{1-x^2-y^2})$. If the quantity under the root is negative (which means that the mouse event happens outside the trackball), scale (x, y) so that it is positive, i.e. replace it with $(x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2})$. In this case, the corresponding point on the trackball will be $(x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2}, 0)$

4 Zooming in and out

I suggest zooming in and out by changing the field of view in `gluPerspective`. Keep the trackball size constant, no matter what the zoom level is.

5 Where is the center of a 3D model??

Here, we'll define it as the center of the bounding box. Compute the maximum ($x_{\max}$) and minumum ($x_{\min}$) x-coordinates of the vertices of the model. Similarly, compute the minimum and maximum of y and z-coordinates ($y_{\min}, y_{\max}, z_{\min}, z_{\max}$). Notice that the paralleliped

$$\{(x, y, z) : x_{\min} \le x \le x_{\max}, y_{\min} \le y \le y_{\max}, z_{\min} \le z \le z_{\max}\}$$

is the smallest axis-oriented paralleliped containing our model. It is called its *bounding box*. By the center of the model we'll mean the center of the bounding box, i.e. the point whose coordinates are the averages of the corresponding maximum and minimum coordinates, i.e.

$$\Big(\frac{x_{\min}+x_{\max}}{2}, \frac{y_{\min}+y_{\max}}{2}, \frac{z_{\min}+z_{\max}}{2}\Big).$$

It may be convenient to scale a model to a unit size, i.e. so that the largest dimension of the bounding box spans from -1 to 1. This may be done by setting

up the modelview matrix carefully. The scaling factor to be used can be:

$$\frac{1}{\max(\frac{x_{\max}-x_{\min}}{2}, \frac{y_{\max}-y_{\min}}{2}, \frac{z_{\max}-z_{\min}}{2})}$$