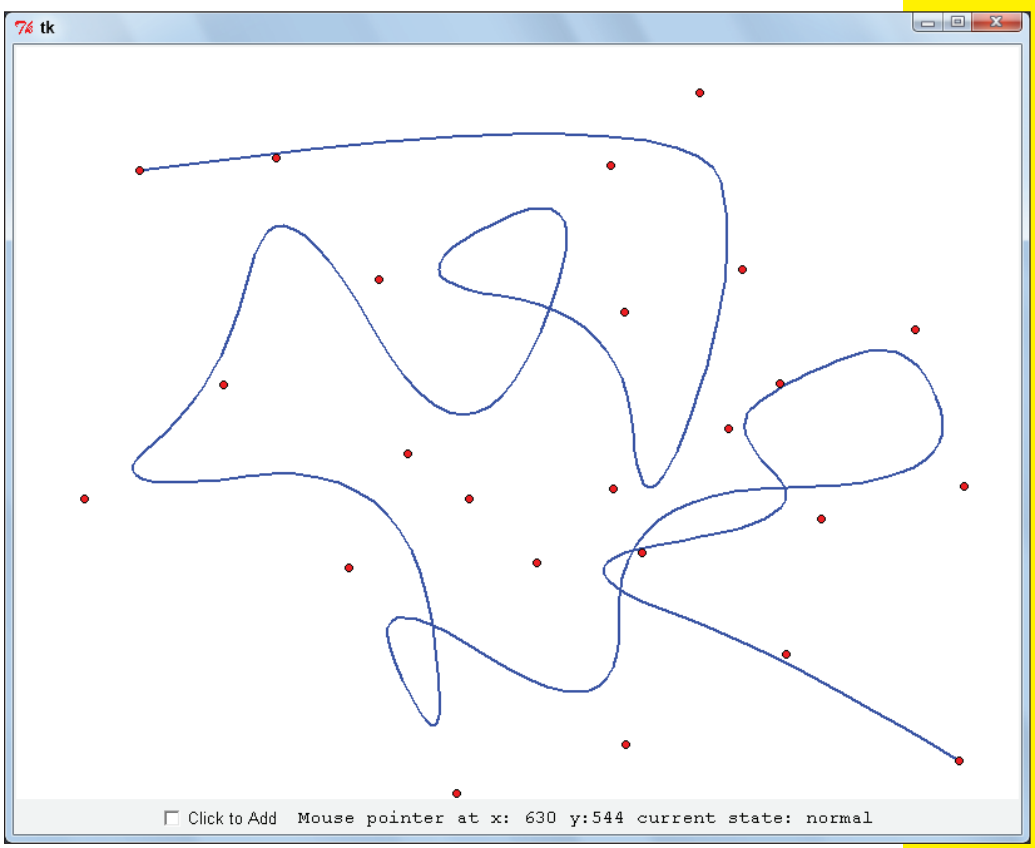
BONUS PROJECT

2

Spline Drawing Program

A *spline* is a smooth-looking line that bends around or between a number of locations called *control points*. You make a spline with math. The good news is that the `Tkinter` module does all the math for you.

You're going to draw with a `Tkinter` widget called a `Canvas`. In the process of drawing and moving around control points, you read about coordinates.



Set Up

In the end, your application will

1. Place control points.
2. Draw a spline defined by those control points.
3. Move control points that are already there and add new control points.
4. Vary the smoothness of the spline.

The first thing to do is to set up a skeleton for the rest of your coding. To cut down on how much you need to type, don't import the whole of `Tkinter`. Instead, import just those objects that you're using. There is a simple syntax to import only specified objects.

If you want to import an object called `object` from a module called `module`, you type this:

```
from module import object
```

To import multiple objects, separate them by a comma:

```
from module import object1, object2
```

For example, if you're going to use the `Tkinter` `Frame` widget, you'd type this:

```
from Tkinter import Frame
```

When you import an object like this, you can refer to the object's name directly. So, after you import `Frame` (like the code just before), you refer to it just as `Frame` (not `Tkinter.Frame`).

The `*` character lets you import every object from a module. Don't! It means that you're *polluting* (messing up) your namespace with object names from the `Tkinter` module. It makes it harder

to understand your code (and can lead to object-naming collisions).

The character * is called a *wildcard* because it stands for everything.

You're probably used to this by now, but just in case, here are some things you need to get started for this project:

1. **Create a new file and call it `spliner.py`.**

2. **Create `Import`, `Class`, and `Main` sections.**

3. **In the `Import` section, import the `Frame` widget from `Tkinter`:**

```
from Tkinter import Frame
```

4. **In the `Class` section, define a class called `SplineDisplay`. Make the class inherit from `Frame`.**

```
class SplineDisplay(Frame):
```

5. **Write a class docstring for `SplineDisplay`.**

```
    """
    This class will display the spline
    """
```

6. **Create a constructor method for `SplineDisplay`. Have it call `Frame's` constructor method.**

   This initializes the `Frame` widget. To do so, `Frame`'s constructor needs to be told what the parent widget is. Pass that as the second argument of the call. To pass this to the `Frame`'s constructor, put a default argument for the parent widget in `SplineDisplay`'s constructor: `def __init__(self, parent=None)`. Then, if you know what the parent widget is, you can pass it when `SplineDisplay` is instantiated.

If it isn't, `parent` defaults to `None` and `Tkinter` will work out the details for you.

```
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
```

7. **In the `Main` section, create an `if __name__ == "__main__"` code block.**

```
if __name__ == "__main__":
```

8. **Inside that code block, instantiate an instance of `SplineDisplay` (`display = SplineDisplay()`) and call the `mainloop` method of that instance.**

```
    display = SplineDisplay()
    display.mainloop()
```

The code you end up with should look like this:

```
"""
Spliner.py
Draw and investigate a spline using Tkinter's Canvas
widget
"""

#### Imports Section
from Tkinter import Frame

#### Class Section
class SplineDisplay(Frame):
    """
    This class will display the spline
    """
    def __init__(self, parent=None):
        Frame.__init__(self, parent)


#### Main Section
if __name__ == "__main__":
    display = SplineDisplay()
        display.mainloop()
```

If you're persnickety, you might not like that there's no docstring for the constructor method. In this case, it's okay not to have one.

Run the code now. An empty `tk` window should pop up when you run this code. The `tk` window is a manageable size because you haven't called `pack()` on the `display` widget. If you had, since `Frame` widgets are invisible, the window would be so small that it would be difficult to find and close.

Flesh Out the Skeleton Application

To draw a decent-sized window, you need

- To resize the widget display so it's a pretty big frame.
- Something to draw on.
- A way to communicate from the program back to you. You did that with `print` in other projects, but hey, you're supposed to be learning GUIs now so you're gonna haffta print it out all GUI style — with a `Label`.

To do these, make the following changes to the `SplineDisplay` constructor:

1. **Pack `SplineDisplay`.**

   For this to work, you have to import the constant `BOTH` from `Tkinter`.

   ```
   self.pack(fill=BOTH, expand=True)
   ```

2. **Add a `Canvas` widget as a child of `SplineDisplay`.**

   You can't use any widget till you `import` it.

   Import the widget `Canvas` from `Tkinter` in the `Imports` section. When you create the canvas you want it to be a child of the `SplineDisplay` widget. However, that's a problem

because the `SplineDisplay` widget doesn't actually exist. You're creating the `Canvas` widget in the course of creating a `SplineDisplay` instance. Therefore, you pass `self` (that is, the instance of `SplineDisplay` that is being created) as the parent widget when you create the `Canvas` widget. At the time the `Canvas` widget is created, the variable `self` refers to `display`, the instance of the `SplineDisplay` class being created. (You named it `display` in earlier code.)

```
self.canvas = Canvas(self)
```

3. **Create two constants: `CANVAS_WIDTH` and `CANVAS_HEIGHT`.**

These set the `config` options `width` and `height` of the `Canvas` widget. You create these constants to avoid using magic numbers in the code.

