Project Senna

Proposal

A program-controlled race car learns (using a genetic algorithm) how to find the fastest time round a track, and the user can choose to race any specific generation (skill level) of the AI to race against on each track.

I would find it interesting since I have a passion for cars and racing, I think it would be cool to be able to see how an AI goes about finding time around a track, and how it may act differently to a human trying to learn how to be the fastest. I'm not sure whether to do it in 3D or 2D yet but if I were to do it in 3D it would be a great way to learn the skills within 3D graphics.

It would require a decent knowledge behind the math's and computation behind a self-learning algorithm, since I want to implement it from scratch (without any external libraries). The project would also need specific attention to speed since ideally there would be loads of cars (controllers) learning at the same time, so learning optimization strategies for the language I use will be key.

I do believe that it's a feasible idea, since the complexity can range from simple to extremely complex, i.e. I can build up to the harder complexity stuff when the base project is working, without ruining the original concept.

After doing some brief research online, I found a number of other implementations of this AI experiment : for example **Code Bullet** on YouTube https://www.youtube.com/watch?v=r428O_CMcpI&t=860s&ab_channel=CodeBullet , and numerous examples on github.com. However, I have yet to see a implementation which boasts realistic, more complex controllers such as a **clutch system**, **standard 6 speed gearing**, or **rpm consideration.** Additionally, with a **PLAY against the AI** system I think this project would provide a unique and more engaging spin on this machine learning simulation.

Possible machine learning algorithms

1) Genetic algorithm with Neural Network – this is my number one pick, since it curiously reproduces the learning behaviour of a human which is particularly entertaining to watch. It also allows for an infinite amount of optimization and customization in the way the neural network is modified and improved which makes this a desirable method for development. Finally, this technique is relatively simple to implement as you avoid a large amount of the frankly painful maths attached to other machine learning algorithms.

2) Classic Reinforcement learning (Q-learning) – a very efficient and optimal algorithm for this project. It involves a predetermined environment, a set of actions and a set of rewards which are distributed to each controller to **reinforce** their behaviour if they are seeing success or to **discourage** suboptimal behaviour. However, this technique is particularly complex, requiring huge amounts of very complicated maths and

optimization algorithms to work properly. Therefore, I will reject this approach due to its dubious feasibility at this stage of my computer science development.

SO Genetic Algorithm approach is my algorithm of choice!

Stakeholders

I predict my stakeholders to be students or young adults with an interest in AI and environmental simulations, with particular interest in motorsport. My application will provide a fun and interactive experience with machine learning, which will hopefully inspire other young developers to experiment with the somewhat ominous field of computer science and to create their own simulations. Teachers and tutors could use my project to introduce basic self-learning techniques and the **RACE against** feature should keep students engaged and entertained while learning.

Additionally, when the AI controllers eventually optimize their lap times, their strategies to traverse the track may unveil new or reinforce preexisting notions about motorsport. Therefore, depending on the results of the simulation – my simulation may be utilized by novice motorsport teams to demonstrate optimal racing lines to their racers.

Requirements

Importance Level:

- Critical
- Would be nice
- Polishing touch

Feature	Description	Importance	WHY?
App must have a UI	Track and cars drawn onto the graphics window	Critical	App needs a graphics interface to illustrate the simulation in action
Main menu	Have choice to start a new simulation	Critical	App requires a interface so user can start a simulation
AI controllers	Have each car controlled by a neural network controller which takes in environment data and outputs an action	Critical	Pretty self-explanatory
Option to select a racer	Select a racer should display a window of stats and info about that controller	Would be nice	Not totally necessary but really useful for development and allows interactivity
Random track generation	Option to choose a new **random map,** should use algorithm to generate a new useable track.	Would be nice	Would be very useful and interesting to see how a controllers skill transfers between tracks
Save and load previous controllers	Provides an option for user to save a current controller (racer) or load an old controller	Polishing touch	Again, not essential but a lovely touch and saves time during my development
Cars should have movement physics	Allow interface for controller to move the car how they want, accelerate, brake, shifting etc.	Critical	Very self-explanatory, cars need a way to move around the track
Timing and checkpoint system	Stopwatch system which times AI cars throughout sectors and gives these times back as feedback	Critical	This is the way AI controllers gain feedback and thus learn and improve
Neural network and genetic algorithm	Implementation of neural networks and genetic algorithm used to optimize the	Critical	How the AI **learns** this is perhaps the most critical feature, its required for the simulation to

	network's weights and sizes.		successfully run. Probably the most time consuming step of development
Pause / Play feature	Feature which allows user to stop and start the simulation during runtime	Would be nice	Useful device for development and also for user's wanting to inspect a controller's data at a single point in time.
Leaderboard	Display section which shows a list of the fastest racer's	Polishing touch	Can tell user which AI racer to watch and inspect.

Limitations

Limitations	Explanation
Runtime speed	For large population generations of controllers, depending on individual hardware performance.
The UI	The library I'm using for UI design is slightly limited so the UI may be a little limited but for a simulation this isn't really a big deal.
Car Physics	The car physics in this game will, naturally, be quite inaccurate since modelling the actual physics of the car would be a bit redundant and kill run speeds.

Hardware requirements?

- The only requirement is a computer with enough hardware to run a moderately heavy application – should run smoothly on any computer produced after the 2000's. There are no specific hardware or software requirements since the project will be packaged as a simple .exe file.

Computational Methods

- *What makes this project suitable for a computer:*

Stage One:

Set up and configure the graphics library to run without errors and open a window with correct settings.

Todo:

- Initialise the graphics library
- Draw window to screen
- Allow user to close screen
- Draw at correct size
- Give window a name

Stakeholders should find the programs window easy to use and devoid of issues (i.e. we don't want the program crashing when the user tries to resize it).

Test No.	Description	Test Type	Test Data	Expected	Actual
1a	Running the program after importing the graphics library	-	-	Should run without errors	
1b	Testing whether window opens and closes correctly	-	-	Window should open and then can closes when **X** button is pressed	
1c	Drawing window at specific size	-	Width = 800 Height = 600	Window should create at correct dimensions (i.e. Should change from default size to the specified one).	
1d	Window displays correct window name	-	Name = "Racers"	The widow should display a title of "Racers" at the	

My Development Plan – 12 Small steps

Stage Two:

Create a car object and draw it to the screen. Implement movement and physics for the car: **steering angle, acceleration, speed, braking**. For development purposes allow movement to be controlled by keyboard

Todo:

- Implement and handle physical attributes to car
- Create a Car class
- Draw the car to the screen
- Implement user controls for the car
- Implement braking, and accelerating
- Implement steering and rotate car sprite accordingly

For the stakeholders, the car's physics should be intuitive enough to observe and see what a car is **trying to do** however the physics doesn't need to be implemented with life-like realism

Test no.	Test Description	Test Data	Test Type	Expected Result	Actual Result
2.a	Test keyboard input accelerate and friction	**W key input**	-	Press w key and the car will accelerate in the direction of the sprite	
2.b	Test steering	-	-	Press a key and the car steers left **only** when moving. And visa-versa for right key.	
2.c	Test braking	**S key input**	-	When the s key is pressed **and** the car is moving, the car's velocity should close to a stop	
2.d.	Car sticking to screen	-	Boundary Test	When the car hits the border of the screen it should be blocked as if it has hit an invisible barrier	

Stage Two: **Stage Three:** Create a *track* object which handles track limits, drawing itself and checkpoints (which divide the track into sectors).

1. **Stage Four:** Implement the backend for neural networks, including **matrices, matrix operators,** and a ***network*** **object** which keeps track of **layers, weights** and allows easy abstraction from the inner complexities.
2. **Stage Five:** Attaching a controller to a **car** object so it can use its own neural network to move the car, this involves choosing what data inputs enter the network and deciding the 'bare metal' anatomy of the network.
3. **Stage Six:** Implementing the genetic algorithm for taking 2 controller's and mixings their networks (plus a little mutation) to create a new controller.
4. **Stage Seven:** Upgrading car physics for **barrier detections, gears, rpm tracking**. These new features will be implemented into the outputs of the **neural network**.
5. **Stage Eight:** Adjust population object to keep track of various sector times, lap times and display them to the user. It should also track the best performing car at all times.
6. **Stage Nine: UI OVERHALL** – make UI pretty and implement the features such as the **controller inspector.** I would aim to make the track and cars easier on the eyes also.
7. **Stage Ten:** AI running and testing! Allow time to tweak the inputs, outputs and starting set up of AI **networks** to allow for quicker progress. Attempt to fix (or at least document) any strange behaviour of the **AI.** Create a smooth progression between generations and optimize the genetic algorithm functionality to produce more effective racing.
8. **Stage 11** – Implement the **RACE** feature where users can race against selected AI across a lap (pausing progress of population for that time).
9. **Stage 12 -** Polishing touches to **AI** and physics. Could implement **sound effects** and **particle effects** etc. Mostly testing **AI** to ensure users will be able to watch the generations progress every time they boot up the application. Also, any **ease-of-use** features to be implemented here.

Development Stage 1

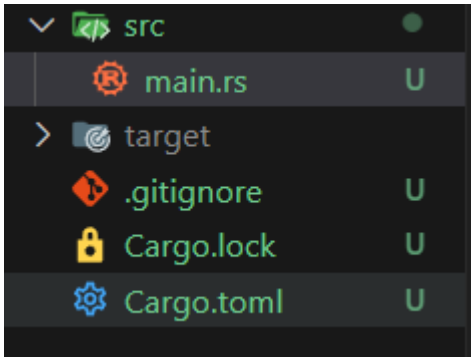
Development environment set up, library initialization and window setup.

My programming language of choice **rust** because of its superior processing speeds and ergonomic syntax. For displaying graphics to the screen, I will be using a relatively high-level graphics library called **macroquad**.

Creating the rust working environment

1. `> cargo new racers` Create a project using **cargo** tool

This is what the project tree looks like after being created



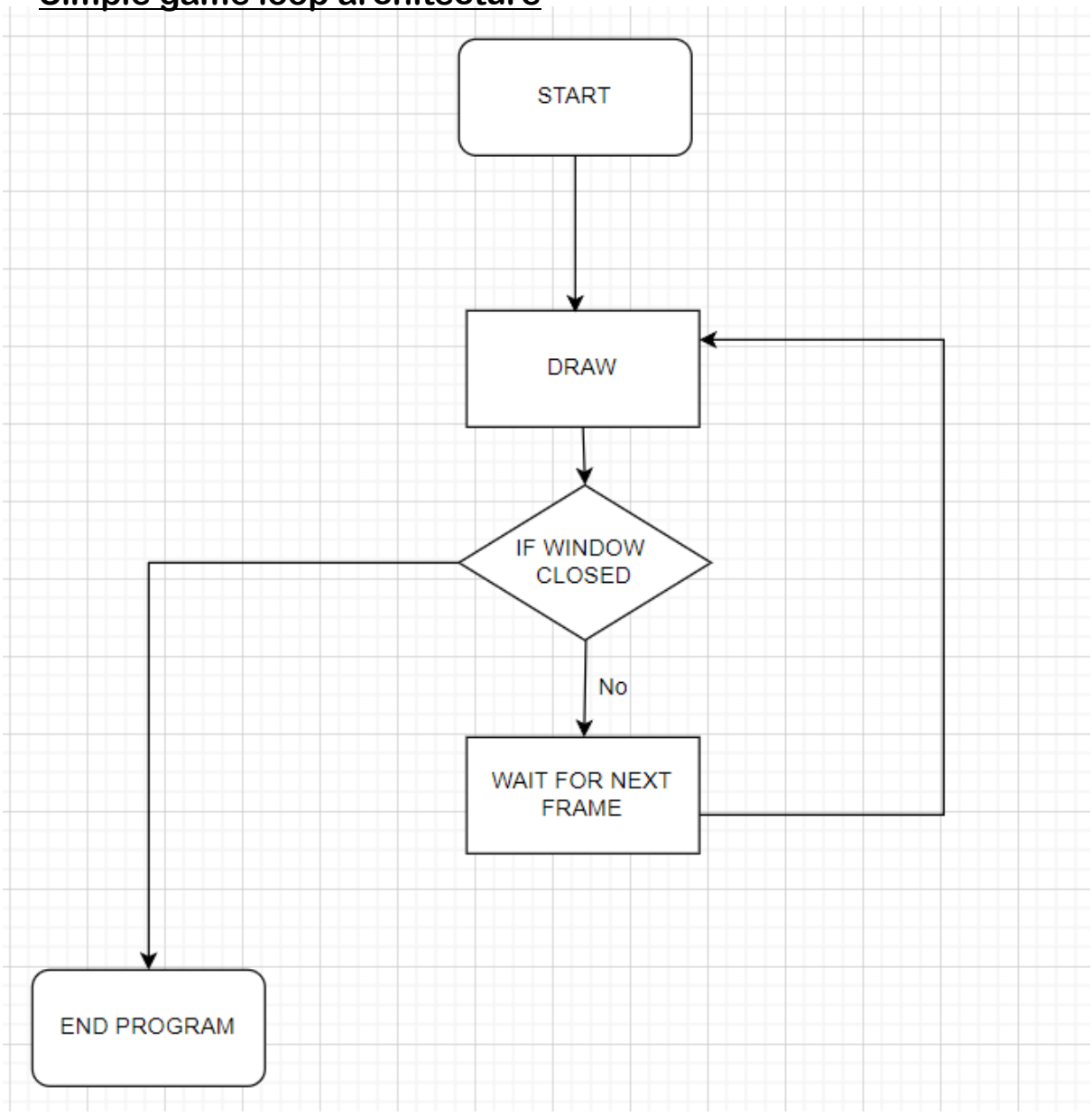
2. **Add the macroquad library**

```
PS C:\Users\fr3nc\CollegeCodingProject\racers> cargo add macroquad
    Updating crates.io index
      Adding macroquad v0.4.13 to dependencies.
             Features:
             - audio
             - backtrace
             - glam-serde
             - log
             - log-rs
             - quad-snd
```

3. **Build to project to fetch all the requirements from webserver**

```
  Updating crates.io index
Downloaded byteorder v1.5.0
Downloaded color_quant v1.1.0
Downloaded quad-rand v0.2.2
Downloaded autocfg v1.3.0
Downloaded cfg-if v1.0.0
Downloaded bitflags v1.3.2
Downloaded adler2 v2.0.0
Downloaded adler v1.0.2
Downloaded simd-adler32 v0.3.7
Downloaded macroquad_macro v0.1.8
Downloaded fdeflate v0.3.4
Downloaded crc32fast v1.4.2
Downloaded version_check v0.9.5
Downloaded once_cell v1.19.0
Downloaded ahash v0.8.11
Downloaded miniz_oxide v0.7.4
Downloaded num-traits v0.2.19
Downloaded slotmap v1.0.7
Downloaded bytemuck v1.18.0
Downloaded miniz_oxide v0.8.0
Downloaded hashbrown v0.13.2
Downloaded png v0.17.13
Downloaded flate2 v1.0.33
Downloaded ttf-parser v0.15.2
Downloaded zerocopy v0.7.35
Downloaded fontdue v0.7.3
Downloaded miniquad v0.4.6
Downloaded glam v0.27.0
Downloaded macroquad v0.4.13
```

Simple game loop architecture



The Boiler Plate Code

This is the code I wrote as a boiler plate which creates a window with any configuration I want.

```
1  use macroquad::prelude::*;
2
3  pub fn window_config() -> Conf {
4      Conf {
5          window_title: "Window Conf".to_owned(),
6          window_height: 600,
7          window_width: 800,
8          window_resizable: false,
9          ..Default::default()
10     }
11 }
12
   ▶ Run | Debug
13 #[macroquad::main(window_config())]
14 async fn main() {
15     loop {
16         next_frame().await;
17     }
18 }
19
```

The **use** statement at the top includes most of the macroquad library into the scope of the main program. This allows me to use objects like **Conf** and functions like **next_frame()**. Also, something to be noted is that the macroquad library requires the use of a asynchronous main function to run, which is why the main function is labeled as **async.**

The **window_config()** function allows me to choose what specifications I want for the window which is displayed, for example the size at the moment would be 600px by 800px.

Now I can change the name of the widow like this:

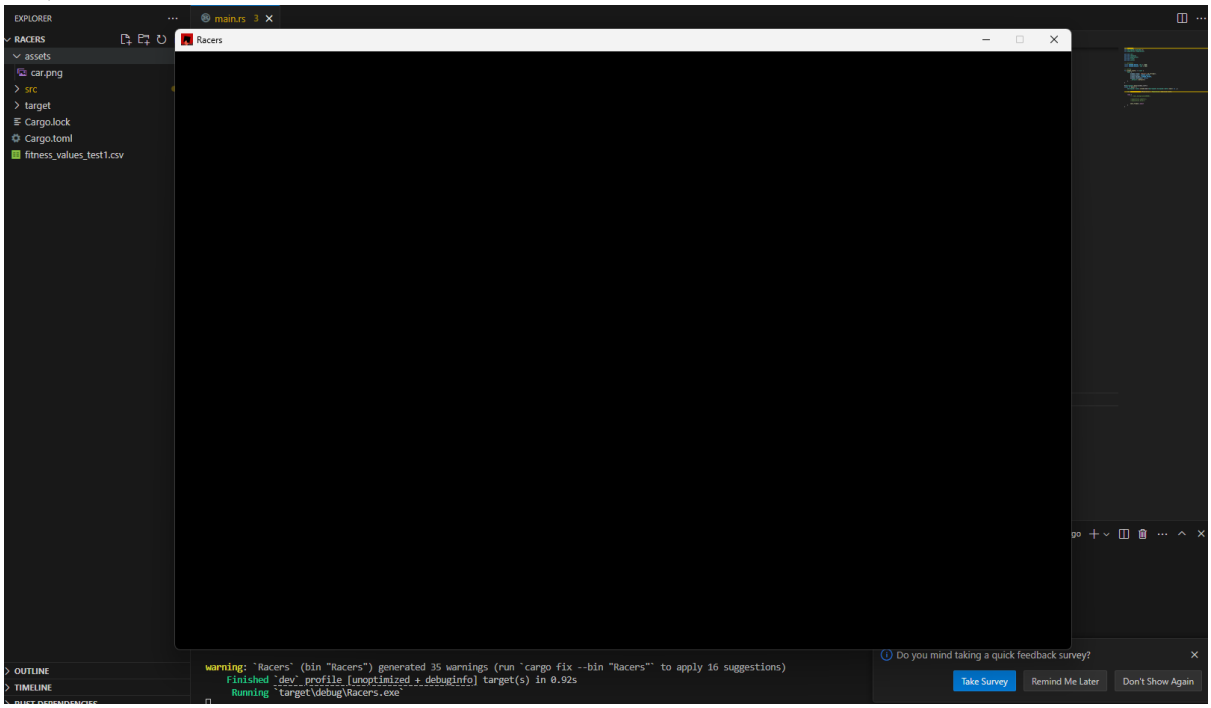
```
Conf {
    window_title: "Racers".to_string(),
```

Stage 1 Conclusion – Reflect on Plan

Testing

Test No.	Description	Test Type	Test Data	Expected	Actual	Video Evidence?
1a	Running the program after importing the graphics library	-	-	Should run without errors	The Program runs without any build or compilation errors **PASS**	**Yes**
1b	Testing whether window opens and closes correctly	-	-	Window should open and then can closes when **X** button is pressed	The window opened and then after a few seconds I closed it with the **X** button **PASS**	**Yes**
1c	Drawing window at specific size	-	Width = 800 Height = 600	Window should create at correct dimensions (i.e. Should change from default size to the specified one).	The window is drawn at the correct size, and altering the width and height values changes the size of the window **PASS**	**No**
1d	Window displays correct window name	-	Name = “Racers”	The widow should display a title of “Racers” at the top	The window is created and given the correct title. **PASS**	**No**

1c, 1d – Photo Evidence



Video evidence for each test that requires it can be found under: **Documentation/Videos/Stage1Testing/[test no.].mp4**

Todo List:

- Initialise the graphics library - **COMPLETE**
- Draw window to screen - **COMPLETE**
- Allow user to close screen - **COMPLETE**
- Draw at correct size - **COMPLETE**
- Give window a name - **COMPLETE**

Development 2

Reminder: **Stage Two**: Create a car object and draw it to the screen. Implement movement and physics for the car: **steering angle, acceleration, speed, braking**. For development purposes allow movement to be controlled by keyboard.

Car Object

Outlining the object, it needs to:

- Move along the screen with velocity, direction and acceleration
- Handle its own rendering, owning its own texture
- Be able to respond to user inputs like steering and accelerating
- Facilitate the use of inputs (weights between 0.0 and 1.0) from a neural network

NOTE: Rust doesn't have classes, but their **structs** are a good enough alternative, so this data is really stored in a **struct** named 'Car'.

Car
Attributes + Velocity: Vector + Direction: Vector + Position: Vector + Acceleration: Vector + angle: Float + steer: Float +texture: Texture2D +rect: Rect +accelerator_input: Input +steering_input: Input +brakes_input: Input
Constants + hitbox_width: float + hitbox_height: float + max_speed: float + max_acceleration: float + steer_weight: float + max_turning_angle: float
Methods + draw()
+ update_pos(x, y)
+ update()
+ keyboard_control()

Here is the class definition for the car obje

Explanation:

The physical quantities like **Velocity**, **direction, acceleration** and **position** are simple to understand, they are what allow the car to move around the screen any which way it wants.

The **texture** attribute is the graphic which is drawn to the screen and is how we display the cars on screen. The **rect** keeps track of the car's hitbox so we can tell when it leaves track, hits something etc… It will also be useful in debugging in the future.

Angle keeps track of the angle the car is directed at (connected to the **direction**) we use the **angle** to tell the renderer how much to rotate the **texture** before drawing to the screen. **Steer** is the change in angle the **inputs** are requesting, however obviously the angle of the car doesn't snap to its steering angle so I will use the **steer** attribute as a target for a linear interpolation.

The **Constants** are self-explanatory apart from the steer_weight which is the factor which the **0.0-1.0 steering input** weight is multiplied by to get the actual angle in radians.

The inputs are basically ways of storing the output of a neural network (which should be between 0 and 1) and are multiplied by scalars later to become interpretable values. It difficult to explain and I will come to the use later, when I make the NN. I will also outline the **input** structure next.

```rust
pub struct Car {
    // Physics variables
    // -- Vectors
    velocity: Vec2,
    direction: Vec2,
    position: Vec2,
    acceleration: Vec2,

    // -- Scalar
    angle: f32,
    steer: f32,

    // Graphics
    texture: Texture2D,
    rect: Rect,

    // inputs for controllers
    accelerator_input: Input,
    steering_input: Input, // radians
    brakes_input: Input,
}
```

You should be able to compare this to the previously shown class diagram to spot the implementation of each attribute.

I also coded the **Car** class's constructor, and it is very intuitive to understand, here it is:

```rust
pub fn new() -> Self {
    // default car setup
    let mut car: Self = Self {
        texture: Texture2D::from_file_with_format(
            include_bytes!("../assets/car.png"),
            format: None,
        ),

        // Defining Vector
        position: Vec2::new(x: crate::WINDOW_WIDTH as f32 / 2.0, y: crate::WINDOW_HEIGHT as f32 / 2.0),
        velocity: Vec2::ZERO,
        direction: Vec2::ZERO,
        acceleration: Vec2::ZERO,

        // Scalar
        angle: 0.0,
        steer: 0.0,

        // other
        rect: Rect::new(x: 0.0, y: 0.0, w: Car::HITBOX_WIDTH, h: Car::HITBOX_HEIGHT),

        // inputs
        accelerator_input: Input::new_default(),
        brakes_input: Input::new_default(),
        steering_input: Input {min: -1.0, max: 1.0, weight: 0.0, default: 0.0},
    };
    car.direction = Vec2::from_angle(car.angle);
    return car;
```

Note:

- I used the macroquad Texture2D structure's functionality to import an asset I found on google for a racing car
- For now, we place the car in the middle of the screen
- We define the rectangular hitbox as macroquad's predefined **rect** class which offers plenty of useful functionality
- The inputs are initialized as **Input** structs which are basically restricted values which can be set within a certain bound.
- We set the car direction vector by creating a unit vector in direction of the car's angle.
- Most other physical quantities are set to zero or other arbitrary values

Deep dive into the methods

```rust
pub fn draw(&self) {
    // just draws to the screen
    let w: f32 = self.rect.w;
    let h: f32 = self.rect.h;
    let x: f32 = self.rect.x;
    let y: f32 = self.rect.y;
    let params: DrawTextureParams = DrawTextureParams{
        dest_size: Some(Vec2::new(x: w, y: h)),
        source:  None,
        flip_x: false,
        flip_y: false,
        rotation: self.angle + PI/2.0,
        pivot: None,
    };
    draw_texture_ex(&self.texture, x, y, color: WHITE, params);
}
```

The **draw** function collects the x, y coordinates of the car's hitbox (which is defined as a Rect class, which is a class included in **macroquad**) then it also retrieves the width and height of the hitbox. These values are passed into draw_texture_ex function which draws the car to the screen. Note we also give the function parameters for rotation since the car should rotate on the screen, but we have to offset the rotation by PI/2 radians or 90 degrees to account for the sprite being at a 90-degree angle. Apart from that it's a pretty simple function.

```rust
pub fn update_pos(&mut self, x: f32, y: f32) {
    // way to safely change position
    let x: f32 = clamp(value: x, min: 0.0, max: WINDOW_WIDTH as f32);
    let y: f32 = clamp(value: y, min: 0.0, max: WINDOW_HEIGHT as f32);

    self.position = Vec2::new(x, y);
    self.rect.x = x;
    self.rect.y = y;
}
```

The **update_pos** function takes 2 new coordinates to set the car's position to, its essentially a *setter* i.e. an public interface to the class's attributes. It also serves the function of clamping the car to the screen and updating the car's hitbox **rect** as well as its actual position vector.

