

Cardiff University School of Computer Science & Informatics

CM3203 - One Semester Individual Project

Final Project

“2D Racing Simulator with a focus on oversteer physics”

By: Bartlomiej Krol

Supervisor: Dr. Bailin Deng

Moderator: Natasha Edwards

Abstract

This paper showcases a process of developing a car simulator in the form of a simple game. The aim of the simulation is to provide an immersive experience with physics based on real physics. The driving simulation focuses on aspects important to drifting, or controlled oversteer, of a car. There are some gameplay elements to enrich the game also based on real racing systems. Along with the game, a basic game engine has been developed and the author talks about the upsides and downsides of creating an engine from scratch.

Table of contents

● Introduction	3
● Background	4
○ Drifting	4
○ Similar Games	4
○ Technology used	4
● Design and Specification	6
○ Entity Component System	6
○ Architecture and importance of clean code	7
○ UI Design	9
○ Identifying the features	11
○ Controls	13
○ Assets	14
○ Testing	15
● Implementation	16
○ Points system and drifting	16
○ Particles	17
○ Tuning	21
○ Engine and architecture	23
○ Physics	25
● Results and evaluation	34
○ Tests	34
○ Physics	39
○ Engine	40
○ Other aspects	41
○ Looking back at the timeline	42
● Reflection	43
● Conclusion	44
● Future work	45
● Appendix A	46
● Appendix B	51
● Appendix C	53
● References and Bibliography	54

Introduction

The aim of this project is to create a game that uses real car physics to program an immersive driving simulator with a focus on drifting. It will include simple gameplay mechanics to enhance the experience and make it a proper game instead of just a simulator. The car should have realistic behaviour which I will verify by modelling it and then comparing it after a real car.

The game will be made from ground up using a barebones game development library meaning that I will also have to create the game engine. This is done more as a learning experience as well as to allow for more freedom in implementation. The game engine should be modular to allow for easier implementation of features however the actual functionality of the game takes priority.

The initial scope of the game was to implement realistic driving physics, with regards to drifting, and ability for the player to tune their car and adjust it to their driving style as well as showcase the physics implementation. There are also some gameplay elements taken from more arcade games to keep the player engaged as the game only has a single player mode available. The graphical are just placeholders if I were to release the game because that is not the focus of this project. In conclusion, it is not the point to create a fully featured project but an enjoyable prototype due to the time constraints and the use of technology.

This project would interest anyone in the area of motion physics, game development or car physics. The audience for the game itself is for people who already enjoy racing games due to real physics and the high entry skill level. Prior experience with driving and/or simulator racing would be beneficial to understand driving mechanics.

Background

Drifting

Drifting is a motorsport where drivers are scored on how they control oversteer over a race course as opposed to a more traditional way of racing against a time clock or other drivers. (Dennis Jarvis, 1995) More intricate details will be mentioned in a further discussion when I will be analyzing the details to find features to implement.

Drifting originated in Japan and has been steadily growing since its inception in 70s and 80s (Lear, 2021). Since then there have been multiple racing games focused on one specific motorsport such as rallying, NASCAR or Formula 1. However up until very recently there have been no games focused specifically on drifting. Meanwhile many racing games included drifting as one of the race modes, they usually have arcadey driving physics.

Similar games

Simulation racing games go way back to the arcade machines. Although at that time the physics were revolutionary and very immersive, there are casual racing games with physics more realistic than those games. There is a good selection of simulator racing games on the market like iRacing, Asseto Corsa or more specialized ones like F1 series or Dirt Rally series. However with the recent surge in popularity of drifting, there has been only one game released with drifting in focus called CarX. There is also DRIFT21 in development, it is available in early access however the release date has been moved a few times so far.

Although drifting can work really well in general simulator games, games with focus on drifting usually allow for deeper modification of cars in relation to drifting, this often includes mechanical and cosmetic modifications, as well as user interface suited towards that gameplay. I will be talking about UI later in the report.

Technology used

I wanted to use a simple game framework instead of a fully fledged game engine such as Unity or Godot. The reason for that is that it allows for more freedom, the learning curve for small projects is much easier and it is a better learning experience as less stuff is hidden behind the scenes. Initially I wanted to create a 3D game as it would allow me for better representation of the car and the physics. This would probably be possible in Unity or such however due to my decision to go with a framework, it was way out of scope as I would have to use Direct wrapper of some sort.

Albeit simpler to program and create/implement assets and animations, this poses problems of indicating car's behaviour as well as translating 3D physics onto a 2D plane. That means I will have to make sure that I convey information about cars behaviour to the player appropriately.

There are plenty of options for simple 2D game frameworks for almost every language out there. I have prior experience with Java, Python and C++. Personally, I do not like Java so that leaves me with a choice of Python or C++. I worked with SDL2 in C++ and there are similar frameworks for C++ like SFML or GLFW. However, personally, coding in C++ is quite

slow compared to Python, most likely due to the lack of experience and it being lower level than Python. The huge advantage of C++ is performance however I do not think I need a lot of performance for such a small game therefore I decided to go with Python.

There are few options in Python as well. I went with PyGame 2 as it is based on SDL2 and I know the basics of SDL2 already. There is Pyglet and its advantage over PyGame was performance however it is not the case anymore due to introduction of PyGame 2. There is also Python Arcade Library which is an extension of Pyglet and has some differences however it is closer to an engine and therefore more constrained.

PyGame 2 is a SDL2 wrapper and SDL2 is a development library. It manages graphics, audio, inputs and all other basic necessities. It can handle 3D graphics however that requires programming Direct3D or OpenGL APIs. It is also multi platform which is advantageous as I will be writing the game on a Linux laptop.

On top of that I used some libraries and programs for processes that are not the focus on the project

- NumPy and SciPy - a very common Python maths and science libraries that extends and speeds up functionally over the built-in Python maths library
- Esper - is a Entity Component System (ECS) library. I also found Snecs which claims to be faster but went with Esper because it is more popular and therefore has more resources online. I will talk more about ECS further down in the report.
- Tiled - a GUI program for drawing tilemaps out of tilesets. It makes creating complex tilemaps much quicker compared to a simple method of writing them out in a text file and displaying each tile corresponding to a certain character in the text file.
- PyTMX - library for loading .tmx format tilemaps created in Tiled

On top of that due to the nature of the game I wanted to use a gamepad for the controls as it allows for analogue inputs for steering, throttle and brake, just like a real car. Fortunately PyGame includes gamepad support. If I owned a steering wheel I could try to use that as a target input for my game. Additionally keyboard controls will be available so that everyone can play it.

For tracking progress I used Kanban board called Trello. It allows me to move tasks between categories such as "To do", "In progress", "Testing" and "Blocked". Trello allows to break down big tasks as a list which is helpful for big modules such as the ones responsible for physics. I updated the progress once a week and added any new tasks I thought of instantly so I do not forget.

I will also be using GitHub for version control in case I need to revert my project to an older state or branch out. It will also give an opportunity for others to look up my project.

Design and Specification

Entity component system (ECS)

Entity Component System is a common pattern for game architecture. I used it once before and I really enjoyed using it because of the modularity and ease of maintainability. I wanted to write my own ECS library but considering I have never done it before it would probably take me a few days that I would rather focus on actual game functionality.

Entity Component System can be broken down into its basic components. Entities are just IDs assigned to each object. Each entity has components which contain no methods, just pure data. Any entity can have any component. Then all operations on the data within components are processed within systems. Systems usually iterate over each entity that contains specific components and does operation on the data within it. (Adam, 2007)

It favors composition over inheritance. Any entity can have any component assigned to it. When it comes to systems we can have one render system and iterate over each entity that has a sprite and render it. (Adam, 2007)

ECS is also part of data-oriented programming as opposed to a popular object oriented programming. One of the advantages of data-oriented programming is smart use of CPU cache. If we look at it through the lens of ECS, ECS runs a system for each entity that contains a specific component. When we load a component, we load the whole array of components into cache therefore when system runs over few entities using the same component it does not need to read the data from memory again. This is used to overcome bottleneck that memory bandwidth is and to combat cache misses. Another advantage is shying away from abstraction which can overcomplicate the code and therefore improve maintainability. Abstraction is not inherently bad but it can become too complex in large projects. (Adam, 2007)

Architecture and importance of clean code

ECS is quite flexible in a lot of areas as it does not force the overall structure all that much. It can be intertwined with OOP or functional programming. It also means there are many ways to do a specific thing and I will have to find architectural solutions appropriate for the scope of this project.

