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	Critical	How the AI **learns** this is perhaps the most critical feature, its

	to optimize the network's weights and sizes.		required for the simulation to 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:*

1) The simulation is easily decomposable into a series of features which need to interface with each other to run

2) Large amounts of arithmetic operations need to be run in milliseconds, perfect for a computer. This includes the genetic algorithm manipulating the neural network's parameters and vast linear algebra calculations required to 'run' through the neural network. These operations happen hundreds of times a second for every controller on the screen – therefore a computer is not only useful but **required** to run this experiment.

3) Visualization is an integral component to my project since the user's must be able to watch along as the AI controllers learn and gradually improve over time, without a UI this would be impossible. With a computer drawing the environment and UI to the screen is a relatively trivial task. Additionally with the aid of IO devices the user will be able to interact with the simulation and ultimately remain engaged.

Development Plan

My Development Plan – 12 Small steps

1. **Stage One**: Set up and configure the graphics library to run without errors and open a window with correct settings. COMPLETE
2. **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.

3. **Stage Three:** Create a *track* object which handles track limits, drawing itself and checkpoints (which divide the track into sectors).
4. **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.
5. **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.
6. **Stage Six:** Implementing the genetic algorithm for taking 2 controller's and mixings their networks (plus a little mutation) to create a new controller.
7. **Stage Seven:** Upgrading car physics for **barrier detections, gears, rpm tracking**. These new features will be implemented into the outputs of the **neural network**.
8. **Stage Eight:** Create population object which handles all the racer's currently racing around the circuit. In charge of setting starting points, restarting crashed cars, managing sector times and giving feedback (**fitness marks**) to cars. This should keep track of best controllers and chose them for reproduction for next generation.
9. **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.
10. **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.
11. **Stage 11** – Implement the **RACE** feature where users can race against selected AI across a lap (pausing progress of population for that time).
12. **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

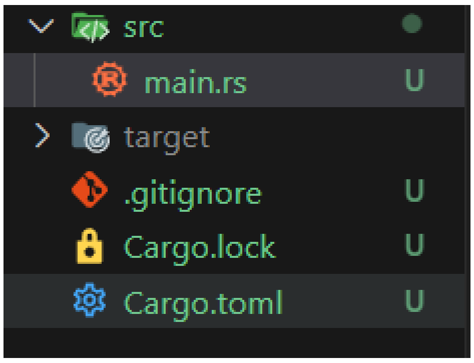
Development environment set up, library initialization and window setup.

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

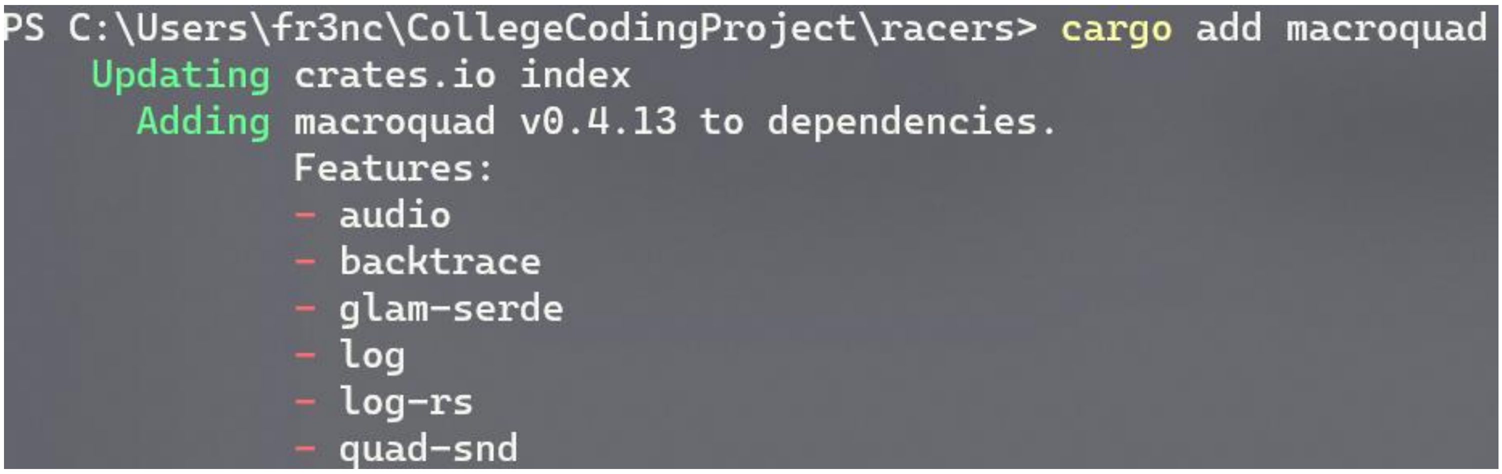
1. `> cargo new racers` Create a project using **cargo** tool

This is what the project tree looks like after being created



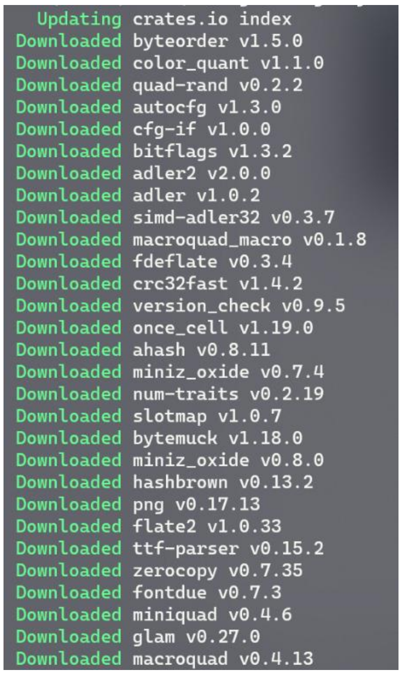
```
src
  main.rs        U
target
.gitignore       U
Cargo.lock       U
Cargo.toml       U
```

