Rotating the Camera with the Mouse

Implementing a Virtual Trackball with the Windows Presentation Foundation (formerly codenamed Avalon)

Daniel Lehenbauer
http://blogs.msdn.com/danlehen

Abstract

Usually the first thing people want to do after they display a 3D model is to click on it and rotate it with the mouse. The most common technique for rotating 3D objects via the mouse is known as a "virtual trackball". This article will describe what a trackball does and walk through the mechanics of implementing one. At the end are links to sample code you can use to rotate the camera with the mouse in your own WPF applications.

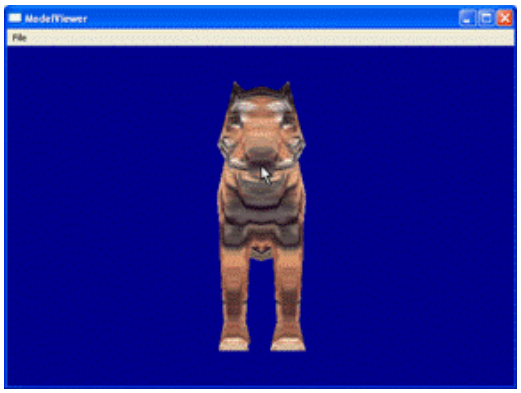

Figure 1a
Tiger model in its original configuration

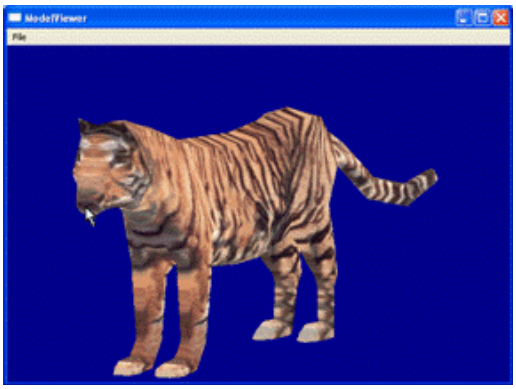

Figure 1b
Tiger model after the mouse has been dragged to the left and slightly down.

1. Introduction

A trackball translates 2D mouse movements into 3D rotations. This is done by projecting the position of the mouse on to an imaginary sphere behind the Viewport3D as shown in Figure 2. As the mouse is moved the camera (or scene) is rotated to keep the same point on the sphere underneath the mouse pointer.

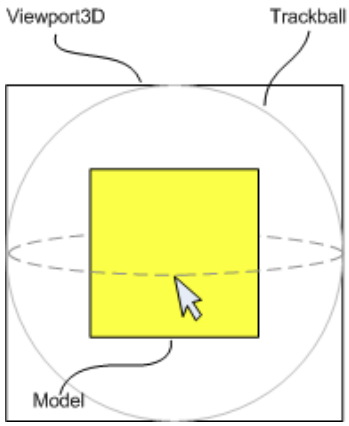

Figure 2a
The Viewport3D with the cube and inscribed trackball from the user's perspective

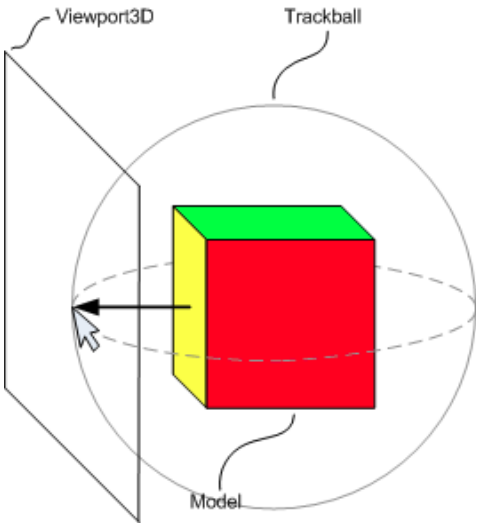

Figure 2b
Side view illustrating the point on the sphere which maps to the mouse position

When the mouse is moved horizontally a rotation about the Y axis is required to keep the same point under the mouse

pointer.