Update Function

The update function is the main function which controls the behaviour of the car's physics. It should apply all the physics onto the car object like acceleration, velocity, friction, update position and steering. The steering is the most complex part of the algorithm so I will go into more detail on that.

My update function algorithm in broken English code:

```
Function Update()
      Calculate delta time, to apply to all physical calculations
      Calculate how much to steer car
      Add steer delta to car's angle
      Calculate new direction vector for car
      Calculate car's acceleration from accelerator input and set
      its direction to the car's direction vector
      Add velocity to the cars position vector
      Update the car's position using the method
End of function
```

My update function algorithm pseudo code:

```
public prodecure update(dt)

    dt = get_frame_time()

    steer = steering_input * steering_weight
    angle += steer * dt;

    direction = new_vector_from_angle(angle)

    acceleration = direction * (accelerator_input * MAX_ACCELERATION)
    velocity += aceleration * dt

    // friction
    normal_fric = -self.velocity * FRICTON_COEFFIENT

    velocity += normal_fric * dt
    position += velocity * dt
    update_pos(position.x, position.y)
end prodecure
```

After planning the method in pseudo code I wrote it in rust, note that the key difference between the rust code and the pseudo code is that we need to call methods and attributes using the **self**-keyword and a few variables are given slightly different names for cleaner code. Here it is:

Rust code:

```
pub fn update(&mut self) {
    let dt: f32 = get_frame_time();

    self.steer = self.steering_input.weight * Car::STEER_WEIGHT;
    let new_angle: f32 = self.angle + self.steer;

    self.direction = Vec2::from_angle(new_angle);
    self.angle = new_angle;

    self.acceleration = self.direction * (self.accelerator_input.weight * Car::MAX_ACC);

    self.velocity += self.acceleration * dt;

    let normal_fric: Vec2 = -self.velocity * FRIC_COEF_ROAD;

    // apply frictions
    self.velocity += (normal_fric) * dt;
    self.position += self.velocity * dt;
    self.update_pos(self.position.x, self.position.y);
}
```

For now, this update code should be robust enough, however I fully anticipate modifying this function **a lot** since it contains the main bulk of the car's physical code.

Modifying the Main function

Now we have created the blueprint for a car object, lets create one and update and draw it from the main function.

Note: I aim to have only 2 public interfaces to the car class (apart from its creation), which are the **update** and **draw** functions. i.e. the only functions which should be called on the class from outside of itself are the **update** and **draw** functions.

Here is the pseudo code and real code for the main function.

```
1  procedure main()
2      car = new Car(); // creates default car object
3      while (True):
4          colour_background(GREEN)
5          car.update()
6          car.draw()
7      end while
8  end procedure
```

```
22  #[macroquad::main(window_conf)]
23  async fn main() {
24      let mut car1: Car = Car::new();
25      loop {
26          clear_background(color: GREEN);
27          car1.update();
28          car1.draw();
29
30          next_frame().await
31      }
32  }
```

Note: The rust code is slightly different, but its just rust's weird implementation of an infinite loop, and the first scary line is just a bit of macroquad boiler plate (since macroquad requires asynchronous rust to be enabled to run).

Also, I'm fully aware that the main function will change a lot during development and this version is very temporary and mainly for development purposes, but I think it's important to show what it looks like now since its really the foundation of the project.

Finally, the user input

After staring at a stationary car on the screen for a while, I realised that I wanted to implement a way for a **human** user to interact and control the car. Although the car's will eventually be fully controlled by **AI,** I need to be able to test out the physics before moving onto that implementation, thus *user input* needs to be implemented.

Since its quite an intuitive and temporary method I won't go into huge detail on it and I will give it to you in rust code:

```rust
fn keyboard_control(&mut self) {
    // get delta time
    let mut steering: i32 = 0;
    let mut accelerating: i32 = 0;

    // loop through keys
    for key: KeyCode in get_keys_down() {
        if (key == KeyCode::Up) {
            self.accelerator_input.weight = 1.0;
            accelerating = 1;
        }
        if (key == KeyCode::Down) {
            self.accelerator_input.weight = -1.0;
            accelerating = 1;
        }
        if (key == KeyCode::Left) {
            self.steering_input.weight = -1.0;
            steering = 1;
        }
        if (key == KeyCode::Right) {
            self.steering_input.weight = 1.0;
            steering = 1;
        }
    }

    if (steering == 0) {
        self.steering_input.weight = 0.0;
    }
    // clamping speeds and steering
    if (self.velocity.length() > Car::MAX_SPEED) {
        self.velocity = ((self.velocity) / self.velocity.length()) * Car::MAX_SPEED;
    }
    if (accelerating == 0) {
        self.accelerator_input.weight = 0.0;
    }
} fn keyboard_control
```

Essentially, for now we are imitating the eventual **neural network's** inputs using the keyboard for development purposes. Also note that we have two Boolean variables called ***steering*** *and* ***accelerating*** which indicates whether the accelerator (forward key) or the steering keys where pressed and if they weren't set the inputs to **zero**.

```rust
// clamping speeds and steering
if (self.velocity.length() > Car::MAX_SPEED) {
    self.velocity = ((self.velocity) / self.velocity.length()) * Car::MAX_SPEED;
}
```

And this section of code caps the speed of the velocity to the car's predefined max speed.

Now we are ready to test the car physics

Let's add the user input method into the **update** method

```
pub fn update(&mut self) {
    let dt: f32 = get_frame_time();

    self.keyboard_control();

    self.steer = self.steering_input.weight * Car::STEER_WEIGHT;
    let new_angle: f32 = self.angle + self.steer;
```

And run the code…

When I ran the code, I noticed that the car would spin instantly when the arrow keys were pressed instead of smoothly rotating, so I added a linear interpolation function to smooth out the rotation.

The Fix

```
self.steer = self.steering_input.weight * Car::STEER_WEIGHT;
let new_angle: f32 = self.angle + self.steer;

self.direction = Vec2::from_angle(new_angle);
self.angle = lerp(val1: self.angle, val2: new_angle, weight: dt * 6.0);
```

The highlighted line is the modified line. The linear interpolation function requires a factor to say how fast the value should reach the target value, with a bit of tweaking I found that using **deltatime * 6.0** was a good factor for smooth steering.

The video of the running car physics is in "**videos/s2-carrunning.mp4"**

Link: (Videos\s2-carrunning.mp4**)**

The final thing missing now for the car physics is a braking system. The car should take its braking input and transfer it into a force opposing the car's motion.

Formula

The brake mechanism will be another frictional component to negate velocity, so we will use a formula which creates an opposing braking frictional acceleration.

Brake_friction = -velocity * brake_input * brake factor

The brake factor is a variable used to tweak how powerful the frictional force on the velocity is.

The code:

```
let brake_friction: Vec2 = -self.velocity * self.brakes_input.weight * Car::BRAKING_FACTOR;
self.velocity += brake_friction * dt;
```

The result:

Videos\s2-braketest.mp4

Testing Stage 2

Test no.	Test Description	Test Data	Expected Result	Actual Result	Pass

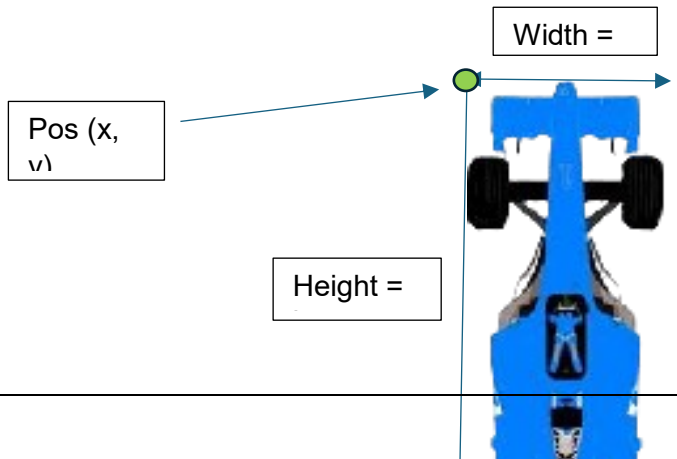
2.a	Test keyboard input accelerate and friction		Press w key and the car will accelerate in the direction of the sprite	It does accelerate and friction eventually brings the car to a stop	✓
2.b	Test steering		Press a key and the car steers left **only** when moving. And visa-versa for right key.	The steering works as expected, turning its velocity and rotation of sprite	✓
2.c	Test braking		When the s key is pressed **and** the car is moving, the car's velocity should close to a stop	The car does brake when the s key is pressed, it slows down quite slowly but the feature does work	✓
2.d.	Car sticking to screen		When the car hits the border of the screen it should be blocked as if it has hit an invisible barrier	The car does stop but doesn't act quite as expected, the car sprite can go fully off the screen on the rhs of the screen but on the lhs of the screen it stops as expected.	✗

Results:

The program passed ¾ tests which is a solid result, the failed test is quite an easy fix. Here's the current code:

```rust
pub fn update_pos(&mut self, x: f32, y: f32) {

    // way to safely change position
    let x: f32 = clamp(value: x, min: 0.0, max: WINDOW_WIDTH as f32); // keep the car on the screen
    let y: f32 = clamp(value: y, min: 0.0, max: WINDOW_HEIGHT as f32);
```

The problem is the sprite's position is the top left corner of that sprite:



Recap

Stage Three: Create a *track* object which handles track limits, drawing itself and checkpoints (which divide the track into sectors)

The solution:

Pseudo Code

```
pos.x = clamp(0, ScreenWidth - w)
pos.y = clamp(0, ScreenHeight - h)
```

Rust Code

```rust
pub fn update_pos(&mut self, x: f32, y: f32) {

    // way to safely change position
    let x: f32 = clamp(value: x, min: 0.0, max: WINDOW_WIDTH as f32 - Car::HITBOX_WIDTH); // keep the car on the screen
    let y: f32 = clamp(value: y, min: 0.0, max: WINDOW_HEIGHT as f32 - Car::HITBOX_HEIGHT);
```

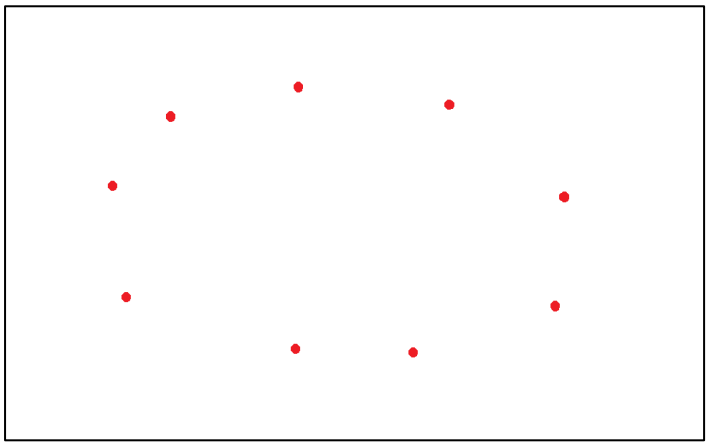
End of Stage 2

Development Stage 3

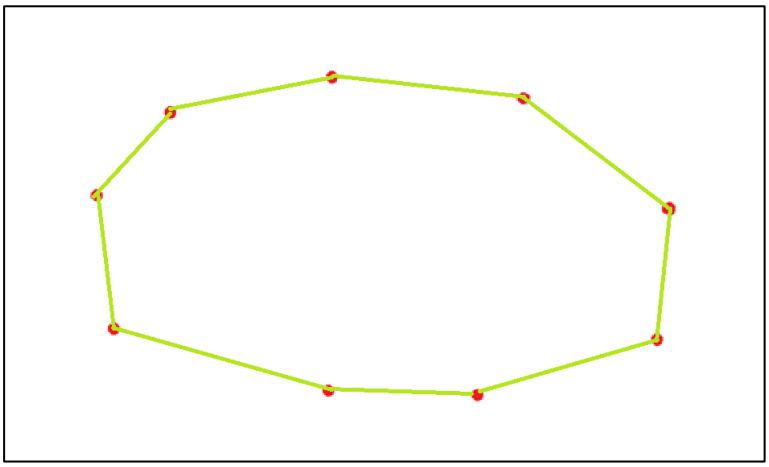
What is a track?

I think a track object can be defined by a series of points and then we can connect them with lines. To make it look more like a standard track, we can then draw these lines with a large thickness (thickness = road width)

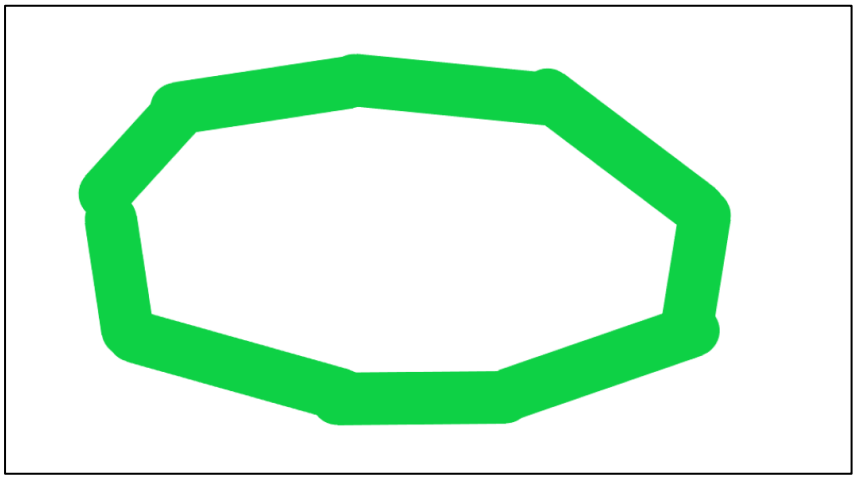
1. Series of dots



2. Connect the dots

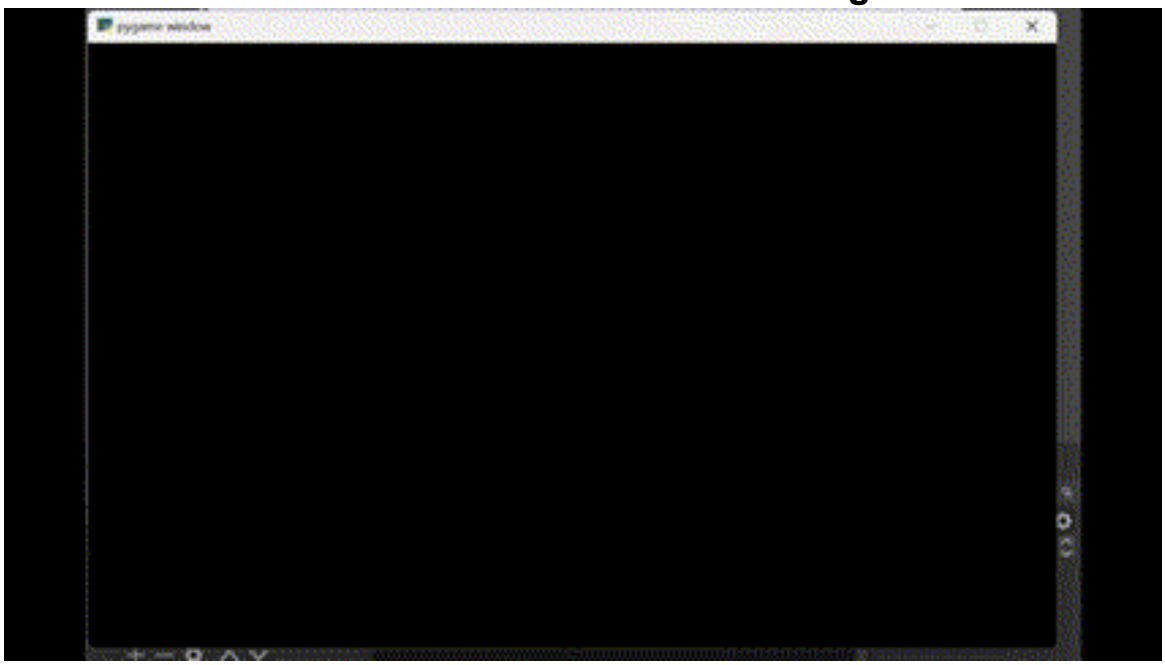


3. Thicker line



I made a little animation in python to visualise and wrap my head around how this would work. The file can be found in

Documentation/Videos/TrackDemonstrationVid.gif



So now I know what to do, however to define a track as a set of coordinates we need to know the size of the screen, which we should keep as a constant throughout the project.

These values are predefined in the main.rs file but let's reiterate them in this document.

```
// constants
const WINDOW_WIDTH: i32 = 1200;
const WINDOW_HEIGHT: i32= 800;
```

So, our track must lie in a plane of pixels **1200** wide and **800** high.

In this stage of development all I need to do is create a track object so we will save the potential random track generation for later. For now, I will predefine a test track with a set of coordinates.

I think a set of 20 points should be able to define any track I will ever make so I chose the set of points to be an array with size **20.**

Track
Attributes: + points_set: int[2][20]

Current Track class diagram

I created a test track in paint and copied the coordinates into my track.rs file as a constant array.

Test Track 1:

Code defining the track

track.rs

```rust
use macroquad::prelude::*;

pub const test_track1: [[i32; 2]; 20] = [
    [507, 142],
    [654, 140],
    [782, 139],
    [851, 165],
    [923, 209],
    [958, 292],
    [965, 394],
    [948, 493],
    [879, 566],
    [774, 585],
    [682, 597],
    [565, 621],
    [479, 530],
    [405, 438],
    [314, 427],
    [205, 425],
    [139, 338],
    [170, 212],
    [272, 165],
    [391, 145],
];

```

Designing Track Object

Now we have a way of defining a track lets design the class around it.

The rust structure in code based off of the class diagram from previous page:

```rust
1 implementation
pub struct Track {
    points_set: [[i32; 2]; 20],
}

impl Track {
    pub fn new(points_set: [[i32; 2]; 20]) -> Self {
        return Self { points_set };
    }
}
```

The first functionality I can think of is **drawing the track** so let's design that function.

The algorithm

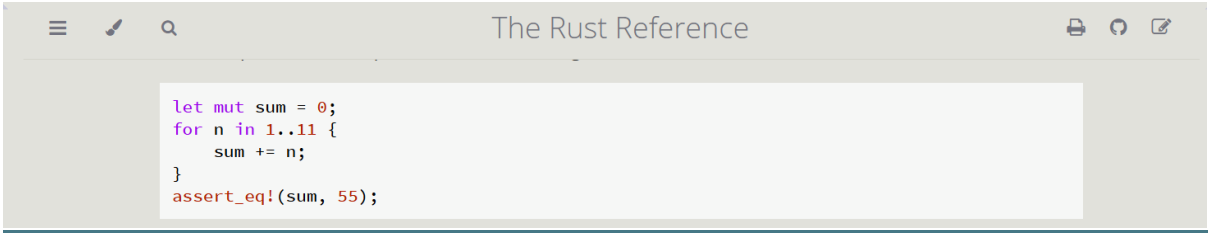
In English the draw algorithm needs to loop through the set of points, draw lines between them of a given thickness (track width) making sure to draw a line between the first and last points. That is the beauty of it, it should be simple.

Pseudocode

```
procedure draw_track()
    for i=0 to (points.length() - 1) Then
        draw_line_inbetween(points[i], points[(i+1) % (point.length())])
        // the modulus in the line above allows the loop to join
        // the first and last points with the others
    next i
end procedure

```

The only bit of complexity here is the modulus described in the code above. It basically means that when the variable **i** reaches the final index of the array **i+1** will loop back to **0,** which essentially means that when the loop finishes it will have joined the first and last point as well as all the others.

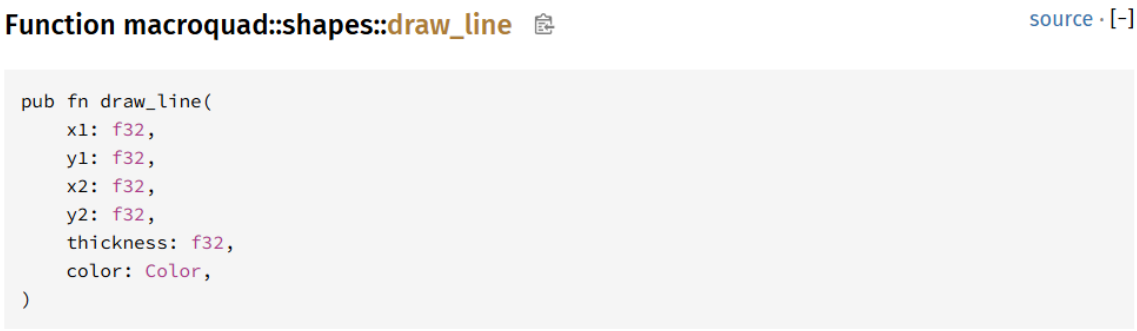
When writing the **rust** code, I had to search up how to use a **for loop** in rust and found the answer on https://doc.rust-lang.org/reference/expressions/loop-expr.html



Importantly, the range **1..11** loops through 1 to 10 so it's **not** inclusive on the upper end.