I will aim to make classes and functions modular however that takes extra time and I will probably have some dependencies and hardcoded variables. Good example of this is the link between the user interface and the car. In my game there will only be one car and therefore I can directly link it to display appropriate values on screen, this however would not be possible with multiple cars all at once.

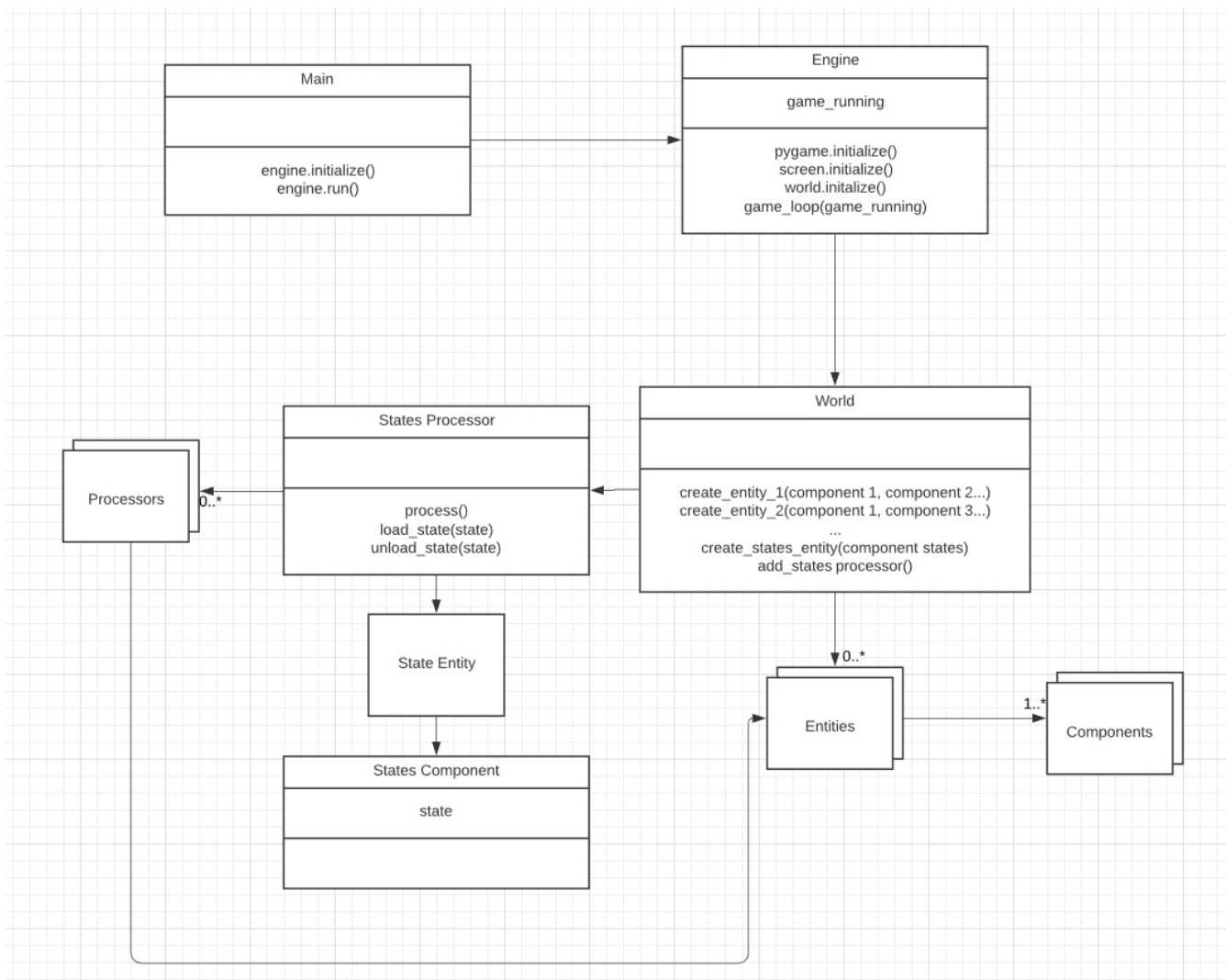


Figure 1

This is only a conceptual UML class diagram so it outlines just the general idea and structure of my code. The final implementation might be different depending on what obstacles I might encounter.

Engine is initialized and run in Main. The engine then initializes PyGame and ECS world as well as runs the game loop. When initializing the world, all of the entities are created as well

as state entity and a state processor. State entity holds the state variable that is evaluated in state processor that loads appropriate state and unloads unused states. These functions either add or remove processors appropriate for the current state.

I will be following PEP 8 style guide for Python. I chose PEP 8 as it is very popular, possibly the most commonly used style guide for Python. Main advantage of using a style guide is making sure that my code is easier to read and therefore easier to understand. This is especially important as my code will be looked at by a supervisor and moderator. I also upload my project to GitHub and there might people interested in looking up my code for their own games and such and I would want them to be able to comprehend it. On top of that, it benefits me as going back to a project after a short break makes you lose the “flow” and I expect to take a 2 week break over Easter so that will make it easier for me to get back into work.

UI Design



Figure 2

If we look at CarX (CarX Technologies, LLC, 2017), we can see (Figure 2) that next to the speedometer there are bars. The vertical bars represent: clutch, break, throttle and. The horizontal bar represent steering input and there is also has an angle indicator at the bottom. On top there is a point counter that displayed the currently accrued points, a multiplier and a extra points for objectives.

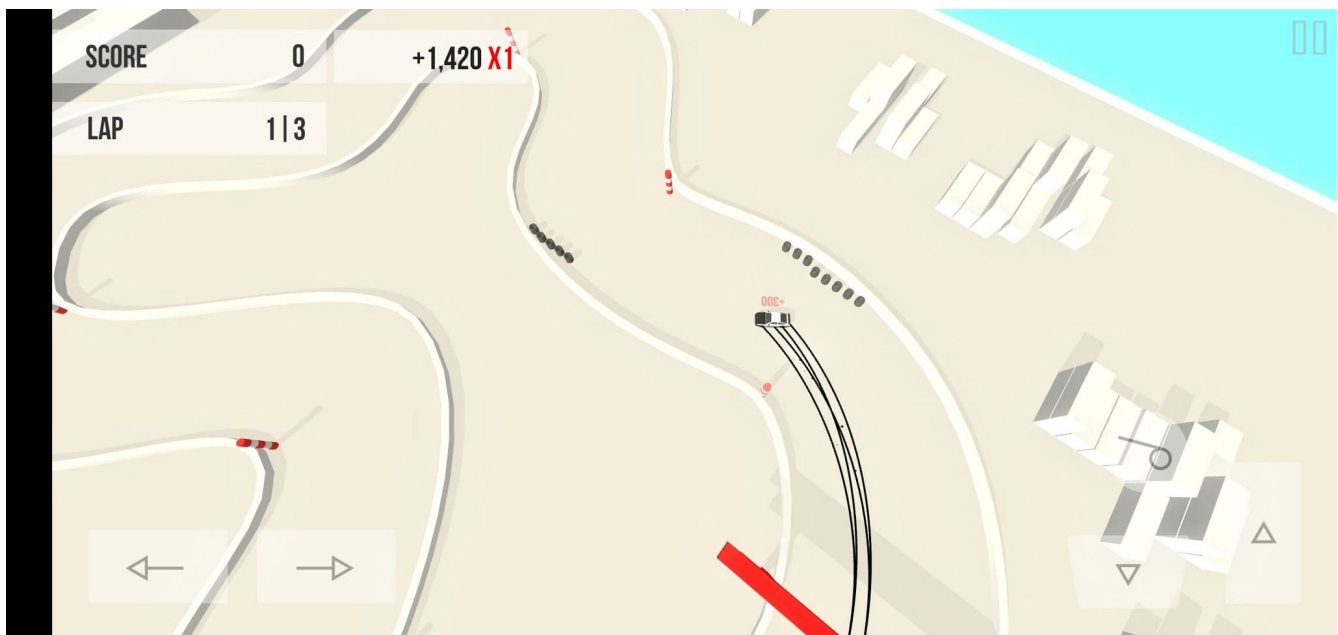


Figure 3

Another example is Absolute Drift (Funselektor Labs Inc., 2015) (Figure 3). This is a 2D game with an arcade gameplay therefore it skips the speed indicator and such. However I really like the layout for the points as it is in the corner so it does not obstruct the immediate view. I also like the idea of the extra points of the objectives to pop up near the car or the objective itself.

