CS4815 Week07 Lab Exercise

Lab Objective: We will take a look at a program for drawing Bézier curves, as considered in lectures recently. We will modify the interface to permit repositioning of the plot and mouse-based zooming.

Here's a **quick summary** of the tasks:

❶ Copy the main source file for this week's lab from the class directory `~cs4815/labs/week07`

❷ Study the workings of the program as it currently stands

❸ Modify the program so that we can use the mouse to a) reposition the plot around, and b) zoom in/out on regions of the curve

❹ Submit your completed program using the handin command

```
handin -m cs4815 -p w07
```

Editorial Comment:

From talking to people in my office I notice that a lot of people are still compiling their files from the command-line using something along the lines of

```
g++ XXX.cc -l GL -l GLU -l glut -o XXX
```

There is a lot of value in you continuing to do things this way as it reinforces in your minds what is happening in the compilation/build process. However, at some point you may want to switch over the the easier `make` and `Makefile` way of doing things.

In Detail

❶ In last week's lab many people asked about the location the zooming-in should be centered around. In the context of a program for drawing a Bézier curve we will consider this issue today.

Firstly, as always, create a new subdirectory in `~/cs4815/labs/week07`. Now copy this week's lab files from `~cs4815/labs/week07` into the directory of yours. There are two

C++ files there, one for you to modify, `bezier.cc`, and one for you to use to help you with this week's task, `viewer.cc`.

They both have a `main()` function so this indicates that they are both to be compiled into separate executables. Just like several times in the past they can be compiled with the command template:

```
make viewer bezier
```

or, even simpler,

```
make
```

❷ Before getting down to study the Bézier curve drawing program you should run both of them. Things to look at are:

- when running `bezier` note how the scaling responds to window resizing (look for the Reshape callback for the code). Note how the viewport is set to a square of dimensions given by the height of the window. The plot is then scaled according to this viewport with the call to `glOrtho2D()` a few lines of code later.[1] This means that windows that are resized to have very large aspect ratio much of the plot will be hidden.

  Viewport vs. Ortho:

  The roles of `glOrtho2D()` and `glViewport()` are a source of confusion for many. Think of the former as defining the four sides of the clipping window in **world** co-ordinates. The function `glViewport()` independently specifies how this will look by specifying 1) **where** on the screen, and 2) what **dimensions**, the clipping window should be drawn into. So this is a "physical" thing.

  This is unsatisfactory, but the fix to it may not be any better. In order to maintain the plot in the window at all times the viewport's dimensions (both) should be set to the *smaller* of the width and height. Try making this change and play around with resizing the window. It will fix the problem mentioned earlier but it can be disconcerting as you drag the corner of the window around quickly.

- run `viewer` and manipulate the view by *dragging* the left button and middle buttons around. That's all there is to this program but you should follow the code back to where the dragging is accomplished – search for `GLUT_MOUSE_DOWN`, etc. – because this is what we will need for dragging our Bézier plot to reposition it.

The program is a demonstration of computing a Bézier curve on 4 points using samples ("time steps") of 0 to 999. That is, the value $B(t), 0 \leq t \leq 1$ is approximated by 1000

[1] In last week's lab people implemented the zooming function by adjusting the dimensions of the viewport. This was not the correct way to implement the zoom function. While the viewport does not have to match exactly the dimensions of the window – just consider what we have described here – for this situation the solution should be based around the `glOrtho2D()` function.

discrete values in this range. These 1000 values give rise to computing $B(t_i), 0 \leq i \leq 999$ and $t_i = i/1000$ and these are what get plotted. Rather than draw lines between adjacent values these 1000 values are just plotted as points using their function `plotPoint()`. Notice that when the curve gets very steep the smoothness falls off. If this was a line we could sample along the x or y axes according to its slope; anti-aliasing effects could be used also if it was a line we were drawing.

Remember from the lecture that the blending function is so named because $B(t)$ is a blend of the original points with respect to the parameter t and this can be seen in the function `computeBezPt()` where the blended point is built up from an appropriate multiple of each of the 4 control points. This "appropriate multiple" is such that the kth point, p_k, contributes

$$\binom{n}{k} t^k (1-t)^{n-k} p_k$$

The program will never win any awards for efficiency because practically all of the computation takes place every time a display event occurs. For example, such things as initialising the array of control points and computing the binomial coefficients are all computed *within* the function `displayFcn()`. **On the other hand**, it does recognise that the binomial coefficient values, $\binom{n}{k}$, are used over and over again and can be precomputed. The array `C` in the function `bezier()` is initialised with all of these values that will be needed.

❸ The first part of the task is to add the ability to drag the plot around the window. The idea here is to click down with the left button and by moving the mouse the plot should be repositioned *continuously*; releasing the mouse positions the plot in its final resting place. Of course, this operation is more commonly known as dragging. This is implemented in the `viewer` program and you should look carefully at the code there for ideas.

The second part of the assignment is to implement a zoom/unzoom facility. You should reuse some of your code from Week06 to respond to 'z' or 'Z' keyboard events so that zooming happens. If you didn't get this working last week we can provide you with access to a sample program for Week06.

The second part of the assignment is to implement a mouse-based zoom/unzoom facility. My first thought on this was to use a double-click operation *à la* Google Maps. This would allow you to zoom and recentre all in one operation. However, other than building up the detection code one click at a time, there appears to be no easy to detect a double click in OpenGL so we will not pursue that.

Instead, we will use the wheel mouse, a technique that also appears in other graphical user interfaces. The idea is that by holding the control-key down and advancing the wheel we should zoom *in* to the viewing area and, conversely, by holding the control-key down and retracting the wheel we should zoom back *out* from the scene. There are two things you will need to know for this. The first is that Linux treats each direction of the wheel on the mouse almost like another button but since OpenGL doesn't specifically handle wheel mouse events you need to define a few extra macros in the spirit of the OpenGL API. So you will need to put the following lines near the top of your program:

```
#if !defined(GLUT_WHEEL_UP)
#  define GLUT_WHEEL_UP   3
#  define GLUT_WHEEL_DOWN 4
#endif
```

In your mouse handling routine you can now say things like
`if (button == GLUT_WHEEL_DOWN)` Note that UP and DOWN as far as the wheel is concerned has to do with scrolling up or down, not a press or release of a button.

The second thing you will need to consider is how to tell if the control-key is being pressed when, say, a `GLUT_WHEEL_DOWN` event occurs. The function you need here is `glutGetModifiers()` whose return value can be tested against the constant `GLUT_ACTIVE_CTRL` to tell if the control-key was held down when the mouse wheel event was detected.

❹ Using the `handin` command given at the top of the lab sheet please submit your lab exercise by the usual deadline. A sliding scale of penalties will apply until the end of that week.