

*Algorithmic
Art
Workbook #8
Perlin and
Fireworks 1*



Table of Contents

MIT Licence	4
Workbook #8 Quick Content Summary	5
p5.js Versions	6
Changing the Version	7
An Infinite Loop	8
The Purpose of the Console	9
Module E Unit #1: perlin noise	11
Sketch E1.1 starting with a standard sketch	12
Sketch E1.2 randomly moving circle	13
Sketch E1.3 smooth random movement	15
Sketch E1.4 random colour B/W	17
Sketch E1.5 random colour RGB	19
Module E Unit #2: 2D Perlin Noise	22
Sketch E2.1 starting sketch	23
Sketch E2.2 row of pixels	24
Sketch E2.3 vertex replaces point	25
Sketch E2.4 random y	26
Sketch E2.5 introducing the perlin noise	28
Sketch E2.6 xoff	30
Sketch E2.7 a bit static	32
Sketch E2.8 scrolling graph	33
Sketch E2.9 noise detail	34
Sketch E2.10 smoothing	35
Sketch E2.11 not so smooth	37
2D perlin noise	39
Sketch E2.12 time for 2D	40
Sketch E2.13 are we ready?	41
Sketch E2.14 loading the pixels	42
Sketch E2.15 four channels	43
Sketch E2.16 random static	45
Sketch E2.17 make it noisy	47
Sketch E2.18 a wee amount	49
Sketch E2.19 reversing the loops	51
Sketch E2.20 yoff	53
Module E Unit #4: smoke and fire	56
Sketch E3.1 starting sketch	57
Sketch E3.2 adding a particle class	58
Sketch E3.3 adding move and show	59
Sketch E3.4 draw a circle	60

Sketch E3.5 particle p	61
Sketch E3.6 an array of particles	63
Sketch E3.7 move the particle	65
Sketch E3.8 many particles	67
Sketch E3.9 no smoke without fire	69
Sketch E3.10 too many particles	71
Sketch E3.11 remove dead particles	73
Sketch E3.12 backwards array	76
Sketch E3.13 adding more particles	79
Sketch E3.14 some adaptations	82
Module E Unit #4: fireworks display Part 1	86
Sketch E4.1 starting sketch	87
Sketch E4.2 create vectors	88
Sketch E4.3 move() function	90
Sketch E4.4 creating a force	91
Sketch E4.5 adding the force	93
Sketch E4.6 draw the particle	94
Sketch E4.7 size and colour	96
Sketch E4.8 moving upwards	98
Sketch E4.9 upwards velocity	100
Sketch E4.10 gravity	102
Sketch E4.11 higher	104
Sketch E4.12 lots of fireworks	106
Sketch E4.13 explode	108
Sketch E4.14 remove redundant code	110
This continues in the next workbook	113



MIT Licence

Copyright ©2026 Warren George

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Brief summary:

Permissions (what you can do)

Commercial use
Modification
Distribution
Private use

Limitations (what is not covered)

Liability
Warranty



Workbook #8 Quick Content Summary

One of the ways you can develop very interesting effects is to use particles which have a particular property. There are various forms of random, and Perlin Noise is one that can be used to great effect.

We will be using classes quite a bit here. One way to manage this is to create new files and put the classes in separate files. This was my original idea, but to avoid the dreariness of jumping out of one file into another, I have kept everything in the main sketch. This does mean that there are longer lines of code in the sketch, which in turn makes the length of the workbook longer.

The creation of files will be demonstrated in the next workbook, so keep the unit on Firework displays for that bit (you will need it).

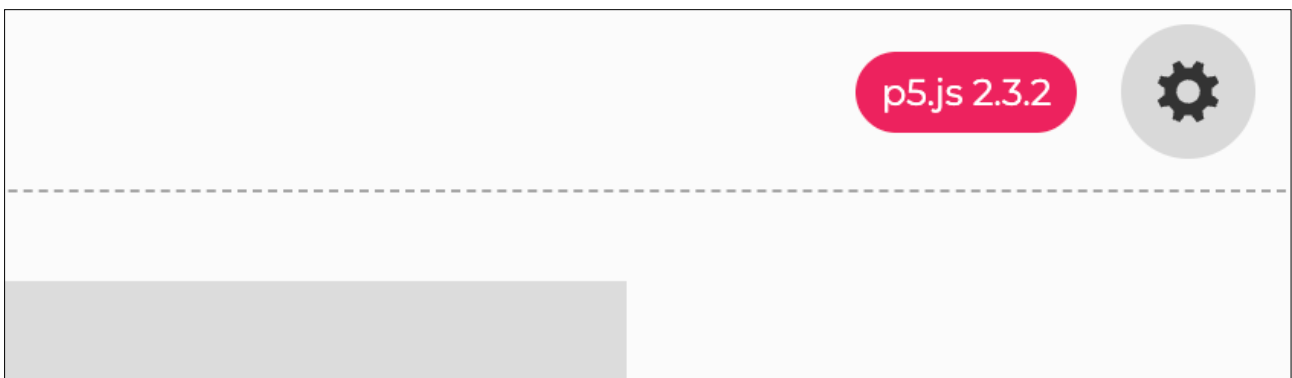


p5.js Versions

The latest version of p5.js (as of September 2026) is **2.3.3**. This has a few bugs which are being ironed out as we speak. So by the time you get to this point you may well find that my advice is irrelevant but just in case it is not I will make the following suggestions.

I suggest that you use version **2.2.3** because I have tested the code on that version and seems to work fine. You can easily switch versions. If in doubt then try different ones if a problem arises. This is especially true if you copy an older example from the internet.

Figure 1: version selection

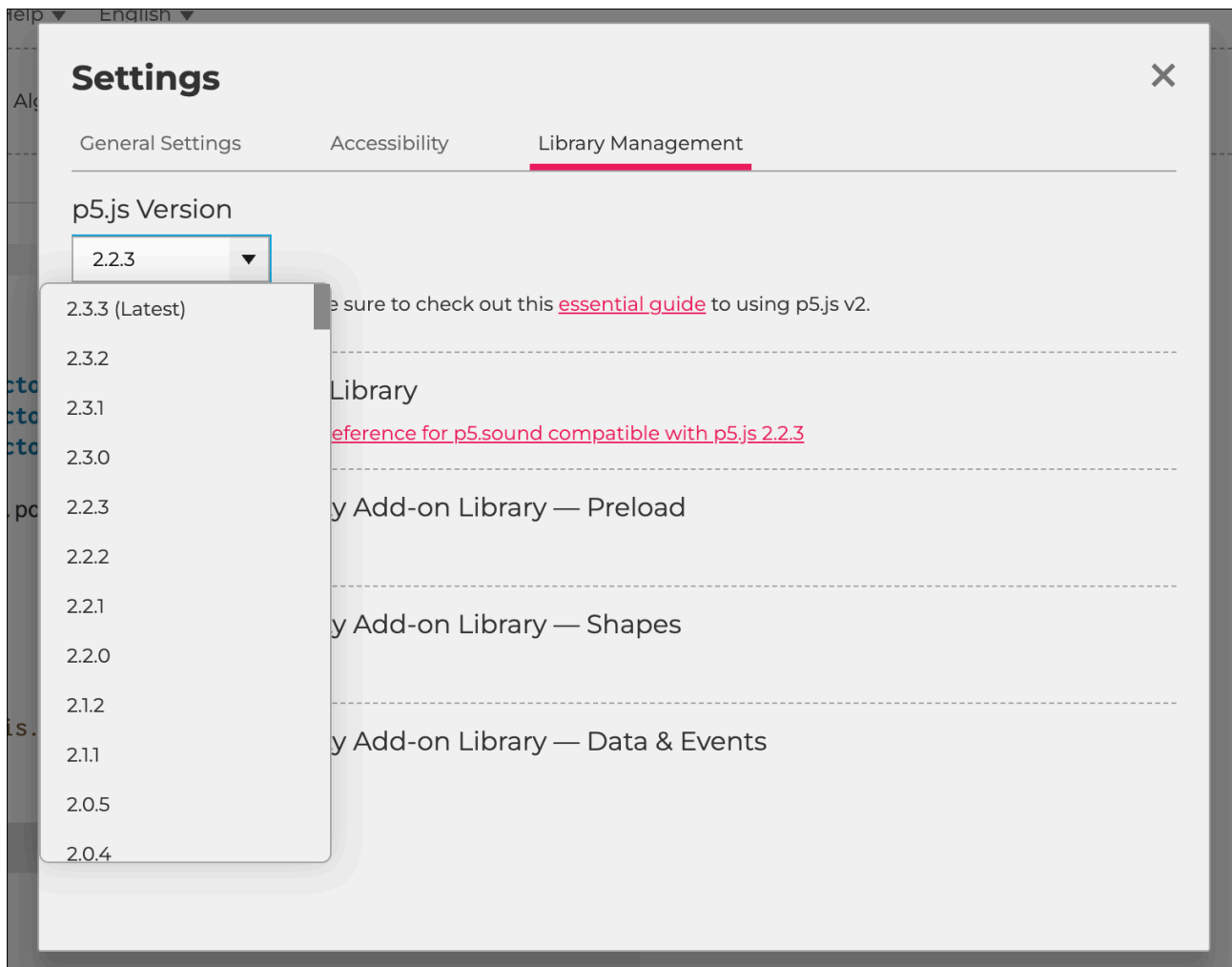




Changing the Version

You get a drop down menu like this shown in Fig.2:

Figure 2: changing the version



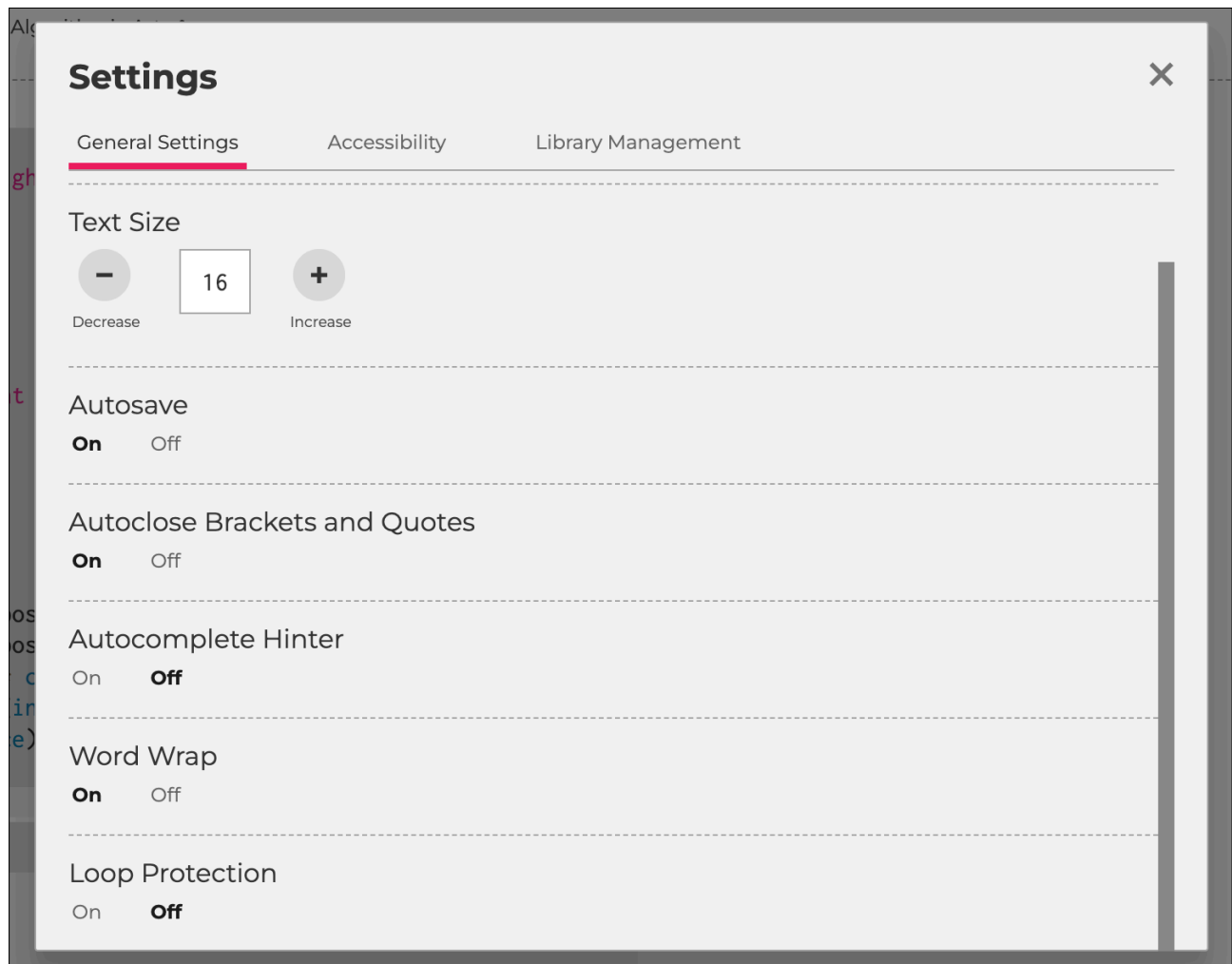


An Infinite Loop

If you get an **infinite loop detected** warning then you need to go to **Settings** (cog symbol top right), scroll down to Loop Protection, then turn this off and continue unless of course you have accidentally created an infinite loop that is.

A few sketches using perlin noise and pixels can create this problem. So I just want to alert you to the fact so that you are aware.

Figure 3: loop protection

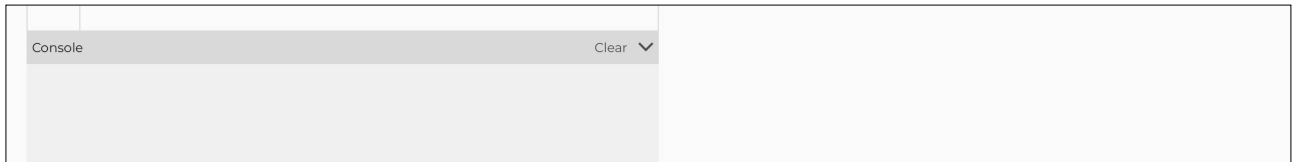




The Purpose of the Console

The **console** is the grey boxed section underneath where you enter your code. This is a very useful function for a number of reasons. It is where you will get error messages and where you can send information. It's main purpose is for debugging which is another word for problem solving.

Figure 4: the console



Error messages

If your editor picks up on some glaring and obvious errors in your code which can be anything from a missed comma or semicolon to an unknown variable appearing. Mostly you will get something useful and informative although sometimes it just flags a problem and leaves you to search for it.

Sometimes the error code may be highlighted in red and it may even give you the line number to identify where the problem is (or the following line number where it does become a problem).

Console log

We can put a line of code in called **console.log()**. This means you are going to log something in the console. Sometimes it can be a piece of text such as **console.log('finished training')**, or it can be the value of a variable such as **console.log(counter)** which will give you the value of the variable at that time in the code. Another really helpful use is debugging a problem where you are not sure what is happening, for example, to an array.



Algorithmic Art Module E Unit #1 Perlin noise



Module E Unit #1: perlin noise

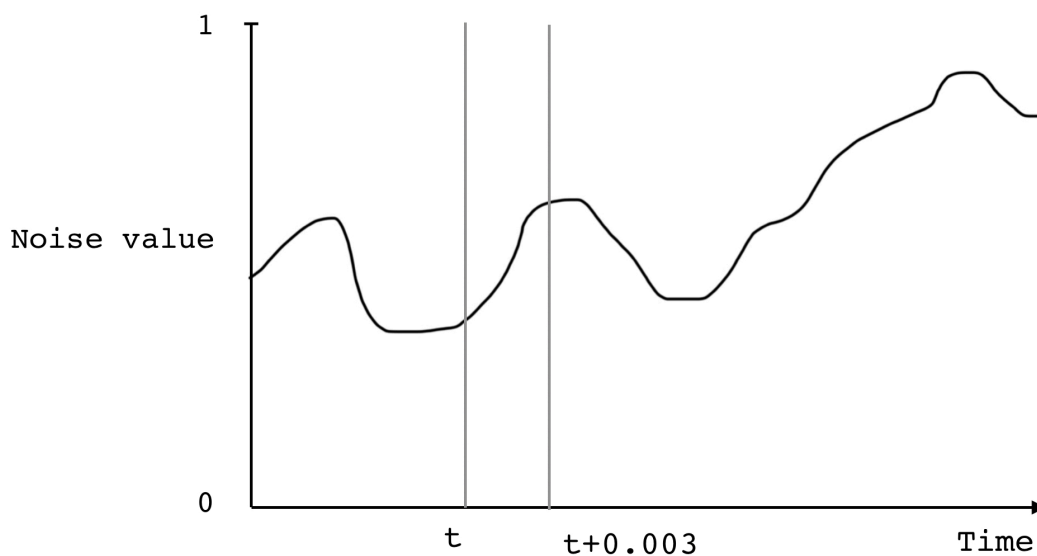
Perlin `noise()` returns a random value between `0.0` and `1.0` at a specific point in time. You specify the start time. It is then incremented along this smooth random time line in steps; the smaller the steps (for instance, `0.005`) means there are very small steps but the changes are therefore smoother, whereas larger steps (`0.03`) obviously create a much greater degree of randomness and possibly less smooth changes. It seems to work best between `0.005` and `0.03`.