Magic numbers don't have a meaning attached to them.

Create a `Constants` section to put them in:

```
#### Constants Section
CANVAS_WIDTH = 800
    CANVAS_HEIGHT = 600
```

`Canvas` widgets have two configuration options: `width=` and `height=`. Setting values for these options sets the `Canvas` widget's width and height.

4. **Use the constants you just defined to configure the `Canvas` widget:**

```
self.canvas.config(width=CANVAS_WIDTH,
                   height=CANVAS_HEIGHT)
```

5. **Pack the `Canvas` widget so that it fills the space in its parent widget: `self.canvas.pack(fill=BOTH, expand=True).`**

```
self.canvas.pack(fill=BOTH, expand=True)
```

Your code will look something like this:

```
"""
Spliner.py
Draw and investigate a spline using Tkinter's Canvas
widget
"""

#### Imports Section
from Tkinter import Frame, Canvas, BOTH

#### Constants Section
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600

#### Class Section
class SplineDisplay(Frame):
    """This class will display the spline"""
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack(fill=BOTH, expand=True)
        self.canvas = Canvas(self)
        self.canvas.config(width=CANVAS_WIDTH,
                           height=CANVAS_HEIGHT)
        self.canvas.pack(fill=BOTH, expand=True)


#### Main Section
if __name__ == "__main__":
    display = SplineDisplay(None)
        display.mainloop()
```

Did you remember to update the import to be `from Tkinter import Frame, Canvas, BOTH`?

When you run this code, you should get a largish, gray window on your screen.

Add a `readout` Widget

Every spline needs *control points* that define it. Before you can put down control points, you have to describe where they go. In this section I explain how to describe where they go.

You'll also track the movement of the mouse pointer and print out details. You do that by hooking up a callback to run when the `"<Motion>"` event is received. The reference to `"<Motion>"` means to the mouse's motion. Whenever you move your mouse over the widget, it creates a `"<Motion>"` event.

Read more about the different types of `Tkinter` event at `http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm`.

1. **Make a `Label`.**

   You'll use `Label` to display the details of the `"<Motion>"` events. Import `Label` and create an instance in an attribute called `readout`. Set the `text` option to be `"Readout Label"` and then pack the widget. Add the code for the label at the bottom of the `SplineDisplay` `__init__` method.

```
self.readout = Label(self, text="Readout Label")
self.readout.pack()
```

2. **Create a new method stub in `SplineDisplay` called `track_mouse_motion`.**

3. **Make it accept two arguments: `self` and `event`.**

   The first argument of a method always should be `self`.

   In the method's code, print the event that's received. In the next step you're going to make this method into a callback. When `Tkinter` invokes the callback, it passes a copy of the event to it as the second argument.

```
def track_mouse_motion(self, event):
    """Update the label to tell user about mouse motion"""
    print("got event: %s"%event)
```

4. **Bind the `"<Motion>"` event in the `Canvas` widget to the new method stub.**

When you link an event to a widget it's called *binding* the event to the widget.

You handle events more generally using the `bind` method of the relevant widget. (All `Tkinter` widgets have a `bind` method.) You pass the name of the event that you want to bind to the method as the first argument, and pass the callback that should be invoked as the second argument of the method.

```
self.canvas.bind("<Motion>",
    self.track_mouse_motion)
```

This is the revised code:

```
"""
Spliner.py
Draw and investigate a spline using Tkinter's Canvas
widget
"""

#### Imports Section
from Tkinter import Frame, Canvas, Label, BOTH


#### Constants Section
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600

#### Class Section
class SplineDisplay(Frame):
    """This class will display the spline"""
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack(fill=BOTH, expand=True)
        self.canvas = Canvas(self)
        self.canvas.config(width=CANVAS_WIDTH,
             height=CANVAS_HEIGHT)
```

```
        self.canvas.pack(fill=BOTH, expand=True)
        self.readout = Label(self, text="Readout Label")
        self.readout.pack()
        self.canvas.bind("<Motion>",
          self.track_mouse_motion)

    def track_mouse_motion(self, event):
        """Update the label to tell user about mouse motion"""
        print("got event: %s"%event)

if __name__ == "__main__":
    display = SplineDisplay(None)
    display.mainloop()
```

When you run the code, make sure you can see IDLE's Shell window. After you've done that, move the mouse across the canvas. IDLE's Shell should go nuts whenever the mouse moves. You should see something like this in the readout: `got event: <Tkinter.Event instance at 0x7fe108041248>` over and over again in the IDLE Shell window.

That's repeated hundreds or thousands of times, depending on how energetic you've been with that mouse.

You can see from the printout that each event that the callback receives is an instance of `Tkinter.Event`. (It says so.) From this you can look up the `Tkinter` docs at at `effbot.org` (`http://effbot.org/tkinterbook`) and New Mexico Tech's `Tkinter` reference (`http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html`).

You can get more detailed info on `Tkinter` events quickly if you start up the Python console and type `from Tkinter import Event` and then `print(Event.__doc__)`.

The docstring text goes on a bit. It actually has a lot of useful stuff in there, which you can read about at your leisure.

Track Your Mouse

To get a feel for what's happening with your mouse, change the printout line in `track_mouse_motion` to show these `x` and `y` attributes. Your new `track_mouse_motion` method looks like this:

```
def track_mouse_motion(self, event):
    """Update the label to tell user about mouse motion"""
    print("got event with x: %s and y: %s"%(event.x, event.y))
```

When you run the file, you should get output in the Shell window that looks like this:

```
>>> ================================ RESTART
================================
>>>
got event with x: 11 and y: 449
got event with x: 38 and y: 449
got event with x: 61 and y: 449
got event with x: 76 and y: 447
```

Notice that the readout changes when you move the mouse over the window, and that it *only* changes when the mouse is over the window. The values `x` and `y` are called *coordinates*. To draw anything on the canvas, you need to tell `Tkinter` where in the canvas you want that thing drawn. You do that using these `x` and `y` coordinates.

- The `x` coordinate describes how far from the left edge of the widget the mouse pointer is.
- The `y` coordinate describes how far from the top edge of the widget the mouse pointer is. This means that as you go down the screen, the value `y` gets bigger.

First move the mouse around and see how these coordinates change. Then, move the mouse pointer to each of the four corners of the window and see what the coordinates are. Now you're

going to update the `x` and `y` attributes directly in the text label. Then your GUI won't need to rely on the Shell window.

1. **Create a formatting template that prints out the values of `x` and `y`.**

   The width of the window is `800` and its height is `600`, so both values will be three digits wide. (Just trust me on that.) In Project 10 you saw how to leave a set amount of space in a formatting template. You do that here by using the `%3i` specifier in the formatting string twice.

   ```
   READOUT_FORMAT = "Mouse pointer at x: %3i y:%3i"
   ```

2. **In the `track_mouse_motion` callback, create a string by plugging the event's `x` and `y` attributes into the formatting template.**

   ```
           readout_text = READOUT_FORMAT%(event.x, event.y)
   ```

3. **Use the `config` method of `self.readout` to set its text to the string you just created and delete the earlier code in the `track_mouse_motion` method.**

   ```
           self.readout.config(text=readout_text)
   ```

   There's a slight problem with the alignment of the text in the readout. By default, `Tkinter` uses a proportional font, which means that the space taken up in the printout will vary a little. Fix this problem by changing this to a fixed-width font.

4. **Import the module `tkFont`.**

   ```
   import tkFont
   ```

5. **Just before you create `self.readout` in the constructor, you set the font to use with the following code.**

   You can treat this as a bit of magic. It's preparing a fixed-width font so that the output lines up right.

