1 Connecting User Input and Viewing

Interactive data viewing requires us to connect user input to changes in the view system parameters. User actions generate either specific changes in the view parameters or incremental modifications. The forms of interactive view modification include panning, scaling, and rotating.

- Panning: should move the VRP around the view plane, keeping the orientation of VUP and U fixed.
- Scaling: should increase or decrease the extent of the view volume in the view plane, keeping other parameters fixed.
- Rotation:
  - In 2D, should rotate the VUP vector about the z-axis, modifying the VUP vector and the derived U vector, but leaving their Z values at 0.
  - In 3D, the VRP may move in a circle around a fixation point (e.g. center of the current view volume). The same rotation can be applied to the VPN, VUP, and U vectors. An alternative interface method is to rotate the view direction according to mouse motions, keeping the VRP constant during the rotation.

The mouse gives the user the ability to move within 2 dimensions. Therefore, whether we are translating, rotating, or scaling, we have to have a method for translating user actions into modifications of the view reference coordinates.

- Translation of the mouse: translation of the VRP in the view plane
- Translation of the mouse: rotation of the VUP vector
- Translation of the mouse: scaling of the extent of the view window

A typical mouse has three buttons (left, right, center) which translate into three separate button clicks. Many mice also have a wheel or trackball on them, which provides an additional input device. The keyboard and menu items provide additional inputs that we can use as modifiers to the mouse action.

The key is how we connect the user interface elements to the parameters of the view transformation. Some key design questions include the following.
- Which mouse actions, buttons, or menu states connect to which view parameters?
- How does each action or button press relate to changes in one or more view parameters?
- What is the control law for the relationship?
- What are the control parameters of the relationship?
- Why is this a useful relationship?

Note that the first four questions are up to the programmer. You can make the relationships be whatever you want. The last question is what should guide the design process, and is often the motivation for user studies that look at the effects of different design decisions.
1.1 Interfaces as Control Laws

Any time we are connecting an input to an action, we have to write a control law. Consider, for example, connecting the motion of a mouse to panning of the data. The mouse moves in screen coordinates. Therefore, the input to the system is in pixels. Pixels, however, are not a meaningful unit in the data space. We can write the relationship between mouse motion and motion in the data space as follows, where $\Delta U$ is the motion in the data space, $\Delta x$ is motion in screen space, and $k_p$ is a proportional constant relating the two motions.

$$\Delta U = k_p \Delta x$$  \hspace{1cm} (1)

We can control the relationship between the two spaces by varying the value of $k_p$. If we set $k_p = \frac{E_d}{E_s}$, which is the ratio of the data space extent to the screen space, then the data will appear to track the mouse as it moves. However, we could make $k_p$ smaller or larger than that ratio and get different effects. If $k_p$ is smaller, the data will lag the mouse motion. If $k_p$ is bigger, the data will precede the mouse motion. The former is useful for fine tuning of a visualization, the latter is useful for moving quickly between different parts of the data.

We are not limited to a simple proportional relationship, however. Consider, for example, the use of an inertia term.

$$\Delta U_t = k_p \Delta x + k_n \Delta U_{t-1}$$  \hspace{1cm} (2)

The inertia term means that the motion of the data is dependent not only on the current user input, but also the prior user input. A more complex form of the motion results if we make $k_n$ dependent upon whether the user is actively controlling the device. Under active control, we can make $k_n = 0$, changing it to something like $k_n = 0.9$ when the user stops actively controlling the mouse. The iPod, and other multi-touch devices make use of this type of relationship to enable quick flipping and scrolling through lists.

Control theory also gives us an intuitive way of understanding the results of different design choices.