If this seems strange at first, it is because it is less intuitive than just plucking a random number out of the air. If you have two variables that you want to have different random noise outcomes, you simply use the `noise()` function but start at different times (points along the line), for instance, one variable could start at `3` and the other starts at `100`.

The beauty of this is that it is random but it bears some relationship with the previous random number at a particular point in time. So if you move it on a small increment, you get a slight adjustment to the random value.

If you want to understand how it works, there are a number of articles around; the [Nature of Code](#) is a good reference point.

Figure 5: perlin noise graph





Sketch E1.1 starting with a standard sketch

Starting with our normal basic sketch.

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

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

Notes

Our familiar friend



Sketch E1.2 randomly moving circle

We start the circle in the centre of the canvas and randomly move it. We are just going to use the `random()` function to move the circle around the canvas. You will notice that it is not very smooth or fluid; `noise` will give you something more natural.

```
let x = 200
let y = 200

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

function draw()
{
  background(220)
  circle(x, y, 200)
  x = x + random(-2, 2)
  y = y + random(-2, 2)
}
```

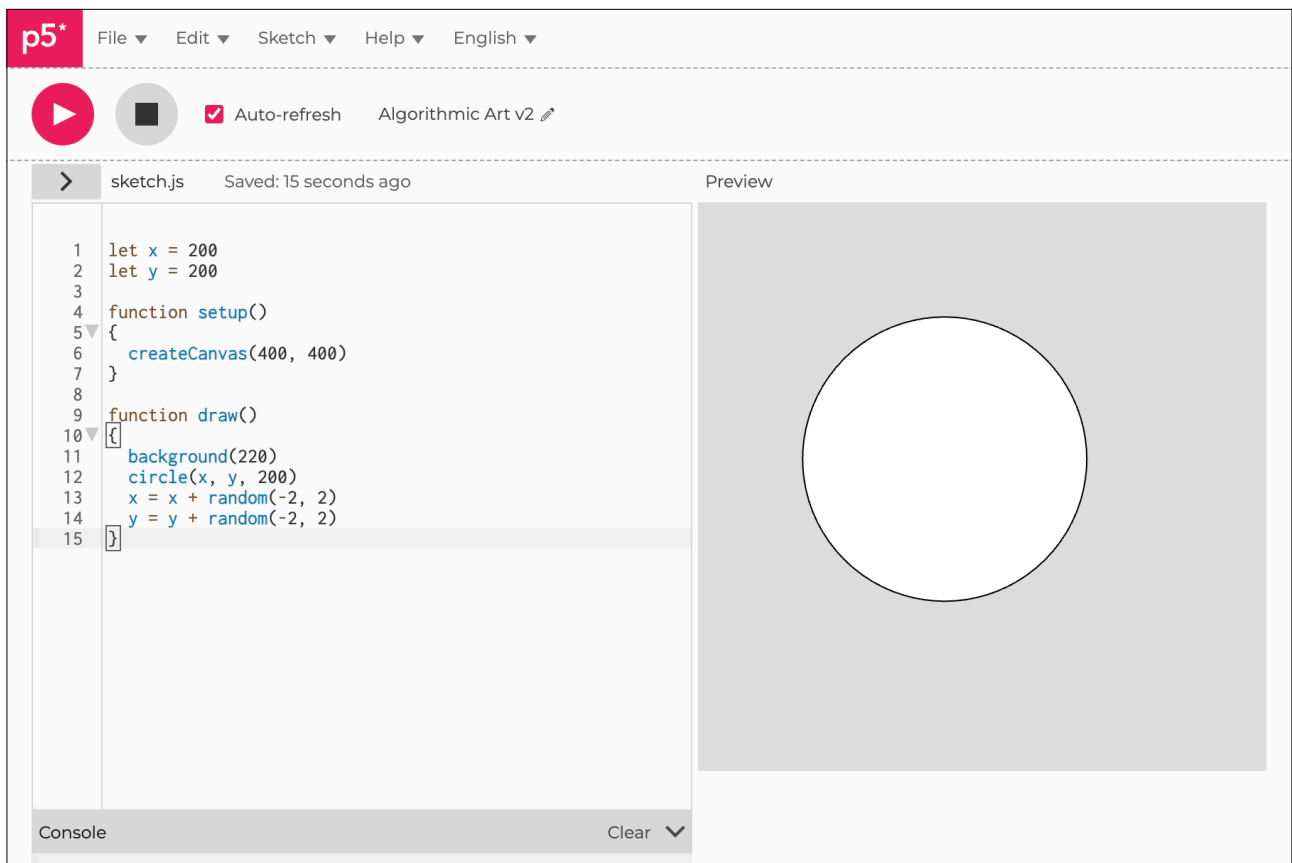
Challenges

1. Try random values of (5, -5).
2. Try different values for x and for y.

Code Explanation

<code>x = x + random(-2, 2)</code>	Add a random value between -2 and 2 to the x value on each iteration.
<code>y = y + random(-2, 2)</code>	Add a random value between -2 and 2 to the y value on each iteration.

Figure E1.2





Sketch E1.3 smooth random movement

We have replaced the `random()` function with the `noise()` function. Notice that the jerkiness has gone, replaced by a much smoother movement, almost as if floating in the air. Also, it looks a lot more complicated. We have two start times (3 and 10). Because `noise` returns values between 0 and 1, we use the `map()` function to scale the movement up to the `width` and `height` of the canvas. Then, on each iteration, we move along the `noise` timeline by 0.005 increments.

```
let timeX = 3
let timeY = 10

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

function draw()
{
  background(220)

  let x = map(noise(timeX), 0, 1, 0, width)
  let y = map(noise(timeY), 0, 1, 0, height)

  circle(x, y, 200)

  timeX = timeX + 0.005
  timeY = timeY + 0.005
}
```

Notes

To see how noise works and why it is so much better than just `random()`, this short programme illustrates the smoothness of the movement.

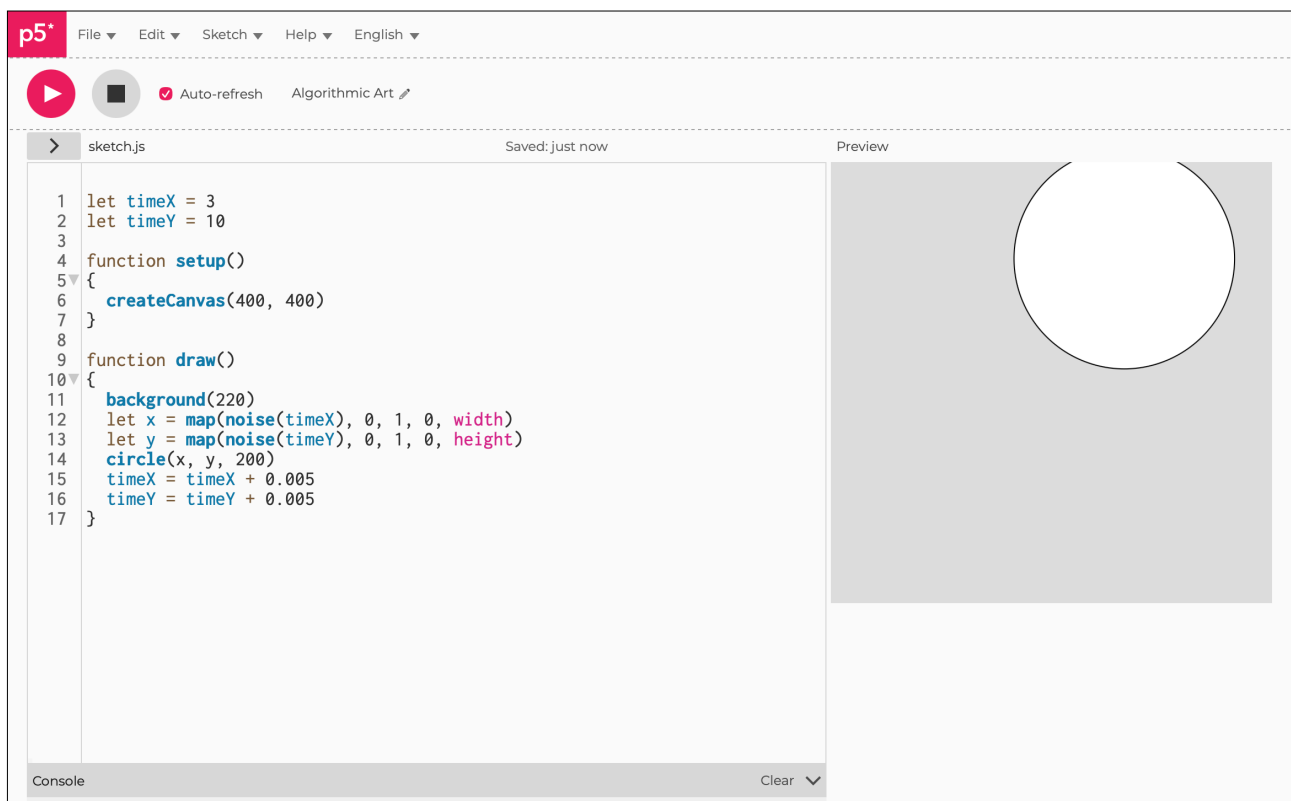
Challenges

1. Try different increments for `timeX` and `timeY`.
2. What happens if you give them both the same start on the timeline?
3. Try: `timeX = timeX + 0.05`.
4. Replace it with `timeX += 0.005`.

Code Explanation

<code>let timeX = 3</code>	The starting value for the x timeline
<code>let timeY = 10</code>	The starting value for the y timeline
<code>let x = map(noise(timeX), 0, 1, 0, width)</code>	Maps the timeX value to the width of the canvas
<code>let y = map(noise(timeY), 0, 1, 0, height)</code>	Maps the timeY value to the height of the canvas
<code>timeX = timeX + 0.005</code>	Adds an increment to the timeX timeline
<code>timeY = timeY + 0.005</code>	Adds an increment to the timeY timeline

Figure E1.3





Sketch E1.4 random colour B/W

! Start a new sketch.

Adding a colour element to change the grayscale.

```
let timeCol = 3
let col = 0

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

function draw()
{
  background(220)
  fill(col)
  col = map(noise(timeCol), 0, 1, 0, 255)
  circle(width/2, height/2, 200)
  timeCol = timeCol + 0.005
}
```

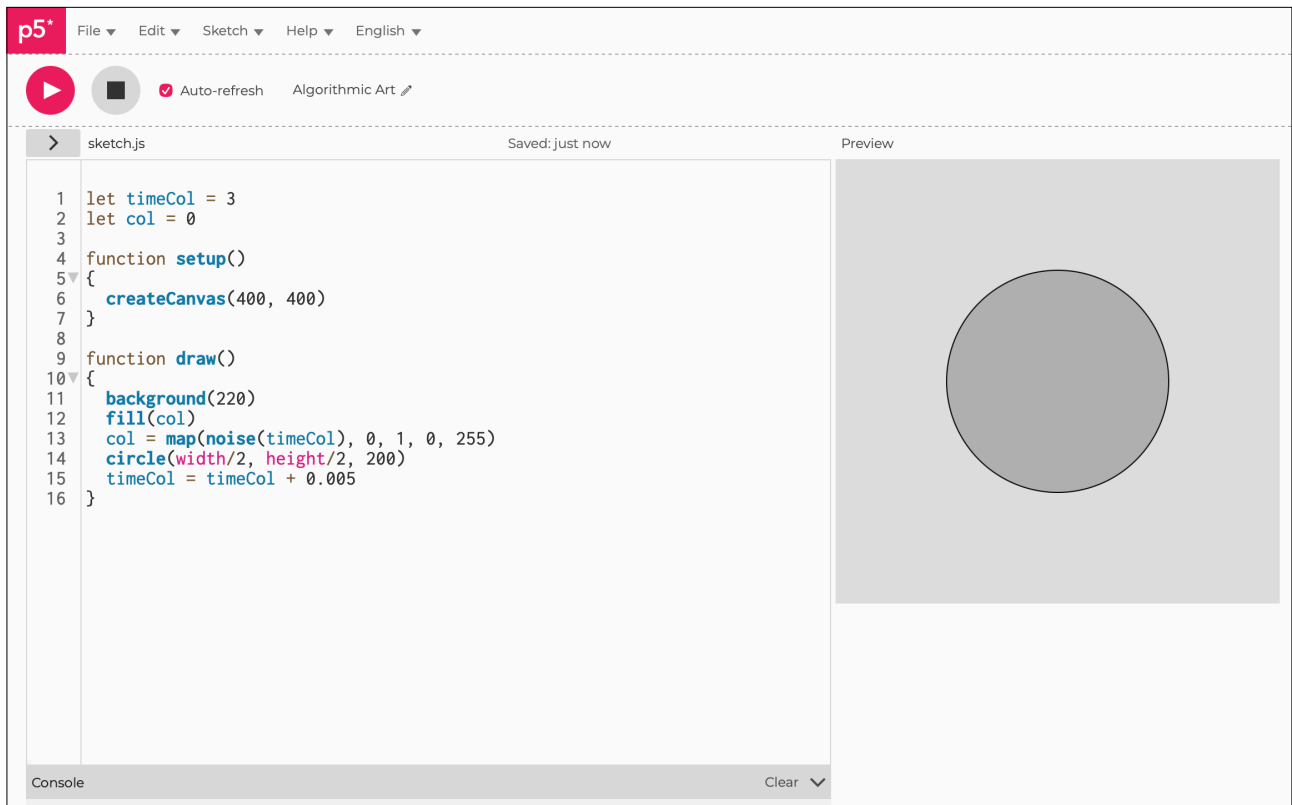
Notes

The colour of the circle changes gradually through the greyscale values between 0 and 255. The transition from one level of grey to another is also smoother and much more pleasing to the eye.

Challenge

Have the background change as well using `noise()`.

Figure E1.4





Sketch E1.5 random colour RGB

! Remove: the non-relevant code, we are replacing a single colour with the RGB ones instead.

