A Level Computer Science NEA Project

Dr Challoner's Grammar School

Candidate Number : 8358

Hannah Lewis

Index

Analysis

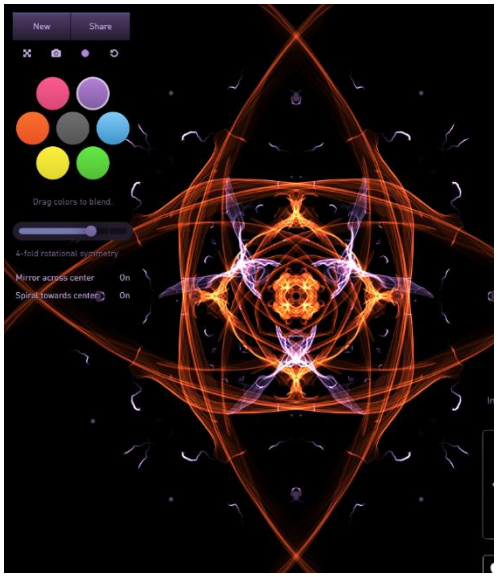
Introduction

When thinking about what I wanted to achieve with this programming project I considered the three A Level subjects that I am currently studying; Computer Science, Maths and Art, and how best I could combine the skills and elements from each into a project that I would find interesting and challenging to work on. This led me to the area of Computer-Generated Art, where a piece of art is created solely by or with aid from a computer program. This category includes Graphic Software, Robot Painting and Algorithmic Art in which the design or nature of the art being produced is defined by a complex algorithm handled by a computer.

Research

I knew that in order to create a program that had the required levels of complexity for an A Level NEA I would have to use complex algorithms and mathematics to form the generated image and therefore began to research existing art generator programs to investigate how similar programs were designed and implemented. In addition, I wanted to research the actual artists who create or utilise such programs, this would provide me with inspiration for the style and nature of the image I wanted to receive as an output.

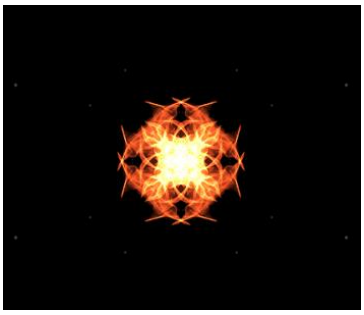
The interactive generative art program named 'Silk'[1] was my first source of investigation into algorithmic art programs. This web program, created by Yuri Vishnevsky, allows the user to create a piece of art using a randomised algorithm which generates a series of wisp-like lines, emitting from the position of the mouse as the user drags it across the screen. The addition of options to change the canvas, such as mirror reflections, and the colour of the lines, was a good way to enhance the program and allow for the user to tailor it to their preferences. This user interaction was something that I was keen to include within my own project, therefore ensuring that my program would not simply apply a filter to an image and produce an output, without any influence from their user and their desires. The Silk program was expanded later so that the user could affect the movement of the art as the mouse is dragged, controlling the speed and direction of wind which changes how the art will flow.



This screenshot of the Silk program shows the nature of the art created. As the user drags the mouse across the screen a series of 'wisp-like' lines appear, with the brightest following the exact path of the mouse and the additional lines surrounding it with decreasing opacity.

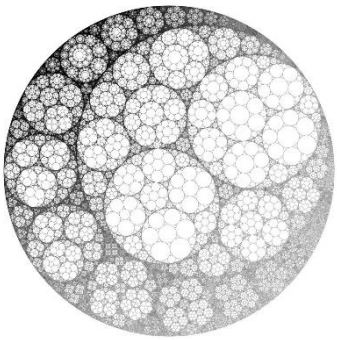
This screenshot also shows the user interface, which is clean and simple to use. Yet the Menu shows the extensive amount of options which can selected by the user, including being able to drag the colours into each other to blend them.

This program also allows the user to share their creation on multiple platforms, including Facebook, Twitter and Pinterest.

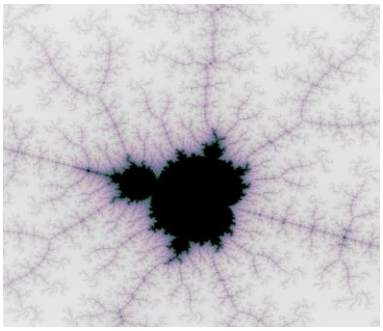


Unfortunately, I could not find any information about the exact algorithm used to create this piece of software however by holding in one place in the canvas, without moving the mouse, the image on the left was created. Forming from the initial place on the canvas and then expanding outwards, creating random lines and shapes as it moved. This implied that the starting point of the art was set as the mouse click, yet a random algorithm was used to determine the nature and direction of the lines that would be generated around the origin. After holding the mouse button down for some time I realised that the lines generated must have a maximum length to ensure that the art generated around a point is limited to a certain radius, ensuring the program produces art that follows the user's path quite closely. This random generation would be a method I would have to use when forming my own algorithmic art program if I wanted the art to be unique each time the program was run.

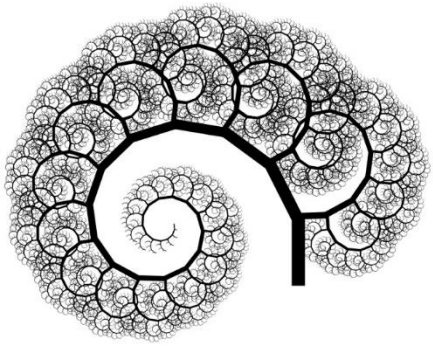
I continued to research examples of generative art in order to decide what kind of images my program would produce. I was really interested in the iterative nature of fractals[2], such as the Mandelbrot algorithm, which are set of complex mathematics that can be used to produce visually stunning and hypnotising pieces of 'recursive drawing'.



https://www.pinterest.co.uk



https://www.contextfreeart.org/gallery2/#design/4028

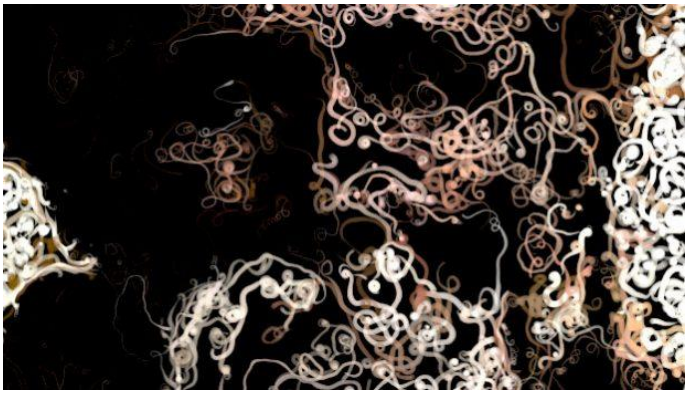


https://courses.ideate.cmu.edu/15-104/f2016/week-14-lab/

Whilst I thought that the repetitive nature of the fractals made them a suitable base for my algorithmic art, I was concerned that a fractal generator would not allow for much user interaction given that the fascinating nature of a fractal is determined by its pre-defined mathematical structure and it would be difficult to allow the user to make alterations to the piece without ruining the fractal effect.

Wanting to keep the mathematically defined patterns but stray away from the more restrictive fixed nature of fractals, I studied 'Organic Art' a type of computer generated art based on the evolution of growing plants, cells and other organic materials which attempt to simulate growth through the behaviour of an expanding pattern. This was a suitable area of computer-generated art for me to follow as it allowed for a solid, algorithm defined structure whilst also having elements of randomness and changeability that would incorporate user interaction. To further my understanding of this branch of computing I researched the evolutionary system 'Photogrowth'[3] designed by Penousal Machado, Tiago Martins, Hugo Amaro and Luís Pereira from 'CDV Lab'. This program creates non-realistic representations of input images based on an ant-colony, with each ant gaining energy, depositing ink as it moves along the lines of the image, producing offspring

and then 'dying' once out of energy. A range of sensory vectors are used to define each ant's energy level, dependent of the direction, length and weight of the area surrounding the ant, for example a brighter area of the image will offer more energy for an ant to 'collect' than a darker part of the source image.



https://cdv.dei.uc.pt/photogrowth-ant-painting/

I found the style of the 'ant-trails' very interesting and wanted to create something that had a similar feel to it, with a line of objects moving in different directions to create spirals and curves. Ultimately I would like to produce a program similar to 'Photogrowth', which takes in an input image and uses an algorithm to trace the outline of the image and then initiates another computer algorithm which produces an evolving form of shapes that follow the outline to produce an abstract representation of the image.

Through reading the 2012 Conference Paper for Photogrowth'[4] I studied the algorithms and mathematics used to create this evolutionary engine.

The amount of energy available to each ant at any point on the canvas is represented by the luminance of that area, therefore a lighter area will provide more energy for an ant as they pass through it. The amount of energy consumed by the ant is then removed from the rest of the environment, ensuring that there is a stable amount of total energy distributed between the ants and canvas.

An ant's energy was calculated using the following :

energy = (energy + b(x,y) * gain) * decay

b(x,y) returns the luminance of an area, and gain and decay are scalar values representing rates of energy gain and decay. By having these scalar quantities, it ensures that the ant's energy is constantly fluctuating up and down, meaning that each ant's trail will never carry on for too long. This produces the small lines and spirals that frequently change colour and opacity as they overlap with another ant's trail.

In order to assess their surroundings and the amount of energy within it, each ant was assigned 10 sensory vectors that would allow them to 'look' around them. Each vector had a given direction and length and would return the luminance of the area where the vector ended on the canvas. The luminance of the area surrounding the ant would then affect its movement and so the sum of all 10 sensory vectors would return the amount of energy the ant could gain.

1. Initialization: n ants are placed on the canvas on pre-established positions; Each ants assumes the color of the area where it was placed; Their energy and deposit transparencies are initialized using the species parameters;

2. For each ant:

   (a) Update the ant's energy following formula 1;

   (b) Update the energy of the environment;

   (c) Place ink on the painting canvas;

   (d) If the ant's energy is bellow the death threshold remove the ant from the colony;

   (e) If the ant's energy is above the reproduction threshold generate an offspring; The offspring assumes the color of the position where it was created and a percentage of the energy of the progenitor (which loses this energy); The offspring inherits the velocity of the parent, but a perturbation is added to the angular velocity by randomly choosing an angle between $descvel_{min}$ and $descvel_{max}$ (both values are species' parameters); Likewise, the deposit transparency is inherited from the progenitor but a perturbation is included by adding a randomly choosen a value between $dtransp_{min}$ and $dtransp_{max}$;

   (f) Update ant's position following formulas 2 and 3;

3. Repeat since 2 until no living ants exist;

Steps (b) and (c) require further explanation. The consumption of energy of the environment is attained by drawing on the living canvas a black circle of size equal to $energy * cons_{rate}$ of given transparency. $cons_{rate}$ and $cons_{trans}$ are parameters of the species. In previewing mode ink is deposited on the paining canvas by drawing a circle of the color of the ant – which is attributed when the ant is born – with a size given by $energy * deposit_{rate}$ and of given transparency. $deposit_{rate}$ is a species parameters, the deposit transparency is a parameter of the ant.

These screenshots of the Conference Paper show the order of processes through the program.

A set number of ants are placed on the canvas, with each being assigned a starting colour, energy level and deposit transparency (how opaque the ink they deposit will be) based on a set of parameters.

As the ants move, their energy and the environments energy is updated, meaning that at any one second, the energy levels will change multiple times given how many ants will be gaining/losing energy.

Each ant deposits ink as they move, with it's opacity being dependent of its current energy level.

Each ant's energy level is constantly checked to see if :

- It is below the death threshold and therefore means the ant must 'die' and be removed from the environment
- If it is above the reproduction threshold and therefore must create offspring. In which case new ants are formed, with the same velocity as the parents yet with a random angular velocity (ensuring it does not immediately travel in the same direction as its parent)

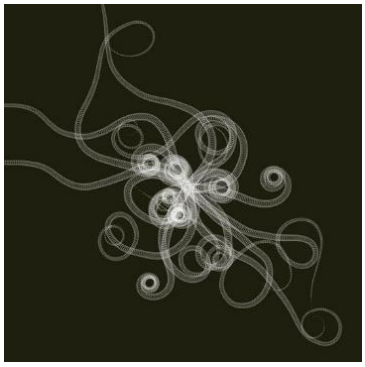
After analysing 'Photogrowth' and the algorithms used to create it, I decided to focus on a program that would use similar ideas of tracing an outline and then generating an abstract representation of the image using that outline. From there I could use this outline to replicate similar ideas of evolutionary art where shapes, such as a series of overlapping circles, originated from the outline and expanded outwards, forming random sequences of shapes coming from the image's edges. The user would be given a range of options to be able to control the way in which the art is generated, ensuring that there is a sufficient level of user interaction during the creation of the final output.

In order to achieve my proposed program, my first task would be to create an edge detection program that would take in an image and perform an abstraction upon it to produce an image that only represents the lines needed to form an outline of the original shape. There are multiple stages of processing that an image must go through before the actual edge detection program

can be run on the file. This includes grey scaling the image to remove the RGB values, focusing only on the contrasts between dark and light in the image. A Gaussian Blur is often used to reduce noise from the image, whilst this may reduce the intensity of edges in the image, this reduction is minimal in comparison to the positive effect filtering has on the effectiveness of the edge detection program. However, as my program intended to create a more abstract representation of the image, the edge detection did not need to produce an exact outline of the image and having a less accurate outline could lead to a more creative and free-flowing outcome. There I would not necessarily have to apply a Gaussian Blur to the images imported.

Through my research I discovered the Sobel Operation, a technique used to calculate the strength of an edge as a product of the horizontal and vertical edges. This method uses a weighted 3x3 grid which loops through each pixel and produces a sum of all the weighted values of the 8 pixels around the central one. This weighting ensures that the pixels closest to the pixel being evaluated are given the highest priority, this will result in a more accurate representation of the edge surrounding that pixel.

