1 User Interaction

So far, we've mostly been creating images and animations, rather than interacting with the user. Our interaction has mostly been limited to keyboard callbacks using TW, and even that has been mostly toggling global settings such as lighting or textures. This is fine for producing animations (after all, Pixar movies aren't interactive), but if we want to do games and other kinds of interactive software, we'll have to do better. In this chapter, we'll dig into how to have a more interactive program.

1.1 OpenGL Keyboard Callbacks

OpenGL has a simple interface for defining keyboard callbacks. Just as with the "display" callback, you *register* a function that is called. This function will be called if *any* regular key on the keyboard is called. (We'll talk about other keys later.) For example, the following C code will exit the program if any key is pressed:

```
void handleKeyboardInput( char key, int x, int y ) {
    // any key to exit
    exit(0);
}
```

Or, in Python:

```
def handleKeyboardInput( key, x, y ):
    # any key to exit
    sys.exit(0)
```

To *register* this function, we just use the following. We'd probably put this in `main`, right next to where we register the display function:

```
    glutKeyboardFunc(handleKeyboardInput);
```

The callback function will always be called with three arguments: the ASCII code of the key that was pressed, and the x and y coordinates of the mouse's location at the time that the key is pressed. Therefore, the callback function must have the following signature in C (recall that a function's *signature* means return type, and the number and types of all its arguments):

```
  void myKeyFunction(char key, int x, int y);
```

Python does not require you to declare the function signature, so there's no explicit syntax for that.

In other words, you can bind a key to any function that takes three arguments (a character and two ints) and returns no value.

Since you can only have one function bound at a time, and it gets all keyboard input, the function has to sort out what key was pressed. Thus, most keyboard callback functions in C look like this:

```
  void myKeyFunction(char key, int x, int y) {
     switch(key) {
     case 'a':
```

```
            // code to do whatever ``a'' does
            glutPostRedisplay();
            break;
        case 'b':
            // code to do whatever ``b'' does
            glutPostRedisplay();
            break;
        ...
        case 'q':
            // q is for ``quit''
            exit(0);
            break;
        ...
        default:
            // handle all other keys
            break;
        } // end switch
}
```

Or, in Python:

```
def myKeyFunction(key, x, y):
    if key == 'a':
         # code to do whatever ``a'' does
         glutPostRedisplay()
    elif key == 'b':
         # code to do whatever ``b'' does
         glutPostRedisplay()
     ...
    elif key == 'q':
         # q is for ``quit''
         sys.exit(0)
     ...
     else:
         # handle all other keys
```

There are, of course, many variations on this kind of code. For example, several similar blocks might be coalesced.

Recall that the function also gets the (x, y) coordinates of the mouse at the time that the key was pressed. This by itself can help make our programs more interactive. For example, we could move the mouse over an object and then by pressing the "delete" key (ASCII code 127), indicate that the object should be deleted. However, we have a bit more work to do before we can see how do to that.

1.2 Mouse Coordinates

When the monitor redraws the screen, it starts in the upper left corner and the electron gun sweeps left to right and top to bottom. For that reason, many graphics programs use a coordinate system where the origin is at the upper left, the x coordinate increases to the left and the y coordinate increases going *down*. Down? Yes, down. This may seem odd, but it's true. The mouse coordinates are reported in *window* coordinates, which is in pixels. If you open up a 500 by 300 window, those values will be, respectively, the largest possible x and y coordinates. See figure 1.

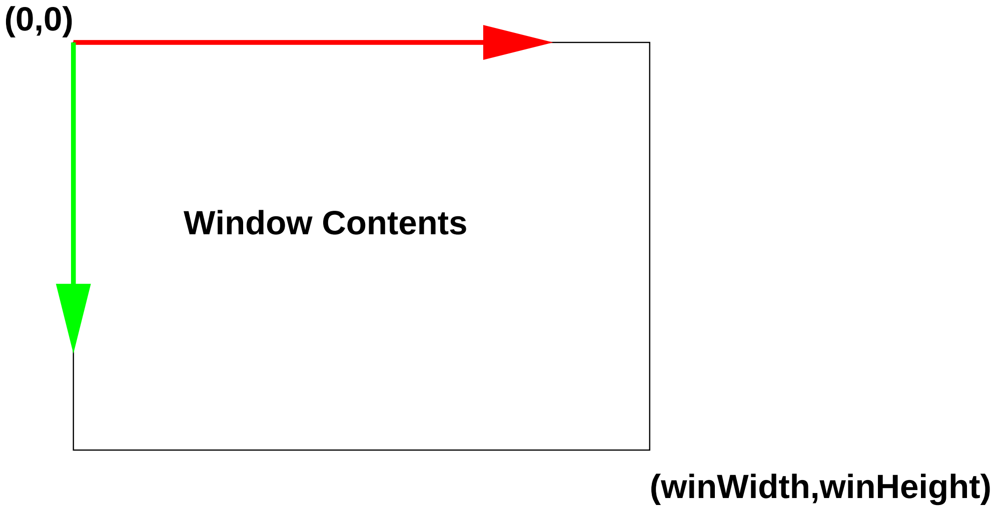

Figure 1: Mouse Coordinates are reported with 0,0 in the upper left, and maximum values of window width and window height.

1.3 Unit Camera Movement

Suppose that we are using the mouse and keyboard callbacks (we'll combine the two soon). When the mouse is in the left half of the window, a key press or mouse click means "move to the left," and similarly if the mouse is in the right half of the window. (I'm told that in the gaming community, this sort of movement is known as "strafing.") Also, if the mouse is in the upper half of the window, a key press or mouse click means "move up," and similarly an action in the lower half means "move down." Assuming (for the sake of simplicity), that the camera is facing down the $-z$ axis, how can we implement this sort of movement?