   ```
           fixed_font = tkFont.nametofont("TkFixedFont")
   ```

6. **On the next line, change the line creating `self.readout` to the following.**

   In it, the option `font=` is telling `Tkinter` to use a specific font.

```
self.readout = Label(self, text="Readout Label",
    font=fixed_font)
```

7. **Run it, sit back, and watch the fun.**

There's a new line in the `Imports` section:

```
import tkFont
```

I added this formatting string to the `Constants` section:

```
READOUT_FORMAT = "Mouse pointer at x: %3i y:%3i"
```

The `SplineDisplay` constructor has two changes:

```
fixed_font = tkFont.nametofont("TkFixedFont")
self.readout = Label(self, text="Readout Label", font=fixed_
    font)
```

The new `track_mouse_motion` function looks like this:

```
def track_mouse_motion(self, event):
    """Update the label to tell user about mouse motion"""
    readout_text = READOUT_FORMAT%(event.x, event.y)
        self.readout.config(text=readout_text)
```

You should have a large, empty, grey window. When you move your mouse around in the window, the `label` widget should change with you. When you're on the left, `x` is a low number. As you move to the right, `x` increases. How high can you make `x` go? See Figure 2-1.

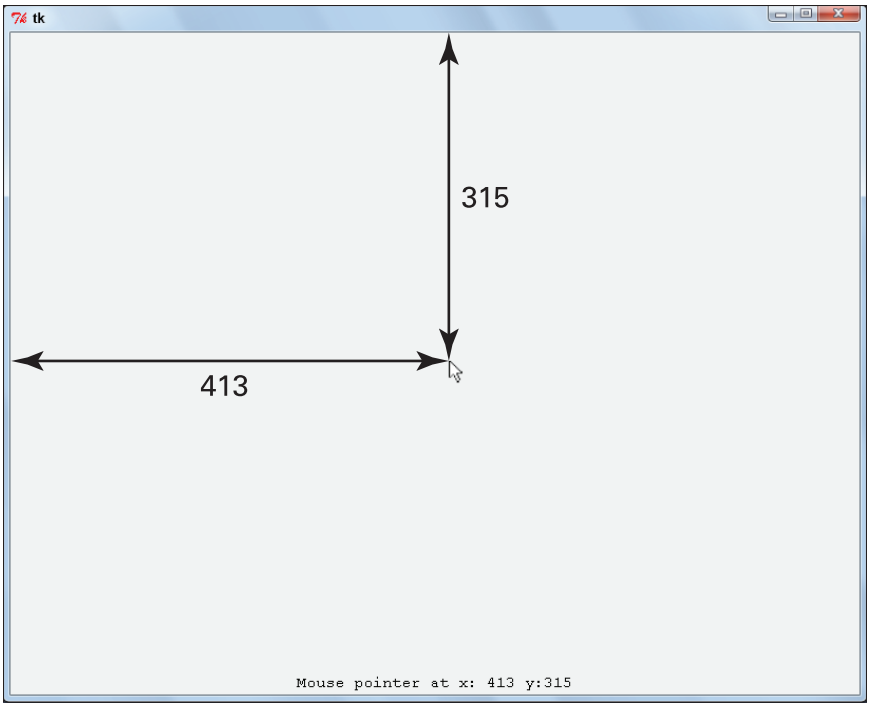


Figure 2-1: Window with `event.x` and `event.y` as received by `track_mouse_motion`.

Cartesian coordinates and Cartesian dualism

When you are using two numbers to describe a location in this way you're using a *Cartesian coordinate system*. This system is named after French philosopher René Descartes. He is known for landing on the Latin phrase "*cogito ergo sum*." It is translated to English as "*I think, therefore I am*." He arrived at his statement by doubting everything. He imagined that he was Neo in The Matrix (or he would have, if he hadn't died many centuries before the film was made). For example, even though he could see stuff, he doubted that it was actually there. He could have been hallucinating it. He doubted everything until he found something that he was unable to doubt. Can you guess what it is?

He couldn't doubt that he was doubting stuff! This is seen as an important statement about our knowledge of what we know. The study of how we know what we know is called *epistemology*. In Monty Python's Pirahna Brothers skit, one of the characters complains that "kids these days" have their heads filled with Cartesian Dualism. That's something else named after Descartes. Cartesian Dualism proposes that we have a mind which is separate from our body and the world of matter. He arrived at his formulation of it as a consequence of his system of doubt.

Draw Your First Line

To draw a line on a `canvas` widget, you use the widget's `create_line` method. The `create_line` method has a heap of options. The most important option is the first argument. The first argument is a list of coordinates. These coordinates are the control points that set where the line is drawn.

The list is read off in x,y pairs. To draw a line from the point where `x = 0` and `y = 0` to the point where `x = 800` and `y = 600`, you'd create a list `[0, 0, 800, 600]` (from the top-left corner to the bottom-right corner of the `Canvas` widget).

To start, you're going to draw a line somewhere random on the canvas to another random ending point. Do it like this:

1. **Import the `random` module.**

```
import random
```

2. **In the `constructor` method, create an attribute called `line_details`.**

   You'll use this attribute to print out information about the line. Set it equal to the empty string.

```
        self.line_details = ""
```

3. **Create a method, called `draw_random_line`, that takes an event as an argument.**

   Because the line is random, you won't use the event. But because you'll set this as a callback, it needs to receive an event argument, even though it doesn't use it.

```
def draw_random_line(self, event):
```

4. **In the `draw_random_line` method, create two random integers between `0` and `800` (the two `x` coordinates) and two random integers between `0` and `600` (the two `y` coordinates).**

```
x = random.randint(0, 800)
y = random.randint(0, 600)
x1 = random.randint(0, 800)
y1 = random.randint(0, 600)
```

5. **Create a list from these four integers: the first `x`, the first `y`, the second `x`, the second `y`.**

   Even though these are in a set order (which suggests using a tuple), you're using a list. This is so that, if you want to, you can later add points or change existing points.

```
point_list = [x, y, x1, y1]
```

6. **Call the `create_line` method on the `canvas` widget, passing this list of integers.**

```
self.canvas.create_line(point_list)
```

7. **Create a `"Line from"` template to use to create the `line_details` attribute.**

   It should add the starting and ending points.

8. **Update the `READOUT_FORMAT` to add a `%s` at the start (where the line details will go):**

```
READOUT_FORMAT = "%s Mouse pointer at x: %3i y:%3i"
LINE_DETAILS_FORMAT = "Line from (%s,%s) to (%s,%s)"
```

9. In the `track_mouse_motion` method, change the format for the `readout_text` to insert the value in the `line_details` attribute.

```
readout_text = READOUT_FORMAT%(self.line_details,
    event.x, event.y)
```

The `readout` widget tells you where the line's supposed to go *from* and *to* and where the mouse pointer is right now.

10. In the constructor, bind the event `"<Button-1>"` to the new `draw_random_line` method.

```
self.canvas.bind("<Button-1>", self.draw_random_line)
```

`"<Button-1>"` is the event generated when the user clicks the left mouse button. Each time you click the canvas, a new line is drawn.