In order to collect a range of influences that would help me decide what style the art generation would be, I researched more examples of Organic Algorithmic Art, discovering a website called NodeBox[6] where programmers could upload and share the computer-generated art they had produced. I liked the effect spiralled tunnels created in Tom De Smedt piece 'Tendrils'[7] and decided that I could create a similar effect by layering circle outlines on top of each other, moving each new circle slightly from the previous so that it created a curving line of circles.



https://www.nodebox.net/code/index.php/Tendrils

Once the edge detection program had been completed, I would then begin to experiment with replacing the pixels of the outlined image with other shapes, which evolved from the outline. This could lead to vanishing spirals, like in Tendrils, coming from the outline, or shapes being placed on each pixel, with their colour and being defined by the colour and gradient of the edge. Before concluding exactly what the style of the final abstract representation would be, I would have to go through an extensive experimentation and testing process and therefore could not design the specifics of the art generation until I had a working edge detection program.

Despite having more experience in Python programming, I decided to complete this project using JavaScript as it is more suited to programs that incorporate drawing and visual simulations making it more suitable for writing algorithms that will generate a visual output. Another advantage of coding in JavaScript is that it can easily be integrated with HTML and CSS, the languages I would use to code the structure and design of the webpage that my program would be hosted in. Therefore, before starting the programming of my project I had to do several coding courses in JavaScript, HTML and CSS to learn the syntax of the languages and increase my understanding of programming for visual outputs, something I had little previous experience in.

End User

In order to ensure that my program met all the criteria required for it to be utilised in real situations and with all the necessary functionalities, I consulted a variety of peers who would be interested in utilising my program. This allowed me to collect a range of features that the user would want, which I needed to consider when building my program. This included how the user wanted the program to work and more importantly what the images produced by the program should look like. By talking with potential end users, with an interest in the artistic ability of the program, it ensured that my program would not only function successfully as a piece of code, but that it also met the appropriate level of artistic creativity that I was aiming for.

What key features would you look for in a generative art program?

"Colourful, dynamic and interactive."

"A wide range of colours"

"A wide range of patterns and shapes"

Would you prefer to see the output of the edge detection program or only see the result after the art generation has taken place?

"Start with the edges as an outline and then see the transformation into the art on screen"

"See it go from outlines to transformation"

What options would you like to have regarding editing/interacting with the artist representation of the image?

"Colour wheel options, granularity (level of detail/size of shapes generated)"

"Be able to change the abstract nature of it (multiple style options)"

"Be able to change the proportions (e.g. slider for how much it sticks to outlines)"

"Different transformation motions e.g. fade in, see generation bit by bit, animation of generation etc."

Objectives

Webpage Structure :

1. Creation of a webpage using HTML and CSS
2. Set up structure for integration of JavaScript graphics within webpage
3. User selects image file from their own directory to import into webpage
4. Creation of a canvas in HTML displayed within webpage
5. Original user-selected image shown on canvas
6. Creation of Control Panel to allow user to change setting of art generator (see User Options)
7. Canvas display updates as result of edge detection/art generation options returned
8. Aesthetically pleasing and accessible user interface

Edge Detection :

9. Image viewed pixel by pixel on canvas
10. Store RGB values of each pixel in array data structure
11. Greyscale pixels : sum RGB values of each pixel and divide by 3
12. Sobel Operation (in X direction and Y direction) using kernel convolutions
    a) Iterate over each pixel
    b) Apply weightings to pixel and the 8 others surrounding (in 3x3 grid)
    c) If a section of the 3x3 grid lies outside of the original image and therefore no greyscale pixel exists, assign a value of 0 to the section
    d) Horizontal weightings = 'x-kernel' array
    e) Vertical weightings = 'y-kernel' array
13. Calculate combination of X and Y edges using Pythagoras' Theorem on the sum of the horizontal weightings and the sum of the vertical weightings
14. Strength of edge converted to greyscale value and added to array
15. Greyscale values painted back onto canvas
16. Output image after edge detection

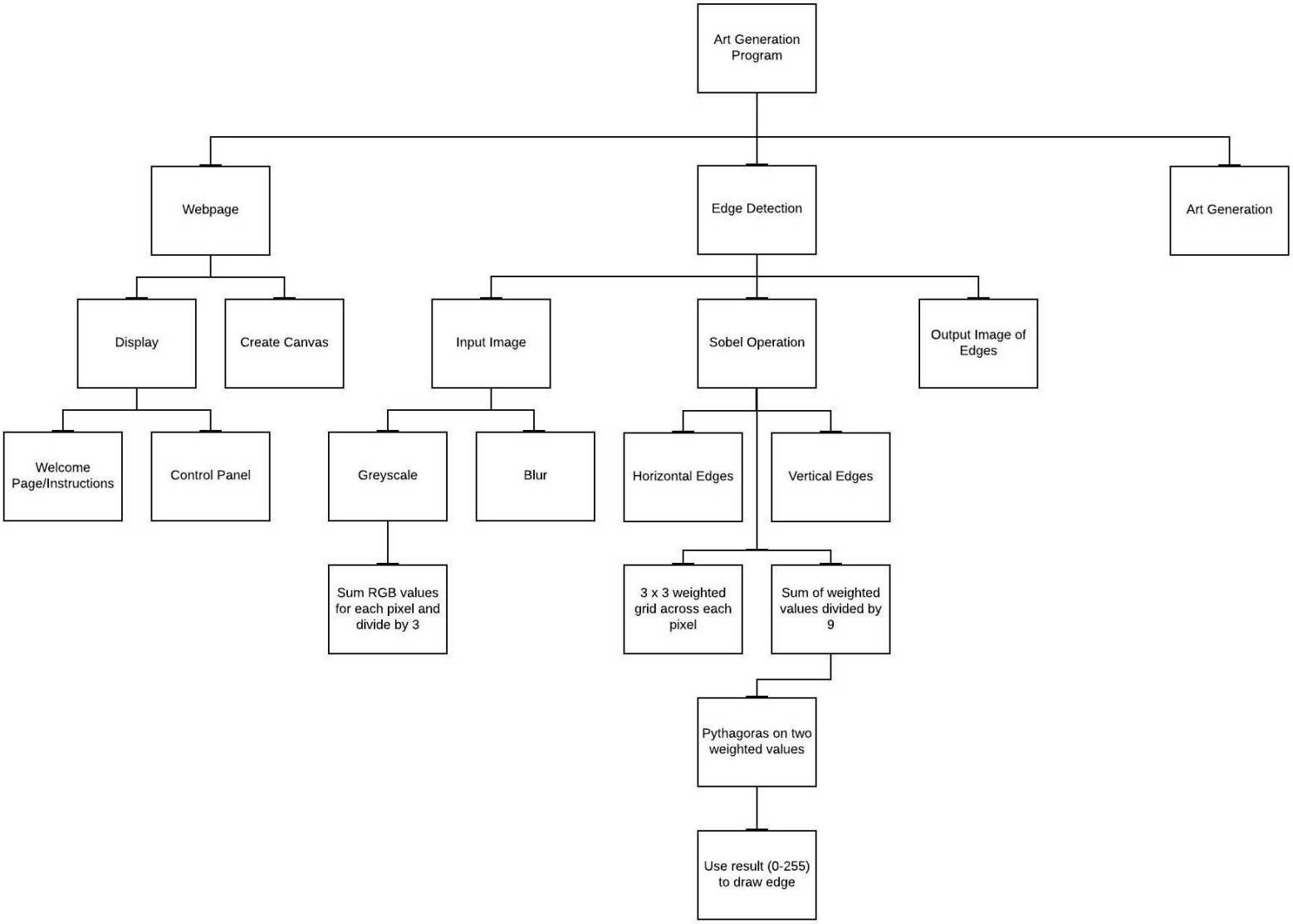
Art Generation :

17. Analyse image produced by Edge Detection and view each pixel's data
18. Examine greyscale value (strength of edge) of each pixel
19. Draw circle in place of pixel (in colour of original pixel?)
20. Circle opacity dependent on strength of edge
21. More circles generated from original points, with random size/direction?
22. Iterations of further circle generation dependent on strength of edge

User Options :

23. Fading : Each circle placed should have a higher transparency than the last
24. Shrinking : Each circle placed should be smaller than the last
25. Multicolour : Each circle takes a random colour
26. Tonal : A three colour palette is selected, and each circle takes one of these colours
27. Mirroring : Central line of symmetry so each circle is reflected in the y-axis
28. Canvas Colour : User select the background colour of the canvas
29. Save Image : Image can be downloaded to user's device

This Hierarchy Chart illustrates the program, separating the three keys focuses of my project; a webpage to host the software, the edge detection program and the art generation, breaking down each into a set of sub-tasks.



This flowchart shows the complete flow of the system, indicating the main processes that will be performed, the data passed between each function and the order in which they shall be completed.

Pixel Manipulation

In order to process the input image pixel by pixel I needed to import the image in a manner that would allow me to deconstruct it into its individual pixels that could be viewed and analysed as single data points.

For this I would use the ImageData[8] object which creates a One-Dimensional Array containing the RGBA (Red, Green, Blue, Alpha) values of each pixel and stores these 4 values consecutively in the array. Therefore, when searching each pixel of the image, the row and column numbers must each by multiplied by 4 to find the index at which that pixel's data is stored.

	0	1	2
0			
1			

Where shaded cells represent pixels in an image with RGBA values between 0 and 255.

This table represents the 1d array that would store the RGBA values for each pixel, with the data structure moving across each row sequentially.

Pixel 0,0				Pixel 0,1				Pixel 0,2				Pixel 1,0				Pixel 1,1				Pixel 1,2			
R	G	B	A	R	G	B	A	R	G	B	A	R	G	B	A	R	G	B	A	R	G	B	A

The array starts at index 0 and would have a length = (number of pixels *4)

The index of an individual pixel would therefore be:

Row (No. columns * 4) + (Column * 4) + RGBA index

Therefore, to access the Green value of Pixel 1,1 the index value would be :

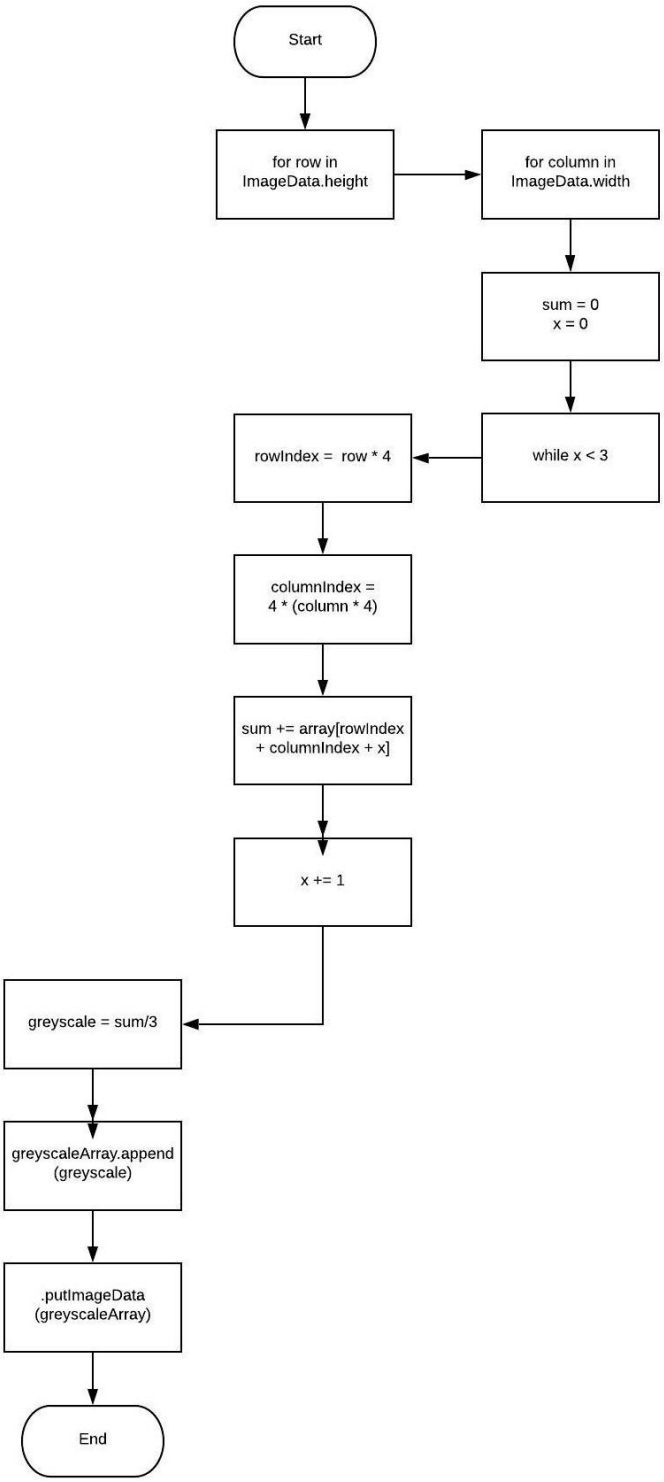
1 (4 * 3) + (1 * 4) + 1 = 17

Greyscale

To convert a pixel to greyscale the RGB values must be summed and divided by 3.

Therefore each pixel in the image must be looped through, and a sum of the first three indexes (RGB values) of that pixel in the array will be collected, then divided by 3 to produce an average, taking into account each colour channel of the pixel and representing it as a greyscale value.

The image data object has a method called putImageData()[9] which can be used to paint pixels back onto a canvas. Therefore, in order to produce a greyscale version of the image I would take the greyscale values for each pixel and push them onto a new canvas, which would then become the image used for the edge detection.



This flowchart illustrates the process of converting the RGB image into a greyscale version, looping through each column in each row and analysing each pixel one by one.

The variable 'x' increments to allow for the red, green and blue value of each pixel to be accessed in sequence. As the alpha channel is not required for the greyscale operation, 'x' only needs to increment to look at the first three indexes of each pixel's data.

The variable 'array' will be the array assigned to the result of calling the 'getImageData'[10] function on the canvas, which will create a 1d array of the RGBA values for each pixel on the canvas.

These three values are totalled and then divided by 3 to produce a greyscale value. Each pixel's greyscale value is appended to an array which will then get passed into the putImageData module.

This will take each greyscale value and paint it onto a pixel in a new canvas, therefore forming a greyscale image.