Since we are going for simulation game, I will be trying to achieve minimalistic UI during gameplay to increase immersion (Figure 4). Most importantly the car's essential information such as RPM, speed and current gear is show in bottom left corner. I modeled the dash after Honda S2000. Other than that there is only point counter in top left corner. It shows the total points, bonus points for a current extra such as perfect line or collecting a token and a combo multiplier. It needs to be there so that the player knows what he is rewarded for.

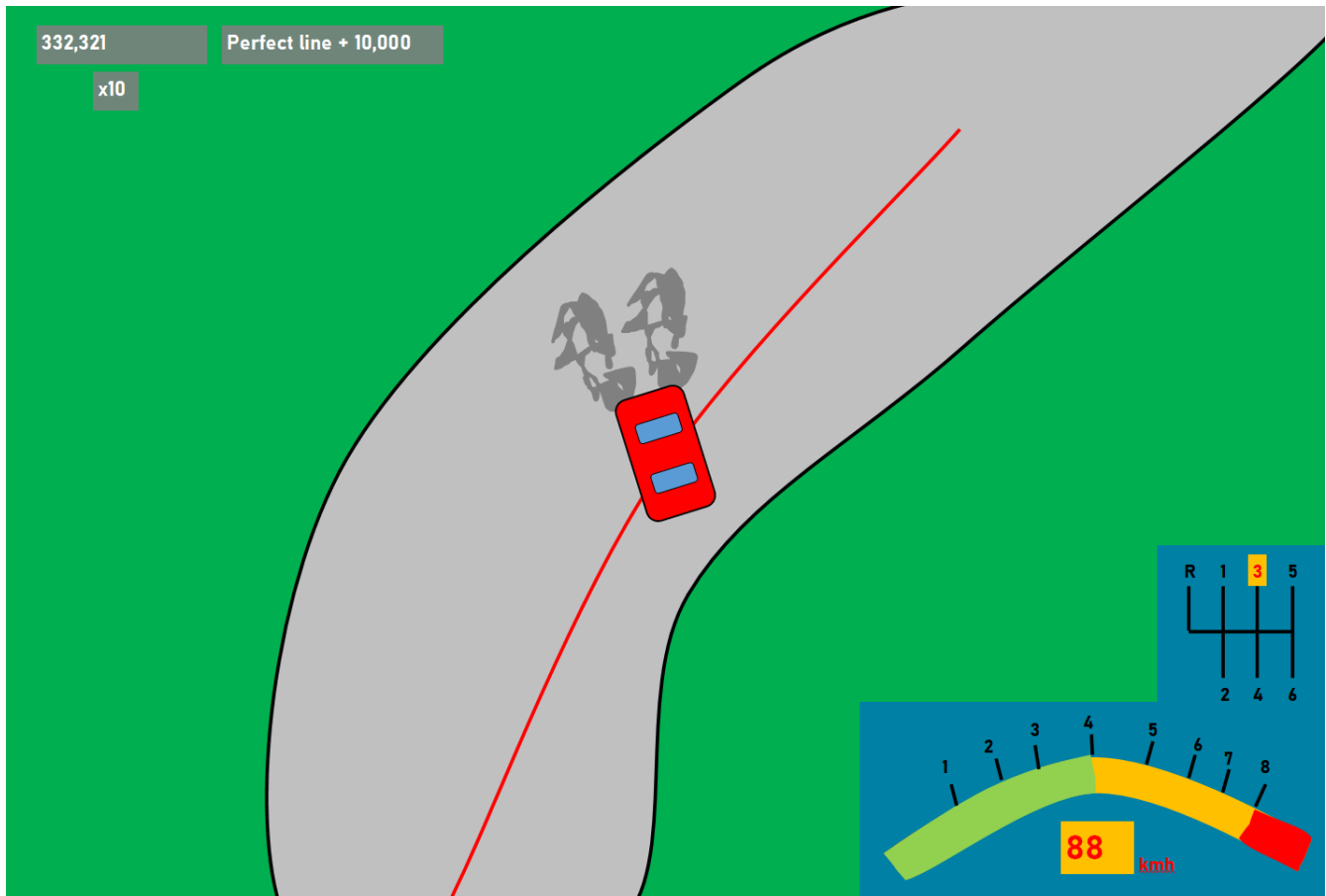


Figure 4

There is also going to be a main menu and tuning menu with some sliders but I do not think that they need an explanation due to how generic they are going to be.

Identifying the features

Car physics

I do not plan to recreate a perfectly realistic simulator as that is too big of a task for a one person on such timescale. Therefore I have to pick out the necessary components that would add value to my game. I will provide a brief explanation now, with a more in depth one in the implementation part of the report.

Let's start with the ways to initiate a drift. (GTChannel et al., 2010) (Tsuchiya, n.d.) Breaking traction with power by opening the throttle on a high powered car (requires high power), clutch kicking which is depressing the clutch while on throttle and then releasing the clutch to upset the rear wheels (manual gearbox with a clutch), changing gears without rev matching, similar to clutch kick, upsets the drivetrain causing loss of traction (manual gearbox again), lift off or braking which causes weight transfer to the front and therefore loss of traction in the rear (medial weight transfer), handbrake to break traction of rear wheels is common for high power cars (handbrake) and scandinavian flick or feint where a car turns opposing direction to the turn and quickly snaps back to initiate oversteer (lateral weight transfer).

Then we can break down car into groups of components. One way of doing it would be: engine, drivetrain, aerodynamics and suspension.

Engine, or rather the power, changes the way a person drifts. (Tsuchiya, n.d.) (GTChannel et al., 2010) Low horsepower cars have to use more clutch kicks to initiate or even stay in a drift as it does not slow the car down as opposed to bigger focus on a handbrake for high horsepower cars. However it does not affect the ability to drift in general so much. It would however be a priority for a drag racing game. Programming an engine simulation with all its ancillaries such as however could be a whole another project.

Then we have a drivetrain, which is everything between the engine and the wheels. There are not many components, mainly gearbox and differential. For drifting manual gearbox is basically essential. There is a tiny minority of people who drift cars with automatic gearbox, however it limits the control over the car as well as takes away the ability to use the clutch which is important for cars with low power. Different gearboxes have different gear ratios and although not a very common modification, it should be easy to program. The cars in drifting are all rear wheel drive therefore they will all have just one rear differential. Differential will be just a simple torque multiplier and we will be using a welded/fully locked differential by default. This means that both wheels spin at the same rate. Limited slip differentials have their place in drifting although they are uncommon and open differentials are not used in motorsports at all.

Drifting is not a high speed motorsport and aerodynamics do not play a huge role therefore I will be skipping simulating advanced aerodynamics. Only the simple stuff to limit a car's top speed. Aerodynamics would be important for endurance racing, time attack or F1 as cars in these types of races achieve high speeds and, in case of endurance racing, fuel consumption also matters.

Finally we have suspension which plays a big role in a car's handling and therefore drifting. (Dennis Jarvis, 1995) There are multitude of modifications that can be done to a car and I will plan on identifying the most important ones. Shocks and spring rates are probably the most common modification. They obviously alter spring rate and damping rate as well as ride

height which alters the centre of gravity. Sway bars alter lateral weight transfer by increasing the total spring rate to the outer wheel. Extended knuckles/suspension arms which increase the steering angle. Alignment which allows for altering of the wheels camber, caster and toe. Tyres and wheels sizes and tyre pressures which alter the contact patch. These are all advanced features and I plan on implementing these only if I have time.

In summary I will need to implement:

- At least two different torque/power curves
- Manual car gearbox with clutch
- Simple locked differential
- Simple aerodynamics, mainly limitation of top speed by drag.
- Variable steering angle
- Tyre pressure and size
- Variable brake force
- Wheel alignment
- Spring rate and damping rate
- Ride height

Gameplay

In real life judges score a drift run based on specific criteria. It is obviously slightly subjective but I will try my best to translate into a computer game.

