Bezier Curves & Buttons

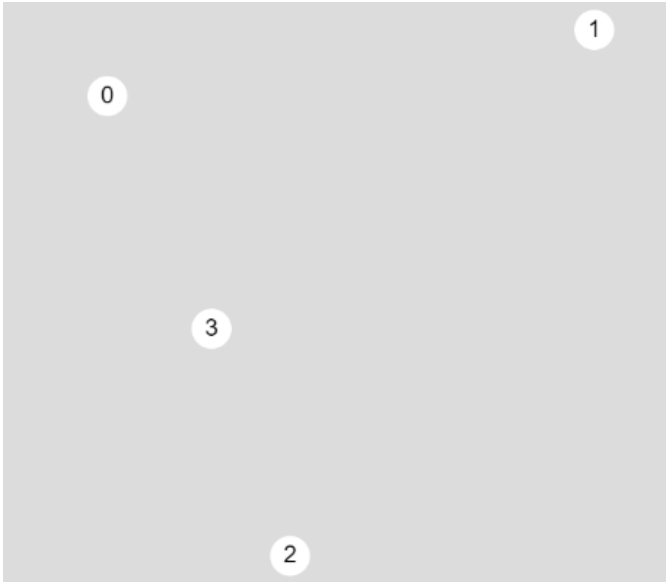
Part 1: Bezier Curves

In this project we're going to do two things: construct Bezier curves from scratch, and build and use buttons in a p5 sketch. It'll be split over two sessions, so be sure to save your work and we'll add button functionality to the sketch in part 2.

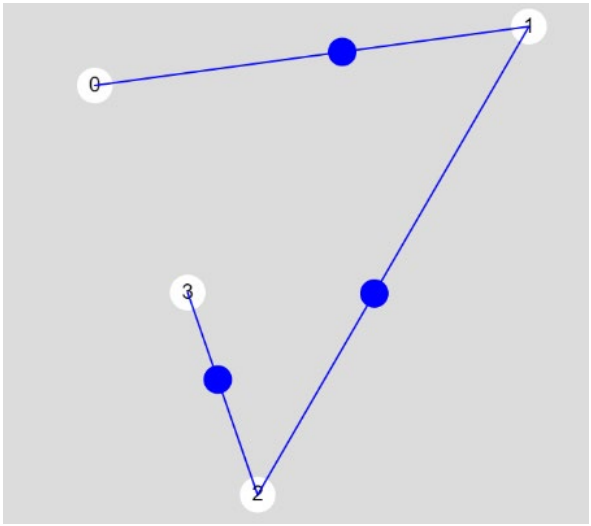
So what exactly is a Bezier curve? Pierre Bezier came up with the idea in the 1960s and 1970s while working for Renault, and it is an elegant solution to the problem of constructing an efficient smooth path between points. If you've ever drawn or viewed a curve created by a computer, you will have already encountered these. In fact, the words you're reading now are rendered using these concepts: smooth curves at any resolution, built from just a handful of 'control points.

The big idea

Consider four random points on the plane:

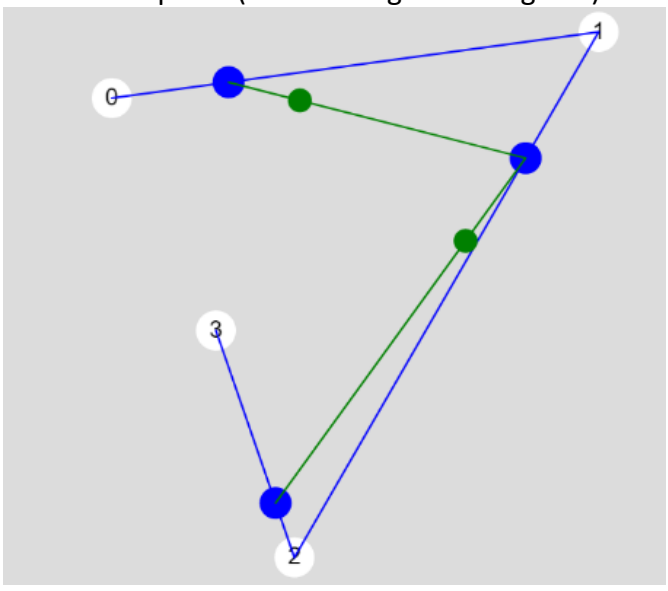


Now imagine three points which move smoothly between each pair of adjacent points, along a straight line between them:

