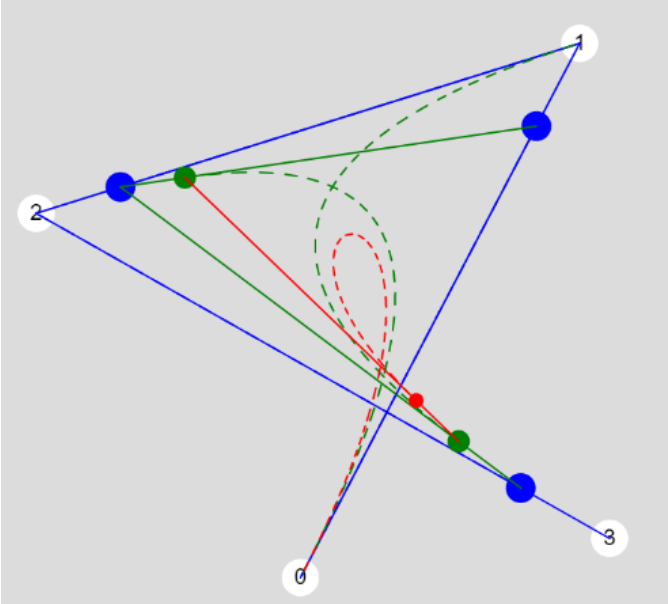
Bezier Curves & Buttons

Part 2: Buttons

This is the second part of the "Bezier Curves & Buttons" project. In part 1, we created an animation showing the construction of a cubic Bezier curve, along with all the construction lines and moving points. In this part we're going to build and use buttons to modify our sketch, allowing the user more direct control over which elements to show, and how.

Recap



Our animation shows four points in white, randomly generated within a region of the screen, three blue 'movers' which follow a straight line path between the white fixed points, then two second generation movers (in green) which trace a (moving) straight line between adjacent pairs of blue (first generation) movers. Finally, the red point traces a line between the two second generation movers. Each moving point traces out its line in the same time interval, so that the red point starts at the first fixed point, and ends at the last one, tracing out as it goes the cubic Bezier curve.

Note that this curve can also be generated directly with the built-in *bezier* function.

The Button class

In part 1, we designed our *Dot* and *Mover* classes by first considering their usage in plain English:

Object	Description	Usage	Methods / attributes
Button	A rectangle on the screen with a text label within which, upon being clicked, toggles various elements within the animation.	We might have buttons for randomly repositioning the dots, or showing or hiding different levels of mover, or their trails.	*Display:* will need to be visible. May change appearance or text based on its state. *State:* should keep track of whether it is on or off. *Toggle:* a simple function which turns the switch on if off, and off if on. *MouseOver:* some way of checking whether or not the mouse is over the button, to combine with the built-in mouse functions to determine whether the button has been clicked.

Making the buttons

It may feel backwards, but working out what distinctive information we'll need to provide to each separate button object is often a good starting point for designing the class. Here's the *createButtons* function:

```
function createButtons(){
  buttons.push(new Button("Hide",    "Show",     "blue",     10,  10, 50, 50));
  buttons.push(new Button("Hide",    "Show",     "green",    10,  70, 50, 50));
  buttons.push(new Button("Hide",    "Show",     "red",      10, 130, 50, 50));
  buttons.push(new Button("Lines",   "No lines", "grey",     10, 190, 50, 50));
  buttons.push(new Button("Refresh", "Refresh",  "darkgrey", 10, 250, 50, 50));
  buttons.push(new Button("Path",    "No path",  "purple",   10, 310, 50, 50));
}
```

I've added spaces to improve readability. The arguments give the button text (for both states: on and off), a background colour for the button, and parameters for its position (*x* and *y*) and size (*w* and *h*).

