Week 3 - Iteration, arrays, objects and pixel arrays

Outcomes

- Iteration using while and for loops
- Understand and using arrays
- Using loops and arrays together
- Understanding and using JavaScript objects
- Understanding how colour data is stored in pixel arrays
- Accessing the webcam

Supporting code

The code for this workshop is hosted on **Github**, which is a web-based repository for hosting and versioning code.

Download the code and unzip it on your desktop.

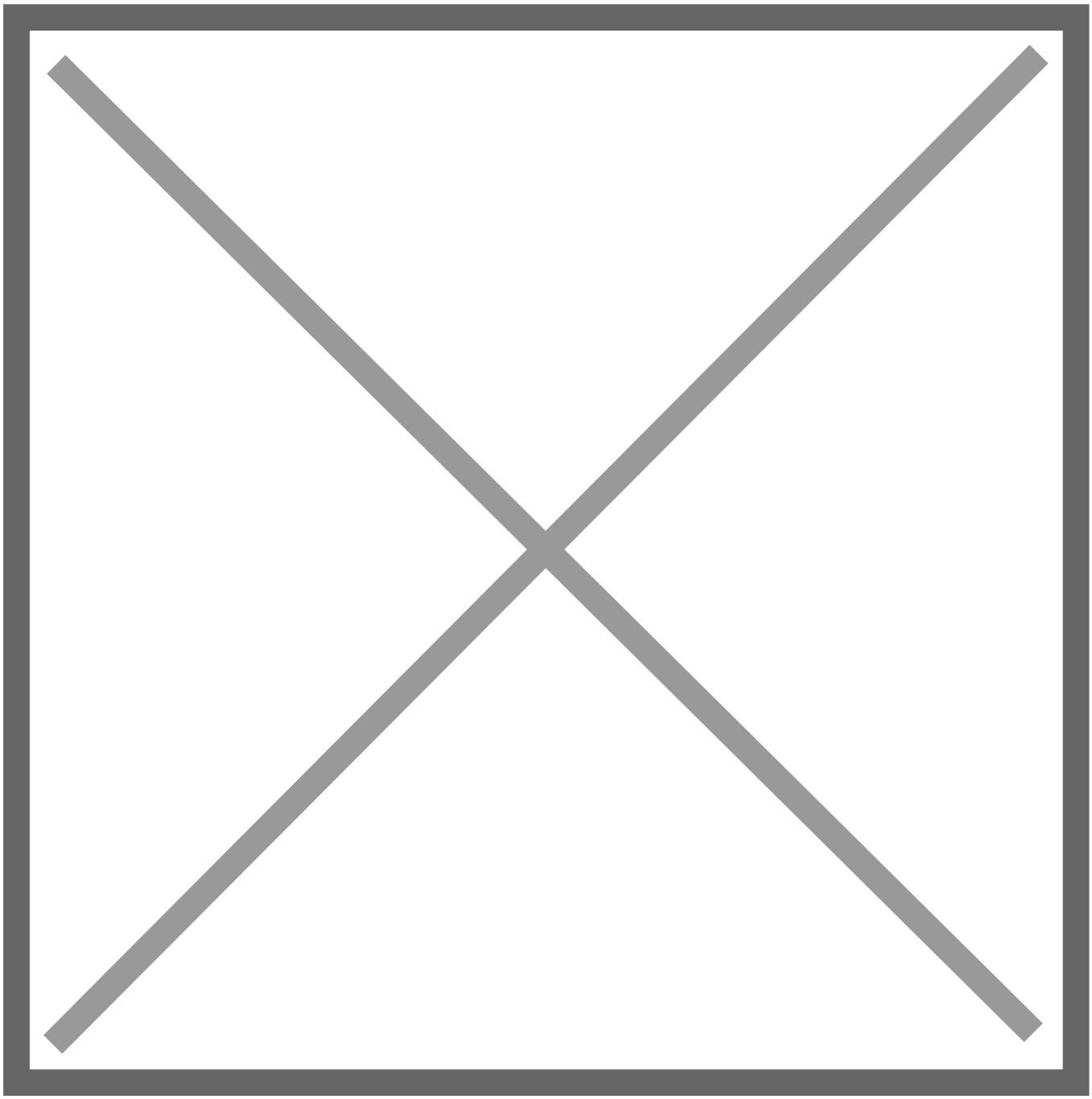
> The code is also available to **view directly on Github's website**.

During this workshop session we will be using the following project directories:

```
11_iteration_and_loops/
12_iteration_02/
13_loops_and_arrays/
14_pixel_array/
15_image_pixel_array/
16_webcam_capture/
```

Local web server

So far during this series of workshops testing your code has involved opening the `index.html` file in your browser, which results in an absolute *file path* in the browser address bar (see below). You can see this indicated by the `file://` protocol followed by the absolute file path to the index.html file:



For some examples you will need to run a local HTTP web server that serves the files in a project. If you have **Node.js** already installed you can run the following command to install an HTTP web server:

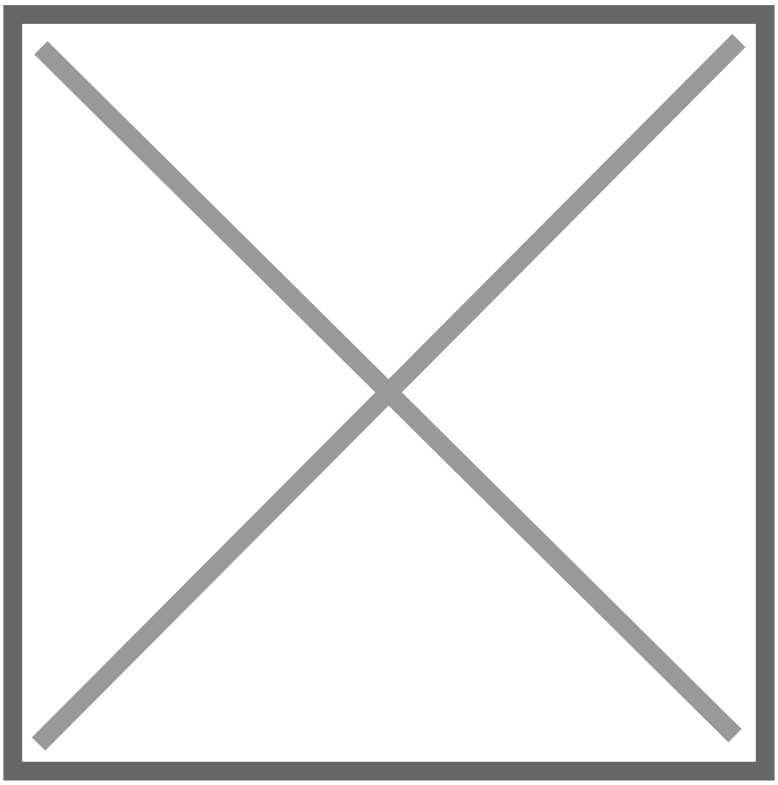
```
sudo npm install -g http-server
```

If you receive an error from the above command it's likely that you *do not* have Node.js installed. In which case visit the **Node.js homepage** and download/install the **LTS** version and repeat the command above.

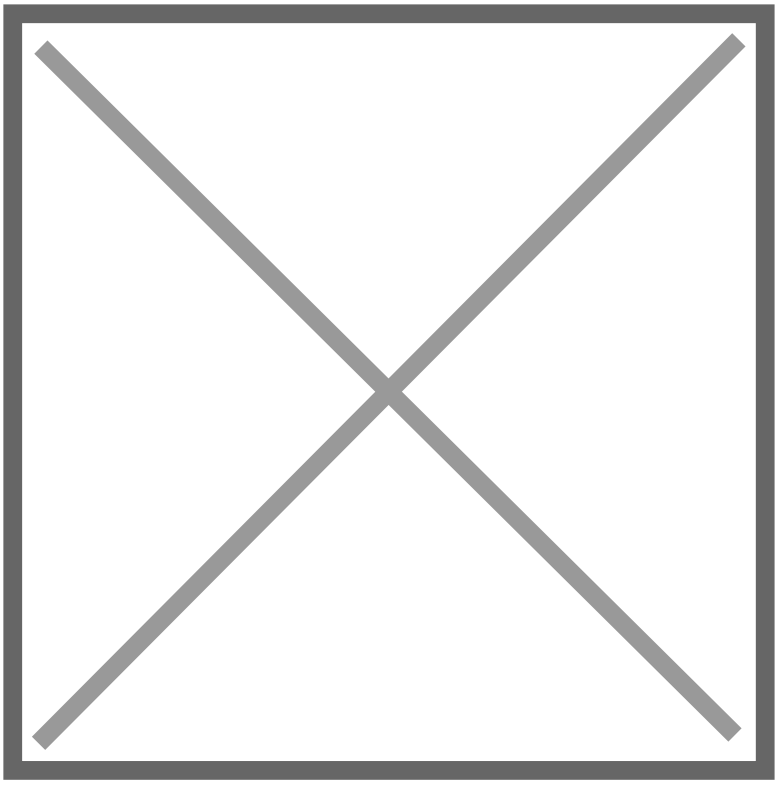
Once you have installed the HTTP web server you will need to *change directory* (`cd`) into the project directory on the command line and run the server:

```
cd ~/Desktop/intro-to-programming-2017/15_image_pixel_array/
http-server
```

If successful you will see messages in the command line similar to this:



You can then copy and paste one of the URLs into you browser:



Iteration: while and for loops

> **Supporting Code**
> The code to support this section is located in the following directory and is available to view on **Github**:
>
> ```
> /11_iteration_and_loops/
> /12_iteration_02/
> ```

Sometimes it is necessary to repeat a task over and over on the same data in order to achieve a desired outcome. This is known as an *iterative process* and each step is an *iteration*.

The most common application for iteration is to create, check, or modify a collection of variables.

In the previous workshop, we were introduced to the idea of conditionals. We saw that an `if` statement can be used to branch code, but this is only performed once.

If we want to perform a conditional operation repeatedly, we need to use a different statement – the `while` loop.

The example below will draw six circles onto the canvas. Note that the circles are identical, apart from the `x` coordinate.

```
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(128);
  ellipse(50, 225, 20, 20);
  ellipse(100, 225, 20, 20);
  ellipse(150, 225, 20, 20);
  ellipse(200, 225, 20, 20);
  ellipse(250, 225, 20, 20);
  ellipse(300, 225, 20, 20);
  ellipse(350, 225, 20, 20);
}
```

We can simplify this code by using a `while` loop.

```
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(128);
  var x = 50;
  while (x <= 350) {
    ellipse(x, 225, 20, 20);
    x = x + 50;
  }
}
```

What's happening in the above example line-by-line:

1. `var x = 50;`
   Here we create a temporary variable to help us iterate. In this case an integer, initially set to 50.
2. `while (x <= 350) {`
   This starts the while loop. As long as the condition inside the parentheses remains **true**, the code that follows the curly brace will be repeatedly executed (forever!)
3. `ellipse(x, 225, 20, 20);`
   We draw a circle. The y-position, height, and width are identical for each; the x-positon is set using the current value of our temporary variable.
4. `x = x + 50;`
   The value of the temporary variable is increased by 50.