First, we need to know how big the window is, so that we can know where the middle is. Let's set up global variables to record this. These could be constants, but if we want to allow the user to reshape the window, we would set up a *reshape* callback that would modify these values if the window changes size.

```
WinWidth = 500
WinHeight = 300
```

Assume that the camera is set up in the *display* callback as follows. Notice how this enforces our assumption that the camera is always facing parallel to $-z$.

```
gluLookAt(eyeX,eyeY,eyeZ,
          eyeX,eyeY,eyeZ-1,
          0,1,0)
```

Our callback function in C can then operate thus:

```
void callback( char key, int x, int y ) {
   x = x - WinWidth/2;
   y = WinHeight/2 - y;
   if( x > 0 )
       eyeX++;  // move right
   else if( x < 0 )
       eyeX--;
   if( y > 0 )
       eyeY++;
   else if( y < 0 )
       eyeY--;
   ...
}
```

Or in Python:

```
def callback( key, x, y ):
   x = x - WinWidth/2
   y = WinHeight/2 - y
   if x > 0:
       eyeX += 1   # move right
   elif x < 0 :
       eyeX -= 1
   if y > 0 :
       eyeY += 1
   elif y < 0 :
       eyeY -= 1
   ...
```

Let's focus on the first two lines of the callback function. Essentially what we're doing is mapping to a coordinate system where (0,0) is in the center of the window, x increases to the right and y increases up (whew!). See figure 2. This easily divides the window into the four signed quadrants that we're used to.

The rest of the callback function is just incrementing or decrementing the eye's x or y coordinate based on the sign of the mouse coordinate. If you want a fancier coding technique for this, we can create a `signum` function (`http://planetmath.org/encyclopedia/SignumFunction.html`) and use it to modify the eye coordinates.

```
// returns 1 if x>0, -1 if x<0, and 0 if x==0
int signum( int x ) {
  return x==0 ? 0 : x<0 ? -1 : 1;
}

void callback( char key, int x, int y ) {
   x = x - WinWidth/2;
   y = WinHeight/2 - y;

   eyeX += signum(x);
   eyeY += signum(y);
```

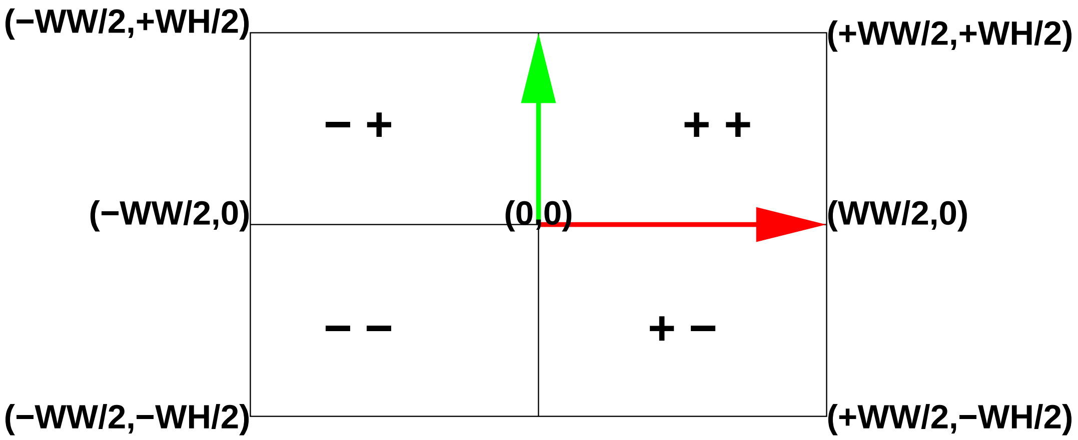

Figure 2: We can map the mouse coordinates to a coordinate system where the center is (0,0) and and the x coordinate can range from negative half the window width (-WW/2) to positive half the window width (WW/2), and similarly the y coordinate ranges from negative half the window height (-WH/2) to positive half the window height (WH/2).

```
    ...
}
```

In Python:

```
# returns 1 if x>0, -1 if x<0, and 0 if x==0
def signum( x ):
  return 0 if x==0 else -1 if x<0 else 1

def callback( key, x, y ):
   x = x - WinWidth/2
   y = WinHeight/2 - y

   eyeX += signum(x)
   eyeY += signum(y)
   ...
```

1.4 Proportional Camera Movement

In the previous section, we're throwing away a lot of information when we just use the *sign* of the mouse coordinates. Why not move the camera a lot if the mouse click is far from the center, but only a little if it is close to the center? That is, we could make the amount of movement *proportional* to the distance from the center. Now our mouse is becoming useful. Building on the ideas from the previous section, our coding is fairly straightforward.

Recall that the maximum absolute value of the mouse coordinates is half the window width or height. If someone clicks at the extreme edge and we want that to result in, say, the camera moving by X or Y units, we can arrange for that with a straightforward mathematical mapping. We first map the x and y coordinates onto the range [-1,1] by dividing by their maximum value. Then, multiply that by the largest amount we would want to move. (Call that the `Xspeed` and `Yspeed`.)

The C code is as follows:

```
const float Xspeed = 3.0;    // just an example
const float Yspeed = 4.0;    // just an example

void callback( char key, int x, int y ) {
   x = x - WinWidth/2;
   y = WinHeight/2 - y;

   float xMove = Xspeed*x/(WinWidth/2);
   float yMove = Yspeed*y/(WinHeight/2);

   eyeX += xMove;
   eyeY += yMove;
   ...
}
```

The Python code is:

```
Xspeed = 3.0    # just an example
Yspeed = 4.0    # just an example

def callback( key, x, y ):
   x = x - WinWidth/2
   y = WinHeight/2 - y

   xMove = Xspeed*x/(WinWidth/2)
   yMove = Yspeed*y/(WinHeight/2)

   eyeX += xMove
   eyeY += yMove
   ...
```

Thus, if the user clicks in the middle of the lower right quadrant (the $+-$ quadrant), x will have a value of $+0.5$ and y will have a value of -0.5, and so `xMove` will be 1.5 and `yMove` will be -2.0.

Of course, we're not limited to a linear proportionality function. If, for example, we used a quadratic function of the distance, mouse clicks near the center could result in slow, fine movements, while clicks far from the center could result in quick, bit movements. This could be useful in some applications.

1.5 Mouse Click Callbacks

Before we proceed to more sophisticated camera movement, let's look at how to set up mouse callbacks in OpenGL. As hinted above, they are very similar to keyboard callbacks. The main difference is that you can find out whether the button has just been *pressed* or just been *released*. That is, you can tell an "up transition" from a "down transition." This ability means you can have behavior like "click and drag," where the mouse is released in a different place than it is pressed. (Recall that TW uses click-and-drag for moving the viewpoint.)