The revised code now looks like this:

```
"""
Spliner.py
Draw and investigate a spline using Tkinter's Canvas
widget
"""

#### Imports Section
from Tkinter import Frame, Canvas, Label, BOTH
import tkFont
import random

#### Constants Section
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600
READOUT_FORMAT = "%s Mouse pointer at x: %3i y:%3i"
LINE_DETAILS_FORMAT = "Line from (%s,%s) to (%s,%s)"

#### Class Section
class SplineDisplay(Frame):
    """This class will display the spline"""
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
```

```
        self.pack(fill=BOTH, expand=True)
        self.canvas = Canvas(self)
        self.canvas.config(width=CANVAS_WIDTH,
            height=CANVAS_HEIGHT)
        self.canvas.pack(fill=BOTH, expand=True)
        fixed_font = tkFont.nametofont("TkFixedFont")
        self.readout = Label(self, text="Readout Label",
            font=fixed_font)
        self.readout.pack()
        self.canvas.bind("<Motion>", self.track_mouse_motion)
        self.canvas.bind("<Button-1>", self.draw_random_line)
        self.line_details = ""

    def draw_random_line(self, event):
        """ Draw a line on self.canvas between two random points
        within the canvas. """

        x = random.randint(0, 800)
        y = random.randint(0, 600)
        x1 = random.randint(0, 800)
        y1 = random.randint(0, 600)
        point_list = [x, y, x1, y1]
        self.canvas.create_line(point_list)
        self.line_details = LINE_DETAILS_FORMAT%(x, y, x1, y1)


    def track_mouse_motion(self, event):
        """Update the label to tell user about mouse motion"""
        readout_text = READOUT_FORMAT%(self.line_details,
            event.x, event.y)
        self.readout.config(text=readout_text)

if __name__ == "__main__":
    display = SplineDisplay(None)
    display.mainloop()
```

11. Click once! Then move your mouse to one end of the line you have created.

It might be tough to do if you have a high-resolution screen.

12. **When you get to the end of the line, make sure that the position it's supposed to be at (from the `line_details` part of the readout) is the same as the mouse position.**

    Then repeat for the other end of the line.

13. **Click again to get a new random line and check the end points.**

Move the Line

When I say *move the line,* I really mean *make it look like you're moving* the line. You do this by deleting the old line, then drawing the new line.

The `Canvas` widget has a method called `delete`. You can pass the constant `Tkinter.ALL` to it and clear the canvas. Import `ALL`:

```
from Tkinter import Frame, Canvas, Label, BOTH, ALL
```

Add `self.canvas.delete(ALL)` in the `draw_random_line` method before you call `create_line`.

```
        self.canvas.delete(ALL)
```

When you run the code, you should get a single line that bounces around the canvas as you click. Yeah! Click that mouse! Click it! Click it!

Show the Points

To click and drag a spline's control points, you first need something on the screen to click (and then drag). You need to draw each control point on the screen. You're going to do that with a red circle where each point is.

`Tkinter` doesn't have circles, but it does have ovals. A circle is just a specific sort of oval. You create a circle by calling `canvas`'s `create_oval` method and passing a square bounding box to it.

A *bounding box* for an object is the smallest rectangle that the object fits into.

You can see one in Figure 2-2.

Make the bounding box by giving it a list with four elements:

- The first two elements are the `x` and `y` coordinates of the top-left corner of the bounding box.
- The last two elements are the `x` and `y` coordinates of the bottom-right corner of the bounding box.

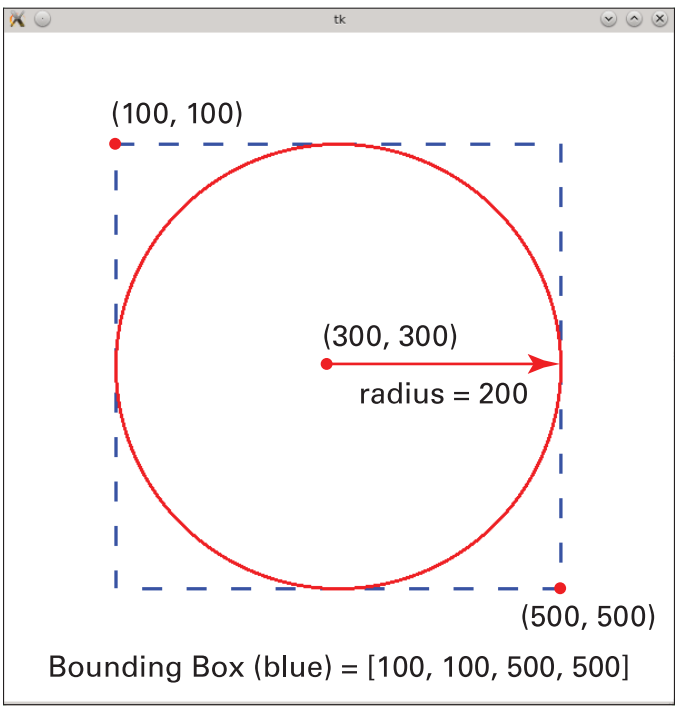


Figure 2-2: A bounding box around a circle.

Follow these steps:

1. **Create a constant called `NODE_RADIUS` that represents the radius of the control point. Set it to `3`.**

```
NODE_RADIUS = 3
```

Change the size later if you don't like it.

2. **Add a new method called `draw_node`.**

   The method should take two numbers as arguments. `x` and `y` are good to use.

```
def draw_node (self, x, y):
```

3. **In the `draw_node` method, use `canvas`'s `create_oval` method.**

   That method takes a bounding box as an argument. First, calculate the bounding box. To calculate `bounding_box`, remember that `draw_node` receives two numbers: `x` and `y`. Assume that `x` and `y` are in the middle of the circle that you're drawing.

   If that's the case:

   - The left side of the circle is at `x-NODE_RADIUS`.
   - The right side of the circle is at `x+NODE_RADIUS`.
   - The top of the circle is at `y-NODE_RADIUS`. (That's minus, because y goes down as you go up the screen.)
   - The bottom of the circle is at `y+NODE_RADIUS`.

```
left = x-NODE_RADIUS
right = x+NODE_RADIUS
top = y-NODE_RADIUS
bottom = y+NODE_RADIUS
```

4. **Work these out and combine them into a list called `bounding_box` with this order: `left`, `top`, `right`, `bottom`.**

```
bounding_box = (left, top, right, bottom)
```

5. **Call `canvas`'s `create_oval` method passing `bounding_box` as the first argument and add `fill="red"` as a second argument.**

The `fill` config option sets the background color of the canvas.

```
self.canvas.create_oval(bounding_box, fill="red")
```

6. **In the `draw_random_line` method, split the variable `point_list` into coordinate pairs.**

7. **For each such pair, call `draw_node`, passing that pair.**

```
pair_list = [(x, y), (x1, y1)]
for x0, y0 in pair_list: # Python will unpack it for you
    self.draw_node(x0, y0)
```

That looks complex, but it's not too bad. If you have trouble, use the debugging techniques from the l33t sp34k3r project and read the error messages. When I first did this, I swapped the `bottom` and `right` values and got all sorts of weird ovals.

This is my new constant:

```
NODE_RADIUS = 3
```

Here's the new method:

```
def draw_node(self, x, y):
    """ Given an x and y coordinate, draw an oval of
    radius NODE_RADIUS, centered at the point (x,y)"""
    left = x-NODE_RADIUS
    right = x+NODE_RADIUS
    top = y-NODE_RADIUS
    bottom = y+NODE_RADIUS
    bounding_box = (left, top, right, bottom)
    self.canvas.create_oval(bounding_box, fill="red")
```

This code is added to the end of `draw_random_line`:

```
pair_list = [(x, y), (x1, y1)]
for x0, y0 in pair_list: # Python will unpack it for you
        self.draw_node(x0, y0)
```

In this code, `pair_list` is a list of pairs. It has only two elements that the `for` loop runs through. The first is `(x,y)` — the starting location of the line — and the second is `(x1,y1)` — the ending location of the line. For each of these, the `for` list unpacks the pair into two separate numbers, stored in the dummy variables `x0` and `y0`. These are passed to `draw_node` (which draws it).

Grab and Move the Control Points

To click and drag control points, you're going to set your line in a default position. This will help make the code easier to debug.

You know that the events you're going to get will have `x` and `y` attributes, so make a custom class that *mimics* (imitates) these attributes. That way you can treat these points as points rather than as two numbers. For example, rather than sending `x` and `y` as arguments to a function, you can send a point `p`.

