ATM Conference: Visual Coding for Beginners

Why p5?

The p5.js coding environment is built on top of JavaScript as its base language. It is designed to make it as easy as possible to create sophisticated visual designs and animations, especially for use in online applications. If you end up doing more web development, you'll probably do a whole lot more work in both html and JavaScript, but p5.js lets us skip all the fiddly steps involved in getting a visual sketch coded and online by providing a coding environment directly in the browser with no installation necessary, much like replit.com if you've used that.

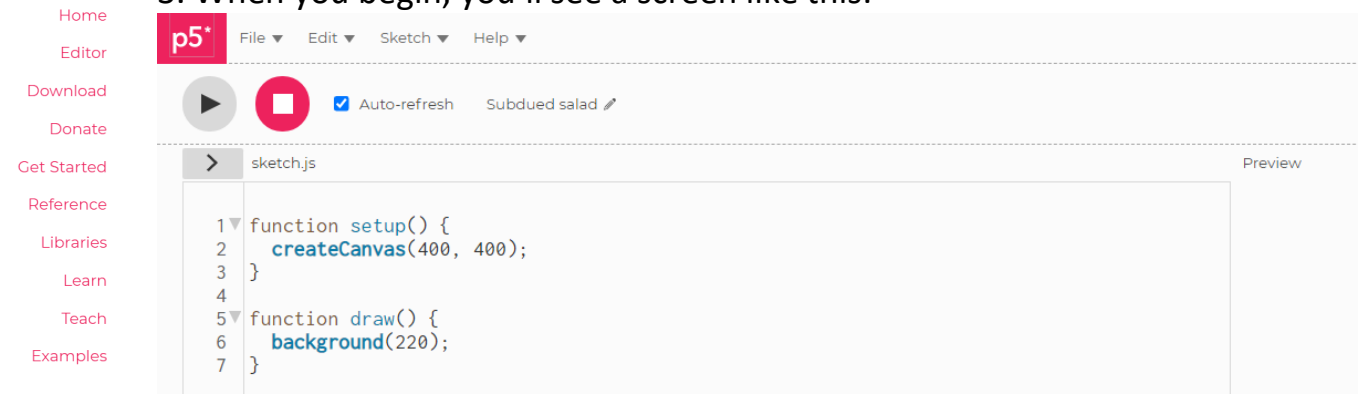
Complete beginner…
If you've never done any programming before, don't worry – although some of the ideas can take some time to properly embed, p5.js is designed with non-coders in mind, and makes it as easy as possible to jump straight in and start creating.

Some prior experience…
If you've already dabbled in Python, or some other language, rest assured that, although the syntax is different with JavaScript, all of the fundamental programming ideas are still there (variables, data types, conditionals, loops, functions, objects and classes), and it's much easier to pick up a second coding language than it is to learn your first!

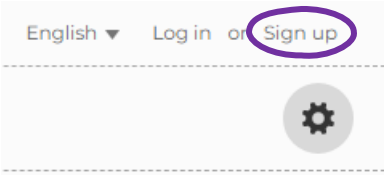
Finally, before diving in, I want to acknowledge the superb resources provided by Daniel Shiffman (aka "The Coding Train") (see thecodingtrain.com). If you want to learn more, I highly recommend his engaging and informative tutorials and coding challenge videos.

Navigating the website

1. Open p5js.org in your browser.
2. Hit *'editor'* or go to editor.p5js.org to start coding.
   (check out the *tutorials*, *examples* and *reference* sections for more info)
3. When you begin, you'll see a screen like this:



You can write and run code without having an account, but if you want to be able to save and share your sketches, it's worth signing up. You (and your students) can then share your saved sketches by simply copying the URL.



Using the editor

The first thing to notice is that your code already contains two functions: *setup* and *draw*.

```
function setup() {
  createCanvas(400, 400);
}
```

The *setup* code – anything within the curly braces – is run just once, at the start. We use it to initialise elements for the sketch.

The default code simply generates a canvas – a space on which to draw – with a default width and height of 400 pixels.

```
function draw() {
  background(220);
}
```

The *draw* function will be run repeatedly for as long as your code is active, at a frame rate of around 60 frames per second.

The default code fills the canvas with a pale grey; 220 is a greyscale colour on a brightness scale of 0 (black) to 255 (white).

In addition to these two main functions, there are a fair few other special functions which are particular to p5.js, such as *mousePressed* which will run every time you hit the mouse.