Normally each judge scores a run from 0-100 however I feel like that is not the best way to implement it in game. It would be hard to implement a scoring system that changes up and down over the course of the run and I want to give the player some immediate feedback. This means an alternative that most video games use, which is increasing the point counter as you drift. (The British Drift Championship, n.d.)

Race tracks have a race line that the car is meant to follow. However instead of implementing the whole race line, it can be summarized into clipping points and clipping zones. Clipping points are places where either the front or the back of the car has to be at, at a specific point on track and clipping zones are whole parts of the track where, usually a rear of the car, is meant to be during a drift. This allows for more freedom between the clipping points/zones. The clipping points will be represented by little tokens to collect where the clipping point should be and clipping zones will highlight parts of the race track where a car gets bonus points. (The British Drift Championship, n.d.)

Another crucial aspects are angle and speed and they are pretty self explanatory. The higher the speed and the higher the angle, the more points the driver gets. Trap speed is used in some competitions to objectively measure the speed at one or more points during the run. However a game allows me to track the speed all throughout the run. I will be multiplying a specific "drift variable" by speed and angle and their respective constants. (The British Drift Championship, n.d.)

Style is the hardest part to implement. There are things such as transitions, initiation, steering input, the flashiness of the run, losing the drift or dropping of the track. This is tricky to implement and I will see what I can get away with during implementation but right now I have the idea of multiplying the final score based on the style during the run. (The British Drift Championship, n.d.)

Controls

I designed the game with a controller in mind as it allows for analogous inputs for steering, throttle and brake just like a real car would. This gives so much more control that is necessary when drifting.

The controls are as follows:

Diagram numbers	Actual name	Keyboard	Function
3	Left Thumbstick	A/D	Steering
6	Left Trigger	S	Brake
9	Right Trigger	W	Throttle
5	Left Bumper	M	Gear down
10	Right Bumper	K	Gear up
A	A	P	Clutch
B	B	Space	Handbrake
8	Start	Esc	Back to menu

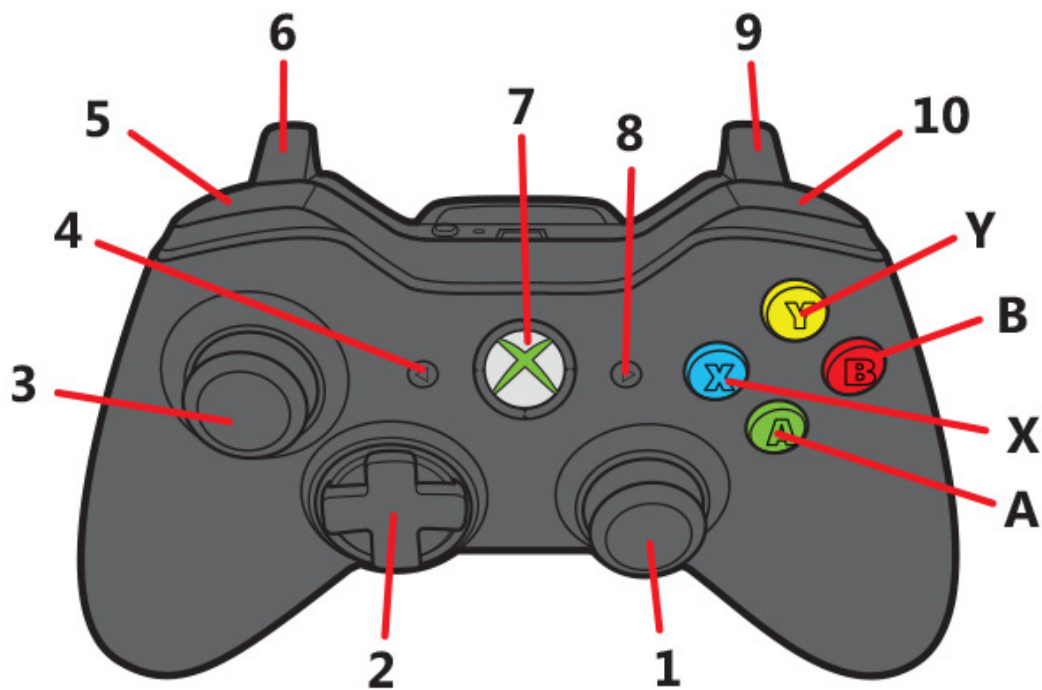


Figure 5

Assets

I have used Racing Pack by Kenney (Kenney, 2019) for my graphical assets as it was the most complete graphical asset pack I could find. They have a variety of different road angles and turns. They also have car sprites. On top of that they provide additional decorations I could use to either decorate the track or use as markers etc. I wanted to use one pack as it creates a cohesive experience. It comes under CC0 1.0 Universal license meaning that it can be used for any purpose without limitations. Due to the complexity I had to use Tiled to put the track together as it would take way too long otherwise.

The particle system does not use any assets but just draws rectangles from built-in PyGame function.

Testing

I will be relying mostly on play testing, where I will be manually playing the game and testing for expected behaviour. The advantage over all other types of testing is that this allows to get a feel of the game and analyze the fun factor as well as test for bugs.

Due to the scope of the game I will not be using unit or automated testing. There are not that many systems that I can unit test. On the other hand there is car tuning which would change lots of variables and unit testing would be beneficial to cut down on manually testing all of these configurations. This would be certainly beneficial if the development time of the game was longer than 12 weeks however right now I feel like the time put into developing the tests would take longer than it is worth. By automated testing I mean setting up input macros to carry out the same inputs each time and either comparing some variables such as position and points with expected values or comparing screenshots of the test and expected screenshot. Automated testing is not gonna be useful with how quick I incorporate changes to the game either. I will be going through many iterations of inputs, variables and outputs that the automated tests would be very quickly outdated.

Most of the test coverage will aim at the physics as that is the essential part of the program and there will be some for UI or menus.

The car will be modelled after Toyota GT86/Subaru BRZ. This allows me to check some aspects of the simulation against the real performance numbers of the car. This will include things like acceleration times, top speed and g forces.

I will also let the game run for a few hours and check for memory leaks. It is unlikely to happen with Python due to its garbage collector but still worth a try.

All of the tests can be seen in Appendix A.

Implementation

Points system and drifting

I tried to resemble a real scoring system for competitive drifting in my game. (The British Drift Championship, n.d.) I took speed and car's angle into consideration for the main calculation. The drift starts when the slip angle of the rear tires is higher than 0.5. At first I thought of using side slip angle, and it worked quite well too, however I found slip angle of rear tires to be more realistic in implementation as it is only the rear tires losing traction that cause oversteer. The equation is:

$$(|\text{SlipAngleRear}| - 0.4) \cdot \text{Velocity} \cdot 10$$

I take away 0.4 from the slip angle of rear tires to create a bigger difference between low angle and high angle. Then I multiply it by velocity and a constant of 10 so that the player gets thousands of points instead of hundreds.

Before the points for a single drift are appended to the total drift point counter, they are multiplied by a multiplier. That multiplier is incremented by 1 for each clipping point (cone), user has hit during the drift. The multiplier is reset back to 0 after each drift. The code does not have any other collidable objects therefore there is no check if it is the right type of object. This could be however quickly fixed by having an if loop that asks 'if object == "clipping point"' and each entity could have an object variable with its own type.

```
def process(self):
    car_sar = self.world.component_for_entity(self.car, com.Steering).sar
    vel = self.world.component_for_entity(self.car, com.Velocity).velV.magnitude()

    for ent, (pnt) in self.world.get_component(com.SinglePoints):
        # When the car drifts
        if abs(car_sar) > 0.5:
            pnt.points += int((abs(car_sar) - 0.4) * vel * 5)
        # When the drift stops
        else:
            if pnt.points != 0:
                for ent, (tpnt) in self.world.get_component(com.TotalPoints):
                    tpnt.points += pnt.points * pnt.multiplier
                pnt.points = 0
                pnt.multiplier = 1
```

Figure 6

```
for ent, (rect) in self.world.get_component(com.Rect):
    if ent != 1:
        if car_rect.colliderect(rect.rect):
            pnt.multiplier += 1
            self.world.delete_entity(ent)
```

Figure 7

Particles

The game needed to indicate to the player when drift occurs in a clear manner. I felt that sound would be not enough, especially considering that sound is optional in a way that it is not needed to actually play a game. This made me turn to smoke coming from the tires and I thought that particle system would look more realistic than a prerendered asset.