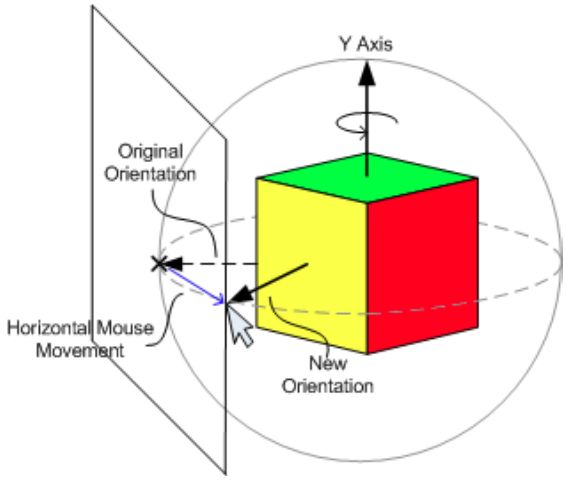

Figure 3
Moving the mouse vertically rotates the scene about the Y axis

Similarly, vertically changes in the mouse position result in rotation about the X axis.

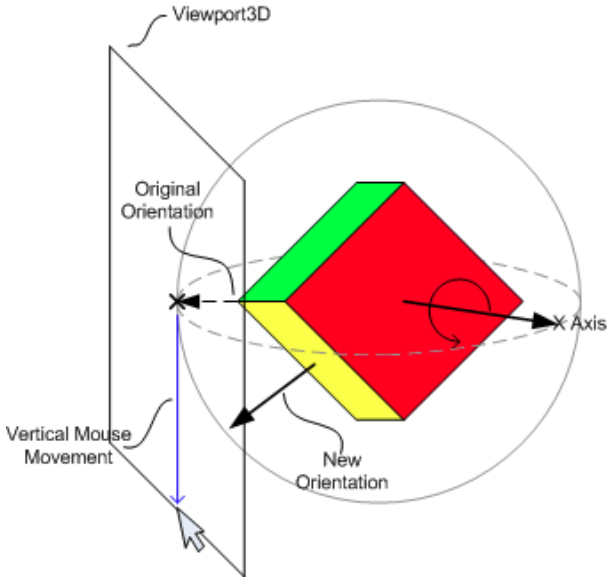

Figure 4
Moving the mouse vertically rotates the scene about the X axis

This interface provides a fairly intuitive method by which a model may be manipulated into any orientation by applying a combination of rotations about the X and Y axes.

2. Computing the Rotation

On each mouse move event we need to calculate a rotation to keep the same point under the mouse pointer. There are two steps to doing this. The first is figuring out what point on the sphere is under the mouse pointer. The second is computing the rotation required to transform the old point onto the new point.

2.1 Finding the Point on the Sphere

In order to find the point on the sphere under the mouse pointer we need to project the 2D point in the UIElement's coordinate system on to the sphere inscribed in the Viewport3D. Figure 5 illustrates the two coordinate systems.

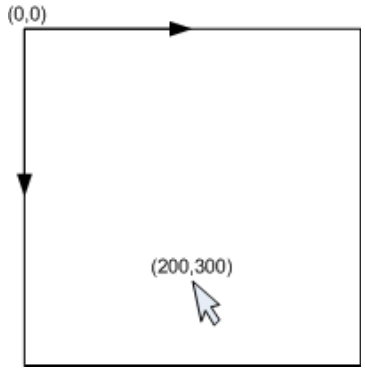

Figure 5a
The mouse reports its position in the coordinate space of the UIElement which has (0,0) in the upper left.

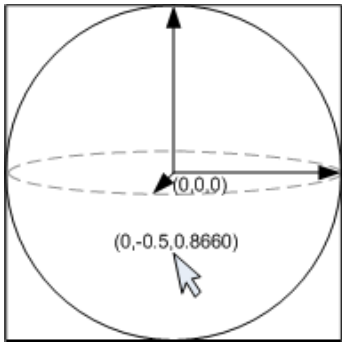

Figure 5b
We project this 2D point on to the sphere inscribed in the Viewport3D. Note that this results in a 3D coordinate.