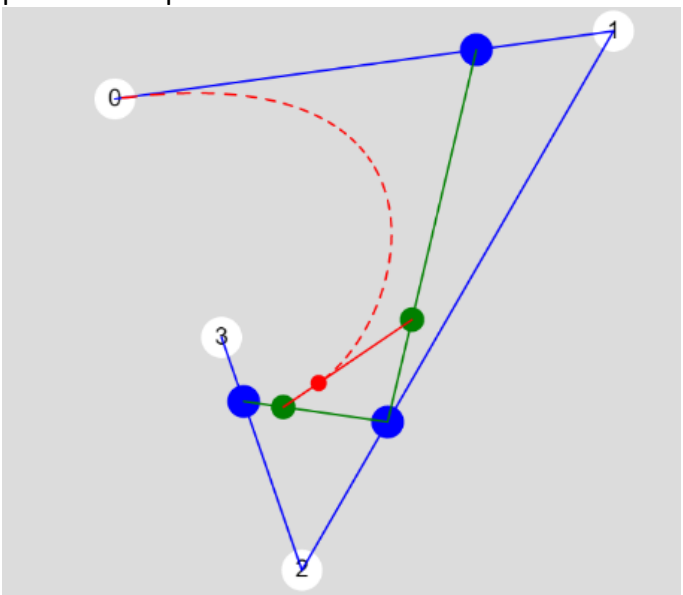


Points move according to a global variable t for time: at $t = 0$ they are at their start, and at $t = 1$ they have reached their end. This means that the greater the distance the faster the moving point will move.

Next, consider lines between the three moving points, and corresponding moving dots which trace those paths (also moving according to t):



Finally, one last point, tracing the same type of path, but this time between the latest two moving points. This point traces out a Bezier curve:



Planning our Classes

The more complicated the sketch , the more important it becomes to plan ahead. What objects would be useful to encode in their own classes? What methods and attributes should they have? Starting with a plain English description and some notes on where and how the objects may be used helps inform this.

Object	Description	Usage	Methods / attributes
Dot	The four original points of our animation. Could appear randomly on the screen, or be placed by the user.	To be shown on the screen, and moved around by the user potentially. Referenced by the moving points which will take the position of these dots as their start and finish points.	*Display:* eg a white circle with text inside showing their order. *Move:* some way for the user to move them. Maybe use built-in mouse functions. *x* and *y*: the moving points will need to have access to their positions.
Mover	A point which moves along a predetermined path between either fixed or moving start and end points. The path itself will change if its start and end points are themselves movers).	To be shown on the screen, including perhaps a line showing the path it follows. Needs to access the position of either a dot or another mover, which will determine the start and end points of its path at any moment.	*Display:* perhaps differently coloured circles depending on the level of mover (eg first level blue, second level green and final level red). *Move:* will need to move automatically based on their start and end points and a global *t* variable. *Trail:* an option to display a trail. This may only ben shown for the final mover, or it could be enabled for any. May be complex enough to warrant its own object.
Button *(more on this in part 2)*	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.

If you are just embarking on a project, you are unlikely to know in advance every detail of every object you will end up making. That's why refactoring (rewriting some or all of a program, often completely replacing your first draft) is such a common practice of developers. The important thing is that you set yourself up for as much scalability as possible so you avoid doing too much work. The encapsulation will also help you to clarify the flow of your project.

Next, we'll look at the *setup* and *draw* sections of the code, and return to the details of the classes once we know how we're going to create them and use them.

