Investigation into neural networks - Car driving around a randomly generated track

AWAB AKRAM - S201322
ROCHDALE SIXTH FORM COLLEGE

Contents:

Analysis:

Program Concept:

The aim of my project is to make a car go around a randomly generated track using a Neural network.

Very simply, Neural networks are designed to model the human brain. They take in input data, process it, and output answer which they then compare to the correct answer and tweak how they process the data accordingly, in order to get closer to the correct answer. They're very good at spotting and adapting to patterns which makes them commonly used in problems like "the traveling salesman problem" or in spam detection filters or stock market predictors. My aim is to get a neural network to learn to drive a car without crashing into the walls.

Interested Parties:

The people that are interested in the development of my project include some of my peers in my computer science course at college. This project will not only allow me to investigate the concept of neural networks, but it also provides others with some insight into how they work buy visualizing it into the example of a car. For this reason, it is important that the cars are clearly recognizable, as well as the surroundings of the car being easily distinguishable so that people are able to more easily understand what is going on.

In addition, it is important that I implement features in a way that doesn't require previous knowledge of the code to use. For example, instead of pressing G to generate a track, I should have a clickable button labelled "Generate Track". It also should be noted that even though these buttons are important, they should not be implemented in a way that takes away from the investigation, buttons should never overlap with the track/car.

Background research:

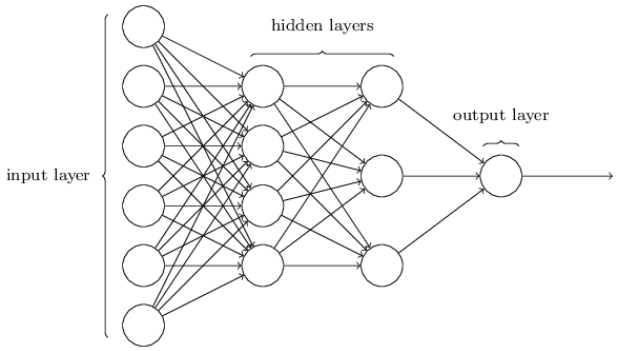
How Neural Networks Work:

Neural networks are made up of individual neurons that are arranged in layers, where each neuron in one layer is connected to at least one or more neuron in the next layer.

There are 3 main types of layers:

- The input layer - where data goes into the neural network, these neurons are the neurons right at the beginning of a network
- the hidden layers – These are the layers in between the input and output layers
- the output layer - where the end result is given out of the network, these are the neurons right at the end of a network

Every layer is essentially a function in itself which narrows down the characteristics of an input and creates a possible output based on that data.



Weights are assigned to every connection between neurons. These weights tell us how important the individual inputs into neurons are towards the output of the neurons. With these weights, we can calculate the "weighted sum" of all the activations - $\text{weight}_1 \times \text{activation}_1 + \text{weight}_2 \times \text{activation}_2 + \ldots + \text{weight}_n \times \text{activation}_n$. This can be written as $\sum_n w_n a_n$ which can also be simplified to $w \cdot a$ (the dot product of w and a)

We also add in a bias to the weighted sum ($w \cdot a + b$) which tells us how high or low the weighted sum has to be for the neuron in the second layer to be active. The higher the bias is, the easier it is for the neuron to be active.

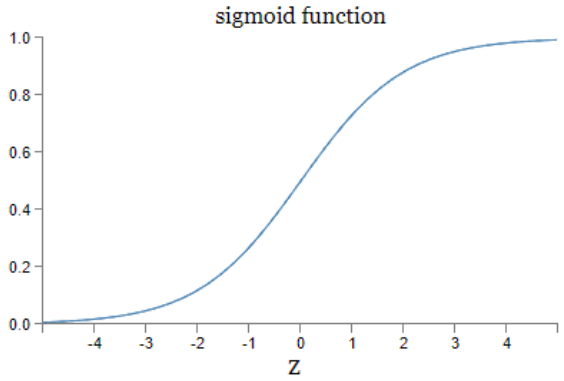
Sigmoid function

This weighted sum + bias ($w \cdot a + b$) could give you any number, large or small so we use a function in order to compress it down into a set area. For example: the sigmoid function (σ) which takes in an input and outputs a number between 0 and 1, making the output of each neuron $\sigma(w \cdot a + b)$.

Equation of sigmoid function:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Graph of sigmoid function:



The smoothness of the sigmoid function means that small changes in the weights and bias will produce a small change in the output from the neuron.

When $w \cdot a + b$ is a large positive number the output from the sigmoid function ≈ 1.
When $w \cdot a + b$ is a large negative number the output from the sigmoid function ≈ 0.

In this way, the sigmoid function relates very close to a binary output (0 for off or 1 for on). However, when $w \cdot a+b$ is a moderate number, then the output of $\sigma(w \cdot a+b)$ gives a number between 0 and 1.

Feed Forward

The process of calculating a weighted sum, adding a bias and then using a compression function is used to find the activation of the neuron in the subsequent layer. It starts for the 1st neuron in the first hidden layer, and is repeated for every neuron in that hidden layer, and then every hidden layer onwards all the way to the output layer. This process is known as Feed Forward, and is how a neural net turns its inputs into outputs.

Training a neural net - Backpropagation

To train a network, Inputs are fed into the neural net and depending on the activations, weights and biases in one layer, neurons are activated in subsequent layers. Eventually the neurons at the end are activated. The given output is compared to what we desired to be the output using what we call a cost function. This cost function gives the network feedback on whether the output it gave was good (close to the right answer, cost sum is small) or bad (cost sum is large, far from the right answer). You do this by adding up the squares of the differences between the given outputs and the outputs you expected:

Where n = number of output neurons:

$$(\text{GivenOutput}_1 - \text{ExpectedOutput}_1)^2 +$$
$$(\text{GivenOutput}_2 - \text{ExpectedOutput}_2)^2 +$$
$$(\text{GivenOutput}_3 - \text{ExpectedOutput}_3)^2 +$$
$$\dots$$
$$+ (\text{GivenOutput}_n - \text{ExpectedOutput}_n)^2$$

At the end, you average the cost over all of your training data (Or subsets of your data called Mini-Batches) which is a measure for how good or bad the network is.
An additional use for the cost function lies in the fact that the magnitudes of the costs can somewhat tell you which neurons are most important to change. i.e., Which changes can you make that will have the biggest difference on the output.
So, we can then look back at the previous neurons and change their weights and biases in order to get a more desirable output. This process is repeated with more training data until the desired output is received or the user is satisfied with the results.

Training a Neural Network – Genetic Algorithm

Since backpropagation requires a desired output, i.e. a car that drives around that specific track in the fastest way possible, using a genetic algorithm may be a better method to train the neural net in this instance.

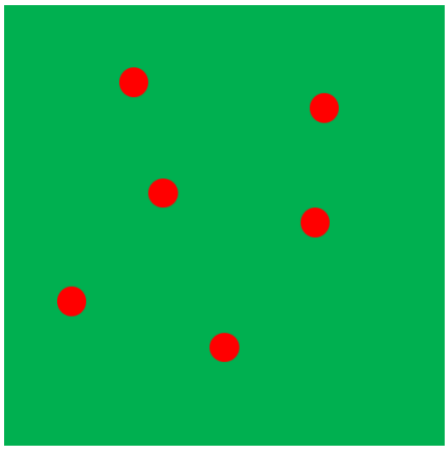
A genetic algorithm works by taking the idea of Darwinian natural selection. You start with an initial population of neural nets where the weights and biases are set randomly. They then work and you find which Networks are the “fittest” (this is usually done by calculating a ‘fitness’). These fittest networks are then taken and mutated into a new generation in hopes that they will evolve to be fitter than the previous generation. There is a chance that they evolve to be less fit than the previous generation, in which case they die out and the previous generation is the generation that carries on. This process is

repeated numerous times until the neural networks act in the way that we want and cannot improve anymore/further improvements are minimal or insignificant.

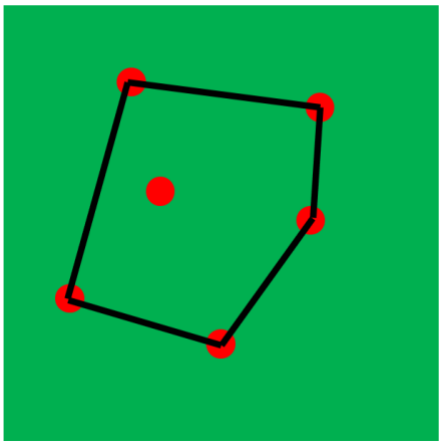
Randomly Generated track:

The idea for the randomly generated track is to start with a 2D plane like this:

Then generate at least 3 random pairs of coordinates which would be mapped to the plane:



Find the Convex Hull of the points using a Graham Scan Algorithm:

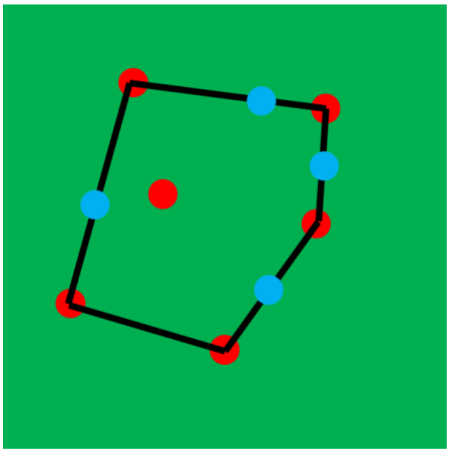


Steps to find the Convex Hull:

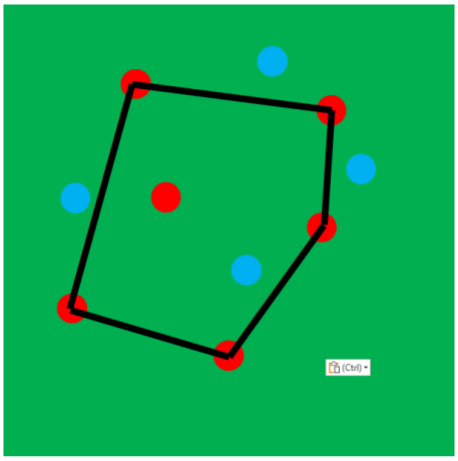
1. Find the Lowest point (point0), in this case it would be the point with the greatest Y-value
2. Find the polar angle of all of the other points in relation to point0, this can be done using trigonometry ($\tan^{-1}$(opposite/adjacent) then + π if the point's x value is less than point0's x value)
3. Find distances between all other points and point0, this can be done with Pythagoras' Theorem
4. Sort the points with point0 first and then the rest of the points sorted by increasing polar angle
5. If two points have the same polar angle, remove the closest point
6. Graham Scan:
    1. Create a new stack
    2. Push points 0, 1 and 2 onto the stack
    3. Iterate through every point and check if it's a left turn or a right turn from the previous points, every time it is a right turn, you pop until it is a left turn again, then push the current point onto the stack

Then Curve the Lines of the track:

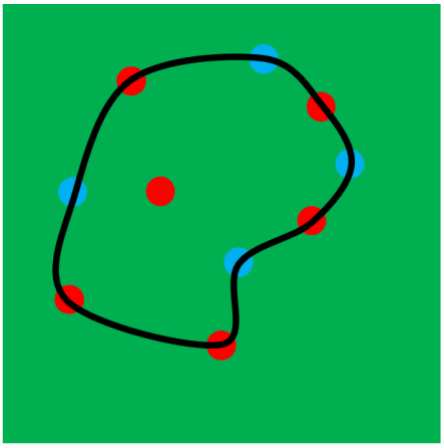
First find random points along the lines



Then offset them by a random value:

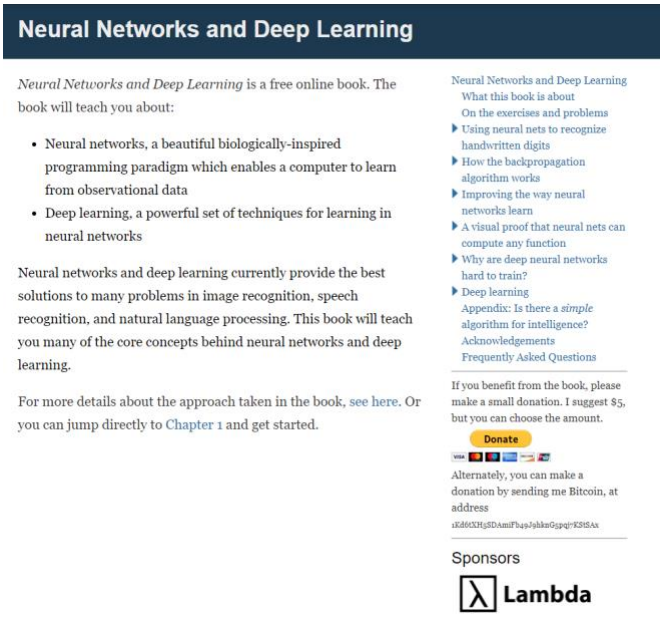


Then Curve the lines about the points using a Bézier curve:



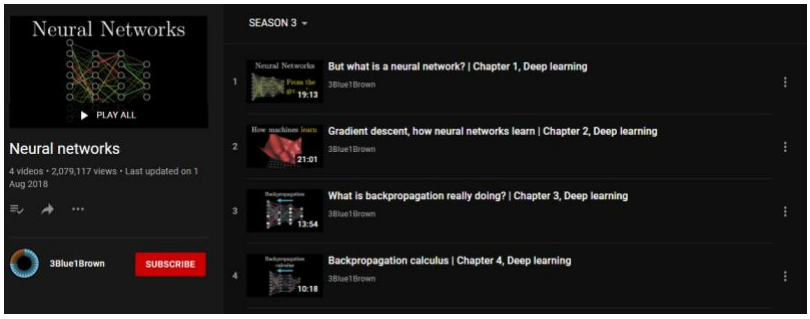
Resources:

http://neuralnetworksanddeeplearning.com/ - A free online book by Michael Nielsen:



3Blue1Brown Videos on YouTube:
https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

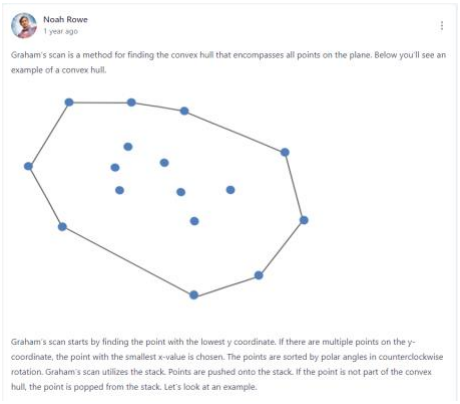
- "But what is a neural network? | Chapter 1, Deep learning" - https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2
- "Gradient descent, how neural networks learn | Chapter 2, Deep learning" - https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2
- "What is backpropagation really doing? | Chapter 3, Deep learning" - https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3
- "Backpropagation calculus | Chapter 4, Deep learning" - https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4



Blog Post by Kip Parker - Building a neural network in C#:
Building a neural network in C#. Creating a neural network with the… | by Kip Parker | Towards Data Science

Blog Post by Vijini Mallawaarachchi - Introduction to Genetic Algorithms — Including Example Code:
https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[Graham's Scan Visually Explained (morioh.com)]:

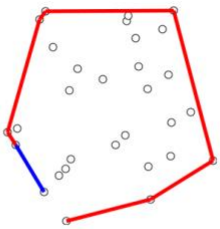


Noah Rowe
1 year ago

Graham's scan is a method for finding the convex hull that encompasses all points on the plane. Below you'll see an example of a convex hull.

Graham's scan starts by finding the point with the lowest y coordinate. If there are multiple points on the y-coordinate, the point with the smallest x-value is chosen. The points are sorted by polar angles in counterclockwise rotation. Graham's scan utilizes the stack. Points are pushed onto the stack. If the point is not part of the convex hull, the point is popped from the stack. Let's look at an example.

[Graham Scan Algorithm to find Convex Hull (opengenus.org)]:



Algorithm

In Jarvis's Algorithm for Convex Hull. Worst case time complexity of Jarvis's Algorithm is O(n^2). Using Graham's scan algorithm, we can find Convex Hull in O(nLogn) time. Following is Graham's algorithm

Let points[0..n-1] be the input array.

1. Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Let the bottom-most point be P0. Put P0 at first position in output hull.
2. Consider the remaining n-1 points and sort them by polor angle in counterclockwise order around points[0]. If polor angle of two points is same, then put the nearest point first.

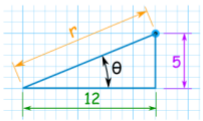
3 After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from P0. Let the size of new array be m.

MathsIsFun - Polar and Cartesian Coordinates:
https://www.mathsisfun.com/polar-cartesian-coordinates.html

To Convert from Cartesian to Polar

When we know a point in Cartesian Coordinates (x,y) and we want it in Polar Coordinates (r,θ) we **solve a right triangle with two known sides.**

Example: What is (12,5) in Polar Coordinates?



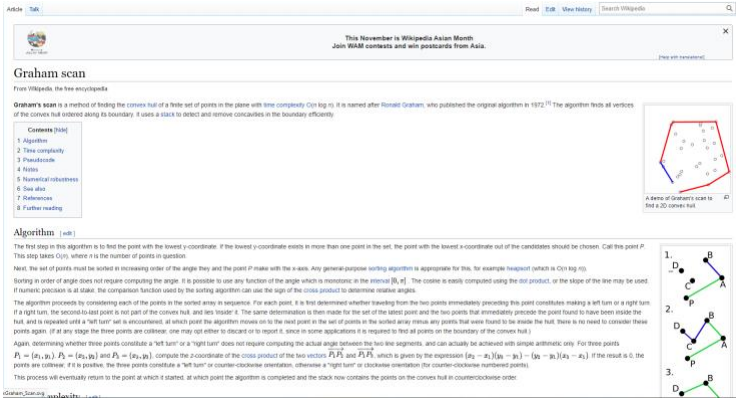
Use Pythagoras Theorem to find the long side (the hypotenuse):

- $r^2 = 12^2 + 5^2$
- $r = \sqrt{(12^2 + 5^2)}$
- $r = \sqrt{(144 + 25)}$
- $r = \sqrt{(169)} = \mathbf{13}$

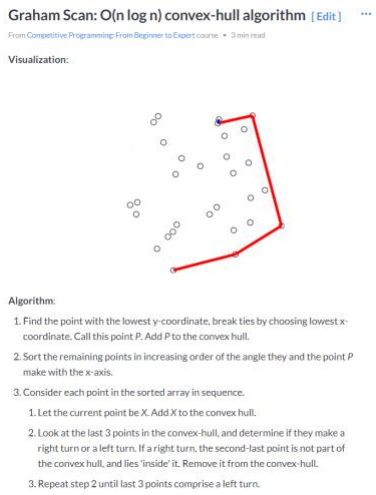
Use the Tangent Function to find the angle:

- $\tan(\theta) = 5 / 12$
- $\theta = \tan^{-1}(5 / 12) = \mathbf{22.6°}$ (to one decimal)

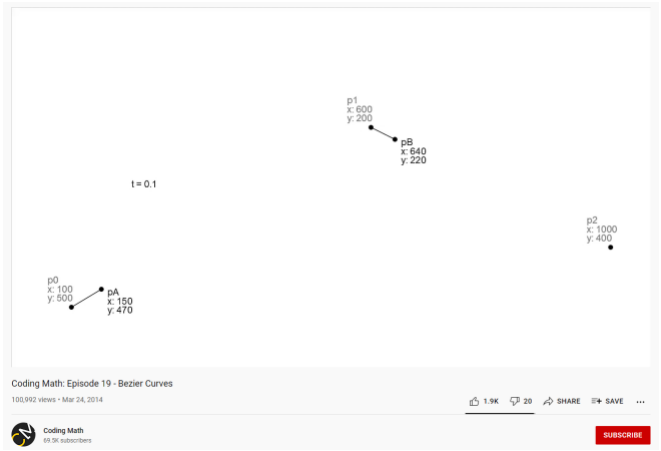
Wikipedia – Graham Scan:
[Graham scan - Wikipedia]



[Graham Scan: O(n log n) convex-hull algorithm | CommonLounge]:



Coding Math: Episode 19 - Bezier Curves:
[https://www.youtube.com/watch?v=dXECQRlmIaE]



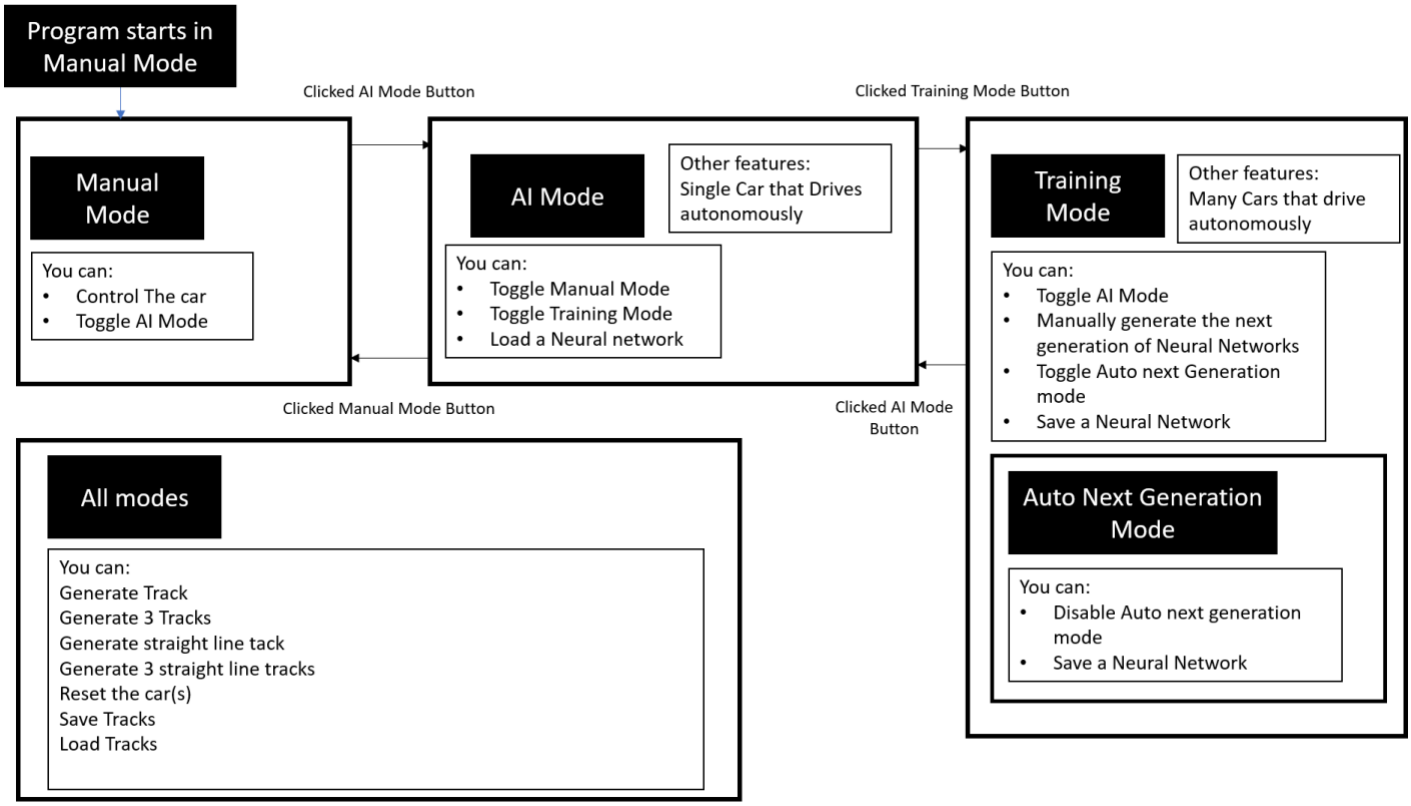
Objectives for the investigation:

1. Randomly generate tracks
    a. Should be able to generate tracks of different sizes
    b. Should be able to generate tracks with different numbers of bends
    c. Should be able to generate tracks with different sharpness of bends
    d. Tracks should have a start line
    e. Tracks should be cyclical (Start and end at the same point)
    f. Implement a Graham Scan Algorithm
        i. Implement a Stack for this algorithm
    g. Implement Bezier Curves
2. Have a button to generate a random track
3. Have a button to generate three tracks at once, which you can then cycle through
4. Be able to save generated tracks
    a. Allow the user to enter a name for the file to save the track to
5. Be able to load generated tracks
    a. Allow the user to enter the name of the file they want to load
6. Create car(s) that drive around the track
    a. Be able to drive this car manually with WASD controls
        i. Drive forward (accelerate) when holding W
        ii. Turn left when holding A
        iii. Reverse when holding S
        iv. Turn right when holding D
    b. Ensure the car slows down when not accelerating
    c. Implement functionality to stop the car once it goes off the track
7. Allow the user to be able to reset the car back to the start line if they want
8. Use Neural Networks to drive the car(s) around a track
    a. Understand how feed forward neural networks function.
    b. Understand how feed forward neural networks are trained.
    c. Implement a feed forward algorithm in order to turn inputs into outputs
    d. Use the outputs from the feed forward algorithm to drive the car
    e. Have a training mode in order to train the neural networks
    f. Implement a way to determine the fitness of the cars
    g. Implement a mutation function for the neural networks
    h. Add a way for the program to automatically move to the next generation
9. Be able to save neural networks
    a. Allow the user to enter a name for the file to save the neural network to
10. Be able to load neural networks
    a. Allow the user to enter the name of the file they want to load
11. Create a graphical user interface to visualise the track, car(s) and Buttons.
    a. Buttons should not overlap the track/car
    b. Should Have a race black race track with a clearly contrasting background.
    c. Car should be clearly distinguishable from the track and background.