2. **Add the macroquad library**



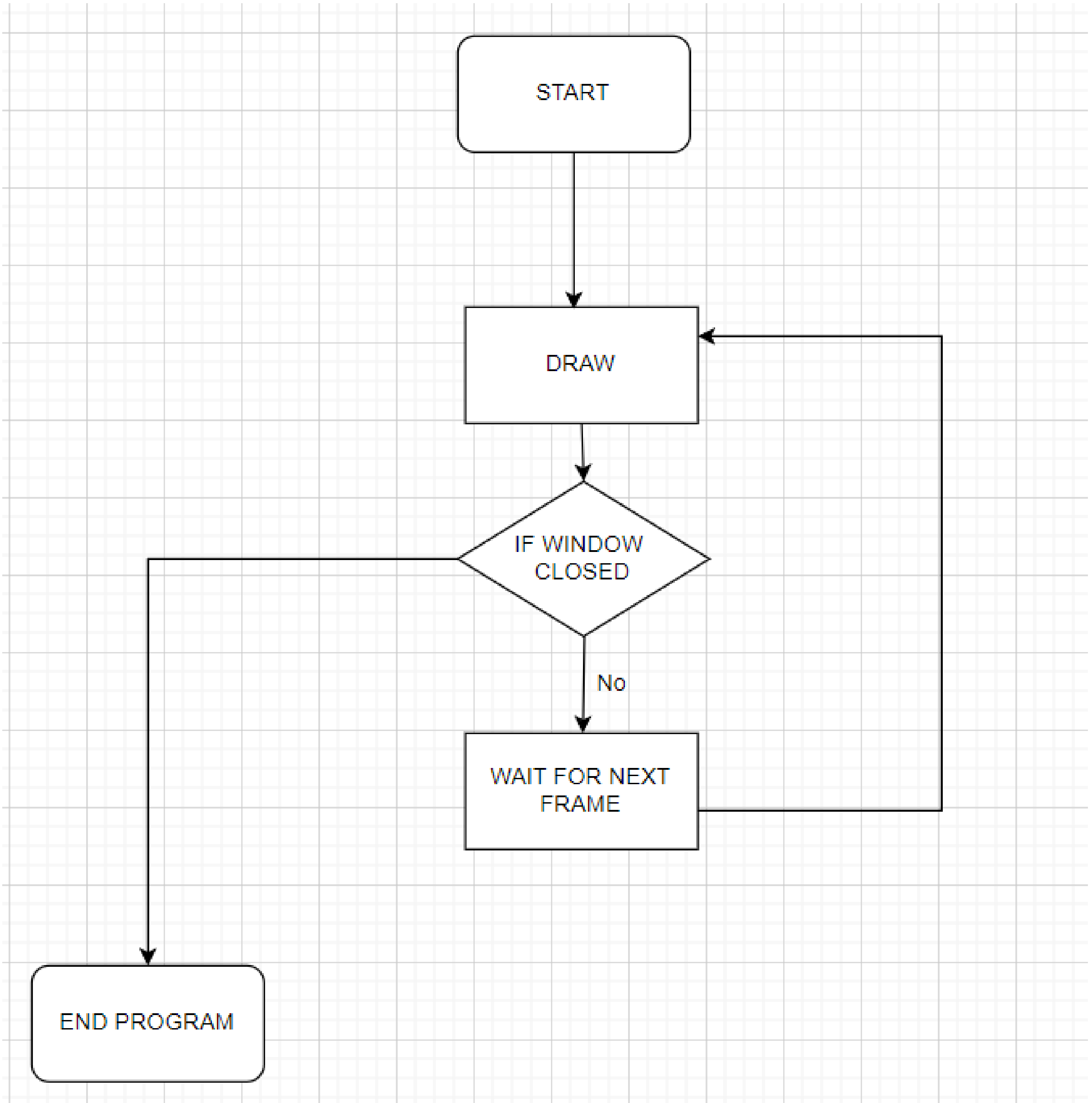
```
PS C:\Users\fr3nc\CollegeCodingProject\racers> cargo add macroquad
    Updating crates.io index
      Adding macroquad v0.4.13 to dependencies.
             Features:
             - audio
             - backtrace
             - glam-serde
             - log
             - log-rs
             - quad-snd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

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



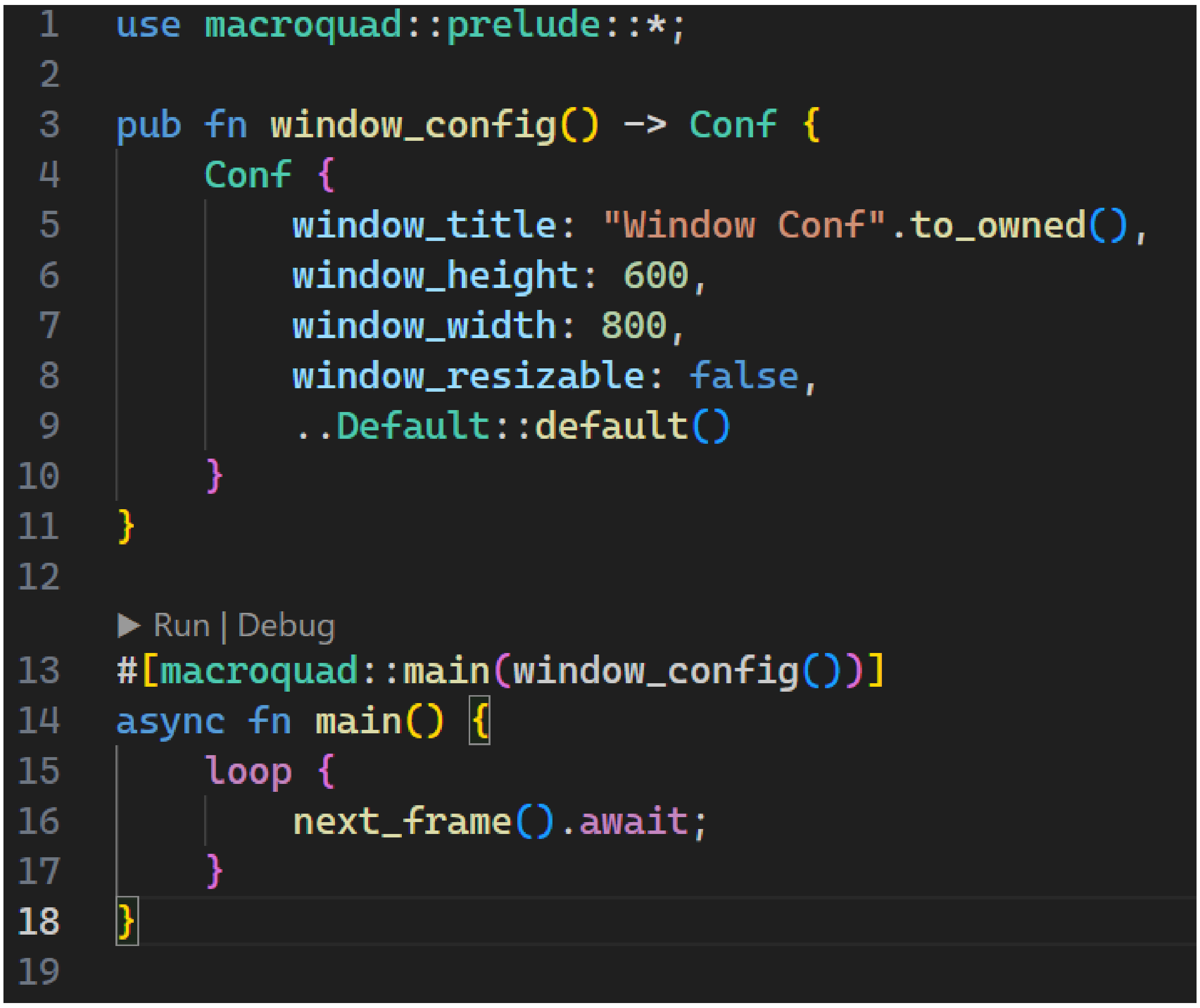
```
  Updating crates.io index
Downloaded byteorder v1.5.0
Downloaded color_quant v1.1.0
Downloaded quad-rand v0.2.2
Downloaded autocfg v1.3.0
Downloaded cfg-if v1.0.0
Downloaded bitflags v1.3.2
Downloaded adler2 v2.0.0
Downloaded adler v1.0.2
Downloaded simd-adler32 v0.3.7
Downloaded macroquad_macro 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.