Changing all three to create a flow of colour changes.

```
let timeRed = 30
let timeGreen = 300
let timeBlue = 3000
let colRed = 0
let colGreen = 0
let colBlue = 0

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

function draw()
{
  background(220)
  fill(colRed, colGreen, colBlue)
  colRed = map(noise(timeRed), 0, 1, 0, 255)
  colGreen = map(noise(timeGreen), 0, 1, 0, 255)
  colBlue = map(noise(timeBlue), 0, 1, 0, 255)
  circle(width/2, height/2, 200)
  timeRed += 0.003
  timeGreen += 0.003
  timeBlue += 0.003
}
```

Notes

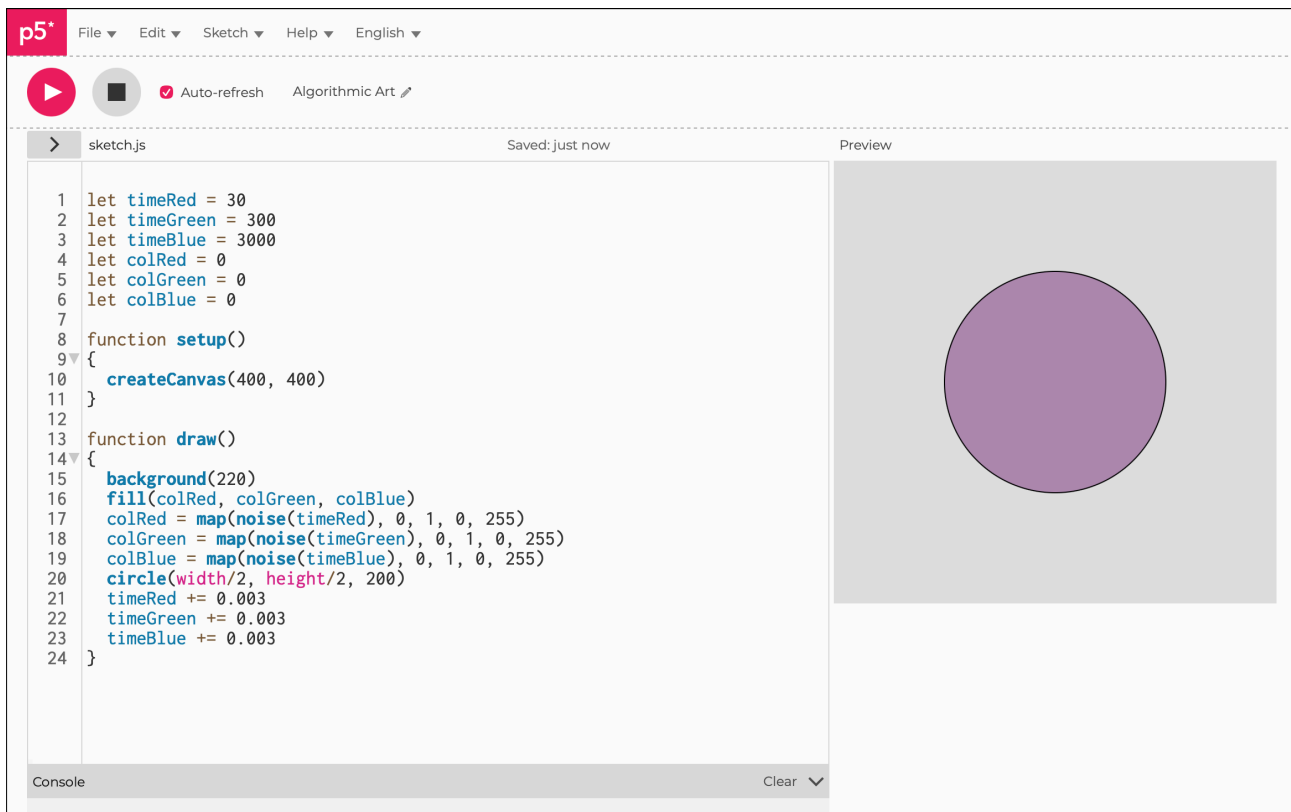
Like the previous sketch, we get a smoother and more subtle transition from one colour to the next.

This can take a little while to get your head around. With random, you give it a number to randomise up to and from, but with noise, you are picking a point on a random timeline. Play with this and persevere; it will become a bit more intuitive.

Challenges

1. Change the background colour, easy hint: just swap the colour variables round, for instance: `background(colGreen, colBlue, colRed)`
2. Movement, colour, and size all at the same time.

Figure E1.5



Algorithmic Art

Module E

Unit #2

2D Perlin Noise





Module E Unit #2: 2D Perlin Noise

Although we touched on Perlin noise in the first unit, we will continue by revisiting it from another angle, which may help you in understanding Perlin noise() intuitively before jumping into 2D Perlin noise.

There is so much benefit in getting your head around how Perlin noise() works. It will add randomness (if needed) to your artwork that can surprise you, hence this is why keep on returning to it, I make no apology.



Sketch E2.1 starting sketch

! Our starting sketch

This is a little bit of a recap of 1D Perlin noise; we will build on this very quickly.

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

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

Notes

We have our simple sketch.



Sketch E2.2 row of pixels

We create a `for()` loop and a point every pixel across the width of the canvas.

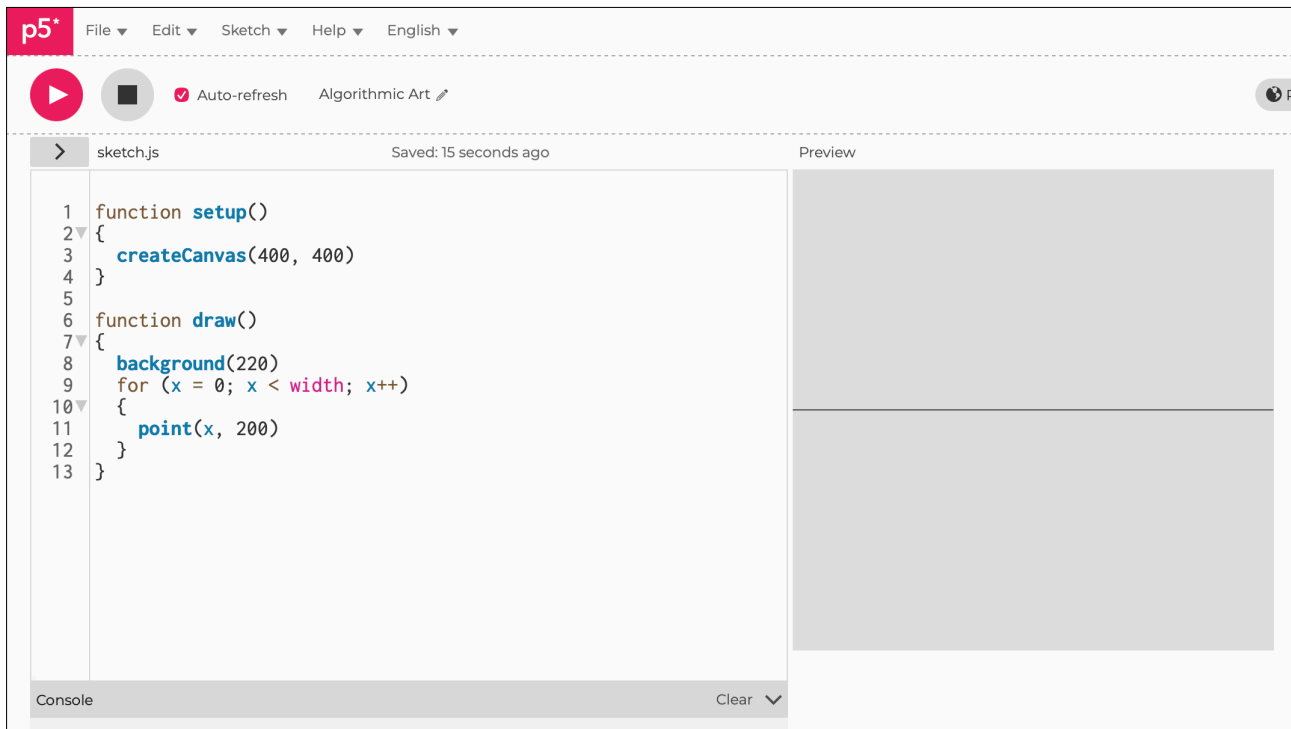
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  for (x = 0; x < width; x++)
  {
    point(x, 200)
  }
}
```

Notes

We have a line of points.

Figure E2.2





Sketch E2.3 vertex replaces point

Replace the point with a **vertex** and join them all up. You should get a line as before.

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

function draw()
{
  background(220)
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, 200)
  }
  endShape()
}
```

Notes

You should have what looks like the same as before. Instead, we have line joined the points (pixels) but too small to see.



Sketch E2.4 random y

Instead of a **y** value of **200**, let's give it a random value between the top and the bottom of the canvas. We also include a **noLoop()** function so that we have a static image; if you comment it out, you will see the random effect animated.

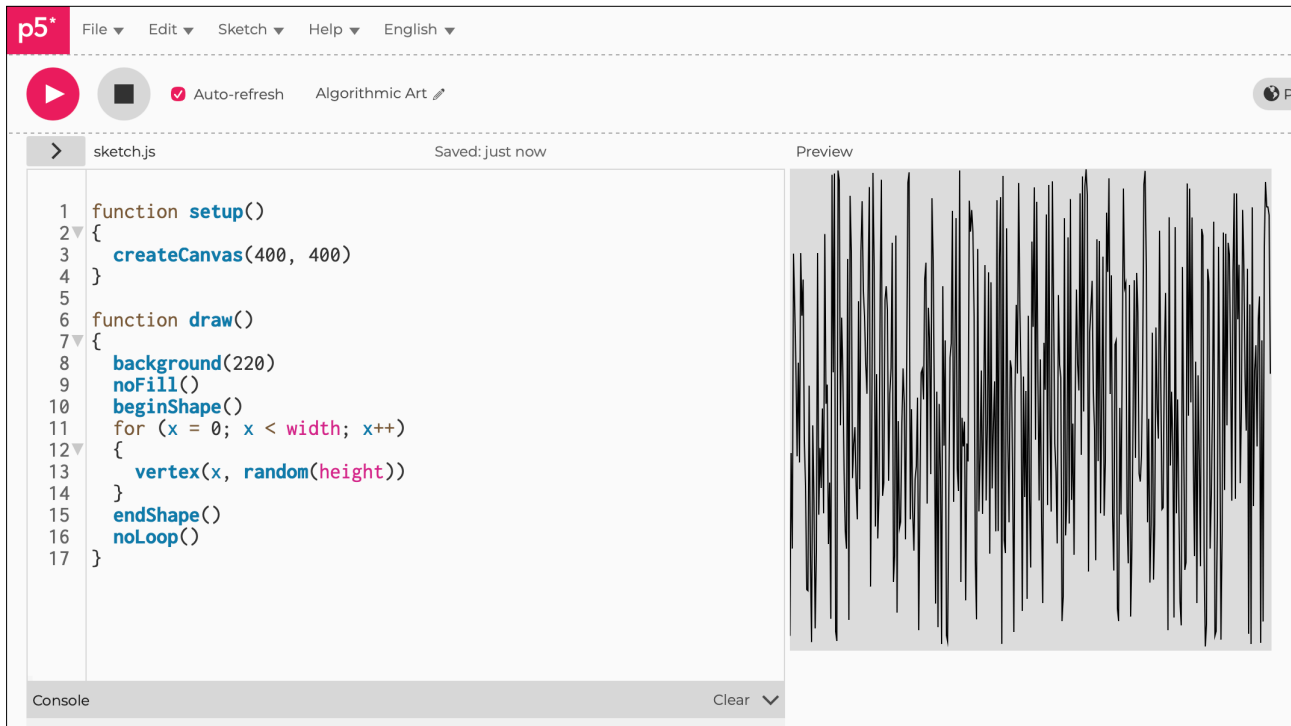
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  noFill()
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, random(height))
  }
  endShape()
  noLoop()
}
```

Notes

A very random line.

Figure E2.4





Sketch E2.5 introducing the perlin noise

We can introduce Perlin `noise()` rather than `random()`. We will get the `y` value at `0` and also multiply it by the `height` because noise returns a value between `0` and `1`. We will get a straight line somewhere between the top and the bottom of the canvas. Refresh and you will get a different value returned.

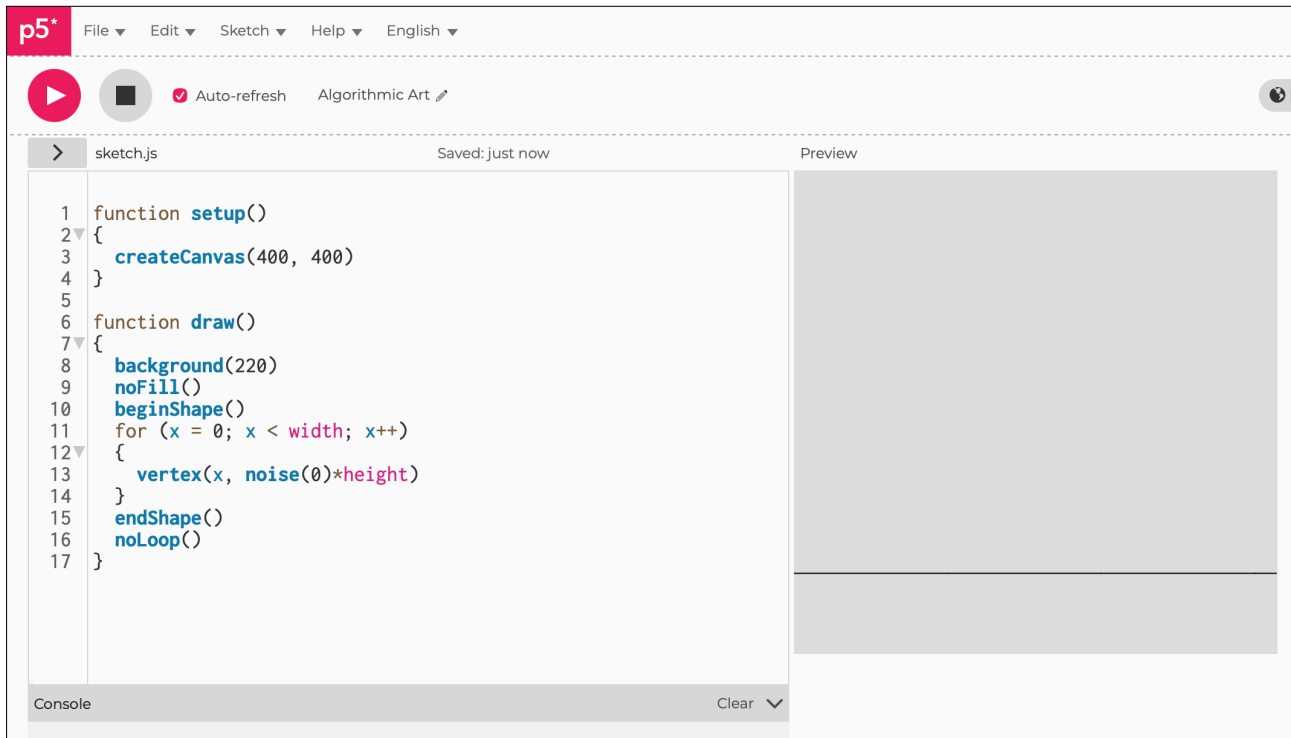
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  noFill()
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(0) * height)
  }
  endShape()
  noLoop()
}
```

Notes

We have line drawn based on where on the `noise()` timeline the `y` value is.

Figure E2.5





Sketch E2.6 xoff

But what we have is a static value of Perlin noise. We want to move through the values, so we need another variable called **xoff** (short for x offset). Previously, we just used the variable `time`. This is what Perlin **noise()** random looks like compared to pure **random()** as we had before. Nice and smooth.

```
let xoff = 0

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

function draw()
{
  background(220)
  noFill()
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(xoff) * height)
    xoff += 0.01
  }
  endShape()
  noLoop()
}
```

Notes

We have a visual representation of the Perlin `noise()` values.

Figure E2.6





Sketch E2.7 a bit static

This is still static, and we want to animate it so that you can see it flowing. We will replace the initial value with a variable called `inc` (short for increment).

```
let xoff = 0
let inc = 0.01

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

function draw()
{
  background(220)
  noFill()
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(xoff) * height)
    xoff += inc
  }
  endShape()
  noLoop()
}
```

Notes

Nothing changes yet.



Sketch E2.8 scrolling graph

Let's add another variable so that it doesn't begin at zero. We also comment out the `noLoop()` so we can animate. What you should get is a scrolling graph as Perlin `noise()` changes. We are effectively moving the `start` forward.

```
let xoff = 0
let inc = 0.01
let start = 0

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

function draw()
{
  background(220)
  noFill()
  xoff = start
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(xoff) * height)
    xoff += inc
  }
  endShape()
  start += inc
  // noLoop()
}
```

Notes

You should have the Perlin `noise()` line scrolling slowly across the canvas.

Challenge

Try different `inc` values.



Sketch E2.9 noise detail

We can control the Perlin `noise()` with `noiseDetail()`. It takes two arguments, the default is `4` and `0.5`. The first one is the `octave` and the second one is the `fall-off`. Think detail rather than worry about what they mean.

```
let xoff = 0
let inc = 0.01
let start = 0

function setup()
{
  createCanvas(400, 400)
  noiseDetail(4, 0.5)
}

function draw()
{
  background(220)
  noFill()
  xoff = start
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(xoff)*height)
    xoff += inc
  }
  endShape()
  start += inc
  // noLoop()
}
```

Notes

There is no appreciable difference immediately.