The implementation of particles is split between two systems. One is responsible for adding the particles to a list (Figure 9) and the other one for displaying and removing them (Figure 13).

Particles are added to the list only when the slip angle ratio of rear wheels is above 0.5, which I decided to when the drift starts.

Each particle has 4 variables, position in the form of x and y coordinates, randomly selected color integer, transparency which is set at 200 and a car's angle/orientation at the time the particle is added to the list. There is only one color integer as I use it for each of RGB channels to create a grey color however I want to a little variation in greyness therefore I generate it in the range of 110 to 150 with Python's built-in function.

Starting off with position. `sprite.get_rect().center[0 or 1]` allows me to get the distance between the corner of the car sprite and the centre on x and y axis. I use these values to calculate the distance between the corner and the centre by using the Pythagorean theorem. However I decrease the radius by 15 pixels because I want the particles to originate roughly where the wheel is and not on the very corner of the car.

Then I need to calculate cartesian coordinates from polar coordinates (Newton, 1736) which I get from the radius I just calculated. For the angle I use car's angle, offset it by π so that it is in the opposite direction of the where the car is heading and also offset it by another variable, in my case 0.4 rad or -0.4 rad depending the left or right wheel, so that the particles originate roughly where the wheels are.

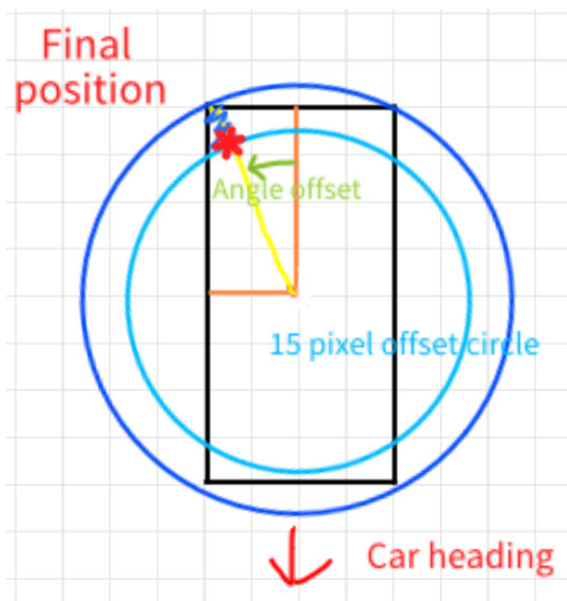


Figure 8

These values are just in relation to the car so for them to display correctly I need to offset them by car's and camera position so they display in the middle of the screen.

```
def process(self):
    car_pos = self.world.component_for_entity(self.car, com.Position).posV*constants.SCALE
    sprite = self.world.component_for_entity(self.car, com.Sprite).sprite
    car_ang = self.world.component_for_entity(self.car, com.Steering).heading
    cam_pos = self.world.component_for_entity(self.tilemap, com.Camera).posV
    car_sar = self.world.component_for_entity(self.car, com.Steering).sar

    if abs(car_sar) > 0.5:
        for ent, (par) in self.world.get_component(com.Particles):
            radius = math.sqrt(sprite.get_rect().center[0]**2 + sprite.get_rect().center[1]**2)-15

            x = radius*math.sin(math.pi+par.angle_offset+car_ang)
            y = radius*math.cos(math.pi+par.angle_offset+car_ang)

            x_src = int(x+car_pos.x-cam_pos.x+30)
            y_src = int(y+car_pos.y-cam_pos.y+55)

            for i in range(5):
                r = random.randrange(110,150)
                par.particles.append([x_src, y_src], r, 200, car_ang)
    else:
        pass
```

Figure 9

Now I want to move the particle every time I display it so that as the car drifts, a huge cloud of smoke forms behind the car.

I programmed the particles so that the next position is randomly selected by moving the particle between 3 and -1 pixels away from the car and 4 and -4 pixels sideways in a relation to the car based on the car's angle when the particle was created.

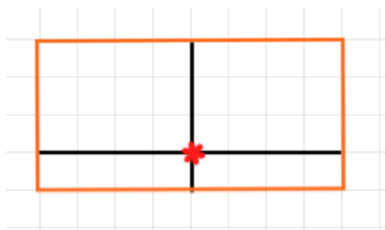


Figure 10

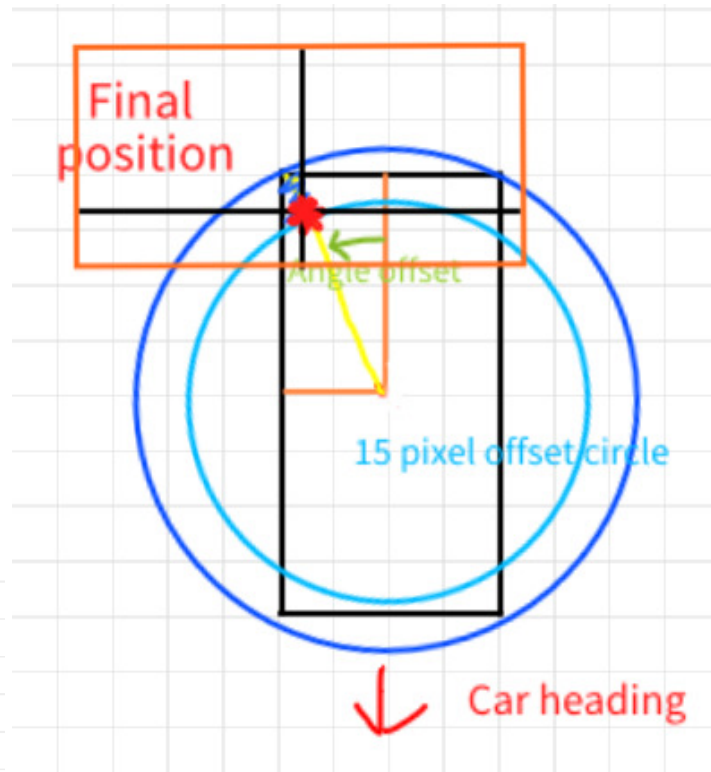


Figure 11

When the car is facing down, that means the particle moves between 3 and -1 pixels in y direction and 4 and -4 in x direction (Figure 10 & 11) but when the car is facing right, the particle would move between 4 and -4 in y direction and 3 and -1 in y direction. Therefore as the car spins around, I need to calculate new ranges of values.

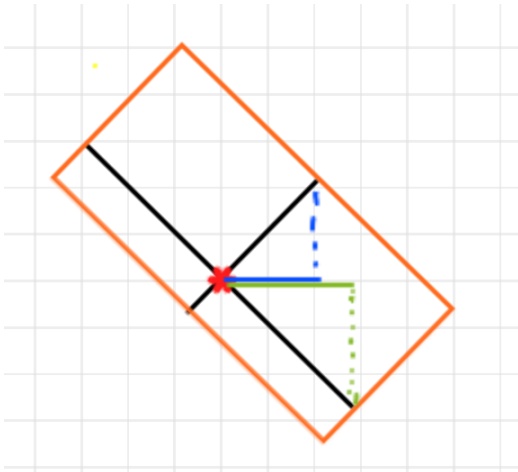


Figure 12

avdict holds the angle and a corresponding range in each particular direction based on polar coordinates for when the car is facing down. Then I run a for loop for each of the values and keys in the dictionary to find component vectors (Figure 12) of each of the initial vectors and add all positive/negative x/y values together to create new ranges from which the random position will be selected. I need to use theory behind quadrants in cartesian geometry to calculate the appropriate direction vectors. (Descartes, 1637)