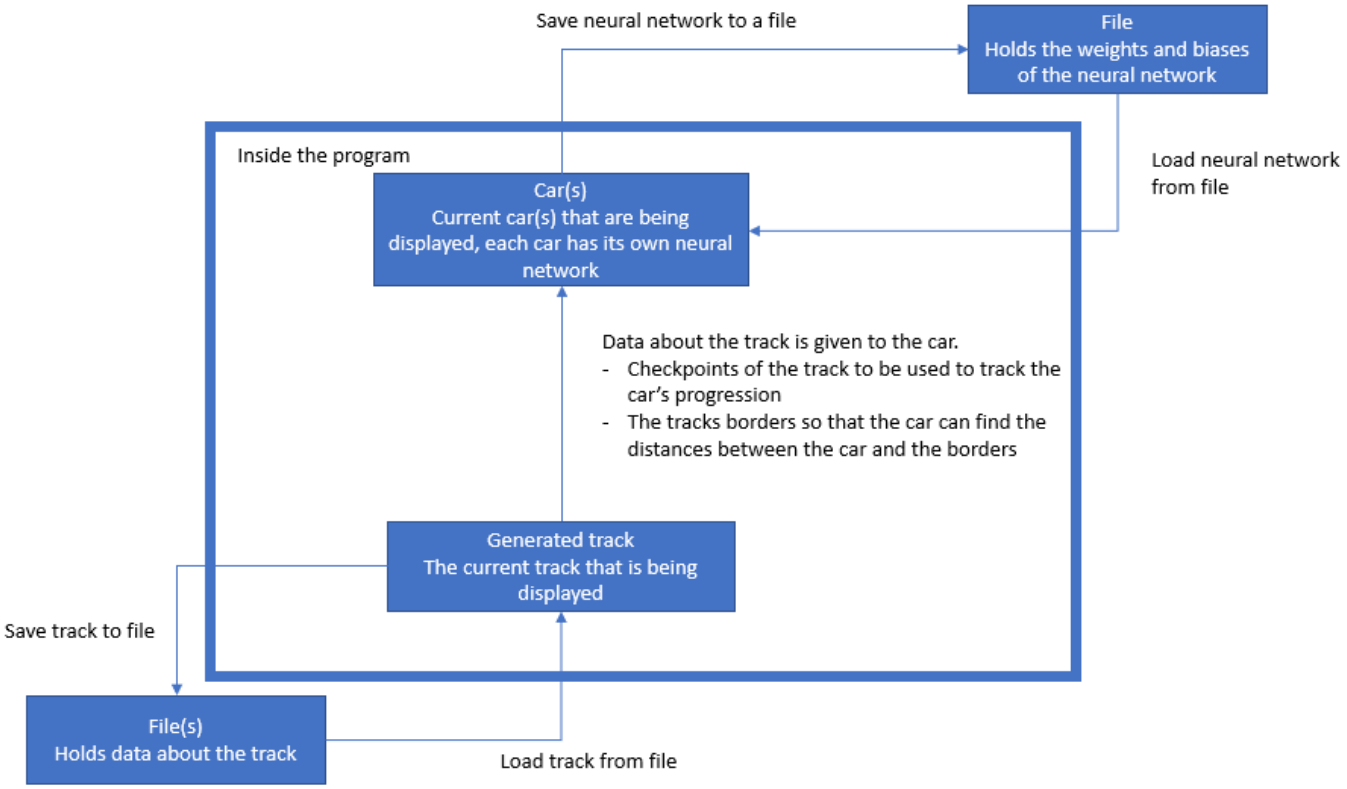
Documented Design:

High Level Diagrams:

The Program will have different modes, which will each have their own purposes:

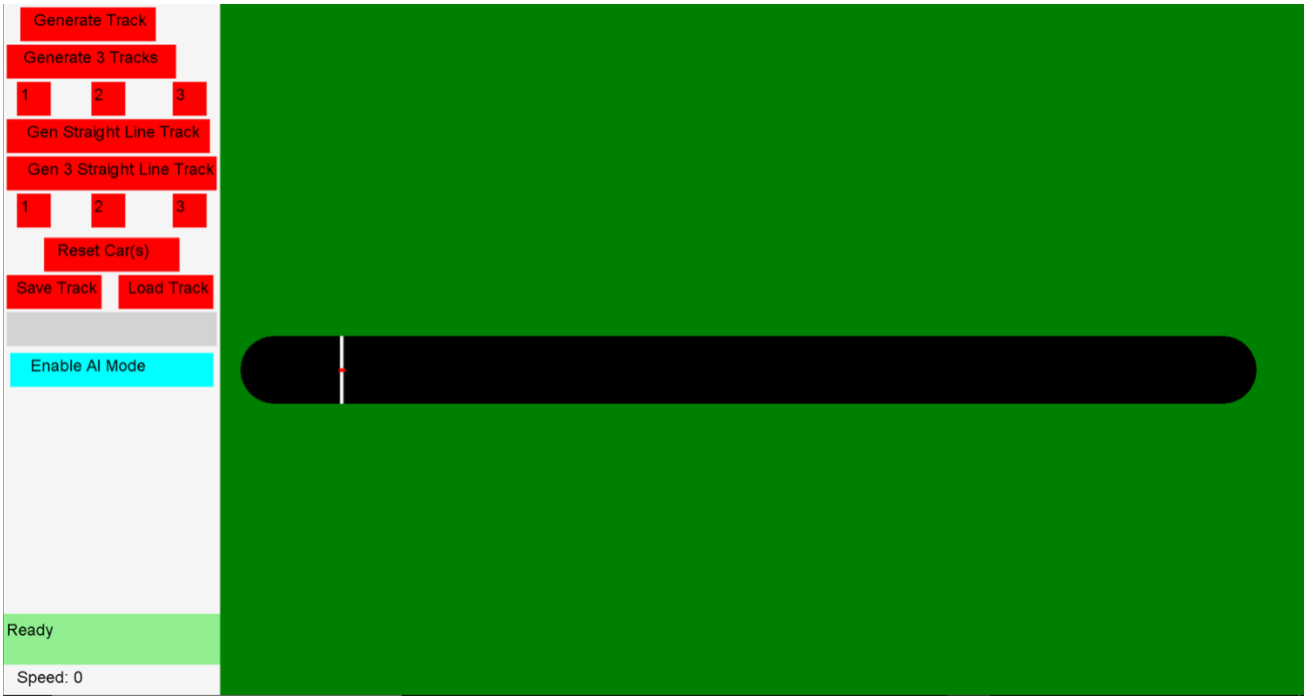


A simple high-level diagram to show data transfer around the program:

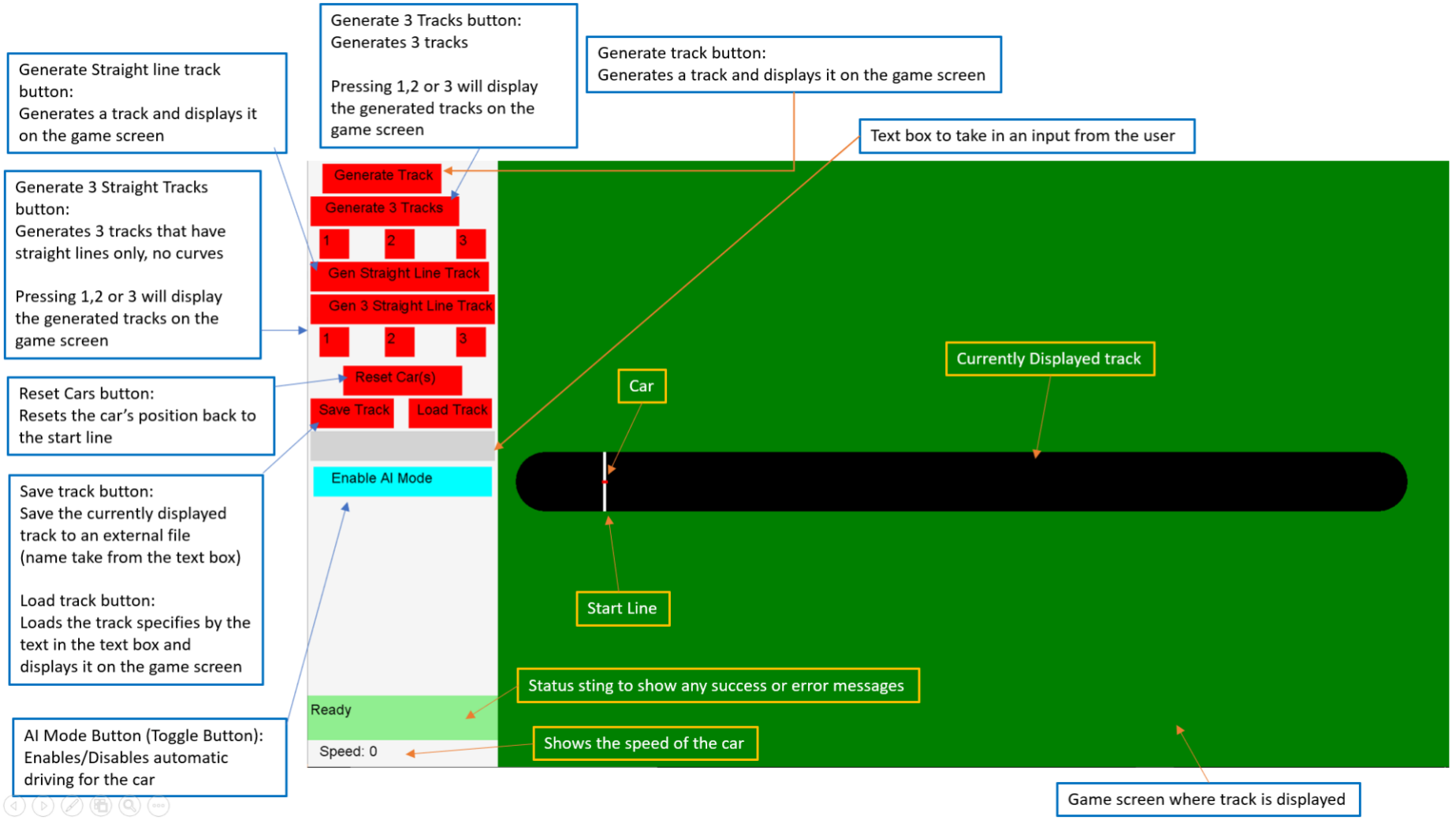


Graphical User Interface:

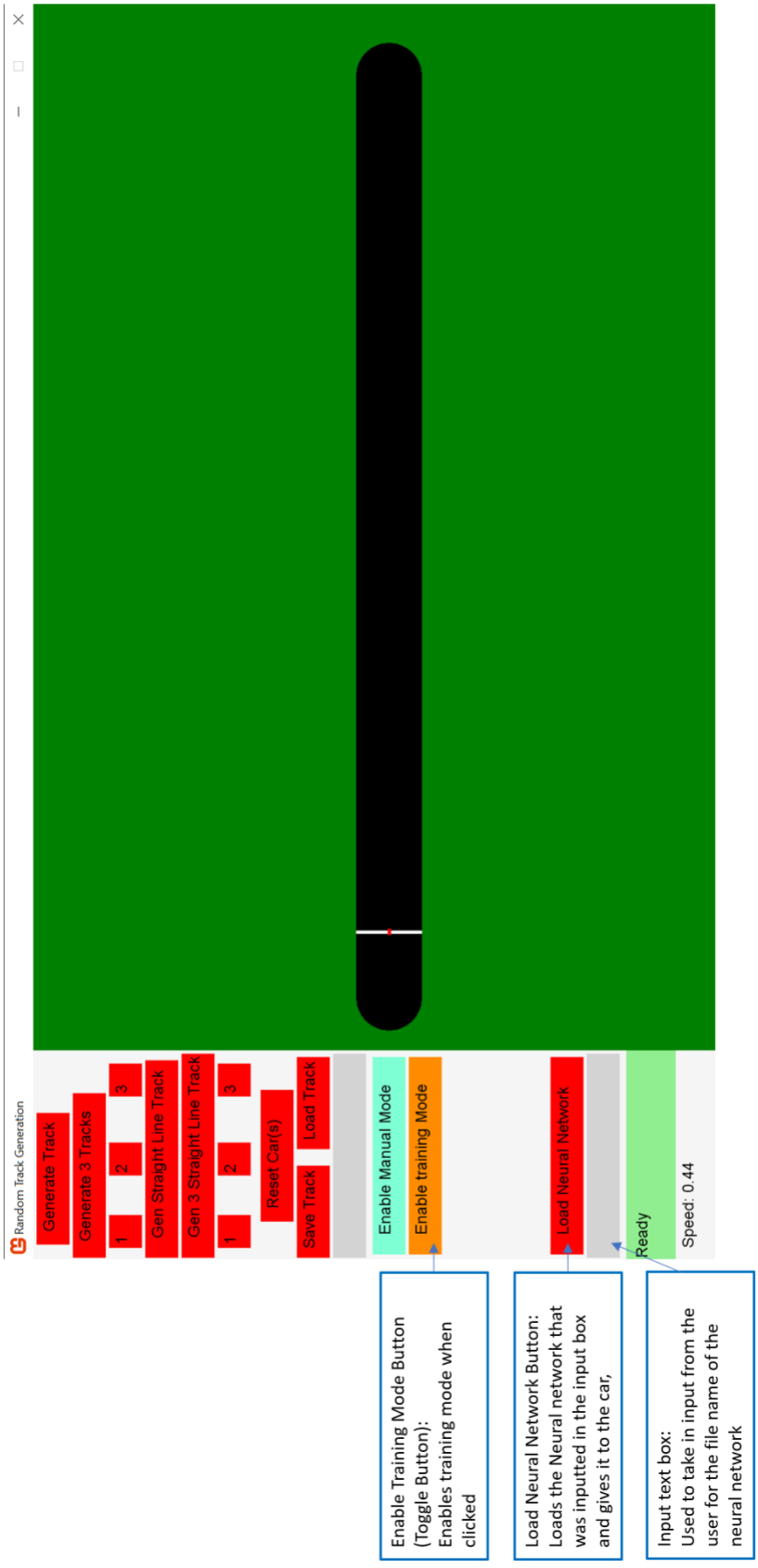
When you start the program, it starts with this screen:



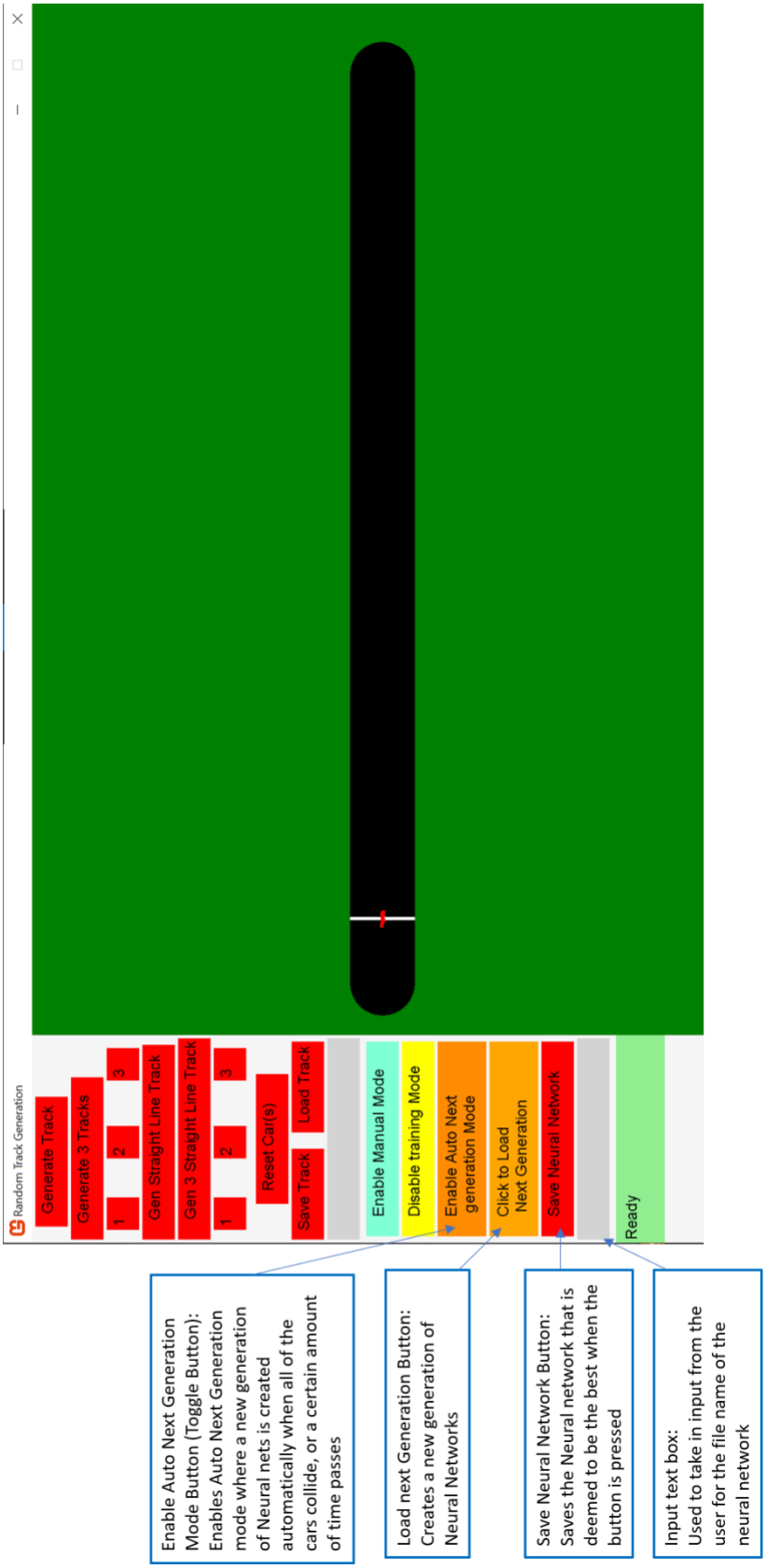
A brief overview of this GUI on the next page:


Generate Straight line track button:
Generates a track and displays it on the game screen
Generate 3 Tracks button:
Generates 3 tracks
Pressing 1,2 or 3 will display the generated tracks on the game screen
Generate track button:
Generates a track and displays it on the game screen
Text box to take in an input from the user
Generate 3 Straight Tracks button:
Generates 3 tracks that have straight lines only, no curves
Pressing 1,2 or 3 will display the generated tracks on the game screen
Generate Track
Generate 3 Tracks
1
2
3
Gen Straight Line Track
Gen 3 Straight Line Track
1
2
3
Reset Car(s)
Save Track
Load Track
Enable AI Mode
Ready
Speed: 0
Reset Cars button:
Resets the car's position back to the start line
Save track button:
Save the currently displayed track to an external file (name take from the text box)
Load track button:
Loads the track specifies by the text in the text box and displays it on the game screen
AI Mode Button (Toggle Button):
Enables/Disables automatic driving for the car
Car
Currently Displayed track
Start Line
Status sting to show any success or error messages
Shows the speed of the car
Game screen where track is displayed

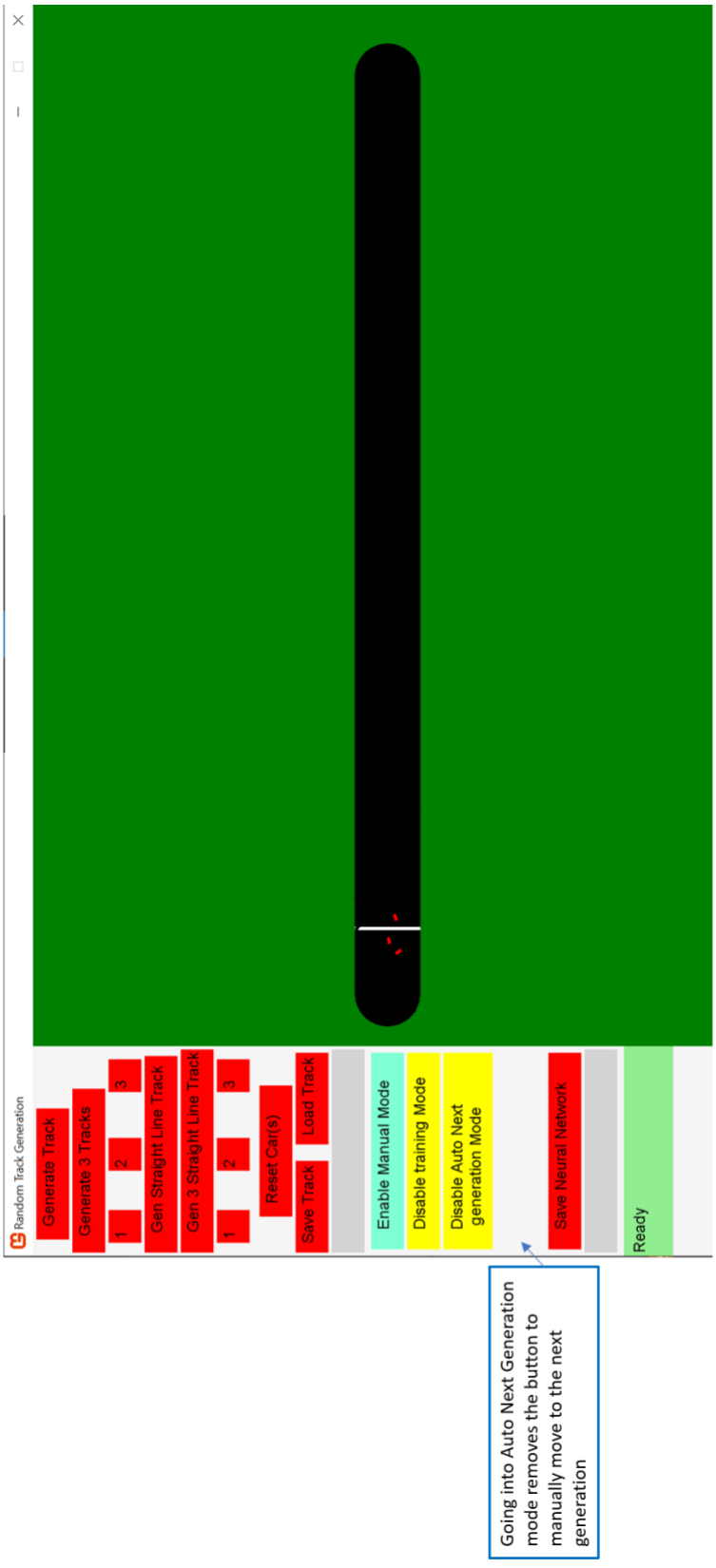
The GUI after clicking the button labelled “Enable AI Mode”:



The GUI after clicking the button labelled “Enable Training Mode”:



The GUI after clicking the button labelled “Enable Auto Next Generation Mode”:



Permanent Data Storage:

Saving the track:

1. Track name is inputted by the user
2. Folder is created with that track name
3. Inside the folder is 2 files, Finalpoints.txt and StartPoints.txt

FinalPoints.txt has all the X and Y values for the track points laid out like this:

Trackpoint[0].X, Trackpoint[0].Y
Trackpoint[1].X, Trackpoint[1].Y
Trackpoint[2].X, Trackpoint[2].Y

…
Trackpoint[n].X, Trackpoint[n]. Y

E.g.

```
1703.7172, 918.40454
1697.8529, 910.19336
1694.5912, 905.6261
1685.7358, 893.22437
1680.6168, 886.05444
1675.3444, 878.66895
1669.6353, 870.67065
1664.6028, 863.6199
1659.8645, 856.98035
1651.2502, 844.90845
1647.4448, 839.57495
```

StartPoints.txt has the X and Y values for the start point of the track as well as the ‘edges’ of the start point which are essentially the two points between which the start line goes.
StartPoints.txt is laid out like this:

StartPoint.X, StartPoint.Y
StartPointEdge1.X, StartPointEdge1.Y
StartPointEdge2.X, StartPointEdge2.Y

E.g.

```
1199, 859.4511
1204.8008, 809.7881
1193.1992, 909.1133
```

Saving the Neural network:

Saving the neural network is done using 1 .txt file which is named based on the users input. The file is laid out like this:

Layers[0], Layers[1], Layers[2], …, Layers[n]
Bias[0]
Bias[1]
Bias[2]
Bias[n]
Weight[0]
Weight{1]
Weight{2]
Weight[n]

E.g.

```
9,7,7,4
0.33216384
0.033031493
-0.35450438
0.37739122
-0.3756681
0.11393764
0.46884522
-0.33090132
0.37752128
0.27091125
-0.4706145
-0.198376
-0.27394864
-0.13333172
0.31359294
0.23986961
0.4271944
0.00025971117
```

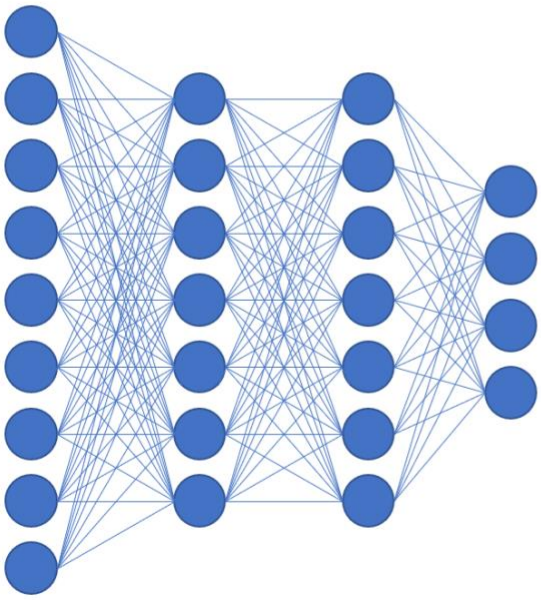
Where Layers is the number of neurons in each layer, in this example, this neural network would have 4 layers (4 numbers in the fist line) with the first layer having 9 neurons, the second layer having 7 neurons, the third layer having 7 neurons and the fourth layer having 4 neurons.

Data Structures:

Neural network:

Layers =
9 input
7 hidden
7 hidden
4 output

Int[] layers = {9, 7,7,4}



Neural network has multiple layers each made of a different number of neurons

Therefore, neurons must be stored as an array of different sized arrays

i.e. a jagged array:
float[][] Neurons

Each neuron has a bias, so biases are stored the same way as neurons:
float[][] Biases

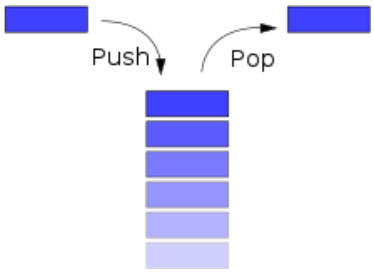
However, Each neuron has multiple weights so we need an array of different sized arrays of different sized arrays:
float[][][] Weights

The inputs to the neural networks will be 8 Distances from the car to the edge of the track, and the car's speed.

