# 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 the ball is centered at the origin and has radius 1. Also, we will assume that parallel projection (with projection rays parallel to the $z$-axis) is used to project the ball to the window. Note that the projection applied to the object you will be rendering will generally be different (and perspective).

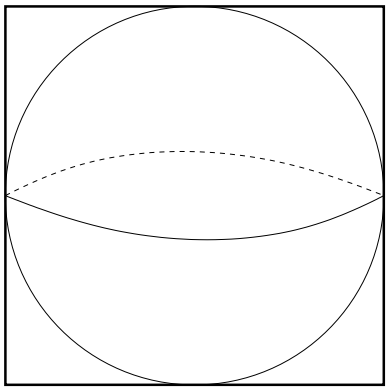

Figure 1: Trackball = imaginary ball whose center projects to the center of the window.

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.

Implementation of this procedure requires:

a function that computes the point of the sphere 'under' a pixel $(x, y)$.

a function that computes a rotation that takes a point $p$ on the sphere into a point $q$ on the sphere

implementing event handlers for mouse events

## 2 Computing point of the imaginary sphere under a pixel

Let's say we want to compute the point of the sphere under pixel $(i, j)$, where $i$ and $j$ are integers. Typically, $i$ and $j$ are going to be between 0 and $d-1$ if $d \times d$ window is used (althought they don't always have to be). Note that the coordinates $(i, j)$ are given relative to the upper left corner of the window (this is where $(0, 0)$ is).

The first step is to convert $(i, j)$ to $(x, y)$ (Figure 2). For the first coordinate, basically means scaling the coordinates to the range $[-1, 1]$: $x = 2i/(d-1)-1$. For the second coordinate, we also need to take into account the directions of the axes ($j$ points down and $y$ points up). The formula is $y = -(2j/(d-1)-1) = 1-2j/(d-1)$.

The next step is to compute the point on the imaginary sphere corresponding to $(x, y)$. In order to do that, we either intersect the line parallel to the $z$-axis and passing through $(x, y, 0)$ with the imaginary sphere or, if there is no intersection, compute the point of the sphere closest to that line.

If the point $(x, y)$ is inside the circle inscribed into the window (point $A$ in Figure 2), then the intersections of the line and the sphere are $(x, y, \pm\sqrt{1 - x^2 - y^2})$. since we are looking at the sphere from a direction along the positive part of the $z$-axis, the point that we see is $(x, y, \sqrt{1 - x^2 - y^2})$.

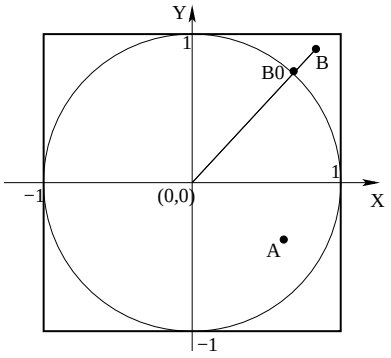


Figure 2: The imaginary ball. The $z$-axis points to the other side of the paper.

If the point $(x, y)$ is outside the circle inscribed into the window (point $B$ in Figure 2), then there is no intersection of the line and the sphere. Thus, in this case, we will use the point of the sphere that is closest to the line. To get the coordinates of that point (for point B in the Figure), we first radially project $B$ to the circle. This yields $B_0 = (x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2})$. Then, we add zero third coordinate to get the point on the sphere corresponding to $B$: $(x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2}, 0)$.

To sum up, to compute the point on the imaginary sphere corresponding to $(i, j)$ we first compute the $x, y$-coordinates of the pixel: $(x, y) := (2i/(d-1) - 1, 1 - 2j/(d-1))$. Then, the corresponding point on the sphere is given by:
$(x, y, \sqrt{1-x^2-y^2})$ if $1 - x^2 - y^2 \geq 0$
or
$(x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2}, 0)$ otherwise.

# 3 Rotation

To compute the rotation that takes a point $p$ on the sphere into a point $q$ on the sphere, we need to apply the method described in p. 14 of the 'Transformations: 3D' slides. Since the center of the sphere is at the origin and the radius is 1, the axis of the rotation is $p \times q$ and the angle is $\cos^{-1}(p \cdot q)$. If the points $p$ and $q$ are the same, the rotation that takes $p$ into $q$ is the identity.