I also had to look up the draw_line function in macroquad docs:

https://docs.rs/macroquad/latest/macroquad/shapes/fn.draw_line.html



Rust code:

```
1 implementation
pub struct Track {
    points_set: [[i32; 2]; 20],
    track_width: f32,
}

impl Track {
    pub fn new(points_set: [[i32; 2]; 20], track_width: f32) -> Self {
        return Self {
            points_set,
            track_width,
        };
    }

    pub fn draw(self) {
        for i: usize in 0..self.points_set.len() {
            let p1: [i32; 2] = self.points_set[i];
            let p2: [i32; 2] = self.points_set[(i + 1) % self.points_set.len()];
            let x1: f32 = p1[0] as f32;
            let y1: f32 = p1[1] as f32;
            let x2: f32 = p2[0] as f32;
            let y2: f32 = p2[1] as f32;
            draw_line(x1, y1, x2, y2, thickness: self.track_width, color: tarmac_colour);
        }
    }
}
```

You may notice a new class attribute **track_width** which I added to the track class as a float value. I also added it in the constructor function. Also, I added a constant tarmac colour at the top of the program here:

```
pub const tarmac_colour: Color = color_u8!(35, 35, 35, 255);
```

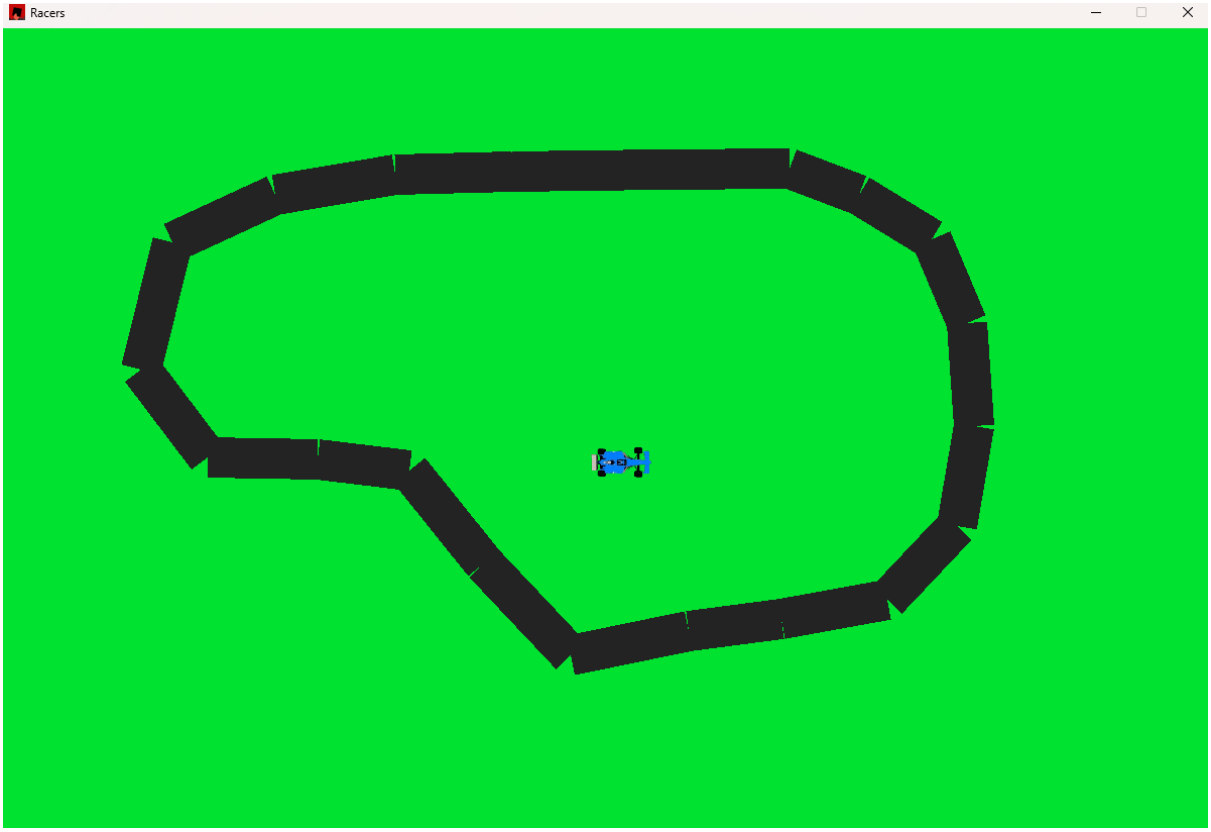
Updated class diagram

Track
Attributes: + points_set: int [2][20] + track_width: float
Methods: + public procedure draw()

So, I added a new track object to the main file and called its **draw** function in the main game loop:

```
async fn main() {
    let mut car1: Car = Car::new();
    let mut track: Track = Track::new(points_set: test_track1, track_width: 40.0);
    loop {
        clear_background(color: GREEN);
        car1.update();
        car1.draw();
        track.draw();

        next_frame().await
    }
}
```

However, when I ran the code it was clear something was a tiny bit off:



The lines are draw like rectangles when ideally they would be drawn as a line of circles, (III implement that now) and the trackwidth is a bit too small so I just needed to tweak that in the class initialisation line from 40px to 75px.

The bespoke draw_line function I wrote in rust:

```rust
pub fn draw_thick_line(x1: f32, y1: f32, x2: f32, y2: f32, thickness: f32, colour: Color) {
    // draw a set of circles along the line
    // the number of circles determines how smooth the line is so i chose 20
    let steps: i32 = 20;
    let vec_x: f32 = x2 - x1;
    let vec_y: f32 = y2 - y1;
    for step: i32 in 0..steps {
        let fstep: f32 = step as f32;
        draw_circle(
            x: x1 + fstep * (vec_x / (steps as f32)),
            y: y1 + fstep * (vec_y / (steps as f32)),
            r: thickness / 2.0,
            color: colour,
        );
    }
}
```

What it does:

1) Calculates the vector (x and y components) from the first point to the second point
2) Loop through all steps and draw a circle at a point which is calculated by **the original point + a step through the vector**
3) Draws all 20 circles and then terminates

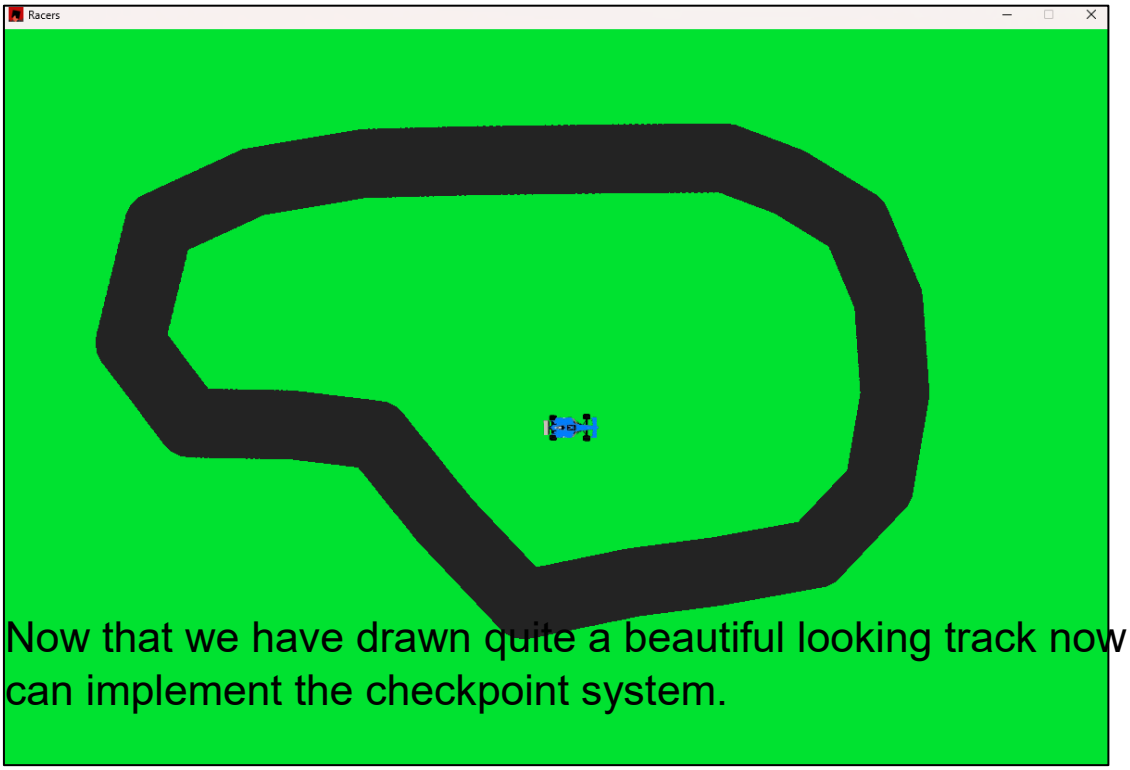
I made sure to use the same number and order of arguments as the **macroquad** library **draw_line** function so that I could easily switch between both.

Here is the new draw_**thick**_line function used in the **track.draw()** method:

```rust
// draw_line(x1, y1, x2, y2, self.track_width, tarmac_colour);
draw_thick_line(x1, y1, x2, y2, thickness: self.track_width, tarmac_colour);
```

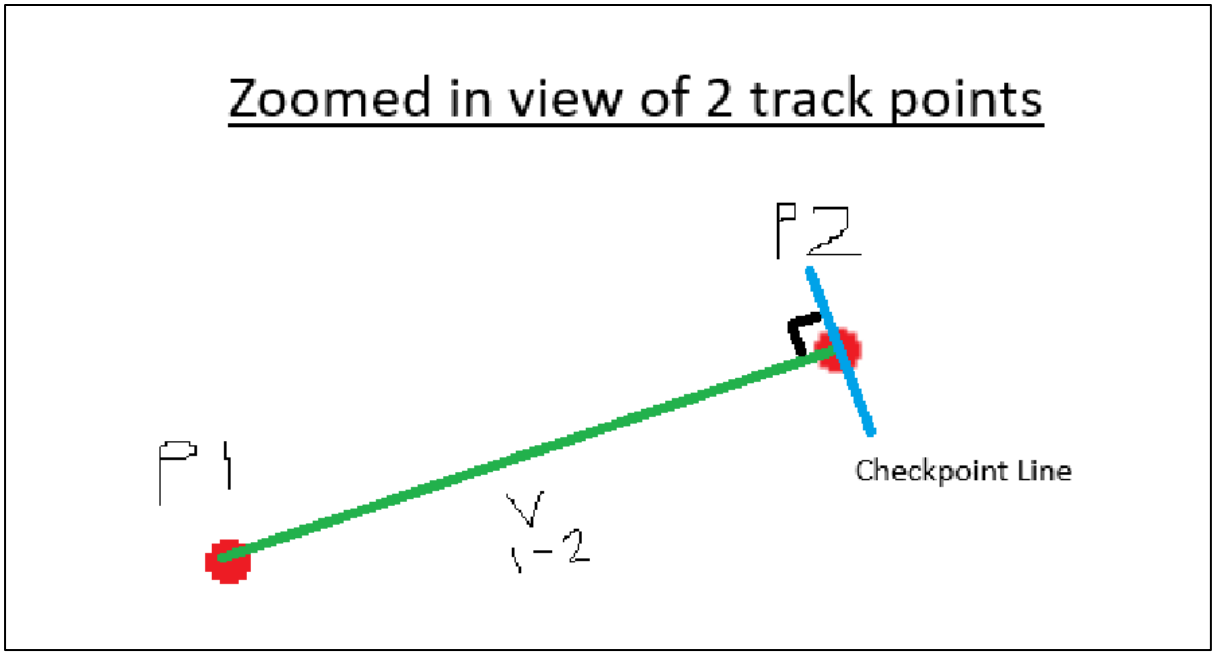
The commented-out line is the previous draw_line library implementation

The new result



Now that we have drawn quite a beautiful looking track now we can implement the checkpoint system.

My idea:



The diagram shows 2 adjacent track points and where the checkpoint line should be drawn. As you can see the checkpoint line is perpendicular to the vector line joining previous point (p1) and the current point (p2).

So, the direction of the checkpoint line can be calculated using some maths (which consequently I have been learning in further maths a level recently).

Let P2 = (x2, y2)

Let P1 = (x1, y1)

$\Delta x = x2 - x1$

$\Delta y = y2 - y1$

The normal vector is $(-\Delta y, \Delta x)$

So now we have an expression for the normal vector direction. However we need to limit its length to **track_width**.

Let normal vector = n

Checkpoint Vector = n/|n| * track_width

The n/|n| is the normal vector divided by its magnitude creating a unit vector in the same direction. Multiplying that by the track_width will give you an aptly sized vector.

Pseudo code

```
procedure draw_checkpoints()

    for i=0 to points_set.length() - 1 Then
        p1 = points[i]
        p2 = points[(i+1) % points_set.length()]
        dx = p2[0] - p1[0]
        dy = p2[1] - p1[1]

        normal = [-dy, dx]
        checkpoint_vector = normal / normal.magnitude * track_width
        start_pos = p2;

        draw_line(
            start_pos[0] - checkpoint_vector[0] * 0.5, start_pos[1] - checkpoint_vector[1] * 0.5,
            start_pos[0] + checkpoint_vector[0] * 0.5, start_pos[1] + checkpoint_vector[1] * 0.5,
            4,   // thickness
            blue // colour
        );
    next i
end procedure
```

After writing this I decided to rewrite some of the track code to use a predefined struct called Vec2 which stores a x and a y component, rather than using an array of length 2 because it would make the code much more readable. The Vec2 object also provides some useful functionality which I would have otherwise had to write myself.

```
pub const test_track1: [Vec2; 20] = [
    vec2(x: 507.0, y: 142.0),
    vec2(x: 654.0, y: 140.0),
    vec2(x: 782.0, y: 139.0),
    vec2(x: 851.0, y: 165.0),
    vec2(x: 923.0, y: 209.0),
    vec2(x: 958.0, y: 292.0),
    vec2(x: 965.0, y: 394.0),
    vec2(x: 948.0, y: 493.0),
    vec2(x: 879.0, y: 566.0),
    vec2(x: 774.0, y: 585.0),
    vec2(x: 682.0, y: 597.0),
    vec2(x: 565.0, y: 621.0),
    vec2(x: 479.0, y: 530.0),
    vec2(x: 405.0, y: 438.0),
    vec2(x: 314.0, y: 427.0),
    vec2(x: 205.0, y: 425.0),
    vec2(x: 139.0, y: 338.0),
    vec2(x: 170.0, y: 212.0),
```

```
pub struct Track {
    points_set: [Vec2; 20],
    track_width: f32,
}

impl Track {
    pub fn new(points_set: [Vec2; 20], track_width: f32) -> Self {
        return Self {
            points_set,
            track_width,
        };
    }
```

```
pub fn draw(&self) {
    for i: usize in 0..self.points_set.len() {
        let p1: Vec2 = self.points_set[i];
        let p2: Vec2 = self.points_set[(i + 1) % self.points_set.len()];

        draw_thick_line(x1: p1.x, y1: p1.y, x2: p2.x, y2: p2.y, thickness: self.track_width, tarmac_colour);
    }
}
```

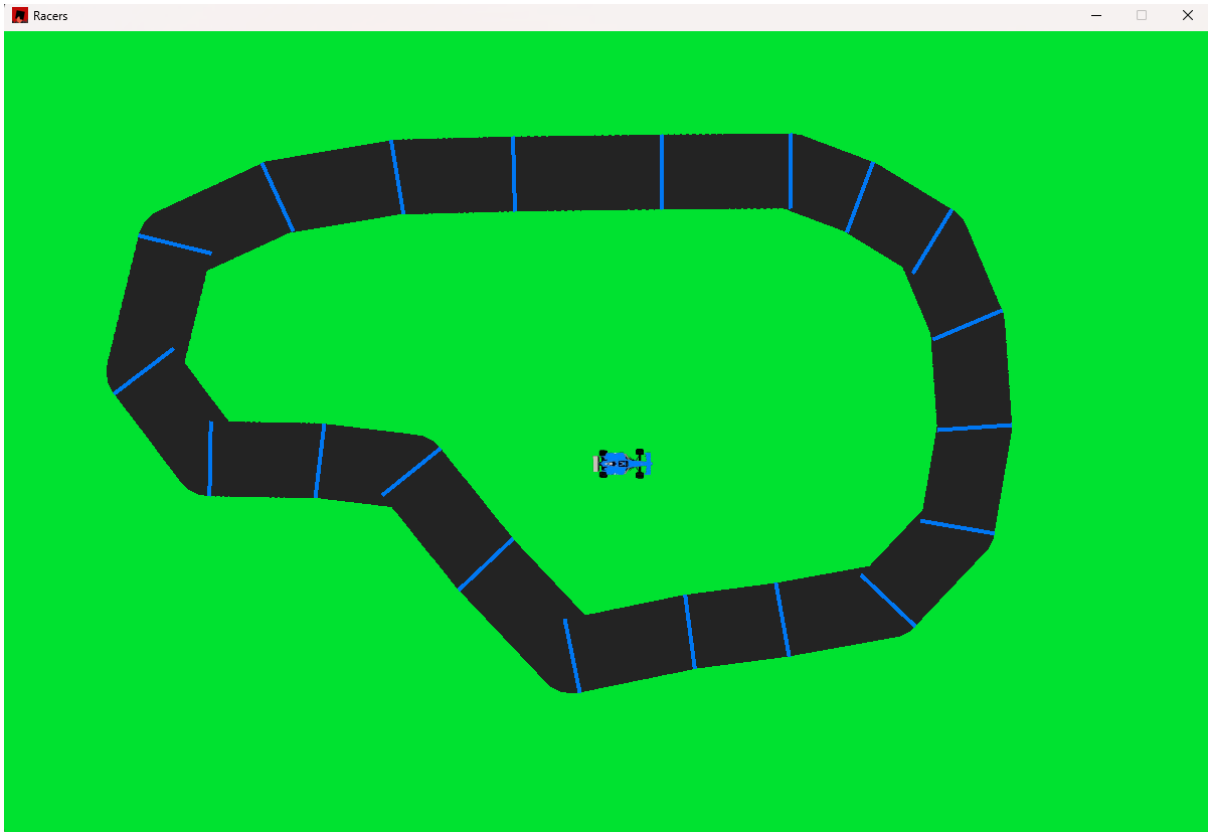
Also, a Vec2's components are float types by default so much of the messy code in the previous draw function can be removed. Overall, changing to this data type will make my code base much more organised. Also note that the new class diagram attribute section looks like this:

Track
Attributes: - points_set: Vec2[20] - track_width: float

Finally, the draw checkpoint method implemented in rust:

```rust
pub fn draw_checkpoints(&self) {
    for i: usize in 0..self.points_set.len() {
        let p1: Vec2 = self.points_set[i];
        let p2: Vec2 = self.points_set[(i + 1) % self.points_set.len()];
        let joining_vec: Vec2 = p2 - p1;
        let normal: Vec2 = vec2(x: -joining_vec.y, y: joining_vec.x);
        let checkpoint_vec: Vec2 = normal.normalize() * self.track_width;
        let start_pos: Vec2 = p2;

        draw_line(
            x1: start_pos.x - checkpoint_vec.x * 0.5,
            y1: start_pos.y - checkpoint_vec.y * 0.5,
            x2: start_pos.x + checkpoint_vec.x * 0.5,
            y2: start_pos.y + checkpoint_vec.y * 0.5,
            thickness: 4.0,
            color: BLUE,
        );
    }
}
```

Now let's run and see the result:



The checkpoint lines are drawn correctly with a little inaccuracy in some places where the line doesn't quite make it to the other side of the track.

Its going to be pretty impossible to make it completely perfect due to the inaccurate nature of drawing the lines as circles, however I have an idea to increase the accuracy of the lines drawn.

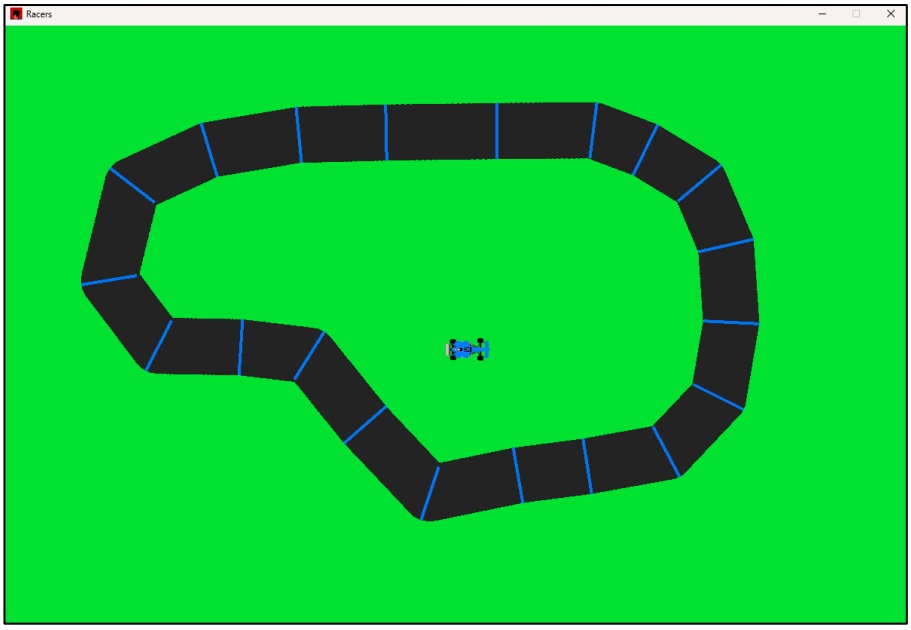
The premise

It works by taking into account the next point in the set of points as well as the last one, calculated both the normals and taking an average of the 2 to create the new line:

Ill just provide the rust code since its quite an intuitive addition:

```rust
pub fn draw_checkpoints(&self) {
    for i: usize in 0..self.points_set.len() {
        let p1: Vec2 = self.points_set[i];
        let p2: Vec2 = self.points_set[(i + 1) % self.points_set.len()]; // this is the one we drawing on
        let p3: Vec2 = self.points_set[(i + 2) % self.points_set.len()];

        let joining_vec1: Vec2 = p2 - p1;
        let normal1: Vec2 = vec2(x: -joining_vec1.y, y: joining_vec1.x);

        let joining_vec2: Vec2 = p3 - p2;
        let normal2: Vec2 = vec2(x: -joining_vec2.y, y: joining_vec2.x);

        let avg_normal: Vec2 = (normal1 + normal2) / 2.0;

        let checkpoint_vec: Vec2 = avg_normal.normalize() * self.track_width;
        let start_pos: Vec2 = p2;

        draw_line(
            x1: start_pos.x - checkpoint_vec.x * 0.5,
            y1: start_pos.y - checkpoint_vec.y * 0.5,
            x2: start_pos.x + checkpoint_vec.x * 0.5,
            y2: start_pos.y + checkpoint_vec.y * 0.5,
            thickness: 4.0,
            color: BLUE,
        );
    }
} fn draw_checkpoints
```

<u>The result:</u>



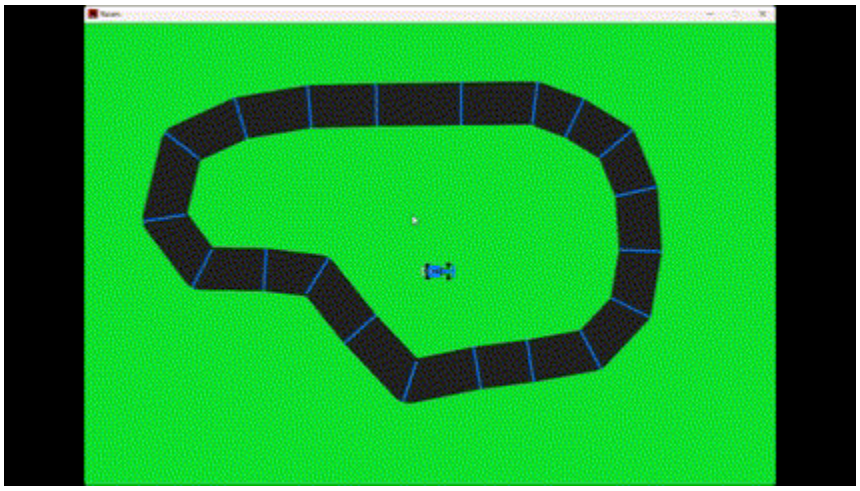
Clearly much more accurately drawn checkpoint lines now!

Now the checkpoint drawing function is complete and should be added to the track main **draw** function

When testing this, the car appeared to be underneath the track, suggesting it must be being drawn before the track, when the car should be drawn after the track.