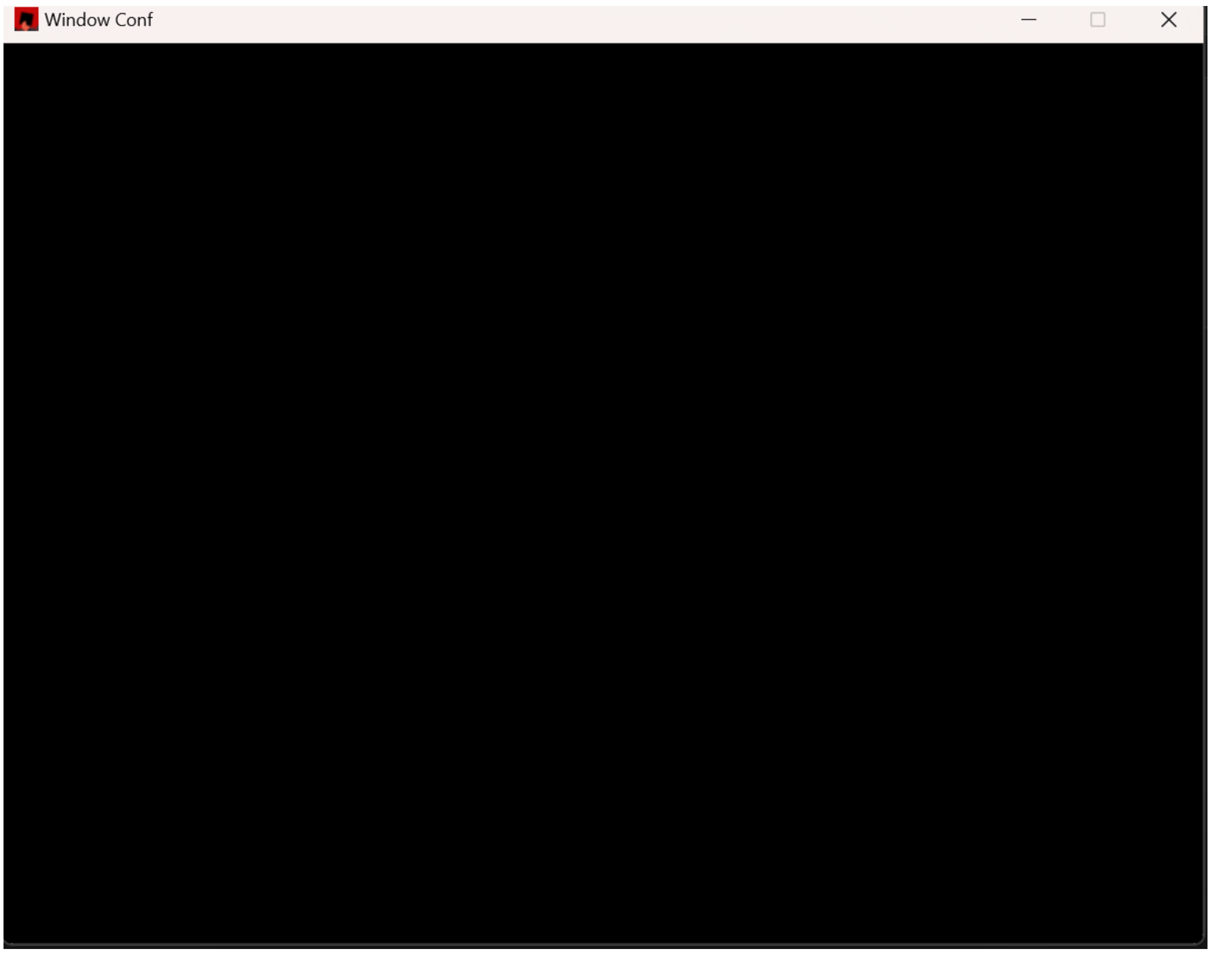


```rust
use macroquad::prelude::*;

pub fn window_config() -> Conf {
    Conf {
        window_title: "Window Conf".to_owned(),
        window_height: 600,
        window_width: 800,
        window_resizable: false,
        ..Default::default()
    }
}

▶ Run | Debug
#[macroquad::main(window_config())]
async fn main() {
    loop {
        next_frame().await;
    }
}
```