Challenge

Play with the values. I recommend between `1` and `24` for the first argument and `0` to `1` for the second.



Sketch E2.10 smoothing

Creating a much smoother graph.

```
let xoff = 0
let inc = 0.01
let start = 0

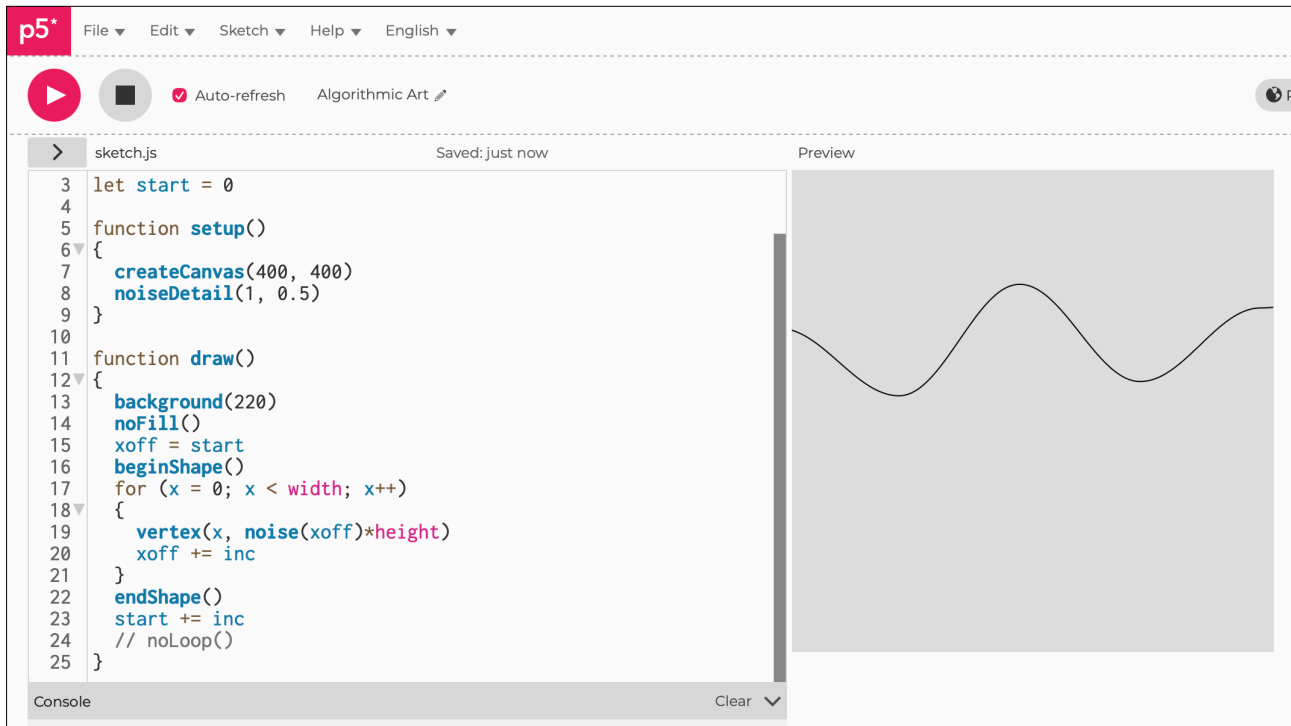
function setup()
{
  createCanvas(400, 400)
  noiseDetail(1, 0.5)
}

function draw()
{
  background(220)
  noFill()
  xoff = start
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(xoff)*height)
    xoff += inc
  }
  endShape()
  start += inc
  // noLoop()
}
```

Notes

A much smoother curve

Figure E2.10





Sketch E2.11 not so smooth

Less smooth.

```
let xoff = 0
let inc = 0.01
let start = 0

function setup()
{
  createCanvas(400, 400)
  noiseDetail(24, 0.5)
}

function draw()
{
  background(220)
  noFill()
  xoff = start
  beginShape()
  for (x = 0; x < width; x++)
  {
    vertex(x, noise(xoff)*height)
    xoff += inc
  }
  endShape()
  start += inc
  // noLoop()
}
```

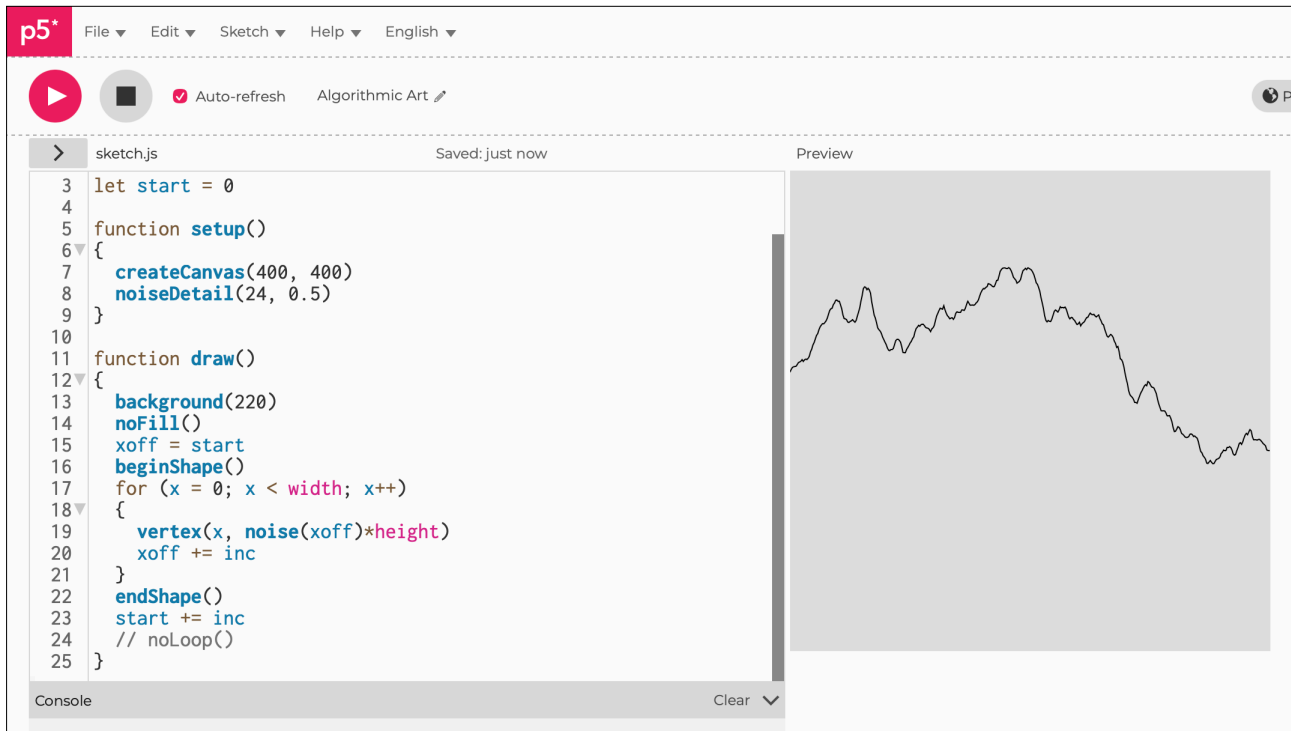
Notes

Now the opposite.

Challenge

See what happens to the circle movement or colour if you alter how much detail you give it, as a comparison.

Figure E2.11





2D perlin noise

So far, we have only explored **1-dimensional (1D) noise()**, which is the value of **y** as **x** moves along the **x-axis (time)**. For **2D Perlin noise()**, we need to consider all the surrounding values for any pixel on the canvas.

We are going to use the basis of the above sketch for the next section, which will generate a more interesting pattern for you to explore. This will form the basis for FlowFields later.



Sketch E2.12 time for 2D

! You will have no canvas.

Remove all the highlighted in blue (and commented out).

```
let xoff = 0
let inc = 0.01
// let start = 0

function setup()
{
  createCanvas(400, 400)
  // noiseDetail(24, 0.5)
}

function draw()
{
  // background(220)
  // noFill()
  // xoff = start
  // beginShape()
  for (x = 0; x < width; x++)
  {
    // vertex(x, noise(xoff)*height)
    // xoff += inc
  }
  // endShape()
  // start += inc
  // noLoop()
}
```

Notes

There is nothing to see not even a canvas.



Sketch E2.13 are we ready?

You should have something like this.

```
let xoff = 0
let inc = 0.01

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

function draw()
{
  for (x = 0; x < width; x++)
  {

  }
}
```

Notes

We have no canvas.



Sketch E2.14 loading the pixels

We are going to use a function called `loadPixels()` to examine whichever pixels we want to look at (in this case, all of them) and then `updatePixels()` after we have done something to them. We cycle through every pixel, hence the `y for()` loop.

```
let xoff = 0
let inc = 0.01

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

function draw()
{
  loadPixels()
  for (x = 0; x < width; x++)
  {
    for (y = 0; y < height; y++)
    {

    }
  }
  updatePixels()
}
```

Notes

Again, no canvas; be patient.



Sketch E2.15 four channels

There are **four** channels: **red**, **green**, **blue**, and **alpha** for each pixel. So we are going to set the red to full (255) with no green or blue and full alpha (255). We have to set the `pixelDensity()` to 1 because of my display (Mac Retina); you may not need it depending on your monitor/screen.

```
let xoff = 0
let inc = 0.01
let index

function setup()
{
  createCanvas(400, 400)
  pixelDensity(1)
}

function draw()
{
  loadPixels()
  for (x = 0; x < width; x++)
  {
    for (y = 0; y < height; y++)
    {
      index = (x + y * width) * 4
      pixels[index + 0] = 255
      pixels[index + 1] = 0
      pixels[index + 2] = 0
      pixels[index + 3] = 255
    }
  }
  updatePixels()
}
```

Notes

What you will see is a red canvas. Play with the other values and see how it works.

Challenge

Create other colours.

Figure E2.15





Sketch E2.16 random static

If you randomise the values for **RGB**, you get a random image (add **noLoop()** to stop it flickering). However, each pixel has no correlation with any of the surrounding pixels; every pixel is random in isolation.

```
let xoff = 0
let inc = 0.01
let index
let val

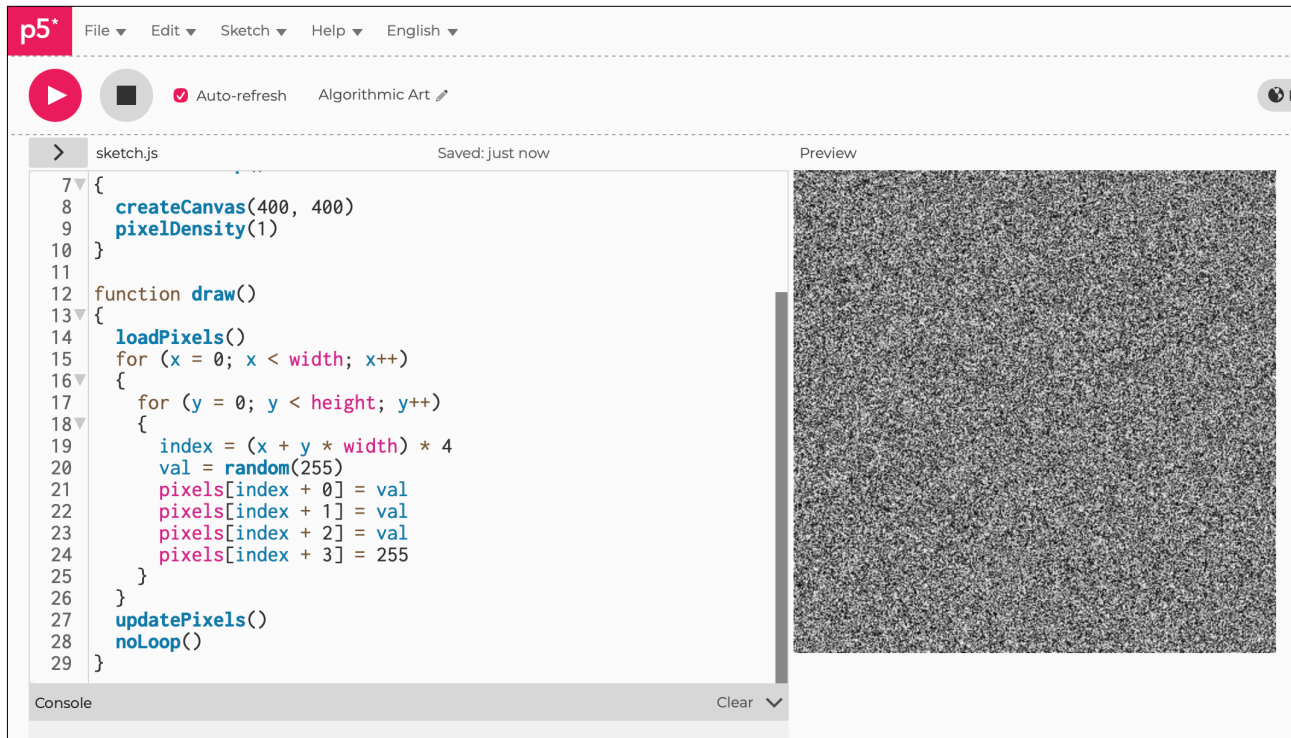
function setup()
{
  createCanvas(400, 400)
  pixelDensity(1)
}

function draw()
{
  loadPixels()
  for (x = 0; x < width; x++)
  {
    for (y = 0; y < height; y++)
    {
      index = (x + y * width) * 4
      val = random(255)
      pixels[index + 0] = val
      pixels[index + 1] = val
      pixels[index + 2] = val
      pixels[index + 3] = 255
    }
  }
  updatePixels()
  noLoop()
}
```

Notes

What do you think would happen if we gave each pixel its separate random value?

Figure E2.16





Sketch E2.17 make it noisy

Next step is to introduce `noise()` so that we can have the situation where each pixel is related to the surrounding pixels. We first replace `random()` with `noise()` and multiply by 255 because `noise()` returns a value between 0 and 1, what we want are values between 0 and 255. Every time you refresh the sketch, you get a different shade of grey.

```
let xoff = 0
let inc = 0.01
let index
let val

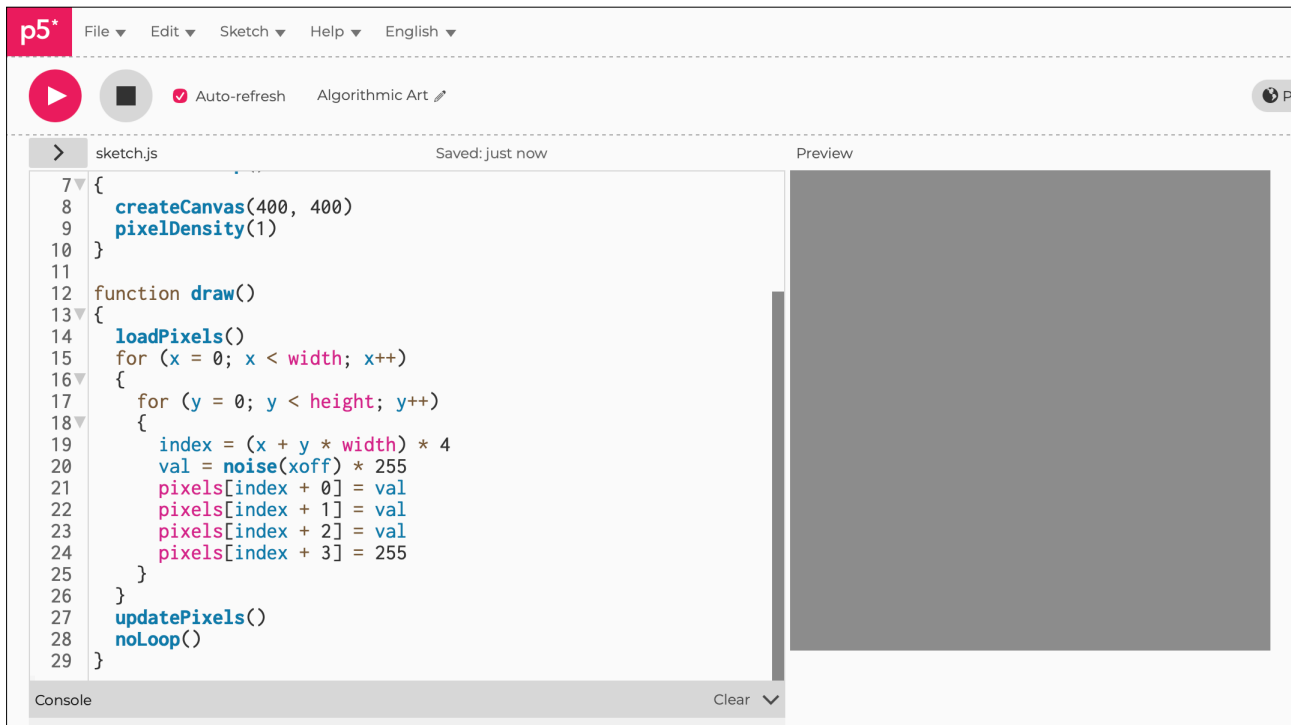
function setup()
{
  createCanvas(400, 400)
  pixelDensity(1)
}

function draw()
{
  loadPixels()
  for (x = 0; x < width; x++)
  {
    for (y = 0; y < height; y++)
    {
      index = (x + y * width) * 4
      val = noise(xoff) * 255
      pixels[index + 0] = val
      pixels[index + 1] = val
      pixels[index + 2] = val
      pixels[index + 3] = 255
    }
  }
  updatePixels()
  noLoop()
}
```

Notes

You get a single colour of some grey value.