As soon as the condition inside the parentheses returns **false**, the `while` loop exits and code execution continues.

Although this is a very common code pattern, it's unusual to see `while` loops actually used in code. This is because most programming languages provide us with a more useful variant – the `for` loop.

A `for` loop is written slightly differently from a `while` loop. The parentheses contain three statements separated by semicolons, rather than a simple test.

```
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(128);
  for (var x = 50; x <= 350; x = x + 50) {
```

```
    ellipse(x, 225, 20, 20);
  }
}
```

What's happening inside the parentheses:

1. `for (` **`var x = 50`** `; x <= 350; x = x + 50) {`
   A temporary variable is declared and initialised in the first statement.
2. `for (var x = 50;` **`x <= 350`** `; x = x + 50) {`
   The second statement contains the condition that is checked. If this returns **false** the loop exits.
3. `for (var x = 50; x <= 350;` **`x = x + 50`** `) {`
   The final statement contains code that is to be executed after each successful loop.

Even in these basic examples, it's clear to see that loops help us avoid repetition and reduce the number of lines of code we write.

Exercise

- Add the `12_iteration_02/` directory to Atom
- Open `index.html` in a browser
- Change the RGBA values of the pixels inside the nested for loop
- Try using the `random()` function to set the colour values
- Try using the `x` and `y` variables to set the colour values

JavaScript Arrays

Arrays are essentially ordered lists of things and each item in that list can be accessed individually. The array itself is a type of variable and it stores other variables inside. The stored variables can be used in the same way as you use any other variable.

Here is a simple array:

```
var sizes = [ 20, 350, 80, 210 ];
```

What's important about an array is the *order* of the items within. To access any individual item of data stored inside the variable, we need to reference the item's position, commonly referred to as the array **index**. Crucially, the index of an array starts at zero:

```
console.log(sizes[0]);  // logs: 20
```

And therefore the index of the last item in the array would be one less than total number of items. In our example above we have 4 items, so the final item is accessed using the index 3:

```
console.log(sizes[3]);  // logs: 210
```

JavaScript arrays are particularly useful since you can store any type of data inside, including integers, strings, objects and—perhaps confusingly—other arrays. Here is an example of an array containing a list of strings:

```
var technicians = [ "Delia", "Will", "Adam", "Gareth", "Tom" ];
```

And, as above, we can access the strings using the array variable `technicians` and counting along the list starting from zero:

```
console.log( technicians[0] ); // "Delia"
console.log( technicians[2] ); // "Adam"
console.log( technicians[4] ); // "Tom"
```

Try this for yourself using this CodePen.

Loops and arrays

> **Supporting Code**
> The code to support this section is located in the following directory and is available to view on **Github**:
>
> ```
> /13_loops_and_arrays/
> ```

When we have only a few items in our arrays, it is not a lot of additional code to access each of them explicitly using their index:

```
ellipse(x, y, sizes[0]);
ellipse(x, y, sizes[1]);
ellipse(x, y, sizes[2]);
ellipse(x, y, sizes[3]);
```

But even this is repeating code unnecessarily. And when we start to hold hundreds or thousands of items in our array, it would become unmanageable to write out the code above.

To unleash the full potential of arrays, they can be combined with looping structures such as `for` loops. As we have seen already, the `for` loop can be used to run a piece of code a number of times, incrementing an index variable on each execution:

```
for( var i = 0; i < 4; i++ ){
  console.log( i );
}
```

A further useful feature of arrays is that they have an internal property that contains the current length of the array:

```
var sizes = [ 20, 350, 80, 210 ];
console.log(sizes.length);        // logs 4
```

The `length` property can be used within our `for` loop to determine how many times the loop runs the code before stopping. In the case of our `sizes` array above, the loop would continue to execute as long as the `i` variable is less than (`<`) the number of items in the array.

This is a very common design pattern.

```
for( var i = 0; i < sizes.length; i++ ){
  console.log( i );
}
```

What is happening here?

1. The variable `i` is set as 0
2. The statement `i < sizes.length` is tested
3. If the condition is true the code inside runs
4. ...and the variable `i` is increased by 1
5. Go back to point 2 and repeat until false

The code would run 4 times and log 0, 1, 2 and 3.

Now we have a loop that runs as many times as there are items in the array. Crucially, inside that loop, the variable `i` is incremented by 1. Each time it increments by 1 we can use it to access the value in the array at that index:

```
for( var i = 0; i < sizes.length; i++ ){
  console.log( sizes[i] );
}
```

If we recall that the first item in an array uses the index zero, we can see why our `i` variable is initialised as 0 rather than 1.

Within our `for` loop we are now running code that accesses each of the items in the array in the correct order.

See this code executed in CodePen.

JavaScript Objects

In JavaScript most things you encounter are actually objects. The strings, arrays and even functions are objects at the most basic level. This is because they can all contain properties and functions *inside them*.