Since we are only interested in calculating the rotation we can choose whichever coordinate system for the sphere that is most convenient for us. It is simplest to use a sphere of *radius* = 1 centered at the origin (0,0,0). This makes finding the X and Y components an exercise in converting between the two 2D coordinate systems shown in Figure 6.

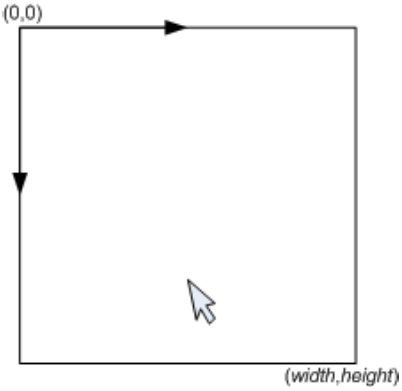

Figure 6a
Coordinate system of the UIElement

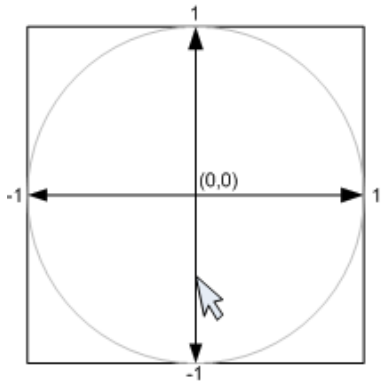

Figure 6b
Coordinate system of our trackball

To do this we construct a scale which will map the bounds of the Viewport3D to the range [0,0] - [2,2]. We then apply a translation to move the origin from the upper left corner to the center. This puts our point in the range [-1,1] - [1,-1]. Finally we account for the Y axis pointing down instead of up in the 2D coordinate system.

$$x = \frac{2 \cdot p_x}{width} - 1$$
$$y = 1 - \frac{2 \cdot p_y}{height}$$

```
// Scale bounds to [0,0] - [2,2]
double x = p.x / (width/2);
double y = p.y / (height/2);

// Translate 0,0 to the center
x = x - 1;

// Flip so +Y is up instead of down
y = 1 - y;
```

Now that we've found our x and y position on the sphere we can find z. Since our sphere is of *radius* = 1 we know that $\sqrt{x^2 + y^2 + z^2} = 1$. Solving for *z* we get:

$$z = \sqrt{1 - x^2 + y^2}$$
$$p = (x, y, z)$$

```
double z2 = 1 - x * x - y * y;
double z = z2 > 0 ? Math.Sqrt(z2) : 0;

Vector3D p = new Vector3D(x, y, z);
p.Normalize();
```

We now have the (*x*,*y*,*z*) coordinates of the point on the sphere beneath the mouse pointer.

2.2 Rotating Between the Points

On each mouse move we want to construct a rotation that will keep the same point on the sphere underneath the mouse pointer. We do this by remembering the previous point on the sphere from the last mouse move event and constructing a rotation that will transform it to the point currently under the mouse pointer.

To compute this rotation we need two things:

1. The axis of rotation
2. The angle of rotation θ

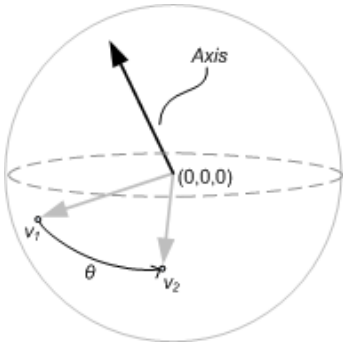

Figure 7
We need to find the axis of rotation and angle θ that will transform v_1 onto v_2.