Figure E2.17





Sketch E2.18 a wee amount

If we change each pixel **xoff** value by a small amount, we get the following effect (which is still not it, by the way).

```
let xoff = 0
let inc = 0.01
let index
let val

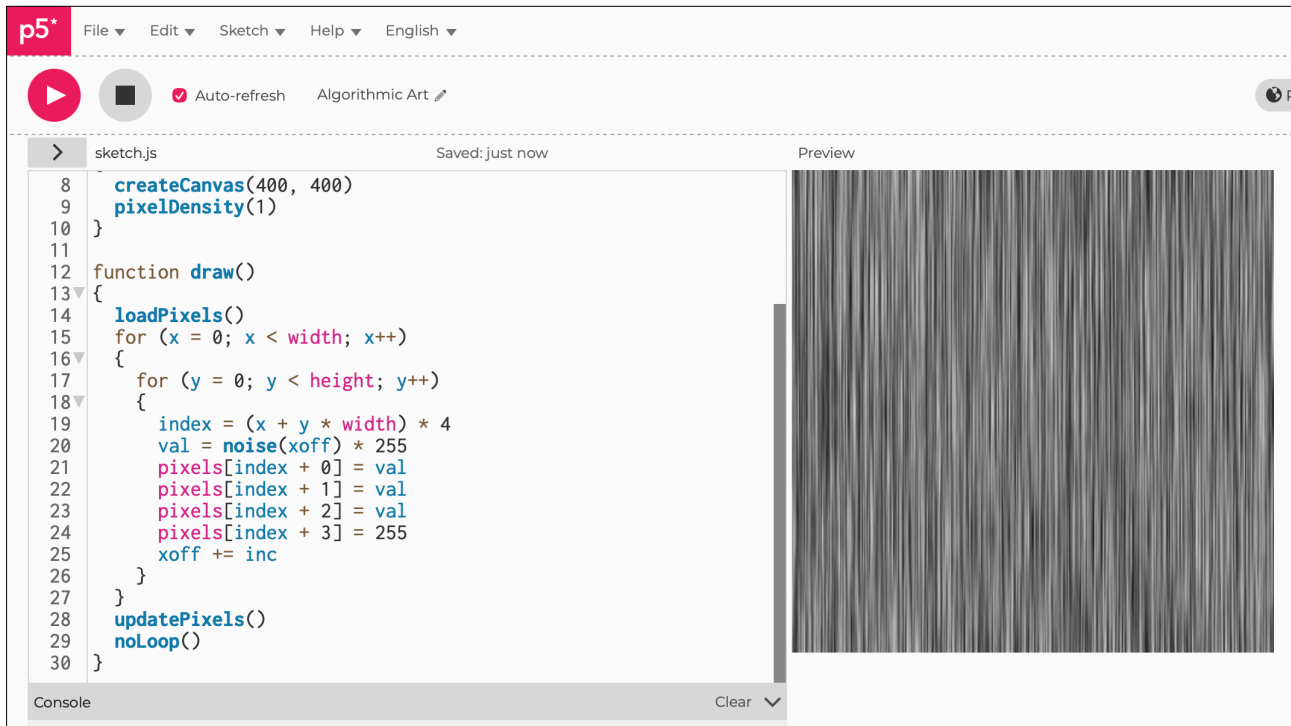
function setup()
{
  createCanvas(400, 400)
  pixelDensity(1)
}

function draw()
{
  loadPixels()
  for (x = 0; x < width; x++)
  {
    for (y = 0; y < height; y++)
    {
      index = (x + y * width) * 4
      val = noise(xoff) * 255
      pixels[index + 0] = val
      pixels[index + 1] = val
      pixels[index + 2] = val
      pixels[index + 3] = 255
      xoff += inc
    }
  }
  updatePixels()
  noLoop()
}
```

Notes

We have our vertical Perlin noise().

Figure E2.18





Sketch E2.19 reversing the loops

Presently, we have the `y loops()` inside the `x`, but they need to be the other way round. We do a `row` first, so `y` needs to stay static as we move through from left to right.

```
let xoff = 0
let inc = 0.01
let index
let val

function setup()
{
  createCanvas(400, 400)
  pixelDensity(1)
}

function draw()
{
  loadPixels()
  for (y = 0; y < height; y++)
  {
    for (x = 0; x < width; x++)
    {
      index = (x + y * width) * 4
      val = noise(xoff) * 255
      pixels[index + 0] = val
      pixels[index + 1] = val
      pixels[index + 2] = val
      pixels[index + 3] = 255
      xoff += inc
    }
  }
  updatePixels()
  noLoop()
}
```

Notes

What is happening is that the `Perlin noise()` value is changing as it goes across the canvas and then continues at the start of the next row of pixels. So what you are seeing is `Perlin noise()`, but it has no relationship with the value above or below.

Figure E2.19





Sketch E2.20 yoff

We want a **yoff** to work downwards as well, but when it moves down a row, we want the **xoff** to reset, not carry on with the **Perlin noise()** increments.

```
let xoff = 0
let yoff = 0
let inc = 0.01
let index
let val

function setup()
{
  createCanvas(400, 400)
  pixelDensity(1)
}

function draw()
{
  loadPixels()
  for (y = 0; y < height; y++)
  {
    xoff = 0
    for (x = 0; x < width; x++)
    {
      index = (x + y * width) * 4
      val = noise(xoff, yoff) * 255
      pixels[index + 0] = val
      pixels[index + 1] = val
      pixels[index + 2] = val
      pixels[index + 3] = 255
      xoff += inc
    }
    yoff += inc
  }
  updatePixels()
  noLoop()
}
```

Notes

What you get is the image where each pixel has now some relationship with its neighbouring ones. If the lines of code above aren't obvious, this is where you take the time to work through slowly what is happening to each pixel as you move across the canvas and then onto the next row of pixels. This is where the logic of coding comes in.

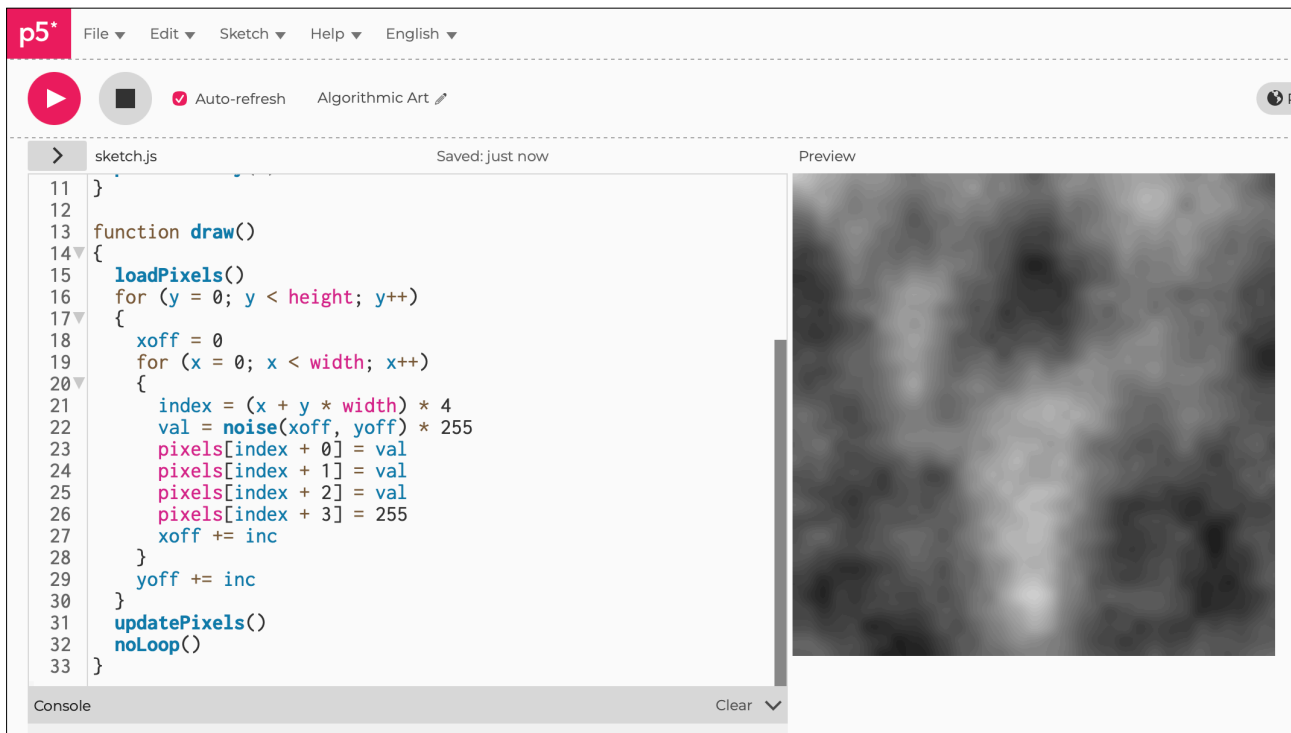
Challenges

Change the `inc` value to see the effect.

Add noise values for each colour (**RGB**).

Use `noiseDetail()` to see what difference you can make to the image.

Figure E2.20



Algorithmic Art

Module E

Unit #4

Smoke and Fire





Module E Unit #4: smoke and fire

Creating a simple smoke or fire effect using a simple particle system with `p5.js`. Particle systems are a great way to create effects that have a degree of realism or to create abstract effects. It can be memory intensive so it could start to slow down dramatically if not managed well.

This unit shows what is possible as an example. You could create something else based on the concepts introduced. There is so much more to particle systems but this is a nice one to start with.



Sketch E3.1 starting sketch

This is our basic sketch; this gives us a nice dark background for a change.

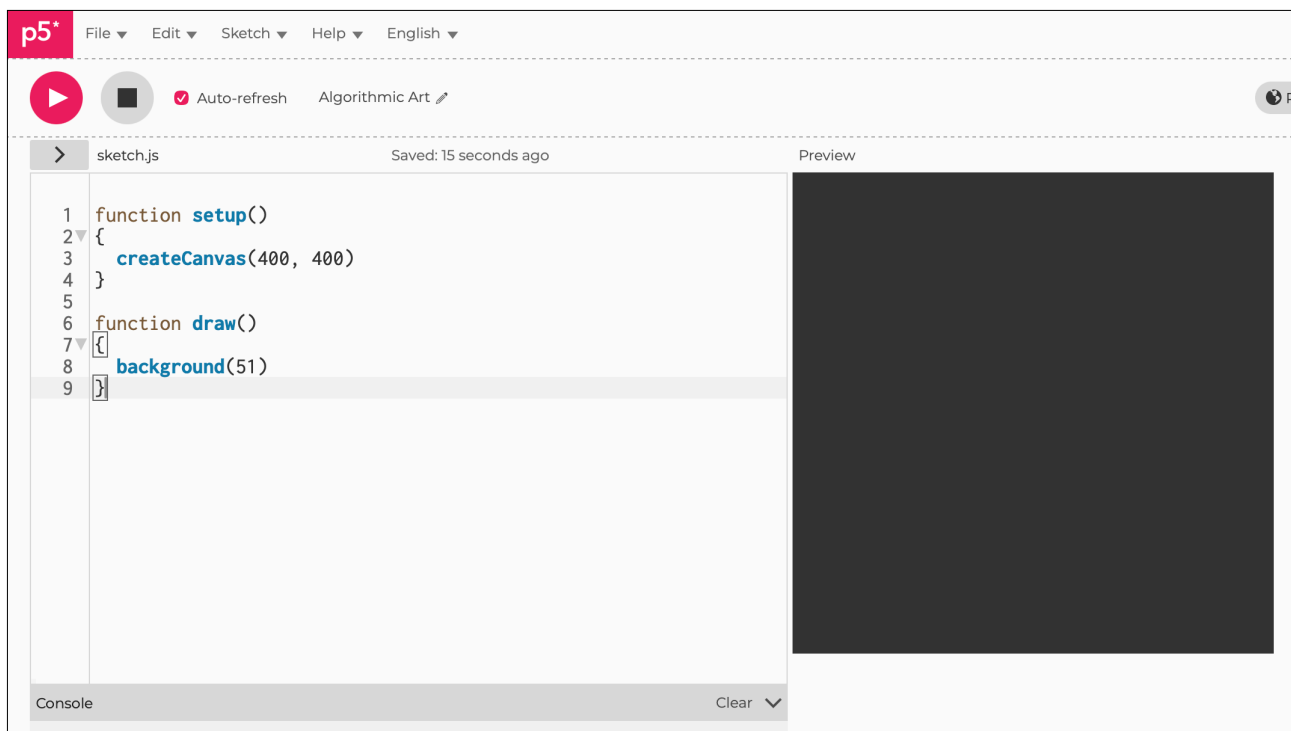
```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(51)
}
```

Notes

Nothing dramatic, a nice dark canvas.

Figure E3.1





Sketch E3.2 adding a particle class

We are going to add a particle class with a constructor function. We want the particles to start near the bottom and in the middle, then travel up and fade.

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

```
function draw()
{
  background(51)
}
```

```
class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
  }
}
```

Notes

Nothing has changed.



Sketch E3.3 adding move and show

We want a `show()` function and a `move()` function.

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

```
function draw()
{
  background(51)
}
```

```
class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
  }
```

```
  show()
  {

  }
```

```
  move()
  {

  }
```

```
}
```

Notes

This adds nothing new to the canvas, just building the class.



Sketch E3.4 draw a circle

We draw a circle at these **x** and **y** co-ordinates filling it with white (**255**).

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

function draw()
{
  background(51)
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
  }

  show()
  {
    stroke(255)
    fill(255, 10)
    circle(this.x, this.y, 16)
  }

  move()
  {
  }
}
```

Notes

We are describing what the particle will look like.



Sketch E3.5 particle p

Let's create one particle called **p** so we can see it.