Program flow

```
let t = 0;
let dots = [];
let movers1 = [];
let movers2 = [];
let movers3 = [];
let buttons = [];

function createDots(){}

function createButtons(){}

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

function draw() {
  background(220);
  t += 0.005;
  t -= int(t);
  for (let dot of dots){
    dot.display();
  }
  for (let mover of movers1){
    mover.update(t);
    mover.display();
  }
  for (let mover of movers2){
    mover.update(t);
    mover.display();
  }
  for (let mover of movers3){
    mover.update(t);
    mover.display();
  }
}
```

We first set some global variables: *t* for time, then arrays to hold the dots, movers and buttons. Note that the different generations of mover (first generation based on dots, second generation based on first, etc) are created in separate arrays so we can work with them independently of one another.

The *createDots* function can take care of making the four original dots and all the movers. Similarly, *createButtons* will make any buttons. We'll put off the job of writing these functions for now: leaving an empty function reminds us it needs doing (note: Python uses 'pass', but in JavaScript, empty curly braces do the same job).

Each time the *draw* loop runs, time is incremented, but if it gets larger than 1, we adjust it back down.

The four *for* loops take care of displaying the dots, and updating and displaying the movers from each of the three arrays.

Note: it may seem simpler to keep all movers in a single array, but since we may want to independently decide which generation to show at different times, it will be easier in this case to keep them separate.

When we build in toggle buttons to show or hide different objects, we'll check their status at the appropriate points in this code.

Creating the dots…

```
function createDots(){
  dots = [];
  movers1 = [];
  movers2 = [];
  movers3 = [];
  for (let i=0; i<4; i++){
    dots.push(new Dot(random(100, width), random(height),i));
  }
```

…

We may make use of a reset function at some point, so to avoid potentially adding more dots or movers to arrays that are already populated, we start by overwriting them with blank arrays.
We make four dots in random places, leaving a 100 pixel margin to the left free of points (for our buttons).

Based on what we wrote earlier about our Dot class, we need to provide it with a position and index:

```
class Dot{
  constructor(x, y, i){
```

… and the movers

```
  ...
16  for (let i=0; i<3; i++){
17    movers1.push(new Mover(dots[i], dots[i+1], 8, "blue"));
18  }
19  for (let i=0; i<2; i++){
20    movers2.push(new Mover(movers1[i], movers1[i+1], 6, "green"));
21  }
22  for (let i=0; i<1; i++){
23    movers3.push(new Mover(movers2[i], movers2[i+1], 4, "red"));
24  }
25 }
```

This is where we create the first, second and third generation movers. Note that each loop iterates 1 fewer time than the one before, with 3 movers being generated to go between dots 0 and 1, dots 1 and 2 and then dots 2 and 3. The next generation comprises just two movers: from 1st generation mover 0 to 1, then 1 to 2. The third and final generation is really just a single mover (from 2nd generation mover 0 to 1), but I've written it in the form of a loop for consistency. If we wanted to build a higher level Bezier curve (ie with more than 4 initial points), I'd be tempted to automate this step, recursively producing new generations until we go down to just one mover.

When instantiating a *Mover* object, I'm passing actual dots or movers as the start and end points (if I just gave x or y values, as soon as the start or end moves they would no longer be valid), and also including some customisation features: a radius for the dot size, and a colour. The class will start like this:

```
class Mover{
  constructor(start, end, r, color){
```

The Dot class

```
class Dot{
  constructor(x, y, i){
    this.x = x;
    this.y = y;
    this.i = i;
  }
  display(){
    noStroke();
    fill("white")
    circle(this.x, this.y, 2*10);
    fill("black");
    text(this.i, this.x, this.y);
  }
}
```

Pretty much all that needs to be done is store the passed positional values *x* and *y*, and the instance number (so we can display it as text).

Since dots don't need to be updated (they don't move), all we need right now is a *display* method.

Remember to set *stroke* and *fill* every time, because otherwise the most recently used values will be used by default.

I use 2*10 to remind myself that the radius is 10 when using the *circle* function.

Recall that *text* draws the text provided at the given *x* and *y* position. In *setup*, add *textAlign* to centre it:

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

Moving the dots

```
function mouseDragged(){
  for (let dot of dots){
    if ((dot.x-mouseX)**2+(dot.y-mouseY)**2 < (10)**2){
      dot.x = mouseX;
      dot.y = mouseY;
    }
  }
}
```

The built-in *mouseDragged* function is called automatically every time you move the mouse while a button is down. If the mouse is close to a dot, that dot's *x* and *y* will change to match the mouse position.