Syntax

Computer instructions must be precise. Code syntax means the rules we have to follow when writing our instructions so that the computer knows exactly how to interpret them. If your code doesn't work as expected, especially if you're trying something new, the most likely culprit is a syntax error – double-check that your {curly braces} match up, for instance.

You can include spaces, indents and blank lines to make your code more readable, and you can write comments (extra information intended for human readers, ignored by the computer) by preceding them with a double slash // as shown in this example:

```
let x = width/2  // rocket should start in the centre
let y = height/2
```

Just like the annotations and explanation in the solution to a maths problem, comments are not strictly necessary for the code to run, but they turn out to be really helpful for the humans writing the code, or returning to it after a time, to understand what we're doing.

Choosing colours

Change the colour of the background to something nicer – there are lots of ways to define colours in p5, such as:

```
background(100);
```

A single value from 0 to 255 is interpreted as greyscale:

```
background(255, 0, 255);
```

Three values are interpreted as Red, Green and Blue:

```
background("blue");
```

Common colours can even be written in speech marks:

Note that if you type the colour name, a small coloured square appears – if you click on it you can select a different shade of colour without having to know its name, or RGB values.

Now try:

Change the colour of the background to something nicer than grey. You can also change the (400, 400) in the *createCanvas* function to see how that affects the size of your sketch.

Warm-up Project: Circle

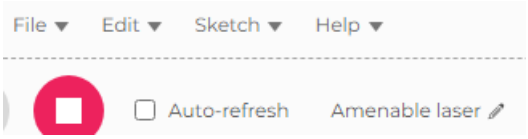
Since the basic format (the *setup* and *draw* functions) are already included by default, we only need to write a single line to make our first sketch:

```
1  function setup() {
2    createCanvas(400, 400);
3  }
4
5  function draw() {
6    background(220);
7    circle(mouseX, 250, 200);
8  }
```

Just below the *background* line, I'm adding a call to the built-in *circle* function. This takes 3 arguments (inputs): x-coordinate, y-coordinate and diameter.

Note that *mouseX* is a built-in variable which is constantly updated to reflect the x-coordinate of the mouse. See what happens when you move your mouse over the sketch.

Enter a recognizable name and using File > Save to make sure you don't lose your sketch when you close the browser.

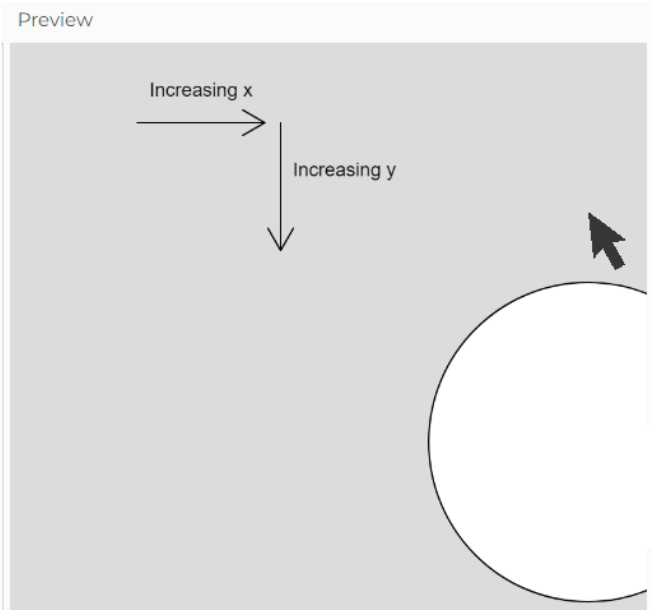


A new coordinate system

Coordinates are defined relative to the top left corner of the canvas, so although x still points right like the Cartesian coordinate system, y now points down.

This is the default for pretty much all computer graphics, and can take some getting used to.

If something doesn't display as expected, try changing the sign of the y-coordinate!



Now try:

Incorporate *mouseY* to make the circle exactly follow the mouse instead of always being 250 pixels down. Get creative with the diameter, too – try replacing it with something like *mouseX* – *mouseY*. What do you notice?

Customisation

One-liners are great, but for more control of colours etc, try these built-in functions:

```
function draw() {
  background("blue");
  fill("yellow");
  stroke("red");
  strokeWeight(width/40);
  circle(width/2, height/2, width*0.9);
  noStroke();
  fill("red");
  circle(width*0.4, height*0.4, width*0.1);
}
```

- *fill* sets the colour of the inside.
- *stroke* sets the boundary colour.
- *strokeWeight* sets the line width.
- *noStroke* removes the border entirely.

Notice how I can use *width* and *height*, built-in variables for the current canvas, so that if I change the canvas size later, I won't have to go through my program changing everything.