Outputs of the neural net will be 4 numbers that are between 0 and 1, the 1st neuron will be for forward movement, 2nd neuron for backwards movement, the 3rd neuron for left movement and the last neuron for right movement.

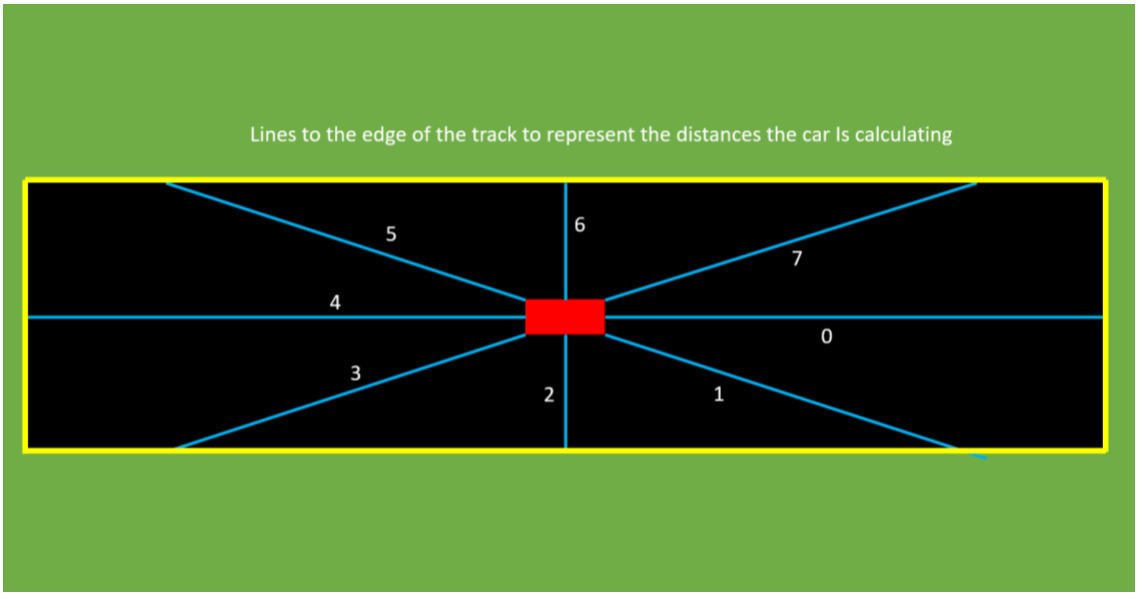
Stack:

My code will also have a stack in it for the graham scan algorithm



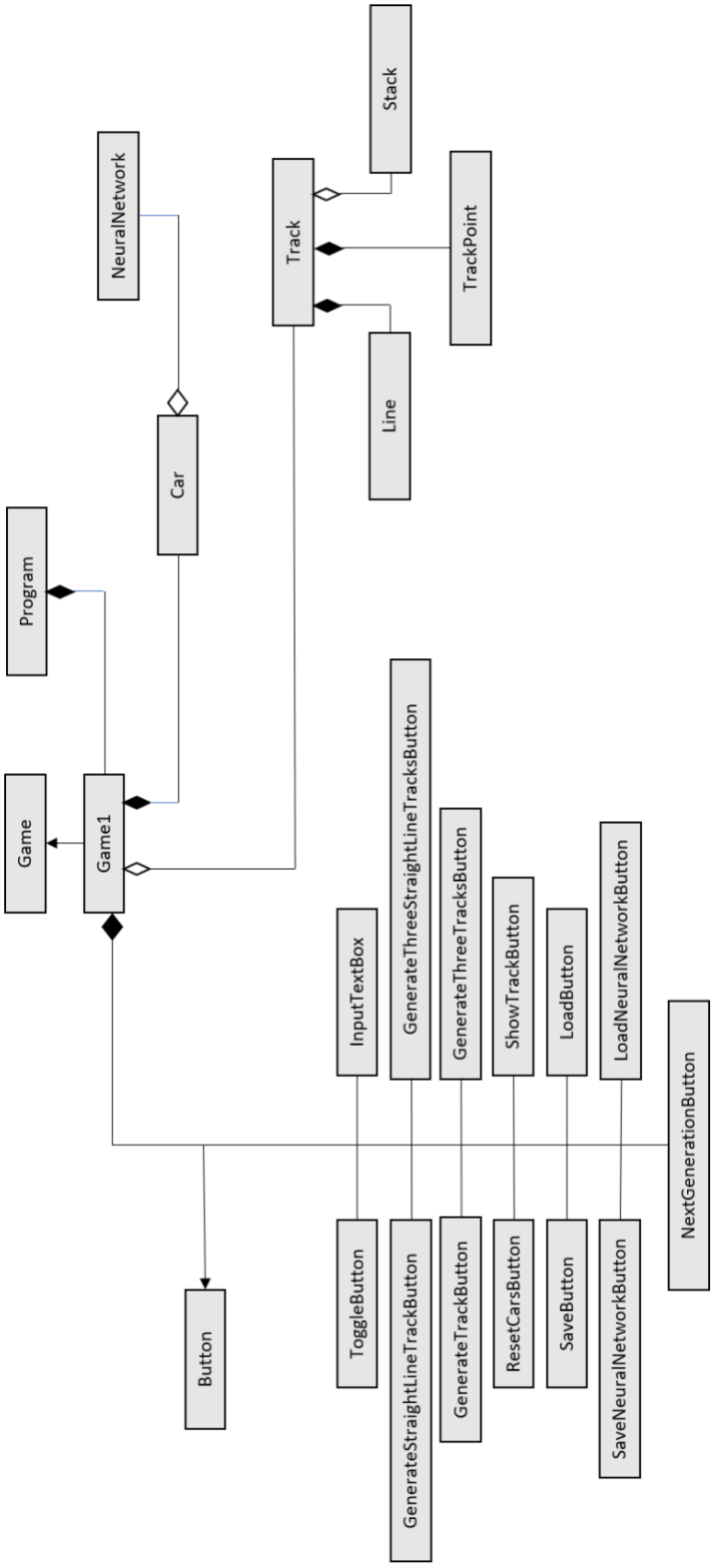
List:

My code will also have lists in it whenever I am unsure of how many items of data I will have in the end, for example in the track generation, the number of points is generated randomly and after the graham scan and use of Bezier curves, it is impossible to know how many points there will be in the end.



Class Association:

Here is a diagram to show the association between all of my classes:



Algorithms:

Generate Track:

Randomly generate a number to pick how many points the track will have

For each point that there is going to be:

    Generate a random X value

    Generate a random Y value

    Pair the X and Y values up

Point0 = point with highest Y value (i.e. the lowest point on the screen)

For each of the points:

    Find the polar angle between point0 and the current point

    Find the distance between point0 and the current point

Order the points by their polar angle

If any of the points have the same polar angle, keep the one with the greatest distance, discard the others

Find the convex hull of the points using a Graham scan

For each of the convex hull points:

    Generate a random coordinate between the point (Point A) and its subsequent point (Point C) (lying on the line between the point and its subsequent point)

    Offset this random coordinate perpendicularly to the original line by some random amount (This gives point B)

    Use the point, the generated coordinate and the subsequent point (Points A, B and C) generate a Bezier Curve and save any points generated

Find the start point.

Find the two points that the start line is drawn in between.

For each line in the track

    Find the two parallel lines to them, offset by the same amount (plus and minus half of the track's width), these are the track's border lines

    Find which lines are on the inside/outside (inside lines will be to the left of the track)

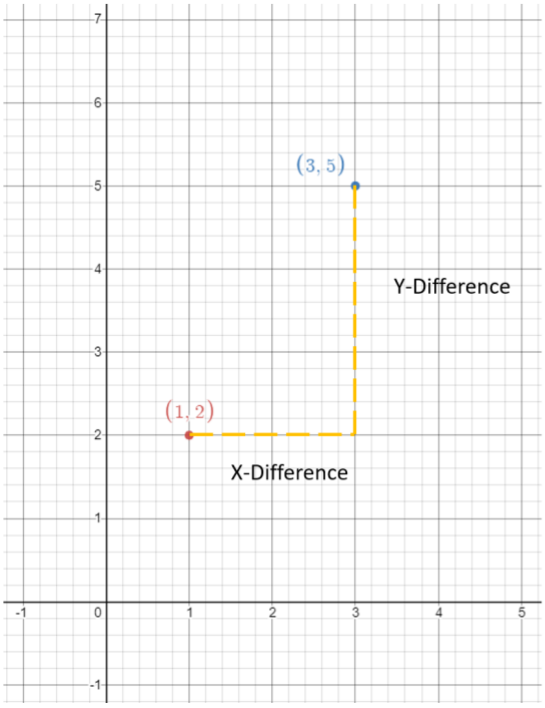
For each of the border lines

    Check if they have overlapped/intersected

    If they have intersected, then change their lengths so that they intersect but don't go further than the point of intersection

    Connect the ends of each of the border lines together so that there are no gaps in the border

Polar angle:

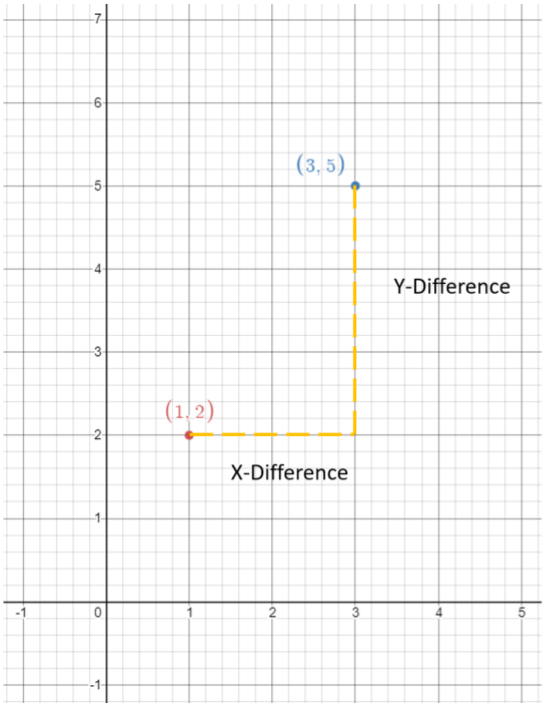


Angle = $\tan^{-1}$(Y-Difference/X-Difference)

If X-difference is negative then:

Angle= $\tan^{-1}$(Y-Difference/X-Difference) + π

Distance:



Pythagoras' Theorem:
$A^2 = b^2 + c^2$

A = distance
B = Y-difference
C = X-difference

So:

Distance = $\sqrt{(\text{Y-Difference}^2 + \text{X-Difference}^2)}$

Ordering – Merge Sort:

Uses Recursion

```
Trackpoint[] merge(Trackpoint[] list1, Trackpoint[] list2)
      Index1 = 0
      Index2 = 0
      indexMerged = 0
      TrackPoint Merged[] = new TrackPoint[list1.length + list2.length]
      While index1 < list1.length AND index2 < list2.Length
            If list1[index1] < list2[index2]
                  Merged[indexMeged] = list1[index1]
                  Index1++
            Else
                  Merged[indexMeged] = list2[index2]
                  Index2++
            indexMerged++

      While index1 < list1.length
            Merged[indexMeged] = list1[index1]
            Index1++
            indexMerged++
      While index2 < list2.length
            Merged[indexMeged] = list2[index2]
            Index2++
            indexMerged++
```

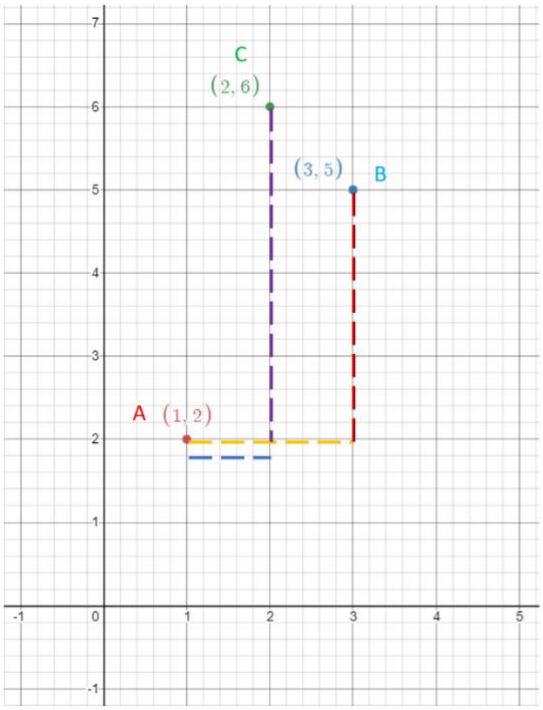
```
Trackpoint[] MergeSort(Trackpoint[] list)
      If list.length < 2
            Return list
      Else
            Midpoint = (list.length – 1)/ 2
            leftHalf = list[0:midpoint]
                  rightHalf = list[midpoint+1:items.Length-1]
            MergeSort(leftHalf)
            MergeSort(rightHalf)

            Return merge(leftHalf, rightHalf)
```

Graham Scan:

```
TrackPoint[] grahamScan(TrackPoint[] Points)
      Stack pointStack = new stack()
      pointStack.push(points[0])
      pointStack.push(points[1])
      pointStack.push(points[2])
      for the remaining points in Points
            While checkLeft(pointStack.SecondToTopPoint, pointStack.TopPoint,
            currentPoint) is false
                  pointStack.Pop
            PointStack.push(currentPoint)
      Return pointStack.toArray
```

Check Left:



Find the Cross Product of AB and AC

result = ((B.X - A.X) * (A.Y - C.Y)) - ((A.Y - B.Y) * (C.X - A.X));

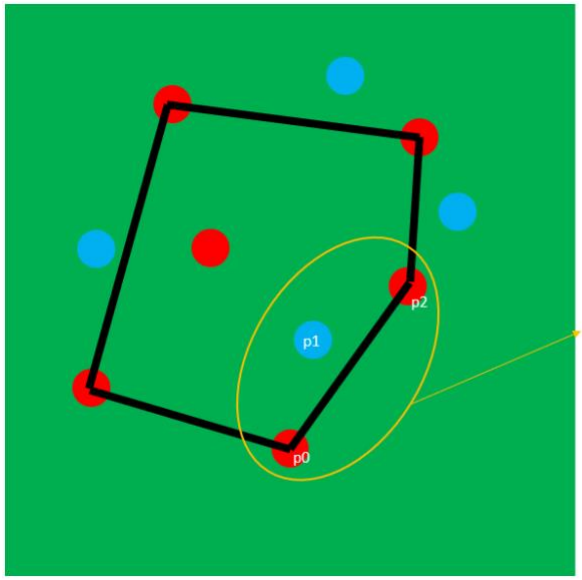
if result < 0 then
return false

Else
Return true

Bezier Curve

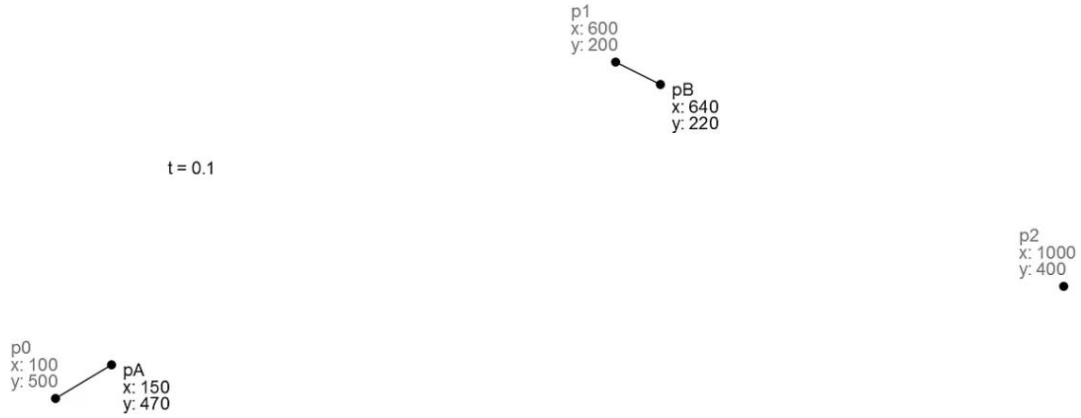
Steps for a Bezier curve:

1. First pick one section of line that you want to curve:

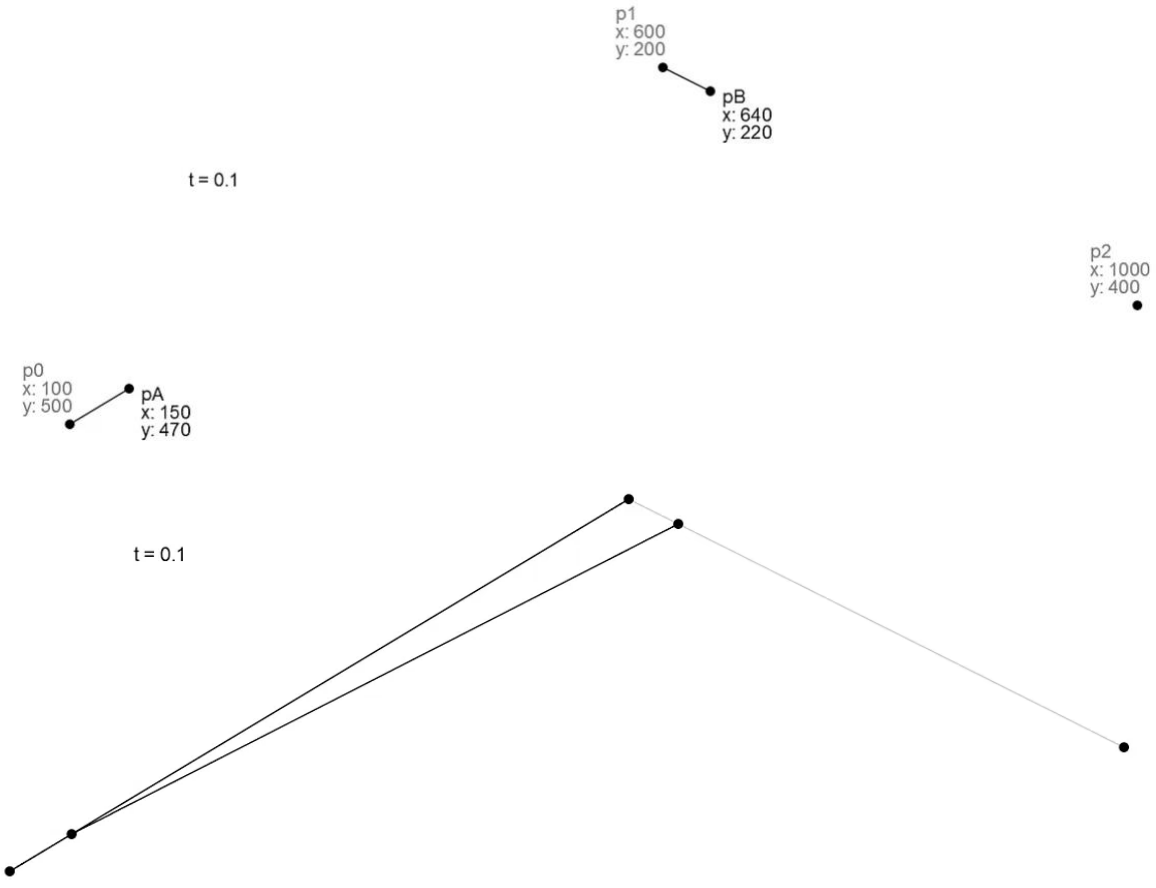


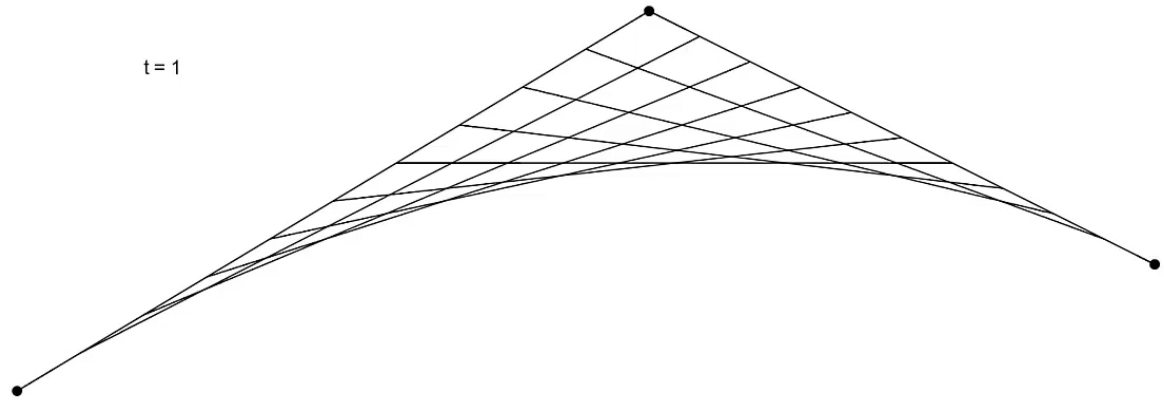
2. Let's call the line p0 -> p1 "Line A" and the line p1 -> p2 "Line B".

3. t is a value that represents how far along a straight line you are.
    a. At t=0 you're at p0 for Line A and you're at p1 for Line B.
    b. At t=1 you're at p1 for Line A and you're at p2 for Line B.
    c. pA and pB are indicators for the position you're at for the current value of t.

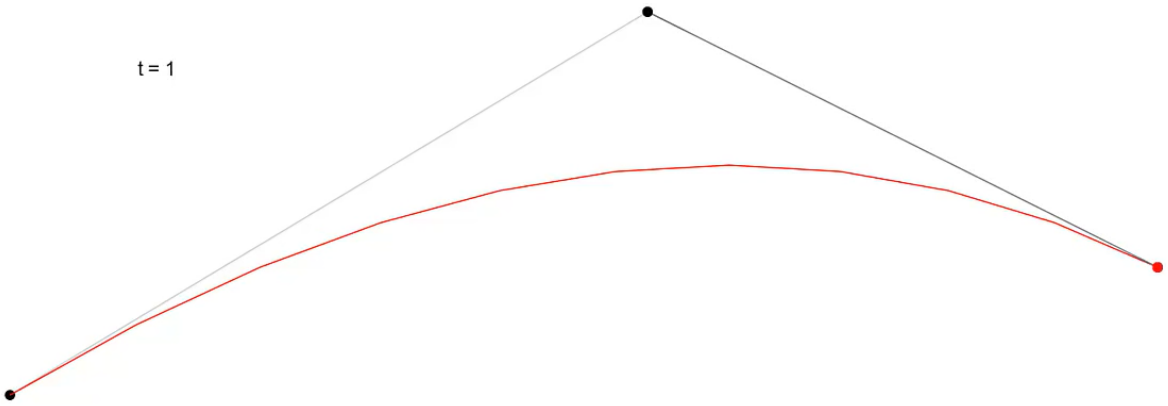


4. To create a curved line between p0 and p2 you have to increase t by regular small intervals and draw lines between pA and pB for each value of t

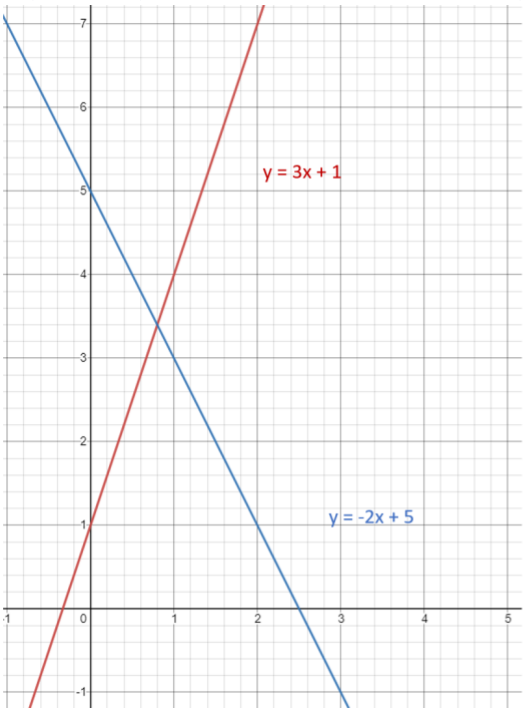




5. In order to get just the curve without all of the extra grid lines, you would have to work out the points of intersection between each line and its proceeding line.



Finding Points of Intersection Between two straight lines:



Given the Equations of two lines:

L1: y = 3x + 1

L2: y = -2x + 5

Mathematically solving this would be:

3x+1 = -2x + 5 5x+1 = 5 5x = 4 X = 4/5	Or	3x+1 = -2x + 5 1 = -5x + 5 -4 = -5x X = 4/5

So in code this would be:

tempM = l1.gradient – l2.gradient
tempC = l2.Yintecept – l1.Yintercept

X = tempC/tempM

Y value can be found by substituting into either line's equation

Finding Start Point

Create a Line between the last TrackPoint and the first TrackPoint, call this LastLine.

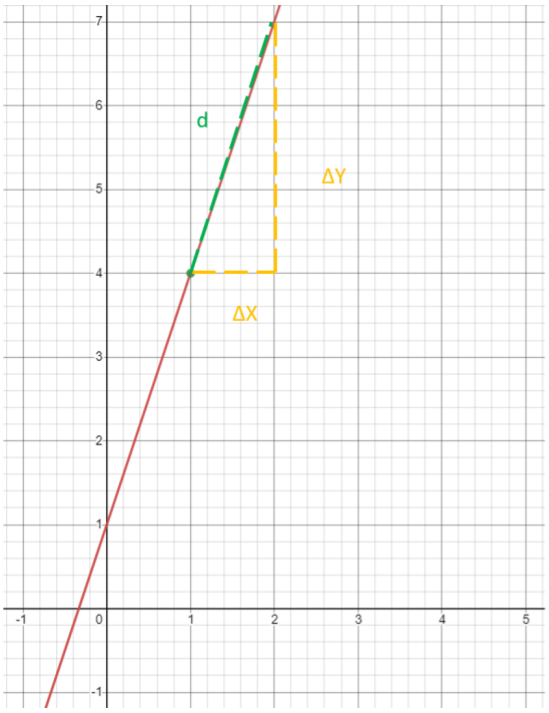
Generate a random X value that lies on the LastLine, and then find the corresponding Y value using the equation of LastLine

Finding Start Point edges:

Get the gradient of LastLine, find the perpendicular gradient (perpGrad) and then find the equation of the line that goes through the start point and has a gradient of perpGrad.