The first callback is

```
glutMouseFunc(yourMouseCallbackFunction)
```

This function registers a function that is called whenever a mouse button is pressed or released. The signature of the callback is as follows:

```
void yourMouseCallbackFunction( int button, int state, int x, int y );
```

- The `button` argument will be one of the following values.

  ```
  GLUT_LEFT_BUTTON
  GLUT_MIDDLE_BUTTON
  GLUT_RIGHT_BUTTON
  ```

- The `state` argument will say the current state of that button (that is, after the transition). It has two possible values:

  ```
  GLUT_UP
  GLUT_DOWN
  ```

  If that argument has the value `GLUT_UP`, that means that the button is up, so it must've just been *released*. Obviously, this means that if that argument has the value `GLUT_DOWN`, it means that the button has just been *pressed*.

- The last two arguments are the same as our other mouse location arguments.

If you are only interested in clicks (down transitions), it's pretty common to have your C callback function start like this:

```
void yourMouseCallbackFunction( int button, int state, int x, int y ) {
    if( button == GLUT_UP ) return;
    ...
```

Or, in Python:

```
def yourMouseCallbackFunction( button, state, x, y ):
    if button == GLUT_UP:
        return
    ...
```

Two other important mouse callbacks are the "motion" function and the "passive motion" function, which you register with one of the following. The "motion" function is called whenever the mouse moves *while* a button is pressed (so it's part of a click-and-drag behavior). The "passive motion" function is called whenever the mouse moves with no buttons pressed.

```
glutMotionFunc( yourMotionFunc )
glutPassiveMotionFunc( yourMotionFunc )
```

The signature for these two callbacks is the same, because they both just get the current location of the mouse. The mouse location is, as always, with the origin in the upper left and y increasing down.

```
void yourMotionFunc( int x, int y );
```

Notice that either of these functions only gives you a snapshot of the mouse motion. It doesn't tell you where the mouse was, how it was moving, or anything like that. In our geometry terminology, it gives you a *point*, not a *vector*. Yet you often want to know what direction the mouse was moving. For example, in the TW interface for moving the camera viewpoint, if you drag the mouse down, you get a very different effect than if you drag it to the right, even if you end up at the same location. The callback will only get the location, not the direction, so how to do this? The answer is simply to keep track of where the mouse was.

In C, we would do:

```
int oldX;
int oldY;

void yourMouseFunc( int button, int state, int x, int y ) {
   if( state == GLUT_UP ) return;
   if( button != GLUT_LEFT_BUTTON ) return;
   // record starting values for click-and-drag
   oldX = x;
   oldY = y;
}

void yourMotionFunc( int x, int y ) {
   int deltax = x - oldX;
   int deltay = y - oldY;
   ...
   // process mouse motion based on vector (deltax,deltay)
   ...
   // record mouse position for next motion
   oldX = x;
   oldY = y;
}
```

Or, in Python:

```
oldX = None
oldY = None

def yourMouseFunc( button, state, x, y ):
    global oldX, oldY
    if state == GLUT_UP:
        return
    if button != GLUT_LEFT_BUTTON:
        return
    # record starting values for click-and-drag
    # note the use of Python's parallel assignment
    oldX, oldY = x, y


def yourMotionFunc( x, y ):
```

```
deltax = x - oldX
deltay = y - oldY
...
# process mouse motion based on vector (deltax,deltay)
...
# record mouse position for next motion
oldX, oldY = x, y
```

1.6 Picking and Projection

So far, our interaction has been only to move the camera, but suppose we want to interact with the objects in the scene. For example, we want to click on a vertex and operate on it (move it, delete it, inspect it, copy it, or whatever). We saw this in the Bézier Curve Tutor (`tw/demos/curves-and-surfaces/BezierTutor.cc`). The notion of "clicking" on a vertex is the crucial part, and is technically known as *picking*, because we must pick one vertex out of the many vertices in our scene. Once a vertex is picked, we can then operate on it. We can also imagine picking line segments, polygons, whole objects or whatever. For now, let's imagine we want to pick a vertex.

Picking is hard because the mouse location is given in window coordinates, which are in a 2D coordinate system, no matter how we translate and scale the coordinate system. The objects we want to pick are in our scene, in world coordinates. What connects these two coordinate systems? *Projection*. The 3D scene is projected to 2D window coordinates when it is rendered.

Thus, one important way to handle picking a vertex is to project every vertex in the scene and determine which one is closest to the mouse location when the click happened. Notice that this means the program will have to have a data structure that contains all the vertices, so that they can each be projected. The vertex arrays that we used for drawing the barn and other polyhedra are very useful here.

But how do we project a vertex? Clearly the OpenGL pipeline does this, but that happens down in the graphics hardware, and our program doesn't have access to the result. However, it turns out that OpenGL will carry out the projection calculation for you and store the result in variables of your choosing. That's done with the following function:

```
GLint gluProject( GLdouble objX,
                  GLdouble objY,
                  GLdouble objZ,
                  const GLdouble *model,
                  const GLdouble *proj,
                  const GLint *view,
                  GLdouble* winX,
                  GLdouble* winY,
                  GLdouble* winZ )
```

The argument list is intimidating (the PyOpenGL version is a little better, as we will see), but let's take it one step at a time.

- The first three arguments are simply the world coordinates of the object that you want to project.

- The next argument is the modelview matrix. This function doesn't use the current modelview, but instead allows you to provide one. That makes the function more general: you can project an object using a different modelview matrix than the one that is currently being applied to the pipeline (maybe you want to test something before you change the modelview matrix). On the other hand, it means that if you want to use the current modelview matrix, you have to get it only to hand it to `gluProject`.

  In Python, this argument is optional, and uses the current modelview matrix if omitted.

- The next argument is the projection matrix. The same discussion applies to this argument as for the modelview matrix: generality at the price of convenience. Again, Python allows this to default to the current projection matrix.

- The next argument is a smaller matrix that represents the viewport. Recall that the result of projection is a location in "normalized device coordinates." That location is then mapped onto the viewport by translation and scaling. This argument to `gluProject` is again generality at the price of convenience, and again Python lets us get both.

- The last three arguments are not supplied to the function but are *set* by the function. You supply storage locations here, and the `gluProject` function fills them in. In Python, these arguments are not supplied. Instead, you put three variables on the left side of the assigment statement and allow parallel assignment set the value of all of them.

The return value of the C version of this function is either `GL_TRUE` for success or `GL_FALSE` for failure. The Python function probably returns the same value, so just check the value of "x." The man page for this function is succinct and somewhat instructive.

How can you find out the current modelview and projection matrices, not to mention the viewport? It turns out that OpenGL will give this information to you via the function `glGet`, which is a general interrogation function, where you supply a constant specifying what information you want, and an array of the appropriate datatype which is filled in with the information you wanted.