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. In the future I plan to move this function into a separate **config** file which separates this type of code into its own section of my project keeping the main file as clean as possible.

This is what the window looks like when the code is run:


Window Conf

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()

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.

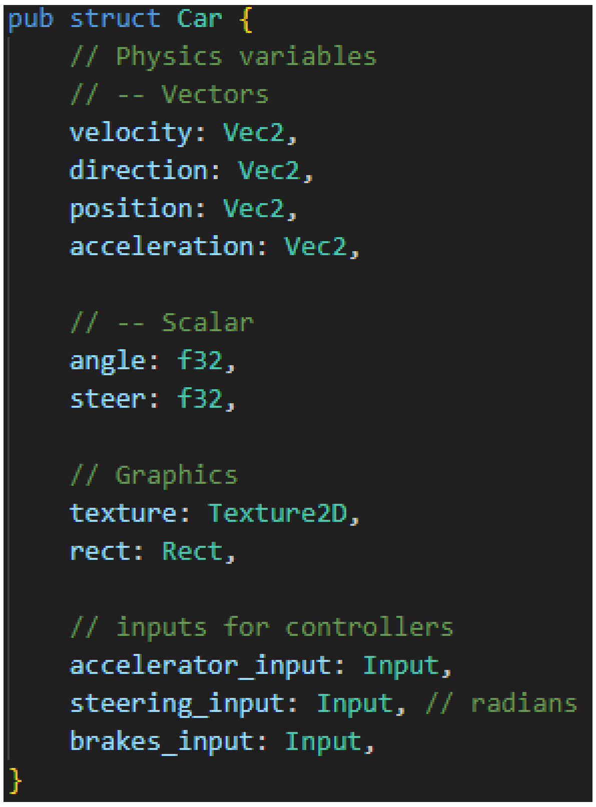
The **texture** attribute is the graphic which is drawn to the screen and is how we display the cars on screen. The **rect** keeps track of the car's hitbox so we can tell when it leaves track, hits something etc… It will also be useful in debugging in the future.

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.