Then find the two points on that line that are a half-trackwidth distance away from TrackPoint. These are the two points which the start line is drawn between.

Finding a point a set distance along a line:



$d = \sqrt{(\Delta Y)^2+(\Delta X)^2}$

$d^2 = (\Delta Y)^2+(\Delta X)^2$
$d^2 - (\Delta X)^2 = (\Delta Y)^2$

$d^2 - (\Delta X)^2 = Gradient^2 * (\Delta X)^2$
$d^2 = Gradient^2 * (\Delta X)^2 + (\Delta X)^2$
$d^2 = (\Delta X)^2 * (Gradient^2 + 1)$
$d^2 / (Gradient^2 + 1) = (\Delta X)^2$

$\sqrt{(d^2 / (Gradient^2 + 1))} = \Delta X$

$Gradient = \Delta Y/\Delta X$

$Gradient * (\Delta X) = (\Delta Y)$
$Gradient^2 * (\Delta X)^2 = (\Delta Y)^2$

Then take ΔX and ± from the original X value to get 2 new values for X

Then substitute your new values for X into the equation of the line to get the corresponding V-values

Car:

Find Distances To edge of track:

```
Float[] findDistances()
      Float distances = new float[8]
      For each of the car's radar lines
            List<Vector2> POIs = new list
            For each of the track borders
                  If the car's line and the border line intersects
                        POIs.add(pointOfIntersection)
            If POIs.count > 0
                  lowestDistance = distance between car and POIs[0]
            For every coordinate in POIs
                  Find the distance between the car and the coordinate
                  If distance < lowestDistance
                        lowestDistance = distance
            distances[index corresponding to line] = lowestDistance
```

Track Fitness of cars:

```
Int[] passedCheckpoints = new int[checkpoints.length]

Initialize all of the numbers in passedCheckpoints to 0

Every frame:

      For each checkpoint

            Find distance between car and checkpoint

            If distance <= (trackwidth/2)

                  If the passedCheckpoint value for the last checkpoint is 1
                  greater than for the current checkpoint

                        passedCheckpoint[current checkpoint index]++

When collided:

      fitness = 0;

      For each number in passedCheckpoint:

            Fitness += number;
```

Neural Network:

FeedForward:

```
Float[] feedforward(float[] inputs)
	Set first layer of neurons to the inputs
	For the other layers
		For the neurons in the layer
			WeightedSum = 0
			For the weights corresponding to that neuron
				WeightedSum += weight * neuronValue
			WeightedSum += bias for the neuron
			Output = sigmoid(WeightedSum)
			Activation of neuron in next layer = output
	Return the last layer of neurons
```

Mutate:

```
Void Mutate(int percentageChanceOfMutation)
	For every bias
		Generate a random number between 0 and 100 (inclusive)
		If the random number <= percentageChanceOfMutation
			Bias += new randomly generated floating point number
	For every Weight
		Generate a random number between 0 and 100 (inclusive)
		If the random number <= percentageChanceOfMutation
			Weight += new randomly generated floating point number
```

Next Generation:

```
Cars[] generateNextGeneration(Car[] TrainingCars)
	Sort TrainingCars by their distance to the next checkpoint
	Sort TrainingCars by their fitness
	Reset all the TrainingCars
	For the top half of TrainingCars
		Copy the Neural Network of the car
		Mutate the neural Network
		Set the neural network of the corresponding car in the bottom
		half to the mutated net
	Increment generation counter
```

Technical Solution:

Program:

```
using System;

namespace Random_Track_Generation
{
    public static class Program
    {
        [STAThread]
        static void Main()
        {
            using (var game = new Game1())
                game.Run();
        }
    }
}
```

Game1:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

using System;
using System.Collections.Generic;

namespace Random_Track_Generation
{
    public class Game1 : Game
    {
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;

        //textures
        Texture2D greenRectangle;
        Texture2D carTexture;

        //coordinates to represent opposite corners of the rectangular game screen
        Vector2 gameBorderTL = new Vector2(320, 0); //Top Left Corner of game section
        Vector2 gameBorderBR = new Vector2(1920, 1080); //Bottom Right Corner of game
section

        //Font used for the writing of any text in the program
        SpriteFont arial;

        KeyboardState previousKState;

        //button attributes
        GenerateTrackButton generateTrackBtn;
        GenerateThreeTracksButton genThreeTracksBtn;
        ShowTrackButton show1;
        ShowTrackButton show2;
        ShowTrackButton show3;

        GenerateStraightLineTrackButton genStraightTrackBtn;
```

```
GenerateThreeStraightLineTracksButton genThreeStraightTracksBtn;
ShowTrackButton showStraight1;
ShowTrackButton showStraight2;
ShowTrackButton showStraight3;

SaveButton saveBtn;
LoadButton loadBtn;
InputTextBox saveLoadInput;

ResetCarsButton resetCarsBtn;

//status string to display any success or error messages
string statusString;

ToggleButton AIModeBtn;
ToggleButton trainingModeBtn;
ToggleButton autoNextGenBtn;

NextGenerationButton nextGenBtn;

SaveNeuralNetworkButton saveNeuralNetBtn;
LoadNeuralNetworkButton loadNeuralNetBtn;
InputTextBox saveLoadNeuralNetInput;

//current track and car attributes
Track currentTrack;
Car currentCar;

//arrays for generate three tracks functionality
Track[] threeTracks;
Track[] threeStraightTracks;

//all attributes used for the training mode of the program
int[] layers = new int[] { 9, 7, 7, 4 };
int mutationChancePercentage = 45;
int numberOfTrainingCars = 20;
Car[] trainingCars;
bool allCarsCollided = false;
int generation = 0;



public Game1()
{
    _graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    IsMouseVisible = true;
}

protected override void Initialize()
{

    _graphics.PreferredBackBufferWidth = 1920;
    _graphics.PreferredBackBufferHeight = 1080;
    //_graphics.ToggleFullScreen();
    _graphics.ApplyChanges();
```

```
            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);

            //Make the Green Rectangle for the background
            greenRectangle =
loadRectangle(Convert.ToInt32(_graphics.PreferredBackBufferWidth - gameBorderTL.X),
_graphics.PreferredBackBufferHeight, Color.Green);

            //load the font that is used to display all of the text in the program
            arial = Content.Load<SpriteFont>("Arial");

            previousKState = Keyboard.GetState();

            //the first track that is displayed is the default track (just a straight
line)
            currentTrack = new Track(gameBorderTL, gameBorderBR, arial, true);

            //Initailise all of the buttons
            generateTrackBtn = new GenerateTrackButton(50, 200, "Generate Track", new
Vector2(25, 5), arial, Mouse.GetState(), gameBorderTL, gameBorderBR, Color.Red);
            genThreeTracksBtn = new GenerateThreeTracksButton(50, 250, "Generate 3
Tracks", new Vector2(5, 60), arial, Mouse.GetState(), gameBorderTL, gameBorderBR,
Color.Red);
            show1 = new ShowTrackButton(50, 50, "1", new Vector2(20, 115), arial,
Mouse.GetState(), Color.Red);
            show2 = new ShowTrackButton(50, 50, "2", new Vector2(130, 115), arial,
Mouse.GetState(), Color.Red);
            show3 = new ShowTrackButton(50, 50, "3", new Vector2(250, 115), arial,
Mouse.GetState(), Color.Red);

            genStraightTrackBtn = new GenerateStraightLineTrackButton(50, 300, "Gen
Straight Line Track", new Vector2(5, 170), arial, Mouse.GetState(), gameBorderTL,
gameBorderBR, Color.Red);
            genThreeStraightTracksBtn = new GenerateThreeStraightLineTracksButton(50,
310, "Gen 3 Straight Line Track", new Vector2(5, 225), arial, Mouse.GetState(),
gameBorderTL, gameBorderBR, Color.Red);
            showStraight1 = new ShowTrackButton(50, 50, "1", new Vector2(20, 280), arial,
Mouse.GetState(), Color.Red);
            showStraight2 = new ShowTrackButton(50, 50, "2", new Vector2(130, 280),
arial, Mouse.GetState(), Color.Red);
            showStraight3 = new ShowTrackButton(50, 50, "3", new Vector2(250, 280),
arial, Mouse.GetState(), Color.Red);

            resetCarsBtn = new ResetCarsButton(50, 200, "Reset Car(s)", new Vector2(60,
345), arial, Mouse.GetState(), gameBorderTL, gameBorderBR, Color.Red);

            saveBtn = new SaveButton(50, 140, "Save Track", new Vector2(5, 400), arial,
Mouse.GetState(), Color.Red);
            loadBtn = new LoadButton(50, 140, "Load Track", new Vector2(170, 400), arial,
Mouse.GetState(), Color.Red);
            saveLoadInput = new InputTextBox(50, 310, "", new Vector2(5, 455), arial,
Mouse.GetState(), Color.LightGray);
```

```
            AIModeBtn = new ToggleButton(50, 300, "Enable AI Mode", "Enable Manual Mode",
new Vector2(10, 515), arial, Mouse.GetState(), Color.Aqua, Color.Aquamarine);
            trainingModeBtn = new ToggleButton(50, 300, "Enable training Mode", "Disable
training Mode", new Vector2(10, 570), arial, Mouse.GetState(), Color.DarkOrange,
Color.Yellow);
            autoNextGenBtn = new ToggleButton(75, 300, "Enable Auto Next \n generation
Mode", "Disable Auto Next \n generation Mode", new Vector2(10, 625), arial,
Mouse.GetState(), Color.DarkOrange, Color.Yellow);
            nextGenBtn = new NextGenerationButton(75, 300, "Click to Load \n Next
Generation", new Vector2(10, 705), arial, Mouse.GetState(), gameBorderTL, gameBorderBR,
Color.Orange);

            saveNeuralNetBtn = new SaveNeuralNetworkButton(50, 300, "Save Neural
Network", new Vector2(10, 785), arial, Mouse.GetState(), Color.Red);
            loadNeuralNetBtn = new LoadNeuralNetworkButton(50, 300, "Load Neural
Network", new Vector2(10, 785), arial, Mouse.GetState(), Color.Red);
            saveLoadNeuralNetInput = new InputTextBox(50, 310, "", new Vector2(5, 840),
arial, Mouse.GetState(), Color.LightGray);

            //initialise arrays used for the functionality of the generate three tracks
buttons
            threeTracks = new Track[3];
            threeStraightTracks = new Track[3];

            //initialise cars
            carTexture = Content.Load<Texture2D>("SmallRectangle");
            double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());
            currentCar = new Car(currentTrack.getStartPoint(), (float) carRotation,
carTexture, currentTrack.getCheckpoints(), currentTrack.getTrackWidth());

            trainingCars = new Car[numberOfTrainingCars];
            for (int i = 0; i < trainingCars.Length; i++)
            {
                trainingCars[i] = new Car(currentTrack.getStartPoint(),
(float)carRotation, carTexture, currentTrack.getCheckpoints(),
currentTrack.getTrackWidth(), layers);
            }

            //initialise the status string
            statusString = "Ready";


        }

        Texture2D loadRectangle(int width, int height, Color color) //I Made this method
before i added monogame extended
        {
            Texture2D texture = new Texture2D(GraphicsDevice, width, height);
            Color[] colourPixels = new Color[width * height];
            for (int i = 0; i < colourPixels.Length; i++)
            {
                colourPixels[i] = color;
            }
            texture.SetData<Color>(colourPixels);

            return texture;
        }
```

```
        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            float[] distances;

            generateTrackBtn.Update(gameTime, Mouse.GetState(), ref currentTrack, ref
currentCar,ref trainingCars, ref statusString);

            genThreeTracksBtn.Update(gameTime, Mouse.GetState(), ref threeTracks[0], ref
threeTracks[1], ref threeTracks[2], ref statusString);
            show1.Update(gameTime, Mouse.GetState(),ref currentTrack, ref currentCar,ref
trainingCars, threeTracks[0], ref statusString);
            show2.Update(gameTime, Mouse.GetState(), ref currentTrack, ref currentCar,
ref trainingCars, threeTracks[1], ref statusString);
            show3.Update(gameTime, Mouse.GetState(), ref currentTrack, ref currentCar,
ref trainingCars, threeTracks[2], ref statusString);

            genStraightTrackBtn.Update(gameTime, Mouse.GetState(), ref currentTrack, ref
currentCar, ref trainingCars, ref statusString);

            genThreeStraightTracksBtn.Update(gameTime, Mouse.GetState(), ref
threeStraightTracks[0], ref threeStraightTracks[1], ref threeStraightTracks[2], ref
statusString);
            showStraight1.Update(gameTime, Mouse.GetState(), ref currentTrack, ref
currentCar, ref trainingCars, threeStraightTracks[0], ref statusString);
            showStraight2.Update(gameTime, Mouse.GetState(), ref currentTrack, ref
currentCar, ref trainingCars, threeStraightTracks[1], ref statusString);
            showStraight3.Update(gameTime, Mouse.GetState(), ref currentTrack, ref
currentCar, ref trainingCars, threeStraightTracks[2], ref statusString);

            saveLoadInput.Update(gameTime, Mouse.GetState(), Keyboard.GetState(),
previousKState);
            saveBtn.Update(gameTime, Mouse.GetState(), ref currentTrack,
saveLoadInput.getText(), ref statusString);
            loadBtn.Update(gameTime, Mouse.GetState(), ref currentTrack, ref currentCar,
ref trainingCars, saveLoadInput.getText(), ref statusString);

            resetCarsBtn.Update(gameTime, Mouse.GetState(), ref currentTrack, ref
currentCar, ref trainingCars);

            AIModeBtn.Update(gameTime, Mouse.GetState());
            if (AIModeBtn.getToggled() == false)
            {
                trainingModeBtn.setToggled(false);
                //could also set autoNextGenBtn to false
                //but i thought it might get annoying if the user is constantly switching
between manual and AI modes
            }
            else if (AIModeBtn.getToggled() == true)
            {
                trainingModeBtn.Update(gameTime, Mouse.GetState());
                autoNextGenBtn.Update(gameTime, Mouse.GetState());
            }
```

```
            if (AIModeBtn.getToggled() == false)
            {
                //if its in manual mode
                currentCar.Update(gameTime, GraphicsDevice);
            }
            else if (AIModeBtn.getToggled() == true && trainingModeBtn.getToggled() ==
false)
            {
                //if its in Ai mode with just 1 car
                distances = findDistancesFromCar(currentCar);
                currentCar.Update(gameTime, GraphicsDevice, distances, false,
autoNextGenBtn.getToggled());
                saveLoadNeuralNetInput.Update(gameTime, Mouse.GetState(),
Keyboard.GetState(), previousKState);
                loadNeuralNetBtn.Update(gameTime, Mouse.GetState(), ref currentCar,
saveLoadNeuralNetInput.getText(), ref statusString);
            }
            else if (trainingModeBtn.getToggled() == true)
            {
                //if its in training mode

                for (int i = 0; i < trainingCars.Length; i++)
                {
                    distances = findDistancesFromCar(trainingCars[i]);
                    trainingCars[i].Update(gameTime, GraphicsDevice, distances, true,
autoNextGenBtn.getToggled());
                }

                saveLoadNeuralNetInput.Update(gameTime, Mouse.GetState(),
Keyboard.GetState(), previousKState);

                //Had to do the save Neural Net logic in here because it requires the use
of the sorting algorithms
                saveNeuralNetBtn.UpdateClicked(gameTime, Mouse.GetState());
                if (saveNeuralNetBtn.getClicked() == true)
                {
                    saveNeuralNetBtn.setClicked(false);

                    Car[] distanceSortedCars = mergeSortDistances(trainingCars);
                    Car[] fitnessSortedCars = mergeSortFitnesses(distanceSortedCars);
                    fitnessSortedCars[fitnessSortedCars.Length -
1].getNeuralNetwork().saveNeuralNetwork(saveLoadNeuralNetInput.getText(), ref
statusString);
                }

                if (autoNextGenBtn.getToggled() == true)
                {
                    //if you want it to automaticcaly load the next generation once all
the cars have crashed, or a certain amount of time has crashed

                    allCarsCollided = true;
                    for (int i = 0; i < trainingCars.Length; i++)
                    {
                        if (trainingCars[i].getCollided() == false)
                        {
                            allCarsCollided = false;
                            break;
```

```
                }
            }

            if (allCarsCollided)
            {
                generateNextGeneration();
            }
        }
        else if (autoNextGenBtn.getToggled() == false)
        {
            //For when you want to manually move to the next generation
            bool nextGen = nextGenBtn.Update(gameTime, Mouse.GetState());

            if (nextGen)
            {
                generateNextGeneration();
            }
        }

    }


    previousKState = Keyboard.GetState();
    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.WhiteSmoke);
    _spriteBatch.Begin();

    //drawing the green background for the
    _spriteBatch.Draw(greenRectangle, gameBorderTL, Color.White);


    generateTrackBtn.Draw(_spriteBatch);
    genThreeTracksBtn.Draw(_spriteBatch);
    show1.Draw(_spriteBatch);
    show2.Draw(_spriteBatch);
    show3.Draw(_spriteBatch);

    genStraightTrackBtn.Draw(_spriteBatch);
    genThreeStraightTracksBtn.Draw(_spriteBatch);
    showStraight1.Draw(_spriteBatch);
    showStraight2.Draw(_spriteBatch);
    showStraight3.Draw(_spriteBatch);

    saveBtn.Draw(_spriteBatch);
    loadBtn.Draw(_spriteBatch);
    saveLoadInput.Draw(_spriteBatch);

    resetCarsBtn.Draw(_spriteBatch);

    AIModeBtn.Draw(_spriteBatch);
    if (AIModeBtn.getToggled() == true)
    {
        // if Ai mode
        trainingModeBtn.Draw(_spriteBatch);
```

```
                saveLoadNeuralNetInput.Draw(_spriteBatch);

            }
            if (AIModeBtn.getToggled() == true && trainingModeBtn.getToggled() == false)
            {
                //if its only 1 car in ai mode
                loadNeuralNetBtn.Draw(_spriteBatch);
            }
            if (trainingModeBtn.getToggled() == true)
            {
                // if training mode
                autoNextGenBtn.Draw(_spriteBatch);
                if (autoNextGenBtn.getToggled() == false)
                {
                    nextGenBtn.Draw(_spriteBatch);
                }

                saveNeuralNetBtn.Draw(_spriteBatch);
            }

            //darwing the status string at the bottom
            _spriteBatch.FillRectangle(new Vector2(0, 900), new Size2(320, 75),
Color.LightGreen);
            _spriteBatch.DrawString(arial, statusString, new Vector2(5, 910),
Color.Black);

            //draw the track
            currentTrack.Draw(_spriteBatch);


            //Drawing the cars/
            if (trainingModeBtn.getToggled() == true)
            {
                for (int i = 0; i < trainingCars.Length; i++)
                {
                    trainingCars[i].Draw(_spriteBatch);
                }
            }
            else
            {
                currentCar.Draw(_spriteBatch);
                _spriteBatch.DrawString(arial, $"Speed:
{Math.Round(currentCar.getSpeed(), 2)}", new Vector2(20, gameBorderBR.Y - 100),
Color.Black);
            }


            _spriteBatch.End();
            base.Draw(gameTime);
        }

        float[] findDistancesFromCar(Car car)
        {
            //this is the method to find the distances between the car and the edge of
the track
            float[] distancesFromCar = new float[8];
            Line[] carLines;
            List<Line> insideLineBorders;
```

```
            List<Line> outsideLineBorders;
            Vector2[] carEdgePositions;
            float minDistance = 0;


            carLines = car.getCarLines();
            carEdgePositions = car.getEdgePoints();
            insideLineBorders = currentTrack.getInsideLineBorders();
            outsideLineBorders = currentTrack.getOutsideLineBorders();

            //go through each of the cars radar lines
            for (int i = 0; i < carLines.Length; i++)
            {
                //check if there are any points of intersection between the car's line
and the track's border-lines
                List<Vector2> insidePOIs = new List<Vector2>();
                for (int j = 0; j < insideLineBorders.Count; j++)
                {
                    insidePOIs.Add(Line.findPOI(carLines[i], insideLineBorders[j]));
                }

                List<Vector2> outsidePOIs = new List<Vector2>();
                for (int j = 0; j < outsideLineBorders.Count; j++)
                {
                    outsidePOIs.Add(Line.findPOI(carLines[i], outsideLineBorders[j]));
                }

                List<Vector2> allPOIs = new List<Vector2>();
                for (int j = 0; j < insidePOIs.Count; j++)
                {
                    if (!float.IsNaN(insidePOIs[j].X))
                    {
                        allPOIs.Add(insidePOIs[j]);
                    }
                }
                for (int j = 0; j < outsidePOIs.Count; j++)
                {
                    if (!float.IsNaN(outsidePOIs[j].X))
                    {
                        allPOIs.Add(outsidePOIs[j]);
                    }
                }


                //find distances between the car's point and any of the points of
intersection
                List<float> distances = new List<float>();

                for (int j = 0; j < allPOIs.Count; j++)
                {
                    distances.Add((float) Track.findDistance(new
TrackPoint(carEdgePositions[i]), new TrackPoint(allPOIs[j])));
                }

                if (distances.Count > 0)
                {
                    minDistance = distances[0];
                }
```

```
                //only keep the distance that is the lowest, i.e. the distance to the
closest wall in that direction
                for (int j = 0; j < distances.Count; j++)
                {
                    if (distances[j] < minDistance)
                    {
                        minDistance = distances[j];
                    }
                }

                distancesFromCar[i] = minDistance;
            }

            return distancesFromCar;
        }


        Car[] mergeSortDistances(Car[] items)
        {
            //Sort cars by their distance to the next checkpoint, highest to lowest
            Car[] left_half;
            Car[] right_half;

            //Base case for recursion
            if (items.Length < 2)
            {
                return items;
            }

            int midpoint = items.Length / 2;

            //Do the left half
            left_half = new Car[midpoint];
            for (int i = 0; i < midpoint; i++)
            {
                left_half[i] = items[i];
            }

            //figure out how big the right half should be
            if (items.Length % 2 == 0)
            {
                right_half = new Car[midpoint];
            }
            else
            {
                right_half = new Car[midpoint + 1];
            }

            //fill in hte right half
            int rightIndex = 0;
            for (int i = midpoint; i < items.Length; i++)
            {
                right_half[rightIndex] = items[i];
                rightIndex++;
            }

            //recursion bit
```

```
            left_half = mergeSortDistances(left_half);
            right_half = mergeSortDistances(right_half);

            items = mergeDistances(left_half, right_half);

            return items;
        }

        Car[] mergeDistances(Car[] list1, Car[] list2)
        {

            Car[] merged = new Car[list1.Length + list2.Length];

            int index1 = 0;
            int index2 = 0;
            int indexMerged = 0;

            while (index1 < list1.Length && index2 < list2.Length)
            {
                if (list1[index1].getDistanceToNextCheckpoint() >
list2[index2].getDistanceToNextCheckpoint())
                {
                    merged[indexMerged] = list1[index1];
                    index1++;
                }
                else
                {
                    merged[indexMerged] = list2[index2];
                    index2++;
                }
                indexMerged++;
            }

            while (index1 < list1.Length)
            {
                merged[indexMerged] = list1[index1];
                index1++;
                indexMerged++;
            }

            while (index2 < list2.Length)
            {
                merged[indexMerged] = list2[index2];
                index2++;
                indexMerged++;
            }

            return merged;
        }

        Car[] mergeSortFitnesses(Car[] items)
        {
            //Sort cars by their fitness, lowest to highest
            Car[] left_half;
            Car[] right_half;

            //Base case for recursion
            if (items.Length < 2)
```

```
    {
        return items;
    }

    int midpoint = items.Length / 2;

    //Do the left half
    left_half = new Car[midpoint];
    for (int i = 0; i < midpoint; i++)
    {
        left_half[i] = items[i];
    }

    //figure out how big the right half should be
    if (items.Length % 2 == 0)
    {
        right_half = new Car[midpoint];
    }
    else
    {
        right_half = new Car[midpoint + 1];
    }

    //fill in hte right half
    int rightIndex = 0;
    for (int i = midpoint; i < items.Length; i++)
    {
        right_half[rightIndex] = items[i];
        rightIndex++;
    }

    //recursion bit
    left_half = mergeSortFitnesses(left_half);
    right_half = mergeSortFitnesses(right_half);

    items = mergeFitnesses(left_half, right_half);

    return items;
}

Car[] mergeFitnesses(Car[] list1, Car[] list2)
{
    Car[] merged = new Car[list1.Length + list2.Length];

    int index1 = 0;
    int index2 = 0;
    int indexMerged = 0;

    while (index1 < list1.Length && index2 < list2.Length)
    {
        if (list1[index1].getFitness() <= list2[index2].getFitness())
        {
            merged[indexMerged] = list1[index1];
            index1++;
        }
        else
        {
            merged[indexMerged] = list2[index2];
```

```
                index2++;
            }
            indexMerged++;
        }

        while (index1 < list1.Length)
        {
            merged[indexMerged] = list1[index1];
            index1++;
            indexMerged++;
        }

        while (index2 < list2.Length)
        {
            merged[indexMerged] = list2[index2];
            index2++;
            indexMerged++;
        }

        return merged;
    }

    void generateNextGeneration()
    {
        //sort cars so that in the end they are sorted by fitness (lowest to highest)
and
        //any cars with the same fitness is sorted by their distance to next
checkpoint (highest to lowest)
        //meaning that the worst cars will be at the lower indexes of the array
        Car[] distanceSortedCars = mergeSortDistances(trainingCars);
        Car[] fitnessSortedCars = mergeSortFitnesses(distanceSortedCars);

        double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());

        for (int i = 0; i < trainingCars.Length; i++)
        {
            trainingCars[i] = fitnessSortedCars[i];

            //reset the cars so that all of their attributes are reset and dont have
an effect on the next generation
            trainingCars[i].reset(currentTrack.getStartPoint(), (float)carRotation);
        }

        //take the Neural networks from the best half of the cars, copy them, mutate
them and then give them to the worst half of the cars
        for (int i = 0; i < (trainingCars.Length / 2); i++)
        {
            NeuralNetwork copyNet = new NeuralNetwork(layers);
            copyNet = trainingCars[i + trainingCars.Length /
2].getNeuralNetwork().copyNetwork(copyNet);
            copyNet.mutate(mutationChancePercentage);
            trainingCars[i].setNeuralNetwork(copyNet);