Rather than walk through how to use these functions, let's instead write a function that is convenient but less general than `gluProject`, namely `twProject`. Here's the C code:

```
// computes the window projection of the vertex v and stores it in w.
// That is, it computes where on the screen v will project to.
void twProject(twTriple w, twTriple v) {
    GLdouble v0, v1, v2;
    GLdouble modelview[16];
    GLdouble proj[16];
    GLint view[4];
    GLdouble winx, winy, winz;

    glGetDoublev(GL_MODELVIEW_MATRIX,modelview);
    glGetDoublev(GL_PROJECTION_MATRIX,proj);
    glGetIntegerv(GL_VIEWPORT,view);
    v0=v[0];
    v1=v[1];
    v2=v[2];
    if(gluProject(v0,v1,v2,modelview,proj,view,&winx,&winy,&winz) !=
```

```
        GL_TRUE)
    {
        printf("Error in gluProject for world location (%f,%f,%f)\n",
               v[0],v[1],v[2])
        exit(1);
    }
    twTripleInit(w,winx,winy,winz);
}
```

In Python, it's almost trivial:

```
def twProject(v):
    return gluProject(*v)
```

Notice that all the C function does is get the various matrices and call `gluProject`. However it takes two `twTriples`, one the vertex to project and the other to store the result. Thus, it's very convenient, if not flexible.

Returning to the "pick" problem, an outline of the idea is as follows. The general approach is just the algorithm for finding the smallest value in any sequence; here we are finding the smallest distance from the mouse of the projection each of a set of objects. Thus, the algorithm is:

- initialize "best so far" var using first in sequence
- for each remaining object
  - compute its distance
  - if this distance is smaller than "best so far," replace "best so far"

In C:

```
void myMouseClick( int button, int state, int x, int y ) {
    twTriple windowLocation;

    if( state == GLUT_UP ) return;
    if( button != GLUT_LEFT_BUTTON ) return;
    float bestObj = 0;
    twProject(windowLocation, Objects[0]);    // project first object
    float bestDist = dist(windowLocation, x, y);    // how far is it?
    for( int i=1 ; i< NUM_OBJECTS; i++ ) {
        twProject(windowLocation, Objects[i]);
        float tmp = dist(windowLocation, x, y);
        if( tmp < bestDist ) {
            bestDist = tmp;
            bestObj = i;
        }
    }
    ...
}
```

In Python:

```
def myMouseClick( button, state, x, y ):
    if state == GLUT_UP:
        return
    if button != GLUT_LEFT_BUTTON:
        return
    bestObj = None
    bestDist = 99999
    for obj in Objects:
        windowLocation = twProject(Objects[i])
        tmp = dist(windowLocation, x, y)
        if tmp < bestDist:
            bestDist = tmp
            bestObj = obj
    ...
```

The distance between two window locations can be found by the Pythagorean theorem:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

However, there is an interesting efficiency trick that is worth knowing. We only want to find the smallest distance, but we don't really care what it is. So, instead of finding the smallest distance, just find the smallest *squared* distance. That is, instead of finding the smallest value of d, find the smallest value of d^2, which is just:

$$d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$$

The second calculation avoids a square root operation (which is slow), and is only two subtractions and two multiplications — very efficient.

TW makes both kinds of calculations available:

```
def twPointDistance2( A, B):
    '''returns the squared distance between points A and B'''
    v = twVector(A,B)
    return twDot(v,v)

def twPointDistance( A, B):
    '''returns the distance between points A and B'''
    return math.sqrt(twPointDistance2(A,B))
```

There are more sophisticated schemes for picking more complex objects, but we will stop here.

1.7 General Camera Movement

The coding in the section 1.4 is very limited because the camera is always forced to look down $-z$. How can we do better?

First, we need some terminology, which will turn into variables in our code. For example, we will say that the way that the camera is currently pointing is VPN. This is just a vector that we can maintain with a global array or a set of three global variables. Suppose that we also maintain a vertex VRP and a vector VUP. (Nothing in the following code changes VUP or VPN, but assume that they may change due to some user interaction, such as a keyboard callback, so we can make no assumptions about their values.) We can set up the camera thus:

```
gluLookAt( VRP[0], VRP[1], VRP[2],
           VRP[0]+VPN[0], VRP[1]+VPN[1], VRP[2]+VPN[2],
           VUP[0], VUP[1], VUP[2])
```

Now, we want the ability to move the camera up/down and left/right. Let's start with up/down. To do that, all we need to do is add/subtract VUP from the VRP. Or, more generally, we add a scalar multiple of the VUP vector to the VRP. Recall our parametric equation of a line:

$$P(t) = A + tV$$

Assuming V is a unit vector, this generates a point $P(t)$ that is a distance t from A in the direction of V. We want to move the VRP in the direction indicated by VUP by an amount that depends on the the position of the mouse click. So, the math is really the same.

Our C code (compare this with the code at the beginning of section 1.4) becomes :

```
const float Xspeed = 3.0;   // just an example
const float Yspeed = 4.0;   // just an example

void callback( char key, int x, int y ) {
   x = x - WinWidth/2;
   y = WinHeight/2 - y;

   float moveH = Xspeed*x/(WinWidth/2);  // horizontal distance
   float moveV = Yspeed*y/(WinHeight/2); // vertical distance

   VRP[0] += moveV*VUP[0];
   VRP[1] += moveV*VUP[1];
   VRP[2] += moveV*VUP[2];
   ...
}
```

Our Python code is:

```
Xspeed = 3.0   # just an example
Yspeed = 4.0   # just an example

def callback( key, x, y ):
    x = x - WinWidth/2
    y = WinHeight/2 - y

    moveH = Xspeed*x/(WinWidth/2)     # horizontal distance
    moveV = Yspeed*y/(WinHeight/2)    # vertical distance

    VRP[0] += moveV*VUP[0]
    VRP[1] += moveV*VUP[1]
    VRP[2] += moveV*VUP[2]
```

What about horizontal movement? If we had a vector that pointed to the right (in the camera frame), we could use the very same idea. Such a vector would be perpendicular to both VPN and VUP, so we can find it as the cross product: VPN $\times$ VUP. Assume that we maintain this vector as VRIGHT and update it whenever VUP or VPN changes. The guts of our callback code then becomes:

```
...
VRP[0] += moveV*VUP[0]+moveH*VRIGHT[0]
VRP[1] += moveV*VUP[1]+moveH*VRIGHT[1]
VRP[2] += moveV*VUP[2]+moveH*VRIGHT[2]
...
```

Given these ideas, we can move the camera horizontally and vertically in its own frame. By adding (say) a keyboard callback that would add a scalar multiple of the VPN to the VRP, we can even move the camera forward/back. However, we haven't figured out code that can rotate the camera. We turn to that next.