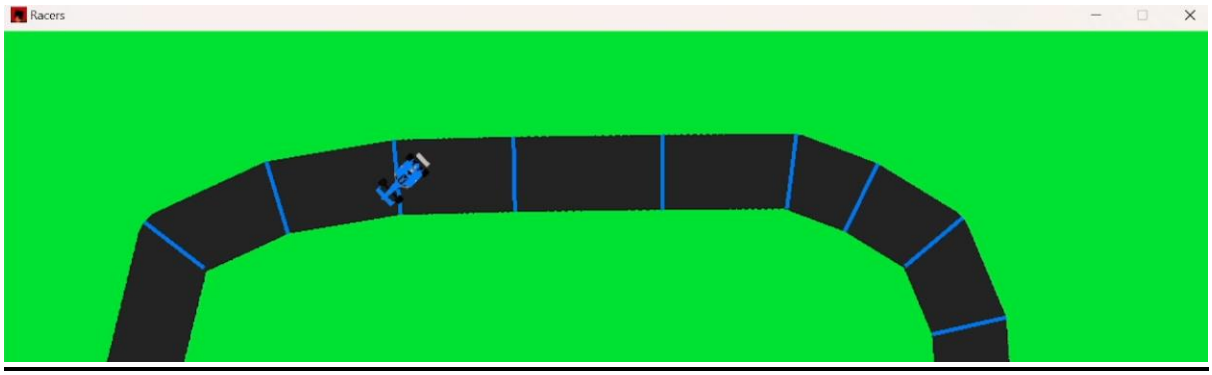
Real time footage of this incident can be found in

Documentation/Videos/carGone.mp4



I fixed this and driving on the track footage can be found

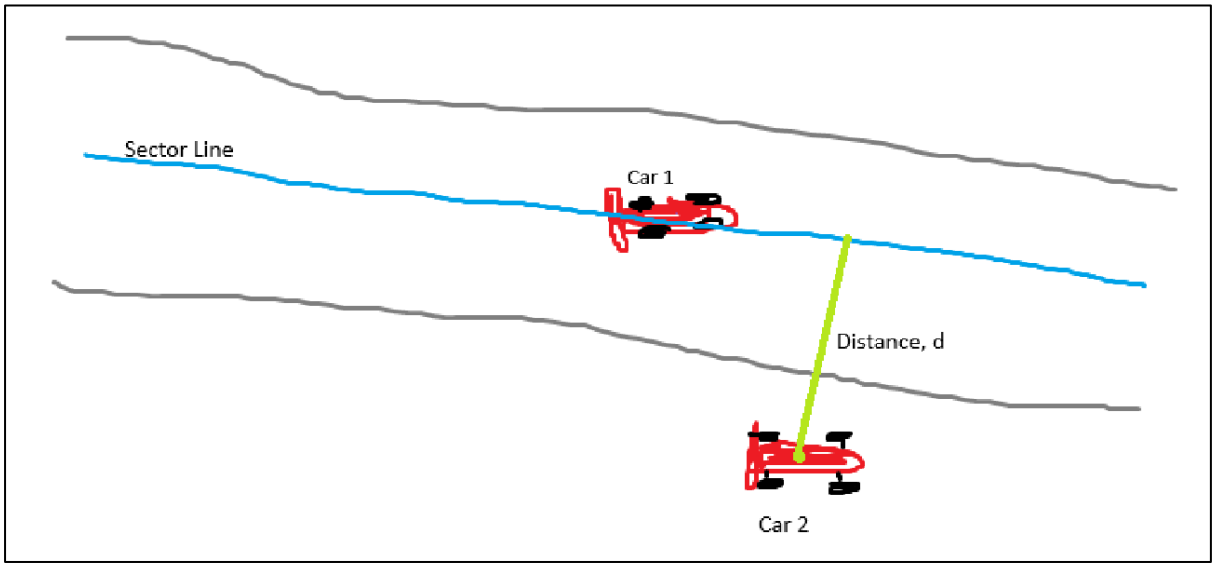
Documentation/Videos/DrivingOnTrack.mkv



Track limits

To handle track limits, I have decided that an effective solution will be to check if the centre of the car's hitbox is outside the track. Ie. It the location of the centre of the hitbox is further than **(track_width / 2)** away from its **sector line**. The **sector line** is the straight line which connects the 2 track points the car is in between.

Diagram



As you can see the Car 1 is pretty much on the sector line so its definitely inside the track however Car 2 is off the track because the distance from the sector line to its centre is greater than the track_width.

To actually implement this, we want an **is_on_track** method attached to the **Car** object which can access the **Track** object.

My idea is to have the is_on_track function take a reference to the track object, then the car methods take over the algorithm, they should identify which sector the car is on (closest to) and then further calculate whether the car is on or off the track in the sector.

So, decomposing the problem we need:

- A function to obtain the current sector
- A function to calculate distance to the sector line

Step 1)

The **get_sector** functionality for the car given the track object as a reference

The outline of the function is as follows

```
fn get_sector(&self, track: &Track) -> i32 {}
```

Note: The sectors should be numbered 0-19 where sector 0 is the sector between trackpoints 0 and 1, and sector 19 is the sector between points 19 and 0.

How to do it:

For every sector

- Find the midpoint between the 2 points which define the sector (i.e. if we are on sector **1** then we look at the points **0 and 1.**
- Calculate distance between car's centre and the midpoint

The sector which gives the shortest distance is the current sector

Implementation

First, I created a getter function which returns a reference to the **Track's** points_set attribute

```
pub fn get_points(&self) -> &[Vec2; 20] {
    return &self.points_set;
}
```

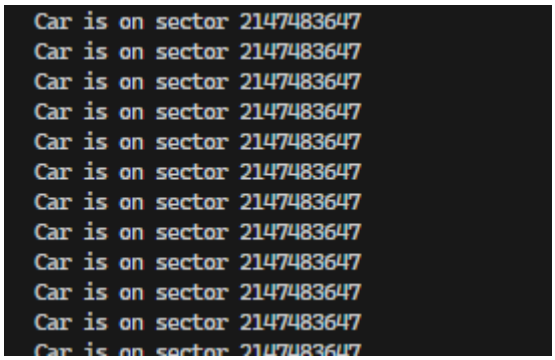
Then I wrote the **get_sector** code.

```
pub fn get_sector(&self, track: &Track) -> i32 {
    let mut closest_sector: i32 = i32::MAX;
    let mut shortest_distance: f32 = 0.0;

    for i: usize in 0..track.get_points().len() {
        // calculate the midpoint
        let p1: Vec2 = track.get_points()[i];
        let p2: Vec2 = track.get_points()[(i + 1) % track.get_points().len()];
        let mp: Vec2 = (p1 + p2) / 2.0;

        let center: Vec2 = self.rect.center();
        let distance: f32 = mp.distance(center);
        if distance < shortest_distance {
            shortest_distance = distance;
            closest_sector = i as i32;
        }
    }
    return closest_sector;
}
```

To test the code, I ran it with a print message as shown

```rust
loop {
    clear_background(color: GREEN);
    car1.update();

    println!("Car is on sector {}", car1.get_sector(&track));

    track.draw();
    car1.draw();

    next_frame().await
}
```

*However, **this** was my terminal:*



Clearly this is not a correct result, however when I inspected the function again I realised a simple error in the code

```rust
pub fn get_sector(&self, track: &Track) -> i32 {
    let mut closest_sector: i32 = i32::MAX;
    let mut shortest_distance: f32 = 0.0;
```

I mixed up the top two variables, the closest_sector should be **0** at default and the shortest distance should be at **f32::MAX** – which is the *largest possible 32-bit float* - at default (it should be as large as possible so any other distance will override it).

I swapped those values around and now the fixed code looks like this

```rust
pub fn get_sector(&self, track: &Track) -> i32 {
    let mut closest_sector: i32 = 0;
    let mut shortest_distance: f32 = f32::MAX;
```

I also decided to highlight the car's sector to make it easier to understand the output of the function.

After fixing a few silly errors the system worked great! The running video of this working can be found in:

Documentation/Videos/SectorHighlighting.mkv

Step 2)

Finding distance to the sector line

The only tough bit about this is the calculation to find the perpendicular distance from a line (vector) to a point. However, there is a pretty simple formula to compute this distance, I found this formula on brilliant.org/wiki/dot-product-between-point-and-a-line

THEOREM

The distance d from a point (x_0, y_0) to the line $ax + by + c = 0$ is

$$d = \frac{|a(x_0) + b(y_0) + c|}{\sqrt{a^2+b^2}}.$$

To find the equation of the line of the track sector we need to make a new function which takes in the two points calculates a gradient and finds the values of **a, b and c** in the formula above. To make computing this easier let's say that **b** is 1 so we just find an equation in the form

y = mx+d

and then rearrange it, to find **a and c**. This means that **m** is the gradient (which is easy to compute) and then we can find **d** using simple maths.

Pseudocode of this:

```
function find_line_eq(x1, y1, x2, y2)
    // FORM: ax + y + c = 0
    // y = mx + d
    m = (y2 - y1)/(x2 - x1)

    // plug point in
    // y1 = m * x1 + d
    // rearrage -> d = y1 - m * x1
    d = y1 - m * x1
    // now we have d and m
    // y - mx - d = 0
    // by +ax + C = 0
    // so b = 1 (we already know)
    // a = -m
    a = -m
    // C = -d = -(y1 - m * x1)
    c = -d
    return a, c
endfunction
```

The maths is very simple but I tried to comment it in any way to explain my thought process to myself.

Here's the function now converted to rust:

Note: I put useful global function like this into a file called utils.rs so that they exist in their own section of the code base in order to declutter the main files.

```rust
pub fn find_line_eq(x1: f32, y1: f32, x2: f32, y2: f32) -> Vec2 {
    // trying to complete form:
    // ax + by + c = 0
    let m: f32 = (y2 - y1) / (x2 - x1);
    let d: f32 = y1 - m * x1;
    let a: f32 = -m;
    let c: f32 = -d;

    // the Vec2 will return (a, c) in this form
    return vec2(x: a, y: c);
}
```

Testing this function

Test Data, Expected result, actual result

Test Data	Expected Result	Actual Result	Result
(3, 5), (4, 6)	A = -1, C= -2	A = -1, C= -2	**Pass**
(-5, 2), (19, 20)	A = -0.75, C=-5.75	A = -0.75, C=-5.75	**Pass**
(0, 0), (0, 0)	A = NaN, B = NaN	A = NaN, B = NaN	**Pass**

Output ->
```
test one: Vec2(-1.0, -2.0)
test two: Vec2(-0.75, -5.75)
test one: Vec2(NaN, NaN)
```

Now it's safe to use it in the main **is_on_track** function. Now we can find the perpendicular distance from the centre of the car to its nearest sector line so we can write the whole function:

```rust
pub fn is_on_track(&self, track: &Track) -> bool {
    // find the current sector
    let sector: usize = self.get_sector(track) as usize;

    // find the sector line equation
    let points: &[Vec2; 20] = track.get_points();
    let p1: Vec2 = points[sector];
    let p2: Vec2 = points[(sector + 1) % points.len()];
    let coef_vec: Vec2 = find_line_eq(x1: p1.x, y1: p1.y, x2: p2.x, y2: p2.y);
    let a: f32 = coef_vec.x;
    let b: f32 = 1.0;
    let c: f32 = coef_vec.y;

    // use the formula for distance
    let center: Vec2 = self.rect.center();
    let distance: f32 = (a * center.x + b * center.y + c).abs() / (a * a + b * b).sqrt();

    if distance > (track.get_width() / 2.0) {
        // off the track
        return false;
    }

    return true;
}
```

I also had to add another getter for the track object, **get_width()**:

```rust
pub fn get_width(&self) -> f32 {
    return self.track_width;
}
```

Now I will add a print alert to show whether the car is on or off the track and run the code!

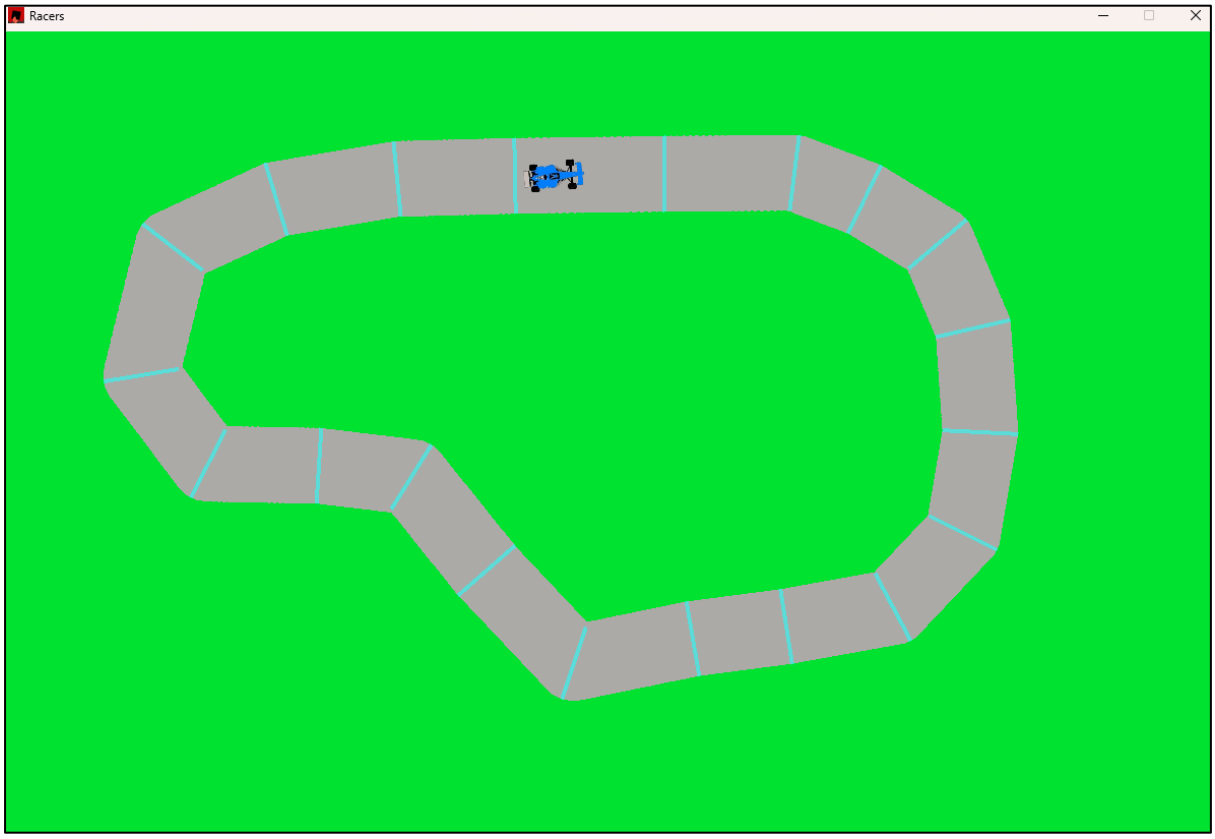
```rust
    println!("Car is on track: {}", car1.is_on_track(&track));
```

The function worked perfectly (which really surprised me) and the footage can be seen in:

Documentation/Videos/TrackLimitTesting1.mkv

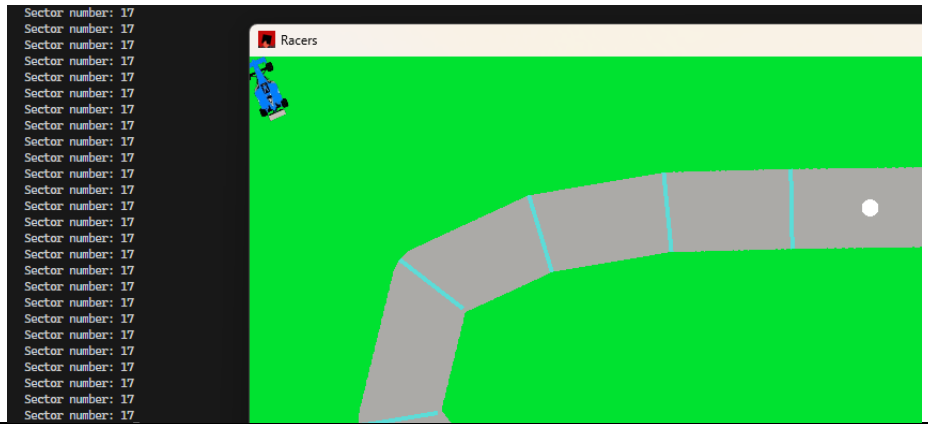
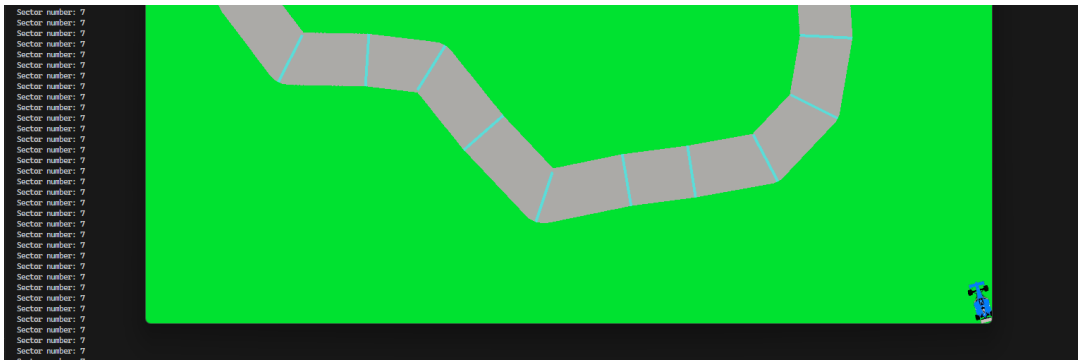
Finally, I removed the temporary code which highlights the sector that the car is on and I tweaked the tarmac colour to be slightly lighter as well as the checkpoint colour.

```rust
pub const tarmac_colour: Color = color_u8!(171, 170, 167, 255);
pub const checkpoint_colour: Color = color_u8!(36, 255, 251, 150);
```



Stage 3 Testing

Test Name	Description	Test Data	Expected Result	Actual Result	Result
3.a	Boundary test for get_sector function	Place car in the top right of screen	The function should return sector number **17**	The output was sector number **17**	**PASS**
3.b	Boundary test for get_sector function	Place car in bottom right of screen	The function should return sector number **7**	The output was sector number **7**	**PASS**
3.c	Full test of **track limit detector**	Drive all around the track both on and off	When the car's centre is visibly off the track the return value of **is_on_track** should be false. When it is on track the return value should be true.	The function results coincided exactly with what was visible on the screen. I found no unexpected results.	**PASS**

Test Name	Evidence
3.a	
3.b	
3.c	Screen recording found *Documentation/Videos/testing3c.mkv*

End of stage 3

Now we are at the end of development stage 3, before I move onto the next stage I want to re-create the respective class diagrams for my structures.

Car
Attributes + Velocity: Vector + Direction: Vector + Position: Vector + Acceleration: Vector + angle: Float + steer: Float +texture: Texture2D +rect: Rect +accelerator_input: Input +steering_input: Input +brakes_input: Input
Constants + hitbox_width: float + hitbox_height: float + max_speed: float + max_acceleration: float + steer_weight: float + max_turning_angle: float + mass: float + braking_factor: float
Methods + draw()
+ update_pos(x, y)
+ update()
+ keyboard_control()
+ get_sector()
+ is_on_track()

Track
Attributes: + points_set: int [2][20] + track_width: float
Methods: + draw() + get_points() + get_width() + draw_checkpoints()

Recap

1. **Stage Four:** Implement the backend for neural networks, including **matrices, matrix operators,** and a ***network*** **object** which keeps track of **layers, weights** and allows easy abstraction from the inner complexities.

Development Stage 4

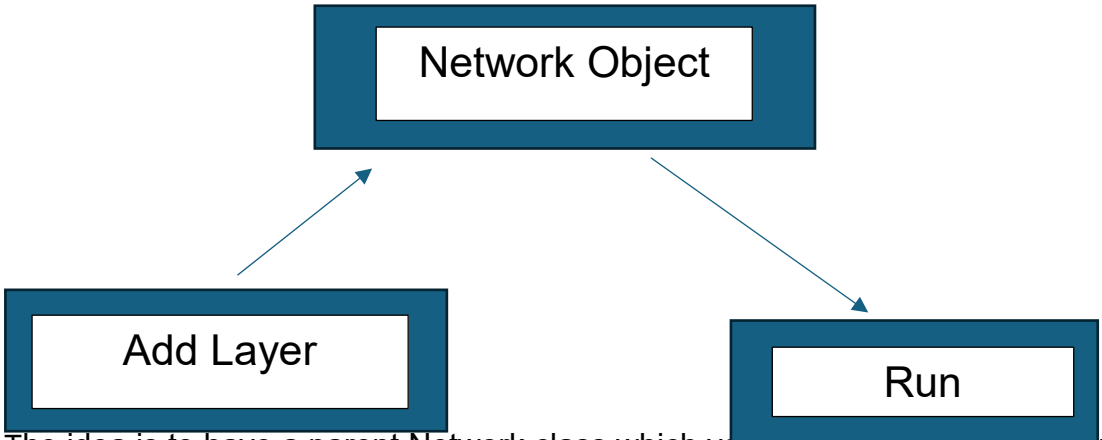
Before attempting to design or write this part of the project I did some extensive research into neural networks, and the implementation of them in various programming languages. Using the following sites:

https://www.digitalocean.com/community/tutorials/constructing-neural-networks-from-scratch

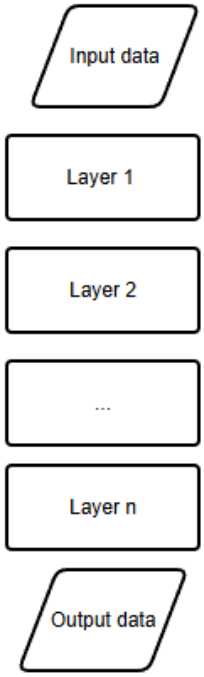
https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

The second article was particularly helpful (I just read around the bits about back propagation since the genetic learning algorithm doesn't require this).

The Network Architecture



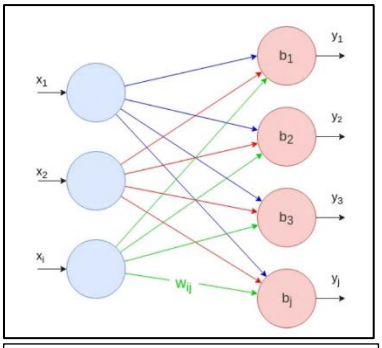
The idea is to have a parent Network class which you can construct by adding layers to it (being able to specify the number of inputs and outputs, and the activation function if its an activation layer). Then we can feed input data into the network and the network will run a forward pass and return the set of output values, the data flows layer by layer until it reaches the output layer.



Using this design, it will be easy to tweak the network structure of the AI controllers without rewriting parts of the class.

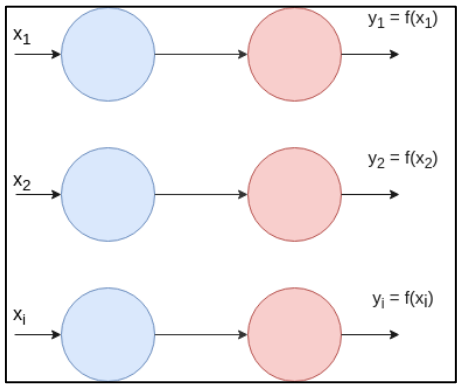
Types of Layers

- **Fully connected layer** – a layer of **n** neurons, where every neuron is connected to every other neuron in the next layer.



Fully connected layer diagram from https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

- **Activation layer** – a layer which applies a given **activation function** (usually an exponential function) to each input neuron.



A diagram of an activation function from https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

Implementation

A fully connected layer needs to have a set of weights, and biases. During forward propagation an output neuron's value is calculated by summing the product of every input value and its respective weight value and then adding the constant bias.

Using linear algebra, you can carry out the process in one big matrix calculation as follows:

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

And this is what we will compute in our program.

Figure 1- https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

Network Class

Network
Attributes: + layers: Layer
Methods: + public add_layer(layer) + public run(inputData)

I'm going to place the neural network code in a new file called network.rs

Network class:

```rust
pub struct Network {
    layers: Vec<Layer>,
}
```

Constructor:

```rust
pub fn new(layers: Vec<Layer>) -> Self {
    Self { layers }
}
```

The network class will intialise as an empty list of layers, and then I can add layers using the **add_layer** function.

> Note: The matrix in the calculation will be structured in the rust **vector** data type. A 2D **vector** will define a matrix and a 1D **vector** will define a column vector.

Layer Class

Layer
Attributes: + weights[]: float + bias[]: float + activation: **function** optional + outputs[] float optional
Methods: + public calculate(input[] float)

Layer class:

```rust
pub struct Layer {
    pub weights: Vec<Vec<f64>>,
    pub bias: Vec<f64>,
    activation: Option<fn(f64) -> f64>,
    output: Option<Vec<f64>>,
}
```

Layer constructor:

```rust
pub fn new(weights: Vec<Vec<f64>>, bias: Vec<f64>, activation: Option<fn(f64) -> f64>) -> Self {
    Self {
        weights,
        bias,
        activation,
        output: None,
    }
}
```

The layer calculation:

Psuedo Code

```
function calculate(inputs)
    // check inputs and weights are compatable sizes
    if (inputs.length() != self.weights.length()) Then
        return ERROR
    Endif

    outputs = []
    for i=0 to self.bias.length() - 1
        outputs.push(0)
    next i

    // perform calculation
    for i=0 to self.bias.length() - 1
        inputVal = inputs[i]
        for j=0 to self.bias.length() - 1
            biasVal = self.bias[j]
            weightVal = self.weights[j]
            outputVal = inputVal * weightVal + bias
        next j
    next inputVal

    if (there is a activation function) Then
        return apply_activation_function(ouputs)
    Endif
    return outputs

end function

```

The whole network pass algorithm

Psuedo Code

```
1  function calculate(inputs)
2      // check inputs and weights are compatable sizes
3      if (inputs.length() != self.weights.length()) Then
4          return ERROR
5      Endif
6
7      outputs = []
8      for i=0 to self.bias.length() - 1
9          outputs.push(0)
10     next i
11
12     // perform calculation
13     for i=0 to self.bias.length() - 1
14         inputVal = inputs[i]
15         for j=0 to self.bias.length() - 1
16             biasVal = self.bias[j]
17             weightVal = self.weights[j]
18             outputVal = inputVal * weightVal + bias
19         next j
20     next inputVal
21
22     if (there is a activation function) Then
23         return apply_activation_function(ouputs)
24     Endif
25
26
27     self.outputs = outputs
28
29     return outputs
30
31 end function
```

Note: in line **27** the output values of the layer are saved to the layer object so it can be used later.