        }

        generation++;
        statusString = $"Generation: {generation}";
```

```
        }


    }
}
```

Button:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class Button
    {
        //Attributes of the button
        int height;
        int width;
        protected string buttonText;
        Vector2 buttonPosition;
        Vector2 stringPos;
        SpriteFont font;

        Color buttonColor; //the original color of the button, this wont change after its
been set

        protected Color currentButtonColor;
        bool isHovering;

        MouseState currentMState;
        MouseState previousMState;

        protected bool isClicked;



        public Button(int newHeight, int newWidth, string text, Vector2 position,
SpriteFont newfont, MouseState mstate, Color newColor)
        {
            //initialise the values of all of the attributes of the button
            height = newHeight;
            width = newWidth;
            buttonText = text;
            buttonColor = newColor;
            buttonPosition = position;
            font = newfont;

            currentButtonColor = buttonColor;
            stringPos = new Vector2(buttonPosition.X + (width / 10), buttonPosition.Y +
(height / 10));
```

```
            isClicked = false;
            isHovering = false;
            previousMState = mstate;
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            //draw the button and the text of the button
            spriteBatch.DrawRectangle(buttonPosition, new Size2(width, height),
currentButtonColor, height/2 + 1);
            spriteBatch.DrawString(font, buttonText, stringPos , Color.Black);
        }

        public void UpdateClicked(GameTime gameTime, MouseState mState)
        {
            //update the mouse state every frame
            currentMState = mState;

            //check if the mouse is hovering over the button in this frame
            isHovering = checkHovering(currentMState);

            //if the mouse is hovering over the button, make it blue, if it isn't then
set it to its normal color
            if (isHovering)
            {
                currentButtonColor = Color.DodgerBlue;
            }
            else
            {
                currentButtonColor = buttonColor;
            }

            //if the button is clicked, run the code that is meant to be run when clicked
            if (checkClicked())
            {
                isClicked = true;
            }


            //update the previous mouse state right at the end of the update method
            previousMState = currentMState;
        }

        bool checkHovering(MouseState mstate)
        {
            //this method checks if the mouse is currently hovering above this button
            if ((mstate.X > buttonPosition.X) && (mstate.X <= (buttonPosition.X + width))
&& (mstate.Y > buttonPosition.Y) && (mstate.Y <= (buttonPosition.Y + height)))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

```
        bool checkClicked()
        {
            //thsi mehtod checks if this button has been clicked
            if (isHovering  && previousMState.LeftButton == ButtonState.Released &&
currentMState.LeftButton == ButtonState.Pressed)
            {
                return true;
            }
            else
            {
                return false;
            }
        }




    }
}
```

ToggleButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class ToggleButton : Button
    {
        bool toggled;
        string textUntoggled;
        string textToggled;
        Color colourUntoggled;
        Color colourToggled;

        public ToggleButton(int newHeight, int newWidth, string textUntoggled, string
textToggled, Vector2 position, SpriteFont newfont, MouseState mstate, Color
colourUntoggled, Color colourToggled) : base(newHeight, newWidth, textUntoggled,
position, newfont, mstate, colourUntoggled)
        {
            this.textUntoggled = textUntoggled;
            this.textToggled = textToggled;
            this.colourUntoggled = colourUntoggled;
            this.colourToggled = colourToggled;

            isClicked = false;
            toggled = false;
        }

        public void Update(GameTime gameTime, MouseState mState)
        {
            UpdateClicked(gameTime, mState);
```

```
            //when clicked, the button changes to the opposite state
            if (isClicked)
            {
                isClicked = false;

                if (toggled)
                {
                    toggled = false;
                }
                else
                {
                    toggled = true;
                }

            }

            if (toggled)
            {
                currentButtonColor = colourToggled;
                buttonText = textToggled;
            }
            else
            {
                currentButtonColor = colourUntoggled;
                buttonText = textUntoggled;
            }

        }

        public bool getToggled()
        {
            return toggled;
        }

        public void setToggled(bool newVal)
        {
            toggled = newVal;
        }
    }
}
```

GenerateTrackButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class GenerateTrackButton : Button
    {
```

```
        Track generatedTrack;
        Vector2 gameBorderTL; //Top Left Corner of game section
        Vector2 gameBorderBR; //Bottom Right Corner of game section
        SpriteFont font;


        public GenerateTrackButton(int newHeight, int newWidth, string text, Vector2
position, SpriteFont newfont, MouseState mstate, Vector2 topLeftBorder, Vector2
bottomRightBorder, Color newColor) : base(newHeight, newWidth, text, position, newfont,
mstate, newColor)
        {
            font = newfont;
            gameBorderTL = topLeftBorder;
            gameBorderBR = bottomRightBorder;

            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState, ref Track currentTrack,
ref Car currentCar, ref Car[] trainingCars, ref string statusString)
        {
            UpdateClicked(gameTime, mState);

            if (isClicked)
            {
                isClicked = false;
                generatedTrack = new Track(gameBorderTL, gameBorderBR, font);
                currentTrack = generatedTrack;

                double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());
                currentCar.reset(currentTrack.getStartPoint(), (float)carRotation,
currentTrack.getCheckpoints());

                for (int i = 0; i < trainingCars.Length; i++)
                {
                    trainingCars[i].reset(currentTrack.getStartPoint(),
(float)carRotation, currentTrack.getCheckpoints());
                }

                statusString = "Track Successfully \nGenerated";
            }
        }



    }
}
```

GenerateThreeTracksButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class GenerateThreeTracksButton : Button
    {
        Track generatedTrack;
        Vector2 gameBorderTL; //Top Left Corner of game section
        Vector2 gameBorderBR; //Bottom Right Corner of game section
        SpriteFont font;

        public GenerateThreeTracksButton(int newHeight, int newWidth, string text,
Vector2 position, SpriteFont newfont, MouseState mstate, Vector2 topLeftBorder, Vector2
bottomRightBorder, Color newColor) : base(newHeight, newWidth, text, position, newfont,
mstate, newColor)
        {
            font = newfont;
            gameBorderTL = topLeftBorder;
            gameBorderBR = bottomRightBorder;

            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState,ref Track currentTrack1,
ref Track currentTrack2, ref Track currentTrack3, ref string statusString)
        {
            UpdateClicked(gameTime, mState);
            if (isClicked)
            {
                isClicked = false;

                generatedTrack = new Track(gameBorderTL, gameBorderBR, font);
                currentTrack1 = generatedTrack;

                generatedTrack = new Track(gameBorderTL, gameBorderBR, font);
                currentTrack2 = generatedTrack;

                generatedTrack = new Track(gameBorderTL, gameBorderBR, font);
                currentTrack3 = generatedTrack;

                statusString = "3 Tracks Generated";
            }
        }
    }
}
```

GenerateStraightLineTrackButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class GenerateStraightLineTrackButton : Button
    {
        Track generatedTrack;
        Vector2 gameBorderTL; //Top Left Corner of game section
        Vector2 gameBorderBR; //Bottom Right Corner of game section
        SpriteFont font;


        public GenerateStraightLineTrackButton(int newHeight, int newWidth, string text,
Vector2 position, SpriteFont newfont, MouseState mstate, Vector2 topLeftBorder, Vector2
bottomRightBorder, Color newColor) : base(newHeight, newWidth, text, position, newfont,
mstate, newColor)
        {
            font = newfont;
            gameBorderTL = topLeftBorder;
            gameBorderBR = bottomRightBorder;

            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState, ref Track currentTrack,
ref Car currentCar, ref Car[] trainingCars, ref string statusString)
        {
            UpdateClicked(gameTime, mState);

            if (isClicked)
            {
                isClicked = false;
                generatedTrack = new Track(gameBorderTL, gameBorderBR, font, true, true);
                currentTrack = generatedTrack;

                double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());
                currentCar.reset(currentTrack.getStartPoint(), (float)carRotation,
currentTrack.getCheckpoints());

                for (int i = 0; i < trainingCars.Length; i++)
                {
                    trainingCars[i].reset(currentTrack.getStartPoint(),
(float)carRotation, currentTrack.getCheckpoints());
                }

                statusString = "Track Successfully \ngenerated";
            }
        }
    }
```

```
}
```

GenerateThreeStraightLineTracksButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class GenerateThreeStraightLineTracksButton : Button
    {
        Track generatedTrack;
        Vector2 gameBorderTL; //Top Left Corner of game section
        Vector2 gameBorderBR; //Bottom Right Corner of game section
        SpriteFont font;

        public GenerateThreeStraightLineTracksButton(int newHeight, int newWidth, string
text, Vector2 position, SpriteFont newfont, MouseState mstate, Vector2 topLeftBorder,
Vector2 bottomRightBorder, Color newColor) : base(newHeight, newWidth, text, position,
newfont, mstate, newColor)
        {
            font = newfont;
            gameBorderTL = topLeftBorder;
            gameBorderBR = bottomRightBorder;

            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState, ref Track currentTrack1,
ref Track currentTrack2, ref Track currentTrack3, ref string statusString)
        {
            UpdateClicked(gameTime, mState);
            if (isClicked)
            {
                isClicked = false;

                generatedTrack = new Track(gameBorderTL, gameBorderBR, font, true, true);
                currentTrack1 = generatedTrack;

                generatedTrack = new Track(gameBorderTL, gameBorderBR, font, true, true);
                currentTrack2 = generatedTrack;

                generatedTrack = new Track(gameBorderTL, gameBorderBR, font, true, true);
                currentTrack3 = generatedTrack;

                statusString = "3 Tracks Generated";
            }
        }
    }
}
```

ShowTrackButton:

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Random_Track_Generation
{
    class ShowTrackButton : Button
    {

        public ShowTrackButton(int newHeight, int newWidth, string text, Vector2
position, SpriteFont newfont, MouseState mstate, Color newColor) : base(newHeight,
newWidth, text, position, newfont, mstate, newColor)
        {
            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState, ref Track currentTrack,
ref Car currentCar, ref Car[] trainingCars, Track trackToBeLoaded, ref string
statusString)
        {
            UpdateClicked(gameTime, mState);
            if (isClicked)
            {
                isClicked = false;
                if (trackToBeLoaded == null)
                {
                    statusString = "Track Not Loaded";
                    return;
                }
                currentTrack = trackToBeLoaded;

                double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());
                currentCar.reset(currentTrack.getStartPoint(), (float)carRotation,
currentTrack.getCheckpoints());

                for (int i = 0; i < trainingCars.Length; i++)
                {
                    trainingCars[i].reset(currentTrack.getStartPoint(),
(float)carRotation, currentTrack.getCheckpoints());
                }

                statusString = "Track Loaded";
            }

        }



    }
}
```

SaveButton

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;



namespace Random_Track_Generation
{
    class SaveButton : Button
    {
        public SaveButton(int newHeight, int newWidth, string text, Vector2 position,
SpriteFont newfont, MouseState mstate, Color newColor) : base(newHeight, newWidth, text,
position,  newfont,  mstate,  newColor)
        {
            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState ,ref Track currentTrack,
string filename, ref string statusString)
        {


            UpdateClicked(gameTime, mState);

            if (isClicked)
            {
                isClicked = false;

                //check if the a track already exists with the given name
                if (filename == "")
                {
                    statusString = "Please Enter a name for the \ntrack. Then try again";
                    return;
                }
                if (Directory.Exists(filename))
                {
                    statusString = "A File already exists with this name";
                    return;
                }

                //Code to save track
                string filepath;
                StreamWriter writer;
                Directory.CreateDirectory(filename);

                //save Startpoint and startpoint edges
                filepath = $@"{filename}/StartPoints.txt";
                writer = new StreamWriter(filepath);
                writer.WriteLine($"{currentTrack.getStartPoint().X},
{currentTrack.getStartPoint().Y}");
```

```
writer.WriteLine($"{currentTrack.getStartPointEdges()[0].getPosition().X},
{currentTrack.getStartPointEdges()[0].getPosition().Y}");

writer.WriteLine($"{currentTrack.getStartPointEdges()[1].getPosition().X},
{currentTrack.getStartPointEdges()[1].getPosition().Y}");
                writer.Close();

                //save the final points
                List<TrackPoint> finalPoints = currentTrack.getFinalPoints();
                filepath = $@"{filename}/Finalpoints.txt";
                writer = new StreamWriter(filepath);
                for (int i = 0; i < finalPoints.Count; i++)
                {
                    writer.WriteLine($"{finalPoints[i].getPosition().X},
{finalPoints[i].getPosition().Y}");
                }
                writer.Close();

                statusString = "Successfully Saved Track";
            }

        }
    }
}
```

LoadButton:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;


using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class LoadButton : Button
    {
        Track LoadedTrack;

        public LoadButton(int newHeight, int newWidth, string text, Vector2 position,
SpriteFont newfont, MouseState mstate,  Color newColor) : base(newHeight, newWidth, text,
position, newfont, mstate, newColor)
        {
            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState, ref Track currentTrack,
ref Car currentCar, ref Car[] trainingCars, string filename, ref string statusString)
        {
            UpdateClicked(gameTime, mState);
            if (isClicked)
```

```
            {
                isClicked = false;
                if (filename == "")
                {
                    statusString = "Please enter the track's\nname that you wish to
load";
                    return;
                }
                LoadedTrack = new Track(filename, ref statusString);
                if (LoadedTrack.getTrackPossible() == false)
                {
                    return;
                }
                currentTrack = LoadedTrack;

                double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());
                currentCar.reset(currentTrack.getStartPoint(), (float)carRotation,
currentTrack.getCheckpoints());

                for (int i = 0; i < trainingCars.Length; i++)
                {
                    trainingCars[i].reset(currentTrack.getStartPoint(),
(float)carRotation, currentTrack.getCheckpoints());
                }

            }
        }
    }
}
```

SaveNeuralNetworkButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class SaveNeuralNetworkButton : Button
    {
        public SaveNeuralNetworkButton(int newHeight, int newWidth, string text, Vector2
position, SpriteFont newfont, MouseState mstate, Color newColor) : base(newHeight,
newWidth, text, position, newfont, mstate, newColor)
        {
            isClicked = false;
        }
        public bool getClicked()
        {
            return isClicked;
        }

        public void setClicked(bool clicked)
```

```
        {
            isClicked = clicked;
        }
    }
}
```

LoadNeuralNetworkButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class LoadNeuralNetworkButton : Button
    {
        public LoadNeuralNetworkButton(int newHeight, int newWidth, string text, Vector2
position, SpriteFont newfont, MouseState mstate, Color newColor) : base(newHeight,
newWidth, text, position, newfont, mstate, newColor)
        {
            isClicked = false;

        }

        public void Update(GameTime gameTime, MouseState mState, ref Car currentCar,
string filename, ref string statusString)
        {
            UpdateClicked(gameTime, mState);
            if (isClicked)
            {
                isClicked = false;
                string filePath = filename + ".txt";
                NeuralNetwork newNet = new NeuralNetwork(filePath, ref statusString);
                if (newNet.getBiases() != null)
                {
                    currentCar.setNeuralNetwork(newNet);
                }


            }
        }
    }
}
```

InputTextBox:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

using Microsoft.Xna.Framework;
```

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class InputTextBox : Button
    {
        //this need attributes to store the keys that re being pressed as well as if
shift is being held
        Keys[] keys;
        bool shift;

        public InputTextBox(int newHeight, int newWidth, string text, Vector2 position,
SpriteFont newfont, MouseState mstate, Color newColor) : base(newHeight, newWidth, text,
position, newfont, mstate, newColor)
        {
            isClicked = false;
            buttonText = "";
        }

        public void Update(GameTime gameTime, MouseState mState, KeyboardState kstate,
KeyboardState previouskstate)
        {
            UpdateClicked(gameTime, mState);
            if (isClicked)
            {
                currentButtonColor = Color.LightBlue;
                keys = kstate.GetPressedKeys();

                //only exit typing mode when enter is pressed
                if (kstate.IsKeyDown(Keys.Enter))
                {
                    isClicked = false;
                    return;
                }

                //check for shift being pressed
                if (kstate.IsKeyDown(Keys.LeftShift) ||
kstate.IsKeyDown(Keys.RightShift))
                {
                    shift = true;
                }
                else
                {
                    shift = false;
                }

                //delete a letter when backspace is pressed
                if (buttonText.Length>0 && kstate.IsKeyDown(Keys.Back) &&
previouskstate.IsKeyDown(Keys.Back) == false)
                {
                    buttonText = buttonText.Substring(0, buttonText.Length - 1);
                }

                //check for vaid inputs and add them to the box's text
                if (keys.Length > 0 && previouskstate.IsKeyDown(keys[0]) == false)
                {
```

```
switch (keys[0])
{
    case Keys.Space:
        buttonText += " ";
        break;
    case Keys.D0:
        if (shift)
        {
            buttonText += ")";
        }
        else
        {
            buttonText += "0";
        }
        break;
    case Keys.D1:
        if (shift)
        {
            buttonText += "!";
        }
        else
        {
            buttonText += "1";
        }
        break;
    case Keys.D2:
        buttonText += "2";
        break;
    case Keys.D3:
        if (shift)
        {
            buttonText += "£";
        }
        else
        {
            buttonText += "3";
        }
        break;
    case Keys.D4:
        if (shift)
        {
            buttonText += "$";
        }
        else
        {
            buttonText += "4";
        }
        break;
    case Keys.D5:
        if (shift)
        {
            buttonText += "%";
        }
        else
        {
            buttonText += "5";
        }
        break;
```

```
case Keys.D6:
    if (shift)
    {
        buttonText += "^";
    }
    else
    {
        buttonText += "6";
    }
    break;
case Keys.D7:
    if (shift)
    {
        buttonText += "&";
    }
    else
    {
        buttonText += "7";
    }
    break;
case Keys.D8:
    buttonText += "8";
    break;
case Keys.D9:
    if (shift)
    {
        buttonText += "(";
    }
    else
    {
        buttonText += "9";
    }
    break;
case Keys.A:
    if (shift)
    {
        buttonText += "A";
    }
    else
    {
        buttonText += "a";
    }
    break;
case Keys.B:
    if (shift)
    {
        buttonText += "B";
    }
    else
    {
        buttonText += "b";
    }
    break;
case Keys.C:
    if (shift)
    {
        buttonText += "C";
    }
```

```
            else
            {
                buttonText += "c";
            }
            break;
        case Keys.D:
            if (shift)
            {
                buttonText += "D";
            }
            else
            {
                buttonText += "d";
            }
            break;
        case Keys.E:
            if (shift)
            {
                buttonText += "E";
            }
            else
            {
                buttonText += "e";
            }
            break;
        case Keys.F:
            if (shift)
            {
                buttonText += "F";
            }
            else
            {
                buttonText += "f";
            }
            break;
        case Keys.G:
            if (shift)
            {
                buttonText += "G";
            }
            else
            {
                buttonText += "g";
            }
            break;
        case Keys.H:
            if (shift)
            {
                buttonText += "H";
            }
            else
            {
                buttonText += "h";
            }
            break;
        case Keys.I:
            if (shift)
            {
```

```
                buttonText += "I";
            }
            else
            {
                buttonText += "i";
            }
            break;
        case Keys.J:
            if (shift)
            {
                buttonText += "J";
            }
            else
            {
                buttonText += "j";
            }
            break;
        case Keys.K:
            if (shift)
            {
                buttonText += "K";
            }
            else
            {
                buttonText += "k";
            }
            break;
        case Keys.L:
            if (shift)
            {
                buttonText += "L";
            }
            else
            {
                buttonText += "l";
            }
            break;
        case Keys.M:
            if (shift)
            {
                buttonText += "M";
            }
            else
            {
                buttonText += "m";
            }
            break;
        case Keys.N:
            if (shift)
            {
                buttonText += "N";
            }
            else
            {
                buttonText += "n";
            }
            break;
        case Keys.O:
```

```
            if (shift)
            {
                buttonText += "O";
            }
            else
            {
                buttonText += "o";
            }
            break;
        case Keys.P:
            if (shift)
            {
                buttonText += "P";
            }
            else
            {
                buttonText += "p";
            }
            break;
        case Keys.Q:
            if (shift)
            {
                buttonText += "Q";
            }
            else
            {
                buttonText += "q";
            }
            break;
        case Keys.R:
            if (shift)
            {
                buttonText += "R";
            }
            else
            {
                buttonText += "r";
            }
            break;
        case Keys.S:
            if (shift)
            {
                buttonText += "S";
            }
            else
            {
                buttonText += "s";
            }
            break;
        case Keys.T:
            if (shift)
            {
                buttonText += "T";
            }
            else
            {
                buttonText += "t";
            }
```

```
            break;
        case Keys.U:
            if (shift)
            {
                buttonText += "U";
            }
            else
            {
                buttonText += "u";
            }
            break;
        case Keys.V:
            if (shift)
            {
                buttonText += "V";
            }
            else
            {
                buttonText += "v";
            }
            break;
        case Keys.W:
            if (shift)
            {
                buttonText += "W";
            }
            else
            {
                buttonText += "w";
            }
            break;
        case Keys.X:
            if (shift)
            {
                buttonText += "X";
            }
            else
            {
                buttonText += "x";
            }
            break;
        case Keys.Y:
            if (shift)
            {
                buttonText += "Y";
            }
            else
            {
                buttonText += "y";
            }
            break;
        case Keys.Z:
            if (shift)
            {
                buttonText += "Z";
            }
            else
            {
```

```
                        buttonText += "z";
                    }
                    break;
                case Keys.NumPad0:
                    buttonText += "0";
                    break;
                case Keys.NumPad1:
                    buttonText += "1";
                    break;
                case Keys.NumPad2:
                    buttonText += "2";
                    break;
                case Keys.NumPad3:
                    buttonText += "3";
                    break;
                case Keys.NumPad4:
                    buttonText += "4";
                    break;
                case Keys.NumPad5:
                    buttonText += "5";
                    break;
                case Keys.NumPad6:
                    buttonText += "6";
                    break;
                case Keys.NumPad7:
                    buttonText += "7";
                    break;
                case Keys.NumPad8:
                    buttonText += "8";
                    break;
                case Keys.NumPad9:
                    buttonText += "9";
                    break;
                case Keys.Add:
                    buttonText += "+";
                    break;
                case Keys.Subtract:
                    buttonText += "-";
                    break;
                case Keys.OemPeriod:
                    buttonText += ".";
                    break;
                case Keys.OemComma:
                    buttonText += ",";
                    break;
                default:
                    break;
            }
        }


    }
}

public string getText()
{
    return buttonText;
}
```

```
        public bool getIsClicked()
        {
            return isClicked;
        }
    }
}
```

ResetCarsButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class ResetCarsButton : Button
    {
        Vector2 gameBorderTL; //Top Left Corner of game section
        Vector2 gameBorderBR; //Bottom Right Corner of game section
        SpriteFont font;


        public ResetCarsButton(int newHeight, int newWidth, string text, Vector2
position, SpriteFont newfont, MouseState mstate, Vector2 topLeftBorder, Vector2
bottomRightBorder, Color newColor) : base(newHeight, newWidth, text, position, newfont,
mstate, newColor)
        {
            font = newfont;
            gameBorderTL = topLeftBorder;
            gameBorderBR = bottomRightBorder;

            isClicked = false;
        }

        public void Update(GameTime gameTime, MouseState mState, ref Track currentTrack,
ref Car currentCar, ref Car[] trainingCars)
        {
            UpdateClicked(gameTime, mState);

            if (isClicked)
            {
                isClicked = false;

                double carRotation = Math.Atan(currentTrack.getLastLine().getGradient());
                currentCar.reset(currentTrack.getStartPoint(), (float)carRotation,
currentTrack.getCheckpoints());

                for (int i = 0; i < trainingCars.Length; i++)
                {
                    trainingCars[i].reset(currentTrack.getStartPoint(),
(float)carRotation, currentTrack.getCheckpoints());
```

```
                }
            }
        }
    }
}
```

NextGenerationButton:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    class NextGenerationButton : Button
    {
        Vector2 gameBorderTL; //Top Left Corner of game section
        Vector2 gameBorderBR; //Bottom Right Corner of game section
        SpriteFont font;

        public NextGenerationButton(int newHeight, int newWidth, string text, Vector2
position, SpriteFont newfont, MouseState mstate, Vector2 topLeftBorder, Vector2
bottomRightBorder, Color newColor) : base(newHeight, newWidth, text, position, newfont,
mstate, newColor)
        {
            font = newfont;
            gameBorderTL = topLeftBorder;
            gameBorderBR = bottomRightBorder;

            isClicked = false;
        }

        public bool Update(GameTime gameTime, MouseState mState)
        {
            UpdateClicked(gameTime, mState);

            if (isClicked)
            {
                isClicked = false;
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

Track:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    public class Track
    {
        Random rand = new Random();

        //Attributes relating to the game border
        Vector2 gameBorderTL;
        Vector2 gameBorderBR;
        const int SpaceOfPointsFromEdge = 120;

        //Attributes relating to track points which can be discarded at the end
        int numberOfPoints;
        TrackPoint[] trackPoints;
        TrackPoint point0; //the point with the lowest Y value
        TrackPoint[] orderedTrackPoints;
        TrackPoint[] convexHullPoints;


        bool trackPossible = true;

        //attributes relating to the track that are needed for the final track
        List<TrackPoint> finalPoints = new List<TrackPoint>();
        Line lastLine;
        TrackPoint startPoint;
        TrackPoint[] startPointEdges;
        List<Line> insideLineBorder = new List<Line>();
        List<Line> outsideLineBorder = new List<Line>();
        List<TrackPoint> checkpoints = new List<TrackPoint>();

        //Attributes relating to drawing
        SpriteFont font;
        const float trackWidth = 100f;


        public Track(Vector2 newGameBorderTL, Vector2 newGameBorderBR, SpriteFont
newfont)
        {
            //Constructor to generate a normal track with curved lines
            font = newfont;
            gameBorderTL = newGameBorderTL;
            gameBorderBR = newGameBorderBR;
            GenerateTrack();
        }

        public Track(string filename, ref string statustring)
```

```
{
    //Constructor to load a track from a file
    StreamReader reader;
    string filepath;
    string line;
    string[] splitline;
    double X;
    double Y;