Here, for example, the variable `message` has a property called length that returns the length of the string:

```
var message1 = "what is an object";
console.log(message1.length); // 15
```

More examples on CodePen.

These are objects within internal properties and functions that are provided by the JavaScript engine inside the browser. We do not need write the code for these objects as it already exists.

However, creating your own objects is a very handy way to encapsulate related functions and variables, and also act as data containers. We can also use this technique to model things in a more helpful way.

Let take a look at the variables needed to draw a circle and then how we would move those variables inside an object. Here we define three variables:

```
var x = 50;
var y = 100;
var size = 20;
ellipse( x, y, size );
```

And here are the same three variables inside an object:

```
var circle = {
  x    : 50,
  y    : 100,
  size : 20
};
```

The first thing to note is that the object starts and ends with curly braces; the same way that we start and end functions and `if` statements.

Pay careful attention to the differences between declaring variables inside and outside of an object. Variables stored inside objects are called *properties* and each property has a *value*. The major difference in syntax is that properties and values are separated by a *colon* (`:`) instead of an equals sign (`=`).

```
x    : 50,  // Note the colon ':' separator...
```

And each of the property/value pairs are separated by a comma (`,`), *not* a semi-colon (`;`). The exception to this is rule is the last pair for which the comma is optional

```
x    : 50,  // ...and each pair separated by a comma
y    : 100,
size : 20   // except the last, which is optional
```

So now that the data that defines our circle is contained within an object how do we *access* that data? To access a property of an object the *dot syntax* is used. For example to access the `x` value:

```
circle.x
```

So to rewrite our code above using an object:

```
var circle = {
  x    : 50,
  y    : 100,
  size : 20
};

ellipse( circle.x, circle.y, circle.size );
```

See a simple example of this on CodePen.

> **Object Oriented Programming**
> A more advanced use of objects is to create templates of things that we want to represent in our code. These templates or models can be used to create different permutations of the same *type*. This is called **abstraction** and is one of the fundamentals of object-oriented programming (OOP). Mozilla Developer Network has a very good **section about objects** and a really interesting page **introducing OOP and how to implement it using JavaScript objects**.

Here is an example on CodePen of the above circle sketch created using a **constructor** function. This is a simple example of using Object Oriented Programming in JavaScript.

Pixel array

>

Supporting Code

The code to support this section is located in the following directory and is available to view on **Github**:

```
/14_pixel_array/
```

Previously we have discussed that our p5.js canvas is made up of individual pixels. Each of them can be located using an X coordinate between 0 and the width and a Y value between 0 and the height. Also known as Cartesian coordinates.

So how many pixels are there in a canvas of 600 pixels in width and 500 pixels in height:

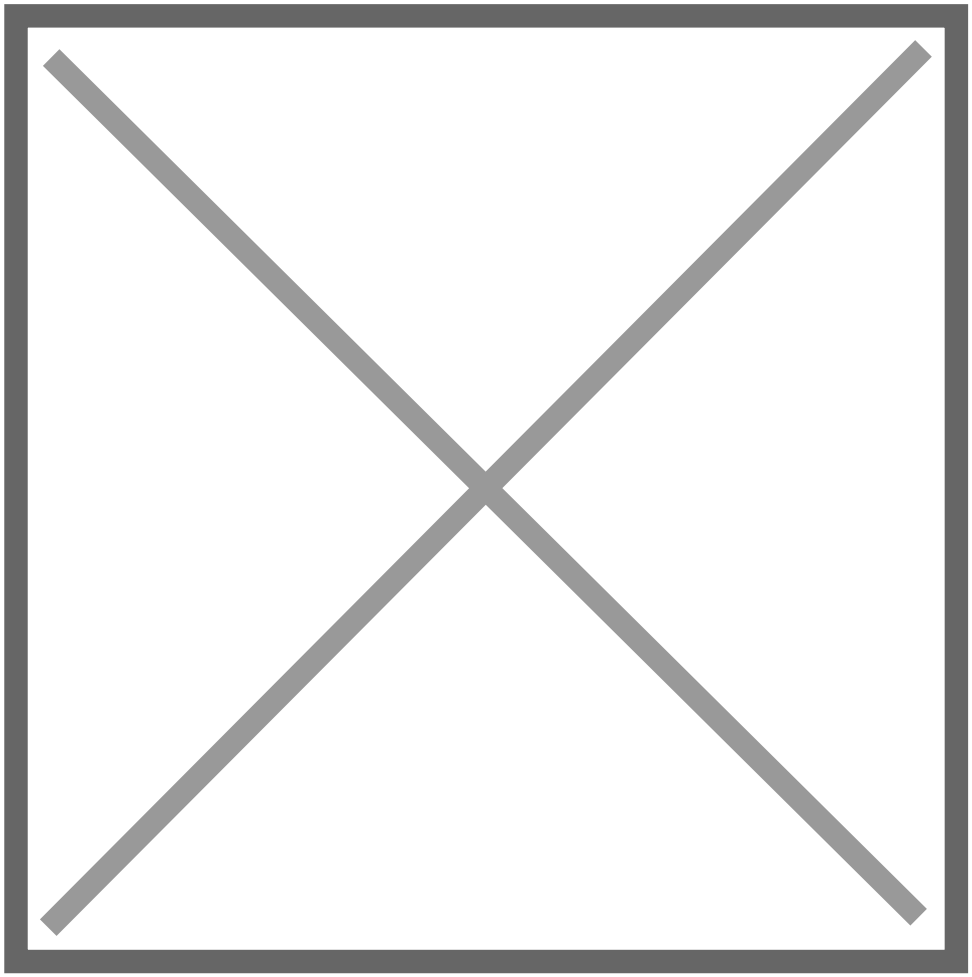
```
600 x 500 = 300000 pixels
```

We have also discussed that each pixel is made up of three values: red, green and blue. Well, there is actually a fourth value, which we haven't discussed in a great detail called **alpha**. This sets the transparency value of the pixel. So for every pixel on the p5.js/HTML canvas there are 4 pieces of information:

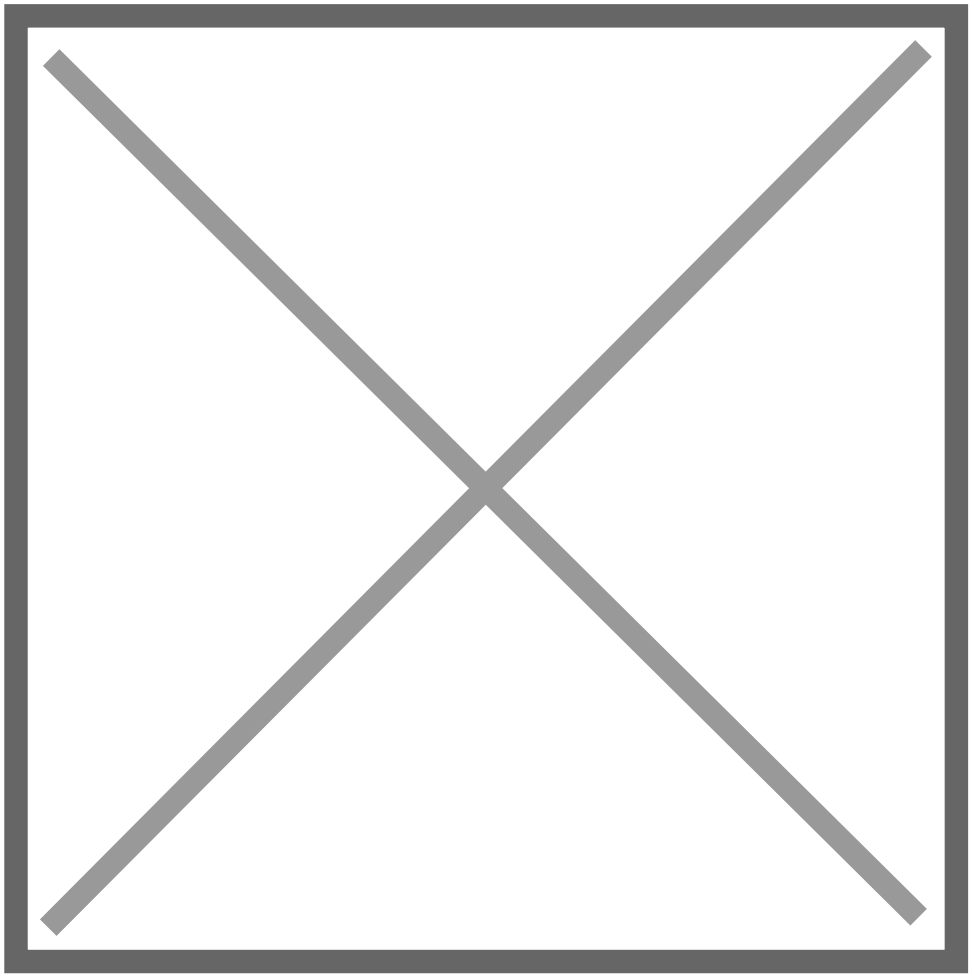
```
red, green, blue, alpha
```

So in total for our canvas of 600 x 500 we have this many pieces of information:

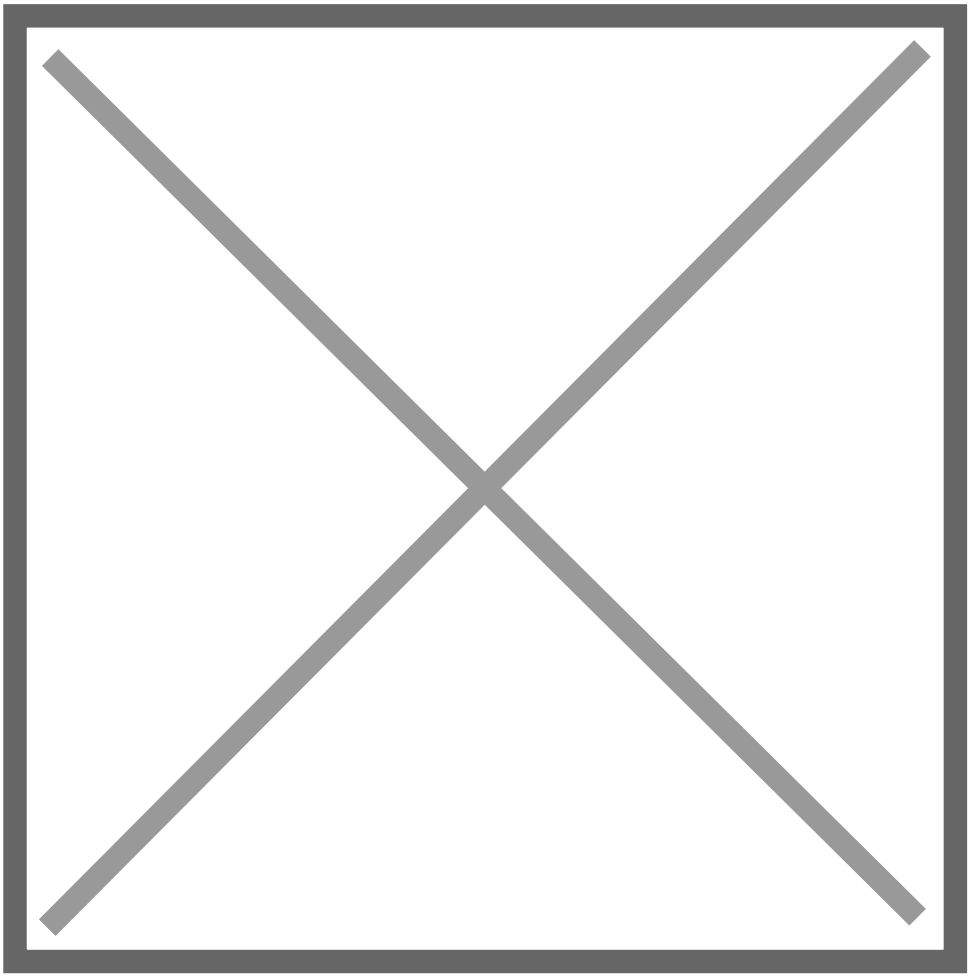
```
300000 (pixels) x 4 (colour value) = 1200000
```



All of this information is stored in one large linear array, which we can easily access and manipulate. However arrays are simply lists so they do not have a concept of which index relates to which X and Y coordinate on our screen.



If we want to access a particular pixel we do not refer to it as, for example, the 29th pixel (the last pixel in our example above). We are more likely to reference it using the X and Y coordinates. So how do we get from an X and Y coordinate to access and manipulate the 4 colour values within the pixel array?



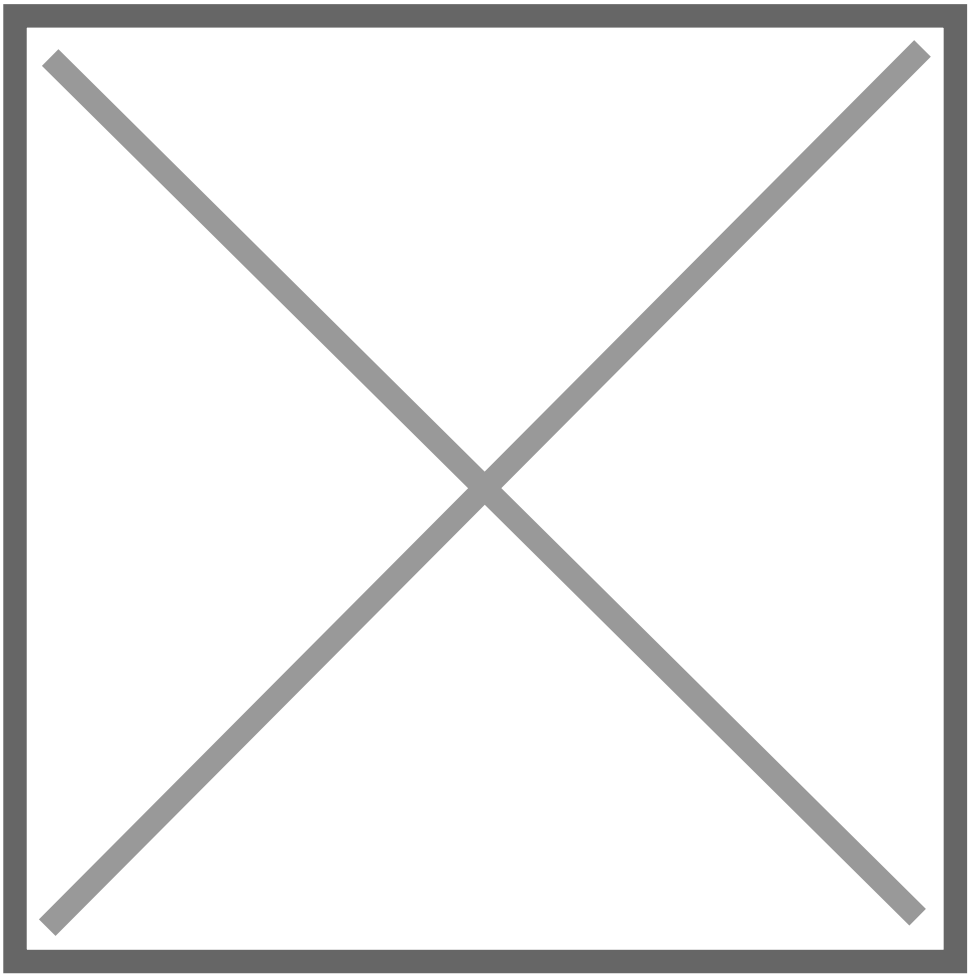
In the image above the red dot represents a pixel on screen that we want to target in the pixel array to access or change the 4 colour values.