The Mover class

```
class Mover{
  constructor(start, end, r, color){
    this.start = start;
    this.end = end;
    this.r = r;
    this.color = color
    this.x = this.start.x;
    this.y = this.start.y;
  }
  update(t){
    this.x = (1-t)*this.start.x + t*this.end.x;
    this.y = (1-t)*this.start.y + t*this.end.y;
  }
  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);
  }
}
```

The start and end points are in fact *Dot* objects, or other *Mover* objects. All that matters for our purposes is that they have *x* and *y* attributes that we can use to determine our position.

The *display* method draws the line the mover will move along, and a circle at its position.

For now, lines will be drawn by default. This is one of the features we'll toggle on or off with buttons later.

Adding a trail

```
class Mover{
  constructor(start, end, r, color){
    this.start = start;
    this.end = end;
    this.r = r;
    this.color = color
    this.x = this.start.x;
    this.y = this.start.y;
    this.trail = [[this.x, this.y]];                    // <-
  }
  update(t){
    this.x = (1-t)*this.start.x + t*this.end.x;
    this.y = (1-t)*this.start.y + t*this.end.y;
    if (t < 0.01){this.trail = [[this.x, this.y]]}      // <-
    let [x, y] = this.trail[this.trail.length - 1];     // <-
    if (abs(x - this.x) + abs(y - this.y) > this.r){    // <-
      this.trail.push([this.x, this.y]);                // <-
    }                                                   // <-
  }
```

Each *Mover* object should ideally have the option to have a trail (we'll be able to choose separately which get displayed with buttons (see part 2 for details).
We start with the very beginning.

Each time *update* is called, we check to see if *t* is reset (so the trail is erased each cycle), and new points added (but only if we are far enough away from the previous trail point to make it worthwhile).

We also need a dedicated method within the class, which uses p5's built-in *beginShape* and *endShape* functions to make a dotted line curve. To display this trial, we'll also need to invoke *showTrail* in *draw*.

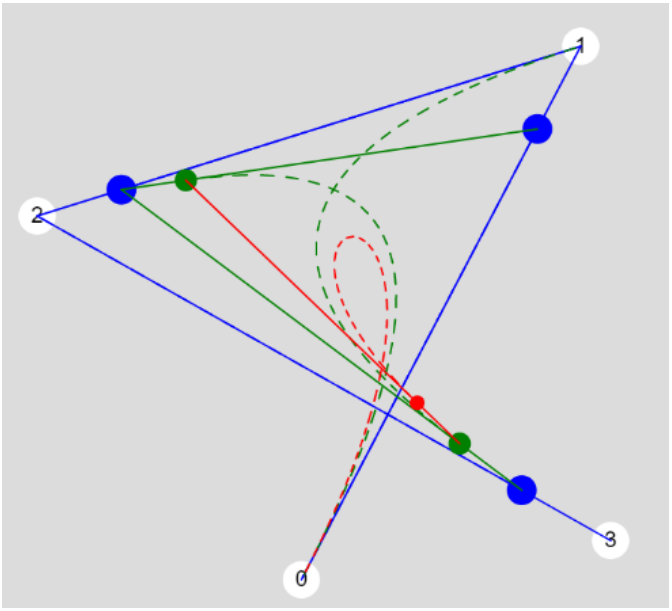
```
showTrail(){
  stroke(this.color);
  beginShape(LINES);
    for (let [x, y] of this.trail){
      vertex(x, y);
    }
  endShape();
}
```

```
for (let mover of movers3){
  mover.update(t);
  mover.showTrail();
  mover.display();
}
```

You can add this *mover.showTrail();* line to each loop that updates and displays movers in *draw* if you want to see all of the trails.