The pseudo code for a network pass

```
1  function run(inputs)
2      prevVal = inputs
3      for i=0 to self.layers.length() - 1
4          prevVal = self.layers[i].calculate(prevVal)
5      next i
6      reutrn prevVal
7  end function
```

Now I can write the code in rust:

The layer calculate code:

```rust
pub fn calculate(&mut self, inputs: Vec<f64>) -> Vec<f64> {
    // make sure that the inputs and weights are compatable sizes
    if (inputs.len() != self.weights.len()) {
        panic!("[network.rs] - Incompatable input and weights sizes!\n");
    }

    // create empty list of zeros
    let mut outputs: Vec<f64> = iter::repeat(elt: 0f64).take(self.bias.len()).collect();

    // perform the matrix multiplication of inputs and weights
    for input_index: usize in 0..inputs.len() {
        let input: f64 = inputs[input_index];
        for output_index: usize in 0..outputs.len() {
            let bias_value: f64 = self.bias[output_index];
            let weight_value: f64 = self.weights[input_index][output_index];
            outputs[output_index] = input * weight_value + bias_value;
        }
    }

    // if there is an activation function return the values with the function applied
    if (self.activation.is_some()) {
        return self.apply_activation(inputs);
    }

    self.output = Some(inputs.clone());

    return inputs;
} fn calculate

fn apply_activation(&self, mut inputs: Vec<f64>) -> Vec<f64> {
    for i: usize in 0..inputs.len() {
        inputs[i] = self.activation.unwrap()(inputs[i]);
    }
    return inputs;
}
```

The Network pass code

```rust
pub fn add_layer(mut self, layer: Layer) -> Self {
    self.layers.push(layer);
    return self;
}

pub fn run(&mut self, inputs: Vec<f64>) -> Vec<f64> {
    // run first layer
    let mut prev_output: Vec<f64> = inputs;
    for layer: &mut Layer in self.layers.iter_mut() {
        prev_output = layer.calculate(inputs: prev_output);
    }
    return prev_output;
}
```

Finally, I will add my chosen activation function – the **sigmoid** function. The **sigmoid** function is a nonlinear activation function defined as the following:

Formula

$$S(x) = \frac{1}{1 + e^{-x}}$$

This should be easy enough to program into a function in rust:

```
pub fn sigmoid(x: f64) -> f64 {
    return 1.0 / (1.0 + f64::consts::E.powf(-x));
}
```

Let's test the network code in isolation first:

1) Create a network with 2 layers, the first with 3 neurons and the second with 2 neurons. Only the second layer is to have a **sigmoid** activation function
2) Provide layer 1 with respective weights and biases:
- Weights: [[0.5, 0.25, 0.9], [0.1, 0, -0.1], [1.0, -1.0, 0.2]]
- Biases: [1.0, 4.0, 1.0]
3) Provide layer 2 with respective weights and biases:
- Weights: [[0.1, 1.0, 0.1], [0.5, -0.5, 0.0]]
- Biases: [10.0, 0.0]
4) Give the network the inputs [2.0, 1.0, 3.0]

5) What are the expected results?

I calculated them on my white board and found that the output values after the first layer should be:

[4.95, 3.9, 2.6]

But I kept getting this output

```
Output of layer 1: Some([2.0, 1.0, 3.0])
```

I realized that this logic bug must be caused somewhere in the **calculate_layer** code and I found that I wrote this

```
self.output = Some(inputs.clone());

return inputs;
```

Clearly, I mixed up the **inputs** and **outputs** variables, so the network was just redistributing the input matrix rather than actually passing along the calculated outputs. I quickly fixed that and now the output is: `Output of layer 1: Some([4.0, 1.0, 1.6])` which is again incorrect, so I need to have another look at the calculation code for errors.

Looking at the previous code, I think the input index and output index were mixed up, I rewrote the code:

```rust
for output_index: usize in 0..outputs.len() {
    let bias: f64 = self.bias[output_index];
    for input_index: usize in 0..inputs.len() {
        let input: f64 = inputs[input_index];
        let weight: f64 = self.weights[output_index][input_index];
        outputs[output_index] += weight * input;
    }
    outputs[output_index] += bias;
}
```

To see if this solution would work, I drew a trace table for the first calculation:

Output Index	Input Index	Bias	Input	Weight	Output
0	0	1.0	2.0	0.5	1.0
0	1		1.0	0.25	1.25
0	2		3.0	0.9	4.95

The trace table showed that the logic was correct and the code should work, I ran the tests again with the new code and got the correct results:

```
Output of layer 1: Some([4.95, 3.9, 2.6])
```

6) Now what are the expected results for the final output layer:

- So, the inputs to the final layers are [4.95, 3.9, 2.6], then we apply the weights [[0.1, 1.0, 0.1], [0.5, -0.5, 0.0]] Which gives us the matrix [4.655, 0.525] Then we apply the bias values [10.0, 0.0] to get [4.655, 0.525]. Then we finally apply the activation function to get [0.9906, 0.6283]

However, after I got a wrong result, I looked over the code again and found this bug

```rust
if (self.activation.is_some()) {
    return self.apply_activation(inputs);
}
```

Instead of applying the activation function to the output values this snippet of code applies it to the input values which is redundant. I fixed that and now the program spits out the correct value:

```
Output layer2: [0.9999995680691935, 0.6283161882953663]
```

Here is the bug free function:

```rust
pub fn calculate(&mut self, inputs: Vec<f64>) -> Vec<f64> {
    // make sure that the inputs and weights are compatable sizes
    if (inputs.len() != self.weights[0].len()) {
        panic!("[network.rs] - Incompatable input and weights sizes!\n");
    }

    // create empty list of zeros
    let mut outputs: Vec<f64> = iter::repeat(elt: 0.0).take(self.bias.len()).collect();
    println!("Outputs length: {}", outputs.len());

    // perform the matrix multiplication of inputs and weights
    for output_index: usize in 0..outputs.len() {
        let bias: f64 = self.bias[output_index];
        for input_index: usize in 0..inputs.len() {
            let input: f64 = inputs[input_index];
            let weight: f64 = self.weights[output_index][input_index];
            outputs[output_index] += weight * input;
        }
        outputs[output_index] += bias;
    }

    // if there is an activation function return the values with the function applied
    if (self.activation.is_some()) {
        return self.apply_activation(inputs: outputs);
    }

    self.output = Some(outputs.clone());

    return outputs;
} fn calculate
```

Now that the neural network code has passed the test, its safe to apply to the main program. However, first, when I was writing the test code, I found that constructing the layers manually is very time consuming and the code will get very messy very quickly so I'm going to make a better layer constructor which randomly initialises the weights and biases ready for use in the genetic algorithm.

```
//testing
let mut network: Network = Network::new_empty();
let layer1: Layer = Layer::new(
    weights: vec![
        vec![0.5, 0.25, 0.9],
        vec![0.1, 0.0, -0.1],
        vec![1.0, -1.0, 0.2],
    ],
    bias: vec![1.0, 4.0, 1.0],
    activation: None,
);
let layer2: Layer = Layer::new(
    weights: vec![vec![0.1, 1.0, 0.1], vec![0.5, -0.5, 0.0]],
    bias: vec![10.0, 0.0],
    activation: Some(sigmoid),
);
network = network.add_layer(layer1).add_layer(layer2);
let outputs: Vec<f64> = network.run(inputs: vec![2.0, 1.0, 3.0]);
println!("Output of layer 1: {:?}", network.layers[0].output);
println!("Output layer2: {:?}", outputs);
```

Figure 2- Test code

The random initialisation of the weights and biases

For the time being I'm going to randomly choice values between -0.75 and 0.75 for the weights and values between -0.25 and 0.25 for the biases. These values are quite important and will most likely need tweaking later, but for now I have just chosen these arbitrary

```
pub fn new_random(inputs: usize, outputs: usize, activation: Option<fn(f64) -> f64>) -> Self {
    macroquad::rand::srand(seed: macroquad::miniquad::date::now() as _);

    // generating the weights between 0.75 and -0.75
    let mut weights: Vec<Vec<f64>> = vec![];
    for i: usize in 0..outputs {
        let mut inner: Vec<f64> = vec![];
        for j: usize in 0..inputs {
            let val: f64 = macroquad::rand::gen_range(low: -0.75, high: 0.75);
            inner.push(val);
        }
        weights.push(inner);
    }

    let mut bias: Vec<f64> = vec![];
    for i: usize in 0..outputs {
        let val: f64 = macroquad::rand::gen_range(low: 0.25, high: -0.25);
        bias.push(val);
    }

    return Self::new(weights, bias, activation);
}
```

Figure 3- The random initialisation code

Creating a network now – the easier way

```
let mut network: Network = Network::new_empty();
network = network.add_layer(Layer::new_random(inputs: 4, outputs: 6, activation: None));
network = network.add_layer(Layer::new_random(inputs: 6, outputs: 3, activation: None));
network = network.add_layer(Layer::new_random(inputs: 3, outputs: 2, activation: Some(sigmoid)));
```

Testing stage 4

Test Name	Network Structure	Weights	Biases	Input Data	Expected Output	Actual output	Result
4.a	Network with 1 layer with 1 neuron	[1.0]	[0.0]	[3.5]	[3.5]	`Network output: [3.5]`	**PASS**
4.b	Network with 2 layers with, of size 2 and 1	[[1.0, 0.5], [0.9, -0.5]] [[0.1, 0.2]]	[3.0, -2.5] [1.0]	[1.0, 2.0]	[0.98]	`Network output: [0.98]`	**PASS**
4.c	Network with 1 layer with 1 neuron, with a sigmoid activation function	[0.9827]	[-1.0]	[123.0]	[1]	`Network output: [1.0]`	**PASS**

Development Stage 5

Recap

Stage Five: Attaching a controller to a **car** object so it can use its own neural network to move the car, this involves choosing what data inputs enter the network and deciding the ‘bare metal’ anatomy of the network.

First, we need a super object to handle all the cars, I’ll call it the **population** class. The population class needs to:

- Store all the **cars** in a list
- Handle drawing and updating all the **cars**
- Eventually handle the evolution cycle, i.e. carry out the genetic algorithm
- Spawn all the **cars** on the track
- Store the **track** object
- Handle drawing the **track**

Note: I’ll be isolating the population code into another file called *population.rs*

```rust
use crate::Car::*;
use crate::Network::*;
use crate::Track::*;
use macroquad::prelude::*;

pub struct Population {
    generation: usize,
    cars: Vec<Car>,
    track: Track,
}
```

Constructor:

```rust
impl Population {
    pub fn new(size: usize) -> Self {
        let mut cars: Vec<Car> = vec![];
        for i: usize in 0..size {
            cars.push(Car::new());
        }
        let track: Track = Track::new(points_set: test_track1, track_width: 75.0);

        Self {
            generation: 0,
            cars,
            track,
        }
    }
}
```

Now the population class needs a **draw** and **update** function to be called every frame from the main function.

```rust
pub fn draw(&self) {
    self.track.draw();

    for car: &Car in self.cars.iter() {
        car.draw();
    }
}

pub fn update(&mut self) {
    for car: &mut Car in self.cars.iter_mut() {
        car.update();
    }
}
```

These functions are very self-explanatory and didn't take much planning.

Now we can create a population in the main function.

> Note: Now the car will no longer be controlled by the keyboard, since we will be implementing the AI controller now.

```rust
#[macroquad::main(window_conf)]
async fn main() {
    let mut population: Population = Population::new(size: 1);
    loop {
        clear_background(color: GREEN);

        population.update();
        population.draw();

        next_frame().await
    }
}
```

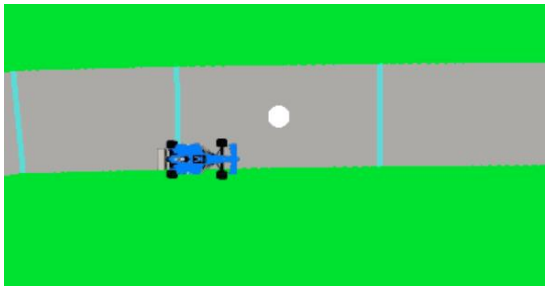
Everything runs fine, but now we need to stop spawning the car in the middle of the screen but instead at the start line. So now I added a **start_position** argument to **car** class constructor. And I added a **get_start_pos** method to the track class:

```rust
pub fn new(start_pos: Vec2) -> Self {
    // default car setup
    let mut car: Self = Self {
        texture: Texture2D::from_file_with_format(include_bytes!("../assets/car.png"), format: None),

        // Defining Vector
        position: start_pos,
```

```rust
pub fn get_start_pos(&self) -> Vec2 {
    let pos: Vec2 = self.points_set[0];
    return pos;
}
```

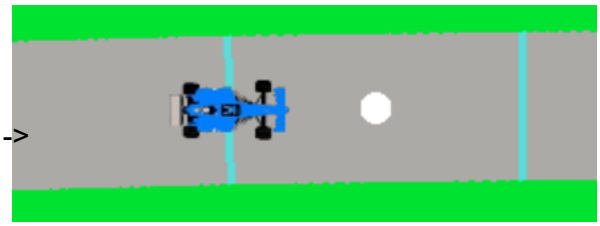
```rust
let track: Track = Track::new(points_set: test_track1, track_width: 75.0);
let mut cars: Vec<Car> = vec![];
for i: usize in 0..size {
    cars.push(Car::new(track.get_start_pos()));
}
```

When running the code the car spawned slightly offset from the start point. I quickly realised this is because the position of the car defines the position of the top left of the car sprite, so I just need to offset the start pos by **car_width / 2** and **car_height / 2**. As shown below:



```
position: vec2(
    x: start_pos.x - Self::HITBOX_WIDTH / 2.0,
    y: start_pos.y - Self::HITBOX_HEIGHT / 2.0,
),
```

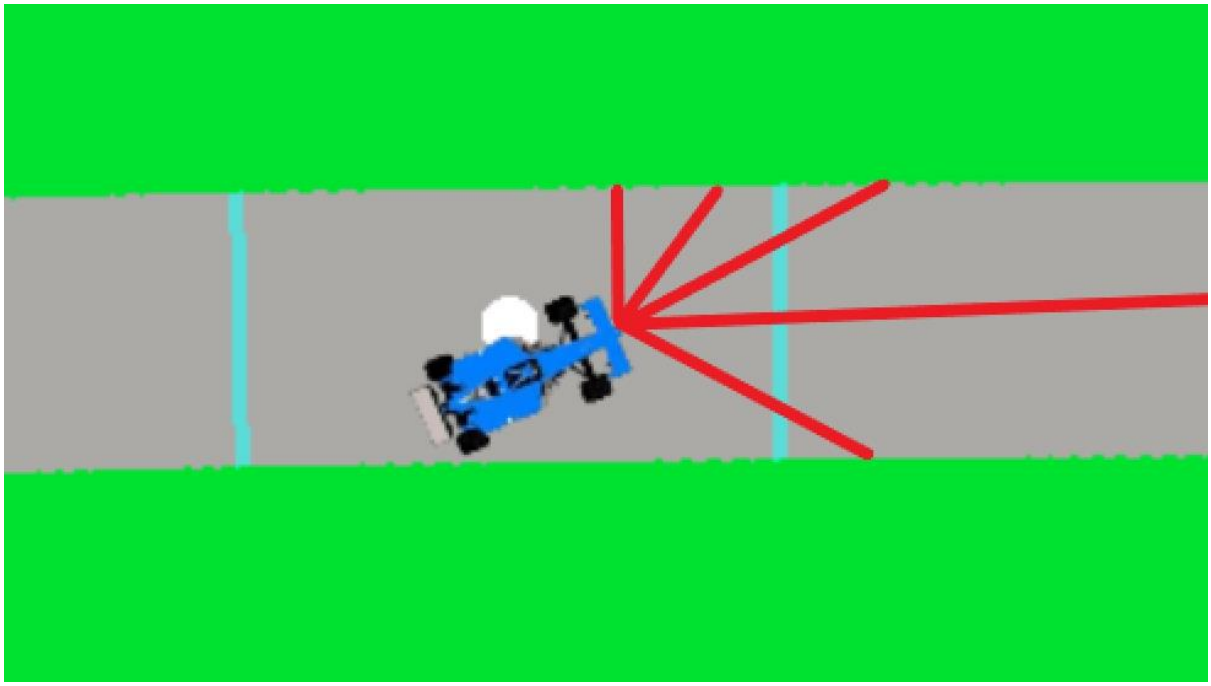
Now when I run the code the screen looks like this ->



Designing the AI controller

What inputs/sensors does the controller need?

- Ray casting sensors to sense the edge of the track, should be cast in a range of directions around the car
- The car's x velocity
- The car's y velocity
- The car's speed
- The car's angle
- The car's steer value



This diagram shows the car casting its sensor rays, and they stop when they hit the barrier.

Note: The diagram shows the rays protruding from the front centre of the car whereas in reality the rays will be cast from the centre of the car's hitbox

How to cast the rays:

- Get the direction vector of the ray, from the car's angle.

- See which sector line the car will interesect with
- Calculate the distance between the source and the sector line

The maths:

Given two points on each line segment [edit]

See also: Intersection_(geometry) § Two_line_segments

The intersection point above is for the infinitely long lines defined by the points, rather than the line segments between the points, and can produce an intersection point not contained in either of the two line segments. In order to find the position of the intersection in respect to the line segments, we can define lines L_1 and L_2 in terms of first degree Bézier parameters:

$$L_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}, \qquad L_2 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} + u \begin{bmatrix} x_4 - x_3 \\ y_4 - y_3 \end{bmatrix}$$

(where t and u are real numbers). The intersection point of the lines is found with one of the following values of t or u, where

$$t = \frac{\begin{vmatrix} x_1 - x_3 & x_3 - x_4 \\ y_1 - y_3 & y_3 - y_4 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

and

$$u = -\frac{\begin{vmatrix} x_1 - x_2 & x_1 - x_3 \\ y_1 - y_2 & y_1 - y_3 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = -\frac{(x_1 - x_2)(y_1 - y_3) - (y_1 - y_2)(x_1 - x_3)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)},$$

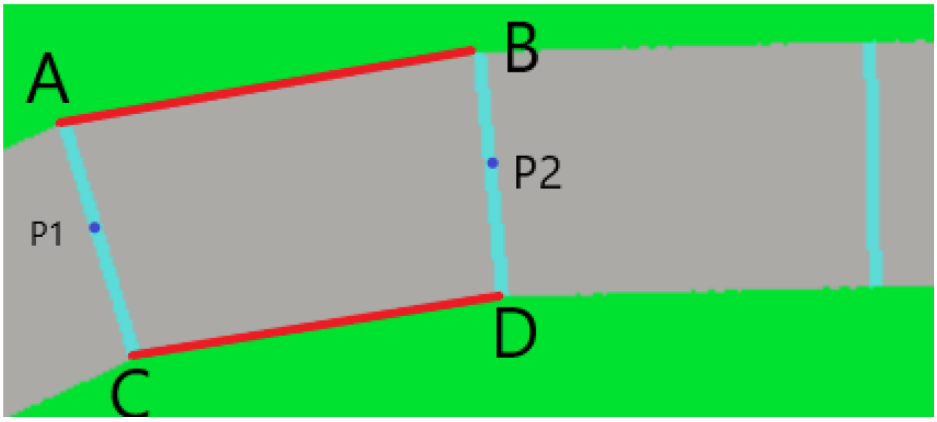
with

$$(P_x, P_y) = \big(x_1 + t(x_2 - x_1),\ y_1 + t(y_2 - y_1)\big) \quad \text{or} \quad (P_x, P_y) = \big(x_3 + u(x_4 - x_3),\ y_3 + u(y_4 - y_3)\big)$$

There will be an intersection if $0 \le t \le 1$ and $0 \le u \le 1$. The intersection point falls within the first line segment if $0 \le t \le 1$, and it falls within the second line segment if $0 \le u \le 1$. These inequalities can be tested without the need for division, allowing rapid determination of the existence of any line segment intersection before calculating its exact point.[3]

Figure 4 - https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection

So all we need to find is the 4 coordinates which define the 2 lines and then apply this series of maths to complete the problem.