Some boring set up

Follow these steps:

1. **Create a new class called `ControlPoint`, prior to the definition of `SplineDisplay`.**

   Have its constructor take two arguments — `x` and `y` — and have it assign each of these to an attribute with the same name.

```
class ControlPoint(object):
    """ A class to hold individual control ControlPoints."""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

2. **Create three attributes in `SplineDisplay`'s constructor. Name them as if they were constants, called `POINT_1`, `POINT_2`, and `POINT3`.**

   Instantiate `POINT_1` as a `ControlPoint` with `x` and `y` arguments of `100` and `100`. Do the same with `POINT_2`, with

arguments of `400` and `500` and `POINT_3` as `700` and `200`. Append them, in order, to the `control_points` attribute.

```
self.POINT_1 = ControlPoint(100, 100)
self.POINT_2 = ControlPoint(400, 500)
self.POINT_3 = ControlPoint(700, 200)
self.control_points = [self.POINT_1, self.POINT_2,
    self.POINT_3]
```

3. **Change `draw_node`'s name to `draw_control_point` and change it to accept a single instance of `ControlPoint` (call it `point`) as an argument, rather than values `x` and `y`.**

   Change references to `x` to `point.x` and references to `y` to `point.y`.

```
def draw_control_point(self, point): #if you're gruff just
        call it p
    """ Given a ControlPoint point draw an oval of
    radius NODE_RADIUS, centered at the point
        (point.x,point.y)"""
    left = point.x-NODE_RADIUS
    right = point.x+NODE_RADIUS
    top = point.y-NODE_RADIUS
    bottom = point.y+NODE_RADIUS
    bounding_box = (left, top, right, bottom)
    self.canvas.create_oval(bounding_box, fill="red")
```

4. **Create a new method, called `draw_line`, that takes a list of the line's control points as an argument.**

```
def draw_line(self, control_points):
    """ Given a list of ControlPoints, draw a line defined
        by those
    points.  Draw a node at each of those points"""
```

5. **In the new `draw_line` method, clear the canvas and create an empty list.**

```
    # Clear the canvas
    self.canvas.delete(ALL)

    # Draw line
    point_list = []
```

6. **Iterate through the list of control points that the method received as an argument.**

   For each element, append the values of its `x` and `y` attributes (in that order) to the list you just created.

```
for p in control_points:
    point_list.append(p.x)
    point_list.append(p.y)
```

7. **Call `create_line`, passing `point_list` you generated in the previous step as the first argument.**

```
self.canvas.create_line(point_list)
```

8. **Iterate through the list of control points again and call `draw_control_point` for each of them.**

   Do this *after* you draw the line. Otherwise, the line will appear above the control points.

```
# draw nodes second otherwise line will be on top of nodes
for p in control_points:
    self.draw_control_point(p)
```

9. **Using the first and last points, set the `line_details` attribute.**

```
point1 = control_points[0]
point2 = control_points[-1]
self.line_details = LINE_DETAILS_FORMAT%(point1.x, point1.y,
                                         point2.x, point2.y)
```

10. **In `SplineDisplay`'s constructor, remove the line binding `"<Button-1>"` to `draw_random_line`.**

11. **Add a call to `draw_line`, passing the `control_points` attribute:**

```
##        self.canvas.bind("<Button-1>", self.draw_random_line)
```

    And, a little later

```
self.draw_line(self.control_points)
```

The consolidated changes look like this. First, there is a new class:

```
class ControlPoint(object):
    """ A class to hold individual control ControlPoints."""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

In `SplineDisplay` the constructor looks like this:

```
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack(fill=BOTH, expand=True)
        self.canvas = Canvas(self)
        self.canvas.config(width=CANVAS_WIDTH, height=CANVAS_HEIGHT)
        self.canvas.pack(fill=BOTH, expand=True)
        fixed_font = tkFont.nametofont("TkFixedFont")
        self.readout = Label(self, text="Readout Label",
             font=fixed_font)
        self.readout.pack()
        self.canvas.bind("<Motion>", self.track_mouse_motion)
##        self.canvas.bind("<Button-1>", self.draw_random_line)
        self.line_details = ""
        self.POINT_1 = ControlPoint(100, 100)
        self.POINT_2 = ControlPoint(400, 500)
        self.POINT_3 = ControlPoint(700, 200)
        self.control_points = [self.POINT_1, self.POINT_2,
             self.POINT_3]
        self.draw_line(self.control_points)
```

The `draw_node` method has been replaced by `draw_control_point`, which is very similar:

```
    def draw_control_point(self, point): #if you're gruff
             just call it p
        """ Given a ControlPoint point draw an oval of
        radius NODE_RADIUS, centered at the point
             (point.x,point.y)"""
        left = point.x-NODE_RADIUS
        right = point.x+NODE_RADIUS
        top = point.y-NODE_RADIUS
```

```
        bottom = point.y+NODE_RADIUS
        bounding_box = (left, top, right, bottom)
        self.canvas.create_oval(bounding_box, fill="red")
```

Finally, there's a new method called `draw_line`:

```
    def draw_line(self, control_points):
        """ Given a list of ControlPoints, draw a line
             defined by those
        points.  Draw a node at each of those points"""

        # Clear the canvas
        self.canvas.delete(ALL)

        # Draw line
        point_list = []
        for p in control_points:
            point_list.append(p.x)
            point_list.append(p.y)

        self.canvas.create_line(point_list)

        # draw nodes second otherwise line will be on top of
             nodes
        for p in control_points:
            self.draw_control_point(p)

        point1 = control_points[0]
        point2 = control_points[-1]
        self.line_details = LINE_DETAILS_FORMAT%(point1.x, point1.y,
                                                 point2.x, point2.y)
```

Look for collisions

When an event occurs somewhere within the area assigned to a given object, that's called a *collision.* For example, you're going to check whether a mouse click event happens when the mouse pointer is over a control point. That's an example of a collision. Game programming uses it all the time, at least in 2D games, to check whether (for example) the missile hit an alien.

You're checking to see whether the location of an event (in this case, a mouse click or a mouse motion) collides with a control point. It works like this: If the mouse is over the control point when the user clicks it, then the user can drag the control point and change where the line is drawn. To make the calculations easier, you're not going to be exact, you're just going to be near enough. You're going to check for a collision if the mouse click is within the bounding box around the control point. See Figure 2-3.

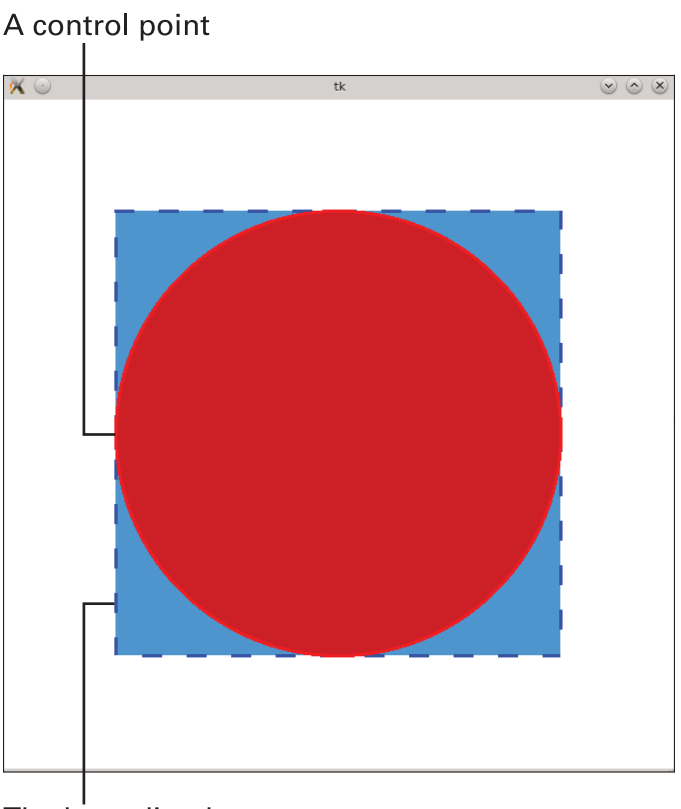


Figure 2-3: Checking for collisions using a bounding box.