1.8 Rotating the Viewpoint in 2D

Many computer games are done in the "first person" (including, of course, the ubiquitous first-person shooter games), in which the player sees his/her own viewpoint only (rather than, say, a "bird's eye view" with an icon to mark "his/her" location). Actions that move the player result in a change in the camera's location. In the last section, we discussed how such a player could sidle left or right, go up and down (jump?) and also go move forward and back up. In this section, we discuss how to turn.

For simplicity, we will disallow vertical movement and assume that the camera moves in a plane of constant y value. Thus, all our rotations will be around the y axis, and therefore modify only the x and z coordinates of VUP. (We would also have to update the VRIGHT vector, but that's left as an exercise to the reader.)

Thinking back (many chapters ago) to our discussion of rotation matrices, we can look up the matrix for a rotation around y and find:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Unraveling the equations, this just means that:

$$\begin{aligned} x' &= x\cos\theta + z\sin\theta \\ z' &= -x\sin\theta + z\cos\theta \end{aligned}$$

Notice that the y coordinate doesn't change, since this is a rotation around y.

This means that if we find θ, we can rotate the VPN very easily as follows:

```
void callback( int button, int state, int x, int y ) {
   ...
   // temporary values
   float x = VPN[0];
   float z = VPN[2];
```

```
    // for efficiency and brevity, compute these once
    float c = cos(theta);
    float s = sin(theta);

    VPN[0] =  x*c+z*s;
    VPN[2] = -x*s+z*c;
    ...
}
```

The Python code is nearly identical:

```
def callback( button, state, x, y ):
    ...
    # temporary values
    x = VPN[0]
    z = VPN[2]
    # for efficiency and brevity, compute these once
    c = math.cos(theta)
    s = math.sin(theta)

    VPN[0] =  x*c+z*s
    VPN[2] = -x*s+z*c
    ...
```

Now, we only need to determine theta as a function of the mouse click. One simple way would be to map the x coordinate of the window click to the range [-1,+1] as we did before, and come up with a proportionality constant so that the maximum rotation is, say, 45 degrees.

We can do better, though. Imagine that the user is viewing a scene and clicks on something in it, and the camera rotates so that the thing clicked on is now directly ahead. That would be pretty cool.

For concreteness, consider figure 3. If we can determine the angle θ between the red and blue lines in that figure, we can rotate the camera so that it exactly faces the blue direction, hence the place that was clicked.

If we knew the lengths of the red and green segments, we could find θ by simple trigonometry. In world units, we do know the length of the red segment: it is NEAR, the distance from the center of projection to the image plane (top of the frustum). In pixels, we know the length of the green segment, namely the distance between the mouse click and the center of the window.

If we let X denote the length of the green line in world units, the solution we want is just

$$\tan(\theta) = \mathrm{X}/\mathrm{NEAR} \tag{1}$$

But how can we find the length of X in world units? If we could convert between pixels and world units, we would be done. This sounds like another projection situation, but we can actually do it in an easier way.

Let FW stand for "frustum width," which is the width of the top of the frustum (the magenta line in the figure). FW can be measured in two different ways:

- In window units (pixels), it is the width of the window. Let WW be that value.

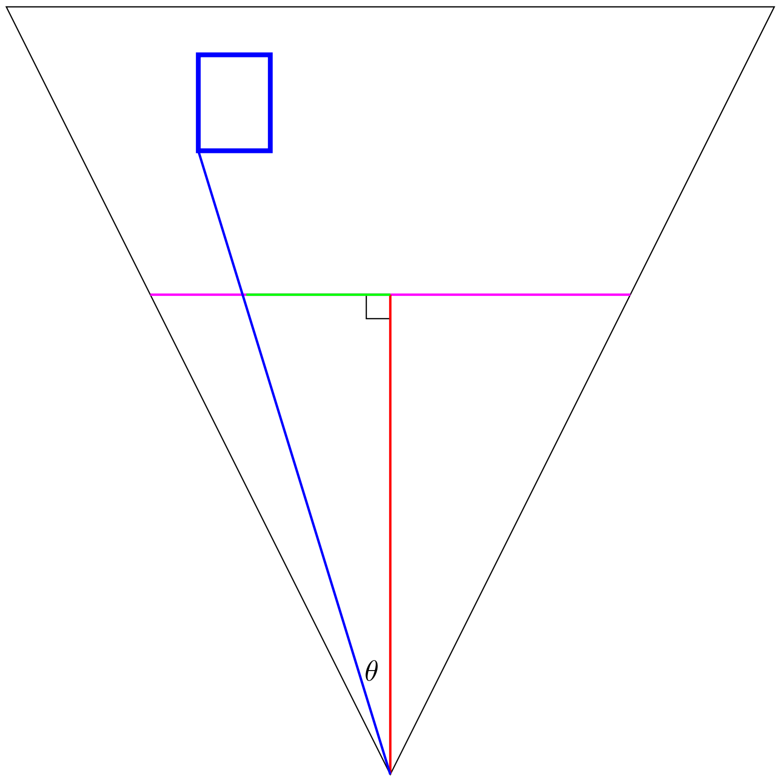

Figure 3: This is a bird's eye view from above the frustum. The magenta line is the image plane or, equivalently, the top of the frustum. If the user clicks on the (projection of) the left front corner of the blue rectangle, the camera needs to turn by the angle between the red line (VPN) and the blue line. Let that be θ. Other labels: (1) the red line is NEAR (in world units); (2) the magenta line is the width of the frustum (in world units) and the width of the window (in pixels); (3) the green line is denoted X (in world units) and wx (in pixels).

- In world units, it is RIGHT - LEFT, where those are the values used to set up the frustum.