    //handling the event that the file that is inputted doesnt exist
    try
    {
        filepath = $@"{filename}/StartPoints.txt";
        reader = new StreamReader(filepath);
    }
    catch (Exception)
    {
        trackPossible = false;
        statustring = "File Not Found";
        return;
    }

    //load startpoint
    line = reader.ReadLine();
    splitline = line.Split(',');
    X = Convert.ToDouble(splitline[0]);
    Y = Convert.ToDouble(splitline[1]);
    startPoint = new TrackPoint(new Vector2((float)X, (float)Y));

    //load startPointEdges
    startPointEdges = new TrackPoint[2];
    line = reader.ReadLine();
    splitline = line.Split(',');
    X = Convert.ToDouble(splitline[0]);
    Y = Convert.ToDouble(splitline[1]);
    startPointEdges[0] = new TrackPoint(new Vector2((float)X, (float)Y));

    line = reader.ReadLine();
    splitline = line.Split(',');
    X = Convert.ToDouble(splitline[0]);
    Y = Convert.ToDouble(splitline[1]);
    startPointEdges[1] = new TrackPoint(new Vector2((float)X, (float)Y));
    reader.Close();

    //loading the FinalPoints
    try
    {
        filepath = $@"{filename}/Finalpoints.txt";
        reader = new StreamReader(filepath);
    }
    catch (Exception)
    {
        trackPossible = false;
        statustring = "File Incomplete";
        return;
    }

    finalPoints = new List<TrackPoint>();
```

```
            while (!reader.EndOfStream)
            {
                line = reader.ReadLine();
                splitline = line.Split(',');
                X = Convert.ToDouble(splitline[0]);
                Y = Convert.ToDouble(splitline[1]);
                finalPoints.Add(new TrackPoint(new Vector2((float)X, (float)Y)));
            }
            reader.Close();

            findLastLine();
            findTrackBorders();
            eliminateOverlapLines();
            connectLineBorders();
            defineCheckpoints();
            statustring = "Successfully Loaded Track";
        }

        public Track(Vector2 newGameBorderTL, Vector2 newGameBorderBR, SpriteFont
newfont, bool DefaultTrack)
        {
            //Constructor to generate the default track (a straight line)
            font = newfont;
            gameBorderTL = newGameBorderTL;
            gameBorderBR = newGameBorderBR;

            generateDefaultTrack();
        }

        public Track(Vector2 newGameBorderTL, Vector2 newGameBorderBR, SpriteFont
newfont, bool DefaultTrack, bool straightLineTrack)
        {
            //Constructor to generate a track with straight lines
            font = newfont;
            gameBorderTL = newGameBorderTL;
            gameBorderBR = newGameBorderBR;

            GenerateStraightLineTrack();
        }

        void generateDefaultTrack()
        {
            //Method to initialise all nesicary attributes for a straight line track
            finalPoints = new List<TrackPoint>();
            finalPoints.Add(new TrackPoint(new Vector2(400, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(600, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(800, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(1000, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(1200, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(1400, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(1600, 540)));
            finalPoints.Add(new TrackPoint(new Vector2(1800, 540)));
            startPoint = new TrackPoint(new Vector2(500, 540));

            startPointEdges = new TrackPoint[2];
            startPointEdges[0] = new TrackPoint(new Vector2(500, 540 - trackWidth / 2));
            startPointEdges[1] = new TrackPoint(new Vector2(500, 540 + trackWidth / 2));
```

```
            lastLine = new Line(finalPoints[finalPoints.Count - 1].getPosition(),
finalPoints[0].getPosition(), true);

            insideLineBorder.Add(new Line(new Vector2(400, 540 - (trackWidth / 2)), new
Vector2(1800, 540 - (trackWidth / 2)), true));
            outsideLineBorder.Add(new Line(new Vector2(400, 540 + (trackWidth / 2)), new
Vector2(1800, 540 + (trackWidth / 2)), true));
            outsideLineBorder.Add(new Line(new Vector2(400, 540 - (trackWidth / 2)), new
Vector2(400, 540 + (trackWidth / 2)), true));
            outsideLineBorder.Add(new Line(new Vector2(1800, 540 + (trackWidth / 2)), new
Vector2(1800, 540 - (trackWidth / 2)), true));

            for (int i = 0; i < finalPoints.Count; i++)
            {
                checkpoints.Add(finalPoints[i]);
            }
            checkpoints[0] = startPoint;


        }

        public void GenerateTrack()
        {
            //Method to generate a track (with curves)

            InitialisePoints(gameBorderTL, gameBorderBR);
            orderTrackpoints();
            trackPossible = checkTrackPossible();
            if (trackPossible == false)
            {
                return;
            }

            grahamScan();
            findFinalTrackPoints();
            trackPossible = checkForNullPoints();
            if (trackPossible == false)
            {
                return;
            }

            excludeOutOfRangePoints();

            findStartPoint();
            //findTrackBorders();
            findTrackBorders();
            eliminateOverlapLines();
            connectLineBorders();
            defineCheckpoints();

        }

        public void GenerateStraightLineTrack()
        {
            //Method to generate a track (straight lines)
            InitialisePoints(gameBorderTL, gameBorderBR);
            orderTrackpoints();
            trackPossible = checkTrackPossible();
```

```
            if (trackPossible == false)
            {
                return;
            }

            grahamScan();
            for (int i = 0; i < convexHullPoints.Length; i++)
            {
                finalPoints.Add(convexHullPoints[i]);
            }

            findStartPoint();
            findTrackBorders();
            eliminateOverlapLines();
            connectLineBorders();
            defineCheckpoints();

        }

        public void Draw(SpriteBatch spriteBatch)
        {
            if (trackPossible == false)
            {
                spriteBatch.DrawString(font, "Track Generation Failed", new
Vector2((gameBorderTL.X + gameBorderBR.X) / 2, (gameBorderTL.Y + gameBorderBR.Y) / 2),
Color.Black);
                return;
            }

            //Draw lines between the points
            for (int i = 0; i < finalPoints.Count - 1; i++)
            {
                spriteBatch.DrawLine(finalPoints[i].getPosition(), finalPoints[i +
1].getPosition(), Color.Black, trackWidth);
            }
            spriteBatch.DrawLine(finalPoints[finalPoints.Count - 1].getPosition(),
finalPoints[0].getPosition(), Color.Black, trackWidth);

            for (int i = 0; i < finalPoints.Count; i++)
            {
                spriteBatch.DrawCircle(finalPoints[i].getPosition(), trackWidth / 2, 32,
Color.Black, trackWidth / 2);
                //spriteBatch.DrawRectangle(new RectangleF(finalPoints[i].getPosition(),
new Size2(trackWidth / 2, trackWidth/2)), Color.Black, trackWidth / 2);

            }

            //Draw the Start Line
            spriteBatch.DrawLine(startPointEdges[0].getPosition(),
startPointEdges[1].getPosition(), Color.White, 5f);



            //for (int i = 0; i < insideLineBorder.Count; i++)
            //{
            //    insideLineBorder[i].Draw(spriteBatch, Color.White, 5);
            //}
```

```
            //for (int i = 0; i < outsideLineBorder.Count; i++)
            //{
            //    outsideLineBorder[i].Draw(spriteBatch, Color.Red, 5);
            //}



        }

        public bool getTrackPossible()
        {
            return trackPossible;
        }

        void InitialisePoints(Vector2 gameBorderTL, Vector2 gameBorderBR)
        {
            //Method to generate a set of random points,
            //find the lowest point
            //and set their polar angles and distances


            //Decide how many points you want to generate - decided randomly
            numberOfPoints = rand.Next(5, 25);
            trackPoints = new TrackPoint[numberOfPoints];

            //generate the random points
            for (int i = 0; i < trackPoints.Length; i++)
            {
                float tempX = rand.Next(Convert.ToInt32(gameBorderTL.X +
SpaceOfPointsFromEdge), Convert.ToInt32(gameBorderBR.X - SpaceOfPointsFromEdge));
                float tempY = rand.Next(Convert.ToInt32(gameBorderTL.Y +
SpaceOfPointsFromEdge), Convert.ToInt32(gameBorderBR.Y - SpaceOfPointsFromEdge));

                trackPoints[i] = new TrackPoint(tempX, tempY);
            }

            //set point0
            point0 = findLowestPoint();

            //Find the polar angles
            for (int i = 0; i < trackPoints.Length; i++)
            {
                trackPoints[i].setPolarAngle(findPolarAngle(trackPoints[i]));
            }

            //Find Distances
            for (int i = 0; i < trackPoints.Length; i++)
            {
                trackPoints[i].setDistance(findDistance(point0,trackPoints[i]));
            }


        }

        TrackPoint findLowestPoint()
        {
            //Method to find the lowest point in trackpoints
            //since y=0 is at the top of the screen in monogame,
```

```
            //it looks for the point with the HIGHEST y value.

            TrackPoint lowestPoint = trackPoints[0];
            for (int i = 1; i < trackPoints.Length; i++)
            {
                if (trackPoints[i].getPosition().Y > lowestPoint.getPosition().Y)
                {
                    lowestPoint = trackPoints[i];
                }
                else if (trackPoints[i].getPosition().Y == lowestPoint.getPosition().Y)
                {
                    if (trackPoints[i].getPosition().X < lowestPoint.getPosition().X)
                    {
                        lowestPoint = trackPoints[i];
                    }
                }
            }

            return lowestPoint;
        }

        double findPolarAngle(TrackPoint point)
        {
            //Method to find the polar angle between point0 and the given point, in
radians

            float yDifference = point0.getPosition().Y - point.getPosition().Y;
            float xDifference = point.getPosition().X - point0.getPosition().X;

            double angle = Math.Atan(yDifference / xDifference);

            if (xDifference < 0)
            {
                angle = angle + Math.PI;
            }

            return angle;
        }

        static public double findDistance(TrackPoint p0, TrackPoint p1)
        {
            //Method to find the distance between two points
            //using pythagoras' theorem a^2 + b^2 = c^2

            float yDifference = p0.getPosition().Y - p1.getPosition().Y;
            float xDifference = p1.getPosition().X - p0.getPosition().X;

            return Math.Sqrt((xDifference * xDifference) + (yDifference * yDifference));
        }

        void orderTrackpoints()
        {
            //Method to order trackpoints so that they can be used in the graham scan

            List<TrackPoint> orderedTrackPointsList = new List<TrackPoint>();
            orderedTrackPointsList.Add(point0);
```

```
            TrackPoint[] sanitisedTrackPoints = removePoint(trackPoints, point0);
//return an array of trackPoints with point0 removed from it

            //they are ordered by polar angle, lowest to highest,
            TrackPoint[] tempSortedPoints = mergeSort(sanitisedTrackPoints);

            //if two points have the same polar angle, then the furthest one is kept
            tempSortedPoints = removePointsWithSamePolarAngle(tempSortedPoints);

            for (int i = 0; i < tempSortedPoints.Length; i++)
            {
                orderedTrackPointsList.Add(tempSortedPoints[i]);
            }

            orderedTrackPoints = orderedTrackPointsList.ToArray();
        }

        TrackPoint[] mergeSort(TrackPoint[] items)
        {
            //This method sorts the Trackpoints by polar angle, lowest to highest

            TrackPoint[] left_half;
            TrackPoint[] right_half;

            //Base case for recursion
            if (items.Length < 2)
            {
                return items;
            }

            int midpoint = items.Length / 2;

            //Do the left half
            left_half = new TrackPoint[midpoint];
            for (int i = 0; i < midpoint; i++)
            {
                left_half[i] = items[i];
            }

            //figure out how big the right half should be
            if (items.Length % 2 == 0)
            {
                right_half = new TrackPoint[midpoint];
            }
            else
            {
                right_half = new TrackPoint[midpoint + 1];
            }

            //fill in hte right half
            int rightIndex = 0;
            for (int i = midpoint; i < items.Length; i++)
            {
                right_half[rightIndex] = items[i];
                rightIndex++;
            }

            //recursion bit
```

```
    left_half = mergeSort(left_half);
    right_half = mergeSort(right_half);

    items = merge(left_half, right_half);

    return items;
}

TrackPoint[] merge(TrackPoint[] list1, TrackPoint[] list2)
{
    TrackPoint[] merged = new TrackPoint[list1.Length + list2.Length];

    int index1 = 0;
    int index2 = 0;
    int indexMerged = 0;

    while (index1 < list1.Length && index2 < list2.Length)
    {
        if (list1[index1].getPolarAngle() < list2[index2].getPolarAngle())
        {
            merged[indexMerged] = list1[index1];
            index1++;
        }
        else if (list1[index1].getPolarAngle() == list2[index2].getPolarAngle())
        {
            if (list1[index1].getDistance() < list2[index2].getDistance())
            {
                merged[indexMerged] = list1[index1];
                index1++;
            }
            else
            {
                merged[indexMerged] = list2[index2];
                index2++;
            }
        }
        else
        {
            merged[indexMerged] = list2[index2];
            index2++;
        }
        indexMerged++;
    }

    while (index1 < list1.Length)
    {
        merged[indexMerged] = list1[index1];
        index1++;
        indexMerged++;
    }

    while (index2 < list2.Length)
    {
        merged[indexMerged] = list2[index2];
        index2++;
        indexMerged++;
    }
```

```
            return merged;
        }

        TrackPoint[] removePoint(TrackPoint[] inputPoints, TrackPoint pointToBeRemoved)
        {
            //Method that removes the specified trackpoint from the specified array

            TrackPoint[] points = new TrackPoint[inputPoints.Length - 1];

            int pointIndex = 0;

            for (int i = 0; i < inputPoints.Length; i++)
            {
                if (trackPoints[i] != pointToBeRemoved)
                {
                    points[pointIndex] = inputPoints[i];
                    pointIndex++;
                }
            }

            return points;
        }

        TrackPoint[] removePointsWithSamePolarAngle(TrackPoint[] inputSortedPoints)
        {
            //Method that goes through the inputted arrays and
            //checks if any of the points have the same polar angle,
            //if they do then the one with the greatest distance is kept
            TrackPoint[] currentSortedPoints = inputSortedPoints;
            int roundingValue = 5; //checks them rounded to 5 d.p.

            for (int i = 0; i < inputSortedPoints.Length - 1; i++)
            {
                if (Math.Round(inputSortedPoints[i].getPolarAngle(), roundingValue) ==
Math.Round(inputSortedPoints[i + 1].getPolarAngle(), roundingValue))
                {
                    if (inputSortedPoints[i].getDistance() > inputSortedPoints[i +
1].getDistance())
                    {
                        currentSortedPoints = removePoint(currentSortedPoints,
inputSortedPoints[i + 1]);
                    }
                    else
                    {
                        currentSortedPoints = removePoint(currentSortedPoints,
inputSortedPoints[i]);
                    }
                }
            }

            return currentSortedPoints;
        }

        bool checkTrackPossible()
        {
            //Method that checks if there are still enough points to make a complete
track
            if (orderedTrackPoints.Length < 3)
```

```
            {
                return false;
            }
            else
            {
                return true;
            }
        }

        void grahamScan()
        {
            //Method for the graham scan algorithm
            Stack pointsStack = new Stack(new TrackPoint[] {orderedTrackPoints[0],
orderedTrackPoints[1], orderedTrackPoints[2]});

            for (int i = 3; i < orderedTrackPoints.Length; i++)
            {
                while (checkLeft(pointsStack.getSTLPoint(), pointsStack.getLastPoint(),
orderedTrackPoints[i]) == false)
                {
                    pointsStack.pop();
                }
                pointsStack.push(orderedTrackPoints[i]);
            }

            convexHullPoints = pointsStack.getStack().ToArray();

        }

        bool checkLeft(TrackPoint A, TrackPoint B, TrackPoint C)
        {
            //Checks if point C is to the left of the line between points A and B

            // Cross-product of lines AB and AC
            double result = ((B.getPosition().X - A.getPosition().X) * (A.getPosition().Y
- C.getPosition().Y)) - ((A.getPosition().Y - B.getPosition().Y) * (C.getPosition().X -
A.getPosition().X));

            if (result < 0)
            {
                return false;
            }

            return true;
        }

        void findFinalTrackPoints()
        {
            //Method that takes the straight line track and adds curves between the
points

            for (int i = 0; i < convexHullPoints.Length - 1; i++)
            {

                TrackPoint[] tempPoints = findBezierCurve(convexHullPoints[i],
findCurvePoint(convexHullPoints[i], convexHullPoints[i + 1]), convexHullPoints[i + 1]);

                for (int j = 0; j < tempPoints.Length; j++)
```

```
                {
                    finalPoints.Add(tempPoints[j]);
                }
            }

        }

        TrackPoint findCurvePoint(TrackPoint point1, TrackPoint point2)
        {
            //Method that finds a random point along the line between the two given
points,
            //then offsets it a random distance perpendicularly from the line

            TrackPoint randomPoint = findRandomPointAlongLine(point1, point2);
            Line line = new Line(point1.getPosition(), point2.getPosition(), true);

            float perpGradient = -1 / line.getGradient();
            Line perpLine = new Line(perpGradient, randomPoint.getPosition());

            float xOffset;
            float curvePointX;
            float curvePointY;

            do
            {
                xOffset = rand.Next(-20, 20);

                curvePointX = randomPoint.getPosition().X + xOffset;
                curvePointY = perpLine.findYValue(curvePointX);
            } while (curvePointY < gameBorderTL.Y || curvePointY > gameBorderBR.Y ||
curvePointX < gameBorderTL.X || curvePointX > gameBorderBR.X);

            return new TrackPoint(curvePointX, curvePointY);
        }

        TrackPoint findMidpoint(TrackPoint point1, TrackPoint point2)
        {
            //Method that find the midpoint between the two given points
            return new TrackPoint(new Vector2((point1.getPosition().X +
point2.getPosition().X) / 2, (point1.getPosition().Y + point2.getPosition().Y) / 2));
        }

        TrackPoint findRandomPointAlongLine(TrackPoint point1, TrackPoint point2)
        {
            //Method that generated a random point along a line between the two given
points

            float X;

            //threshold is so that the generated point isnt too close to the given points
            float threshold = 0.1f; //between 0 and 1
            Line line = new Line(point1.getPosition(), point2.getPosition(), true);
            float xDifference = point2.getPosition().X - point1.getPosition().X;

            float minX = xDifference * threshold + point1.getPosition().X;
            float maxX = xDifference * (1 - threshold) + point1.getPosition().X;

            if (minX < maxX)
```

```
            {
                X = rand.Next(Convert.ToInt32(minX), Convert.ToInt32(maxX));
            }
            else if (minX > maxX)
            {
                X = rand.Next(Convert.ToInt32(maxX), Convert.ToInt32(minX));
            }
            else
            {
                X = findMidpoint(point1, point2).getPosition().X;
            }

            float Y = line.findYValue(X);

            return new TrackPoint(X, Y);
        }

        TrackPoint[] findBezierCurve(TrackPoint p0, TrackPoint p1, TrackPoint p2)
        {
            //Method that takes 3 points and creates a curved line based on those three
points

            List<Line> curveLines = new List<Line>();
            List<Vector2> POIs = new List<Vector2>();

            float tIncrement = 0.01f;

            for (float t = tIncrement; t < 1; t += tIncrement)
            {
                float pA_X = ((p1.getPosition().X - p0.getPosition().X) * t) +
p0.getPosition().X;
                float pA_Y = ((p1.getPosition().Y - p0.getPosition().Y) * t) +
p0.getPosition().Y;
                TrackPoint pA = new TrackPoint(new Vector2(pA_X, pA_Y));

                float pB_X = ((p2.getPosition().X - p1.getPosition().X) * t) +
p1.getPosition().X;
                float pB_Y = ((p2.getPosition().Y - p1.getPosition().Y) * t) +
p1.getPosition().Y;
                TrackPoint pB = new TrackPoint(new Vector2(pB_X, pB_Y));

                curveLines.Add(new Line(pA.getPosition(), pB.getPosition(), true));
            }

            for (int i = 0; i < curveLines.Count - 1; i++)
            {
                POIs.Add(Line.findPOI(curveLines[i], curveLines[i + 1]));
            }

            TrackPoint[] returnArray = new TrackPoint[POIs.Count];
            for (int i = 0; i < returnArray.Length; i++)
            {
                returnArray[i] = new TrackPoint(POIs[i]);
            }

            return returnArray;

        }
```

```
        void  findStartPoint()
        {
            //Method that generates the start-line of the generated track
            startPoint = findRandomPointAlongLine(finalPoints[finalPoints.Count - 1],
finalPoints[0]);
            findLastLine();

            float perpendicularGradient = (-1) / lastLine.getGradient();
            Line perpendicularLine = new Line(perpendicularGradient,
startPoint.getPosition());

            Vector2 edge1 =
perpendicularLine.findPointAtDistance(startPoint.getPosition(), trackWidth / 2, true);
            Vector2 edge2 =
perpendicularLine.findPointAtDistance(startPoint.getPosition(), trackWidth / 2, false);

            startPointEdges = new TrackPoint[] { new TrackPoint(edge1), new
TrackPoint(edge2) };

        }

        void findLastLine()
        {
            //Method that specifies the lastline of the track
            lastLine = new Line(finalPoints[finalPoints.Count - 1].getPosition(),
finalPoints[0].getPosition(), true);
        }

        bool checkForNullPoints()
        {
            //Method that checks if any of the points are null
            //returns false if there are null values and therefore the track is not
possible
            bool returnVal = true;
            for (int i = 0; i < finalPoints.Count; i++)
            {
                if (finalPoints[i].getPosition() == null)
                {
                    returnVal = false;
                    break;
                }

            }

            return returnVal;
        }

        public Vector2 getStartPoint()
        {
            return startPoint.getPosition();
        }

        public TrackPoint[] getStartPointEdges()
        {
            return startPointEdges;
        }
```

```
        public Line getLastLine()
        {
            return lastLine;
        }

        void excludeOutOfRangePoints()
        {
            //Method that gets rid of any points that are outside of the gamescreen
borders

            List<TrackPoint> tempFinalPonts = new List<TrackPoint>();
            for (int i = 0; i < finalPoints.Count; i++)
            {
                bool faulty = false;
                if (finalPoints[i].getPosition().X < gameBorderTL.X)
                {
                    faulty = true;
                }
                else if (finalPoints[i].getPosition().X > gameBorderBR.X)
                {
                    faulty = true;
                }
                else if (finalPoints[i].getPosition().Y < gameBorderTL.Y)
                {
                    faulty = true;
                }
                else if (finalPoints[i].getPosition().Y > gameBorderBR.Y)
                {
                    faulty = true;
                }
                else if (float.IsNaN(finalPoints[i].getPosition().X) ||
float.IsNaN(finalPoints[i].getPosition().Y))
                {
                    faulty = true;
                }

                if (!faulty)
                {
                    tempFinalPonts.Add(finalPoints[i]);
                }
            }

            finalPoints = tempFinalPonts;
        }

        public List<TrackPoint> getFinalPoints()
        {
            return finalPoints;
        }

        void findTrackBorders()
        {
            //Method that find the borders of the track

            Line line;
            Line parallelLine1;
            Line parallelLine2;
            bool Line1Left;
```

```
            for (int i = 0; i < finalPoints.Count - 1; i++)
            {
                line = new Line(finalPoints[i].getPosition(), finalPoints[i +
1].getPosition(), true);

                parallelLine1 = line.findParallelLine((trackWidth / 2), true);
                parallelLine2 = line.findParallelLine((trackWidth / 2), false);
                Line1Left = checkLeft(new TrackPoint(line.getPoint1()), new
TrackPoint(line.getPoint2()), new TrackPoint(parallelLine1.getPoint2()));
                if (Line1Left)
                {
                    insideLineBorder.Add(parallelLine1);
                    outsideLineBorder.Add(parallelLine2);
                }
                else
                {
                    insideLineBorder.Add(parallelLine2);
                    outsideLineBorder.Add(parallelLine1);
                }
            }
            line = new Line(finalPoints[finalPoints.Count - 1].getPosition(),
finalPoints[0].getPosition(), true);
            parallelLine1 = line.findParallelLine(trackWidth / 2, true);
            parallelLine2 = line.findParallelLine(trackWidth / 2, false);
            Line1Left = checkLeft(new TrackPoint(line.getPoint1()), new
TrackPoint(line.getPoint2()), new TrackPoint(parallelLine1.getPoint2()));
            if (Line1Left)
            {
                insideLineBorder.Add(parallelLine1);
                outsideLineBorder.Add(parallelLine2);
            }
            else
            {
                insideLineBorder.Add(parallelLine2);
                outsideLineBorder.Add(parallelLine1);
            }

        }

        public List<Line> getInsideLineBorders()
        {
            return insideLineBorder;
        }

        public List<Line> getOutsideLineBorders()
        {
            return outsideLineBorder;
        }

        void defineCheckpoints()
        {
            //Method that defines the list of checkpoints associated with the track
            checkpoints.Add(startPoint);
            for (int i = 0; i < finalPoints.Count; i++)
            {
                checkpoints.Add(finalPoints[i]);
            }
```

```
        }

        public List<TrackPoint> getCheckpoints()
        {
            return checkpoints;
        }

        public float getTrackWidth()
        {
            return trackWidth;
        }

        void eliminateOverlapLines()
        {
            //this is the code to check for POIs between each line and its subsequent
line and set their lengths so that they dont overlap anymore
            Vector2 POI;
            for (int i = 0; i < insideLineBorder.Count - 1; i++)
            {
                POI = Line.findPOI(insideLineBorder[i], insideLineBorder[i + 1]);
                if (!float.IsNaN(POI.X) && !float.IsNaN(POI.X))
                {
                    insideLineBorder[i].setPoint2(POI);
                    insideLineBorder[i + 1].setPoint1(POI);
                }
            }
            POI = Line.findPOI(insideLineBorder[insideLineBorder.Count - 1],
insideLineBorder[0]);
            if (!float.IsNaN(POI.X) && !float.IsNaN(POI.X))
            {
                insideLineBorder[insideLineBorder.Count - 1].setPoint2(POI);
                insideLineBorder[0].setPoint1(POI);
            }


            for (int i = 0; i < outsideLineBorder.Count - 1; i++)
            {
                POI = Line.findPOI(outsideLineBorder[i], outsideLineBorder[i + 1]);
                if (!float.IsNaN(POI.X) && !float.IsNaN(POI.X))
                {
                    outsideLineBorder[i].setPoint2(POI);
                    outsideLineBorder[i + 1].setPoint1(POI);
                }
            }
            POI = Line.findPOI(outsideLineBorder[outsideLineBorder.Count - 1],
outsideLineBorder[0]);
            if (!float.IsNaN(POI.X) && !float.IsNaN(POI.X))
            {
                outsideLineBorder[outsideLineBorder.Count - 1].setPoint2(POI);
                outsideLineBorder[0].setPoint1(POI);
            }
        }

        void connectLineBorders()
        {
            //Method that connects the point2 of each line with the point1 of the
subsequent line so that there are no gaps between the track's borders
            List<Line> newInsideLineBorder = new List<Line>();
```