The Button class

```
class Button{
  constructor(onText, offText, color x, y, w, h){
    this.on = true;
    this.onText = onText;
    this.offText = offText;
    this.color = color;
    [this.x, this.y, this.w, this.h] = [x, y, w, h];
  }
```

The parameters which will vary from case to case are: text to display when the button is 'on' or 'off', the button colour and its position and size. Notice that we can set many variables at once using arrays.

```
  mouseOver(){
    return this.x < mouseX && mouseX < this.x + this.w &&
      this.y < mouseY && mouseY < this.y + this.h;
  }
```

The *mouseOver* method is a helper function we'll use to decide if the button has been clicked.

```
  toggle(){
    this.on = !this.on;
  }
```

The *toggle* method simply turns the switch on or off (*true* or *false*).

```
  display(){
    fill(this.color)
    noStroke();
    rect(this.x, this.y, this.w, this.h);
    fill("white");
    let txt;
    if (this.on) {txt = this.onText}
    else {txt = this.offText}
    text(txt, this.x + this.w/2, this.y + this.h/2);
  }
}
```

In *display*, we draw the button as a rectangle, then use white for the text.
I've condensed the *if else* blocks since they're very short, and read well on just one line each.
We've used the centre of the box for text since…

```
function setup() {
  createCanvas(400, 400);
  textAlign(CENTER, CENTER);
  createDots()
  createButtons()
}
```

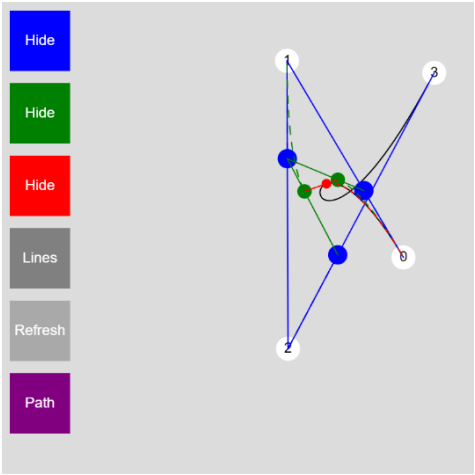
…in *setup* we used the *textAlign* function to centre text.

Note that text looks strange if you don't use *noStroke*, so if you want a border around your button, don't forget to change *Stroke* back to *noStroke* before rendering text.

Displaying the buttons

```
function draw() {
  background(220);
  t += 0.005;
  t -= int(t);
  for (let button of buttons){
    button.display();
  }
```

Near the start of the *draw* loop I've inserted another *for* loop, to ensure our buttons are displayed. If you recall, we made six of them (before even writing the class code), so they should all appear down the left-hand side of the canvas.



Note that clicking on them won't do anything yet – for that we'll need the built-in mouse functions…

Clicking the buttons

It looks promising, but so far nothing happens when you click on a button. For that, the most efficient approach is not to constantly check where the mouse is (unless you also want a mouse-over effect, such as the button changing colour or text going bold when you hover over a button). Instead, we can simply make use of p5's built-in *mousePressed* function, which 'listens' for a mouse click. Essentially this means that a JavaScript array of 'events' is constantly being added to whenever the user hits a key or moves the mouse (or taps a touchscreen), and *mousePressed* taps into a loop running behind the scenes, listening out for the relevant 'event' (in this case, a click). When the mouse is pressed, the code within the function is executed.

```
function mousePressed(){
  let [x, y] = [mouseX, mouseY];
  for (let button of buttons){
    if (button.mouseOver(x, y)){
      button.toggle();
      return false;
    }
  }
  return false;
}
```

This code creates variables *x* and *y* for the built-in *mouseX* and *mouseY*, so that p5 doesn't have to request the mouse coordinates repeatedly while looping through the buttons (and also avoids any issues that a fast moving mouse across multiple buttons might potentially cause!)

We loop through each button, using its *mouseOver* method to see if the mouse is over that particular button. The *return false* that follows ensures that once a button has been found, the loop doesn't have to continue.