In most programs, we set up the frustum using `gluPerspective` rather than `glFrustum`, so we may not have RIGHT and LEFT. But if we use `gluPerspective`, we know that the frustum is symmetrical around zero, so LEFT = -RIGHT and BOTTOM = -TOP. We also know that:

$$\begin{aligned} \tan(\mathrm{FOVY}/2) &= \mathrm{TOP}/\mathrm{NEAR} \\ \mathrm{TOP} &= \tan(\mathrm{FOVY}/2) * \mathrm{NEAR} \end{aligned}$$

but that means that the frustum height (call it FH) is twice this:

$$\mathrm{FH} = 2\tan(\mathrm{FOVY}/2) * \mathrm{NEAR}$$

Of course, the frustum width is just the frustum height multiplied by the aspect ratio, so we get:

$$\mathrm{FW} = 2\tan(\mathrm{FOVY}/2) * \mathrm{NEAR} * \mathrm{ASPECTRATIO} \tag{2}$$

For notational simplicity, let

- "FW" be the frustum width (the magenta line) in world units,
- "WW" be the window width, which is the same line in pixels.
- "X" be the length of the green line in world units, and
- "wx" be the length of the green line in pixels.

Therefore, we can set up a proportion:

$$FW/WW = X/wx \tag{3}$$

Note that equation 3 is not a similar triangles proportion; it is a change-of-units proportion. It's like converting between English and Metric units. We've measured the magenta line using two different rulers, one world units and the other pixels, and we've measured the green line using the ruler for pixels. We can then calculate its length in world units using that proportion.

All the values in equation 3 are known except for X. The value of "wx" is the distance of the mouse-click (the x coordinate) from the centerline of the window, which is just WW/2. To solve for X, we compute:

$$X = wx * FW/WW \tag{4}$$

and, if we don't know FW, we go back to equation 2 and substitute that, to get:

$$X = wx * (2 * \tan(\mathrm{FOVY}/2) * \mathrm{NEAR} * \mathrm{ASPECTRATIO})/WW \tag{5}$$

The numerator looks horrendous, of course, but those values are typically constants that are known at compile-time, so the programmer can pre-compute this. Or they can be computed in the reshape callback, if there is one, since that's when WW might change. For example, in TW the FOVY is 90 degrees, and the tangent of 45 degrees is 1. Also, TW uses a frustum with an aspect ratio of 1. In this case, then, the numerator is just wx*NEAR.

We then combine equations 5 and 1 to finish the calculation. In practice, the computation in C looks something like:

```
void callback( int button, int state, int mx, int my ) {
    int wx = mx - WinWidth/2;
    ...
    float x = (FW*wx)/WinWidth;  // FW is precomputed in reshape
    float theta = atan(x/NEAR);

    // temporary values
    float x = VPN[0];
    float z = VPN[2];
    // for efficiency and brevity, compute these once
    float c = cos(theta);
    float s = sin(theta);

    VPN[0] =  x*c+z*s;
    VPN[2] = -x*s+z*c;
    ...
    glutPostRedisplay();
}
```

Or, in Python

```
def callback( button, state, mx, my ):
    wx = x - WinWidth/2
    ...
    x = (FW*wx)/WinWidth        # FW is precomputed in reshape
    theta = atan(x/NEAR)

    # temporary values
    x = VPN[0]
    z = VPN[2]
    # for efficiency and brevity, compute these once
    c = math.cos(theta)
    s = math.sin(theta)

    VPN[0] =  x*c+z*s
    VPN[2] = -x*s+z*c
    ...
    glutPostRedisplay()
```

There's just one last, tiny detail. The θ shown in figure 3 is a *positive* (counterclockwise) rotation. A click on the other side of the midline would be a *negative* rotation. The angle is computed by the `atan` function that we see in the code above. Since NEAR is always positive, the sign of the angle depends on `ix`, which in turn depends on the sign of `mx`. However, clicks to the left of the midline result in a negative `mx` and therefore a negative θ and similarly clicks to the right of the midline result in a positive `mx` and therefore a positive θ. So the sign is wrong. We could flip the sign anywhere we like, but one easy place would be in the computation of θ:

```
    theta = - atan(x/NEAR)    # flip the sign
```

1.9 Rotating the Viewpoint in 3D

TW has this cool (well, I think it's cool) way to modify the viewpoint by clicking and dragging. The idea is that you imagine that the whole scene is inside a clear sphere (the bounding sphere) and that by clicking and dragging, you are rotating this bounding sphere as if it were a trackball. How is this done?

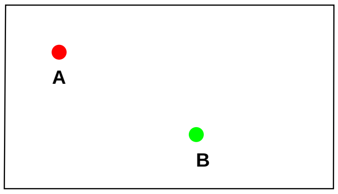

Figure 4: Clicking and dragging from point A to point B on the window

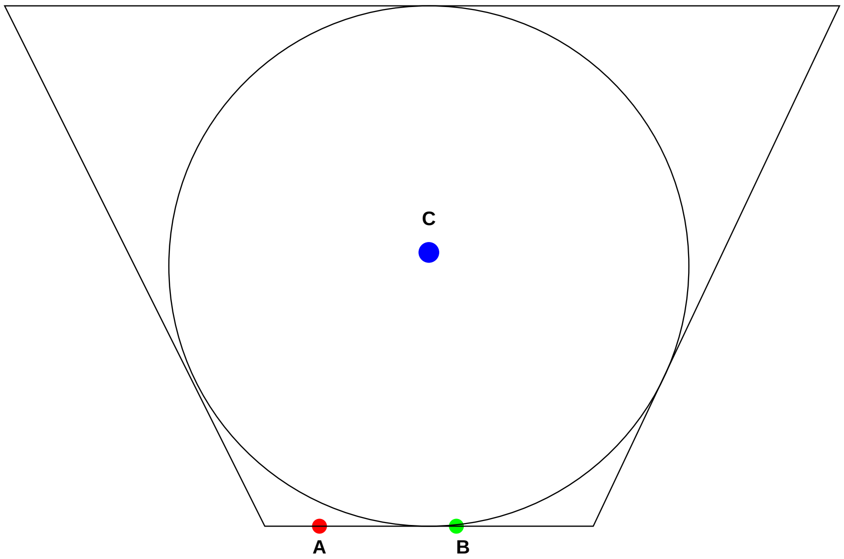

Figure 5: This view from above shows the frustum, the bounding sphere for the scene, and the center of the bounding sphere (the blue point).

Consider figure 4, in which we click the mouse at point A (the red point) and drag it to point B (the green) point. This figure is what the user sees. However, if we view the frustum from above as in figure 5, we can see the center of the bounding sphere, which is the point C. The plane formed by triangle ACB is the plane that we want to rotate the scene in, and the angle of the rotation is exactly the angle ACB.

The axis around which we want to rotate is perpendicular to the plane ACB, and we can find that axis by taking the cross product of the vector v=A-C and the vector w=B-C.