Stuck? Visit https://editor.p5js.org/thechalkface/sketches/9p5v0O2mC to see an example.

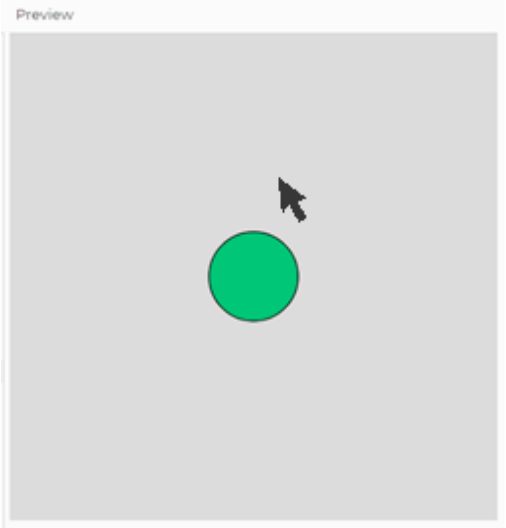
Project 1: Animation

This starter sketch will give you a taste for what p5.js can do, and give you a template from which to develop your own. Can you figure out what this code should do before you run it?

```
let x = 0;

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

function draw() {
  frameRate(20);
  x += 1;
  background(220);
  fill(0, mouseX, mouseY);
  circle(width / 2, height / 2, x);
}
```



Declaring variables

Note that (unlike, for instance, Python) variables must be declared before use. The declaration happens at the very beginning with the key word *let*, so that the variable defined is a global variable, accessible within the *setup* and *draw* functions, for instance. You can think of an *x* that gets declared only within a specific function as being like the x that you define at the start of a specific maths question. When you move on to the next question, the value is forgotten. Declaring it outside any functions lets you modify it from within any function, but it's independent of any one function.

Simulating motion

By making the diameter of a circle depend on a variable that may change, we can modify its size as time goes by. This circle's diameter will grow by 1 pixel per frame (that is, 20 pixels per second at the current frame rate).

Setting the frame rate

The frame rate is just the number of times the draw loop repeats per second. Although it is not necessary to set the frame rate, slowing it way down by using, say, *frameRate(1)*, will make it easier to see what's happening, especially if your sketch isn't behaving as expected.

Changing colours

If you want colour to change over time, or depending on the position of the mouse, it can be helpful to use the 3 number R, G, B notation. Here we've set the amount of red to zero, and the amount of green and blue is recalculated every frame based on the mouse position.

Now try:

Instead of having *x* determine the diameter of the circle, try modifying the code so that it determines the x-coordinate of the centre of the circle. Declare another variable, *y*, at the start, and use that to set the y-coordinate of the circle's centre each frame.

Stuck? Visit https://editor.p5js.org/thechalkface/sketches/pNZ6sBmCr to see an example.

Project 2: Bouncing ball

We're going to need to keep track of a few variables here, to determine the current position and current velocity of our ball. We'll use *r* for the radius, *x* and *y* for horizontal and vertical displacement, *vx* and *vy* for horizontal and vertical components of velocity.

```
1  let r = 30;
2  let x = r;
3  let y = r;
4  let vx = 5;
5  let vy = 2;
6
7  function setup() {
8    createCanvas(400, 400);
9  }
```

The ball will start at the top left of the screen, with velocity $\begin{pmatrix}5\\2\end{pmatrix}$ (right and down, not right and up!)

Note that I've both declared and defined these variables at the very beginning.
If I wanted to make use of p5's built-in variables like *mouseX* or *height*, or built-in functions like *random*, I'd still declare them at the top (eg *let x*), but define them inside the *setup* function itself.

For more information on the random function, see Additional Information at the end of the document.

The circle itself will be drawn in the *draw* function, using *x* and *y* as the coordinates of the centre. Since these values will change, we need the circle to be repeatedly redrawn at the new *x* and *y* coordinates. We'll also make it so that *x* and *y* are incremented by *vx* and *vy*.

Incrementing position by velocity

```
12 function draw() {
13   background(220);
14   circle(x, y, d);
15   x += vx;
16   y += vy;
17 }
```

We need to add the horizontal speed to the horizontal position, and the vertical speed to the vertical position, so we use *x += vx* and *y += vy*, redefining *x* as *x + vx* etc.
For more information on the += notation, see Additional Information at the end of the document.