Sobel Operation:

The Sobel Operator[11] works with the X and Y directions separately, calculating the gradient of the x values and then the gradient of the y values and the final output is a combination of both that represents the edges of the entire image. This method uses Kernel Convolution to look at each pixel in the image and apply a weighted 3x3 matrix over it to calculate the gradient of the pixel, considering those surrounding it.

Unlike the Prewitt Operator, the Sobel Operator has no set values for weighting the pixels, therefore any reasonable values could be used provided that they followed the correct negative/positive pattern for a horizontal and vertical kernel. This made the Sobel Operator very suitable to use in my project as I could experiment with different weightings and see how they affected the accuracy and strength of the edges detected. This would allow me to find a balance between an abstract yet still accurate representation of an image, by changing the behaviour of the edge detection program to produce varying outline representations that I could then use as the base for the art generation to see what the output looked like.

-1	0	+1
-2	0	+2
-1	0	+1

This is the kernel matrix used for the gradient in the Y-direction, with negative weightings on the left-hand side and positive on the right.

The weighted values of the 9 pixels are then summed and the result determines the strength of the edge in the X-direction, with a larger value indicating a more noticeable edge. For example, if there was no change in greyscale values between two pixels (i.e. no edge) the negative and positive values would cancel out and result would equal 0, indicating that there is not an edge in the X-direction between those two pixels.

+1	+2	+1
0	0	0
-1	-2	-1

Similarly, this diagram shows the kernel matrix used for gradients in the X-direction, with the same negative and positive weightings but in a different configuration, studying the change in the pixels above and below the central pixel. These weighted values are once again summed to represent the strength of the edge in the Y-direction.

The magnitude of the gradient of the pixel is then calculated using Pythagoras' Theorem on the two values from the X and Y kernels.

When looking at the pixels on the edge of an image a section of the kernel will lie outside the border of the image, and therefore attempting to apply a weighting to pixels that do not exist[12]. There were several different ways I could approach this problem; I could multiply all the cells lying outside the image by 0 or ignore those cells and remove them completely from the calculation. Both would achieve the same result, yet if I removed the cells I would then have to account for a reduction in the size of the kernel and therefore change the number divided by at the end. Therefore, I concluded that, at an edge when a cell was trying to read a value from a non-existing pixel, I would overwrite the code to multiple the cell by 0 and therefore leave the kernel unchanged.

Traditionally each kernel is stored as a 2-dimensional array, however I found that if I ordered the values correctly in a one-dimensional array then I could use the same subroutine to move the kernel across each pixel in both the x and y direction, by moving across the pixels row by row and simply changing the 1d array passed into the function, depending on whether the x or y direction is being weighted.

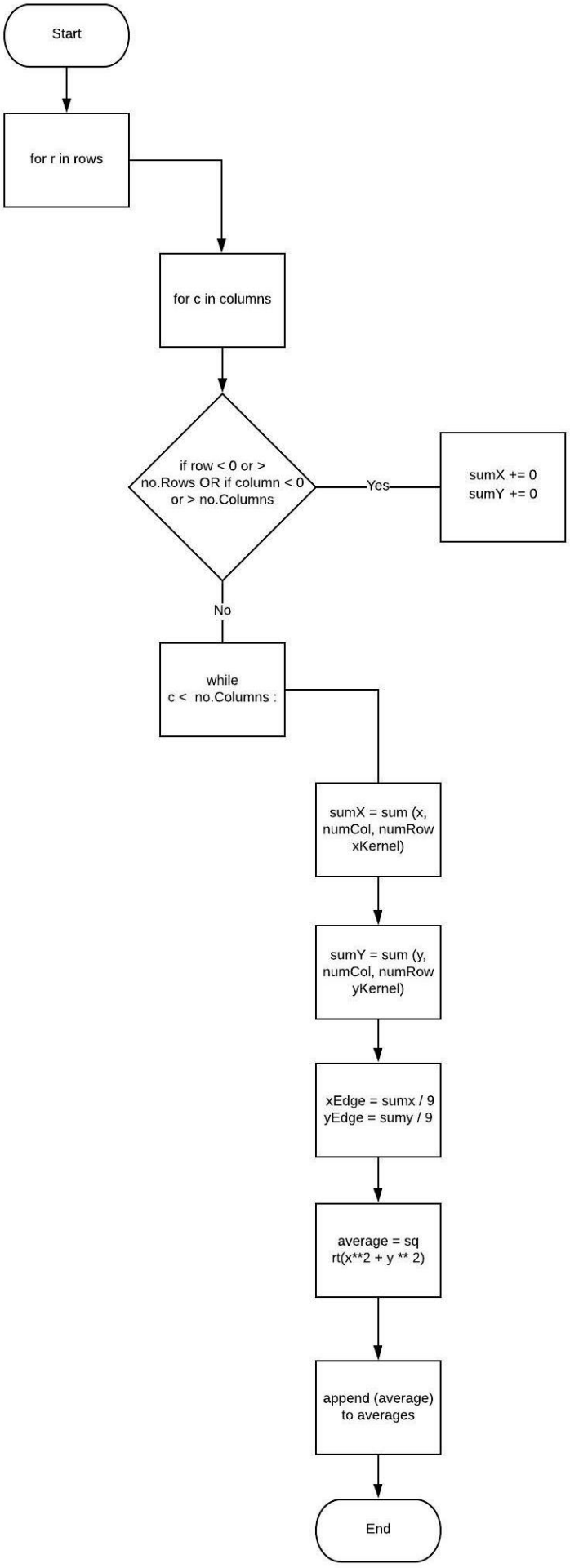
yKernel = [-1, -2, -1, 0, 0, 0, 1, 2, 1]

xKernel = [1, 0, -1, 2, 0, -2, 1, 0, -1]

1	4	7
2	5	8
3	6	9

This diagram shows the order in which each pixel contained within the kernel would be weighted; with the top left pixel given the weighting in the 0th index of each kernel array, and the bottom right given the weighting in the 8th index.

Sobel Operation :



This flowchart illustrates the main process of the edge detection, the Sobel operation performed on each pixel to find the strength of it as an edge.

The algorithm loops through each row, and for each row every column is analysed sequentially.

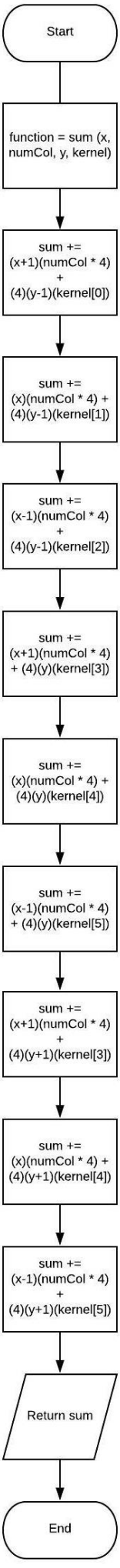
If the row or column selected is less than 0 or exceeding the number of rows/columns, it means that the row/column does not exist and lies outside the edge of the image. Therefore, if this event occurs, the program is overwritten so that the pixels' values are assigned as 0. This ensures that the pixels lying on the edge of the image can be analysed using the same algorithm instead of having to treat them separately to handle the issue of the kernel lying outside the image's boundary.

Otherwise, the pixel in the row and column will be found, in the array storing the greyscale values of each pixel. The value of this, and the surrounding 8 pixels, will then be weighted using the Kernel arrays. Firstly, in the horizontal direction using the 'xKernel' and then in the vertical direction using the 'yKernel'.

These weightings will be added to a sum variable which when divided by the number of pixels examined (9) will represent the strength on the edge in the x and y directions respectively.

An average of these two values will be calculated using 'Pythagoras' Theorem' to produce an overall estimation of the strength of the pixel as an edge.

This value will be appended to a list which will hold the 'edge strength' values for each pixel in the image.



This flowchart explains the flow of the 'sum' function in more detail, showing how the pixel and the 8 surrounding it are examined and weighted accordingly. To make this function reusable for both the horizontal and vertical directions the function takes in the X coordinate, Y Coordinate, Number of Columns and either the xKernel or yKernel (in turn) as parameters. Each pixel in the 3x3 grid around the pixel being examined is then weighted according to the values in the kernel and added to an overall sum.

Row above, Column to left

Multiplied By : x(-1), y(1)

Row, Column to left

Multiplied By : x(-2), y(0)

Row below, Column to left

Multiplied By : x(-1), y(-1

Row above, Column

Multiplied By : x(0), y(2)

Row, Column

Multiplied By : x(0), y(0)

Row below, Column

Multiplied By : x(0), y(-2)

Row above, Column to right

Multiplied By : x(1), y(1)

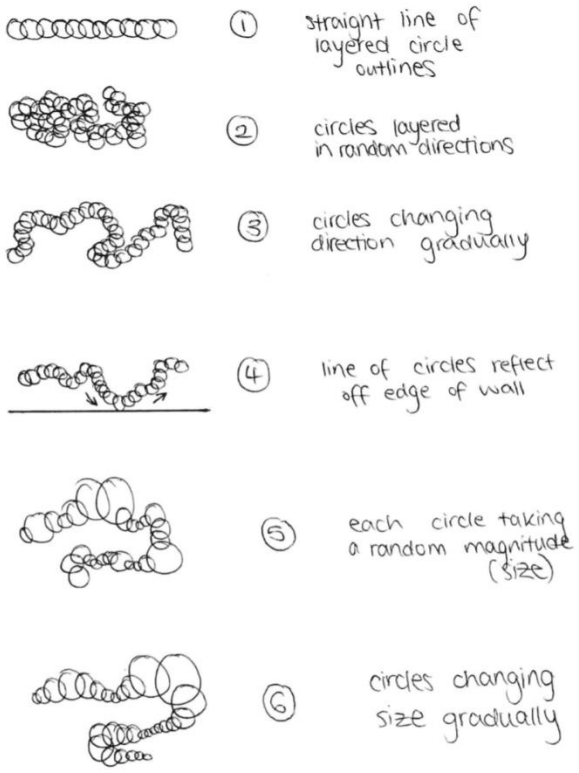
Row, Column to right

Multiplied By : x(2), y(0)

Row below, Column to right

Multiplied By : x(1), y(-1)

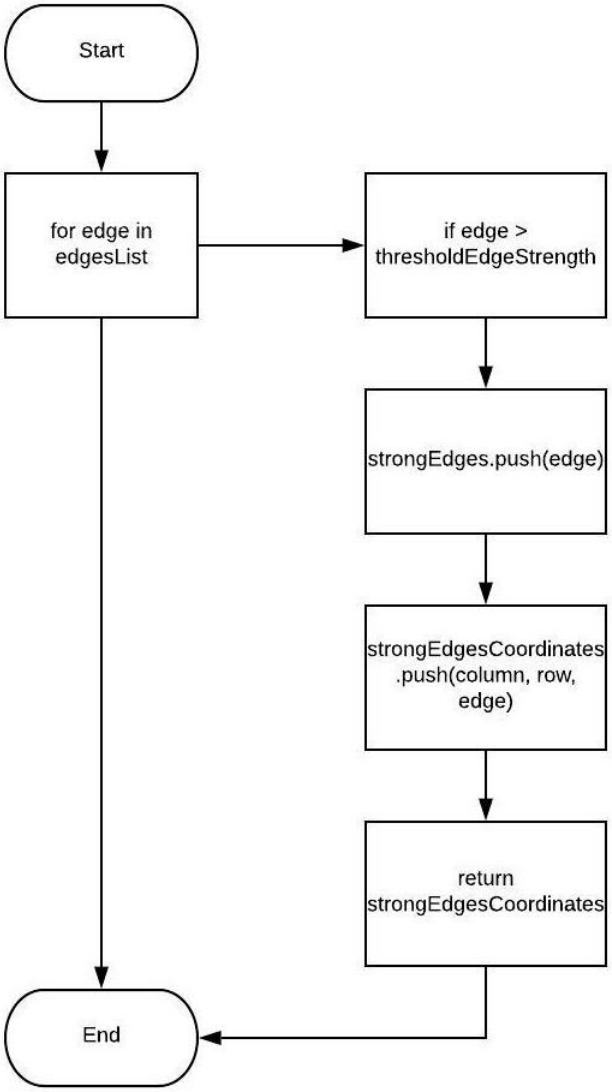
As my project is based around a visual output that would develop and evolve over the course of the project as I created my algorithm, I was not completely certain what the end product would look like and therefore could not design the algorithm in pseudocode/flowchart knowing that this would be what the final code would follow. However, in order to allow my algorithm to develop so that it would create successful organic-based art I could design the initial stages of simple art generation, allowing me to practise how the art could be generated and therefore adapt my algorithm as it progressed.



Preliminary drawing of a potential design for the art generation, consisting of a path of circles which undulated and travelled across the canvas.

Having decided to program the greyscale and edge detection parts of my project first, I returned to this design section later, to design the art generation, now knowing what the output of the previous algorithms would be.

I knew that the art was to be generated from a series of circles drawn on the canvas, and that I wanted to size and concentration of these circles to be dependent on the strength of the edge detected at each pixel. Therefore, I would have to analyse the edge strength of each pixel in the edge detection output and construct an algorithm to draw a circle dependent on a parameter of the edge strength.