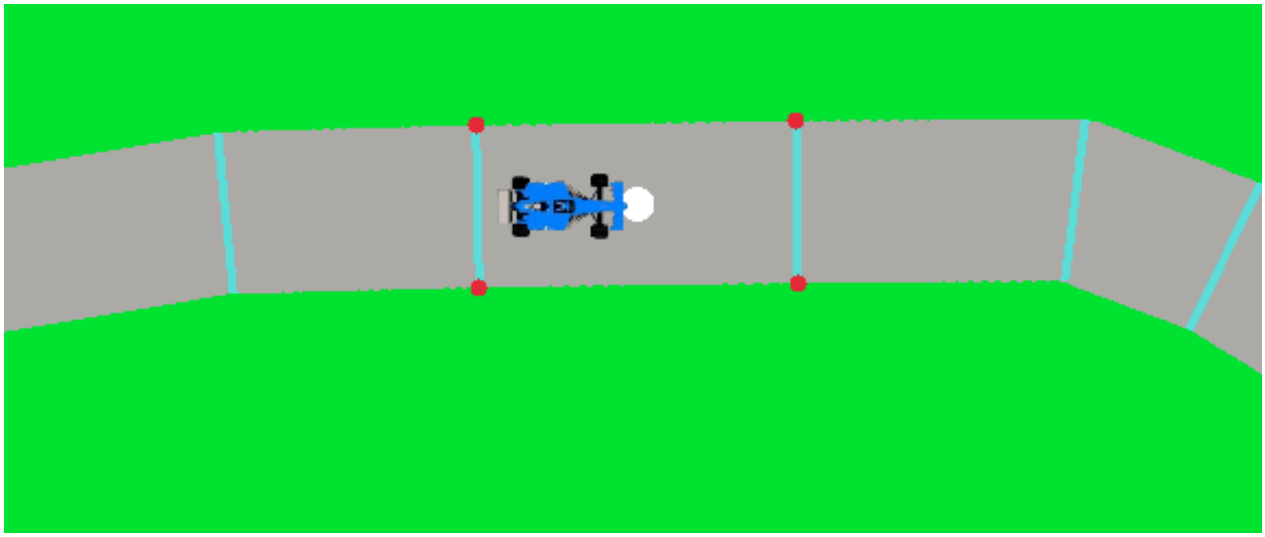
We need to find the points A, B, C, D from the diagram above, given points p1 and p2 which are the sector points.

$$L_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}, \qquad L_2 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} + u \begin{bmatrix} x_4 - x_3 \\ y_4 - y_3 \end{bmatrix}$$

L2 is the sector boundary line and we have the points x3 and x4 so can complete the form. However, for L1 (the ray line) x1, y1 is the center of the car hitbox but instead of doing x2-x1 and y2-y1 we can just find a direction vector from the ray's angle instead of finding a second point on the ray line.

```
let track_width: f32 = track.get_width();
let sector: usize = self.get_sector(track) as usize;
let points: &[Vec2; 20] = track.get_points();
let p1: Vec2 = points[sector];
let p2: Vec2 = points[(sector + 1) % points.len()];

let direction: Vec2 = p2 - p1;
let normal: Vec2 = vec2(x: -direction.y, y: direction.x).normalize();

let A: Vec2 = p1 + normal * (track_width / 2.0);
let B: Vec2 = p2 + normal * (track_width / 2.0);
let C: Vec2 = p1 - normal * (track_width / 2.0);
let D: Vec2 = p2 - normal * (track_width / 2.0);
```

I wrote this code to find points A, B, C and D in the diagram above. Lets draw these points to the screen to test if it works:



And it works a treat! No onto the maths:

So we need to calculate the values of **u** and **t**. It's just the the matter of copying down the formula on the right of the two below diagrams into the rust code.

$$u = -\frac{\begin{vmatrix} x_1 - x_2 & x_1 - x_3 \\ y_1 - y_2 & y_1 - y_3 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = -\frac{(x_1 - x_2)(y_1 - y_3) - (y_1 - y_2)(x_1 - x_3)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)},$$

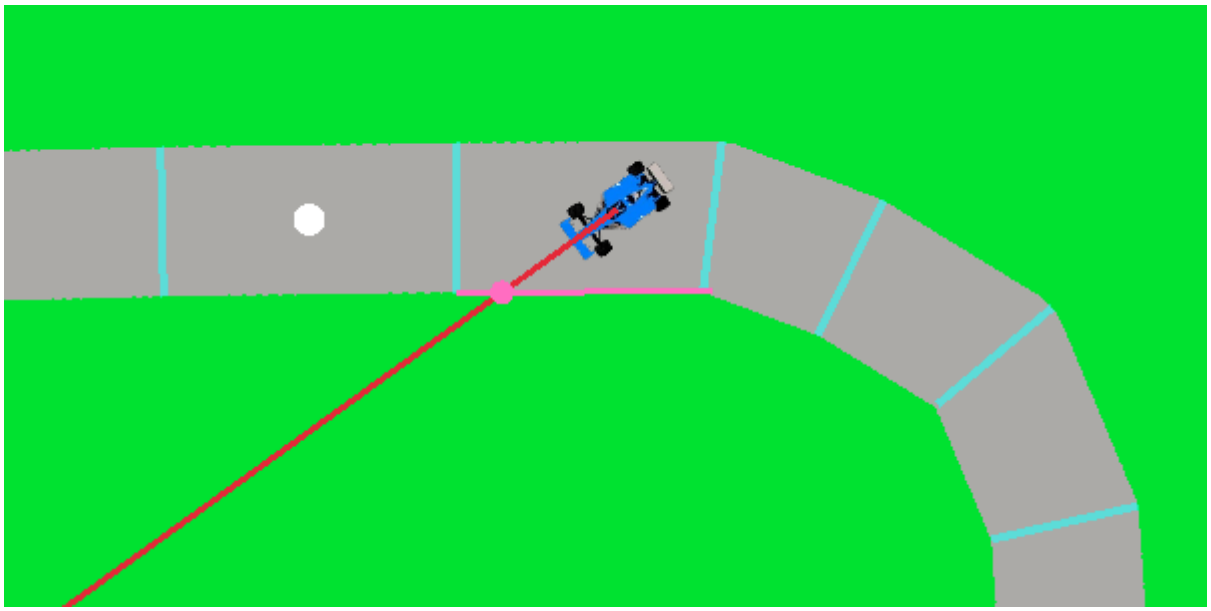
$$t = \frac{\begin{vmatrix} x_1 - x_3 & x_3 - x_4 \\ y_1 - y_3 & y_3 - y_4 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

$$(P_x, P_y) = \big(x_1 + t(x_2 - x_1),\ y_1 + t(y_2 - y_1)\big) \quad \text{or} \quad (P_x, P_y) = \big(x_3 + u(x_4 - x_3),\ y_3 + u(y_4 - y_3)\big)$$

Note: We can only check if 2-line sectors meet at a time. So, the code below will check the intersection between the ray and **A-B** line.

```rust
pub fn cast_ray(&self, track: &Track, ray_direction: Vec2) -> f32 {
    // returns distance to line sector

    let track_width: f32 = track.get_width();
    let sector: usize = self.get_sector(track) as usize;
    let points: &[Vec2; 20] = track.get_points();
    let p1: Vec2 = points[sector];
#   let p2: Vec2 = points[(sector + 1) % points.len()];

    let direction: Vec2 = p2 - p1;
    let normal: Vec2 = vec2(x: -direction.y, y: direction.x).normalize();

    let A: Vec2 = p1 + normal * (track_width / 2.0);
    let B: Vec2 = p2 + normal * (track_width / 2.0);
    let C: Vec2 = p1 - normal * (track_width / 2.0);
    let D: Vec2 = p2 - normal * (track_width / 2.0);

    draw_line(x1: A.x, y1: A.y, x2: B.x, y2: B.y, thickness: 3.0, color: PINK);

    let s1: Vec2 = self.rect.center();
    let s2: Vec2 = s1 + ray_direction * WINDOW_WIDTH as f32 * 5.0;

    draw_line(x1: s1.x, y1: s1.y, x2: s2.x, y2: s2.y, thickness: 3.0, color: RED);

    // check if ray hits sector AB
    let t: f32 = ((s1.x - A.x) * (A.y - B.y) - (s1.y - A.y) * (A.x - B.x))
        / ((s1.x - s2.x) * (A.y - B.y) - (s1.y - s2.y) * (A.x - B.x));

    let u: f32 = -(((s1.x - s2.x) * (s1.y - A.y) - (s1.y - s2.y) * (s1.x - A.x))
        / ((s1.x - s2.x) * (A.y - B.y) - (s1.y - s2.y) * (A.x - B.x)));

    let point_intersection: Vec2 = vec2(x: A.x + u * (B.x - A.x), y: A.y + u * (B.y - A.y));
    if (0.0 <= u && u <= 1.0) && (0.0 <= t && t <= 1.0) {
        draw_circle(point_intersection.x, point_intersection.y, r: 6.0, color: PINK);
    }

    0.0
} fn cast_ray
```

For testing, the current code will draw a point at the calculated point of intersection in pink on the screen.



As you can see the intersection code is currently working. However, its only checking intersection with one set of sector lines in the car's current sector (the one coloured in pink). But we need to check for intersections across the whole track and take the shortest intersection point as the one to return.

Algorithm

What needs to happen:

Starting on the current sector, check if the ray intersects with either of the sector's boundary lines. (It can only intersect with one of them)

If it does, save that point to intersection point variable and the ray's distance to the point to the maximum distance variable.

Go through all the other sectors on the track and if they intersect with the line **and** they intersect at a point closer to the car than the previous intersection points then save that new intersection point as the intersection point variable and its distance to the maximum distance variable.

The intersection point will be the point which gives the shortest distance to the car.

Writing it in the code

In order to keep the code, clean I wrote a function which handles the line intersection formula in the *utils.rs* file.

```rust
pub fn line_intersection(p1: Vec2, p2: Vec2, q1: Vec2, q2: Vec2) -> Option<Vec2> {
    let t: f32 = ((p1.x - q1.x) * (q1.y - q2.y) - (p1.y - q1.y) * (q1.x - q2.x))
        / ((p1.x - p2.x) * (q1.y - q2.y) - (p1.y - p2.y) * (q1.x - q2.x));

    let u: f32 = -(((p1.x - p2.x) * (p1.y - q1.y) - (p1.y - p2.y) * (p1.x - q1.x))
        / ((p1.x - p2.x) * (q1.y - q2.y) - (p1.y - p2.y) * (q1.x - q2.x)));

    if (0.0 <= t && t <= 1.0) && (0.0 <= u && u <= 1.0) {
        let point_intersection: Vec2 = vec2(x: q1.x + u * (q2.x - q1.x), y: q1.y + u * (q2.y - q1.y));
        return Some(point_intersection);
    }

    None
}
```

Here is the final completed function:

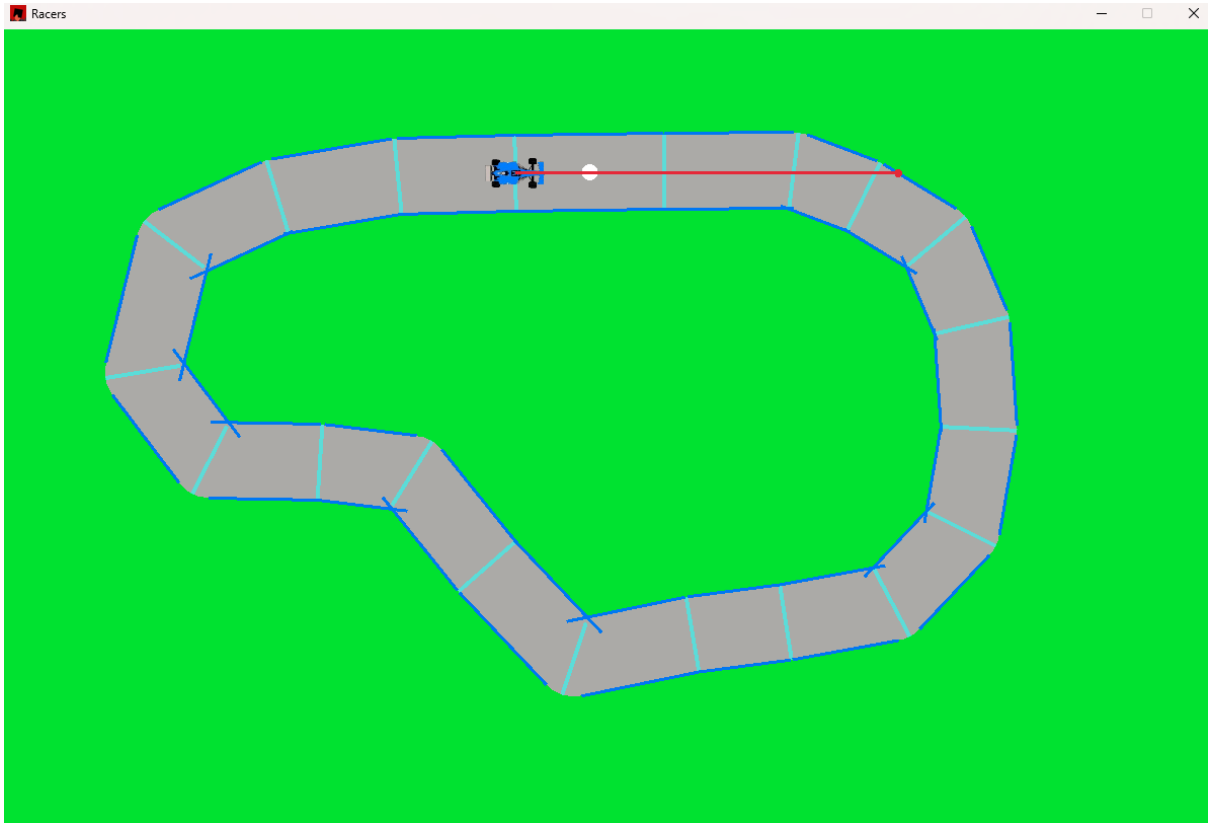
```rust
pub fn cast_ray(&self, track: &Track, ray_direction: Vec2) -> f32 {
    // returns distance to line sector

    let track_width: f32 = track.get_width();
    let current_sector: usize = self.get_sector(track) as usize;
    let points: &[Vec2; 20] = track.get_points();

    let s1: Vec2 = self.rect.center();
    let s2: Vec2 = s1 + ray_direction * WINDOW_WIDTH as f32 * 5.0;

    let mut shortest_interection_point: Vec2 = Vec2::MAX;
    let mut shortest_distance: f32 = f32::MAX;

    for i: usize in 0..points.len() {
        // loop through all the points
        let sector: Vec2 = points[(current_sector + i) % points.len()];
        let next_sector: Vec2 = points[(current_sector + i + 1) % points.len()];

        let direction: Vec2 = next_sector - sector;
        let normal: Vec2 = vec2(x: -direction.y, y: direction.x).normalize();

        let A: Vec2 = sector + normal * (track_width / 2.0);
        let B: Vec2 = next_sector + normal * (track_width / 2.0);
        let C: Vec2 = sector - normal * (track_width / 2.0);
        let D: Vec2 = next_sector - normal * (track_width / 2.0);

        if let Some(point1: Vec2) = line_intersection(p1: s1, p2: s2, q1: A, q2: B) {
            let distance: f32 = (point1 - s1).length();
            if distance < shortest_distance {
                shortest_interection_point = point1;
                shortest_distance = distance;
            }
        } else if let Some(point2: Vec2) = line_intersection(p1: s1, p2: s2, q1: C, q2: D) {
            let distance: f32 = (point2 - s1).length();
            if distance < shortest_distance {
                shortest_interection_point = point2;
                shortest_distance = distance;
            }
        }
    }

    draw_line(x1: s1.x, y1: s1.y, x2: s2.x, y2: s2.y, thickness: 3.0, color: RED);
    draw_circle(
        shortest_interection_point.x,
        shortest_interection_point.y,
        r: 4.0,
        color: RED,
    );
    shortest_distance
} fn cast_ray
```

Testing the code

After running the code, the line intersection seemed to work correctly. You can find the video of the running code at ***Documentation/Video/ray_cast_test.mkv***

However, as is visible in the video, sometimes the ray would not appear at all or it would end early etc. This wouldn't happen very often during the simulation but over time would probably negatively affect the AI learning process. So, it needs to be fixed.

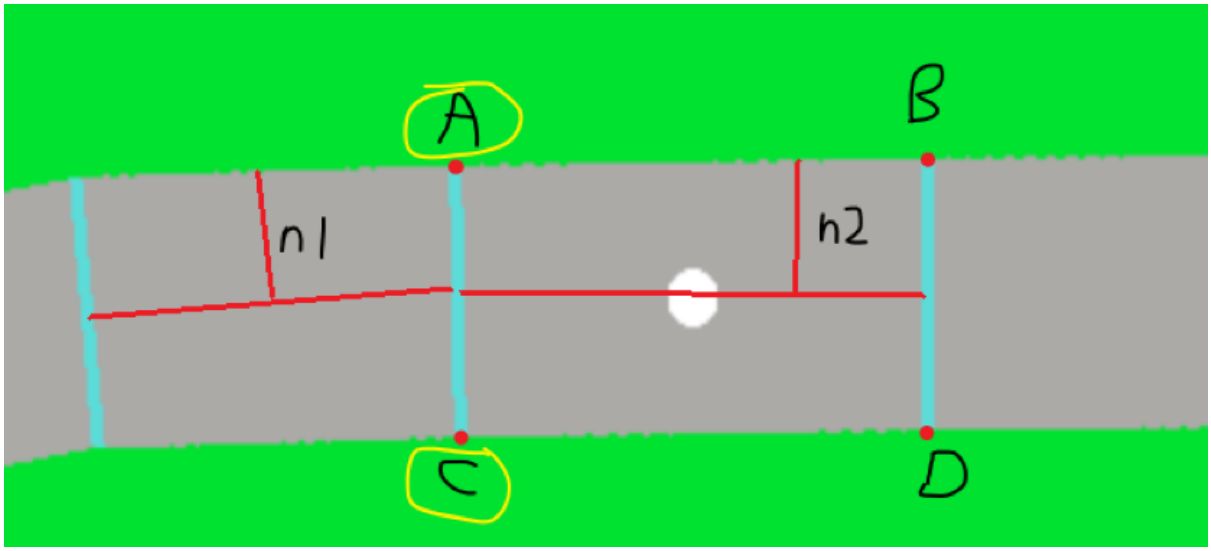
The problem was obvious when I drew the sector lines onto the track:



The error occurs because when calculating the sector corners, **A, B, C and D** the normal calculated only takes into account the next sector and not the previous ones or the next, next sector along.

```
let direction: Vec2 = next_sector - sector;
let normal: Vec2 = vec2(x: -direction.y, y: direction.x).normalize();

let A: Vec2 = sector + normal * (track_width / 2.0);
let B: Vec2 = next_sector + normal * (track_width / 2.0);
let C: Vec2 = sector - normal * (track_width / 2.0);
let D: Vec2 = next_sector - normal * (track_width / 2.0);
```

So, when calculating the normal to get the points **A** and **C** (the points on the left) You need to take the average normals of the 2 sector lines between which it lies as shown in the diagram below:

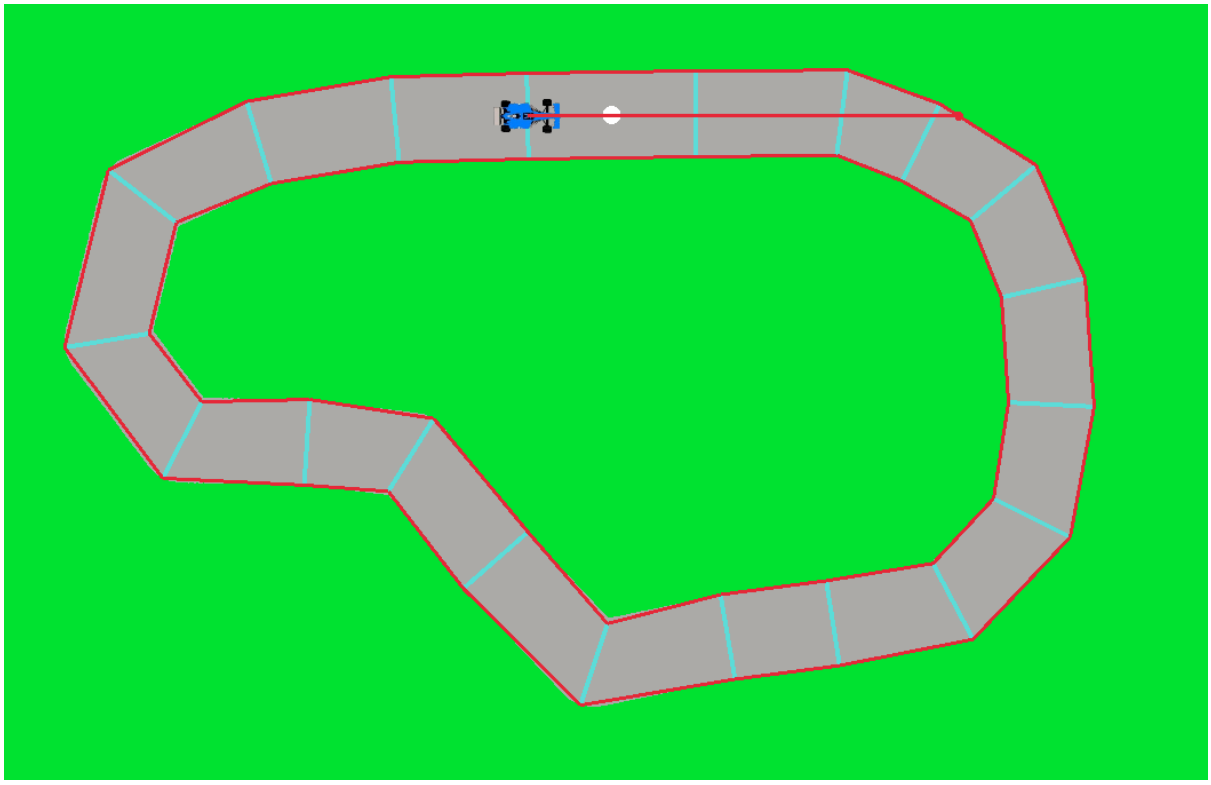


The points **A and C** will need to be found using the average normal vector of **n1** and **n2** and then the points **B and D** will need to be found using **n3 and n2**.

The code for this looks a big disgusting, the variables using the word **left** are used to describe the values needed to calculate **A and C** (the points to the left of the sector) and visa versa for the keyword **right**.

```
// finding points A, B, C, D
let sector_index: usize = (current_sector + i) % points.len();
let sector: Vec2 = points[sector_index];
let next_sector: Vec2 = points[(sector_index + 1) % points.len()];
let prev_sector: Vec2 = points[(sector_index + (points.len() - 1)) % points.len()];
let next_next_sector: Vec2 = points[(sector_index + 2) % points.len()];

// finding normal for A and C
let direction1_left: Vec2 = sector - prev_sector;
let direction2_left: Vec2 = next_sector - sector;
let normal1_left: Vec2 = vec2(x: -direction1_left.y, y: direction1_left.x);
let normal2_left: Vec2 = vec2(x: -direction2_left.y, y: direction2_left.x);
let avg_normal_left: Vec2 = ((normal1_left + normal2_left) / 2.0).normalize();

// finding normal for B and D
let direction1_right: Vec2 = next_sector - sector;
let direction2_right: Vec2 = next_next_sector - next_sector;
let normal1_right: Vec2 = vec2(x: -direction1_right.y, y: direction1_right.x);
let normal2_right: Vec2 = vec2(x: -direction2_right.y, y: direction2_right.x);
let avg_normal_right: Vec2 = ((normal1_right + normal2_right) / 2.0).normalize();

// calculating points
let A: Vec2 = sector + avg_normal_left * (track_width / 2.0);
let B: Vec2 = next_sector + avg_normal_right * (track_width / 2.0);
let C: Vec2 = sector - avg_normal_left * (track_width / 2.0);
let D: Vec2 = next_sector - avg_normal_right * (track_width / 2.0);
```

The Result:



Wow! What a beauty! Now with this fix, the sector lines are defined with much more accuracy. Now the problems discussed above will no longer occur.

cast_ray whole function:

```rust
pub fn cast_ray(&self, track: &Track, ray_direction: Vec2) -> f32 {
    // returns distance to line sector

    let track_width: f32 = track.get_width();
    let current_sector: usize = self.get_sector(track) as usize;
    let points: &[Vec2; 20] = track.get_points();

    let s1: Vec2 = self.rect.center();
    let s2: Vec2 = s1 + ray_direction * WINDOW_WIDTH as f32 * 5.0;

    let mut shortest_interection_point: Vec2 = Vec2::MAX;
    let mut shortest_distance: f32 = f32::MAX;

    for i: usize in 0..points.len() {
        // finding points A, B, C, D
        let sector_index: usize = (current_sector + i) % points.len();
        let sector: Vec2 = points[sector_index];
        let next_sector: Vec2 = points[(sector_index + 1) % points.len()];
        let prev_sector: Vec2 = points[(sector_index + (points.len() - 1)) % points.len()];
        let next_next_sector: Vec2 = points[(sector_index + 2) % points.len()];

        // finding normal for A and C
        let direction1_left: Vec2 = sector - prev_sector;
        let direction2_left: Vec2 = next_sector - sector;
        let normal1_left: Vec2 = vec2(x: -direction1_left.y, y: direction1_left.x);
        let normal2_left: Vec2 = vec2(x: -direction2_left.y, y: direction2_left.x);
        let avg_normal_left: Vec2 = ((normal1_left + normal2_left) / 2.0).normalize();

        // finding normal for B and D
        let direction1_right: Vec2 = next_sector - sector;
        let direction2_right: Vec2 = next_next_sector - next_sector;
        let normal1_right: Vec2 = vec2(x: -direction1_right.y, y: direction1_right.x);
        let normal2_right: Vec2 = vec2(x: -direction2_right.y, y: direction2_right.x);
        let avg_normal_right: Vec2 = ((normal1_right + normal2_right) / 2.0).normalize();

        // calculating points
        let A: Vec2 = sector + avg_normal_left * (track_width / 2.0);
        let B: Vec2 = next_sector + avg_normal_right * (track_width / 2.0);
        let C: Vec2 = sector - avg_normal_left * (track_width / 2.0);
        let D: Vec2 = next_sector - avg_normal_right * (track_width / 2.0);

        if let Some(point1: Vec2) = line_intersection(p1: s1, p2: s2, q1: A, q2: B) {
            let distance: f32 = (point1 - s1).length();
            if distance < shortest_distance {
                shortest_interection_point = point1;
                shortest_distance = distance;
            }
        } else if let Some(point2: Vec2) = line_intersection(p1: s1, p2: s2, q1: C, q2: D) {
            let distance: f32 = (point2 - s1).length();
            if distance < shortest_distance {
                shortest_interection_point = point2;
                shortest_distance = distance;
            }
        }
    }

    return shortest_distance;
```

Now let’s cast multiple rays:

Since the ray casting is isolated in its own function, it should be easy to cast multiple of them:

The cast ray's function arguments:

- Number of rays
- Car's field of view – FOV
- Reference to track object

The code in pseudo code:

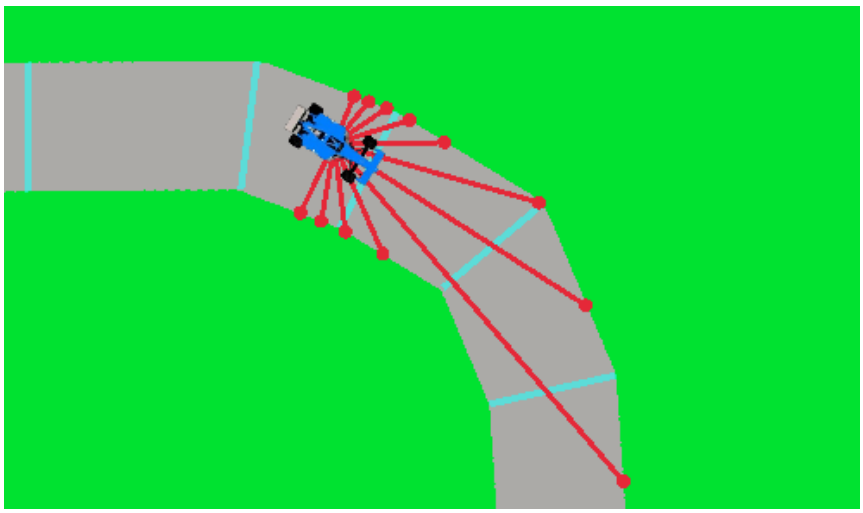
```
procedure cast_rays(rays, fov, track by ref)
    start_angle = degrees(self.angle) - fov / 2
    step = fov / rays
    for i=0 to rays-1
        direction = Vector.from_angle(start_angle + step * i)
        self.cast_ray(ref(track), direction)
    next i
end procedure
```

Note: The FOV argument will be measured in degrees, so I need to translate the car's angle attribute into degrees before doing any calculating.