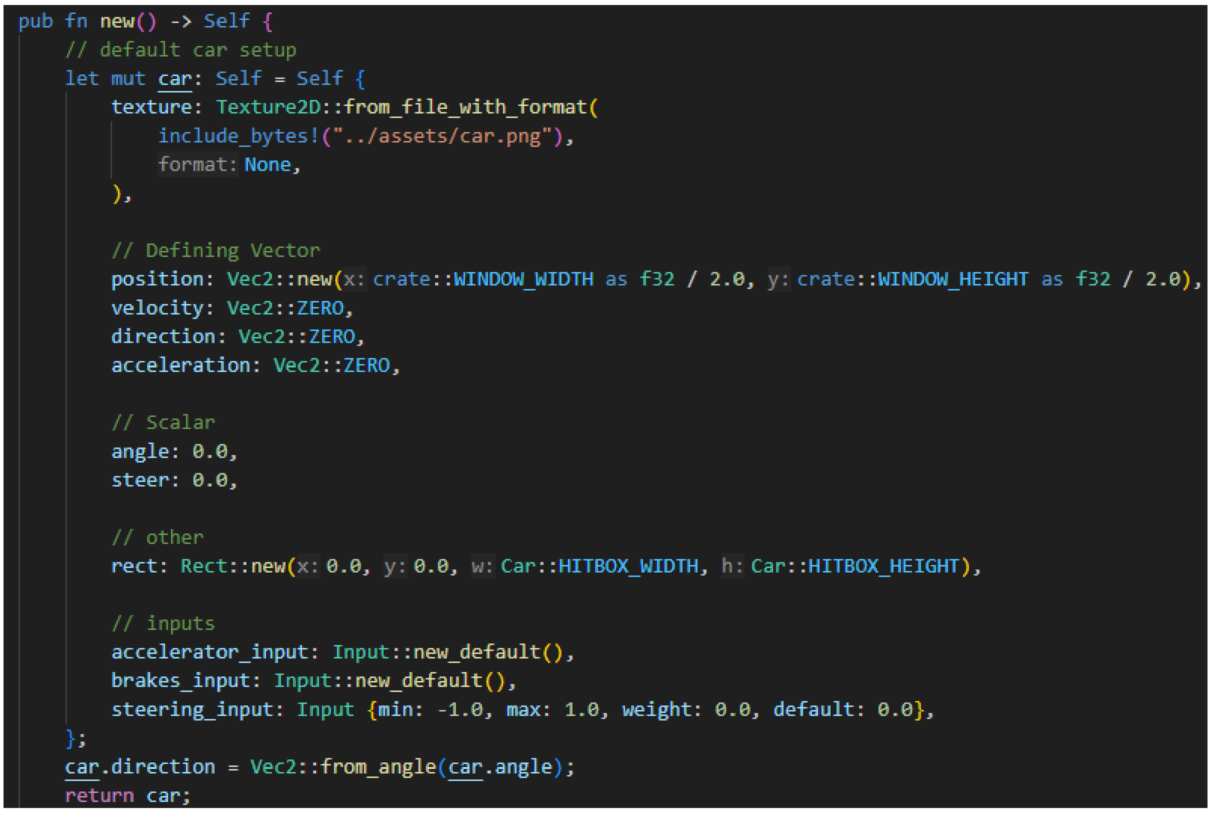
Here is the class definition for the car object in rust:



```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:

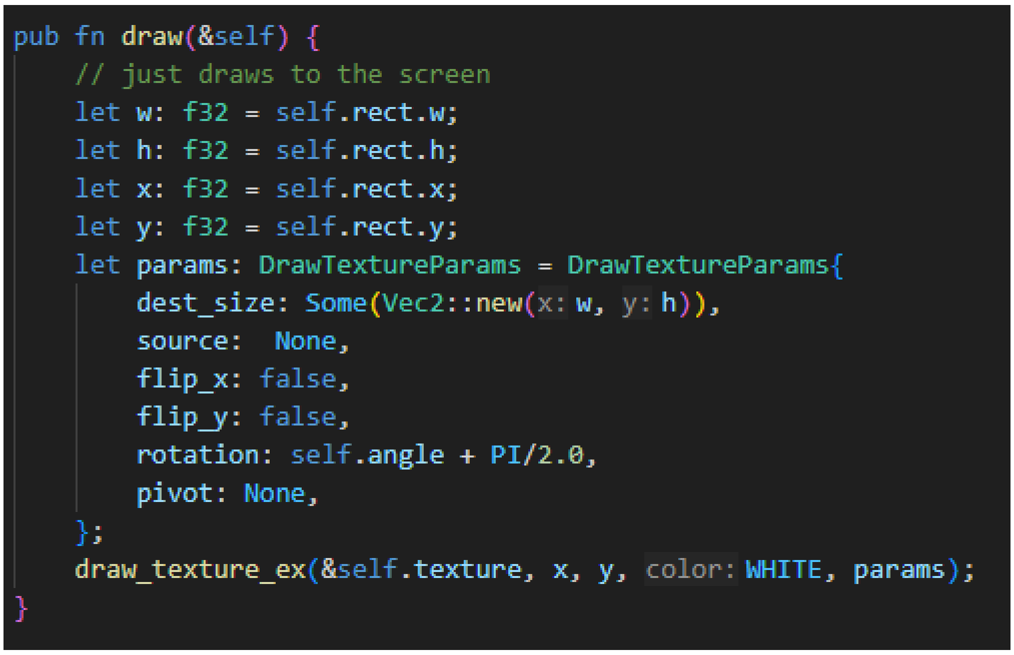


```rust
pub fn new() -> Self {
    // default car setup
    let mut car: Self = Self {
        texture: Texture2D::from_file_with_format(
            include_bytes!("../assets/car.png"),
            format: None,
        ),

        // 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:

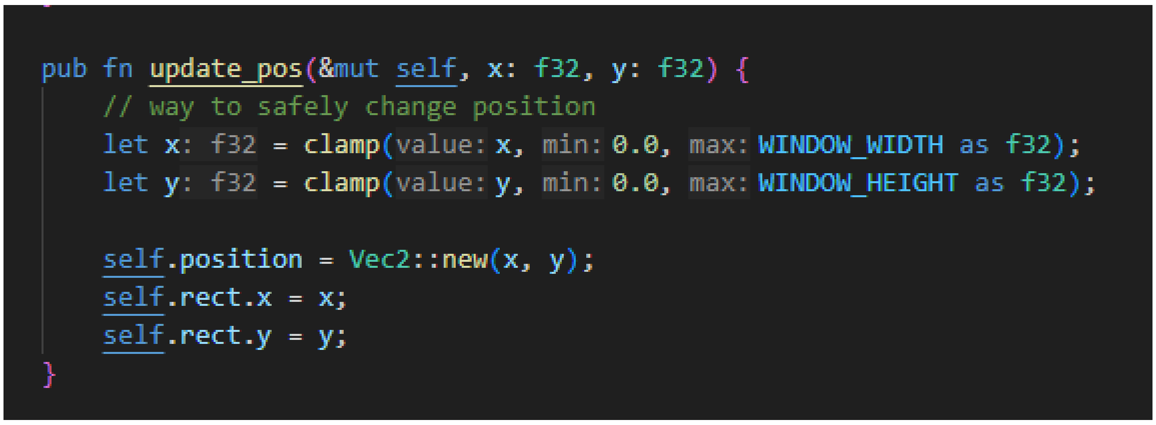
- I used the macroquad Texture2D structure's functionality to import an asset I found on google for a racing car
- For now, we place the car in the middle of the screen
- We define the rectangular hitbox as macroquad's predefined **rect** class which offers plenty of useful functionality
- The inputs 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



```
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.

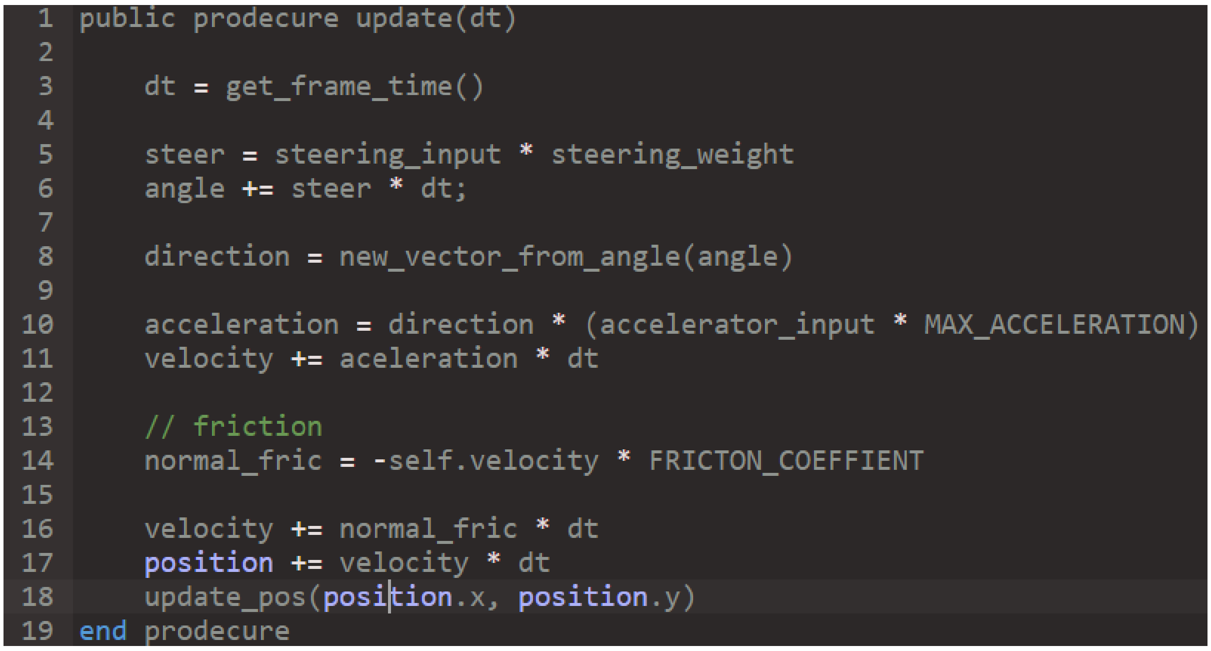
<u>**Update** Function</u>

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
```