More Info

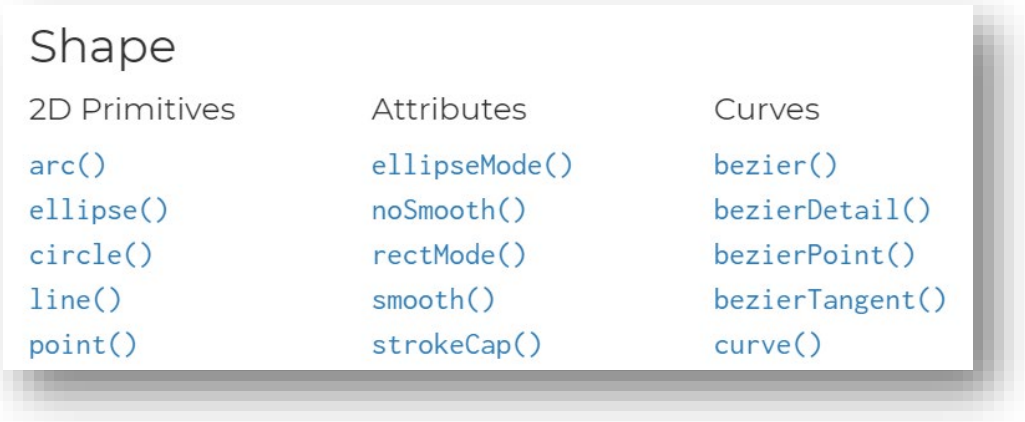
So far we've managed to create a graphical depiction of Bezier curve creation. In fact, what we've made here is known as a cubic Bezier curve. It's the most common type, and nearly the simplest.



The very simplest (which is still a curve and not a line) is the quadratic Bezier curve. The green (2nd generation mover) curves shown here are examples of quadratic Bezier curves. They require 3 points: a start, end and a single additional so-called 'control point' in between.

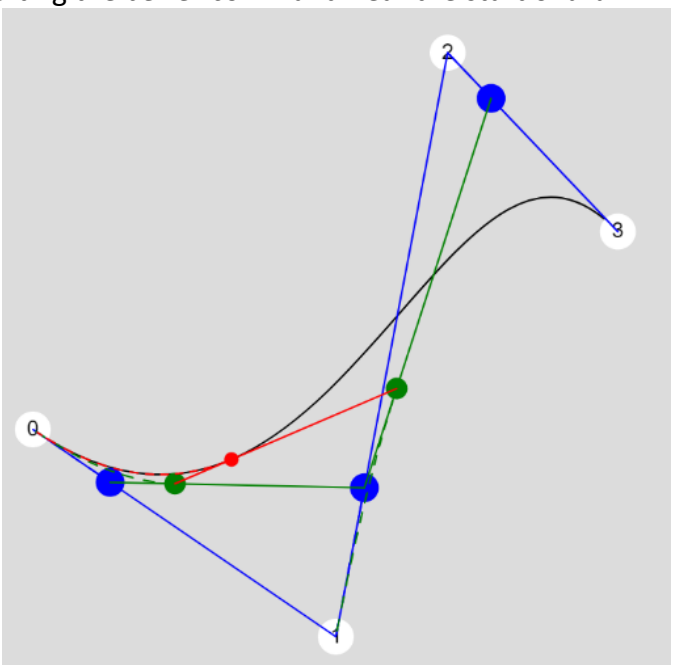
Cubic Bezier curves are defined by four points.

As we said, this is a commonly used technique in rendering curves in computer graphics, and as such it is exactly the kind of thing which is likely to have been coded for you somewhere behind the scenes in p5.js's library: Navigate to https://p5js.org/reference/ to see a list of many more built-in functions, including:



We can actually recreate our cubic Bezier curve by inserting the *bezier* command near the start of *draw*:

```
function draw() {
  background(220);
  t += 0.005;
  t -= int(t);
  noFill();
  stroke(0);
  bezier(dots[0].x, dots[0].y,
         dots[1].x, dots[1].y,
         dots[2].x, dots[2].y,
         dots[3].x, dots[3].y);
```



Note that the *bezier* function call here is all one line, but we can hit 'enter' to improve readability provided we still close brackets in the usual way.

Note the S-shaped curve now shown in black (which our red point will faithfully trace out).

In part 2, we'll return to this program and add functionality in the form of buttons to toggle visibility of various features. In the meantime, consider how our functions could be adapted based on a button's state.