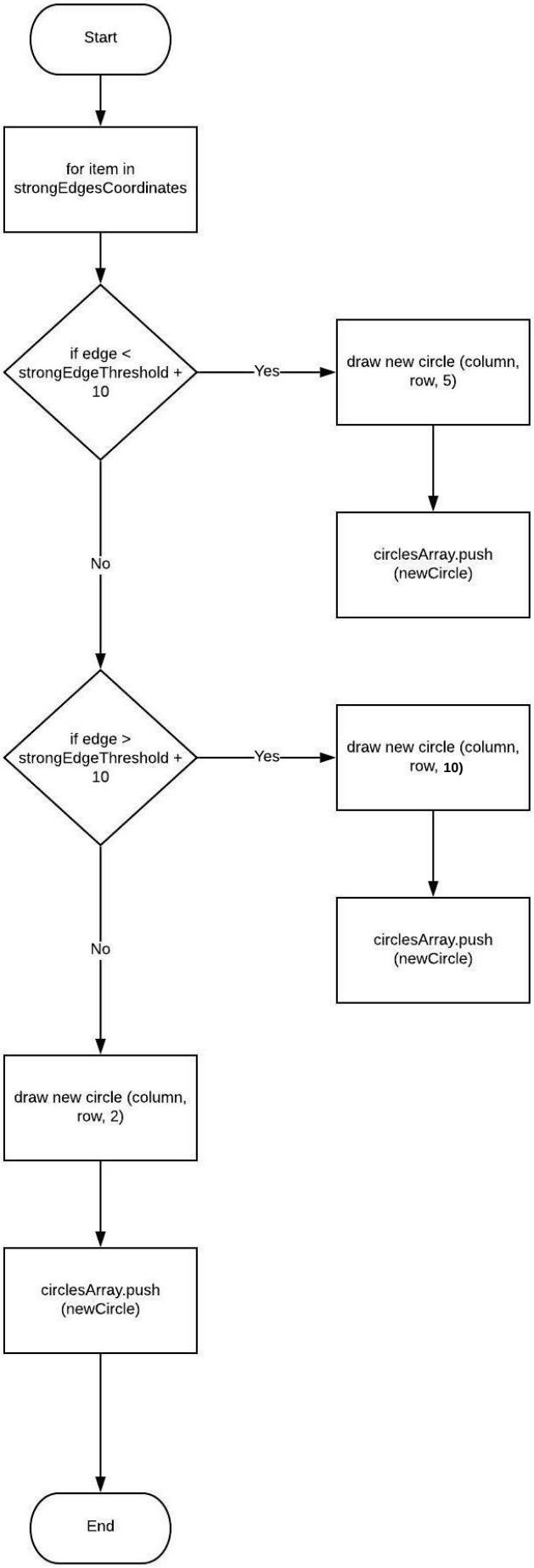
To start with I would draw a circle on the canvas when a pixel was considered to have a 'strong edge'.

A 'Strong Edge' could be defined one of two ways:

- A pre-set value that would remain constant throughout the running of the program. However, as the user could input any image, with any varying strength of edges, a constant value may be inappropriate as it could lead to too many or too few edges being considered 'strong' for the art generation to be based on.

- A variable slider that the user could change to be able to choose what the 'strong edge threshold' should be, and therefore, allowing them to alter how many edges to generate art from. Not only ensuring any image could produce an effective visual output, but also allowing more user interaction to experiment and change the art generation themselves

The algorithm would then compare each edge in the image (using the ImageData array from the edge detection canvas) against this 'strong edge threshold' and if the edge is greater than this value, then it will be added to a list 'strongEdges', with a separate list 'strongEdgesCoordiantes' storing the pixel's row and column in the canvas.



Once a list of 'strong edges' has been compiled the art can begin to be generated. Iterating through each strong edge in the list, the edge will be compared to a variation of the strong edge threshold.

These will consist of the threshold with a set mount added to it (e.g. 10) and the threshold with the same amount subtracted from it.

A circle with then be generated in the same row and column as this edge, with the circle's size depending on the size of the edge strength in comparison to the threshold. This should result with a stronger edge having a larger circle placed on top of it.

The circles drawn on the canvas will be created as multiple instances of a 'Circle' class. With an associated method that would control the drawing of each circle on the canvas. By using a class structure to handle the circle generation it would ensure that each shape drawn on the canvas would have shared properties, allowing me to use the same functions and logic to manipulate each instance of the Circle class.

An array will be used to store the data, including the size and position of each circle, for the shapes drawn onto the canvas. Therefore, if I wanted to generate more shapes from the first set of circles, I could access this array to find the strongest edges and therefore the concentration of circles on the canvas.

Class Design :

Class **CanvasData**

#original : array

#greyscale : array

#edgeDetection : array

#selectedStrongEdges : array

As each algorithm performed on the input image rewrote the pixel data on the canvas, If I was to be able to reference the canvas and it's contents at different stages throughout the program then I would need to have a class that could store each of these canvas states as unchanging values. Hence the CanvasData class would be used to store the 4 key arrays consisting of lists of pixels and their data, that I could then pass in as parameters to the necessary functions in different parts of the program.

Class **ColourPixel**

#x : integer

#y : integer

#radius : integer

+ highlightPixel (x, y, radius)

The ColourPixel class was used to show the user which/how many pixels would be selected as stongg edges to be used in the art generation. If a pixel's strength was over a certain threshold then a new instance of the class would be created for that pixel, with the method 'highlightPixel' changing the colour and opacity of the pixel so that it appeared 'highlighted' on the canvas.

Class **Circle**

#x : integer

#y : integer

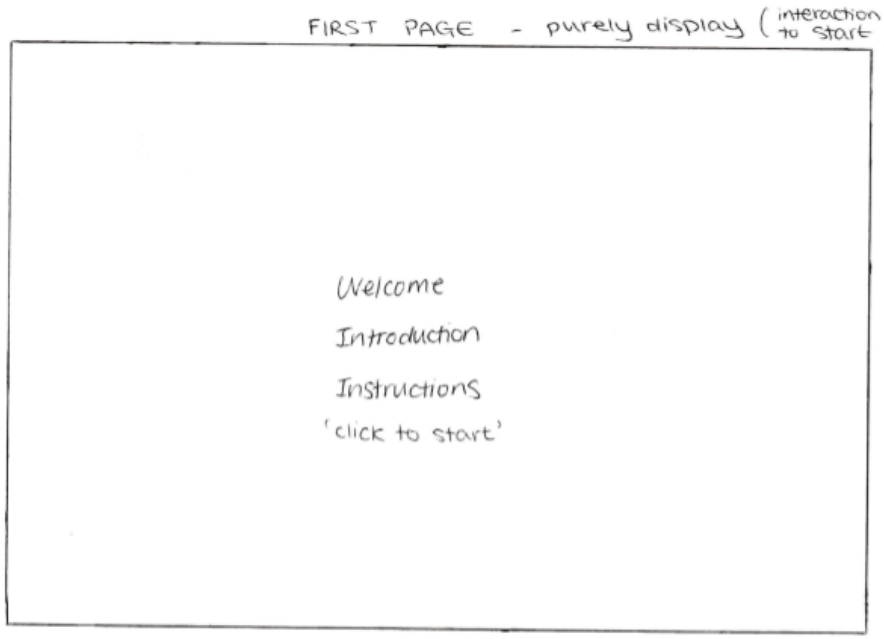
#radius : integer

#imageData : array

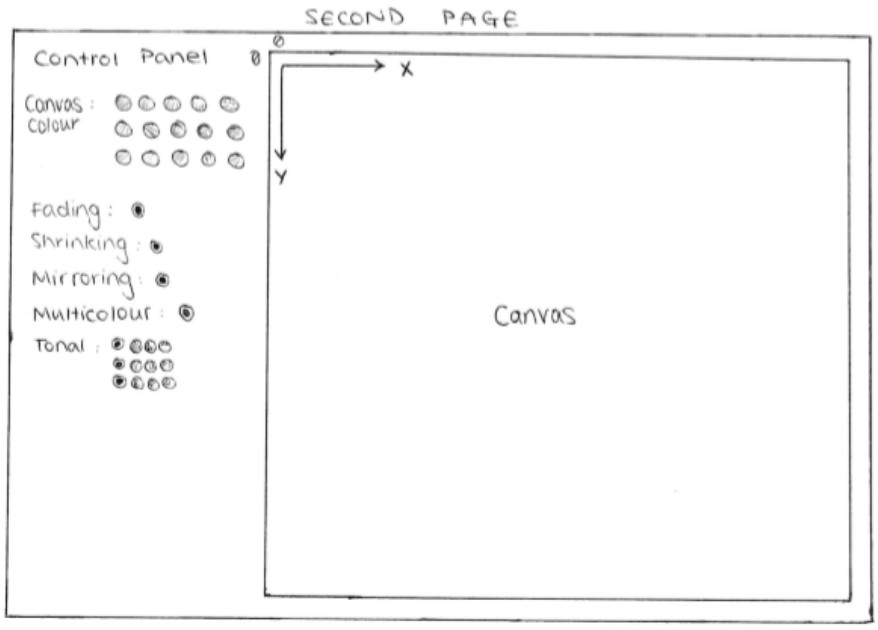
+ place (x, y, radius, imageData)

The Circle class would be used the most in my program as this class would generate circle shapes on the canvas, taking the x and y coordinates of the pixel the circle is to be placed over and a radius defining the size of each circle. The attached method 'place' would draw the circle on the canvas using the colour of the pixel of the original image.

Designing the Interface :



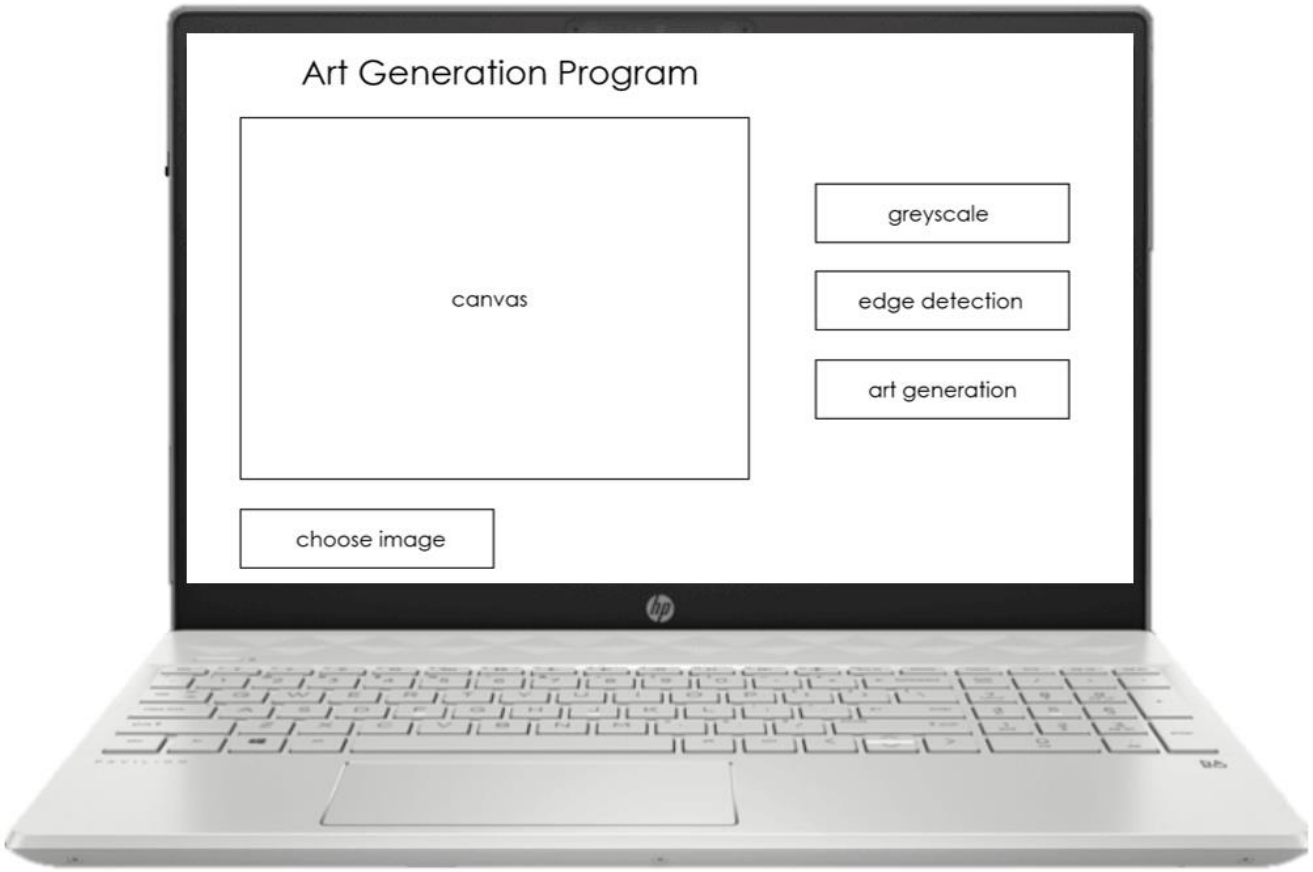
This was the first mock-up that I created when considering what the interface would look like and how the user would interact with it to control the program. Initially the user would be presented with a welcome page, which would introduce the program, explain its purpose and how to use it. As there are very few instructions that the user would need to be aware of, it is unlikely that I would need to create a whole new page to host these, therefore a small pop up box at the side of the welcome page would probably suffice to host the instructions. When 'Click to Start' is selected the user will be presented with the main page, hosting the HTML canvas and various buttons to control the art generation.



The control panel would include buttons to allow the user to tailor the colour, style and shape of the art generation to their own personal preferences. The user must be able to see the canvas and the activity on it very clearly, therefore the canvas should take up the majority of the screen and the buttons all pushed to one side.

After writing the majority of my final program I returned to my interface design to evaluate how appropriate it was for the program I had created. As I had developed my program and its function as I went, I found that many of the user controls were no longer valid and did not fit with the direction my art generation program had taken. Therefore, I produced a new mock-up design, featuring many of the same elements but with more specificity for what my current program needed.

In this new design the user was presented with buttons to trigger the four key events that would occur when using the program; the selection of an image to load onto the canvas, the grey scaling of that image, the edge detection algorithm and finally the art generation. This led to a minimalistic, yet simple design that would not draw away from the art generation on the canvas, as this result of the art generation should be the main visual on the page, and any complex buttons or titles would distract from the canvas and lead to an overcrowded interface.