You're going to reuse the `<Motion>` event to check whether the mouse has a collision with the two points that make the line. You're going to check for collisions within the `ControlPoint` class itself. This way you run through all your control points and ask the control point itself whether it had a collision.

First, though, you're going to see what a collision is and when it occurs by making your control points go BANG!

1. **In the `ControlPoint` constructor, figre out attributes `left`, `right`, `top`, and `bottom`.**

   You can do it the same way you did in the `draw_control_point` method.

```
self.left = x-NODE_RADIUS
self.right = x+NODE_RADIUS
self.top = y-NODE_RADIUS
self.bottom = y+NODE_RADIUS
```

2. **Create a new method in the `ControlPoint` class called `collide_point`, which takes `self` and `event` as arguments.**

   It's called `collide_point` because it's testing whether a collision has occurred with a single point rather than a line, oval, or rectangle.

```
def collide_point(self, event):
    """ given any object with x and y coordindates
    calculate whether that location (x,y) is within
    the control point's bounding box"""
```

3. **In that method, work out whether the `x`, `y` values in the event correspond to an area within the `ControlPoint's` bounding box.**

   When you instantiate a `ControlPoint`, you already calculate `left`, `right`, `top`, and `bottom`. Just check whether

   - `left` is less than `x`
   - `x` is less than `right`
   - `top` is less than `y`
   - `y` is less than `bottom`

In the `y` coordinate, a higher number is lower on the screen. (It's confusing, I know). If those are all true, return `True`. Otherwise, return `False`. There's a shorthand to do this:

```
return self.left < event.x < self.right and \
        self.top < event.y < self.bottom
```

4. **Rename the `track_mouse_motion` method to be `on_mouse_motion`.**

```
def on_mouse_motion(self, event):
    """Update the label to tell user about mouse motion"""
```

A line binding this method in `SplineDisplay`'s constructor also needs to be updated:

```
self.canvas.bind("<Motion>", self.on_mouse_motion)
```

If you have an event named `event_name`, then the conventional way of naming a callback for the event is `on_event_name`.

5. **In the `on_mouse_motion` method, iterate through each control point, calling its `collide_point` method and passing `event` to that method.**

6. **If the call to `collide_point` method returns `True`, add `BANG!` to the end of the readout.**

If you want to really challenge yourself, change the color of the control point that you've collided with!

```
readout_text = READOUT_FORMAT%(self.line_details,
    event.x, event.y)
for p in self.control_points:
    if p.collide_point(event):
        readout_text = readout_text +" BANG!"
self.readout.config(text=readout_text)
```

7. **In `draw_control_point`, delete the calculations and get the `left`, `right`, `top`, and `bottom` values.**

Get them directly from the point itself. They're now attributes.

```
    def draw_control_point(self, point): #if you're gruff just
            call it p
        """ Given a ControlPoint point draw an oval of
        radius NODE_RADIUS, centered at the point
            (point.x,point.y)"""

        bounding_box = (point.left, point.top, point.right,
            point.bottom)
        self.canvas.create_oval(bounding_box, fill="red")
```

The new `ControlPoint` class looks like this:

```
class ControlPoint(object):
    """ A class to hold individual control ControlPoints."""
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.left = x-NODE_RADIUS
        self.right = x+NODE_RADIUS
        self.top = y-NODE_RADIUS
        self.bottom = y+NODE_RADIUS

    def collide_point(self, event):
        """ given any object with x and y coordindates
        calculate whether that location (x,y) is within
        the control point's bounding box"""
        return self.left < event.x < self.right and \
                self.top < event.y < self.bottom
```

The `draw_control_point` method is now more compact:

```
    def draw_control_point(self, point): #if you're gruff
            just call it p
        """ Given a ControlPoint point draw an oval of
        radius NODE_RADIUS, centered at the point
            (point.x,point.y)"""

        bounding_box = (point.left, point.top, point.right, point.
            bottom)
        self.canvas.create_oval(bounding_box, fill="red")
```

And code tests for collisions in the renamed `on_mouse_motion` callback:

```
def on_mouse_motion(self, event):
    """Update the label to tell user about mouse motion"""

    readout_text = READOUT_FORMAT%(self.line_details, event.x,
         event.y)
    for p in self.control_points:
        if p.collide_point(event):
            readout_text = readout_text +" BANG!"
    self.readout.config(text=readout_text)
```

Finally, the line binding `on_mouse_motion` in `SplineDisplay`'s constructor:

```
self.canvas.bind("<Motion>", self.on_mouse_motion)
```

Run the program and move your mouse around. When the mouse runs over one of the control points, the readout should go `BANG!` Make sure it reports collisions when they occur and *doesn't* report collisions when they don't occur.

To see how the bounding box concept is working, change the value of `NODE_RADIUS` to make it much bigger; then re-run the code. It'll be obvious that you're getting collisions in corners when the mouse isn't over the control point. Near enough is good enough here because it makes the code easier and faster. See Figure 2-4.

Add click and drag

You've clicked and dragged things on a computer before. That's what you want to do with the control points. The mouse motion callback needs to do different things depending on whether you're clicking a control point or dragging it.

Whether you're clicking or dragging is called *state information* because it's information about the current state of the application. (In this case, the user is clicking or the user is dragging.)

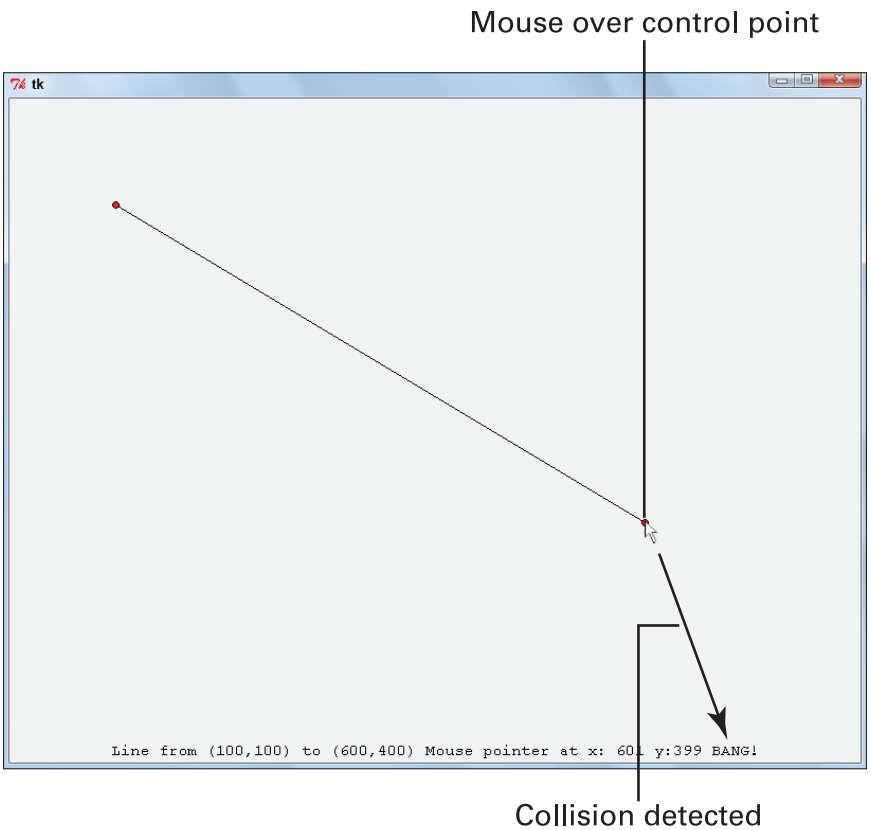


Figure 2-4: Exploding control points.