After that I decrease the transparency of each particle by 2 and delete the particle from the list of particles if it is transparency is 0. So given 200 initial transparency, it takes 100 frames for a particle to disappear. Otherwise I create a surface and render it.

```
def process(self):
    for ent, (par) in self.world.get_component(com.Particles):
        for particle in par.particles:
            ang = particle[3]

            avdict = {0:4, 90:3, 180:4, 270:1}

            vert_vectors_pos = 0
            vert_vectors_neg = 0
            horz_vectors_pos = 0
            horz_vectors_neg = 0

            for key, value in avdict.items():
                vert_vector = round(value*math.sin(math.radians(key) + ang), 4)
                horz_vector = round(value*math.cos(math.radians(key) + ang), 4)
                if vert_vector > 0:
                    vert_vectors_pos += vert_vector
                elif vert_vector < 0:
                    vert_vectors_neg += vert_vector
                if horz_vector > 0:
                    horz_vectors_pos += horz_vector
                elif horz_vector < 0:
                    horz_vectors_neg += horz_vector

            particle[0][1] -= random.randint(int(round(vert_vectors_neg)), int(round(vert_vectors_pos)))
            particle[0][0] += random.randint(int(round(horz_vectors_neg)), int(round(horz_vectors_pos)))

            #i[1] -= random.randint(-1, 3)
            #i[0] += random.randint(-4, 4)
            #r = random.randrange(1, 200)
            r = particle[1]
            particle[2] -= 2
            a = particle[2]
            if particle[0][1] < 0 or a <= 0:
                par.particles.remove(particle)
            else:
                s = pygame.Surface((10, 10), pygame.SRCALPHA)
                s.fill((r, r, r, a))
                self.renderer.blit(s, (particle[0][0], particle[0][1]))
```

Figure 13

Tuning

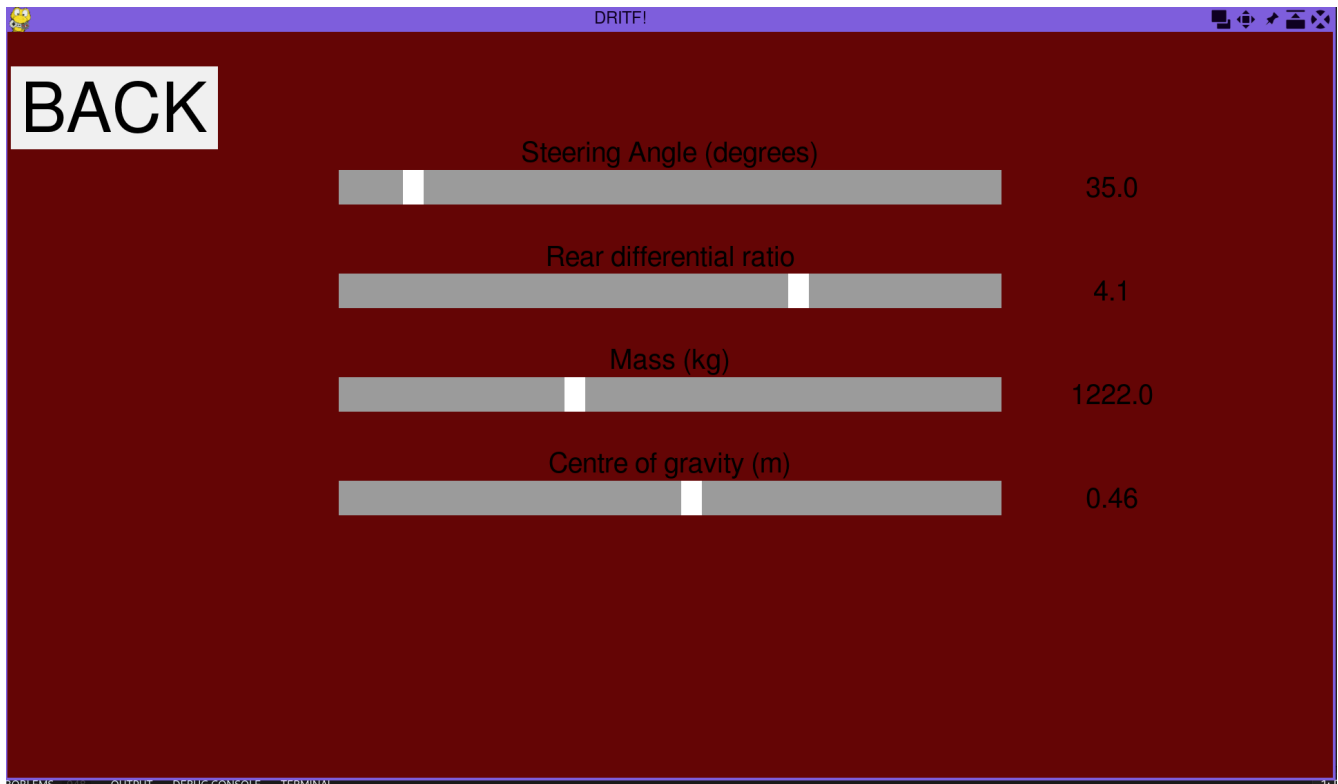


Figure 14

I created my own sliders for car tuning. Now the actual implementation of the slider (Figure 16) is pretty straight forward however it required some clever problem solving to make it work and look good.

First problem I encountered was that the slider was going beyond the bar's length by 30 pixels which is the slider's width. To fix it I had to shorten the bar for calculations by the slider's width, but display it normally.

Then I had to work out the correct range and interval so that it would start at a set lower value instead of 0 and work out the slider position from the variable value or vice versa. I achieved it by offsetting the pixel position by the point where the bar starts, then I divided the how far forward in pixels was the slider and divided it by the interval, then added the minimum value to get the final value corresponding to the sliders position.

Finally, when creating a slider entity I pass the component variable that I want the slider to modify. Due to the way Python works with passing integer or float variable, I could not make the function modular. Therefore I had to use a list of 1 length that contains an integer or a float. This way I would pass the variable reference instead of the variable itself which allowed me to modify it. This likely slows down the performance and it might not be "the right way" to do it but it works great.

```
self.actual_bar_width = bar_width - slider_width
self.range_val = max_val - min_val
self.interval = (self.actual_bar_width/self.range_val)
self.slider_pos = (current_val[0]-min_val)*self.interval+bar_x
```

Figure 15

```

def process(self):
    for ent, (sldr, txt) in self.world.get_components(com.Slider, com.Text):
        click = pygame.mouse.get_pressed()[0]
        mx, my = pygame.mouse.get_pos()

        slider_bar = pygame.Rect(sldr.bar_x, sldr.bar_y, sldr.bar_width, sldr.bar_height)
        pygame.draw.rect(self.renderer, sldr.bar_color, slider_bar)

        if slider_bar.collidepoint((mx, my)):
            if click:
                sldr.slider_pos = mx
                if mx < sldr.bar_x:
                    sldr.slider_pos = sldr.bar_x
                elif mx > sldr.bar_x - sldr.slider_width + sldr.bar_width:
                    sldr.slider_pos = sldr.bar_x - sldr.slider_width + sldr.bar_width

            slider_slider = pygame.Rect(sldr.slider_pos, sldr.bar_y, sldr.slider_width, sldr.bar_height)
            pygame.draw.rect(self.renderer, sldr.slider_color, slider_slider)
            actual_slider_pos = sldr.slider_pos - sldr.bar_x
            val = round(sldr.min_val + actual_slider_pos/sldr.interval, sldr.rounding)
            sldr.current_val[0] = val
            self.drawTextCentred(txt.font, str(sldr.current_val[0]), (0, 0, 0), self.renderer, self.renderer.get_width()/6*5, sldr.bar_y+sldr.bar_height/2)
            self.drawTextCentred(txt.font, txt.text, (0, 0, 0), self.renderer, self.renderer.get_width()/2, sldr.bar_y-25)

```

Figure 16

Engine and architecture

There are some aspects of the engine and the general architecture I would like to talk about.

The way that the pygame and screen initialization as well as game loop are handled is very generic. More interesting is managing the states. There is not one way to manage game loops or states when it comes to game development, even if ECS is used. Only implementations I could find were either super simplistic nested while loops, they do not allow for easy movement between two specific scenes. They are also hard to manage and not modular. I have turned to GitHub for some ideas for game state implementation in ECS however they turned out to be all too complex for what I was doing.

I decided to dedicate one system, it being the only system loaded on world initialization, to manage the state of the game (Figure 17). Then I use a separate class to store load and unload functions for each state as seen in (an example of each of those shown in Figure 18). They add and remove appropriate processes so that they are not executed anymore which saves performance. To make it modular I could put each and everyone function in a dictionary with the state attached to them. If the state is selected then I would go over a dictionary of unload functions and execute all of them but the one with the selected state. Same goes for loading state but I would select only the state that is currently selected.