1. overdamped - system doesn’t respond rapidly enough (sluggish)
2. critically damped - system responds just right
3. underdamped - system responds too strongly and in an effort to correct the overshoot, you end up oscillating
4. unstable - system just doesn’t do the right thing
1.1.1 Panning within the view plane

- Each change in the mouse position corresponds to a change in the position in the data space
- Horizontal motion in the view plane should move along the U axis
- Vertical motion in the view plane should move along the VUP axis
- The view volume extent and the screen size tell us how to scale pixel motion to data space motion

Process

1. Calculate how much the mouse moved on the screen $\Delta x, \Delta y$.

2. Scale the motion into data space by dividing by the screen size and multiplying by the extent.

   \[
   (\Delta u, \Delta v) = \left(\frac{\Delta x}{S_x}, \frac{\Delta y}{S_y}\right) \tag{3}
   \]

3. Multiply the horizontal screen motion by the U axis and the vertical screen motion by the V axis to get the motion of the VRP in data space.

   \[
   \begin{align*}
   \Delta \text{VRP}_x &= \Delta uU_x + \Delta vVUP_x \\
   \Delta \text{VRP}_y &= \Delta uU_y + \Delta vVUP_y \\
   \Delta \text{VRP}_z &= \Delta uU_z + \Delta vVUP_z \tag{4}
   \end{align*}
   \]

4. Add the motions onto the VRP.

   \[
   \text{VRP} = (\text{VRP}_x + \Delta \text{VRP}_x, \text{VRP}_y + \Delta \text{VRP}_y, \text{VRP}_z + \Delta \text{VRP}_z) \tag{5}
   \]

5. Recalculate the view transformation matrix

6. Calculate the new view locations of the data

7. Adjust the coordinates of the visual objects
1.1.2 Scaling the extent

- The extent should scale uniformly in all directions
- The user should be able to scale up and down using the same motion
- An easy solution is to translate vertical motion into scaling

Process

1. Store the initial mouse click as a reference point $P_0$
2. Store the initial view extent $E_0$
3. For each mouse motion
   (a) Calculate the vertical distance between the initial click and the current mouse location.

   $$\Delta v = P_{iy} - P_{0y}$$

   (6)

   (b) Generate a multiplication factor from the difference. Set the value of $k$ to control the speed of the scaling.

   $$f = 1.0 + k\Delta v$$

   (7)

   (c) Bound the factor on the zoom side to a small number larger than zero (e.g. 0.05)

   (d) Multiply the initial view extent by the factor and update the view extent

   $$E_i = fE_0$$

   (8)

   (e) Recalculate the view matrix

   (f) Calculate the new view locations of the data

   (g) Adjust the coordinates of the visual objects
1.1.3 Rotating the view up vector

1. Rotation for 2D data reorients the view volume, but the view window stays in the x-y plane.
2. Rotation should not change the location of the VRP or the view direction (VPN)
3. The visual point of rotation is generally the middle of the viewing screen

Process

1. Store the initial click and calculate the angle relative to horizontal (that passes through the center of the screen, because that is the horizontal that passes through the view reference point). in python, the function 
   \[ \text{math.atan2}(y, x) \]
   calculates the angle relative to the x-axis defined by the vector \((x, y)\).
2. Store the initial VUP vector: \(VUP_0\)
3. For each mouse motion
   (a) Calculate the current mouse angle relative to horizontal (that passes through the center of the screen) using \(\text{atan2}\).
   (b) Subtract the angles to get the amount of rotation \(\alpha\)
   (c) Generate a rotation matrix for the angle \(-\alpha\)
      \[
      R(-\alpha) = \begin{bmatrix}
      \cos \alpha & \sin \alpha & 0 & 0 \\
      -\sin \alpha & \cos \alpha & 0 & 0 \\
      0 & 0 & 1 & 0 \\
      0 & 0 & 0 & 1
      \end{bmatrix}
      \]
      (9)
   (d) Rotate the initial VUP vector and update the view object
      \[
      VUP_\alpha = R(-\alpha)VUP_0
      \]
      (10)
   (e) Recalculate the view matrix
   (f) Calculate the new view locations of the data
   (g) Adjust the coordinates of the visual objects