```
let p

function setup()
{
  createCanvas(400, 400)
  p = new Particle()
}

function draw()
{
  background(51)
  p.show()
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
  }

  show()
  {
    stroke(255)
    fill(255, 10)
    circle(this.x, this.y, 16)
  }

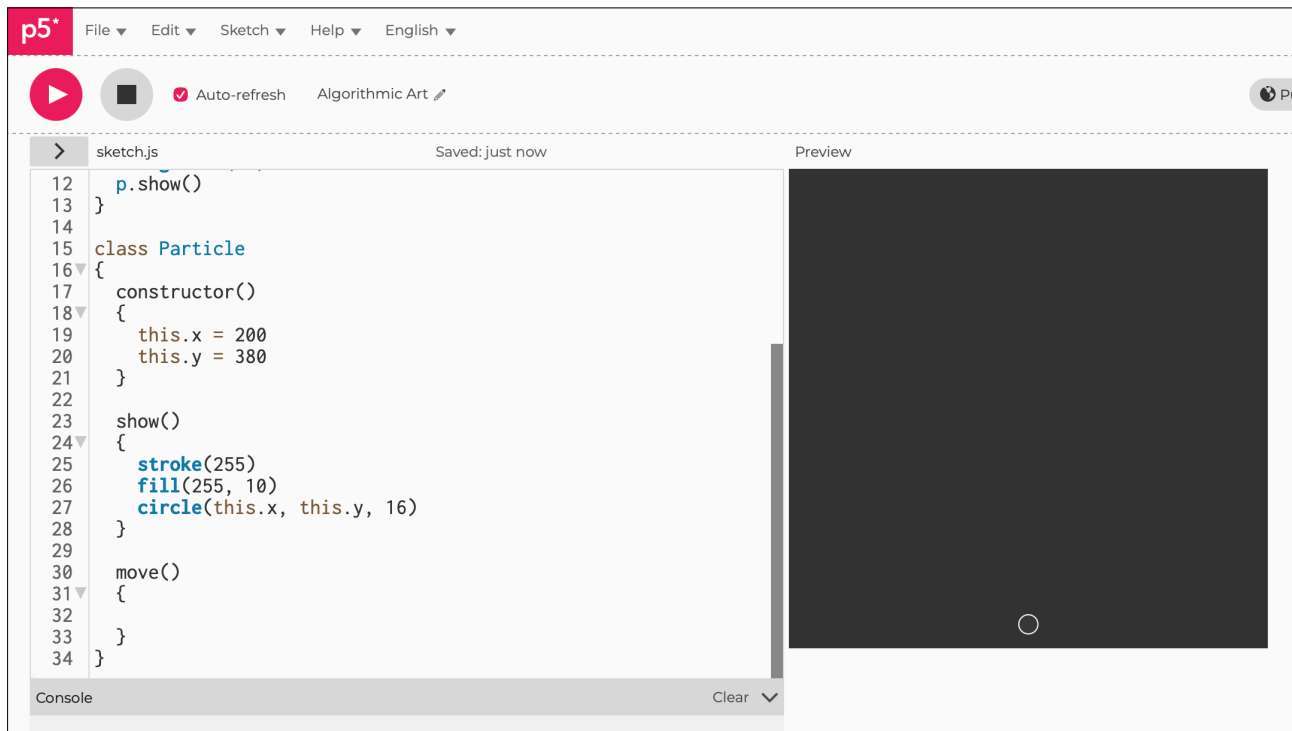
  move()
  {

  }
}
```

Notes

At last, we have something to see!

Figure E3.5





Sketch E3.6 an array of particles

We want to create many particles, so we create an array and `push()` them into the array using a `for()` loop. We will start with just one particle.

```
let p
let particles = []

function setup()
{
  createCanvas(400, 400)
  p = new Particle()
  particles.push(p)
}

function draw()
{
  background(51)
  for (let i = 0; i < particles.length; i++)
  {
    particles[i].show()
  }
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
  }

  show()
  {
    stroke(255)
    fill(255, 10)
    circle(this.x, this.y, 16)
  }
}
```

```
move()  
{  
  
}  
}
```

Notes

You should have exactly the same as before.



Sketch E3.7 move the particle

Let's give it some movement, velocity. We create two variables called `velX` and `velY`.

```
let p
let particles = []

function setup()
{
  createCanvas(400, 400)
  p = new Particle()
  particles.push(p)
}

function draw()
{
  background(51)
  for (let i = 0; i < particles.length; i++)
  {
    particles[i].move()
    particles[i].show()
  }
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
    this.velX = random(-1, 1)
    this.velY = random(-5, -1)
  }

  show()
  {
    stroke(255)
    fill(255, 10)
    circle(this.x, this.y, 16)
```

```
}  
  
move()  
{  
  this.x += this.velX  
  this.y += this.velY  
}  
}
```

Notes

The particle (circle) will drift upwards. Every time you replay the sketch, the particle will have a different (random) velocity.

The `+=` symbol is shorthand for:

```
this.x = this.x + this.velX  
this.y = this.y + this.velY
```



Sketch E3.8 many particles

We want more than one particle. To do this quickly, we cut and paste the following code from `setup()` into `draw()`.

! You delete those lines of code in `setup()`.

```
let p
let particles = []

function setup()
{
  createCanvas(400, 400)
  // p = new Particle()
  // particles.push(p)
}

function draw()
{
  background(51)
  p = new Particle()
  particles.push(p)
  for (let i = 0; i < particles.length; i++)
  {
    particles[i].move()
    particles[i].show()
  }
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
    this.velX = random(-1, 1)
    this.velY = random(-5, -1)
  }
}
```

```

show()
{
  stroke(255)
  fill(255, 10)
  circle(this.x, this.y, 16)
}

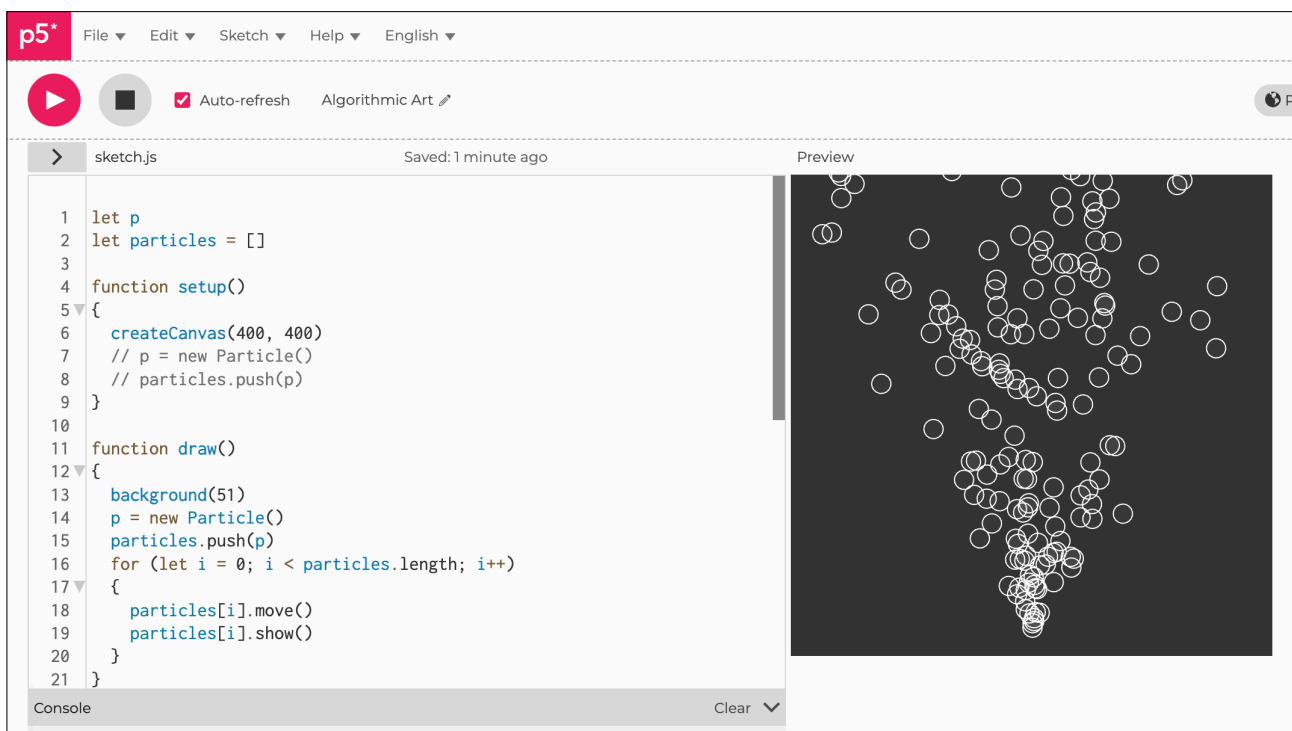
move()
{
  this.x += this.velX
  this.y += this.velY
}
}

```

Notes

You get a stream of particles.

Figure E3.8





Sketch E3.9 no smoke without fire

This is a pleasing effect but still not like smoke or fire. So we will introduce some **alpha**. This is so we can fade it and remove it altogether when it has disappeared; otherwise, the programme will run slower and slower as we create more and more articles.

! We replace **stroke(255)** with **noStroke()**.

```
let p
let particles = []

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

function draw()
{
  background(51)
  p = new Particle()
  particles.push(p)
  for (let i = 0; i < particles.length; i++)
  {
    particles[i].move()
    particles[i].show()
  }
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
    this.velX = random(-1, 1)
    this.velY = random(-5, -1)
    this.alpha = 255
  }
}
```

```

show()
{
  noStroke()
  fill(255, this.alpha)
  circle(this.x, this.y, 16)
}

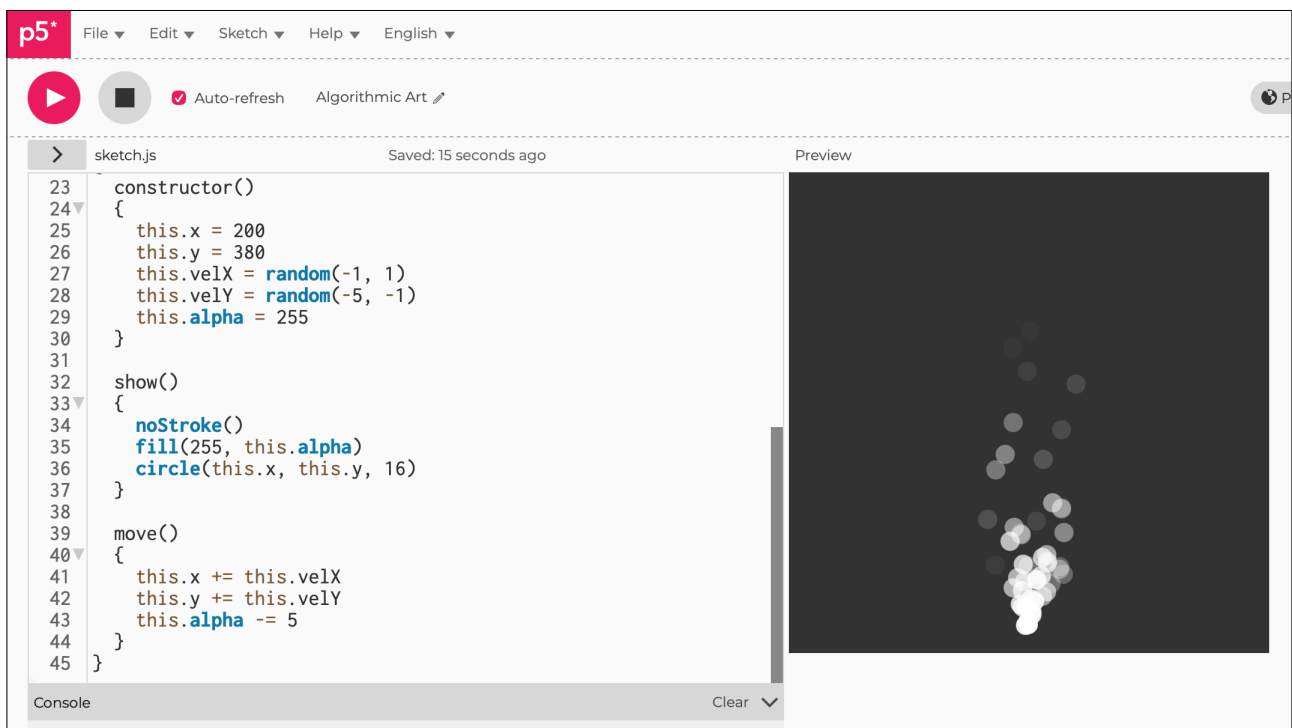
move()
{
  this.x += this.velX
  this.y += this.velY
  this.alpha -= 5
}
}

```

Notes

The line of code: `this.alpha -= 5` means `this.alpha = this.alpha - 5`.

Figure E3.9





Sketch E3.10 too many particles

We need to address the issue. If we include a line to see how many particles we have created and still exist, we can use `console.log()` in `draw()`. We can give it the parameter of `particles.length`, and as you let it run for a short while, the number just keeps on going up.

```
let p
let particles = []

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

function draw()
{
  background(51)
  p = new Particle()
  particles.push(p)
  for (let i = 0; i < particles.length; i++)
  {
    particles[i].move()
    particles[i].show()
  }
  console.log(particles.length)
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
    this.velX = random(-1, 1)
    this.velY = random(-5, -1)
    this.alpha = 255
  }
}
```

```

show()
{
  noStroke()
  fill(255, this.alpha)
  circle(this.x, this.y, 16)
}

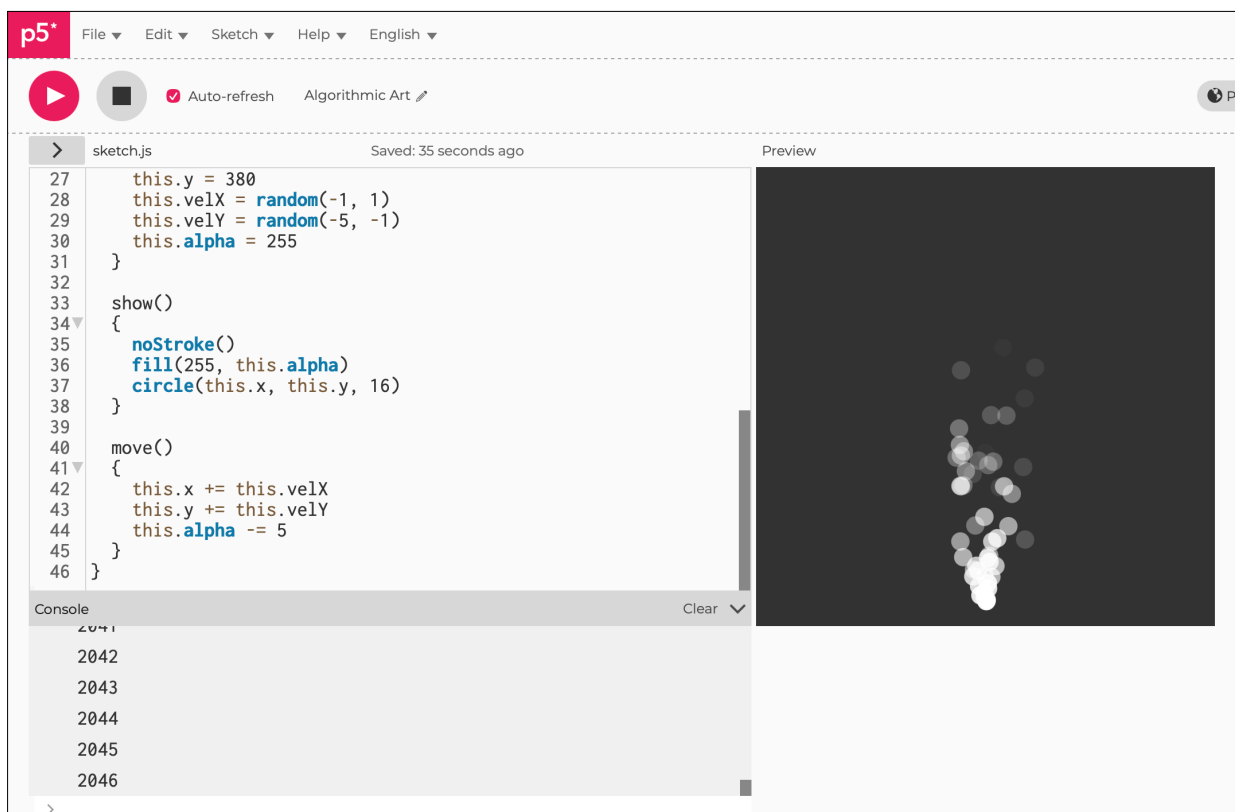
move()
{
  this.x += this.velX
  this.y += this.velY
  this.alpha -= 5
}
}

```

Notes

After running this for a few minutes, you will get the idea. We will keep it there for the moment and delete it later after we have removed the particles that have faded to zero. You will notice that it is fine for a few minutes and then starts to run very slowly.

Figure E3.10





Sketch E3.11 remove dead particles

To remove the particles that are less than zero (**fade**), we use the **splice()** function in a new function that checks to see if it is zero or less. We will call this function **finished()**.