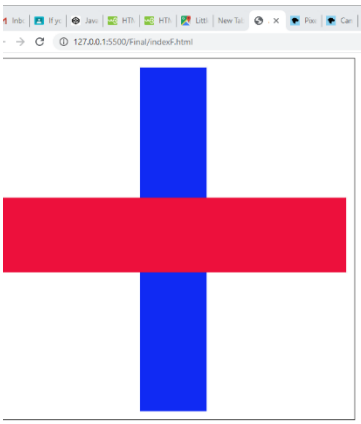


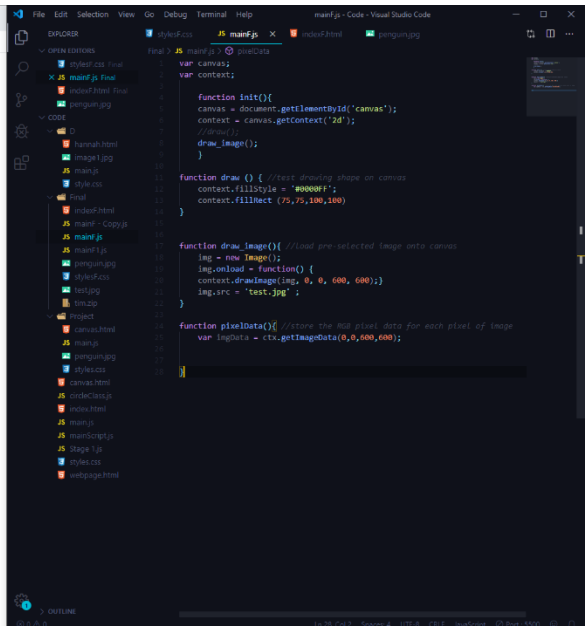
Updated Interface Design

Technical Solution

When it came to the implementation of my project, I started with the edge detection. This included loading an image into a canvas hosted on a webpage, grey scaling the image and then performing the Sobel Operation on it.

Firstly, I experimented with initialising a canvas environment and drawing basic shapes such as a rectangle on the canvas.



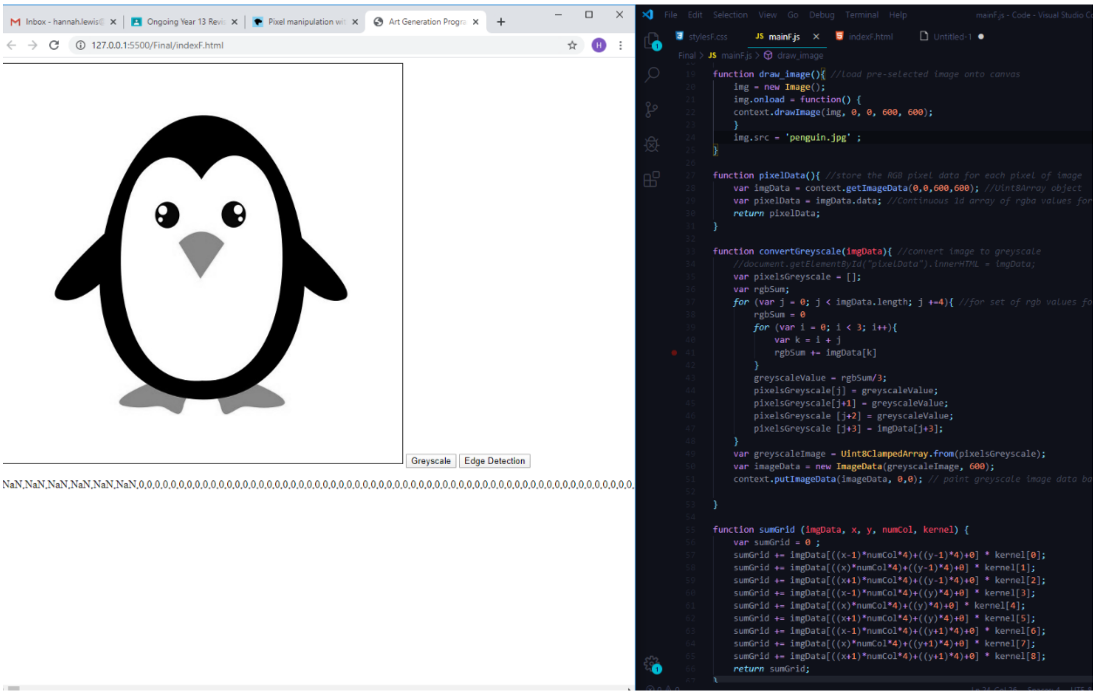


I created a blue and red cross 'test image' in Photoshop as the pure colours would make it easier to check if the red, green and blue channels were producing the correct values when analysing each pixel of the image using the 'imageData' object

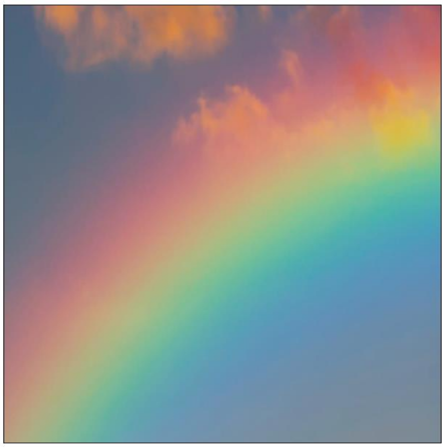
The 'getImageData' method created a 1d array consisting of the RGBA values for each pixel in the canvas. By outputting the array to the screen, I could check that the RGB values were changing when the image changed from a section of pure red or blue to white by seeing that the RGB values were no longer 0.

Knowing that I could store the pixel data for the canvas I began to code the grey scaling function; this involved looping through each pixel's data in the array, summing the RGB values and dividing by 3 to obtain the average of the pixel's colour (greyscale). For this I needed to set up two for loops, one to loop through each pixel's data (as each pixel took up 4 indexes for RGBA values) which would increment by 4 each time to analyse the next pixel. The second loop would run 4 times, to loop through each red, green, blue and alpha value of a pixel separately. In order to check that the greyscale function was outputting the correct values I intended to print the array of calculated greyscale values to the screen, however this array's length was equal to the number of pixels in the image, making it very long and therefore it took a long time to output the array, to the extent where the webpage would load for so long that it would eventually crash. In order to resolve this, I reduced the size of the image to 600px by 600px, this allowed the program to print a shorter greyscale array to the webpage, yet the array consisted of all 0 values. I traced this back to the 'pixelData' array also containing only 0s. I knew that the 'pixelData' function was working correctly before and so something must have changed when the size of the canvas had been reduced.

After checking that all of the logic of my was correct I concluded that there must have been some issue in the HTML's running of the functions, as all of the pixel values were equal to 0 I assumed that this meant that these functions were running on an empty canvas, concluding that the 'pixelData' and 'convertGreyscale' functions were being run before the initial image had even been loaded onto the canvas. To resolve this I implemented a button that the user would press to trigger the running of the greyscaling function. This ensured that the image would be loaded onto the canvas, before the user pressed the 'Greyscale' button and activated the function which then ran successfully.

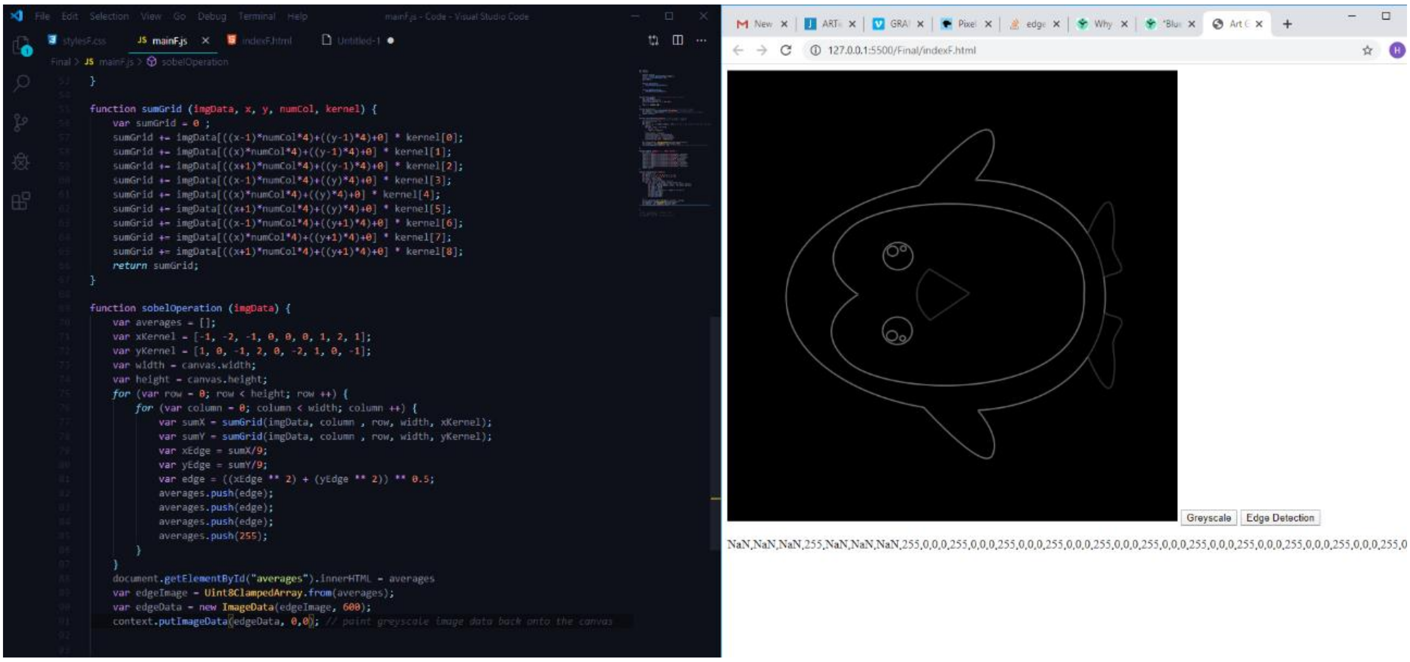


Testing Functionality of Greyscale Subroutine :

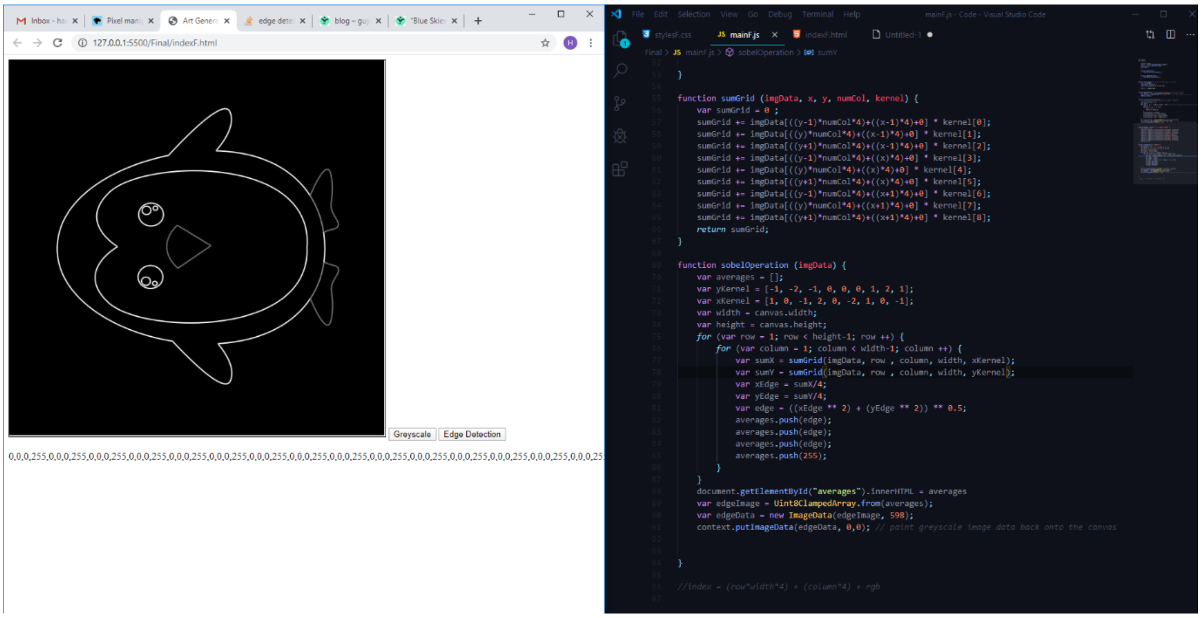



Before moving onto implementing the edge detection I completed a series of tests to check how effective my greyscale function was. Firstly, testing with images with a range of colours to check how the function handled more saturated images. I then inputted images with a more consistent colour palette to test if the function could distinguish between slight changes in colour. These test images showed that the grey scaling function was working to a satisfactory level.

When it came to the implementation of the edge detection subroutine, I referred to the flowcharts I had created describing the Sobel Operation, allowing me to easily translate this logic into code.