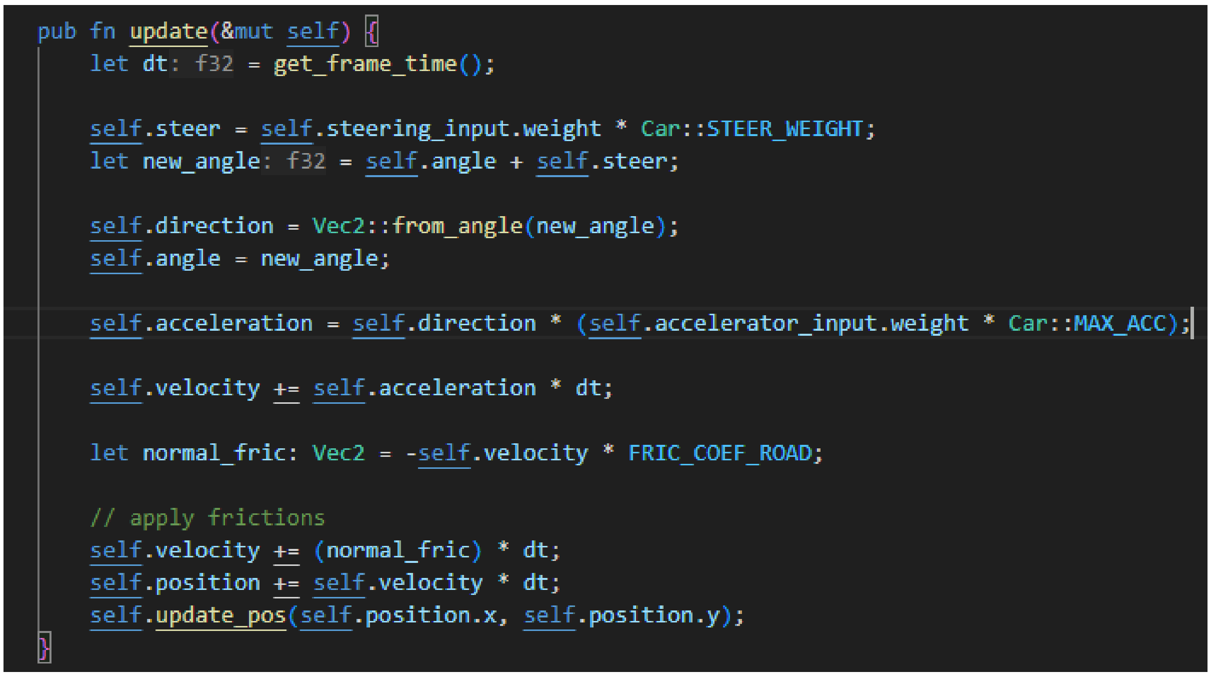
My update function algorithm pseudo code:



```
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:



```rust
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);
}
```

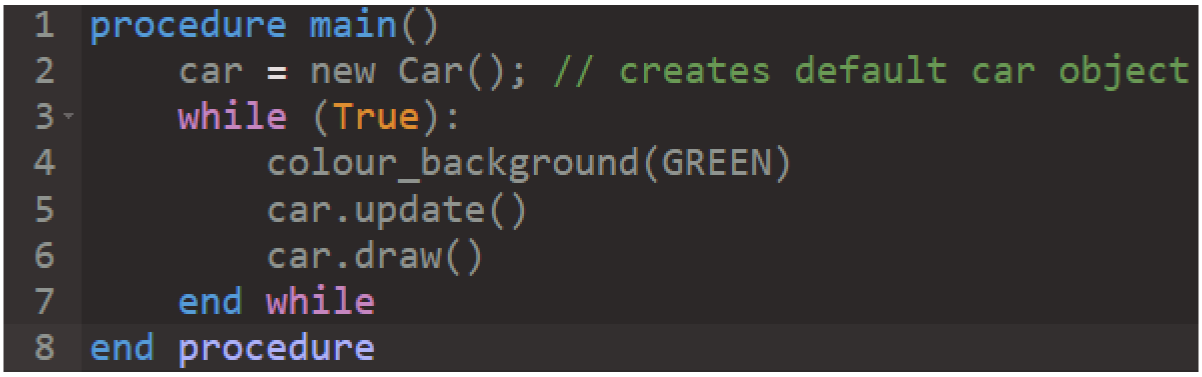
For now, this update code should be robust enough, however I fully anticipate modifying this function **a lot** since it contains the main bulk of the car's physical code.

Modifying the Main function

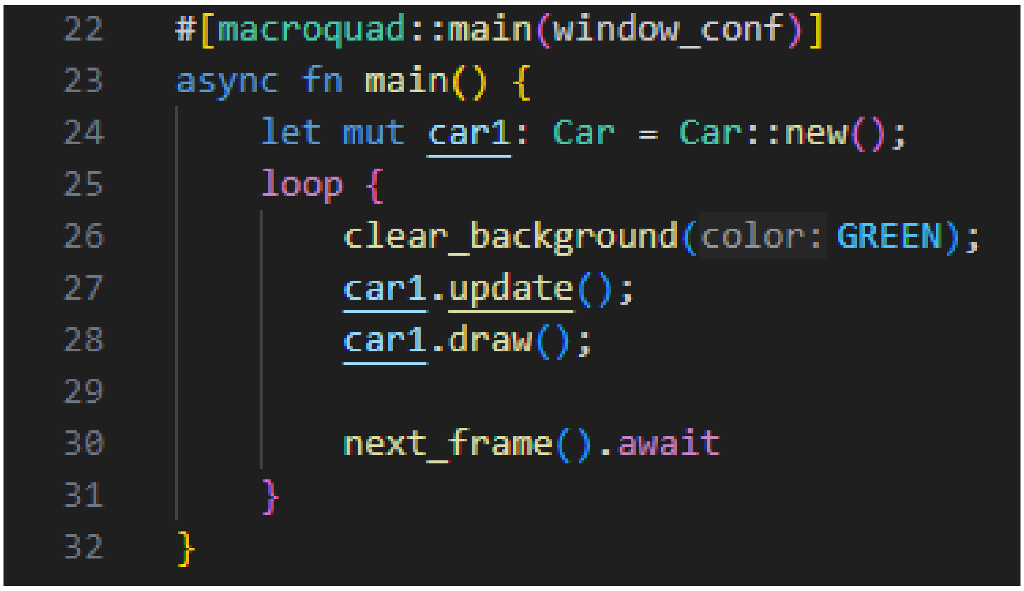
Now we have created the blueprint for a car object, lets create one and update and draw it from the main function.

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.



```
procedure main()
    car = new Car(); // creates default car object
    while (True):
        colour_background(GREEN)
        car.update()
        car.draw()
    end while
end procedure
```



```
#[macroquad::main(window_conf)]
async fn main() {
    let mut car1: Car = Car::new();
    loop {
        clear_background(color: GREEN);
        car1.update();
        car1.draw();