```
let p
let particles = []

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

function draw()
{
  background(51)
  p = new Particle()
  particles.push(p)
  for (let i = 0; i < particles.length; i++)
  {
    particles[i].move()
    particles[i].show()
    if (particles[i].finished())
    {
      particles.splice(i, 1)
    }
  }
  console.log(particles.length)
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
    this.velX = random(-1, 1)
    this.velY = random(-5, -1)
```

```
    this.alpha = 255
  }

  show()
  {
    noStroke()
    fill(255, this.alpha)
    circle(this.x, this.y, 16)
  }

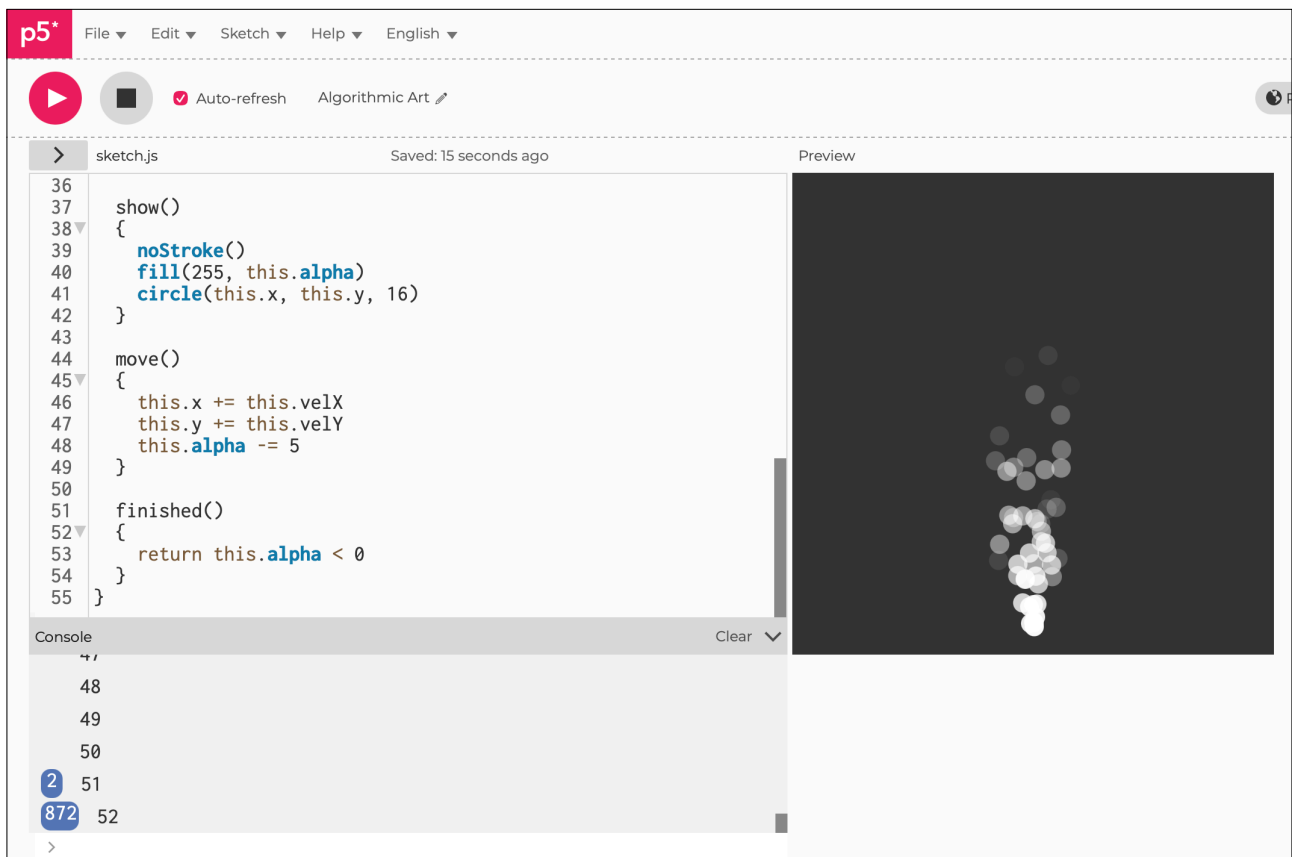
  move()
  {
    this.x += this.velX
    this.y += this.velY
    this.alpha -= 5
  }

  finished()
  {
    return this.alpha < 0
  }
}
```

Notes

The code: `return this.alpha < 0` means the function `finished()` will return `true` when the alpha is less than zero. Think boolean (true/false). In my case, the counter stops at 52.

Figure E3.11





Sketch E3.12 backwards array

Although this looks fine, there is a problem. When you are using `splice()`, we are doing something a bit odd when we delete something from an array counting from the front. This can mean that you jump past the next one after you have deleted a particle.

The way around it is to start from the end and check through the array backwards. We start with the last element in the array, which is at `particles.length - 1` because the numbering system starts from `0` (not 1), count backwards (`i--`) while `i` is greater than zero (`> 0`). It may take a bit of thinking through, but the logic is very sound. It is a good idea to explore the nature of arrays and how they are spliced.

```
let p
let particles = []

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

function draw()
{
  background(51)
  p = new Particle()
  particles.push(p)
  for (let i = particles.length - 1; i > 0; i--)
  {
    particles[i].move()
    particles[i].show()
    if (particles[i].finished())
    {
      particles.splice(i, 1)
    }
  }
  console.log(particles.length)
}

class Particle
{
  constructor()
```

```

{
  this.x = 200
  this.y = 380
  this.velX = random(-1, 1)
  this.velY = random(-5, -1)
  this.alpha = 255
}

show()
{
  noStroke()
  fill(255, this.alpha)
  circle(this.x, this.y, 16)
}

move()
{
  this.x += this.velX
  this.y += this.velY
  this.alpha -= 5
}

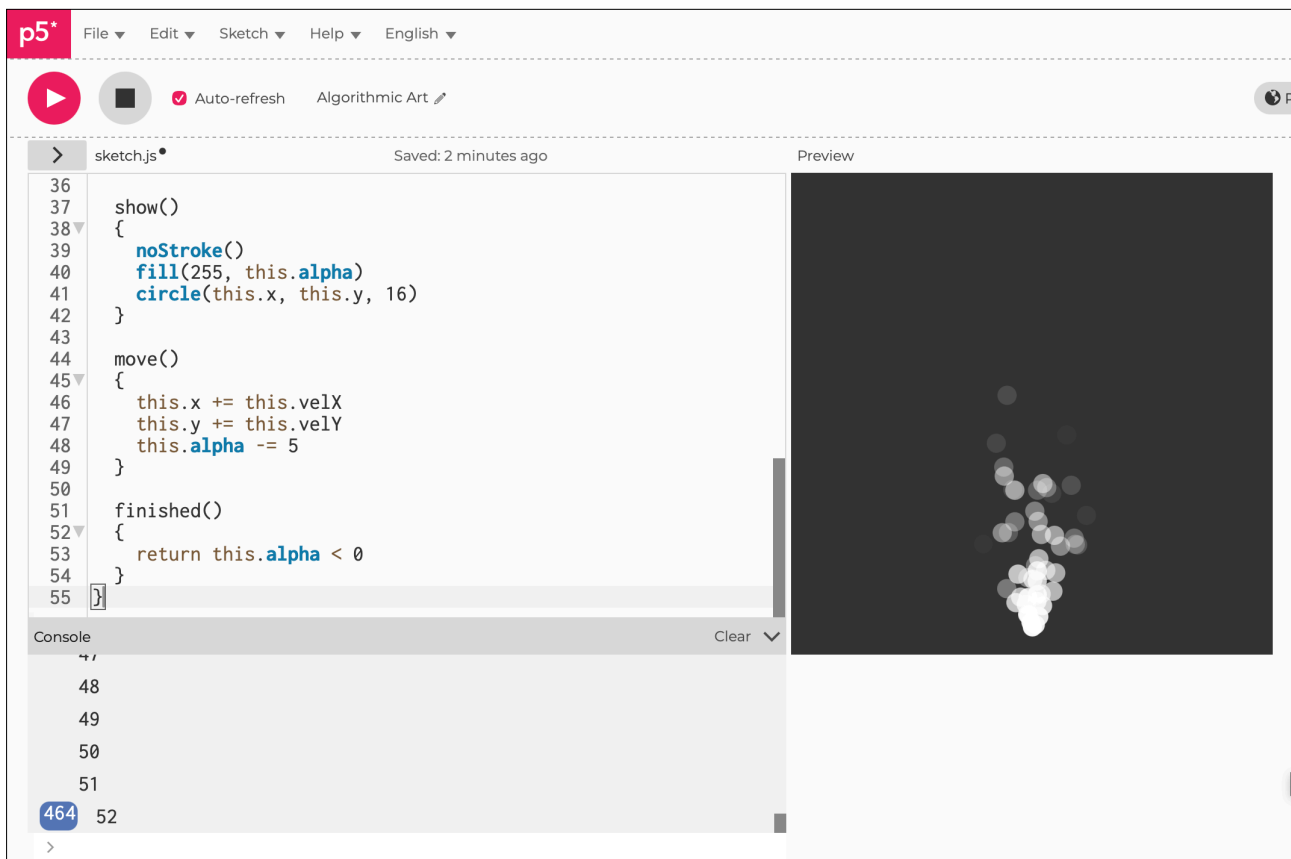
finished()
{
  return this.alpha < 0
}
}

```

Notes

When you run this with the `console.log()`, you will see that it has the same effect; you may see a slight improvement. It didn't stick at **51**, much cleaner and better coding.

Figure E3.12





Sketch E3.13 adding more particles

We only added **one** particle per frame in `draw()`, now let's add, say, **five** particles per frame.

! Remove `console.log()`.

```
let p
let particles = []

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

function draw()
{
  background(51)
  for (let i = 0; i < 5; i++)
  {
    p = new Particle()
    particles.push(p)
  }
  for (let i = particles.length - 1; i > 0; i--)
  {
    particles[i].move()
    particles[i].show()
    if (particles[i].finished())
    {
      particles.splice(i, 1)
    }
  }
  // console.log(particles.length)
}

class Particle
{
  constructor()
  {
```

```
this.x = 200
this.y = 380
this.velX = random(-1, 1)
this.velY = random(-5, -1)
this.alpha = 255
}

show()
{
  noStroke()
  fill(255, this.alpha)
  circle(this.x, this.y, 16)
}

move()
{
  this.x += this.velX
  this.y += this.velY
  this.alpha -= 5
}

finished()
{
  return this.alpha < 0
}
}
```

Notes

It looks so much better.

Figure E3.13





Sketch E3.14 some adaptations

Just some little adjustments.

```
let p
let particles = []

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

function draw()
{
  background(51)
  for (let i = 0; i < 5; i++)
  {
    p = new Particle()
    particles.push(p)
  }
  for (let i = particles.length - 1; i > 0; i--)
  {
    particles[i].move()
    particles[i].show()
    if (particles[i].finished())
    {
      particles.splice(i, 1)
    }
  }
}

class Particle
{
  constructor()
  {
    this.x = 200
    this.y = 380
    this.velX = random(-1, 1)
    this.velY = random(-6, -2)
```

```
this.alpha = 255
this.radius = 20
}

show()
{
  noStroke()
  fill(255, this.alpha)
  circle(this.x, this.y, this.radius)
}

move()
{
  this.x += this.velX
  this.y += this.velY
  this.alpha -= 5
  this.radius -= 0.001
}

finished()
{
  return this.alpha < 0
}
}
```

Notes

Even better.

Challenges

1. Change the diameter randomly.
2. Add some colour and then fade.

Figure E3.14



Algorithmic
Art
Module E
Unit #4
Fireworks
Display Part 1



Module E Unit #4: fireworks display Part 1

Creating a firework display using particles. This builds on the previous unit covering simple particle systems. This is a bit all over the place in terms of simple tutorials; that doesn't mean that it is no good, rather it will need refactoring in the future. The final result is brilliant, but we do go round the houses a little bit.

We will be using two classes for this unit. This means there is going to be a lot of back and forth between the main sketch and the two classes. I had originally put them in separate files, but that adds a level of complexity. I might provide a video to explain how to do that, which would make the finished product nice and tidy.

This unit is quite long. It is a bit like painting by numbers. What it does do is introduce you to key concepts as you go along. What you get is something quite beautiful (in my opinion). Much of this unit is about physics as much as anything, so you may not feel that it is directly relevant to Algorithmic Art yet, the final result is.

If you are not in the mood for a long, drawn-out example, then feel free to skip this (and the one after) unit. My argument for including this (and the following one) is that muscle is memory. You will learn by doing, and it will become intuitive. The particles and Perlin noise will be covered in more depth in a series of workbooks on Physics Simulation (when I have published them!).



Sketch E4.1 starting sketch

! New sketch

Our starting sketch in the normal place.

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

function draw()
{
  background(100)
}
```

Notes

Nice and normal, a slightly darker canvas.



Sketch E4.2 create vectors

In a previous module, you were introduced to classes, so we create a particle class with a constructor that takes two variables (**x**, **y**) with three vectors: position (**pos**), velocity (**vel**), and acceleration (**acc**)

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

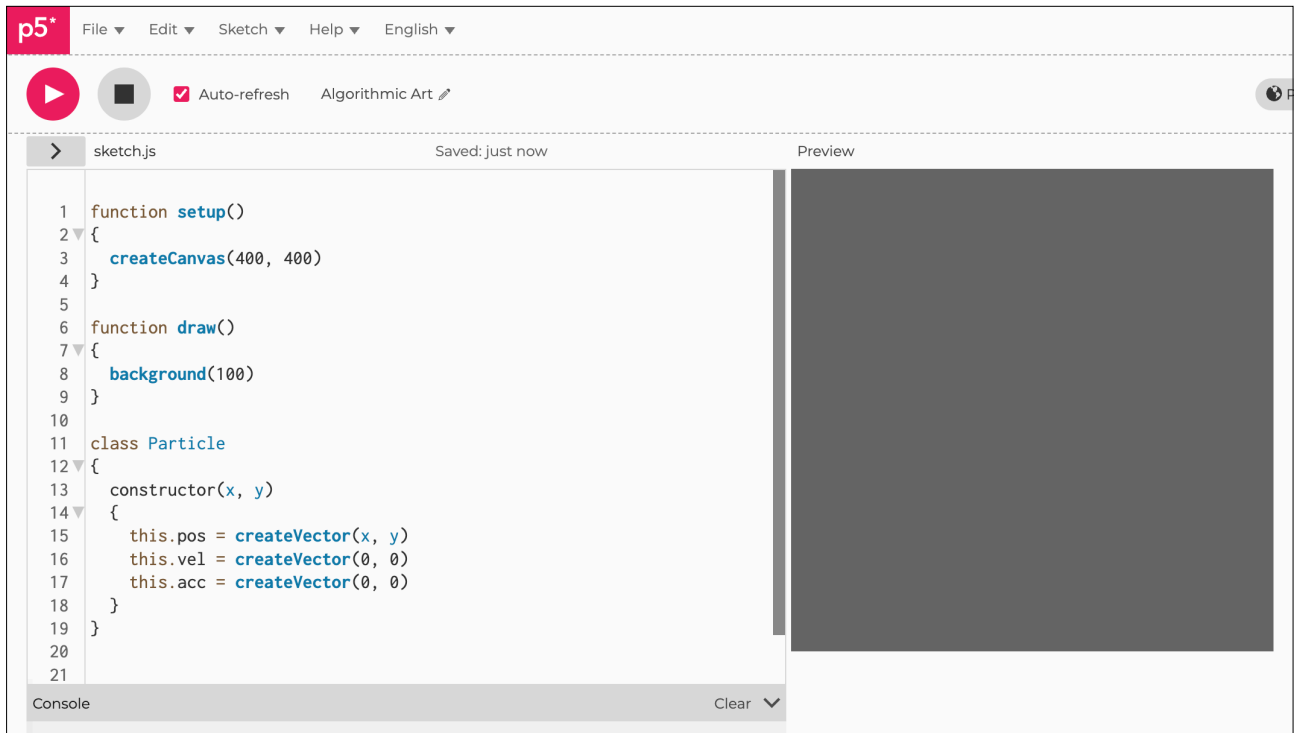
function draw()
{
  background(100)
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }
}
```

Notes

Creating the three vectors we need.

Figure E4.2





Sketch E4.3 move() function

We create a `move()` function which adds the velocity to the position of the particle.

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

function draw()
{
  background(100)
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }

  move()
  {
    this.pos.add(this.vel)
  }
}
```

Notes

Nothing new to see here



Sketch E4.4 creating a force

To get the acceleration, we create a force. It will be fired upwards and fall under gravity; both are forces. We create a variable called force within which we will apply it, hence the function `applyForce()`. This is so the acceleration can accumulate.