We can find the cosine of the angle of rotation by computing the cosine of the angle between vectors v=A-C and w=B-C. Taking the arccosine of that value gives us the angle we want.

However, there's one problem remaining and it's a big one: The points A and B are only known through mouse locations, and so are in window coordinates. The "real" points A and B are on the top of the frustum and are in world coordinates, not window coordinates.

Consider this: if we had the world coordinates for A and B and we were to *project* them onto the window, they would project to exactly the two mouse locations we got. So, what we really want to do is to *unproject* the two mouse locations.

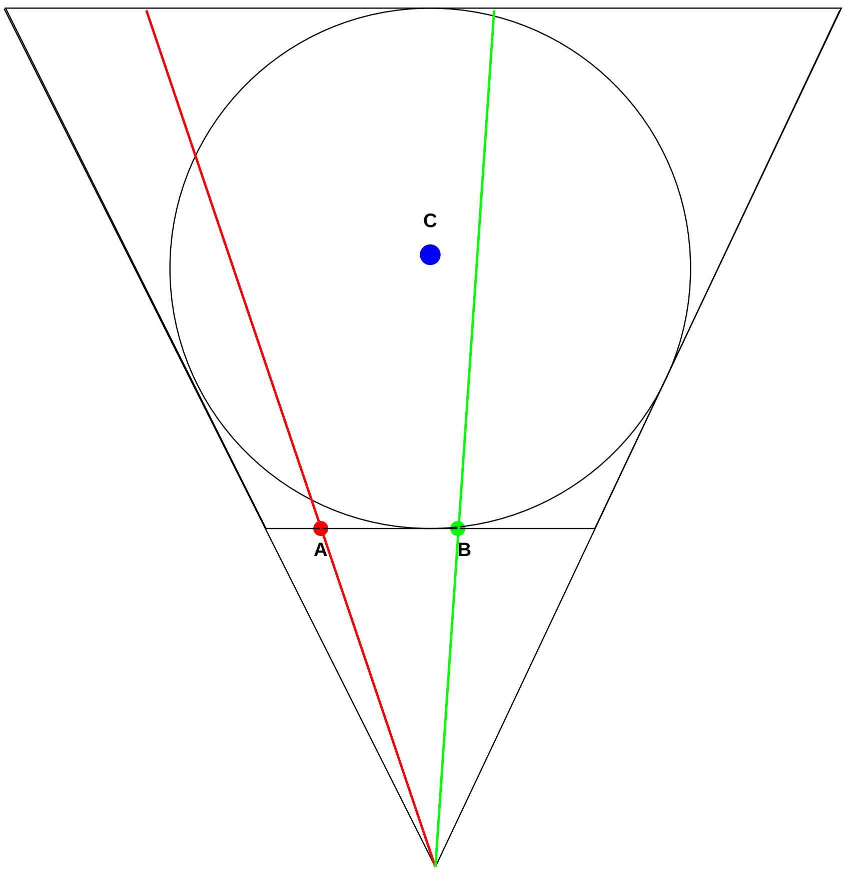

Figure 6: This view from above shows the center of projection and the lines that project to the points A and B.

But what does it mean to "unproject" a point? Projection maps from 3D to 2D: a 3D view volume is mapped onto a rectangle (the top of the frustum). Each point on the rectangle is mapped to by a *line* going through the frustum. See figure 6. Unprojecting a point means to map it to a line: it's inherently ambiguous.

OpenGL resolves that ambiguity by having a z coordinate for each window location: $z = 0$ means the NEAR plane of the frustum and $z = 1$ means the FAR plane of the frustum (values between 0 and 1 lie within the frustum). In our click-and-drag situation, we clearly want A and B to unproject to the NEAR plane, so we will assign them $z = 0$.

Note that this also explains why in the `gluProject` function that we saw earlier, there is a `win_z` argument that `gluProject` assigns. By examining that value, you can tell the depth of the vertex within the frustum. That is, OpenGL retains the depth information even after projection. (This will become important later when we discuss hidden-surface removal.)

With this in mind, the C code becomes almost straightforward:

```
// Rotates the viewpoint by an angle defined by two window locations:
// (Ax,Ay) and (Bx,By).  The y coordinate of the window locations
```

```
// should already have been subtracted from the window height.
void twTrackballOrientation(int Ax, int Ay, int Bx, int By) {
    twTriple A, B, C;
    twTriple v,w,n;
    GLfloat angle;

    if(!BoundingBoxInitialized) {
        printf("Bounding Box not initialized\n");
        return;
    }
    twTriple winA = {Ax, Ay, 0};
    twTriple winB = {Bx, By, 0};
    twUnProject(A,winA);
    twUnProject(B,winB);
    twVector(v,A,BBCenter);
    twVector(w,B,BBCenter);
    angle = acos(twCosAngle(v,w))/M_PI*180.0;
    twCrossProduct(n,v,w);
    twRotateViewpoint(-angle,n);
}
```

In Python:

```
# Rotates the viewpoint by an angle defined by two window locations:
# (Ax,Ay) and (Bx,By).  The y coordinate of the window locations
# should already have been subtracted from the window height.
def twTrackballOrientation(Ax, Ay, Bx, By):
    if not BoundingBoxInitialized:
        print "Bounding Box not initialized"
        return
    winA = (Ax, Ay, 0)
    winB = (Bx, By, 0)
    A = twUnProject(winA)
    B = twUnProject(winB)
    v = twVector(A,BBCenter)
    w = twVector(B,BBCenter)
    angle = acos(twCosAngle(v,w))/M_PI*180.0
    n = twCrossProduct(v,w)
    twRotateViewpoint(-angle,n)
```

There is some additional complexity hidden in `twRotateViewpoint`. We won't go through the code, but we'll summarize. That function computes the VPN and VUP by rotating the current vectors. (It does those rotations by explicitly constructing a rotation matrix around this arbitrary axis and multiplying the vectors.) It then uses the new VPN and the center of the bounding sphere to compute the new VRP from which we will view the scene.

Here's a useful example of rotating around an arbitrary axis rather than one of the major axes!

1.10 Special Keys

Special keys are the non-ASCII keys such as the function keys F1-F12, the arrow keys, and miscellaneous ones such as "home," "insert," and "page up." Also, note that if "num lock" is off, the keys on your keypad are usually equivalents of other special keys, such as "home" or "page up." I would think that the keypad arrow keys would also be enabled, but that doesn't seem to be true for my keypad. Try yours, using the `demos/interaction/ButtonDiscover.cc` program which shows how to react to special keys.