```
            for (int i = 0; i < insideLineBorder.Count - 1; i++)
            {
                newInsideLineBorder.Add(insideLineBorder[i]);
                newInsideLineBorder.Add( new Line(insideLineBorder[i].getPoint2(),
insideLineBorder[i + 1].getPoint1(), true));
            }
            newInsideLineBorder.Add(insideLineBorder[insideLineBorder.Count - 1]);
            newInsideLineBorder.Add(new Line(insideLineBorder[insideLineBorder.Count -
1].getPoint2(), insideLineBorder[0].getPoint1(), true));

            List<Line> newOutsideLineBorder = new List<Line>();
            for (int i = 0; i < outsideLineBorder.Count - 1; i++)
            {
                newOutsideLineBorder.Add(outsideLineBorder[i]);
                newOutsideLineBorder.Add(new Line(outsideLineBorder[i].getPoint2(),
outsideLineBorder[i + 1].getPoint1(), true));
            }
            newOutsideLineBorder.Add(outsideLineBorder[outsideLineBorder.Count - 1]);
            newOutsideLineBorder.Add(new Line(outsideLineBorder[outsideLineBorder.Count -
1].getPoint2(), outsideLineBorder[0].getPoint1(), true));

            insideLineBorder = newInsideLineBorder;
            outsideLineBorder = newOutsideLineBorder;
        }

    }
}
```

TrackPoint:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    public class TrackPoint
    {
        //x and y coordinates
        Vector2 position;

        //angle between p0's positive x-axis and the point
        double polarAngle;

        //distance from p0
        double distance;

        public TrackPoint(Vector2 newPos)
        {
            position = newPos;
        }
```

```
        public TrackPoint(float newX, float newY)
        {
            position.X = newX;
            position.Y = newY;
        }

        public Vector2 getPosition()
        {
            return position;
        }

        public double getPolarAngle()
        {
            return polarAngle;
        }

        public void setPolarAngle(double angle)
        {
            polarAngle = angle;
        }

        public double getDistance()
        {
            return distance;
        }

        public void setDistance(double newDistance)
        {
            distance = newDistance;
        }
    }
}
```

Stack:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Random_Track_Generation
{
    class Stack
    {
        List<TrackPoint> stack = new List<TrackPoint>();

        public Stack(TrackPoint[] array)
        {
            for (int i = 0; i < array.Length; i++)
            {
                stack.Add(array[i]);
            }
        }

        public void push(TrackPoint addPoint)
        {
            //Method to push the given value to the top of the stack
            stack.Add(addPoint);
```

```
}

public TrackPoint pop()
{
    //Method to remove the value at the top of the stack and return it
    try
    {
        TrackPoint tempPoint = stack[stack.Count - 1];
        stack.RemoveAt(stack.Count - 1);
        return tempPoint;
    }
    catch (ArgumentOutOfRangeException)
    {

        return null;
    }
}

public TrackPoint popSTL() //pop Second to Last
{
    //Method to remove the value that is 2nd to top of the stack and return it
    try
    {
        TrackPoint tempPoint = stack[stack.Count - 2];
        stack.RemoveAt(stack.Count - 2);
        return tempPoint;
    }
    catch (ArgumentOutOfRangeException)
    {

        return null;
    }
}


public List<TrackPoint> getStack()
{
    return stack;
}

public TrackPoint getLastPoint()
{
    //Method to return the value at the top of the stack
    try
    {
        return stack[stack.Count - 1];
    }
    catch (ArgumentOutOfRangeException)
    {

        return null;
    }
}

public TrackPoint getSTLPoint() //Get second to last point
{
    //Method to return the value that is 2nd to the top of the stack
    try
```

```
            {
                return stack[stack.Count - 2];
            }
            catch (ArgumentOutOfRangeException)
            {

                return null;
            }
        }
    }
}
```

Line:

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

namespace Random_Track_Generation
{
    public class Line
    {
        //attributes of a straight line, modelled after y=mx+c
        float gradient;
        float y_intercept;
        float x_intercept;
        bool fixedLength = false;
        bool inequality = false;
        Vector2 point1;
        Vector2 point2;

        float minX;
        float minY;
        float maxX;
        float maxY;



        public Line(Vector2 p1, Vector2 p2, bool fxdlength)
        {
            //constructor for when two points are given

            //calculate attributes based on data given
            gradient = (p2.Y - p1.Y) / (p2.X - p1.X);
            y_intercept = (gradient * -p1.X) + p1.Y;

            fixedLength = fxdlength;

            if (fixedLength)
            {
                point1 = p1;
                point2 = p2;
```

```
            }
            x_intercept = (-y_intercept) / gradient;
            if (float.IsNaN(x_intercept))
            {
                x_intercept = p1.X;
            }
        }

        public Line(float m, Vector2 p1)
        {
            //constructor for when the lines gradient is provided as well as a point
            gradient = m;

            //calculate attributes based on data given
            y_intercept = (gradient * -p1.X) + p1.Y;
            x_intercept = (-y_intercept) / gradient;
        }

        public Line(float m, float yInt)
        {
            //constructor for when the gradient and y intercept have been provided
            gradient = m;
            y_intercept = yInt;

            //calculate attributes based on data given
            x_intercept = (-y_intercept) / gradient;
        }

        public Line(float m, float yInt, float xInt, bool inequality, float minX, float
maxX, float minY, float maxY)
        {
            //constructor for when attributes have been provided as well as any
inequality values
            gradient = m;
            y_intercept = yInt;
            x_intercept = xInt;
            this.inequality = inequality;
            this.minX = minX;
            this.maxX = maxX;
            this.minY = minY;
            this.maxY = maxY;
        }

        public Line(float m, Vector2 p1, bool inequality, float minX, float maxX, float
minY, float maxY)
        {
            //constructor for when gradient and a point is provided, as well as any
inequality values

            //calculate attributes based on data given
            gradient = m;
            y_intercept = (gradient * -p1.X) + p1.Y;

            if (float.IsInfinity(gradient))
            {
                x_intercept = p1.X;
            }
            else
```

```
            {
                x_intercept = (-y_intercept) / gradient;
            }

            //set inequality values
            this.inequality = inequality;
            this.minX = minX;
            this.maxX = maxX;
            this.minY = minY;
            this.maxY = maxY;
        }

        static public Vector2 findPOI(Line l1, Line l2)
        {
            //Method to find the point of intersection between two lines

            //first calculates it as if both lines are simple straight lines with no
inequalitys or 0/infinite gradients
            float tempY = l2.getYIntercept() - l1.getYIntercept();
            float tempM = l1.getGradient() - l2.getGradient();
            float X = tempY / tempM;
            float Y = l1.findYValue(X);

            //handles l1 being a line where y=a or x=a, not y=mx+c
            if (l1.gradient == 0)
            {
                Y = l1.getYIntercept();
                X = l2.findXValue(Y);
            }
            else if (float.IsInfinity(l1.getGradient()))
            {
                X = l1.getXIntercept();
                Y = l2.findYValue(X);
            }

            //handles l2 being a line where y=a or x=a, not y=mx+c
            if (l2.gradient == 0)
            {
                Y = l2.getYIntercept();
                X = l1.findXValue(Y);
            }
            else if (float.IsInfinity(l2.getGradient()))
            {
                X = l2.getXIntercept();
                Y = l1.findYValue(X);
            }

            //handles l1 being a line where y=a and l2 being a line where x=a
            if (l1.getGradient() == 0 && float.IsInfinity(l2.getGradient()))
            {
                X = l2.getXIntercept();
                Y = l1.getYIntercept();
            }
            if (l2.getGradient() == 0 && float.IsInfinity(l1.getGradient()))
            {
                X = l1.getXIntercept();
                Y = l2.getYIntercept();
            }
```

```
            if (l1.getPoint1() != null && l1.getPoint2() != null && l2.getPoint1() !=
null && l2.getPoint2() != null)
            {
                //handles lines of finite length and/or inequalites
                if (l1.checkPointInRange(new Vector2(X,Y)))
                {
                    if (l2.checkPointInRange(new Vector2(X,Y)))
                    {
                        return new Vector2(X, Y);
                    }
                    else
                    {
                        return new Vector2(float.NaN, float.NaN);
                    }
                }
                else
                {
                    return new Vector2(float.NaN, float.NaN);
                }
            }
            else
            {
                //if both lines are infinitely long
                return new Vector2(X, Y);
            }


        }

        public float getGradient()
        {
            return gradient;
        }

        public float getYIntercept()
        {
            return y_intercept;
        }

        public Vector2 getPoint1()
        {
            return point1;
        }

        public void setPoint1(Vector2 newPoint1)
        {
            point1 = newPoint1;
        }
        public Vector2 getPoint2()
        {
            return point2;
        }

        public void setPoint2(Vector2 newPoint2)
        {
```

```
            point2 = newPoint2;
        }

        public float findYValue(float X)
        {
            //Method that substitues an X value in y=mx+c and gives the Y value
            return gradient * X + y_intercept;
        }

        public float findXValue(float Y)
        {
            //Method that substitues a Y value in y=mx+c gives the X value

            //checks if its a line where x=a
            if (float.IsInfinity(gradient))
            {
                return x_intercept;
            }
            return (Y - y_intercept) / gradient;
        }

        public Vector2 findPointAtDistance(Vector2 point1, float distance, bool
addDistance)
        {
            //Method that finds a point at a specified distance along a line
            //addDistance is to see if we're adding the x value or taking it away

            float xValue;
            float yValue;

            double changeInX = Math.Sqrt((distance * distance) / ((gradient * gradient) +
1));

            if (addDistance)
            {
                xValue = point1.X + (float) changeInX;
            }
            else
            {
                xValue = point1.X - (float)changeInX;
            }

            yValue = findYValue(xValue);

            return new Vector2(xValue, yValue);
        }

        public Line findParallelLine(float displacement, bool addDisplacement)
        {
            //Method to find a line parallel to this line and a given distance away

            Line perpLine1;
            Line perpLine2;
            Vector2 parallelPoint1;
            Vector2 parallelPoint2;

            float perpGradient = (-1) / gradient;
            perpLine1 = new Line(perpGradient, point1);
```

```
            perpLine2 = new Line(perpGradient, point2);
            if (addDisplacement)
            {
                parallelPoint1 = perpLine1.findPointAtDistance(point1, displacement,
true);
                parallelPoint2 = perpLine2.findPointAtDistance(point2, displacement,
true);
            }
            else
            {
                parallelPoint1 = perpLine1.findPointAtDistance(point1, displacement,
false);
                parallelPoint2 = perpLine2.findPointAtDistance(point2, displacement,
false);
            }

            Line parallelLine = new Line(parallelPoint1, parallelPoint2, true);

            return parallelLine;
        }

        public void Draw(SpriteBatch spriteBatch, Color colour, float thickness)
        {
            spriteBatch.DrawLine(point1, point2, colour, thickness);
        }
        public void findRange(out float minX, out float maxX, out float minY, out float
maxY)
        {
            //Method to find the range of this line based on if it has any inequalities
or is of a fixed length between two points
            if (!inequality)
            {
                try
                {
                    if (point1.X <= point2.X)
                    {
                        minX = point1.X;
                        maxX = point2.X;
                    }
                    else
                    {
                        minX = point2.X;
                        maxX = point1.X;
                    }

                    if (point1.Y <= point2.Y)
                    {
                        minY = point1.Y;
                        maxY = point2.Y;
                    }
                    else
                    {
                        minY = point2.Y;
                        maxY = point1.Y;
                    }
                }
                catch (NullReferenceException)
                {
```

```
            minX = float.NaN;
            maxX = float.NaN;
            minY = float.NaN;
            maxY = float.NaN;
        }

    }
    else
    {
        minX = this.minX;
        minY = this.minY;
        maxX = this.maxX;
        maxY = this.maxY;
    }


}

public bool checkPointInRange(Vector2 point)
{
    //Method to check if a point is within the range of this line
    if (checkXInRange(point.X) && checkYInRange(point.Y))
    {
        return true;
    }
    else
    {
        return false;
    }

}

public bool checkXInRange(double X)
{
    //Method to check if the given X value is within the range of this line

    float minX, maxX, minY, maxY;
    findRange(out minX, out maxX, out minY, out maxY);

    if (!float.IsNaN(minX) && !float.IsNaN(maxX))
    {
        if (minX <= X && X <= maxX)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else if (!float.IsNaN(minX) && float.IsNaN(maxX))
    {
        if (minX <= X)
        {
            return true;
        }
        else
```

```
            {
                return false;
            }
        }
        else if (float.IsNaN(minX) && !float.IsNaN(maxX))
        {
            if (X <= maxX)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            return true;
        }
    }

    public bool checkYInRange(double Y)
    {
        //Method to check if the given Y value is within the range of this line
        float minX, maxX, minY, maxY;
        findRange(out minX, out maxX, out minY, out maxY);

        if (!float.IsNaN(minY) && !float.IsNaN(maxY))
        {
            if (minY <= Y && Y <= maxY)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else if (!float.IsNaN(minY) && float.IsNaN(maxY))
        {
            if (minY <= Y)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else if (float.IsNaN(minY) && !float.IsNaN(maxY))
        {
            if (Y <= maxY)
            {
                return true;
            }
            else
            {
                return false;
```

```
                }
            }
            else
            {
                return true;
            }

        }

        public float getXIntercept()
        {
            return x_intercept;
        }



    }


}
```

Car

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using MonoGame.Extended;

using System;
using System.Collections.Generic;
using System.Text;

namespace Random_Track_Generation
{
    class Car
    {
        //attributes of the car
        Texture2D texture;
        Vector2 position;
        float rotation;
        double speed;

        KeyboardState previousKState;

        const double acceleration = 2;
        const double anglularVelocity = 0.04 * (180/Math.PI);
        bool collided;

        //the radar lines and the positions they originate from
        Line[] Lines = new Line[8];
        Vector2[] EdgePositions = new Vector2[8];

        //checkpoints on the track that they must pass through
        List<TrackPoint> checkpoints = new List<TrackPoint>();
        float trackwidth;

        //Used for the AI driving and training
        NeuralNetwork neuralNet;
```

```
        double totalTimeAlive;
        int[] passedCheckpointNumbers;
        float fitness;
        float distanceToNextCheckPoint;

        public Car(Vector2 position, float rotation, Texture2D texture, List<TrackPoint>
checkpoints, float trackwidth, int[] layers)
        {
            //constructor for car when given the layer coposition of the neural network
            this.position = position;
            this.rotation = rotation;
            this.texture = texture;

            previousKState = Keyboard.GetState();
            speed = 0;
            collided = false;

            updateLines();
            this.checkpoints = checkpoints;
            this.trackwidth = trackwidth;

            totalTimeAlive = 0;

            passedCheckpointNumbers = new int[checkpoints.Count];
            for (int i = 0; i < passedCheckpointNumbers.Length; i++)
            {
                passedCheckpointNumbers[i] = 0;
            }

            neuralNet = new NeuralNetwork(layers);
        }
        public Car(Vector2 position, float rotation, Texture2D texture, List<TrackPoint>
checkpoints, float trackwidth)
        {
            //constructor for when layer composition of the ent isnt given, so a default
composition has been set for the neural net
            this.position = position;
            this.rotation = rotation;
            this.texture = texture;

            previousKState = Keyboard.GetState();
            speed = 0;
            collided = false;

            updateLines();
            this.checkpoints = checkpoints;
            this.trackwidth = trackwidth;

            totalTimeAlive = 0;

            passedCheckpointNumbers = new int[checkpoints.Count];
            for (int i = 0; i < passedCheckpointNumbers.Length; i++)
            {
                passedCheckpointNumbers[i] = 0;
            }

            neuralNet = new NeuralNetwork(new int[] { 9, 7, 7, 4 });
        }
```

```
        public Car(Vector2 position, float rotation, Texture2D texture, List<TrackPoint>
checkpoints, float trackwidth, NeuralNetwork neuralNet)
        {
            //Constructor for when a car is to be made with a premade neural network
            this.position = position;
            this.rotation = rotation;
            this.texture = texture;

            previousKState = Keyboard.GetState();
            speed = 0;
            collided = false;

            updateLines();
            this.checkpoints = checkpoints;
            this.trackwidth = trackwidth;

            totalTimeAlive = 0;

            passedCheckpointNumbers = new int[checkpoints.Count];
            for (int i = 0; i < passedCheckpointNumbers.Length; i++)
            {
                passedCheckpointNumbers[i] = 0;
            }

            this.neuralNet = neuralNet;
        }


        public void reset(Vector2 position, float rotation)
        {
            //reset method to reset the car to its original position on a track
            //and to reset all other relevant attributes so they dont interfere with the
next generation
            this.position = position;
            this.rotation = rotation;
            collided = false;
            speed = 0;
            updateLines();
            totalTimeAlive = 0;

            passedCheckpointNumbers = new int[checkpoints.Count];
            for (int i = 0; i < passedCheckpointNumbers.Length; i++)
            {
                passedCheckpointNumbers[i] = 0;
            }
        }

        public void reset(Vector2 position, float rotation, List<TrackPoint> checkpoints)
        {
            //Reset method for reseting to a new track, so the checkpoints are different
            this.position = position;
            this.rotation = rotation;
            this.checkpoints = checkpoints;
            collided = false;
            speed = 0;
            updateLines();
            totalTimeAlive = 0;
```

```
            passedCheckpointNumbers = new int[checkpoints.Count];
            for (int i = 0; i < passedCheckpointNumbers.Length; i++)
            {
                passedCheckpointNumbers[i] = 0;
            }
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            //Drawing ther car

            //Draw the car a different colour if it has already collided with the wall
            if (!collided)
            {
                spriteBatch.Draw(texture, position, null, Color.White, rotation, new
Vector2(texture.Width / 2, texture.Height / 2), 1f, SpriteEffects.None, 0f);
            }
            else
            {
                spriteBatch.Draw(texture, position, null, Color.Blue, rotation, new
Vector2(texture.Width / 2, texture.Height / 2), 1f, SpriteEffects.None, 0f);
            }
        }

        public void Update(GameTime gameTime, GraphicsDevice GraphicsDevice)
        {
            //This update method is for manual driving of the car
            float delta = (float)gameTime.ElapsedGameTime.TotalSeconds;

            if (!collided)
            {
                //Values decided based on car
                int excessWidth = texture.Width / 2;
                int hitBoxSize = texture.Width + 2 * excessWidth;
                Color carColour = new Color(250, 0, 0);

                checkPixelCollision(GraphicsDevice, new Rectangle((int)position.X -
excessWidth, (int)position.Y - excessWidth, hitBoxSize, hitBoxSize), hitBoxSize,
carColour);
                if (collided)
                {
                    speed = 0;
                    return;
                }

                move(delta);
            }
            else if (collided)
            {
                //if the car has hit the wall, we want it to stop
                speed = 0;

            }

        }

        public void Update(GameTime gameTime, GraphicsDevice GraphicsDevice, float[]
distancesToWall, bool TrainingMode, bool autoNextGen)
```

```
        {
            //This Update is for the neural network to drive the car
            float delta = (float) gameTime.ElapsedGameTime.TotalSeconds;

            //in training mode, we may want it to end the generation after a set amount
of time
            //since there are some cars that may just go in circles and never collide
            if (TrainingMode && autoNextGen && totalTimeAlive > 60)
            {
                collided = true;
            }

            if (!collided)
            {
                //Values decided based on car
                int excessWidth = 5;
                int hitBoxSize = texture.Width + 2 * excessWidth;
                Color carColour = new Color(250, 0, 0);

                checkPixelCollision(GraphicsDevice, new Rectangle((int)position.X -
excessWidth, (int)position.Y - excessWidth, hitBoxSize, hitBoxSize), hitBoxSize,
carColour);
                if (collided)
                {
                    speed = 0;

                    fitness = calculateFitness();
                    distanceToNextCheckPoint = findDistanceToNextCheckpoint();
                    return;
                }

                totalTimeAlive += gameTime.ElapsedGameTime.TotalSeconds;

                float[] inputs = new float[9];
                for (int i = 0; i < distancesToWall.Length; i++)
                {
                    inputs[i] = distancesToWall[i];
                }
                inputs[8] = (float) speed;

                aiMove(delta, inputs);

                updateLines();
                checkCheckPointPassed();

                //move(delta);
            }
            else if (collided)
            {
                //if the car has hit the wall, we want it to stop and then evaluate how
well it did
                speed = 0;
                if (TrainingMode)
                {
                    fitness = calculateFitness();
                    distanceToNextCheckPoint = findDistanceToNextCheckpoint();
                }
```

```
    }

}

void move(float delta)
{
    //This is the move method to drive the car manually using WASD

    KeyboardState currentKState = Keyboard.GetState();

    if (currentKState.IsKeyDown(Keys.W))
    {
        speed += acceleration;
    }
    if (currentKState.IsKeyDown(Keys.S))
    {
        speed -= acceleration;
    }
    if (currentKState.IsKeyDown(Keys.A))
    {
        rotation -= (float) anglularVelocity * delta;
        if (rotation <= 2 * Math.PI)
        {
            rotation += (float) (2 * Math.PI);
        }
    }
    if (currentKState.IsKeyDown(Keys.D))
    {
        rotation += (float)anglularVelocity * delta;
        if (rotation >= 2 * Math.PI)
        {
            rotation -= (float) (2 * Math.PI);
        }
    }

    speed = speed * 0.995;


    position.X += (float) (Math.Cos(rotation) * speed * delta);
    position.Y += (float)(Math.Sin(rotation) * speed * delta);
}

public double getSpeed()
{
    return speed;
}

public Vector2 getPosition()
{
    return position;
}

public float getRotation()
{
    return rotation;
}
```

```
        void checkPixelCollision(GraphicsDevice GraphicsDevice, Rectangle HitBox, int
HitBoxSize, Color CarColor)
        {
            //this method is to check the pixels around the car to see if it has hit
(collided with) any green pixels

            //get all of the pixel data around the car
            Color[] colourData = new Color[HitBoxSize * HitBoxSize];
            GraphicsDevice.GetBackBufferData<Color>(HitBox, colourData, 0,
colourData.Length);
            Color[,] newColourData = new Color[HitBoxSize, HitBoxSize];
            int tempx = 0;
            int tempy = 0;

            //converts the 1D array to a 2D array so that it is easier to work with
            for (int i = 0; i < colourData.Length; i++)
            {
                if (tempx == HitBoxSize)
                {
                    tempx -= HitBoxSize;
                    tempy++;
                }
                newColourData[tempx, tempy] = colourData[i];
                tempx++;
            }

            //check for collision
            for (int x = 1; x < newColourData.GetLength(0) - 1; x++)
            {
                for (int y = 1; y < newColourData.GetLength(1) - 1; y++)
                {
                    if (newColourData[x,y] == CarColor)
                    {
                        if (newColourData[x + 1, y] == Color.Green)
                        {
                            collided = true;
                        }
                        else if (newColourData[x - 1, y] == Color.Green)
                        {
                            collided = true;
                        }
                        else if (newColourData[x, y + 1] == Color.Green)
                        {
                            collided = true;
                        }
                        else if (newColourData[x, y - 1] == Color.Green)
                        {
                            collided = true;
                        }
                    }
                }

            }

        }

        void updateLines()
        {
```

```
            updateEdgePositions();

            for (int i = 0; i < 4; i++)
            {
                //float minX, maxX, minY, maxY;
                double rotationalAngle = rotation;
                switch (i)
                {
                    case 0:
                        rotationalAngle = rotation;
                        break;
                    case 1:
                        rotationalAngle = rotation + Math.Atan((double)(texture.Height /
2) / (double)(texture.Width / 2));
                        break;
                    case 2:
                        rotationalAngle = rotation + (Math.PI / 2);
                        break;
                    case 3:
                        rotationalAngle = rotation +  Math.PI -
Math.Atan((double)(texture.Height / 2) / (double)(texture.Width / 2));
                        break;
                }

                float LineGradient = (float)Math.Tan(rotationalAngle);
                if (rotationalAngle == Math.PI/2)
                {
                    LineGradient = float.PositiveInfinity;
                }
                if (rotationalAngle == -Math.PI / 2)
                {
                    LineGradient = float.NegativeInfinity;
                }



                if (EdgePositions[i].X < EdgePositions[i + 4].X) //minimum x value is i
                {
                    if (EdgePositions[i].Y > EdgePositions[i + 4].Y) //minimum y value is
i
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
float.NaN, EdgePositions[i].X, EdgePositions[i].Y, float.NaN);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
EdgePositions[i + 4].X, float.NaN, float.NaN, EdgePositions[i + 4].Y);

                    }
                    else if (EdgePositions[i].Y < EdgePositions[i + 4].Y)
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
float.NaN, EdgePositions[i].X, float.NaN, EdgePositions[i].Y);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
EdgePositions[i + 4].X, float.NaN, EdgePositions[i + 4].Y, float.NaN);
                    }
                    else
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
float.NaN, EdgePositions[i].X, float.NaN, float.NaN);
```

```
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
EdgePositions[i + 4].X, float.NaN, float.NaN, float.NaN);
                    }

                }
                else if (EdgePositions[i].X > EdgePositions[i + 4].X) //minimum x value
is i
                {
                    if (EdgePositions[i].Y > EdgePositions[i + 4].Y) //minimum y value is
i
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
EdgePositions[i].X, float.NaN, EdgePositions[i].Y, float.NaN);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
float.NaN, EdgePositions[i + 4].X, float.NaN, EdgePositions[i + 4].Y);

                    }
                    else if (EdgePositions[i].Y < EdgePositions[i + 4].Y)
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
EdgePositions[i].X, float.NaN, float.NaN, EdgePositions[i].Y);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
float.NaN, EdgePositions[i + 4].X, EdgePositions[i + 4].Y, float.NaN);
                    }
                    else
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
EdgePositions[i].X, float.NaN, float.NaN, float.NaN);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
float.NaN, EdgePositions[i + 4].X, float.NaN, float.NaN);
                    }