From the basic equation $\text{force} = \text{mass} \times \text{acceleration}$, but we can ignore mass, so $\text{force} = \text{acceleration}$.

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

function draw()
{
  background(100)
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.pos.add(this.vel)
  }
}
```

Notes

Still the same grey canvas.



Sketch E4.5 adding the force

Adding this to the `move()` function. We multiply the acceleration (`acc`) by zero so that on each frame the acceleration is constant. (You can comment out this line to see what it does later.)

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

function draw()
{
  background(100)
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.mult(0)
  }
}
```



Sketch E4.6 draw the particle

We now need to draw the particle, and so we create a `show()` function.

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

function draw()
{
  background(100)
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.mult(0)
  }
}
```

```
}
```

Notes

There is a reason why you cannot see anything and it isn't your eyesight.



Sketch E4.7 size and colour

You will notice that nothing happens when you run the sketch. So we need to move back to the main sketch. Here we give the particle a white colour (255) and a size (4). To test that this works, we add a variable called **firework** and use the **Particle** class to draw it.

```
let firework

function setup()
{
  createCanvas(400, 400)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(200, 200)
}

function draw()
{
  background(100)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
```

```

{
  this.acc.add(force)
}

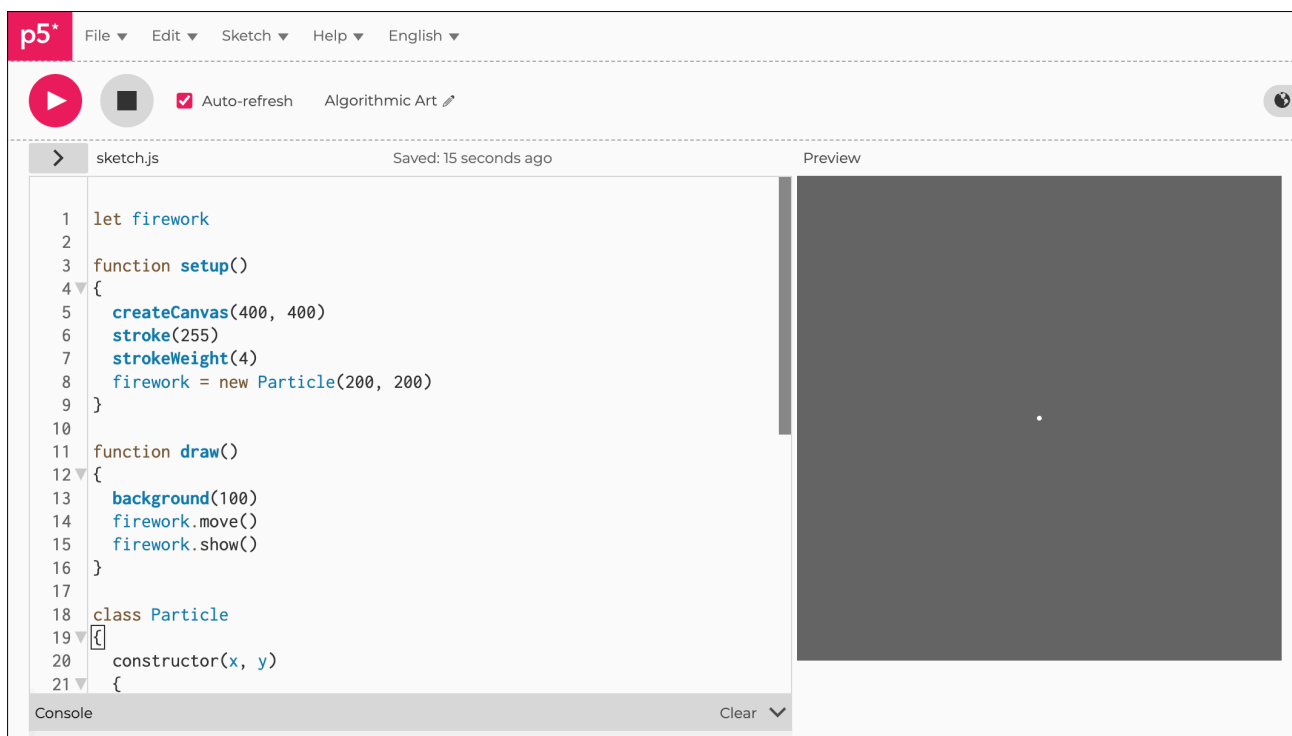
move()
{
  this.vel.add(this.acc)
  this.pos.add(this.vel)
  this.acc.mult(0)
}
}

```

Notes

A new particle is drawn in the centre, something to see at last.

Figure E4.7





Sketch E4.8 moving upwards

Let's get it moving upwards from a new starting position on the ground at random.

```
let firework

function setup()
{
  createCanvas(400, 400)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, 0)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }
}
```

```

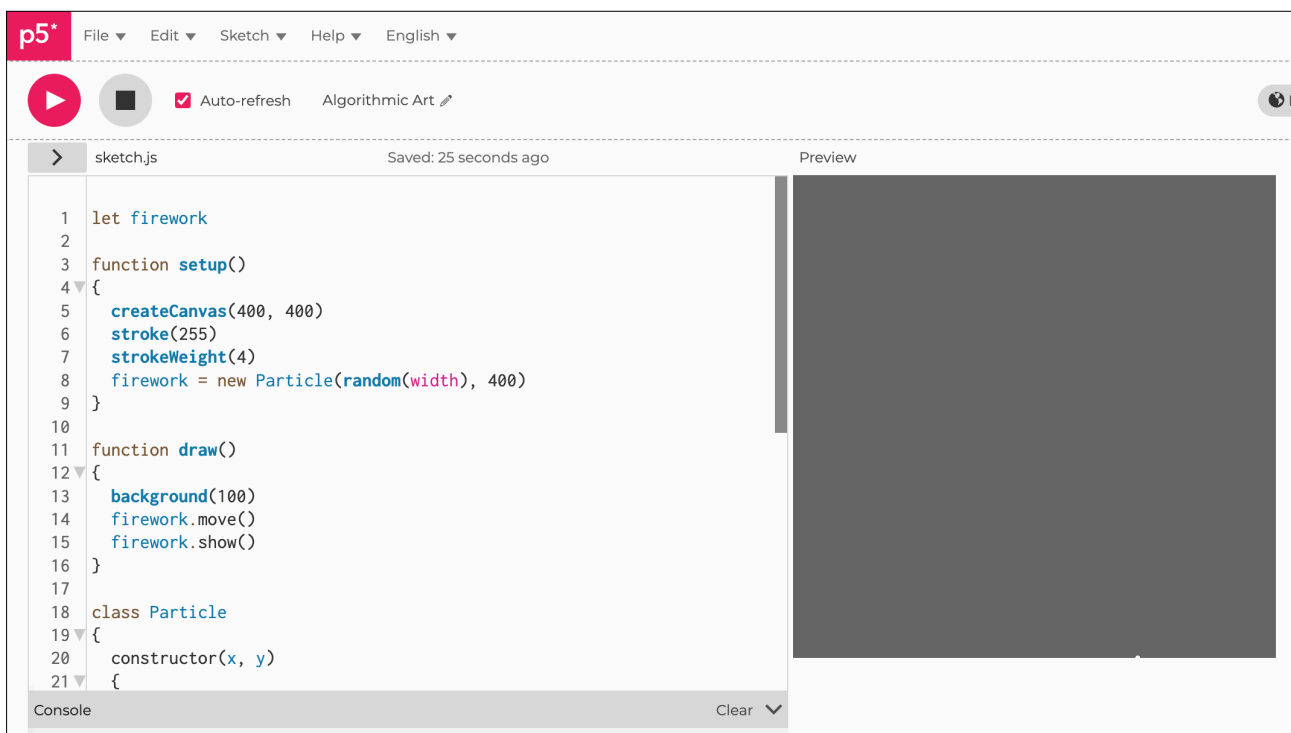
move()
{
  this.vel.add(this.acc)
  this.pos.add(this.vel)
  this.acc.mult(0)
}
}

```

Notes

You can just see it sitting at the bottom.

Figure E4.8





Sketch E4.9 upwards velocity

In the **Particle** class, we need to give an upwards velocity of **-5**.

```
let firework

function setup()
{
  createCanvas(400, 400)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, -5)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
  {
    this.acc.add(force)
  }
}
```

```

move()
{
  this.vel.add(this.acc)
  this.pos.add(this.vel)
  this.acc.mult(0)
}
}

```

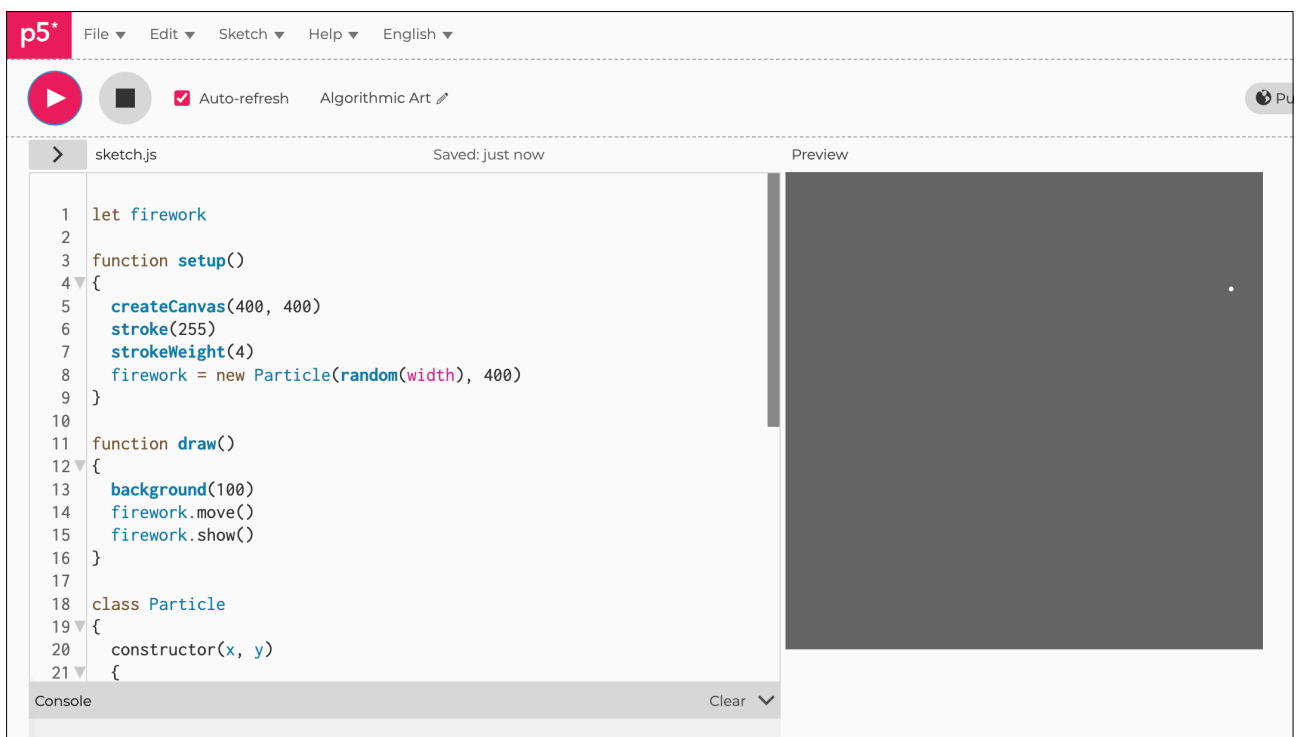
Notes

What you should see is the particle start at the bottom and move upwards and off the canvas.

Challenge

Alter the speed of ascent.

Figure E4.9





Sketch E4.10 gravity

Now let us add **gravity**, which is a force, and it points down (**0.1**).

```
let firework
let gravity

function setup()
{
  createCanvas(400, 400)
  gravity = createVector(0, 0.1)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  firework.applyForce(gravity)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, -5)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
```

```
{
  this.acc.add(force)
}

move()
{
  this.vel.add(this.acc)
  this.pos.add(this.vel)
  this.acc.mult(0)
}
}
```

Notes

You notice it doesn't go very high, so let's increase the vertical velocity to **8**.



Sketch E4.11 higher

Making it go a little higher.

```
let firework
let gravity

function setup()
{
  createCanvas(400, 400)
  gravity = createVector(0, 0.1)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  firework.applyForce(gravity)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, -8)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
```

```
{
  this.acc.add(force)
}

move()
{
  this.vel.add(this.acc)
  this.pos.add(this.vel)
  this.acc.mult(0)
}
}
```

Notes

You should see a slight change in height of the particle.



Sketch E4.12 lots of fireworks

At the moment, we have one firework shooting up, but we want lots of them (and then explode later on), so we create an array of fireworks.

```
let fireworks = []

let gravity

function setup()
{
  createCanvas(400, 400)
  gravity = createVector(0, 0.1)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  firework.applyForce(gravity)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, -8)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }
}
```

```
    applyForce(force)
  {
    this.acc.add(force)
  }

  move()
  {
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.mult(0)
  }
}
```

Notes

We still have the one particle going and returning to the ground.



Sketch E4.13 explode

We create a Firework class that is going to explode.

```
let fireworks = []
let gravity

function setup()
{
  createCanvas(400, 400)
  gravity = createVector(0, 0.1)
  stroke(255)
  strokeWeight(4)
  firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  firework.applyForce(gravity)
  firework.move()
  firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, -8)
    this.acc = createVector(0, 0)
  }

  show()
  {
    point(this.pos.x, this.pos.y)
  }

  applyForce(force)
```

```
{
  this.acc.add(force)
}

move()
{
  this.vel.add(this.acc)
  this.pos.add(this.vel)
  this.acc.mult(0)
}
}
```

```
class Firework
{
  constructor()
  {
    this.firework = new Particle(random(width), height)
  }

  show()
  {
    this.firework.show()
  }

  move()
  {
    this.firework.applyForce(gravity)
    this.firework.move()
  }
}
```

Notes

Yet another class.



Sketch E4.14 remove redundant code

We need to reference these in the main sketch. No changes will be seen just yet.

! Remove the redundant lines of code (commented `//`).

```
let fireworks = []
let gravity

function setup()
{
  createCanvas(400, 400)
  gravity = createVector(0, 0.1)
  fireworks.push(new Firework())
  stroke(255)
  strokeWeight(4)
  // firework = new Particle(random(width), 400)
}

function draw()
{
  background(100)
  for (let i = 0; i < fireworks.length; i++)
  {
    fireworks[i].move()
    fireworks[i].show()
  }
  // firework.applyForce(gravity)
  // firework.move()
  // firework.show()
}

class Particle
{
  constructor(x, y)
  {
    this.pos = createVector(x, y)
    this.vel = createVector(0, -8)
    this.acc = createVector(0, 0)
  }
}
```

```

}

show()
{
    point(this.pos.x, this.pos.y)
}

applyForce(force)
{
    this.acc.add(force)
}

move()
{
    this.vel.add(this.acc)
    this.pos.add(this.vel)
    this.acc.mult(0)
}
}

class Firework
{
    constructor()
    {
        this.firework = new Particle(random(width), height)
    }

    show()
    {
        this.firework.show()
    }

    move()
    {
        this.firework.applyForce(gravity)
        this.firework.move()
    }
}
}

```

Notes

Nothing new to see just yet. This is just getting the ducks in a row.



This continues in the next workbook

Because the unit is so long I have stretched it over two workbooks otherwise we will have a workbook much larger than other ones and a lot more to print or read.