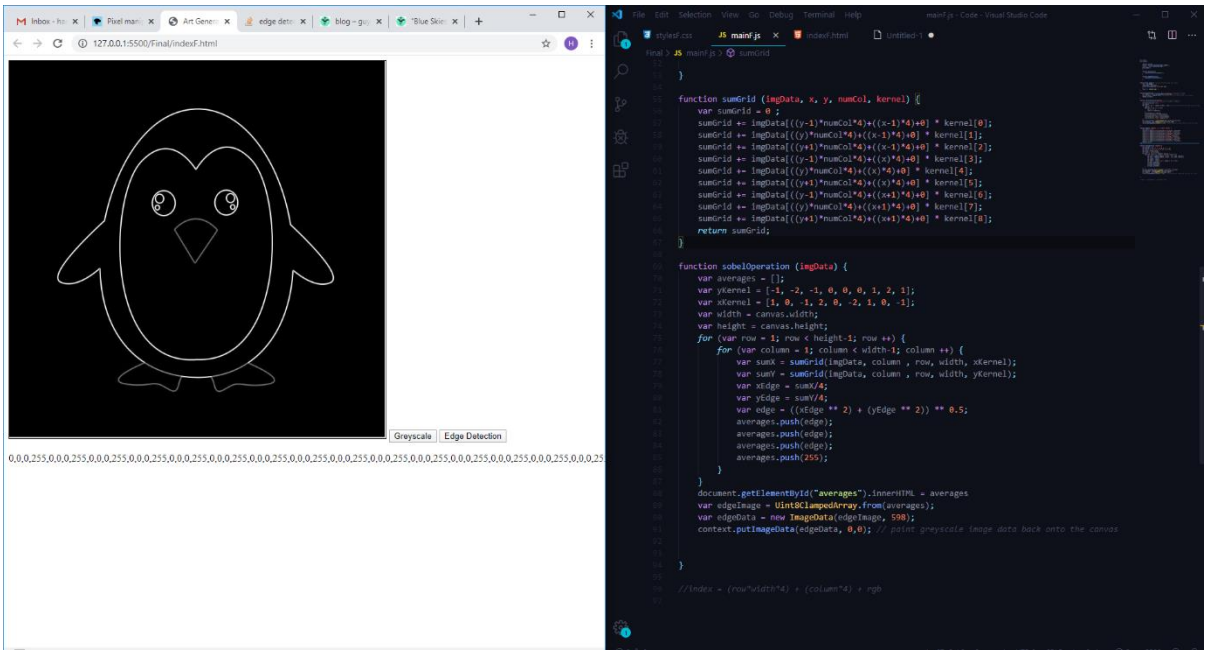
However, there were three obvious errors in the running of the function:



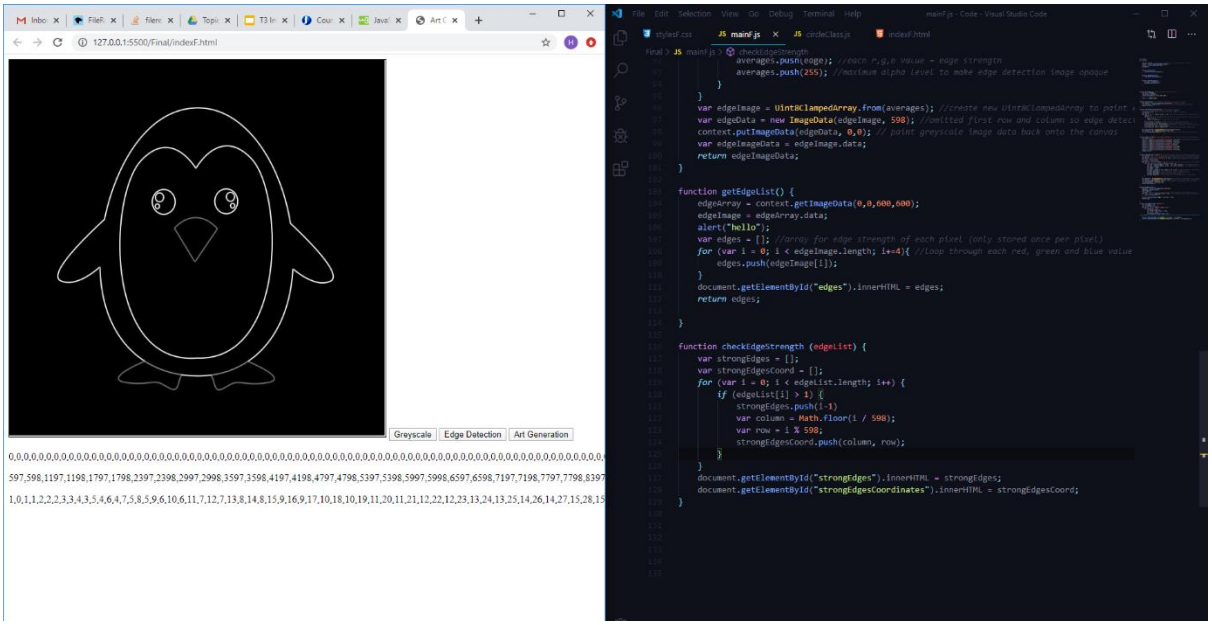
Firstly, the edges did not appear as strong as they should be, this meant that less strong edges were barely visible against the black background. This meant that the edge values being outputted must have been smaller than they should have been, I traced this back to the division of the sum of the horizontal and vertical edge strength around a pixel. Originally, I had been dividing the sum of the edges in each direction by 9, believing that I needed to divide by the number of pixels being analysed in each kernel convolution. However, having studied the logic of my function again I realised that because of the weighting of each pixel's greyscale value the maximum value for the weighted kernel was 4. This is because the strongest edge would be a change from pure white to pure black, meaning the pixels either side of the pixel being examined would have the greyscale value of 0 and 1, meaning that after applying the weighted kernel the greatest result would be 4. Therefore, I needed to be dividing by the greatest possible value to create the correct fractional value, and after making this adjustment the edge detection output was a lot clearer.

Secondly, although the outline of the image's edges was being painted back onto the canvas, it was rotated 90 degrees to the left. This orientation issue implied that there was a problem in the logic of the function regarding the x and y values being used. Because the image was being painted as it the x values were relating to the y values and vice versa this implied that somewhere in my code the x and y values needed to be swapped.

Lastly, when outputting the edge detection array some of the indexes held the value "NaN" indicating that these values were "Not A Number" and therefore that the edge detection was not being fed number values for some pixels. I had expected this error as at this point I had not coded how to handle the edges of the canvas, as when the edge detection was run on these pixels the function was attempting to read the pixels above and to the left/right yet these indexes did not exist in the 'pixelData' array as they lay outside the canvas. This issue was one that I was aware would occur and decided to ignore initially, wanting to focus on getting the majority of the art generation finished before returning to create a solution to handle this issue. Therefore, in the meantime I decided to adapt my code around this problem by starting the edge detection on the second row and column of the image, meaning that no edge pixels were being passed into the edge detection function. This meant that the resulting image from the edge detection had a width and height of 598 pixels instead, meaning that there was an empty row and column on the canvas after the edge detection occurred.



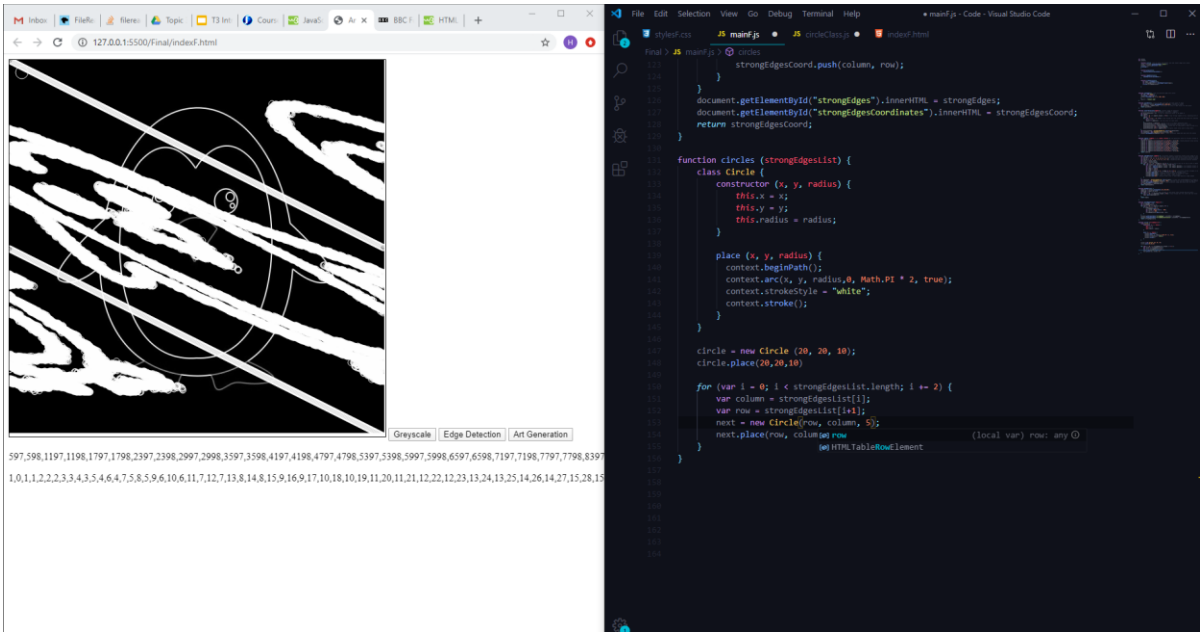
After having mostly completed the edge detection function I moved onto the art generation part of my project. The process to follow for this was less clear as I was expecting the algorithm to develop as I progressed and tested different methods of generating visuals images. I knew that I wanted there too be more focus on the stronger of the image's edges, therefore I created a function that would create a list of coordinates where the edge was over a certain threshold. Eventually this threshold would be calculated based on the strength of all the edges in the input image, yet initially to get the logic working I predefined all the edge strength thresholds used.



I outputted the canvas coordinates of all pixels with an edge strength over 1 to the screen.

I then created a 'Circle' class with a function 'place' that would draw a small circle on the canvas, with the coordinates and size of the circle being determined by parameters.

I began by simply coding a function to place a circle over every pixel where an edge had been detected. Upon running this there was an obvious error in the code, as the circles were not being placed over the original image, instead the circles were forming a skewed and stretched version of the image.



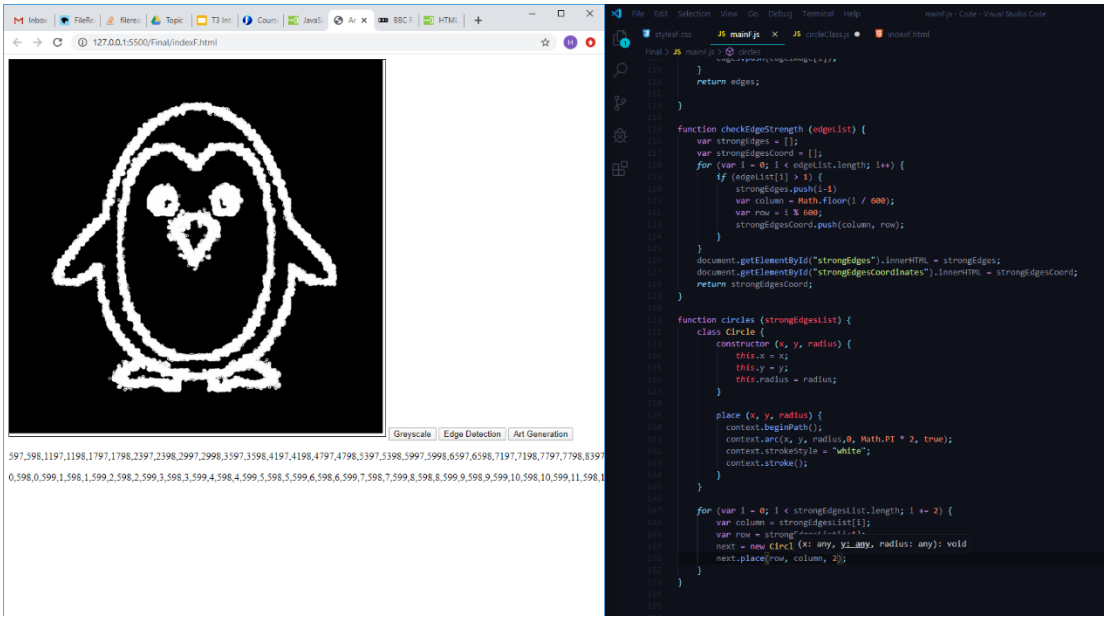
As the circles were still forming the original shape of the image, just a distorted version of it, I knew that the Circle class and functions were working correctly and that the issue must be with the parameters passed in. To investigate this I printed the contents of the 'strongEdgesCoord' array and saw that edges were being recorded at coordinates such as (1,0) were the image was just a single block of colour, meaning that edges were being recorded in coordinates where there were no edges. From my previous testing (edge detection rotated 90 degrees) I had learnt that the orientation of a canvas drawing could be altered by the x and y values being the wrong way round, and so I returned to where the coordinates of the edge were being stored in 'strongEdgesCoord'.

This was the original declaration of the row and column in which a ‘strong’ edge was situated

```
var row = Math.floor(i / 598); //find y coordinate in canvas
var column = i % 598;
```

I swapped the variables ‘row’ and ‘column’ so that they now were assigned to each other’s value, hoping that this would solve the orientation problem. This was successful and the overlayed circles were now the same orientation as the original image. However, it was still stretched across the canvas, as if it was trying to fill the full space of the canvas. This led me to believe that the division by 598 was incorrect, as I had thought that I needed to divide by the size of the image in order to gain the row/column. However, this column and row was in relation to the entire canvas, which still had the original width and height of 600. Therefore, by changing the code to divide by 600 the row and column of each edge was now correct, and the placement of the circles fit the image.

```
var column = Math.floor(i / 600); //find y coordinate in canvas
var row = i % 598;
```

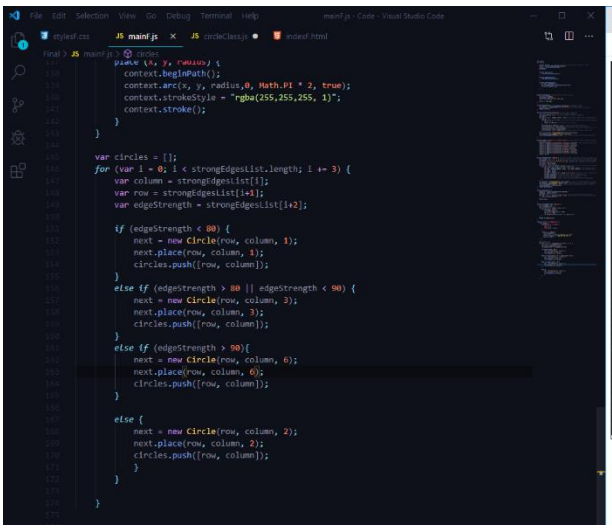


This shows the circle placement in the correct orientation. I had also increased the edge strength threshold to greater than 20 here, experimenting with how high the threshold needed to be to ensure that a circle was not placed at almost every pixel as this led to an unattractive fuzzy and blurry outline.

Given that the end goal was for the use rot be able to input any image to run through the edge detection and art generation functions this would mean that the threshold would have to change each time as some images will have fewer/weaker edges than others. Therefore, I would eventually include a slider to allow the user to determine the threshold of a ‘strong’ edge and hence choose how many circles should be placed and how closely they follow the original image outline. This slider would update in real time so that the user could see how abstract or realistic the art generation would be, based on the placement of the circles. Yet I would add this feature in later if I had enough time to develop the user interface further and for now continued to adjust the threshold manually, according to the detail and edge strength present in whichever test image I was using.

These screenshots show the change from an edge strength threshold of > 20 to a threshold of > 60. A lower threshold builds a more uniform line of circles around the edge of the bird and branch, yet a greater threshold creates a sparser image, allowing you to see some individual circles. This proved that a lower threshold would be more appropriate for my project as I wanted the individual shapes to be visible and for a pattern to be generated which loosely resembles the original image and does not simply outline the original image.