This application has two states. The first is the normal state and the second is dragging. To keep track of the state of the application, create some new constants to record the states:

1. **Create two new constants: `STATE_NORMAL` and `STATE_DRAGGING`.**

   Assign them the strings `"normal"` and `"dragging a control point"`, in that order.

You can assign numbers here instead (like `0` and `1`). It doesn't matter as long as you don't repeat any of the state identifiers. That said, I think using strings makes it easier to understand.

```
STATE_NORMAL = "normal"
STATE_DRAGGING = "dragging a control point"
```

2. **Create a new attribute in `SplineDisplay` called `state`. Set it to `STATE_NORMAL`.**

   This attribute records the current state of the application.

3. **Create another attribute called `dragging_point` and set it to `None`.**

   When the application is in the dragging state, it'll need to keep track of what control point is actually being dragged. You use `dragging_point` to store this.

```
self.state = STATE_NORMAL
self.dragging_point = None
```

4. **Create a new method in `SplineDisplay` called `on_button_one`.**

   It takes an event as an argument.

```
def on_button_one(self, event):
    """ Action depends on state
    If over a control point and state is normal, start
        dragging,
    change state
       """
```

5. **In `on_button_one`, use `enumerate` to go through all of the control points and check if there's a collision with any control point.**

   If there is, set the state variable to `STATE_DRAGGING` and store the index (the first value returned by `enumerate`) in the `dragging_point` attribute.

   You're saving the location, in the list of control points, of the point being dragged. You do that so you can replace that entry with a new one later.

6. **You need to `break` when you find your first collision.**

   Maybe there are two or more objects that are being collided with (like if two control points are on top of each other). If you

have more than one collision, choose the first one. This choice is arbitrary.

```
for i, cp in enumerate(self.control_points):
    if cp.collide_point(event):
        self.state = STATE_DRAGGING
        self.dragging_point = i
        # this is the index in self.control_points
        break
```

7. **Bind the `"<Button-1>"` event to this new method.**

   I like to bind the button to its callback when I create the callback skeleton. I've delayed it here so it doesn't interrupt the steps describing the callback.

```
self.canvas.bind("<Button-1>", self.on_button_one)
```

8. **Change the `on_mouse_motion` callback so that if the user is dragging a control point, the control point gets updated.**

   To start, comment out the collision-related code. It was only there to test that the collision function was working. Keep the code that updates the readout.

9. **In `on_mouse_motion`, see if the state is `STATE_DRAGGING`.**

   If it is, create a new `ControlPoint` using the event's `x` and `y` coordinates. Then replace the control point with the new one you've just manufactured.

   You can do this because you've kept a record of which point the user is dragging in the `on_button_one` method. If `index` is the index of the point being dragged, and the new `ControlPoint` you've just instantiated is called `new_point`, then you can replace the old point with the new one using the code `self.control_points[index] = new_point`.

   Your control points are stored in a list. That entry in the list gets changed as you drag a control point. Then call `draw_line` using the new control points.

The code for Steps 7 and 8 follows:

```
if self.state == STATE_DRAGGING:
    new_point = ControlPoint(event.x, event.y)
    index = self.dragging_point
    self.control_points[index] = new_point
    self.draw_line(self.control_points)
```

10. **In `SplineDisplay`, create a new method called `on_button_one_release`.**

```
def on_button_one_release(self, event):
    """ Drop a control point that you are dragging """
```

This callback is called when a button is released. If the user is dragging a control point at the time, then the application should stop dragging and leave the control point where the button is released.

To do this in the `on_button_one_release` method, see if the state is `STATE_DRAGGING`. If not, just return from the callback because there is nothing to do. Otherwise, set the state to `STATE_NORMAL` and set the variable recording the point being dragged to `None`.

```
if self.state == STATE_DRAGGING:
    self.state = STATE_NORMAL
    self.dragging_point = None
```

This drops the point (because the code that updates the position only works when the state is `STATE_DRAGGING`). When you change it to `STATE_NORMAL`, the application automatically stops dragging the point. It's a bit magical, but event-driven programming is like that. Take a bit of time to make sure that this works.

11. **Bind the `"<ButtonRelease-1>"` event to this new method.**

```
self.canvas.bind("<ButtonRelease-1>",
    self.on_button_one_release)
```

12. **Delete the `draw_random_line`.**

The consolidated changes from earlier follow.

Here are the additions to the `Constants` section:

```
STATE_NORMAL = "normal"
STATE_DRAGGING = "dragging a control point"
```

I added these new attributes of `SplineDisplay` in its constructor. I added them to the end of the constructor, but I don't think it matters a whole lot where in the constructor you put them:

```
        self.state = STATE_NORMAL
        self.dragging_point = None
```

My new event bindings. The first has the callback changed to `self.on_mouse_motion`, and the other two are new:

```
        self.canvas.bind("<Motion>", self.on_mouse_motion)
        self.canvas.bind("<Button-1>", self.on_button_one)
        self.canvas.bind("<ButtonRelease-1>",
            self.on_button_one_release)
```

The two new callbacks added to `SplineDisplay`:

```
    def on_button_one(self, event):
        """ Action depends on state
        If over a control point and state is normal, start dragging,
        change state
        """
        for i, cp in enumerate(self.control_points):
            if cp.collide_point(event):
                self.state = STATE_DRAGGING
                self.dragging_point = i
                # this is the index in self.control_points
                break

    def on_button_one_release(self, event):
        """ Drop a control point that you are dragging """
        if self.state == STATE_DRAGGING:
            self.state = STATE_NORMAL
            self.dragging_point = None
```

This is the revised `on_mouse_motion` method:

```
    def on_mouse_motion(self, event):
        """Update the label to tell user about mouse motion"""
##        for p in self.control_points:
##            if p.collide_point(event):
##                readout_text = readout_text +" BANG!"
        if self.state == STATE_DRAGGING:
            new_point = ControlPoint(event.x, event.y)
            index = self.dragging_point
            self.control_points[index] = new_point
            self.draw_line(self.control_points)

        # otherwise, just update the readout
        readout_text = READOUT_FORMAT%(self.line_details, event.x,
             event.y)
        readout_text = readout_text + " current state:
             %s"%self.state
        self.readout.config(text=readout_text)
```

Now you should be able to click, drag, and drop the control points. When you do, the line updates. Woohoo!

You can drag the control points outside the window, but if you drop them there you won't get them back.

One of the interesting things about this code is that all the action happens in response to specific events — mouse clicks and mouse motion.

Spline That Line!

Straight lines are sooo boring. They're not awesome looking at all, like splines are.

In the `draw_line` method, add the option `smooth=True` to the call to `create_line`.

```
        self.canvas.create_line(point_list, smooth=True)
```

That's it. Done. `Tkinter` does the rest for you. Simple, huh?

You can delete the `import random` line, since it's not doing anything anymore. Run it. Now you can click and drag around the control points and the spline moves with you.

You can make it a little prettier by

- Changing the canvas background to white (in `SplineDisplay`'s constructor):

```
        self.canvas.config(width=CANVAS_WIDTH,
                           height=CANVAS_HEIGHT,
                           background="white")
```

- Changing the line color to blue and increasing its width a little (in `SplineDisplay`'s `draw_line` method):

```
        self.canvas.create_line(point_list, smooth=True,
                                fill="blue", width=2)
```

Finally, add more control points. You're just going to add some randomly generated ones and it's pretty easy. For more of a challenge, change the `on_button_one` method to drop a control point where you click (if you're not clicking to drag).