If we were to count the grey boxes you can see that before we reach the red dot we have 2 full rows, which equates to `(y * width)`. Then we count in (or add) `x` positions. The formula to calculate this for any x and y value is therefore:

```
x + (y * width)
```

So far so good. However now we know that the number of the pixel in the canvas but for every pixel there are *4 values* in the array. Therefore to calculate the first of four positions in the array that contains the RGBA values for our pixel we simply multiple by 4. In our above example we have calculated the pixel position to be the 16th:

```
16 (pixel position) * 4 (colour values) = 64 (array index)
```



So now we know that the four positions in the array that represent our pixel are 64, 65, 66 and 67. We can therefore write the following code to manually set the colour of that pixel:

```
function draw() {
  loadPixels();



  pixels[64] = 255;    // red
  pixels[65] = 255;    // green
  pixels[66] = 255;    // blue
  pixels[67] = 255;    // alpha

  updatePixels();
}
```

But that is not very reusable code and we would have to manually calculate the index again every time we wanted to address a new pixel. What would be much better is to put all of those calculations into variables so we can simply change the X & Y value with ease:

> I've increased the size of the canvas to 60 pixels in width by 50 pixels in height so we have a slightly larger area to spot our pixel in.

```
function draw() {
  loadPixels();

  var x = 40;
  var y = 20;
  var index = ( x + (y * width) ) * 4;

  pixels[index] = 255;      // red
  pixels[index+1] = 255;    // green
  pixels[index+2] = 255;    // blue
  pixels[index+3] = 255;    // alpha

  updatePixels();
}
```

Using the above code we can address a particular pixel and then access the colours within the pixel array.

Try changing the X and Y values on this CodePen. You may need to look closely or zoom in to see the single coloured pixel.

So we now can access individual pixels based on their X & Y coordinates, what if we wanted to modify *all the pixels*. We can do this by using a nested `for` loop to iterate along every pixel on the X and Y axis. A nested `for` loop is one loop within another:

```
// Loop through every pixel on the X axis...
for ( var x = 0; x < width; x++ ) {
  // ...and for each X, loop through every pixels on the Y axis
  for ( var y = 0; y < height; y++ ) {
    // Every (x, y) coordinate is looped here:
    var index = (x + y * width) * 4;
    pixels[index] = 255;     // red
    pixels[index+1] = 0;     // green
    pixels[index+2] = 0;     // blue
    pixels[index+3] = 255;   // alpha
  }
}
```

In this example above every pixel is set to full red, no green, no blue and full transparency.

Exercise

- Add the `12_pixel_array` directory to Atom
- Open `index.html` in a browser
- Change the RGBA values of the pixels inside the nested for loop
- Try using the `random()` function to set the colour values
- Try using the `x` and `y` variables to set the colour values

Image pixel data

> **Supporting Code**
> The code to support this section is located in the following directory and is available to view on **Github**:
>
> /15_image_pixel_array/

> **Use a local web server**
> p5.js cannot access the image pixel data from an image that is loaded directly from the file system. Therefore you will need to install and run a HTTP server in order to complete the next exercise. To set up an local web server **follow these instructions**.

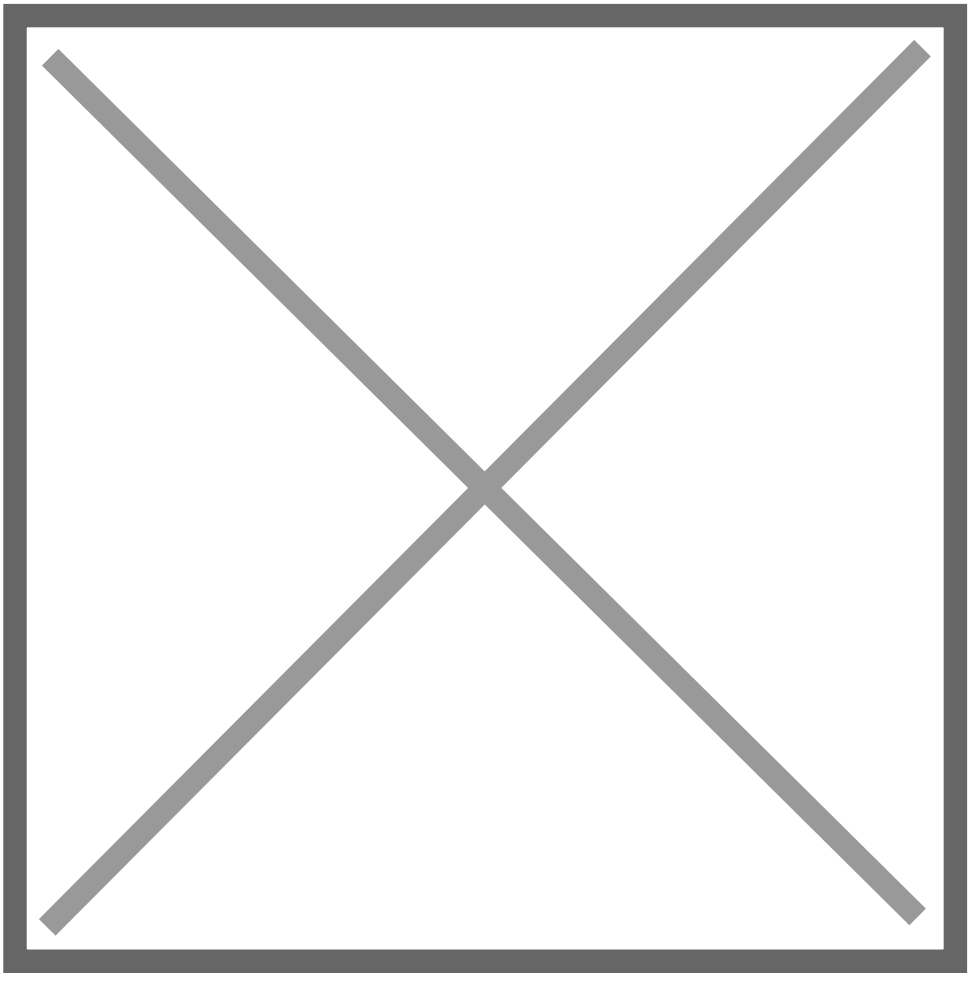
So far we have been manipulating the pixel colour values of an empty canvas; or more precisely a canvas full of a single colour. The exact same process is possible but instead of manipulating an empty canvas we can manipulate image data loaded in from an external file.