I also experimented with the opacity and colour of the circles. Finding that a less opaque circle made it blurrier and therefore more difficult to see the image, whereas a completely opaque circle creating a clearer image. Placing black circles created an interesting inverted effect, and although I liked this 'embossed' look, it was not easy to see the image and circle generation.

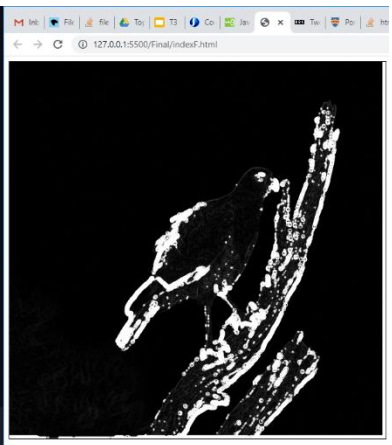


```
var circles = [];
for (var i = 0; i < strongEdgesList.length; i += 3) {
    var column = strongEdgesList[i];
    var row = strongEdgesList[i+1];
    var edgeStrength = strongEdgesList[i+2];

    if (edgeStrength < 80) {
        next = new Circle(row, column, 1);
        next.place(row, column, 1);
        circles.push([row, column]);
    }
    else if (edgeStrength > 80 || edgeStrength < 90) {
        next = new Circle(row, column, 3);
        next.place(row, column, 3);
        circles.push([row, column]);
    }
    else if (edgeStrength > 90){
        next = new Circle(row, column, 6);
        next.place(row, column, 6);
        circles.push([row, column]);
    }

    else {
        next = new Circle(row, column, 2);
        next.place(row, column, 2);
        circles.push([row, column]);
        }
    }

}
```



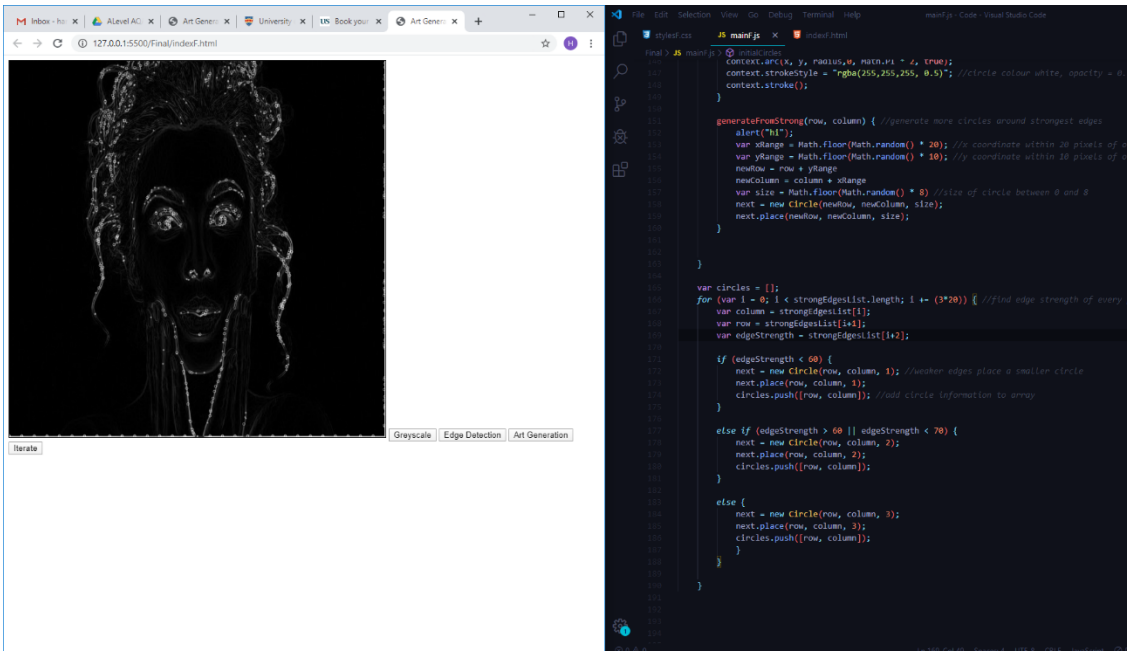
In order to vary the pattern created by the circle placement I decided to change the size of the circle depending on the strength of the edge. Firstly I code an if statemebt that increased the size of the circle as the edge strength increased.

I then tested how decreasing the circle size as the edge strength increased would change the art produced, hypothesising that smaller circles would create a harsher line, therefore highlighting a strong edge. However I found that this negative correlation between circle size and edge strength reduced the definiton of the image and did not look appealing, making it unsuitable for the art generation.

Increasing circle size with edge strength

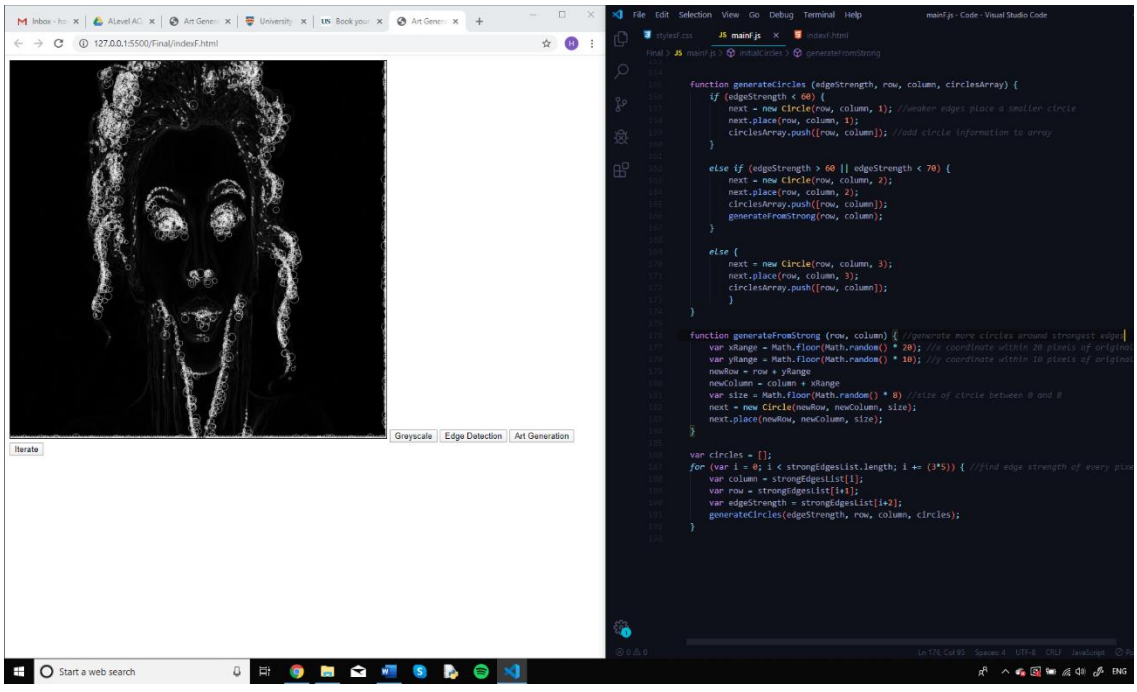
Decreasing circle size with edge strength

However, conscious that I was only using one image to make a big creative decision about the art generation, I tested each method with another image, finding that I still preferred the look of circles increasing as edge strength increased.



In the previous screenshots I thought that the circles were too concentrated as there were often too many circles in one vicinity. Therefore I changed the code to only analyse every 30th pixel, limiting the number of circle placements in one section of the image, and therefore creating a less overpowering and more subtle image.

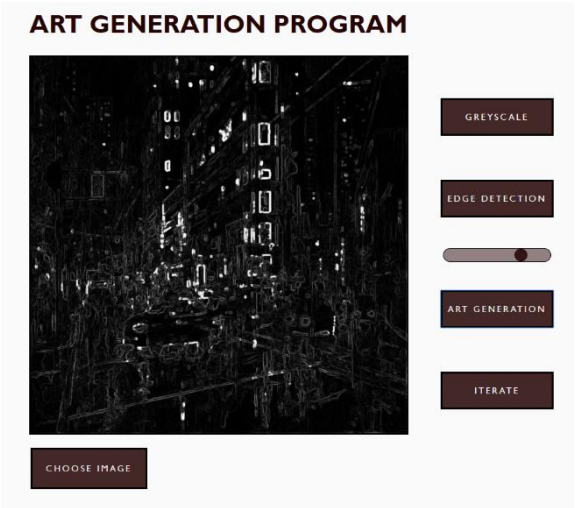
In order to achieve a more random art generation, I added a function named 'generateFromStrong' which would generate a random number of randomly sized circles, which would be placed within a certain x and y range of a pixel. Only edges over a higher threshold would have these extra circles generated around them.



This was still resulting in some clustering of circles which I found unappealing, and to solve this I introduced a Boolean variable 'circlePlaced' which would change value depending on whether a circle had been placed for each pixel, if a circle had been drawn onto the canvas, the algorithm would skip a certain number of pixels. This ensured that if a pixel had a circle placed over it, then all the pixels within a certain range around it would not have any initial circles, therefore preventing the clustering.

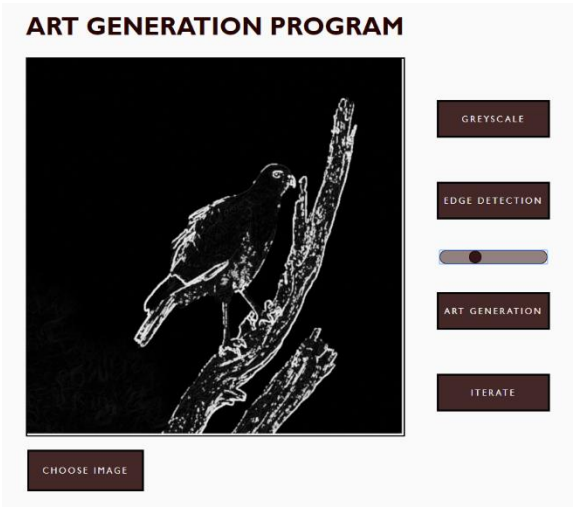


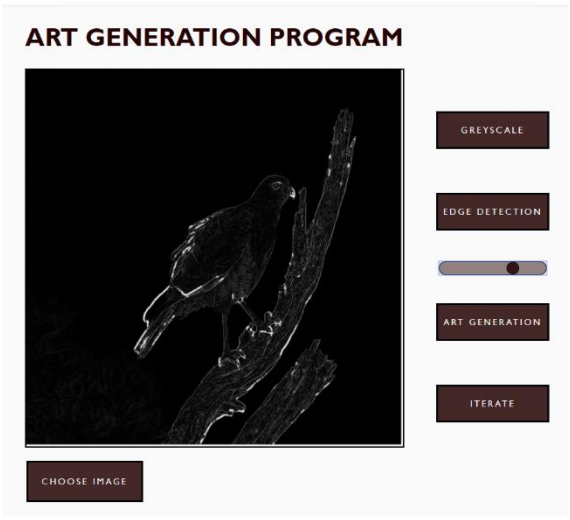
This sequence of screenshots shows how changing the number of indexes skipped from 15, to 45 to 75 increased the sparseness of the overlayed circles. The number of indexes jumped had to be a multiple of 3 as each pixel took up 3 indexes for its RGB values and therefore it was crucial that the entirety of a pixel's data was skipped.



Whilst deciding what artistic direction to take the program in next, I spent some time designing the user interface so that it was more interactive and aesthetic for the user to engage with.

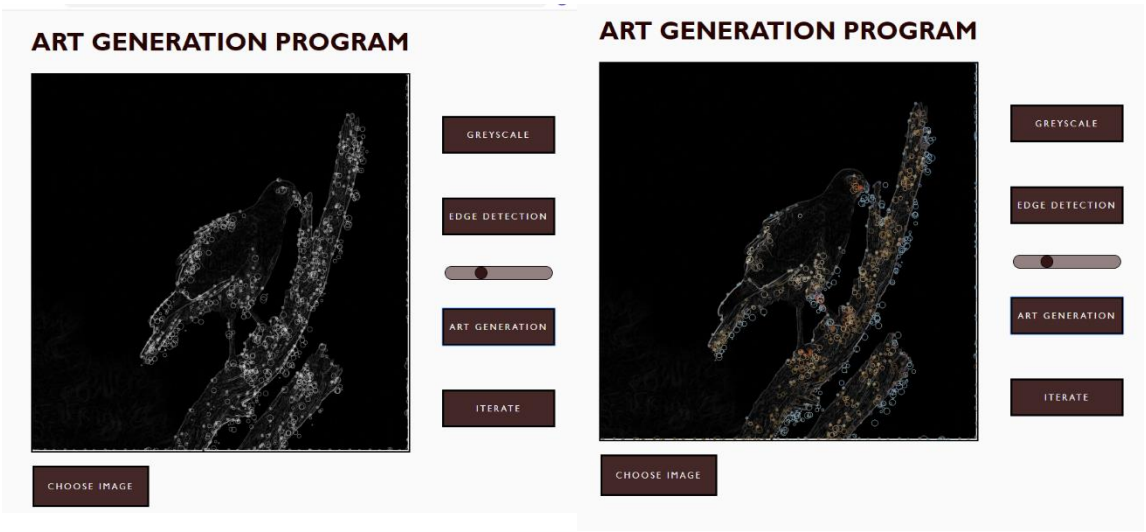
The slider would be an input form that I would use to allow the user to adjust the number of pixels to have as starting points for the art generation. This would mean that the user would use the slider to set the threshold for an edge to be considered 'strong'.