One thing I have not foreseen was the fact that some states share systems. This caused for example, buttons from the main menu to display in the tuning menu. I managed to quickly fix it by adding an extra variable to the button entity with a state variable and button system checks if the button belongs to a certain state. This is not a long term solution but I did not have time to refactor it further. I would fix it by creating and removing entities however that would require me to store variables for creation somewhere and on top of that I would have to find a way to use the functions from within MyWorld class.

```
def process(self):
    LoadUnload = LoadUnloadd()
    for ent, (state) in self.world.get_component(com.States):
        if state.current_state == "game":
            if state.loaded_state == "game":
                pass
            else:
                LoadUnload.unload_menu(self.world)
                LoadUnload.unload_tuning(self.world)
                LoadUnload.load_game(self.world, self.renderer, self.entities)
                state.loaded_state = "game"
        elif state.current_state == "menu":
            if state.loaded_state == "menu":
                pass
            else:
                LoadUnload.unload_tuning(self.world)
                LoadUnload.unload_game(self.world)
                LoadUnload.load_menu(self.world, self.renderer, self.entities)
                state.loaded_state = "menu"
        elif state.current_state == "tuning":
            if state.loaded_state == "tuning":
                pass
            else:
                LoadUnload.unload_menu(self.world)
                LoadUnload.unload_game(self.world)
                LoadUnload.load_tuning(self.world, self.renderer, self.entities)
                state.loaded_state = "tuning"
```

Figure 17

```

def load_tuning(self, world, renderer, entities):
    world.add_processor(menus.BackgroundProcessor(renderer))
    world.add_processor(menus.ButtonProcessor(renderer))
    world.add_processor(menus.SliderProcessor(renderer))

def unload_tuning(self, world):
    world.remove_processor(menus.BackgroundProcessor)
    world.remove_processor(menus.ButtonProcessor)
    world.remove_processor(menus.SliderProcessor)

```

Figure 18

I managed to achieve the design architecture I have shown in my planning section (Figure 1) with a few alterations. The biggest difference is that I put code responsible for loading states in a different class (Figure 19).

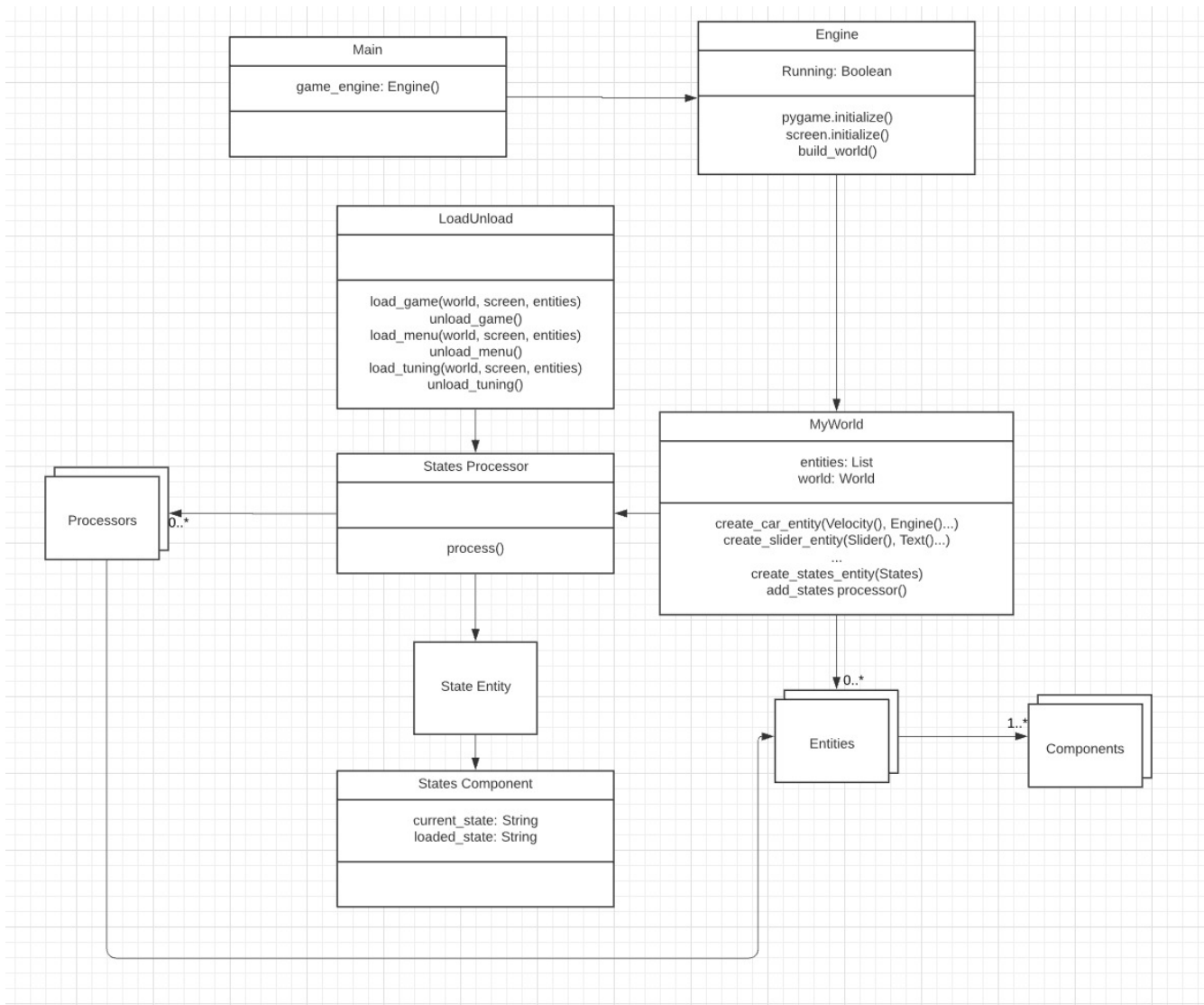


Figure 19

Additionally, the file structure can be seen in Appendix C.

Physics

Appendix B shows names of all the variables I have used in this passage. All of the paragraphs are followed by a screenshot of the code which implements them.

I divided the implementation of car physics into smaller, more digestible systems. From what I noticed when implementing them, the order in which they are run does not matter that much, however it can technically introduce 1 frame of lag so I will try to avoid it.

All of the calculations are done in SI units. Another important thing is delta time. Delta time is a time since the last frame and it is needed for some physics calculations in case that the frames in game are not consistent. It basically serves as time variable in calculations such as $velocity = acceleration \cdot time$

One very important thing to note before starting is that because the car will have a locked differential, meaning both tires spin at the same rate, there is no need to split physics between two tires and we can simulate everything as just a force going forward.

Drivetrain

Firstly, we need to get torque. To get torque at any RPM for our car, we need to calculate the torque the engine produces. This is something beyond the scope of this project.

I decided to just create a torque curve from actual torque values from dyno sheets of Subaru BRZ (Kavanagh, 2012), (DD Performance Research LLC., 2013). Using a dyno sheet of wheel horsepower and torque numbers also saves me from estimating the drivetrain efficiency that reduces the power to the wheels. I put in torque in values from idle RPM of 700, all the way to 7500 RPM at 500 RPM intervals aside of the first 300 RPM. Then I used NumPy to interpolate the values in between and returned it as a list to be used for later calculations.

```
def torque_calc(self):  
    rpm = [700, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500]  
    torque = [94, 108, 122, 148, 176, 179, 167, 156, 174, 180, 177, 177, 172, 163, 133]  
  
    yinterp = interpolate.interpld(rpm, torque, kind="cubic")  
    xnew = np.arange(700, 7500, 1)  
    ynew = yinterp(xnew)  
    return ynew
```

Figure 20

Although unnecessary in the current state, I decided to leave the code for managing engine RPM, RPM_e , in neutral or when clutch is disengaged.

Before calculating the RPM I perform a check whether it is below idle RPM and if it is, I set it to idle RPM. This is done to keep the immersion because if RPM falls too low in a real car, the engine would shut off.

I assume that the only resistance is the friction between the piston and the piston walls and that it is linear. I calculate a friction τ_f opposing the engine torque τ_e that the engine creates so that RPM falls when not on throttle. The friction force equals to 1/60th of the current engine

RPM as I found that to work quite realistic. It does not directly follow physics but it works and creates better immersion. (eran0004, 2015-2019)

$$\tau_f = (RPM_e - RPM_i + 1) \div 60$$

Then I retrieve the torque from the torque curve created earlier and the current RPM. We use torque at the engine and a throttle position to get the torque that the engine is outputting. (Monster, 2003)

I also convert from the engine RPM to SI units of Rad/s for ease of calculations.

