Assignment 01 - Report.

Yelugoti Mohana Datta

January 26, 2019

Contents

1 Problem Statement:

To Create a 2D graphical application for rendering a touch-screen calculator.

2 Approach:

2.1 Modelling:

Instead of using an algorithm, i modelled the co-ordinates manually.

First, i created a rectangle, then i used following algorithms to create the entire calculator.

```
def translate_left(l):
    tl = []

    a = [l[0][0] - 0.01, l[0][1]]
    b = [a[0] - 0.124, l[1][1]]
    c = [b[0], b[1] + 0.126]
    d = [a[0], c[1]]

    tl.append(b)
    tl.append(a)
    tl.append(d)
    tl.append(c)

    return tl

def translate_right(l):
    tl = []

    a = [l[0][0] + 0.134, l[0][1]]
    b = [a[0] + 0.124, l[1][1]]
    c = [b[0], b[1] + 0.126]
    d = [a[0], c[1]]

    tl.append(a)
    tl.append(b)
    tl.append(c)
    tl.append(d)
    return tl

def translate_up(l):
       tl = []
```

```
        a = [l[0][0], l[0][1] + 0.137]
        b = [l[1][0], a[1]]
        c = [b[0], b[1] + 0.126]
        d = [a[0], a[1] + 0.126]

        tl.append(a)
        tl.append(b)
        tl.append(c)
        tl.append(d)
        return tl

def translate_down(l):
tl = []

a = [l[0][0], l[0][1] - 0.01]
b = [l[1][0], a[1]]
c = [b[0], b[1] - 0.126]
d = [a[0], a[1] - 0.126]

tl.append(d)
tl.append(c)
tl.append(b)
tl.append(a)
return tl
```

2.2 Rendering:

Given that i have all the co-ordinates, i created the vertices using *glVertex3f*.

Now, we use the concepts of *signals* to handle mouse and keyboard actions. So, i defined two functions, one to handle mouse actions and other to handle keyboard actions.

2.3 Effects:

- Moving the calculator when an arrow key is pressed, breaks down to translating the calculator using *glTranslatef*.
- Rotating the calculator when 'R' is pressed, breaks down to rotating the calculator using *glRotatef*.

  - The trick i used here was, i started with a global variable called c_{rta}, and set it to 0. Everytime the key is pressed, the c_{rta} is incremented by 5. So, it gives us the effect of rotation.

- Drawing the red overlay box breaks down to two things.

  - Finding on which key mouse button is pressed.

    For this, i wrote down a function called *getIndex* which gets the index of the key.

    (There are three vectors: $\mathrm{down}_{\mathrm{pos}}$ (which gives bottom left of rectangle), $\mathrm{up}_{\mathrm{pos}}$(which gives top right of rectangle) and $\mathrm{calc}_{\mathrm{ids}}$ (which gives the id of the rectangle (i.e what it is representing.))).

    Now corresponding values of $\mathrm{down}_{\mathrm{pos}}$, and $\mathrm{up}_{\mathrm{pos}}$ vectors with the index found above gives us the bottom left and top right positions.

  - Drawing the rectangle on that key.

    Since, we got the bottom left and top right, drawing a rectangle is nothing but drawing another quad. We just to need to make small modifications to show difference between actual key quad and the one which we are drawing.

    Also to not fill color inside the rectangle, i used
    glPolygonMode($GL_{FRONTANDBACK}$, GL_{LINE});.
    which basically doesn't do interpolation along the diagnols.

- Generating effect of pick and point is described in the *Answers to Questions* Section.

3 Answers to Questions:

3.1 How is rotating each key different from rotating the entire calculator?

When we rotate the calculator we rotate it from the *centre of calculator*, whereas when we rotate a key we rotate it about the *key's centre*.

Initially, when we draw the calculator, we fix an *origin* and draw the calculator, when we want to rotate it, we rotate it about this point. Now, to rotate a key we first *translate* the origin to the centre of the key, so when we do glRotatef(), we will be rotating about this point.

3.2 How does picking the point work? Suppose we zoom in, and rotate the calculator by 90, the pick-point action must still work correctly, i.e, the zoomed-in and rotated key must be moved in the expected orientation. How can pick-point action accomodate these prior transformation?

When right button of mouse is clicked, using *glfwGetCursorPos* , we get the point where mouse was clicked, we then *unProject* the point, which maps window co-ordinates to object co-ordinates, we then check whether this point is inside the calculator, if its present, we get the key in which it is present.

Now, we keep looping for mouse keys, when right button(which was clicked – and as essentially dragging is nothing but holding the right mouse button and moving the mouse) is released, we get the point where it was released and unproject it, and next time when we render the 'calculator key' which we found earlier(when the right mouse button was clicked) we translate to released point and draw the key there.

The important part in all of this is *unProjecting.* This can be done easily with the help of *glut's gluUnProject.* It takes care of all the scenarios (i.e, zoom-in, rotating the calculator).

The documentation of it is gluUnProject. So, by using *glut's gluUnProject* we can accomodate all this transformations.

3.3 Role of *main* function.

First it sets the stage, by setting the *glViewport* which specifies the affine transformations of x and y from normalized co-ordinates to window co-ordinates.

It also sets the *glMatrixMode*, all the operations we do after this will be done on the specified matrix.

It then sets the *glOrtho*, this will affect the parallel projection. (Say we have a 3-D cube, if we place it on a table we can view it from top, from bottom, from sides etc *glOrtho* helps us in that.) With this we basically describe a transformation that produces from parallel projection.

Since, we need to start with a matrix first, we start with an identity matrix. So, each time we set *glMatrixMode* we need to give it a matrix, which can be done by *glLoadIdentity*.

Now, we start an infinite loop, which stops only when window is closed. If mouse/keyboard is pressed, we can handle accordingly using the concept of *signals*, we can configure our windowing system in such a way that if it receives a signal regarding changes in mouse/keyboard state, it calls a function, which then decides what to do. This can be achieved using *glfwSetKeyCallback* for keyboard actions and *glfwSetMouseButtonCallback* for mouse actions.

We then clear the window using *glClear*, so that the window starts with a colour first. Now, we call our drawCalculator() function, which draws the calculator on the screen.

4 Conclusions:

I learnt how complex it is to just render a 2D calculator, and how we need to take care of very minute details and how all parts need to work together, more importantly in a *sequential* way to get what we want.