Bouncing

Whenever the ball gets within one radius (*r*) of a boundary, I want to reverse its direction, and also undo the latest move in that direction (the one that sent it over the boundary).

```
11 function draw() {
12   background(220);
13   x += vx;
14   y += vy;
15   if (x > width - r || x < r){
16     x -= vx;
17     vx *= -1;
18   }
19   if (y > height - r || y < r){
20     y -= vy;
21     vy *= -1;
22   }
23   circle(x, y, 2*r);
24 }
```

Notice how the *if* conditional syntax works: the condition goes in brackets after the *if* key word, and the code block that follows is enclosed in curly braces { }.

Also note the use of || to mean 'or'. This saves me writing out four different *if* functions.
For more information on the if function and the || operator, see Additional Information at the end of the document.

Stuck? Visit https://editor.p5js.org/thechalkface/sketches/yEQTYWiaF to see an example.

Extension Challenges:

- Comment out the *background* call on line 12 to see what happens if we don't redraw the grey background every frame.

```
12    //background(220);
```

- Try inserting something like *vy* += 0.2 at the start of *draw* to simulate gravity – this increases the downward velocity by a fixed amount each frame, giving acceleration.

```
13    vy += 0.2;
```

- Try multiplying *vx* and *vy* both by 0.99 each time *draw* runs to simulate air resistance.

```
14    vx *= 0.99;
15    vy *= 0.99;
```

- Implement an element of randomness in the initial position or velocity of the bouncing ball. *For more information about the random function, see Additional Information at the end of the document.*

```
7  function setup() {
8    createCanvas(400, 400);
9    x = random(r, width - r);
10   vx = random(5);
11 }
```

Curriculum links

I run coding enrichment for my A-level Further Maths students, and often design projects around a particular aspect of the syllabus such as projectile motion, complex numbers or differential equations. But the main value in my opinion is in the overlap between the core skills that coding develops and those required by mathematicians. A combination of creativity and rigour, the attention to detail and an appreciation for an elegant solution, the unforgiving nature of bad code or faulty reasoning and the satisfaction of a problem solved or a project completed successfully.

For more p5 information, examples and tutorials, see p5js.org/reference

Check out the videos and coding challenges from thecodingtrain.com

Tweet me your sketches, questions and ideas: **@the_chalkface**

Finally, this is my first time teaching teachers, so thank you for your forbearance! If you have a minute, please fill in this form to let me know what worked for you and what didn't, so I can do a better job next time. Thanks!

Appendix: Additional Information

Plus-Equals

Since it's very common in programming to increment a value, we often replace or overwrite a variable with itself-plus-one. To save time writing *x* = *x* + 1, we use the shorthand *x* += 1. This can even be abbreviated to *x* ++ (by default adding 1, eg keeping a tally of something).

For instance:

```
12 function draw() {
13   background(220);
14   circle(x, y, d);
15   x += vx;
```

In our code, we wanted to increase the horizontal displacement, *x*, by the horizontal velocity, *vx*.
This code simply overwrites *x* with the new value, *x* + *vx*.

Conditional functions

One of the most important concepts in programming is the *if* function. It allows us to direct our program to do different things depending on a certain condition. Although the structure is different in different coding languages, the fundamentals are the same: you'll need a condition which can be evaluated to either true or false, then what should be done if the condition is met (the condition evaluates to true), and, optionally, what should be done if the condition is not met (it evaluates to false).

For instance:

```
function draw() {
  background(220);
  if (mouseX > 200){
    circle(200, 200, 100);
  }
  else{
    square(200, 200, 100);
  }
}
```

The condition to evaluate goes after the *if* keyword, enclosed in parentheses ( ), then curly braces { } go around the code to be executed if the condition is true.

After that block of code is closed by the }, we can choose to include an *else* block, which produces a different result when the condition is false.

Logical operators

If you find yourself writing too many *if* functions, it may be because you're not making use of logical operators that can save you some time.

For instance:

```
if (mouseX > 200 && mouseY > 200){
  circle(200, 200, 100);
}
```

The double ampersand && means 'and'. In this case, a circle only appears if the mouse pointer is in the bottom right quadrant of the screen.

```
if (mouseX > 200 || mouseY > 200){
  circle(200, 200, 100);
}
```

The double pipe || (type with shift and \ ) means 'or'. The circle appears if the mouse is to the right or below the centre (or both).

Random function

The built-in *random* function gives random outputs in a few different ways.

For instance:

```
function setup() {
  createCanvas(400, 400);
  x = random(r, width - r);
  vx = random(5);
```

random() returns a random value from 0 to 1.
random(a) returns a random value from 0 to a.
random(a, b) returns a random value from a and b.