Here's some quick code to add some random control points. It goes in `SplineDisplay`'s constructor immediately before the line `self.draw_line(self.control_points)`. It has a magic number too. See Figure 2-5.

```
        for i in range(10):
            x = random.randint(0, 800)
            y = random.randint(0, 600)
            random_point = ControlPoint(x, y)
            self.control_points.append(random_point)
```

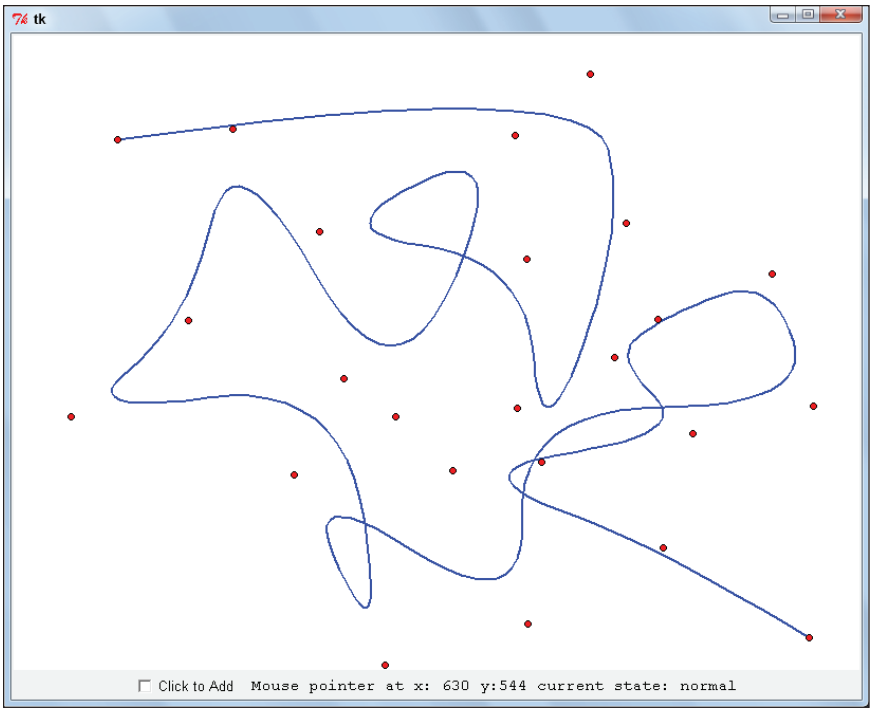


Figure 2-5: This spline has moveable control points.

The Complete Code

Here's the complete code, including some extra random control points, a white background, and a blue line two pixels wide.

```
"""
Spliner.py
Draw and investigate a spline using Tkinter's Canvas
widget
"""

#### Imports Section
from Tkinter import Frame, Canvas, Label, BOTH, ALL
import tkFont
import random

#### Constants Section
CANVAS_WIDTH = 800
```

```
CANVAS_HEIGHT = 600
READOUT_FORMAT = "%s Mouse pointer at x: %3i y:%3i"
LINE_DETAILS_FORMAT = "Line from (%s,%s) to (%s,%s)"
NODE_RADIUS = 3
STATE_NORMAL = "normal"
STATE_DRAGGING = "dragging a control point"

#### Class Section
class ControlPoint(object):
    """ A class to hold individual control ControlPoints."""
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.left = x-NODE_RADIUS
        self.right = x+NODE_RADIUS
        self.top = y-NODE_RADIUS
        self.bottom = y+NODE_RADIUS

    def collide_point(self, event):
        """ given any object with x and y coordindates
        calculate whether that location (x,y) is within
        the control point's bounding box"""
        return self.left < event.x < self.right and \
                self.top < event.y < self.bottom


class SplineDisplay(Frame):
    """This class will display the spline"""
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack(fill=BOTH, expand=True)
        self.canvas = Canvas(self)
        self.canvas.config(width=CANVAS_WIDTH,
                           height=CANVAS_HEIGHT,
                           background="white")
        self.canvas.pack(fill=BOTH, expand=True)
        fixed_font = tkFont.nametofont("TkFixedFont")
        self.readout = Label(self, text="Readout Label",
             font=fixed_font)
        self.readout.pack()
```

```
        self.canvas.bind("<Motion>", self.on_mouse_motion)
        self.canvas.bind("<Button-1>", self.on_button_one)
        self.canvas.bind("<ButtonRelease-1>", self.on_button_one_
            release)
##        self.canvas.bind("<Button-1>", self.draw_random_line)
        self.line_details = ""
        self.POINT_1 = ControlPoint(100, 100)
        self.POINT_2 = ControlPoint(400, 500)
        self.POINT_3 = ControlPoint(700, 200)
        self.control_points = [self.POINT_1, self.POINT_2, self.POINT_3]
        for i in range(10):
            x = random.randint(0, 800)
            y = random.randint(0, 600)
            random_point = ControlPoint(x, y)
            self.control_points.append(random_point)
        self.draw_line(self.control_points)
        self.state = STATE_NORMAL
        self.dragging_point = None

    def on_mouse_motion(self, event):
        """Update the label to tell user about mouse motion"""

        if self.state == STATE_DRAGGING:
            new_point = ControlPoint(event.x, event.y)
            index = self.dragging_point
            self.control_points[index] = new_point
            self.draw_line(self.control_points)

        readout_text = READOUT_FORMAT%(self.line_details,
            event.x, event.y)
        self.readout.config(text=readout_text)

    def draw_control_point(self, point): #if you're gruff
            just call it p
        """ Given a ControlPoint point draw an oval of
        radius NODE_RADIUS, centered at the point
            (point.x,point.y)"""

        bounding_box = (point.left, point.top, point.right, point.bottom)
        self.canvas.create_oval(bounding_box, fill="red")
```

```
def draw_line(self, control_points):
    """ Given a list of ControlPoints, draw a line defined by those
    points.  Draw a node at each of those points"""

    # Clear the canvas
    self.canvas.delete(ALL)

    # Draw line
    point_list = []
    for p in control_points:
        point_list.append(p.x)
        point_list.append(p.y)

    self.canvas.create_line(point_list, smooth=True,
                            fill="blue", width=2)

    # draw nodes second otherwise line will be on top of nodes
    for p in control_points:
        self.draw_control_point(p)

    point1 = control_points[0]
    point2 = control_points[-1]
    self.line_details = LINE_DETAILS_FORMAT%(point1.x, point1.y,
                                             point2.x, point2.y)

def on_button_one(self, event):
    """ Action depends on state
    If over a control point and state is normal, start dragging,
    change state
    """
    for i, cp in enumerate(self.control_points):
        if cp.collide_point(event):
            self.state = STATE_DRAGGING
            self.dragging_point = i
            # this is the index in self.control_points
            break

def on_button_one_release(self, event):
    """ Drop a control point that you are dragging """
    if self.state == STATE_DRAGGING:
```

```
                self.state = STATE_NORMAL
                self.dragging_point = None

if __name__ == "__main__":
    display = SplineDisplay(None)
    display.mainloop()
```

Summary

You did a lot of important stuff:

- Sized a canvas widget, used the `x` coordinate to talk about left-right location, and used the `y` coordinate to talk about up-down location.
- Determined what `x` and `y` go up to, and that those numbers are determined by the size of the window.
- Discovered that a number for `x` and a number for `y` can be used together to identify each and every location within a window or screen.
- Used `Tkinter` event objects.
- Changed mouse motion events into mouse position info.
- Reacted to mouse clicks.
- Detected collisions and used collision detection to grab and move control points.
- Tested the form `a<x<b`.
- Made friends with the `Canvas` widget.
- Drew lines and ovals and discovered how to change their color with `fill`.
- Had a more interesting example of an event-driven program.