        next_frame().await
    }
}
```

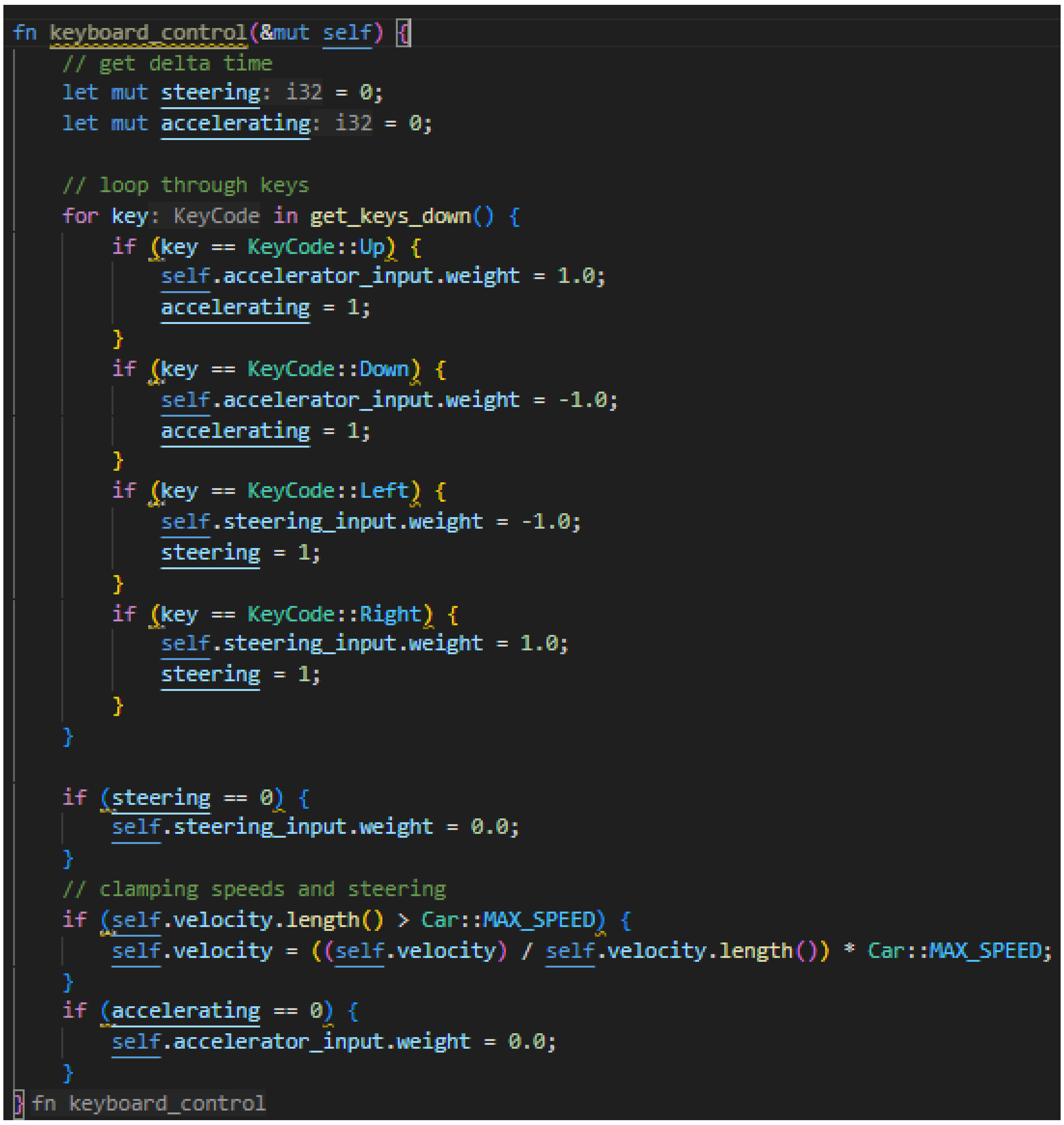
Note: The rust code is slightly different, but its just rust's weird implementation of an infinite loop, and the first scary line is just a bit of macroquad boiler plate (since macroquad requires asynchronous rust to be enabled to run).

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

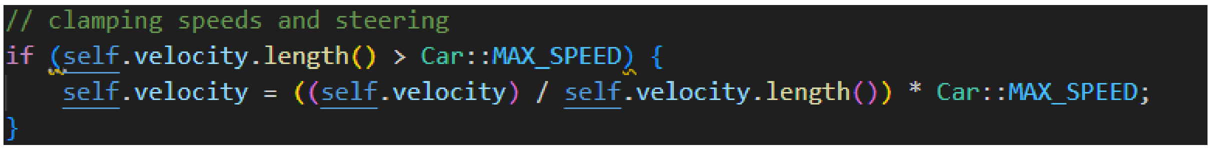
After staring at a stationary car on the screen for a while, I realised that I wanted to implement a way for a **human** user to interact and control the car. Although the car's will eventually be fully controlled by **AI,** I need to be able to test out the physics before moving onto that implementation,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



```
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**.

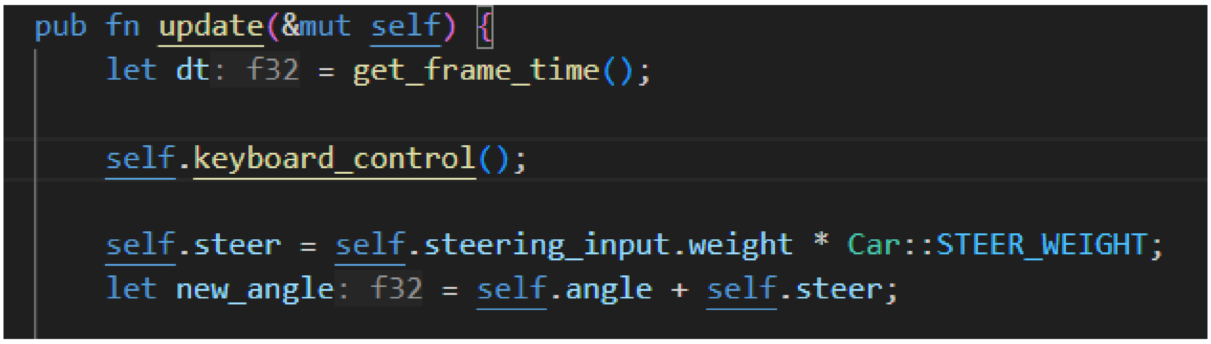


```rust
// clamping speeds and steering
if (self.velocity.length() > Car::MAX_SPEED) {
    self.velocity = ((self.velocity) / self.velocity.length()) * Car::MAX_SPEED;
}
```

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

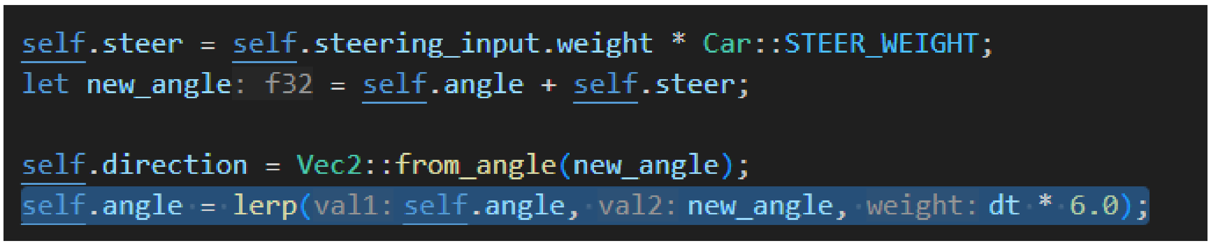


```rust
pub fn update(&mut self) {
    let dt: f32 = get_frame_time();

    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



```rust
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.

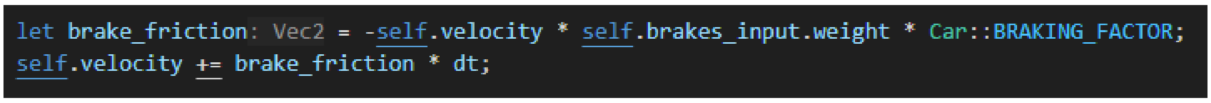
Formula

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