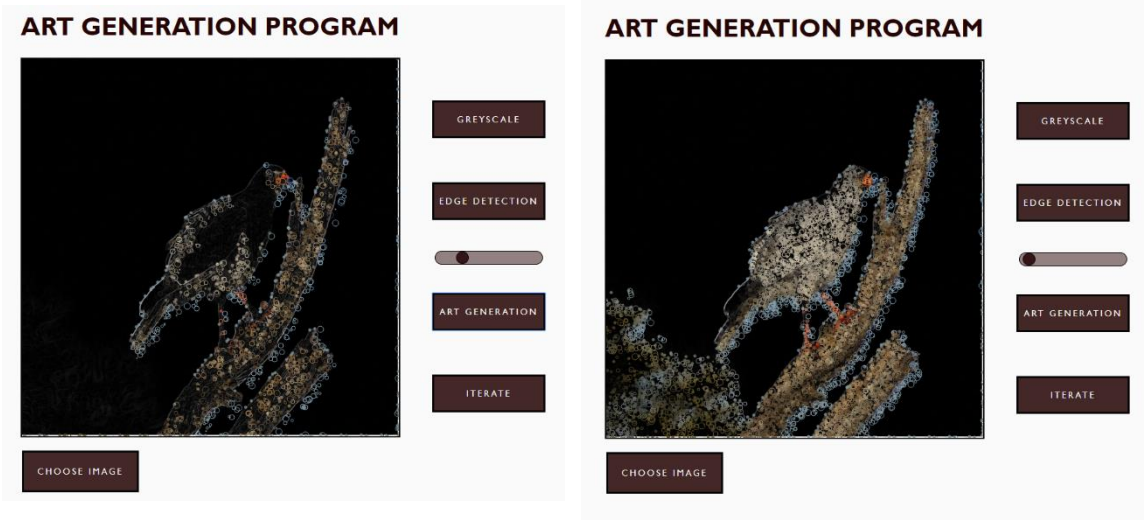


These screenshots show how adjusting the slider changed the number of pixels 'selected' to use as 'strong edges'. To show the user how many edges they had selected I drew a 1px circle over each circle with an edge over the threshold described by the slider value.

To progress the art generation further I wanted to include some variations in colour across the canvas. I decided that each circle placed should take the colour of the pixel on the original image. This would mean that the art generated would resemble the colour palette of the original image, ensuring that no matter how abstract the shapes generated were, the image would always be slightly recognisable. To do this I used the same structure to access a specific pixel as I had used in the Sobel Operation function, yet as the canvas had been painted over with the greyscale image and then the edge detection I had no way of accessing the original image's pixel data. Therefore, I had to create a class in the main 'init' routine of my code, which consisted of arrays representing the greyscale, edge detection, selected 'strong' edges and art generation canvas data. By creating an instance of the class and writing the pixel data array of each canvas state to this object I could then access all the previous canvas data arrays globally, allowing me to access the original pixel data of the image loaded onto the canvas from within another function. After setting up this class I could then pass the pixel data array into my function as a parameter and find the red, green and blue values for any pixel, using the row and column to find it in the array.



The output of my program looked a lot more interesting when each circle was taking the colour of the initial image, rather than limiting it to a simple black and white output



By experimenting with the slider, I found that if the edge strength threshold was at its maximum the program could output an image that very closely resembled the original's shape and colour

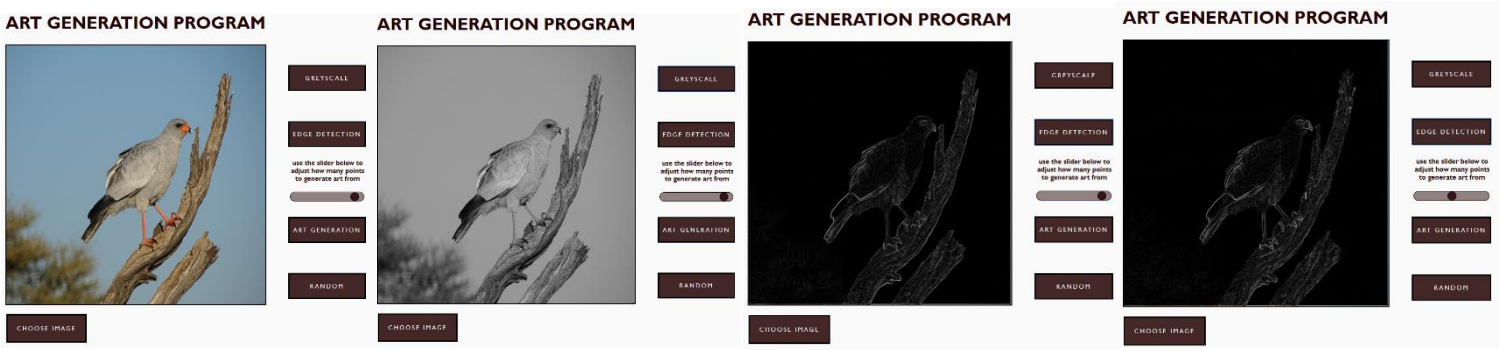
Initially there was a slight error in the implementation of the slider as the values seemed to be inverted, with the edge strength threshold increasing as you moved the slider further down the bar. This orientation seemed counter-intuitive and confusing for the user, yet by styling the slider's direction as right to left this problem was easily fixed.

Testing

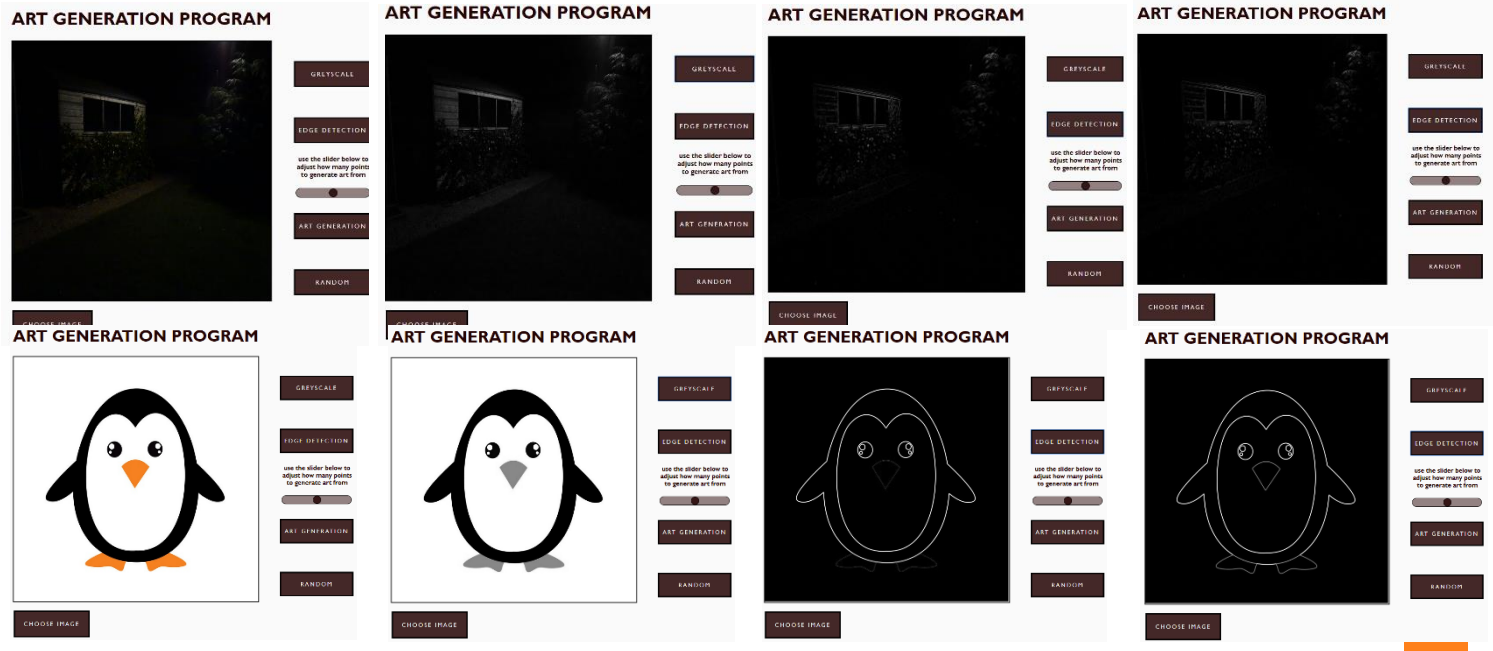
As the output of my program was visual and generated after a series of algorithms, some of which had predictable outputs and others which used a degree of randomness that meant the outcome could not be predicted, it was difficult for me to test inputs against an 'expected output' as to some extent I could not know what to 'expect' from the art generator.

However as shown from the screenshots I had taken during the Technical Solution section, I could test the greyscale and edge detection algorithms as these had no randomness involved, meaning there should be a certain output for each input image. Therefore, I could test that these algorithms produced the expected results for the grey scaling and edge detection algorithms. The following screenshots show my testing of the greyscale and edge detection algorithms. As the edge detection algorithm used the current canvas data as a parameter no matter what the canvas' state, the edge detection still worked even when the image had not been converted to greyscale, it just led to a slightly less accurate and strong edge detection (e.g. the penguin in the third set of photos has not had the edges on the feet so well detected when the image was not greyscaled first).

INPUT IMAGE | **GREYSCALE** | **EDGE DETECTION with greyscale** | **EDGE DETECTION no greyscale**



As expected, if the input image is less clear with less edges (such as the dark and less defined image below) then the edge detection output is likely to be very minimal as edges cannot be easily identified. However if the image is bright and clear, with harsh edges or outlines (such as the penguin image) then the edge detection will be crisper and with brighter lines representing the edges, resulting in a better and more usable output for the art generation algorithm.



The most obvious erroneous input for my program would be if a user tried to select a file that was not an image type, such as a text file. However, because the file picker I used specifies that only image files of JPEG or PNG can be selected, the user would be unable to select any 'non-image' files as the file picker pop-up would only show acceptable image files, meaning I would not need to handle any error messages of a user inputting an invalid file to be loaded onto the canvas.

Evaluation

Overall, my end program meets the general objective of my project, I have produced a web-based application that takes an image inputted by the user, loads it onto a canvas and outputs an artistic representation of this image, using a combination of greyscale, edge detection and shape generation algorithms.

More specifically, my program meets all of Objectives 1-16, with the webpage structure functioning exactly as intended, with an interactive webpage hosting an HTML canvas, with JavaScript code being able to manipulate and control the activity on this canvas. This canvas updated in real-time, showing the user the effects of the program on their input image, as the art generation algorithms were performed on the canvas.

I also believe that the user interface was accessible and easy for the user to navigate, with a simple and clean design that did not distract from the program's main purpose. However, I do think that the webpage could have benefitted from further design, with a welcome page to introduce the program and explain in more depth the purpose of the program and how to use it. This improvement would allow for greater accessibility as currently the control panel lacks a degree of explanation that would benefit users who are less familiar with the concepts of grey scaling and edge detection, and the addition of text alongside the buttons to explain their function and subsequent result would give the user a greater understanding of the program and how it works.

The greyscale and edge detection algorithms both work as expected, with each changing the canvas image data to display their effect on the canvas and updating the 'CanvasData'class with the canvas information at each stage, ensuring I could access each array of pixel data at any time.

As the art generation algorithm developed throughout the coding process, I knew that some of my initial objectives would change or become redundant. However, I met most of the key Art Generation objectives numbered 17-22, with the exception of Number 20 (Circle Opacity dependent on strength of edge). This was an effect that, if I had time to implement, would have created an output with more tonal variation, as currently all generated circles had a constant opacity. An opacity based on edge strength would not only produce an image more similar to the original, with stronger edges being more opaque and therefore more noticeable, but this may have also resulted in a final image with more depth, with the layering of circles more prominent, therefore making the canvas appear more full with more dimension.

Unfortunately, I was unable to implement the User Options Objectives (numbered 23-29). Some, such as the Fading option, had become unsuitable for the program I had since programmed, because the image was entirely made up of circles it would not be effective to have an option where each circle placed was more transparent than the last, as very quickly the circles would have an opacity of 0, meaning that hardly any of the image would actually be generated onto the canvas. However, there were many options that I would have liked to have introduced, including the option to choose a different shape to build the art generation with, allowing for not only circles, but also triangles, squares and many more. Additionally, I would have expanded the colour options available to the user, so that each circle could be drawn onto the canvas with any random colour, a colour of the user's choice, the inverted version of the pixel's original colour or one colour from a three colour palette selected by the user. It would also have been useful to allow the user to download a copy of the final output, this would have made the program more suitable for users who wished to save or obtain a copy of the final creation.

In conclusion I believe that my program generates a unique and colourful artistic representation of an image and allows for sufficient user interaction to specify how abstract or realistic the final result should be, although this could have been extended to allow the user to control more aspects of the art generation.

Bibliography

1. Yuri Vishnevsky : 'Silk Interactive Art' http://weavesilk.com/ (accessed 20/06/19)
2. Daniel Shiffman : 'Chapter 8. Fractals' https://natureofcode.com/book/chapter-8-fractals/ (accessed 08/07/19)
3. CDV Lab : 'Photogrowth : Ant Painting' https://cdv.dei.uc.pt/photogrowth-ant-painting/ (accessed 12/07/19)
4. CDV Lab, Penousal Machado, Tiago Martins, Hugo Amaro and Luís Pereira: 'Photogrowth 2012 Conference Paper) https://www.researchgate.net/publication/232590126_Photogrowth_Non-Photorealistic_Renderings_Through_Ant_Paintings (accessed 12/07/19)
5. Ramesh Jain, Rangachar Kasturi, Brian G. Schunck : 'Machine Vision Chapter 5 – Edge Detection' https://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter5.pdf (accessed 20/07/19)
6. Experimental Media Research Group : 'NodeBox Gallery' https://www.nodebox.net/code/index.php/Gallery.html (accessed 13/07/19)
7. NodeBox : Tom De Smedt (artist), algorithm by ART+COM : 'Tendrils' https://www.nodebox.net/code/index.php/Tendrils (accessed 13/07/19)
8. Mozilla Developer Network : 'Pixel Manipulation With Canvas' ImageData object https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas (accessed 28/07/19)
9. Mozilla Developer Network : 'Canvas Rendering Context 2D' putImageData https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/putImageData (accessed 29/07/19)
10. Mozilla Developer Network : 'Canvas Rendering Context 2D' getImageData https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getImageData (accessed 29/07/19)
11. Medium : Understanding Edge Detection (Sobel Operator) https://medium.com/datadriveninvestor/understanding-edge-detection-sobel-operator-2aada303b900 (accessed 16/08/19)
12. Computerphile : 'Finding the Edges (Sobel Operation) https://www.youtube.com/watch?v=uihBwtPIBxM (accessed 20/08/19)