Note that different browsers may interpret this function differently, so the p5 documentation suggests that we always have a return value, hence the final *return false* statement (which only occurs if the user clicks somewhere which isn't on any of the buttons).

Making the buttons do something

Now the buttons not only exist and are visible, but by clicking on them we can change their state (on or off, represented by *this.on* being either *true* or *false*). However, unless we make the other elements of our sketch depend upon the state of these buttons, nothing (other than the button text itself) will change when we click them. Time to change that. The simplest way is to take each button in turn, work out exactly what we intend it to do, and build in some *if* blocks to our *draw* loop code.

Button 0

When clicked, this should hide the first generation movers (the blue dots). Note that this button (like the others) is, by default 'on', so the 'Hide' text tells the user what it will do if clicked, not it's current state.

```
for (let mover of movers1){
  mover.update(t);
  mover.showTrail();
  if (buttons[0].on){mover.display()}
}
```

Button 1

When clicked, this should hide the second generation movers (the green dots).

```
for (let mover of movers2){
  mover.update(t);
  mover.showTrail();
  if (buttons[1].on){mover.display()}
}
```

Button 2

When clicked, this should hide the third - and final - generation movers (the single red dot).

```
for (let mover of movers3){
  mover.update(t);
  mover.showTrail();
  if (buttons[2].on) {mover.display()}
}
```

Button 3

This button toggles lines for the movers. It makes sense that if the mover is hidden, its lines are too, so its only if the movers are displayed that this is relevant. Note that displaying of lines happens within the *display* method of individual movers. Fortunately we built this functionality into the *display* method already:

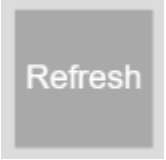
```
display(lines=true){
  if (lines){
    stroke(this.color);
    line(this.start.x, this.start.y, this.end.x, this.end.y);
  }
  noStroke();
  fill(this.color);
  circle(this.x, this.y, 2*this.r);
}
```

This code (the *display* method within the *Button* class) takes either a *true* or *false* to determine whether a line is drawn in addition to displaying the mover itself.

```
for (let mover of movers1){
  mover.update(t);
  mover.showTrail();
  if (buttons[0].on){mover.display(buttons[3].on)}
}
```

This code (in the *draw* loop) uses the state of button 3 to tell the relevant *display* method whether to draw lines. Use exactly the same input to the *display* methods for *movers2* and *movers3*.

Button 4



This button should allow the user to restart the simulation, but unlike simply refreshing the whole sketch it should preserve the settings of the other buttons. Essentially, regenerate the dots and movers. Fortunately, we already created the code to overwrite existing dots and regenerate random dots and movers:

```
function createDots(){...}
```

```
function draw() {
  background(220);
  if (!buttons[4].on){      // <--
    createDots();           // <--
    t = 0;                  // <--
    buttons[4].toggle();    // <--
  }                         // <--
  t += 0.005;
  t -= int(t);
```

Within the *draw* loop, before doing anything else, we check to see if the 'refresh' button has been pressed. If it has, we run the *createDots* function just like we did at the very start. We also reset *t* to zero so the animation begins at the beginning once more.

Also – importantly! – we turn the button off. Otherwise every time the *draw* loop runs it'll reset again.

Button 5

Much like using buttons 0, 1 and 2 to decide if we should display the buttons, we can use button 5 to decide if we should show the trail:

You can duplicate this code for the other movers or just delete them so only red is shown.

```
for (let mover of movers3){
  mover.update(t);
  if (buttons[5].on) {mover.showTrail()};
  if (buttons[2].on) {mover.display(buttons[3].on)}
}
```

What's next?

Now you have what's known as 'boiler-plate': a chunk of code that does something useful and reusable. If you want to incorporate buttons into any future (or previous) sketches, you can just copy and paste your *Button* class code. Combined with the built-in *sliders* we've already seen, we now have the capacity to give the user significant control over the various features of the sketch.