The code in rust:

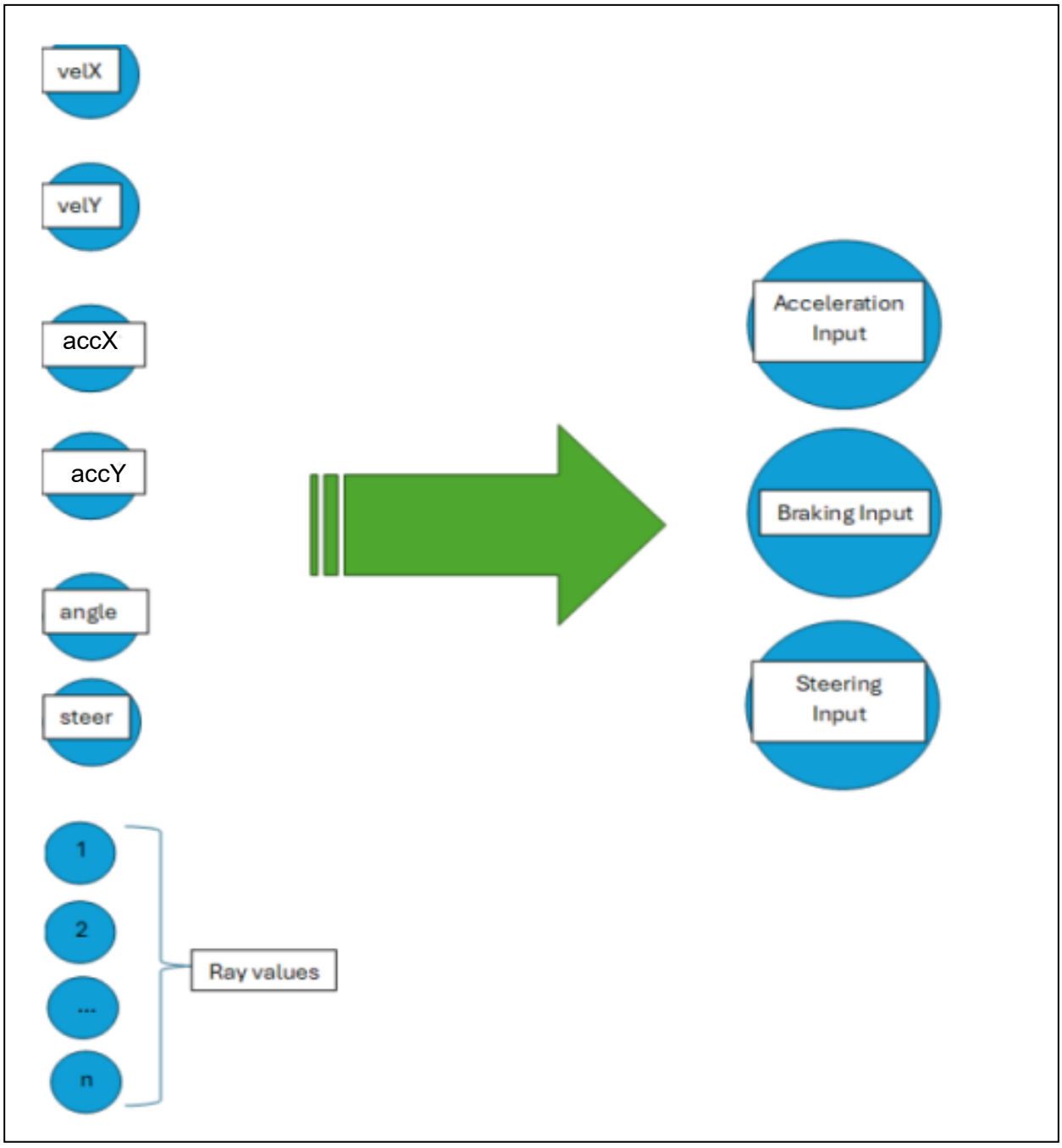
```rust
pub fn cast_rays(&self, rays: usize, fov: f32, track: &Track) {
    // fov in degrees
    let start_angle: f32 = self.angle.to_degrees() - fov / 2.0;
    let step: f32 = fov / rays as f32;

    for ray: usize in 0..rays {
        let angle: f32 = start_angle + step * ray as f32;
        let dir: Vec2 = Vec2::from_angle(angle.to_radians());
        self.cast_ray(track, ray_direction: dir);
    }
}
```

The result:



Implementing the Car's Neural Network

Inputs	Outputs
• Ray casting sensors to sense the edge of the track, should be cast in a range of directions around the car • The car's x velocity • The car's y velocity • The car's angle • The car's steer value • The car's acceleration x • The car's acceleration y	• Acceleration_input • Brakes_input • Steering_input



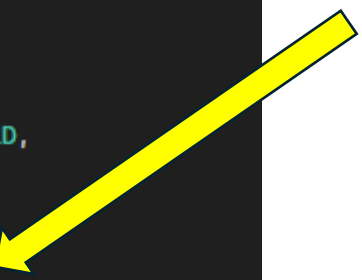
Building the neural network

Layers:

- Input layer – **6 + (number of rays)** neurons, no activation
- Hidden Layer 1 – **8 neurons**, no activation
- Hidden Layer 2 – **5 neurons**, no activation
- Output Layer – **3 neurons** activation sigmoid

So we need to add a **brain** to the car class

```rust
pub struct Car {
    // Physics variables
    // -- Vectors
    velocity: Vec2,
    pub direction: Vec2,
    position: Vec2,
    acceleration: Vec2,
    // -- Scalar
    angle: f32,
    steer: f32,

    // Graphics
    texture: Texture2D,
    rect: Rect,

    // network
    brain: Network,

    // inputs for controllers
    accelerator_input: Input,
    steering_input: Input, // radians
    brakes_input: Input,
}
```



Network Initalization

```rust
let mut brain: Network = Network::new_empty();
brain = brain Network
    .add_layer(Layer::new_random(inputs: 6 + Car::RAYS, outputs: 8, activation: None)) Network
    .add_layer(Layer::new_random(inputs: 8, outputs: 5, activation: None)) Network
    .add_layer(Layer::new_random(inputs: 5, outputs: 3, activation: Some(sigmoid)));
```

So that's our network created, now I need to generate the network's input every frame to get the car's movements.

The inputs need to be normalized between passing them into the neural network, i.e. change them to values between -1 and 1.

I modified the **cast_rays** function to return a vector (list) of the ray distances

```rust
pub fn cast_rays(&self, rays: usize, fov: f32, track: &Track) -> Vec<f32> {
    // fov in degrees

    let mut ray_list: Vec<f32> = vec![];

    let start_angle: f32 = self.angle.to_degrees() - fov / 2.0;
    let step: f32 = fov / rays as f32;

    for ray: usize in 0..rays {
        let angle: f32 = start_angle + step * ray as f32;
        let dir: Vec2 = Vec2::from_angle(angle.to_radians());
        let distance: f32 = self.cast_ray(track, ray_direction: dir);
        // normalize the distance against the window width
        let normalized: f32 = distance / (WINDOW_WIDTH as f32);
        ray_list.push(normalized);
    }

    return ray_list;
}
```

```rust
pub fn update(&mut self, track: &Track) {
    let dt: f32 = get_frame_time();

    // run the neural network with inputs
    let rays: Vec<f32> = self.cast_rays(rays: 12, fov: 200.0, track);
```

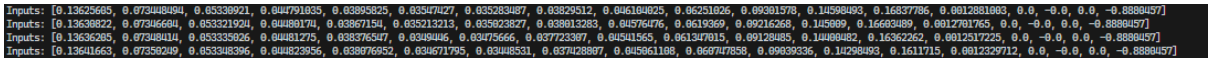
Note: that now the update function needs access to the track object, a reference to the track object must be passed to the update function – hence the new function signature.

```rust
// run the neural network with inputs
let rays: Vec<f32> = self.cast_rays(Car::RAYS, fov: 200.0, track);
let velx_norm: f32  = self.velocity.x / Car::MAX_SPEED;
let vely_norm: f32 = self.velocity.y / Car::MAX_SPEED;
let accx_norm: f32 = self.acceleration.x / Car::MAX_ACC;
let accy_norm: f32 = self.acceleration.y / Car::MAX_ACC;
let steer_norm: f32 = self.steer / Car::STEER_WEIGHT;
let angle_norm: f32 = (self.angle).sin();

let mut inputs: Vec<f64> = vec![];
for ray: &f32 in rays.iter() {
    inputs.push((*ray) as f64);
}
inputs.push(velx_norm as f64);
inputs.push(vely_norm as f64);
inputs.push(accx_norm as f64);
inputs.push(accy_norm as f64);
inputs.push(steer_norm as f64);
inputs.push(angle_norm as f64);
```

Figure 5-Collecting normalized inputs

Normalising most of the input data is quite easy since the class has constant MAX values for most physical quantities, and the angle will benormalised by passing it as sin(x) rather than the angle x.

I printed the inputs array to the console to ensure they were all normalized



If you zoom in enough you can see that they are all between -1.0 and 1.0

Running the network

Now we have the inputs in the correct format, we can pass it into the network and set its outputs to the corresponding car's input variables.

```
// run the network
let outputs: Vec<f64> = self.brain.run(inputs);
self.accelerator_input.weight = outputs[0] as f32;
self.steering_input.weight = ((outputs[1] - 0.5) * 2.0) as f32; // convert to value between -1.0 and 1.0
self.brakes_input.weight = outputs[2] as f32;
```

Even though its pretty useless because there's no population control its quite interesting to watch the current state of the AI.

The video is in ***Documentation/Videos/AI_first_run.mkv***

Development Stage 6

Recap

Stage Six: Implementing the genetic algorithm for taking 2 controller's and mixings their networks (plus a little mutation) to create a new controller.

Refining the population class

Current Class Diagram

Population
Attributes: + generation : int + cars: Car [] + track: Track
Methods: + draw() + update()

Draw and Update functions

```rust
pub fn update(&mut self) {
    for car: &mut Car in self.cars.iter_mut() {
        car.update(&self.track);
    }
}
```

```rust
pub fn draw(&self) {
    self.track.draw();
    // cast rays
    for car: &Car in self.cars.iter() {
        car.draw();
    }
}
```

What does the population update function need to do?

- Go through every car and call its individual **update** function
- If a car goes off the track, freeze it as if it crashed
- Calculate the fitness of each car
- Reset the population if the generation timer is up

```rust
for car: &mut Car in self.cars.iter_mut() {
    car.update(&self.track);
    if !car.is_on_track(&self.track) {
        car.crashed();
    }
}
```

This loop in the population update function, will update the cars and then if they are off the track then they call the setter function car.crashed() which sets the car's crashed boolean value to true:

```rust
pub fn crashed(&mut self) {
    self.crashed = true;
}
```

```rust
// others
crashed: bool,
```

Figure 6-Adding crashed attribute to the car class

I added a timer to the population class as well as a constant **generation time** value

```rust
Self {
    generation: 0,
    cars,
    track,
    timer: 0,
}
```

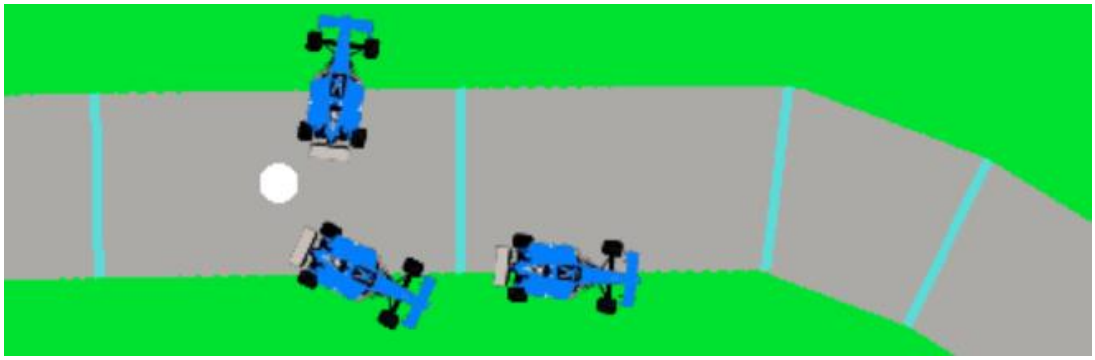
```rust
pub const GENERATION_TIME: u32 = 10000;
```

And the image below shows the **new_popluation** function which calls the reset function on every car object.

```rust
pub fn reset(&mut self, position: Vec2) {
    self.position = position;
    self.acceleration = Vec2::ZERO;
    self.velocity = Vec2::ZERO;
    self.angle = 0.0;
    self.steer = 0.0;
    self.direction = Vec2::ZERO;
    self.accelerator_input.weight = 0.0;
    self.brakes_input.weight = 0.0;
    self.steering_input.weight = 0.0;
    self.update_pos(self.position.x, self.position.y);
    self.crashed = false;
}
```

```rust
pub fn update(&mut self) {
    if self.timer >= GENERATION_TIME {
        self.new_population();
    }

    for car: &mut Car in self.cars.iter_mut() {
        car.update(&self.track);
        if !car.is_on_track(&self.track) {
            car.crashed();
        }
    }

    self.timer += 1;
}

fn new_population(&mut self) {
    // reset all the cars to start position
    let start_pos: Vec2 = self.track.get_start_pos();
    for car: &mut Car in self.cars.iter_mut() {
        car.reset(position: start_pos);
    }
    self.timer = 0;
}
```

Figure 7- Shows all 3 cars crashed well before the timer went off



RESULT: I ran the code and the timer was a little long and the cars would all crash before the timer was up. So I added a conditional statement which also resets the popluation if all the cars have crashed:

First, I have to create a function which determines if all the cars are crashed

```
fn all_cars_crashed(&self) -> bool {
    for car: &Car in self.cars.iter() {
        if !car.crashed {
            return false;
        }
    }
    return true;
}
```

```
if self.timer >= GENERATION_TIME  || self.all_cars_crashed() {
    self.new_population();
}
```

Here is a video of the current state of the project at

Documentation/Videos/three_car_population.mkv

As you can see in the video, the three cars' move the same way and thus crash in the same place every population, this is because they still share the same neural network from the previous generation therefore, they respond the exact same way to their environment

So, now its time to implement the reproduction part of the genetic algorithm.

The genetic algorithm steps

1) Pick the 2 well performing individuals
2) Perform a genetic 1-point-crossover on their weights and biases
3) Add some minor mutation to the weights and biases
4) Repeat until you have a new population of the same size

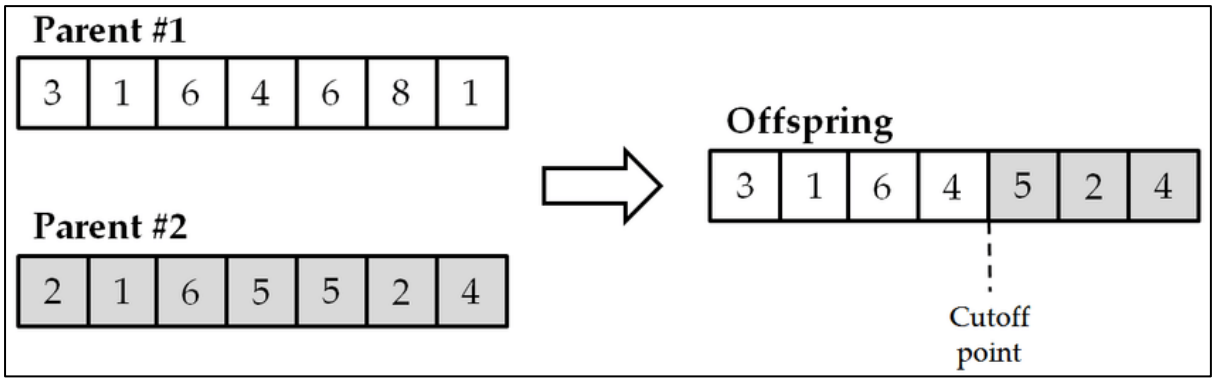


Figure 8- single point crossover diagram from https://www.researchgate.net/figure/One-point-crossover-operator-example_fig2_322351489

Genetic crossover algorithm in psuedo code

```
// genetic cross over

// there are 2 networks:
// network1 and network2

child_network = network1.copy()

for i=0 to child_network.layers.length() - 1
    layer2 = network2.layers[i];
    biases2 = layer2.biases
    weights2 = layer2.weights

    weights_size = weights.length() * weights[0].length()
    biases_size = biases.length()

    weights_crossover = random_between(0, weights_size - 1)
    biases_crossover = random_between(0, biases_size - 1)

    child_layer = child_network.layers[i]

    for j=0 to weights_crossover
        new_weight = weights2[weights_size DIV j][weights_size MOD j]
        child_layer.weights[weights_size DIV j][weights_size MOD j] = new_weight
    next j

    for k=0 to biases_crossover
        new_bias = biases2[k]
        child_layer.biases[k] = new_bias
    next k
next i
```

In rust:

```rust
fn reproduce(&self, car1: Car, car2: Car) -> Car {
    let mut child_car: Car = Car::new(self.track.get_start_pos());
    let mut child_net: Network = car1.brain.clone();
    let network2: Network = car2.brain.clone();

    for i: usize in 0..child_net.layers.len() {
        let layer2: Layer = network2.layers[i].clone();
        let biases2: Vec<f64> = layer2.bias;
        let weights2: Vec<Vec<f64>> = layer2.weights;

        let weights_size: usize = weights2.len() * weights2[0].len();
        let biases_size: usize = biases2.len();

        let weights_crossover: usize = gen_range(low: 0, high: weights_size - 1);
        let biases_crossover: usize = gen_range(low: 0, high: biases_size - 1);

        let mut child_layer: &mut Layer = &mut child_net.layers[0];

        // cross over the weights
        for j: usize in 0..=weights_crossover {
            let new_weight: f64 = weights2[weights_size / j][weights_size % j];
            child_layer.weights[weights_size / j][weights_size % j] = new_weight;
        }

        // cross over the biases
        for k: usize in 0..=biases_crossover {
            let new_bias: f64 = biases2[k];
            child_layer.bias[k] = new_bias;
        }
    }

    child_car.brain = child_net;

    return child_car;
} fn reproduce
```

Now lets sprinkle in some mutations to the weights and biases randomly

```rust
// apply mutations
for row: &mut Vec<f64> in child_layer.weights.iter_mut() {
    for weight: &mut f64 in row.iter_mut() {
        if gen_range(low: 0.0, high: 1.0) <= 0.02 {
            *weight = gen_range(low: -1.0, high: 1.0);
        }
        if gen_range(low: 0.0, high: 1.0) <= 0.03 {
            *weight += gen_range(low: -0.5, high: 0.5);
        }
    }
}

for bias: &mut f64 in child_layer.bias.iter_mut() {
    if (gen_range(low: 0.0, high: 1.0)) <= 0.02 {
        *bias = gen_range(low: -0.5, high: 0.5);
    }
    if (gen_range(low: 0.0, high: 1.0) <= 0.03) {
        *bias += gen_range(low: -0.5, high: 0.5);
    }
}
```

Chance of total mutation: 2%

Change of partial mutation: 3%

These mutation values are temporary for now, but after analyzing the algorithms efficiency these values may be tweaked later!

After writing this code I realise that the previous car's **reset** function is not necacary since we will be creating new fresh **Car** objects every time we create a new population. So I deleted that!

Simple fitness function:

A fitness function will determine the perfomance of a individual controller so that the best controllers can be chosen for reproduction.

Need to add a fitness value to the Car's object so that it can be continuously calculated.

```rust
    // others
    pub crashed: bool,
    pub fitness: f32,
}
```

```rust
crashed: false,
fitness: 0.0,
```

For now lets create a fitness function which only tracks how long the car has been alive for so the algorihtm will promote the longest survivors:

```rust
fn toll_fitness(&mut self) {
    if !self.crashed {
        self.fitness += 1.0;
    }
}
```

This will be called every frame!

And then a **final** fitness calculation:

```rust
fn get_final_fitness(&self) -> f32 {
    if self.crashed {
        return self.fitness + 100.0;
    }
    return self.fitness;
}
```

The new **new_population** function:

```rust
fn new_population(&mut self) {
    // reset all the cars to start position
    let size: usize = self.cars.len();
    let mut cars: Vec<Car> = vec![];

    // 100% of children made by top 2 performers
    self.cars Vec<Car>
        .sort_by(compare: |a: &Car, b: &Car| a.fitness.partial_cmp(&b.get_final_fitness()).unwrap());
    self.cars.reverse();

    for i: usize in 0..size {
        cars.push(self.reproduce(car1: self.cars[0].clone(), car2: self.cars[1].clone()));
    }

    self.cars = cars;

    self.timer = 0;
}
```

This function takes the 2 best perfomers and creates a new population by reproducing these two individuals **n** times. Note that line **61** is a one line solution way to sort a vector in rust which I found here. But it sorts it in ascending order when we want decending order so I reversed the list afterwards.

When I was running the code I got an division by zero **error**

I traced the code and found a possible division error in the reproduce function here:

```rust
let mut child_layer: &mut Layer = &mut child_net.layers[0];

// cross over the weights
for j: usize in 0..=weights_crossover {
    let new_weight: f64 = weights2[weights_size / j][weights_size % j];
    child_layer.weights[weights_size / j][weights_size % j] = new_weight;
}
```

The division should be the other way round, i.e.

- **j / weights_size**
- **j % weights size**

The issue is that the j value can take a zero value whereas the weight_size variable cannot take a zero value.

There is also another issue, the indexing through **errors** because I used the total size of the weights matrix rather than just the size of one row of it:

The size of one row of the weights matrix is given by **weights2[0].len(),** you can use the first index **0** because the matrix should be of uniform size.

Here is the new fixed code:

```
// cross over the weights
for j: usize in 0..=weights_crossover {
    let new_weight: f64 = weights2[j / weights2[0].len()][j % weights2[0].len()];
    child_layer.weights[j / weights2[0].len()][j % weights2[0].len()] = new_weight;
}
```

Now the simulation technically works, however its result is rather redundant. The cars will try to survive the longest, this is **not** the goal of the simulation. My point is that the *fitness function defines the behaviour of the simulation.* So now the algorithm's infrastructure is working, we can alter the **goals** of the simulation by modifying the car's fitness function.