                }
                else
                {
                    if (EdgePositions[i].Y > EdgePositions[i + 4].Y) //minimum y value is
i
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
float.NaN, float.NaN , EdgePositions[i].Y, float.NaN);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
float.NaN, float.NaN, float.NaN, EdgePositions[i + 4].Y);

                    }
                    else if (EdgePositions[i].Y < EdgePositions[i + 4].Y)
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
float.NaN, float.NaN, float.NaN, EdgePositions[i].Y);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
float.NaN, float.NaN, EdgePositions[i + 4].Y, float.NaN);
                    }
                    else
                    {
                        Lines[i] = new Line(LineGradient, EdgePositions[i], true,
float.NaN, float.NaN, float.NaN, float.NaN);
                        Lines[i + 4] = new Line(LineGradient, EdgePositions[i + 4], true,
float.NaN, float.NaN, float.NaN, float.NaN);
                    }
```

```
                }

            }

        }

        void updateEdgePositions()
        {
            Double diagonalDistance = Math.Sqrt(((texture.Width / 2) * (texture.Width /
2)) + ((texture.Height / 2) * (texture.Height / 2)));
            Double diagonalAngle = Math.Atan((double) (texture.Height / 2) / (double)
(texture.Width / 2));
            EdgePositions[0] = new Vector2(position.X + (float)(Math.Cos(rotation) *
(texture.Width / 2)), (float)(position.Y + (float)(Math.Sin(rotation) * (texture.Width /
2))));
            EdgePositions[1] = new Vector2(position.X + (float)(Math.Cos(rotation +
diagonalAngle) * diagonalDistance), (float)(position.Y + (float)(Math.Sin(rotation +
diagonalAngle) * diagonalDistance)));
            EdgePositions[2] = new Vector2(position.X + (float)(Math.Cos(rotation +
MathHelper.ToRadians(90)) * (texture.Height / 2)), (float)(position.Y +
(float)(Math.Sin(rotation + MathHelper.ToRadians(90)) * (texture.Height / 2))));
            EdgePositions[3] = new Vector2(position.X + (float)(Math.Cos(rotation +
Math.PI - diagonalAngle) * diagonalDistance), (float)(position.Y +
(float)(Math.Sin(rotation + Math.PI - diagonalAngle) * diagonalDistance)));
            EdgePositions[4] = new Vector2(position.X + (float)(Math.Cos(rotation +
MathHelper.ToRadians(180)) * (texture.Width / 2)), (float)(position.Y +
(float)(Math.Sin(rotation + MathHelper.ToRadians(180)) * (texture.Width / 2))));
            EdgePositions[5] = new Vector2(position.X + (float)(Math.Cos(rotation +
Math.PI + diagonalAngle) * diagonalDistance), (float)(position.Y +
(float)(Math.Sin(rotation + Math.PI + diagonalAngle) * diagonalDistance)));
            EdgePositions[6] = new Vector2(position.X + (float)(Math.Cos(rotation +
MathHelper.ToRadians(270)) * (texture.Height / 2)), (float)(position.Y +
(float)(Math.Sin(rotation + MathHelper.ToRadians(270)) * (texture.Height / 2))));
            EdgePositions[7] = new Vector2(position.X + (float)(Math.Cos(rotation + (2 *
Math.PI) - diagonalAngle) * diagonalDistance), (float)(position.Y +
(float)(Math.Sin(rotation + (2 * Math.PI) - diagonalAngle) * diagonalDistance)));
        }

        public Line[] getCarLines()
        {
            return Lines;
        }

        public Vector2[] getEdgePoints()
        {
            return EdgePositions;
        }

        void aiMove(float delta, float[] inputs)
        {
            //this is the method to make the car drive using its neural network

            float[] outputs = neuralNet.FeedForward(inputs);

            //methods are given in the outputs of the neural net as inputs
            //how much movement is caused will be based on these outputs
            moveForward(outputs[0], delta);
            moveBackward(outputs[1], delta);
```

```
            turnLeft(outputs[2], delta);
            turnRight(outputs[3], delta);

            speed = speed * 0.995;

            position.X += (float)(Math.Cos(rotation) * speed * delta);
            position.Y += (float)(Math.Sin(rotation) * speed * delta);
        }

        void turnLeft(float input, float delta)
        {
            rotation -= (float)anglularVelocity * input * delta;
            if (rotation <= 2 * Math.PI)
            {
                rotation += (float)(2 * Math.PI);
            }
        }

        void turnRight(float input, float delta)
        {
            rotation += (float)anglularVelocity * input * delta;
            if (rotation >= 2 * Math.PI)
            {
                rotation -= (float)(2 * Math.PI);
            }
        }

        void moveForward(float input, float delta)
        {
            speed += acceleration * input;
        }

        void moveBackward(float input, float delta)
        {
            speed -= acceleration * input;
        }

        void checkCheckPointPassed()
        {
            //method to check if the car has passed a checkpoint

            //itterates through all of the checkpoints
            for (int i = 0; i < checkpoints.Count; i++)
            {
                //finds distance between car and checkpoint
                double distance = Track.findDistance(new TrackPoint(position),
checkpoints[i]);

                // if the distance is less than half of the trackwidth, then it has
passed the checkpoint
                if (distance <= (trackwidth / 2) + 10)
                {
                    //This is to check that the car hasnt skipped any checkpoints and to
make sure its not being rewarded for going backwards
                    //it only counts as passing a checkpoint once the previous one has
been passed as well
                    if (i == 0)
                    {
```

```
                        if (passedCheckpointNumbers[passedCheckpointNumbers.Length - 1]
== passedCheckpointNumbers[i])
                        {
                            passedCheckpointNumbers[i]++;
                        }


                    }
                    else if (passedCheckpointNumbers[i - 1] == passedCheckpointNumbers[i]
+ 1)
                    {
                        passedCheckpointNumbers[i]++;
                    }
                }
            }
        }

        float calculateFitness()
        {
            //this method is to calculate the fitness of the car

            //it just takes the sum of the
            float sum = 0;
            for (int i = 0; i < passedCheckpointNumbers.Length; i++)
            {
                sum += passedCheckpointNumbers[i];
            }
            return sum;
        }

        public float getFitness()
        {
            return fitness;
        }

        public bool getCollided()
        {
            return collided;
        }

        int findNextCheckpointIndex(int[] passedCheckpoints)
        {
            //method to find the index of the next checkpoint that the car has to pass
            int lap = passedCheckpoints[passedCheckpoints.Length - 1];
            for (int i = 0; i < passedCheckpoints.Length; i++)
            {
                passedCheckpoints[i] = passedCheckpoints[i] - lap;
            }

            for (int i = 0; i < passedCheckpoints.Length; i++)
            {
                if (passedCheckpoints[i] == 0)
                {
                    return i;
                }
            }

            return 0;
```

```
        }

        float findDistanceToNextCheckpoint()
        {
            //method to find the distance between the next car and the next checkpoint
            int index = findNextCheckpointIndex(passedCheckpointNumbers);
            TrackPoint nextCheckPoint = checkpoints[index];

            return (float) Track.findDistance(new TrackPoint(position), nextCheckPoint);
        }

        public float getDistanceToNextCheckpoint()
        {
            return distanceToNextCheckPoint;
        }

        public NeuralNetwork getNeuralNetwork()
        {
            return neuralNet;
        }

        public void setNeuralNetwork(NeuralNetwork neuralNet)
        {
            this.neuralNet = neuralNet;
        }



    }
}
```

Neural Network:

For the Neural Nework code/implementation I took inspiration from the aforementioned blog post by Kip Parker. However, I have adapted it to fit my needs:

- He build his solution in unity and used some functions that came with unity, I am making my solution in monogame and dont have access to these functions so I have come up with my own.
- An example of one such unity function is his use of “UnityEngine.Random.Range(-0.5f, 0.5f)” which I did not have access to, nor did I use.
- As his inputs to the neural network, he had 5 distances to input, whereas I have given in 9 inputs, including 8 distances and the car’s speed.
- He seems to use a private int[] activations array which I did not use.
- He also used a tanh function as his compression function, whereas I use the sigmoid function
- He has a comparison method for his fitness. I calculate my fitness completely differently to him and I also take into account distances to the next checkpoint as indicates for which nets to mutate

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace Random_Track_Generation
```

```
{
    class NeuralNetwork
    {
        //Attributes of the neural network

        //layers will be in the format {9,7,7,4} as in, 9 neurons in the first layer, 7
neurons in the second layer, 7 in the third layer, and 4 in the last layer
        int[] layers;
        float[][] neurons;
        float[][] biases;
        float[][][] weights;

        static Random rand = new Random();



        public NeuralNetwork(int[] layers)
        {
            //Standard constructor for a neural netwotk where all weights and biases are
initiallised randomly

            this.layers = layers;
            InitialiseNeurons();
            InitialiseBiases();
            InitialiseWeights();
        }

        public NeuralNetwork(string netName, int[] layers, float[][] biases, float[][][]
weights)
        {
            //Constructor for a neural network where biases and weights are given
            this.layers = layers;
            InitialiseNeurons();
            this.biases = biases;
            this.weights = weights;
        }

        public NeuralNetwork(string FilePath, ref string statusString)
        {
            //Constructor for loading a neural network from a file
            if (FilePath == ".txt")
            {
                statusString = "Enter a file Name to\nload the neural net from";
                return;
            }

            if (File.Exists(FilePath) == false)
            {
                statusString = "File does not exist";
                return;
            }
            using (StreamReader reader = new StreamReader(FilePath))
            {
                string layersString = reader.ReadLine();
                string[] layersStrings = layersString.Split(',');

                List<int> layersList = new List<int>();
                for (int i = 0; i < layersStrings.Length; i++)
```

```
                {
                    layersList.Add(Convert.ToInt32(layersStrings[i]));
                }
                layers = layersList.ToArray();

                InitialiseNeurons();
                InitialiseBiases();
                InitialiseWeights();

                for (int i = 0; i < biases.Length; i++)
                {
                    for (int j = 0; j < biases[i].Length; j++)
                    {
                        biases[i][j] = (float)Convert.ToDouble(reader.ReadLine());
                    }
                }

                for (int i = 0; i < weights.Length; i++)
                {
                    for (int j = 0; j < weights[i].Length; j++)
                    {
                        for (int k = 0; k < weights[i][j].Length; k++)
                        {
                            weights[i][j][k] =
(float)Convert.ToDouble(reader.ReadLine());
                        }
                    }
                }

                statusString = "Neural Network Loaded\nSuccessfully";
            }
        }

        void InitialiseNeurons()
        {
            //Method to initialise all neurons,
            //initially their values are all set randomly but theyre changed during each
feedforward
            List<float[]> list = new List<float[]>();
            for (int i = 0; i < layers.Length; i++)
            {
                list.Add(new float[layers[i]]);
            }
            neurons = list.ToArray();
        }

        void InitialiseBiases()
        {
            //Method to initialise all biases to a random value

            List<float[]> biasesList = new List<float[]>();
            for (int i = 0; i < layers.Length; i++)
            {
                float[] layerBiases = new float[layers[i]];

                for (int j = 0; j < layerBiases.Length; j++)
                {
```

```
                    double doubleVal = rand.NextDouble(); //generates random double
between 0 and 1
                    doubleVal = doubleVal - 0.5; //changes the number so that its between
-0.5 and 0.5
                    layerBiases[j] = (float)doubleVal;
                }
                biasesList.Add(layerBiases);
            }
            biases = biasesList.ToArray();
        }

        void InitialiseWeights()
        {
            //Method to initialise all Weights to a random value

            List<float[][]> weightsList = new List<float[][]>();
            for (int i = 1; i < layers.Length; i++)
            {
                List<float[]> listLayerWeights = new List<float[]>();

                for (int j = 0; j < neurons[i].Length; j++)
                {
                    float[] neuronWeights = new float[layers[i - 1]];

                    for (int k = 0; k < layers[i-1]; k++)
                    {
                        double doubleVal = rand.NextDouble(); //generates random double
between 0 and 1
                        doubleVal = doubleVal - 0.5; //changes the number so that its
between -0.5 and 0.5
                        neuronWeights[k] = (float)doubleVal;
                    }
                    listLayerWeights.Add(neuronWeights);
                }
                weightsList.Add(listLayerWeights.ToArray());
            }
            weights = weightsList.ToArray();
        }

        public float[] FeedForward(float[] inputs)
        {
            //put the inputs into the input layer
            for (int i = 0; i < inputs.Length; i++)
            {
                neurons[0][i] = inputs[i];
            }

            //start with the first hidden layer, multiply weights by activations and add
them up, add a bias at the end and then use the compression function
            for (int i = 1; i < layers.Length; i++)
            {
                for (int j = 0; j < neurons[i].Length; j++)
                {
                    float weightedSum = 0f;
                    for (int k = 0; k < neurons[i-1].Length; k++)
                    {
                        weightedSum += weights[i - 1][j][k] * neurons[i - 1][k];
                    }
```

```
                weightedSum += biases[i][j];
                neurons[i][j] = SigmoidFunction(weightedSum);
            }
        }

        //return the last (output) layer
        return neurons[neurons.Length - 1];
    }

    float SigmoidFunction(float input)
    {
        //Method to act as a Sigmoid compression function for any input

        return (float) ((float) 1 / (1 + Math.Pow(Math.E, -input)));
    }

    public void mutate(int percentChanceOfMutation)
    {
        //Method to mutate the neural network by
        //randomly adding or subtracting a random amount from random weights and
biases
        for (int i = 0; i < biases.Length; i++)
        {
            for (int j = 0; j < biases[i].Length; j++)
            {
                int randomNumber = rand.Next(0, 101);
                if (randomNumber <= percentChanceOfMutation)
                {
                    double doubleVal = rand.NextDouble();
                    doubleVal = doubleVal - 0.5;
                    biases[i][j] += (float)doubleVal;

                }
            }
        }

        for (int i = 0; i < weights.Length; i++)
        {
            for (int j = 0; j < weights[i].Length; j++)
            {
                for (int k = 0; k < weights[i][j].Length; k++)
                {
                    int randomNumber = rand.Next(0, 101);
                    if (randomNumber <= percentChanceOfMutation)
                    {
                        double doubleVal = rand.NextDouble();
                        doubleVal = doubleVal - 0.5;
                        weights[i][j][k] += (float) doubleVal;
                    }
                }
            }
        }

    }

    public int[] getLayers()
    {
        return layers;
```

```
}

public float[][] getBiases()
{
    return biases;
}

public float[][][] getWeights()
{
    return weights;
}

public void setBiases(float[][] biases)
{
    this.biases = biases;
}

public void setWeights(float[][][] weights)
{
    this.weights = weights;
}

public NeuralNetwork copyNetwork(NeuralNetwork copyNet)
{
    //Method to that return a copy of this neural network
    for (int i = 0; i < biases.Length; i++)
    {
        for (int j = 0; j < biases[i].Length; j++)
        {
            copyNet.biases[i][j] = biases[i][j];
        }
    }
    for (int i = 0; i < weights.Length; i++)
    {
        for (int j = 0; j < weights[i].Length; j++)
        {
            for (int k = 0; k < weights[i][j].Length; k++)
            {
               copyNet.weights[i][j][k] = weights[i][j][k];
            }
        }
    }

    return copyNet;
}

public void saveNeuralNetwork(string fileName, ref string statusString)
{
    //Method to save this neural network's weights and biases

    if (fileName == "")
    {
        statusString = "Enter a filename to\nsave the Neural Network";
        return;
    }
    string filePath = fileName + ".txt";
    if (File.Exists(filePath))
    {
```

```
                statusString = "A Neural Network with this name Already Exists";
                return;
            }

            using (StreamWriter writer = new StreamWriter(filePath))
            {
                string layersString = "";
                for (int i = 0; i < layers.Length; i++)
                {
                    layersString += layers[i] + ",";
                }
                layersString = layersString.Substring(0, layersString.Length - 1);

                writer.WriteLine(layersString);
                for (int i = 0; i < biases.Length; i++)
                {
                    for (int j = 0; j < biases[i].Length; j++)
                    {
                        writer.WriteLine(biases[i][j]);
                    }
                }

                for (int i = 0; i < weights.Length; i++)
                {
                    for (int j = 0; j < weights[i].Length; j++)
                    {
                        for (int k = 0; k < weights[i][j].Length; k++)
                        {
                            writer.WriteLine(weights[i][j][k]);
                        }
                    }
                }
            }

            statusString = "Neural Network Successfully \nSaved";

        }
    }
}
```

Testing:

Test Plan:

Test ID	Description	Test Data	Expected Results
Tests for Track Generation			
01	Test that clicking the Generate Track Button works	Click on Generate Track button	• A random track is generated and displayed on the game screen. • Track should be with curves, is cyclical, has a start line with a car on top of it
02	Test that clicking the display buttons before tracks are generated works	Click on all of the buttons that are labelled '1', '2' and '3'	• Should not display any track • Status string should display an error message
03	Test that clicking the Generate 3 tracks button works as well as testing that clicking the three display buttons works as well.	Click on Generate 3 Tracks button. Then click on '1' button. Then click on '2' button. Then click on '3' button. Then click on the '1', '2' and '3' buttons again	• 3 random tracks are generated, but not displayed on the game screen. • Upon clicking each of the three display buttons, a different track should be displayed each time. • Tracks should be the same whenever clicking a specific button (Clicking button 1 will show the same track 1 every time until a new set of tracks is generated). • Tracks should be with curves, is cyclical, has a start line with a car on top of it
04	Test that clicking the Generate Straight Line Track Button works	Click on Generate Straight Line track button	• A random track is generated and displayed on the game screen. • Track should have straight edges, is cyclical, has a start line with a car on top of it
05	Test that clicking the Generate 3 Straight Line tracks button works as well as testing that clicking the three display buttons works as well.	Click on Generate 3 Straight Line Tracks button Then click on '1' button. Then click on '2' button. Then click on '3' button. Then click on the '1', '2' and '3' buttons again	• 3 random tracks are generated, but not displayed on the game screen. • Upon clicking each of the three display buttons, a different track should be displayed each time. • Tracks should be the same whenever clicking a specific button (Clicking button 1 will show the same track 1 every time until a new set of tracks is generated). • Tracks should have straight edges, is cyclical, has a start line with a car on top of it
Test Track Saving/Loading			
01	Tests that pressing the save button with no input in the text box works	Click on the save track button with no text in the input box	• Track should not save • Error message should show in the status box
02	Tests that pressing the load button with no input in the text box works	Click on the Load track button with no text in the input box	• Track should not load • Error message should show in the status box
			•

03	Test that the Save Track button works with a curved track	Input a name for the track (TestTrack1) in the input box, then click the save track button	• Success message should show in the status box • File with the inputted name should be created in the programs file directory
04	Test that the Load Track button works with a Curved track	Input the same name as test 03 (TestTrack1) in the input box, then click the Load Track button track	• Success message should show in the status box • Previously saved track should be displayed on the game screen
05	Test that the Save Track button works with a straight Line track	Input a name for the track (TestTrack2) in the input box, then click the save track button	• Success message should show in the status box • File with the inputted name should be created in the programs file directory
06	Test that the Load Track button works with a straight Line track	Input the same name as test 05 (TestTrack2) in the input box, then click the Load Track button track	• Success message should show in the status box • Previously saved track should be displayed on the game screen
07	Test that entering a name for a track that doesn't exists works	Enter just a random string which doesn't correspond to a track e.g. "a" Then press the load track button	• Should display an error message • No track should be loaded
Test Car			
01	Test that Manual Driving for the car works	Press W, then A then S, then D	• When holding W, car should accelerate forwards • When holding A, car should turn left • When holding S, car should accelerate backwards • When holding D, car should turn right
02	Test that the Reset car button works	Drive the car a bit and then press the reset car button	• Car should be put back to the start line • Car should be lined up with the track (not rotated in a random direction) • Car should have a speed of 0 (stopped)
03	Test that the Car's collision works	Drive the car into the green section of the game screen	• Car should stop moving • Car should change colour to indicate that it has collided
04	Test that the Car slows down when not holding the accelerator	Drive the car a bit and then let go	• Car should slow down and eventually come to a stop
Test AI Mode			
01	Test that AI Mode works on a straight track (1 straight line, not a cyclical track)	Press the enable AI mode button	• Car should begin to drive without human input • A button to enable training mode should appear • Car should collide when hitting the edge of the tack

02	Test that AI Mode works on a straight line track	Press the enable AI mode button	• Car should begin to drive without human input • A button to enable training mode should appear
03	Test that manual input isn't registered when in AI mode	While in AI mode, press WASD	• Car should not move in accordance to the human input being given. • Car will still be moving because the neural network is controlling it
04	Test/Showcase that a pre-trained neural network is able to drive around a track	Load the track called FirstLapNetTrack2. Turn on AI mode. Load the Neural Net called MostlyWorkingNet.	• Car should drive a whole lap around the track on its own without crashing
Test Training Mode			
01	Test that Training mode works on a straight track (1 straight line, not a cyclical track)	While in AI mode, Click the button to enable training mode	• Many cars should appear, going in different directions • Cars should begin to drive without human input • Most/all should collide with the edge of the track and stop
02	Test that Training mode works on a straight line track	While in AI mode, Click the button to enable training mode	• Many cars should appear, going in different directions • Cars should begin to drive without human input • Most/all should collide with the edge of the track and stop
03	Test that the next generation button works	When in training mode, Allow all cars to collide and then click the next generation button	• All cars should be reset to the start line position • Some cars should drive differently than in the previous generation, showing that they have mutated • Generation counter should be incremented
04	Test that the Auto-Next generation mode works	When in training mode, click the enable auto next generation mode button	• Once all of the cars collide, they should automatically be reset to the start position • Some cars should drive differently than in the previous generation, showing that they have mutated • Generation Counter should be incremented
05	Full showcase/Test that the training mode works on a straight track	On a straight track, enable training mode and train the cars to the point where they reach the end of the track	• Cars should begin to drive themselves • Cars should get progressively closer to the end • Eventually, there should be a car that reaches the end
Testing Saving and Loading Neural networks			
01	Tests that pressing the save Neural network button with no input in the text box works	While in training mode, click on the save track button with no text in the input box	• Neural Network should not save • Error message should show in the status box

02	Tests that pressing the load Neural network button with no input in the text box works	While in AI mode, Click on the Load Neural Network button with no text in the input box	• Neural Network should not load • Error message should show in the status box
03	Test that the Save Neural Network button works	While in training mode, input a name (NetTest1) for the Neural Network in the input box, then click the save Neural Network button.	• Success message should show in the status box • File with the inputted name should be created in the programs file directory
04	Test that the Load Neural Network button works	While in AI Mode, Input the same name as test 10 (NetTest1) in the input box, then click the Load Neural Network button track	• Success message should show in the status box • Car should start driving differently

Testing showcase:

Please refer to the provided videos to see the evidence of testing

Evaluation:

A Review of my objectives:

11:

I successfully implemented a graphical user interface to show the buttons, tracks and cars. The buttons did not overlap the track/car as they were put to the side of the screen, separate from the game screen. A black race track was clearly visible and there was a clearly distinguishable car on the track

1:

I have successfully managed to randomly generate tracks that are all of different sizes, with different numbers of bends and sharpeness of bends. The track all have a start line and are cyclical. These were implemented using a graham scan (which did use a stack) and Bezier curves.

2 & 3:

As seen in the tesing videos and the GUI plans in my design, I do have a button to generate a random track as well as a button to generate three tracks at once. The display buttons for the three tracks also function perfectly.

4 & 5:

I have met these objectives and the program allow the user to input a file name and then click the save track or load track buttons to save/load their generated tracks. The loaded tracks function exactly the same as generated tracks.

6 & 7:

As seen in the testing, the user is able to use W A S and D to manually drive the car around the track. The car does stop when it hits the edge of the track and the green background. The car also slows down when it is not accelerating and eventually comes to a stop.

The reset Car button also works, resetting the car's position back to the start line of the current track. The car is also orientated as expected and at a halt when resetted.

8, 9, 10 & B1:

After doing this project, I feel that I have a much higher understanding of neural networks and how they function than when I first started the project. I managed to successfully implement a feed forward neural network that is then trained using a genetic algorithm. The outputs of the neural network do drive the car(s) around the track. I did implement a way to determine the fitness of the cars, and I was successfully able to mutate the neural nets of the best cars in each generation. As a result, the training mode does work and it is interesting to see all of the cars driving at the same time, as well as the progression between the generations. The Auto next generation mode is functional and extremely beneficial to the program. It is a fun experience to set the training mode running on auto next generation mode and then leaving it for a while, and coming back to see many cars all going around the track without crashing.

The program is able to save and load neural networks that the user has inputted the name for.

Independent Feedback:

Upon showcasing my program to my peers, I was given some feedback:

"The user interface is simple and easy to understand." –
One of my objectives was to ensure that users could run the program and understand what to do with it. A simple UI was not a part of my objectives, however as a consequence of my other objectives to ensure the buttons of the program do not interfere with the track, it has resulted in a simple UI which is a nice positive to have.

"There is an option to automatically move onto the next generation of cars, which mean the user doesn't have to babysit the application" – this was my goal with this objective since I realized that it can get boring looking at the cars just colliding with the wall for 50 generations, as a result people would probably lose focus and take longer to move to the next generation since they aren't paying attention, so the automatic next generation feature also saves time when training the networks.

Improvements for the future

I was unable to get Cars to drive around a curved track this time since I was unable to generate appropriate borders for the curved track and was therefore unable to get accurate distance as input to the neural networks. Next time, I would venture to attempt to find these border lines for the curved track to allow autonomous driving on curved tracks.

I think a major that would be beneficial to the program if I were to revisit the problem is a way to speed up the game time when training the neural networks so that training them was a quicker process. It is very apparent that there is a large chunk of time that needs to be dedicated every time you want to train a network, and the better you want the network to act, the longer you need to dedicate to it.

"You should be able to load your old nets as a start point for the training" -
I think this is a brilliant idea that wouldn't be that hard to implement either since functionality for saving and loading neural networks is already implemented and fully working. It would also cut down on the time it takes to get a better neural network since you wouldn't be starting from scratch every time.