# 4 Event handlers

Here is one way to implement the mouse event handlers that make the trackball work. It uses the following global variables:

$R$: the superposition of all 'finished' rotations (matrix)

$R_0$: the rotation that is currently being specified using the trackball interface (matrix)

$i_0, j_0$: coordinates of the last mouse button down event

$R$ and $R_0$ are initialized to identity at startup.

Note that all the mouse event handlers get the coordinates of the mouse cursor $(i, j)$ as arguments.

## 4.1 Mouse button down

Assign the event coordinates to $i_0$ and $j_0$.

## 4.2 Mouse moves with button down

We are in process of specifying a rotation. This means we need to update $R_0$. If $(i, j)$ are the event coordinates, then compute the points $p$ and $q$ corresponding the $(i_0, j_0)$ and $(i, j)$ (repsectively) as described in Section 2. Now, put the rotation that moves $p$ into $q$ (Section 3) into $R_0$. However, if $i_0 = i$ and $j_0 = j$, make $R_0$ equal to identity (note that in this case you need to be careful since by formulas in Section 3 the axis of rotation is the zero vector - attempting to use `glRotate*` or `glm::rotate` to find rotation with zero axis – Section 4.5 – will cause a problem).

Finally, we can request redrawing the contents of the window to give the user immediate feedback. To do that, call `glutPostRedisplay()`. This is a function that will add a redraw request to the event queue.

## 4.3 Mouse button up

We are now done specifying a new rotation. If $(i, j)$ are the event coordinates, then compute the points $p$ and $q$ corresponding the $(i_0, j_0)$ and $(i, j)$ (repsectively) as described in Section 2. Let $R'$ be the rotation that moves $p$ into $q$ (Section 3) or identity if $i_0 = i$ and $j_0 = j$. Do the following: $R := R' * R$.

Also, reset $R_0$ to identity (since we are not in process of specifying a new rotation any more) and request redrawing the contents of the window.

## 4.4 Drawing function

Apply rotation $R$ and then rotation $R_0$ when drawing the object.

## 4.5 A note on computing rotation matrices

If you are working on the OpenGL 4 version of the project, you are lucky: the `glm` library overloads matrix multiplication (as operator*) so it is really easy to use. My recommendation is to always use the first matrix argument of any `glm` transformation call to identity – then the transformation functions will just return a transformation matrix. You can multiply matrices using the * operator. See sample code for examples. If you are working on OpenGL 4 version, you can stop reading here.

In the OpenGL 1 version of the code, you can use the OpenGL command `glGetDoublev` with `GL_MODELVIEW_MATRIX` argument to read the current modelview matrix. This means that you can use OpenGL transformation calls to construct and multiply matrices.

Here is a piece of code that would compute the rotation matrix (useful in event handlers 4.2 and 4.3) :

```
double R0[16]; // entries of 4x4 matrix in order that OpenGL likes
glMatrixMode(GL_MODELVIEW); // we'll operate on modelview matrices
glPushMatrix(); // save the current matrix; just in case
glLoadIdentity(); // make the current modelview matrix equal to identity
glRotated(angle,axisx,axisy,axisz); // apply the rotation
glGetDoublev(GL_MODELVIEW_MATRIX,R0); // get the modelview matrix, (the rotation matrix)
glPopMatrix(); // restore the original modelview matrix; just in case
```

Note that the `glRotate*` function angle argument specifies the rotation angle in *degrees* not in radians.

You can use the same idea to multiply the matrix $R$ by the rotation matrix $R'$ on the left (event handler 4.3). Assuming that $R$ is a 16-dimensional vector of doubles representing a $4 \times 4$ matrix with OpenGL entry order, here is how to do that:

```
glMatrixMode(GL_MODELVIEW); // we'll operate on modelview matrices
glPushMatrix(); // save the current matrix; just in case
glLoadIdentity(); // make the current modelview matrix equal to identity
glRotated(angle,axisx,axisy,axisz); // apply the rotation R'
glMultMatrixd(R); // apply R
glGetDoublev(GL_MODELVIEW_MATRIX,R); // get the modelview matrix, put it to R
```

```
glPopMatrix(); // restore the original modelview matrix; just in case
```

Use `glMultMatrixd` in the drawing function to apply matrices $R$ and $R_0$. Be careful about the transformation order.

Note that the identity matrix in the 16-dimensional vector representation is

$$\{1, 0, 0, 0, \quad 0, 1, 0, 0, \quad 0, 0, 1, 0, \quad 0, 0, 0, 1\}$$

.