The data that represents an image is also made up of individual pixels (this is called a **raster graphic** and therefore within p5.js we access the image pixel data in the exact same way as we have already been accessing pixels in an array. Here is an example of this using a loaded image:

```
var img;

function preload() {
  img = loadImage("images/maxernst.jpg");
}

function mouseDragged(){
```

```
  var index = (mouseX + mouseY * width)*4;

  img.loadPixels();
  var r = img.pixels[index];
  var g = img.pixels[index+1];
  var b = img.pixels[index+2];
  var a = img.pixels[index+3];

  fill(r, g, b, a);
  ellipse(mouseX, mouseY, 40, 40);
}
```

You will notice a new function being used called `preload()`. This is a handy function provided by p5.js that ensures that images or external data such as API data are finished loading before calling the `setup()` and `draw()` functions:



Inside the preload function we give a relative path as an argument to the `loadImage()` function. The results of this are stored in a global variable `img`.

```
var img;


function preload() {
  img = loadImage("images/maxernst.jpg");
}
```

Then later in our `draw()` function we can access the pixel array as a property of the `img` object. We do this using the *dot syntax*. Here I am setting the first pixels colour to green:

```
img.pixel[0] = 0;
img.pixel[1] = 255;
img.pixel[2] = 0;
```

Exercise

- Add the `13_pixel_array` directory to Atom
- Set up a local web server and run it within the `13_pixel_array` directory.
- Open the URL provided by the local web server in a browser.
- Click and drag the mouse around the canvas to see the pixel colours being rendered in circles
- Uncomment the lines in the nested `for` loop and play with the rgba values:

```
img.pixels[index] = r;
img.pixels[index+1] = g;
img.pixels[index+2] = b;
img.pixels[index+3] = a;
```

Webcam capturing

The code to support this section is located in the following directory and is available to view on **Github**:

```
/16_webcam_capture/
```

> **Using a local web server**
> This is another instance when you won't be able to run this sketch directly from your filesystem, you will need a local web server running. To set up an local web server **follow these instructions**.

Using p5.js accessing the webcam is quite straightforward. It takes just a few lines of code:

```
var capture;

function setup() {
  createCanvas(400, 300);
  pixelDensity(1);

  // Create video capture object.
  capture = createCapture(VIDEO);
  capture.size(width, height);
}

function draw() {
  clear();
  // Draw capture to the canvas.
  image(capture, 0, 0, width, height);
}
```

Behind the scenes the `createCapture()` function does a few clever things. Firstly it causes the browser to ask the user if they want their camera to be opened and used. This is a security provision to ensure nefarious programmers cannot access webcams without permission. Secondly it creates a HTML Video element in the browser and places it next to our p5.js canvas. We can then use the image data from inside that HTML Video object to draw into our canvas.

See this in action on CodePen.

However this leaves us with two copies of the webcam video. That is why we call `capture.hide()` in all the following examples.

What is extremely useful about this object stored in the `capture` variable is that the pixels inside it can be treated exactly the same as the pixel array and image pixel array examples.

```
clear();
capture.loadPixels();
for ( var x = 0; x < width; x++ ) {
  for ( var y = 0; y < height; y++ ) {
    // Get the pixel at x and y position
    var index = (x + y * width) * 4;
    capture.pixels[index] = 255;     // red
    // capture.pixels[index+1] = 0;   // green
    // capture.pixels[index+2] = 0;   // blue
    // capture.pixels[index+3] = 0;   // alpha
```

```
  }
}
capture.updatePixels();
image(capture, 0, 0, width, height);
```

In this example above every pixels has had it's red value cranked up to maximum giving the captured image a distinctly red tint.

Within the exercise code you will also find a call to the `saveCanvas()` function being used within the `keyPressed()` function:

```
function keyPressed(){
  if (keyCode == RETURN) {
    saveCanvas("webcam", "jpg");
  }
}
```

This 4 lines of code allows the webcam image to be saved and downloaded as a JPG when the return key is pressed.

Exercise

- Add the `16_webcam_capture` directory to Atom
- Set up a local web server and run it within the `16_webcam_capture` directory.
- Open the URL provided by the local web server in a browser.
- Hit the enter key to download a frame of the webcam video
- Uncomment the lines in the nested `for` loop and play with the rgba values:

```
capture.pixels[index] = 255;     // red
capture.pixels[index+1] = 255;   // green
capture.pixels[index+2] = 255;   // blue
capture.pixels[index+3] = 255;   // alpha
```

Revision #48
Created 4 November 2016 16:12:55
Updated 20 August 2018 11:13:02