Because our sphere is centered at the origin we may interpret our points as vectors. Doing so it is trivial to find the axis and angle of rotation using the cross product and dot product respectively:

$$Axis = v_1 \times v_2$$

$$\theta = \arccos\left(\frac{v_1}{|v_1|} \cdot \frac{v_2}{|v_2|}\right)$$

```
Vector3D axis = Vector3D.CrossProduct(v1, v2);
double theta = Vector3D.AngleBetween(v1, v2);
```

Once we have the axis and angle all that remains is to apply the new rotation to the current orientation:

```
// We negate the angle because we are rotating the camera.
// Do not do this if you are rotating the scene instead.
Quaternion delta = new Quaternion(axis, -angle);

// Get the current orientantion from the RotateTransform3D
RotateTransform3D rt = (RotateTransform3D) camera.Transform;
AxisAngleRotation3D r = (AxisAngleRotation3D) rt.Rotation;
Quaternion q = new Quaternion(r.Axis, r.Angle);

// Compose the delta with the previous orientation
q *= delta;

// Write the new orientation back to the Rotation3D
r.Axis = q.Axis;
r.Angle = q.Angle;
```

3. Other Details

There are a few details we glossed over in Section 2. The first is that the calculations to project the mouse pointer onto the sphere assume that the Viewport3D is square. If the Viewport3D is oblong the inscribed trackball will actually be an ellipsoid:

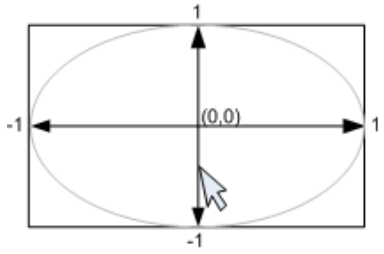

Figure 8
If the Viewport3D is oblong the inscribed trackball will actually be an ellipsoid

This effect isn't usually noticeable, but if the aspect ratio is extreme this will cause the rate of rotation to be significantly faster when moving along the shorter axes. To correct for this you can apply a uniform scale when mapping the 2D point to the sphere instead of (*width*, *height*). For example, min(*width*, *height*) would work. Whatever you choose, remember to account for this when translating to the origin.

Another issue is how to handle the case when the mouse pointer does not map to position on the trackball:

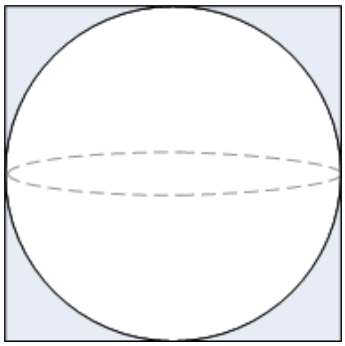

Figure 9
The shaded region does not map to a point on the trackball

One possible solution is to clamp *z* to zero in this case as is shown at then end of Section 2.1:

```
double z = z2 > 0 ? Math.Sqrt(z2) : 0;
```

Technically we should also normalize *x* and *y* to find the nearest point on the trackball in the $Z = 0$ plane, otherwise the point we return is not on the sphere:

$$x = \frac{x}{\sqrt{x^2 + y^2}}$$

$$y = \frac{y}{\sqrt{x^2 + y^2}}$$

However, in Section 2.2 we use `Vector3D.AngleBetween(v1, v2)` which accounts for the non-normalized vectors. This is yields equivalent results to normalizing *x* and *y* as shown above.

We also did not discuss the initial placement of the model and camera. This implementation assumes that the model is centered at the origin and that the camera is looking at the origin and positioned at a distance such that the model is visible.

Finally, this article does not discuss how zoom is implemented, although the sample code includes a reasonable implementation.

4. Sample Code

The sample code has contains three reusable pieces:

Trackball.cs	A utility class which observes mouse events on a FrameworkElement

	to update a Transform3D with the resultant rotation and scale.
Trackport.proj	A UserControl which loads and displays a Model3D from loose .xaml and allows the view to be manipulated via the mouse (an example of using Trackball.cs)
ModelViewer.proj	The Model Viewer application pictured in Figure 1 (an example of using Trackport.proj).

These are included in the “3D Tools for the Windows Presentation Foundation” workspace at this URL: http://workspaces.gotdotnet.com/3DTools

You need not join the workspace to download the releases which include both the binaries and source (here).

Acknowledgements

I would like to thank my wife, Bonnie, for her contributions to the model viewer sample.