The video of the simple **survival** simulation working can be found:

Documentation/Videos/survival_simulation1.mkv

RESULT

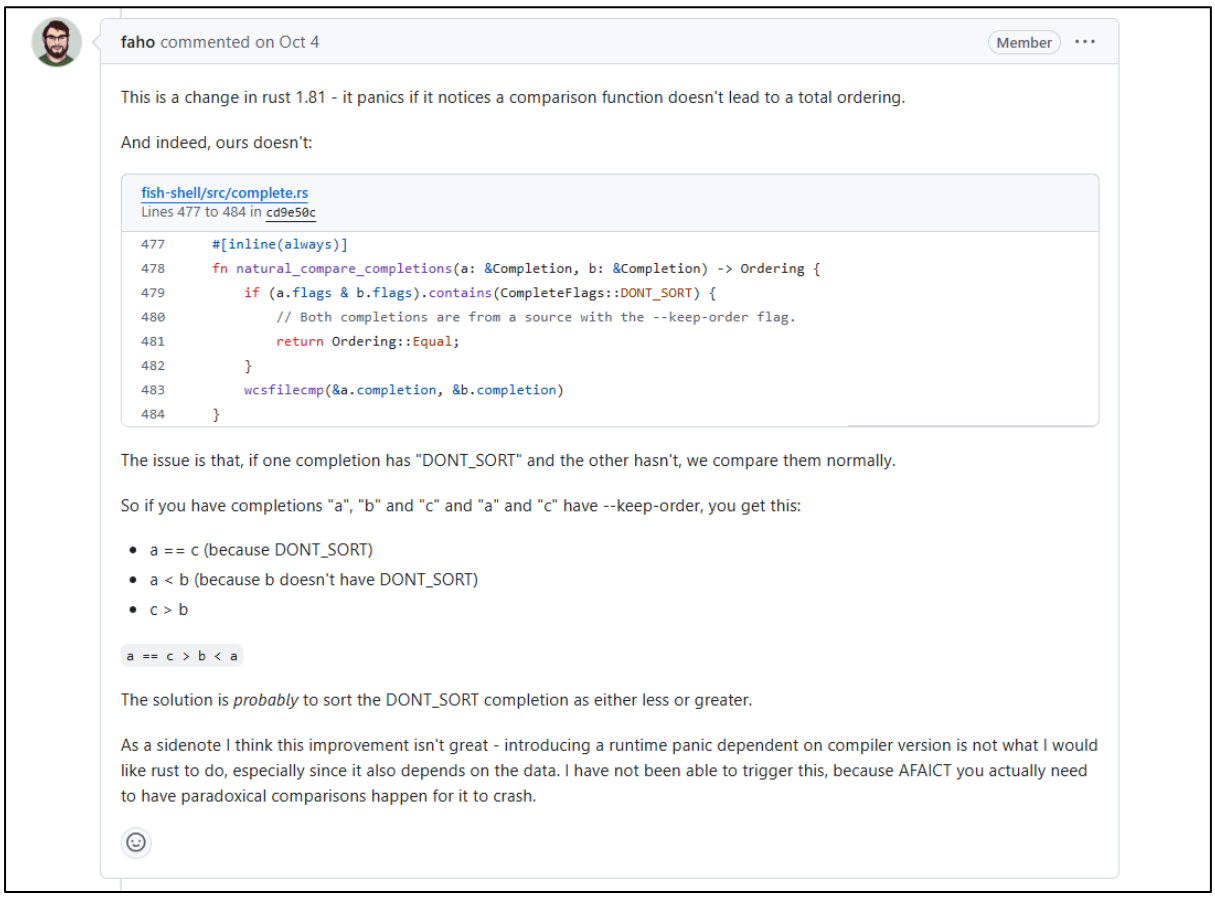
As evident in the video, the car's attempt to maximise their lifespan in a very visible way between generations. However, after a few generations the result is not very good since the population is so small.

Note: The ray tracing intersection points are still being drawn and I removed that now.

I'll rerun the simulation with a population of 25, and print the max fitness to the console after every generation. But every few generations I'd get this error:

```
thread 'main' panicked at library\core\src\slice\sort\shared\smallsort.rs:862:5:
user-provided comparison function does not correctly implement a total order
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\Racers.exe` (exit code: 101)
```

I searched up this error and found this on the github forums:



faho commented on Oct 4 — Member

This is a change in rust 1.81 - it panics if it notices a comparison function doesn't lead to a total ordering.

And indeed, ours doesn't:

fish-shell/src/complete.rs
Lines 477 to 484 in cd9e50c

```
477     #[inline(always)]
478     fn natural_compare_completions(a: &Completion, b: &Completion) -> Ordering {
479         if (a.flags & b.flags).contains(CompleteFlags::DONT_SORT) {
480             // Both completions are from a source with the --keep-order flag.
481             return Ordering::Equal;
482         }
483         wcsfilecmp(&a.completion, &b.completion)
484     }
```

The issue is that, if one completion has "DONT_SORT" and the other hasn't, we compare them normally.

So if you have completions "a", "b" and "c" and "a" and "c" have --keep-order, you get this:

- a == c (because DONT_SORT)
- a < b (because b doesn't have DONT_SORT)
- c > b

`a == c > b < a`

The solution is *probably* to sort the DONT_SORT completion as either less or greater.

As a sidenote I think this improvement isn't great - introducing a runtime panic dependent on compiler version is not what I would like rust to do, especially since it also depends on the data. I have not been able to trigger this, because AFAICT you actually need to have paradoxical comparisons happen for it to crash.

Figure 9 - https://github.com/fish-shell/fish-shell/issues/10763-

The problem is that in rust the float 32 bit types cannot be ordered (they don't exhibit the **Ord** trait) so the solution is to store fitness as an integer value instead of a float value, this makes more sense anyway (you can't have a fraction of a 'well done').

Changing the fitness attribute to a Int in the class's blueprint

```
// others
pub crashed: bool,
pub fitness: i32,
```

I also need to edit the return types of the fitness functions:

```rust
fn toll_fitness(&mut self) {
    if !self.crashed {
        self.fitness += 1;
    }
}

pub fn get_final_fitness(&self) -> i32 {
    if self.crashed {
        // return self.fitness + 100.0;
        return self.fitness - 500;
    }
    return self.fitness;
}
```

Note: When rewriting these functions, I noticed that before, the **get_final_fitness** function would add 100 fitness if the car had crashed instead of decrementing the fitness. This is shown by the commented-out code and its replacement below.

Now with this fixed, I reran the program and let it run for 5 minutes to see the results.

RESULT

However, after watching for a minute or two the cars were not improving at all really, then I printed out the list of cars which were sorted according to their fitness values.

```
Fitness: -407
Fitness: -399
Fitness: -388
Fitness: -432
Fitness: -448
Fitness: -424
Fitness: -405
Fitness: -321
Fitness: -332
Fitness: -442
```

It must be a logical error with the **one-line** sorting implementation in the **new_population** function. Simply re-writing the line to this form seemed to correctly sort the list:

```rust
self.cars Vec<Car>
    .sort_by(compare: |a: &Car, b: &Car| b.get_final_fitness().cmp(&a.get_final_fitness()));
```

Now the list is sorted:

```
Fitness: -312
Fitness: -323
Fitness: -352
Fitness: -366
Fitness: -400
Fitness: -401
Fitness: -408
Fitness: -409
Fitness: -409
Fitness: -417
Fitness: -419
Fitness: -422
Fitness: -424
Fitness: -433
```

Note: Since its now ordering **b against a** rather than **a against g** like before, there is no need for the reversal of the list after the sort.

Results of second 5-minute run:

Here is the state of the simulation after about 5 minutes of run time:

Documentation/Videos/full_run_1_result.mkv

End of run report:

After watching the results of the 5-minute run, there are 3 notable behaviours that have appeared:

- The cars appear to be moving very slowly along the race track. At first, I thought that my program might be lacking some fps but then I realised that the cars are actually driving at such low speeds in order to increase their life span and thus their fitness value. This is confirmation of the behaviour-fitness relationship since the population is finding ways to increase their potential fitness value. It's also a good sign that the genetic algorithm is working as expected and quite efficiently – since only after 5 minutes the car's had already nailed this technique.

- The cars attempt to make it round the race course rather than just sitting idly at the start. I suspected that after 5-minutes of training the AI would just let their car sit in the middle of the race track and not move, since that's a guaranteed way to boost fitness. However, most of the 25 cars in the population would attempt to move around the track without stopping – I believe that it is due to the fact that the nature of the neural network promotes non-zero inputs (due to non-zero biases and non-zero weights). I.e. its much harder for a NN to output the value 0.0 (as in **do not accelerate**) than it is to output a value like 0.2 (as in **accelerate a little bit**).

- The cars would make an effort to remain on the race track even when moving at slow speeds. This is because the fitness function punishes crashing (going off track) and encourages surviving, so naturally the AI controllers attempt to keep the cars inside the track limits to maximise their fitness values.

ISSUES

- The population now survives longer than the generation time, thus it will hit a maximum fitness value and not improve from there. Therefore, now I'm going to make the generation time limit very large so the cars will much sooner all crash than reach the time limit. This will promote continuous improvement.

Let's update the fitness function:

Here's a table which displays behaviours to encourage and behaviours to discourage

Encourage	Discourage

Staying on the track	Crashing
High average speed	
Reaching checkpoints	
Time it takes to reach checkpoints	
Finishing a lap	

New additions to car class:

- **timer - int** – to time sector times (incremented every frame, reset when car hits new sector)
- **cumulative_speed - float**– to calculate average speed
- **prev_sector - int** – a variable which saves the car's current sector, so we know when it reaches a new one

Additions **to toll_fitness** function

```
fn toll_fitness(&mut self, track: &Track) {
    if !self.crashed {
        // increase fitness while not crashed
        self.fitness += 1;

        // increase
        self.cumulative_speed += self.velocity.length();

        // if reached a NEXT checkpoint
        if self.get_sector(track) == (self.prev_checkpoint as i32 + 1) {
            self.prev_checkpoint += 1;
            self.fitness += 400;

            let sector_time: i32 = self.timer;
            self.fitness += 1 / sector_time * 1000;

            self.timer = 0;
        }
    }
}
```

Note: the function only rewards extra fitness when a car goes to the **NEXT** sector (so it can't gain fitness by going backwards). Also, the function adds an inverse of the sector time to the fitness value, so that going quicker through a sector rewards fitness point.

Additions to the **get_final_fitness** function

```rust
pub fn get_final_fitness(&self) -> i32 {
    let mut fitness: i32 = self.fitness;
    if self.crashed {
        // return self.fitness + 100.0;
        fitness -= 500;
    }
    let avg_speed: i32 = self.cumulative_speed as i32 / GENERATION_TIME as i32;
    fitness += avg_speed * 10;

    return fitness;
}
```

Now this function calculates average speed and awards fitness points accordingly.

The code for encouraging finishing a lap

```rust
} else if (sector == 0 && self.prev_checkpoint == last_sector) {
    // done a lap
    self.fitness += 1000; // bonus
    let sector_time: i32 = self.timer;
    self.fitness += 1 / sector_time * 1000;
    self.prev_checkpoint = 0;

    self.timer = 0;

    self.laps += 1;
}
```

The code awards a bonus for completing a lap, and also this code is required so that if a car completes a lap, its sector times will reset and it can continue gathering fitness from completing sectors.

Note: When the cars are placed at the start line, their current sector is set to the final sector so as soon they move, they will ‘complete a lap’ thus the cars are rewarded for moving off the start line. However, it also means that the car’s lap counter will be **1** if the car moves off the line.

Testing and Fixing

For testing purposes I'm going to export the best fitness data from each generation into a csv file so I can analyse it in excel:

The rust code:

```
Self {
    generation: 0,
    cars,
    track,
    timer: 0,
    data_file: File::create(path: "fitness_values_test1.csv").unwrap(),
}
```

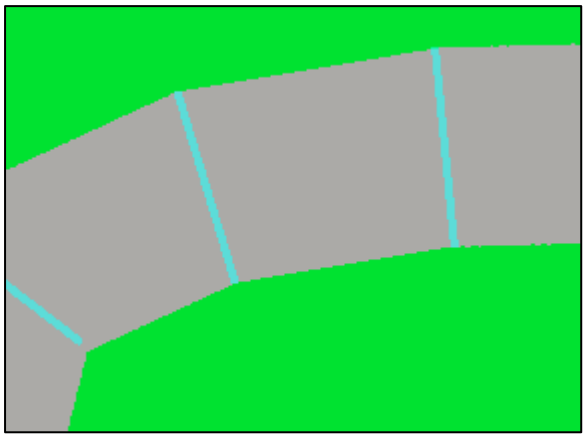
This creates a csv file object which can be accessed inside the code.

```
// add data to csv file
let best_fitness: i32 = self.cars[0].get_final_fitness();
writeln!(self.data_file, "{},{}", self.generation, best_fitness).unwrap();
```

This code writes the data for the current generation to the csv file. I also had to add the code to increment the **self.generation** attribute every time **new_population** has been called.

For each test run I'm going to create a test report which demonstrates the results of the simulation and the potential improvements of the simulation code.

Note, For the following tests the track width is no 100px which looks like this

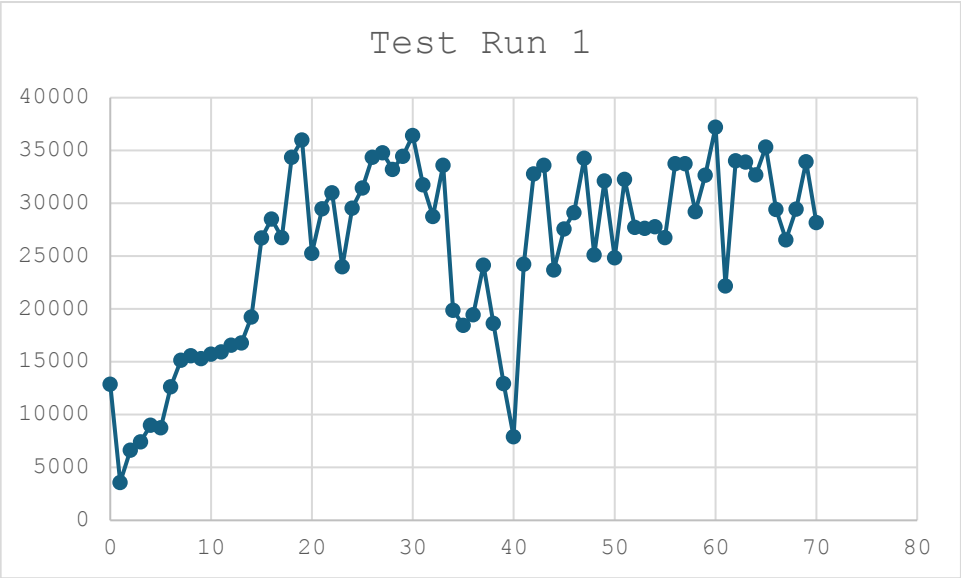


Test Run 1:

Test settings:

- Generation size: 100
- Car rays: 20
- Generation Time: 1000 ticks
- Data Length: 70 generations

Car performance graph:



Time lapse footage here:

Documentation/Videos/TestRun1.mp4

Analysing Results:

The cars made great progress in the first 20 generations probably due to them discovering all the ways to gain fitness. However, after generation 20 the cars improvement stagnates and also is heavily volatile for the rest of the simulation – without improving very much. The cars managed to finish a lap very quickly into the simulation however they didn't really learn how to take the corners very quickly or accelerate hard on the straights. Out of the 100 cars, the majority of them crash moments into the generation whereas only a few of them are still racing when the timer runs out. This means that during the cross over and mutation algorithm most of the car's networks are being altered such that they cannot run effectively anymore. I believe this has something to do with the nature of my cross-over function (which cross overs the weights and biases in every layer rather than just crossing over the **layer** objects), since this would create layers which largely mismatch with each other. Finally, during the middle generations, some cars were exploiting the 'finish lap' and 'finish sector' rewards by spinning in circle or going backwards round the track. So, I need to discourage going backwards! I also think that the car's intelligence is limited by the size of its network so perhaps adding another hidden layer would increase learning efficiency.

Discouraging moving backwards through the sectors:

```
} else {
    if self.prev_checkpoint == 0 && sector == last_sector as i32 {
        // gone backwards past the finish line
        self.timer = 0;
        self.prev_checkpoint = last_sector;
        self.fitness -= 5000; // DONT GO BACKWARDS
    } else if sector < self.prev_checkpoint as i32 {
        // going backwards
        self.timer = 0;
        self.prev_checkpoint = sector as usize;
        self.fitness -= 5000;
    }
}
```

This else clause added has been added to the **toll_fitness** function, it docks fitness by 5000 points if the car is going backwards

Also, for this function to work properly we need to change the car's starting position to in the centre of the first sector (otherwise the program thinks that the car is in the final sector when the simulation beings):

```
pub fn get_start_pos(&self) -> Vec2 {
    let pos: Vec2 = self.points_set[0];
    let pos1: Vec2 = self.points_set[1];

    return (pos + pos1) / 2.0;
}
```

Adding layer to network

```
let mut brain: Network = Network::new_empty();
brain = brain.Network
    .add_layer(Layer::new_random(inputs: 6 + Car::RAYS, outputs: 12, activation: None)).Network
    .add_layer(Layer::new_random(inputs: 12, outputs: 8, activation: None)).Network
    .add_layer(Layer::new_random(inputs: 8, outputs: 5, activation: None)).Network
    .add_layer(Layer::new_random(inputs: 5, outputs: 3, activation: Some(sigmoid)));
```

This adds another layer between the input layer and the first hidden layer! It should add a layer of potential complexity to the car's learning ability.

Tweaking Fitness Values

1) I increased the fitness bonus for **sector time** which should encourage the cars to run quicker
2) I increased the fitness decrease when the car has crashed to discourage crashing
3) Increased the bonus for high **average speeds**

```
let avg_speed: i32 = self.cumulative_speed as i32 / GENERATION_TIME as i32;
fitness += avg_speed * 500;
```

```
if self.crashed {
    fitness -= 50000;
}
```

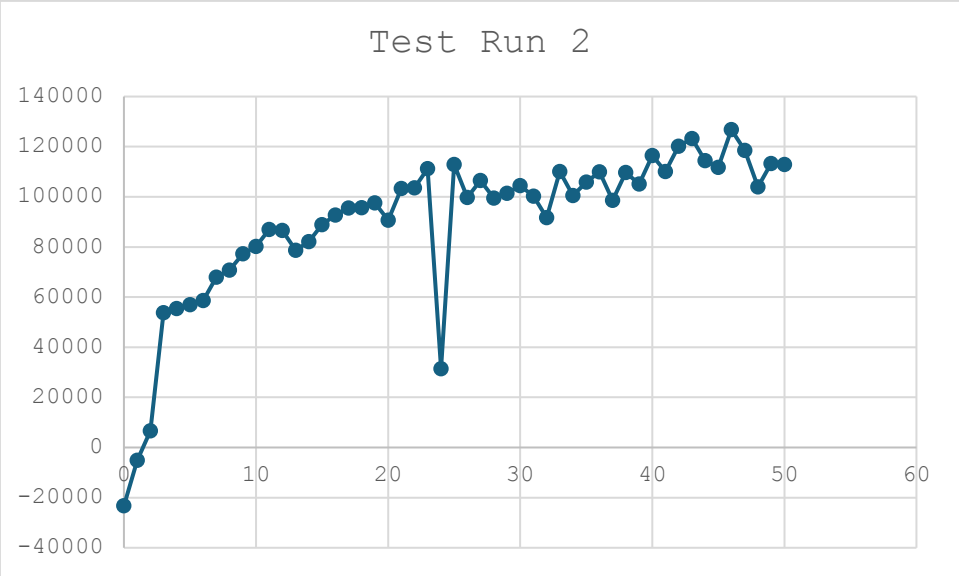
```
let sector_time: i32 = self.timer;
let speed_bonus: i32 = (300000.0 * (1.0 / (sector_time as f32).powf(1.5))) as i32;
```

Test Run 2:

Test settings:

- Generation size: 150
- Car rays: 20
- Generation Time: 1500 ticks
- Data Length: 50 generations

Car performance graph:



Time lapse footage here:

Documentation/Videos/TestRun2.mp4

Note: I also made cars that have crashed be drawn using a slightly transparent hue so that they don't clutter the screen as much.

Analysing Results:

These results were quite promising however there were some clear issues. Again, out of the 150 cars a lot of them would crash at the start (again I think it's a natural problem with the crossover step during network reproduction). However, the fitness function seemed more promising – at first it would encourage making it to the checkpoints and not crashing (for the top performers) and quite quickly the cars could complete a lap. Once, they had completed a lap they look to improve their sector times so that they can earn more fitness points and make it further along the track.

The shape of the graph is far smoother than test run 1, however you could argue that the top controllers from test run 1 were faster than those from this test run. But the common feature in both the test runs is that after a quick initial learning phase the cars fitness values oscillate (i.e. a generation makes some progress and then the next generation loses that progress). This is a relatively natural occurrence in a genetic algorithm approach (see this one https://i.stack.imgur.com/jnGvT.png). However, there are some improvements to be made.

Conclusion:

The results from these two tests are not perfect, however they facilitate the improvement of the generation over time. In other words, the algorithm **works!**

There are certainly some steps towards improvement that can be taken, i.e. various tweaks and tests to the **cross over** algorithm and more tweaking of the fitness function. However, down the line there is *Development Stage 10* which will tackle all this improvement along with further testing of the genetic algorithm.

Therefore, we have completed the requirements for this stage of development and can move on.

Current Analysis

But before we move on, lets list some of the current issues which will offer context for when we eventually reach **Stage 10**:

- Many cars will crash at the start no matter what, due to the cross-over function.
- After, a while the population seems to reach a plateau where it continues learning in cycles but never shows much forward progress
- Some cars seem to reap rewards by continuously spinning around a checkpoint, thus grabbing the rewards – for that checkpoint – repeatedly in quick succession. This should be discouraged.

Development Stage 7

Recap

1. **Stage Seven:** Upgrading car physics for **barrier detections, gears, rpm tracking**. These new features will be implemented into the outputs of the **neural network**.

Note 1: After coding the bare-bones physics of the car earlier in development, I have made the decision to **not** implement a **rpm** system on top of the current *acceleration-speed* system. This is because it would over complicate the code and it involves a dangerous step into realism which would inevitably lead to further complications. So, I strongly believe that the best way forward is to **scrap** this feature and instead implement the **gears** using a simpler approach (only involving the speed of the car rather than the **rpm**).

Note 2: I have also already implemented **barrier detections** earlier in development stage three whilst creating the **track** class. It seemed more appropriate and natural (in the flow of development) to implement barrier collisions back then.

The Revised Development Stage 7 Plan

Due to the unexpected circumstances explained above, I feel it necessary to revise the plan for **Stage Seven** of development here. Note that I will keep the original plan (on page 6) unaltered for reader clarity.

Revised Development Stage 7 Plan

Stage Seven: Upgrade the car's physics to be more realistic, involving the implementation of a **gearing system** which will add a layer of complexity to the cars learning process. The car network inputs and outputs should be altered to facilitate these new changes.

Lateral Friction

In its current state, the car's physics produce a rather severe problem. The cars can slide around the track as if their wheels were just one large ice cube. The solution to this (as I found on https://gamedev.stackexchange.com/questions/26845/i-am-looking-to-create-realistic-car-movement-using-vectors) is to implement lateral friction to the car's velocity ever frame. This can be done quite simply using some vector maths.

We need to:

1) Calculate the component of the velocity vector which is acting perpendicular (lateral) to the car's direction.
2) Calculate lateral friction as a fixed scalar multiplied by this component
3) Take away the lateral friction calculated above from the car's velocity every frame

Algorithm Design in Pseudo Code

```
// find perpendicular direction vector
perp_direction = direction.perpendicular()

// the magnitude of velocity in the perpendicular direction:
lateral_velocity_mag = velocity.dot(perp_direction)

// to get a vector quanitity
lateral_velocity = lateral_velocity_mag * perp_direction


lateral_friction = -lateral_velocity * FRIC_COEF


// apply friction to velocity
// ...
```

The code in rust

```
let perp_direction: Vec2 = self.direction.perp();
let lateral_velocity: Vec2 =
    self.velocity.dot(perp_direction.normalize()) * perp_direction.normalize();
let lateral_fric: Vec2 = -lateral_velocity * LAT_FRIC_COEF;

// apply frictions
self.velocity += (normal_fric) * dt;
self.velocity += (lateral_fric) * dt; // apply lateral friction
```

The dot produce of the car's velocity and the perpendicular direction (normalized) gives us the magnitude of the velocity acting in the

perpendicular direction. Then we need to multiply it with the perpendicular direction vector to get back a **vector** value which denotes the component of the velocity in the car's perpendicular direction, or its **lateral velocity**.

Then we multiply this vector by a frictional coefficient defined here:

```
// consts
const FRIC_COEF: f32 = 0.88;
const LAT_FRIC_COEF: f32 = 0.9;
```

And finally negate this value and add it to the current velocity (in order works, take it away from the current velocity).

The Result:

I ran the simulation with a small population size to properly see the effect.

Find the video in ***Documentation/Videos/lateral_friction.mkv***

The cars now move in a much more realistic fashion than before – compare this movement to the movement shown in the **TestRun1** and **TestRun2** footage.