The note focuses on how to use a scene-graph to easily support a virtual track-ball or examiner-style interaction. Additionally, support for zooming (dolly) into the scene is provided.

Given: A model (or scene) centered at (0,0,0) in world-space. We will call this $\text{Scene}$. The key to getting this to work is to realize that you want all rotations about the point (0,0,0) in world-space. However, it is much easier to perform the rotations in eye-space. The camera is now the parent of the scene, performing a transformation of the scene coordinates from world-space to eye-space. The camera (or eye) is now at (0,0,0), and the scene is centered at (0,0,-$d$), where $d$ is the distance from the camera’s world-space location to the origin (or zero for orthogonal projections). Recall that in eye-space, the negative z-axis points into the screen.

Steps:

1) Let’s first add a camera. It does not matter if this is a perspective or orthographic camera (orthographic cameras could have a slightly simpler model). We define a camera in world-space with a position and a viewing direction or look-at point. We are going to constrain the look-at point to be the world-space point (0,0,0) such that the model is centered on the screen. What does this do to our scene-graph? Our camera is now the parent of the scene, performing a transformation of the scene coordinates from world-space to eye-space. The camera (or eye) is now at (0,0,0), and the scene is centered at (0,0,-$d$), where $d$ is the distance from the camera’s world-space location to the origin (or zero for orthogonal projections). Recall that in eye-space, the negative z-axis points into the screen.

2) Now, we want a virtual track-ball to provide rotations in eye-space about the point (0,0,-$d$). The standard rotation about a fixed-point applies, translate, rotate, and translate back. We will add three new scene-nodes to accomplish each of these.

3) The Rotation node is simply a matrix node with operations to apply a rotation about an arbitrary axis. We will accumulate these as the mouse interacts. The virtual-trackball can now determine the rotation axis and angle from the mouse handler and apply a rotation to this Rotation node. However, after one rotation, we want the next rotation to occur in the new eye-space. This is the space with the previous rotation already applied. It may appear that this is impossible, as we want the translate-rotate-translate for the new eye-space. However, our distance, $d$, has not changed so if we were to logically apply the same process again we would get: translate-new rotate-translate-translate-translate. The two inner translates cancel each other out and we can see that we need our new rotation to be pre-multiplied with the current set of rotations.

4) To allow the camera distance to change, we simply need to change the $d$ value in both translate nodes. One implementation of this is to have keyboard input add or subtract a delta from the translate nodes. This supports a dolly operation on the camera. A zoom or lens can also be achieved by changing the perspective projection, which would be typically be the parent node (and root node) of the entire scene graph.

Final Thoughts: A clean separation of concerns is now possible; your mouse handler for the virtual track-ball simply has access to either a matrix or the Rotation node. On redraw the new matrix will be applied. This does require your own matrix code with support for matrix multiplication and rotation matrices. I will leave it as an exercise to you as to whether the rotations should be positive or negative in your trackball system. Another common use of a track-ball is to orient individual models in a scene. Here we want the rotations in eye-space, but applied in model-space!