The API is almost exactly like that for regular (ASCII) keyboard callbacks, except that instead of the first argument being a character, it's an integer that is a code for the special key. GLUT defines a bunch of constants that you can compare to the integer to test for different values, so the structure of your callback code (several "if" statements or a big "switch" statement) is the same. Some of the constants are:

```
GLUT_KEY_F1
GLUT_KEY_F2
...
GLUT_KEY_LEFT
...
GLUT_KEY_PAGE_UP
GLUT_KEY_HOME
GLUT_KEY_INSERT
```

See the man page or `/usr/include/GL/glut.h` for the complete list.

What makes these keys special? Not much. Historically, once ASCII won the standards contest, every OS only accepted ASCII characters, so all the "special" keys communicated with your program via an "escape" sequence, that your program had to recognize. In recent decades, things have changed so that your program doesn't have to do the recognition, but instead it's handled at deeper layers. At this point, the only reason to have separate callbacks is the different datatype in the API for the callback function: a character for the ASCII keys and an integer for the special keys.

We have two demos, namely `demos/interaction/NavigateTown` and `pytw/demos/interaction/NavigateYard.py`, that set up the arrow keys to move the camera horizontally and vertically, the page up/down keys to move the camera forward and back, and the "home" and "end" keys to turn by 10 degrees. They also set up the mouse so that clicks will rotate proportionally.

1.11 TW Keyboard Callbacks

TW aims to have a slightly more convenient, self-documenting interface to the keyboard callbacks, using the following function signature:

```
twKeyCallback(char key, twKeyFunc fun, char* doc);
```

The first argument is an ASCII character, such as "t" or "B." Remember that uppercase and lowercase characters are distinct. When that key is pressed, the supplied function is called. The last argument describes what happens and is printed in the transcript window when the user types a question mark. We think of this as "binding" a function and a key together. (Emacs uses the same idea, and one way of customizing Emacs is to modify the keyboard bindings or add new ones. You can find out the current keyboard bindings by using C-h b.)

Note that the same function can be bound to several keys, which is why it's important that the function get the key as an argument, so that it can tell which key was actually pressed. Such a function usually has "switch" statment code as we saw for OpenGL keyboard callbacks in section 1.1.

The keyboard callback function has the same signature as an OpenGL keyboard callback, namely

```
void myKeyFunction(char key, int x, int y);
```

That's because what TW does is set up a "main" keyboard callback that uses the key that was pressed to look up the function that is bound to that key and then calls that function with the same arguments. We can look at how that is implemented.

```
def twKeyboardCallback (key, x, y):
    global twKeyCallbackArray
    keycode = ord(key)
    if keycode > 128:
        return
    fun = twKeyCallbackArray[keycode];
    if fun == None:
        print "No callback for %c (%d)" %(key,keycode)
        return
    fun(key,x,y)
```

What this does is have a table of functions, indexed by the ASCII code for a key, and just looks up the function bound to a key and invokes it. To create the array is pretty easy:

```
KeyCallbacksInitialized = False
twKeyCallbackArray = [ None for x in range(128) ]
twKeyCallbackDoc   = [ None for x in range(128) ]
```

To store something in the array, use:

```
##keyboard settings
def twKeyCallback(key, fun, doc):
    global twKeyCallbackArray, twKeyCallbackDoc
    if type(key) == type('a'):
        key = ord(key)
    if KeyCallbacksInitialized:
        twKeyCallbackArray[key] = fun;
        twKeyCallbackDoc[key] = doc;
    else:
        print "Keyboard Callbacks not yet initialized. "
        print "Call this after twMainInit"
```

And here's the ever-useful help function, bound to the "?" key:

```
def twHelp(key, x, y):
    print '''
This software is designed to help in learning and teaching
about using OpenGL to build 3D graphics programs.
Copyright (C) 2012 Scott D. Anderson
scott.anderson@acm.org
This is free software distributed under the GNU General Public License.
'''
    for i in range(128):
        if twKeyCallbackArray[i] != None:
            if isgraph(i):
                print "%3c: %s" % (i,twKeyCallbackDoc[i])
            elif i == 32:
              print "SPC: %s" %(twKeyCallbackDoc[i])
            elif i == 9:
                print "TAB: %s" % (twKeyCallbackDoc[i])
            elif i == 27:
                print "ESC: %s" % (twKeyCallbackDoc[i])
            else:
                print "^%c : %s" % (i+64,twKeyCallbackDoc[i])
```

We can “seed” that array very easily:

```
def twKeyInit():
    global twKeyCallbackArray, KeyCallbacksInitialized
    for i in range(128): twKeyCallbackArray[i] = None
    KeyCallbacksInitialized = True
    twKeyCallback(27,  twQuit, "Quit")
    twKeyCallback(32,  twStopAndRefresh, "Stop animation and refresh")
    twKeyCallback('+', twSpinCommand, "double spin step")
    twKeyCallback('-', twSpinCommand, "halve spin step")
    twKeyCallback('a', twAxesToggle, "Toggle Axes")
    twKeyCallback('b', twBoundingBoxToggle, "Toggle Bounding Box")
    twKeyCallback('l', twLightingToggle, "Toggle Lighting")
    twKeyCallback('o', twOriginToggle, "Toggle Origin")
    twKeyCallback('s', twShadingToggle, "Toggle Smooth Shading")
    twKeyCallback('q', twQuit, "Quit")
    twKeyCallback('p', twPause, "Pause animation")
    twKeyCallback('r', twReset, "Reset to original screen")
    twKeyCallback('n', twNextFrameCallback, "next frame of animation")
    twKeyCallback('x', twSpinCommand, "Spin around the x axis")
    twKeyCallback('y', twSpinCommand, "Spin around the y axis")
    twKeyCallback('z', twSpinCommand, "Spin around the z axis")
    twKeyCallback('X', twViewCommand, "View from positive X axis")
    twKeyCallback('Y', twViewCommand, "View from positive Y axis")
    twKeyCallback('Z', twViewCommand, "View from positive Z axis")
    ## twKeyCallback('s', twRotViewY, "10 degrees around y")
    ## twKeyCallback('t', twRotViewX, "10 degrees around x")
```

```
## twKeyCallback('P', twPanView, "10 degrees pan around y")
twKeyCallback('?', twHelp,"Help")
twKeyCallback('i', twStartZooming, "start zooming in")
twKeyCallback('S', twSave, "save frame")
```