We also need an inertia of the engine I so that we can follow Newton's second law for rotational motion to calculate angular acceleration of the flywheel α . Assuming that the flywheel is a perfect cylinder and that the mass of every aspect of the rotating assembly is negligible compared to the flywheel, we can calculate the inertia from flywheels mass m and its radius r . (eran0004, 2015-2019)

$$I = 0.5 \cdot m \cdot r^2$$

$$\alpha = (\tau_e \cdot t - \tau_f) \div I$$

This is also the first time where dt comes into use to calculate the angular velocity from angular acceleration.

$$\omega = \alpha \cdot dt$$

After that it is just a matter of converting the angular velocity back to RPM and then I perform another check to see if the RPM is within its operating range and limit it if it is not.

Figure 22 shows code of unfinished feature of flywheel and clutch mating. I have not found a way to send the impulse to the wheels before the deadline.

```

def process(self):
    for ent, (eng, grb, cha, dt, cvelo) in self.world.get_components(com.Engine, com.GearBox, com.Chassis, com.DeltaTime, com.CarVelocity):

        if grb.gear_ratios[grb.current_gear] == 0 or grb.clutch == True:
            # Manages the idle RPM
            if eng.rpm < eng.idle:
                eng.rpm = eng.idle

            grb.eng_inertia = 0.5 * 9.34 * 0.16**2
            eng_friction_torque = (eng.rpm - eng.idle + 1)/60
            eng_torque = (eng.torque_curve[int(eng.rpm)-eng.idle] * eng.throttle)
            eng_ang_vel = eng.rpm * constants.RPM_to_RADS
            eng_ang_acc = (eng.torque_curve[int(eng.rpm)-eng.idle] * eng.throttle - eng_friction_torque) / grb.eng_inertia

            eng_ang_vel += eng_ang_acc * dt.dt
            eng.rpm = int(round(eng_ang_vel * constants.RADS_to_RPM))

            # Does not let RPM go over or under the rev limit
            if eng.rpm < eng.idle:
                eng.rpm = eng.idle
            elif eng.rpm > eng.rev_limit:
                eng.rpm = eng.rev_limit

            # Calculate engine momentum
            grb.eng_momentum = grb.eng_inertia * eng.rpm * constants.RPM_to_RADS

            grb.trans_inertia = 1.5
            clutch_rpm = int((abs(cvelo.velV.x)/cha.wheel_radius) * (grb.rear_diff[0] * abs(grb.gear_ratios[grb.current_gear])) * constants.RADS_to_RPM)

            grb.trans_momentum = grb.trans_inertia * clutch_rpm * constants.RPM_to_RADS

```

Figure 21

```

else:
    grb.eng_inertia = 0.5 * 9.34 * 0.16**2
    grb.trans_inertia = 1.5

    grb.eng_momentum = grb.eng_inertia * eng.rpm * constants.RPM_to_RADS
    clutch_rpm = int((abs(cvelo.velV.x)/cha.wheel_radius) * (grb.rear_diff[0] * abs(grb.gear_ratios[grb.current_gear])) * constants.RADS_to_RPM)
    grb.trans_momentum = grb.trans_inertia * clutch_rpm * constants.RPM_to_RADS

```

Figure 22

Now having torque values for each specific engine RPM we can calculate actual torque at any given time. We also need to know combined gear ratios of differential and gearbox. Both of these values are set for each car, we just need to retrieve them and multiply them to get total gear ratio GR. Then to get acceleration of the wheel, we divide the car's longitudinal velocity v_{cx} with the radius of the wheel r_w . At the end we need to convert it from SI radians/s to RPM with a constant of $60 \div 2\pi$. (Monster, 2003) This gives as an equation:

$$RPM_e = v_{cx} \div r_w \cdot GR \cdot (Rad/s \text{ to } RPM)$$

I was confused at first as to how I get engine RPM from the velocity of the car if I need the engine RPM to actually move the car. On top of that I tried implementing a clutch at the same time which furthered the confusion and I spent a week with no progress.

After calculating the RPM I perform a check whether it is below idle RPM and if it is, I set it to idle RPM.

Then if the RPM is within the engine's operating range and the handbrake is not applied, a torque value from the torque curve is assigned to a variable.

Then I perform another check to see if the RPM is within its operating range and limit it if it is not.


```

eng.rpm = int((abs(cvelo.velV.x)/cha.wheel_radius) * (grb.rear_diff[0] * abs(grb.gear_ratios[grb.current_gear])) * constants.RADS_to_RPM)

# Manages the idle RPM
if eng.rpm < eng.idle:
    eng.rpm = eng.idle

# Calculates the torque
if eng.rpm >= eng.idle and eng.rpm < eng.rev_limit:
    if cha.ebrake == 1:
        eng.torque = 0
    else:
        eng.torque = eng.torque_curve[int(eng.rpm) - eng.idle]
else:
    eng.torque = 0

# Does not let RPM go over or under the rev limit
if eng.rpm < eng.idle:
    eng.rpm = eng.idle
elif eng.rpm > eng.rev_limit:
    eng.rpm = eng.rev_limit

```

Figure 23

The torque at the wheels τ_w differs from the torque at the engine so we need to account for that. If clutch is pressed in or neutral is selected, the torque at the wheels is obviously equal to zero. We use torque at the wheels τ_e and a throttle position t to get the torque that the engine is outputting but to get the torque at the wheels, we need to multiply it by total gear ratio. (Monster, 2003)

$$\tau_w = \tau_e \cdot t \cdot GR$$

In real world throttle position does not linearly affect the torque but the approximation is intuitive to the user and works quite well to what is experienced in a real car.

It is also important to touch upon gear ratios here. Higher gears in a car have lower gear ratios. Therefore from this information and the equation, we can infer that as we go up the gears, we get lower torque at the wheels. This is because for each engine revolution, there are more revolutions of the wheels but they are weaker as there are less engine revolutions behind them. This in turn allows for higher top speed as the wheels turn faster. (Monster, 2003)

Before we can calculate the total forward force, we also need to calculate the total brake force F_{tb} . I achieve it by applying brake force linearly by just multiplying the input by the maximum brake force F_{fb} . However because hand brake, hb , and foot brake, fb , can be applied simultaneously, I want to limit the brake power so that it does not add up beyond the maximum brake force that the stronger brake, in this case a foot brake, can exert. This results in such equation:

$$F_{tb} = \min(fb \cdot F_{fb} + hb \cdot F_{hb}, F_{fb})$$

Then I perform a check so that when the driving very slow, brake cannot be applied. Otherwise it caused the car to be jerked back. (Monster, 2003)

Finally we can calculate total forward force F_w from torque at the wheels with:

$$F_w = \tau_w \div r_w - F_{tb}$$

Although the physics work great on paper. While programmed braking caused my car to accelerate backwards. I fixed it by multiplying the total brake force by the sign of the car's velocity so that it opposes the acceleration force.

```
def process(self):
    for ent, (velo, ff, eng, grb, cha, cvelo) in self.world.get_components(com.Velocity, com.ForwardForce,
    # Calculates the wheel torque from engine torque and throttle
    if grb.gear_ratios[grb.current_gear] == 0 or grb.clutch == True:
        torque = 0
    else:
        torque = (eng.throttle * eng.torque) * (grb.rear_diff[0] * grb.gear_ratios[grb.current_gear])

    total_brake = min(cha.brake*cha.brake_power + cha.ebrake*cha.ebrake_power, cha.brake_power)

    if(velo.velV.magnitude() < 0.5 and eng.throttle == 0):
        total_brake = 0

    ff.forward_force = torque / cha.wheel_radius - total_brake*np.sign(cvelo.velV.x)
    ff.sideway_force = 0
```

Figure 24

Having found a way to apply force to the car directly, we need to find resistance as it is a real life simulation and resistance is a big part of a car's performance, especially when it comes to top speed. Each car has a different coefficient of drag C_d and frontal area A_f which dictates air resistance AR . There is also density of air coming into play but I will default it to average density of air at sea level. For rolling resistance RR I used a constant that I found through some experimentation. Rolling resistance has a linear relation with car's speed and air resistance grows exponentially. That means that at low speeds rolling resistance matters more than air resistance, however their impact on the performance reverse at higher speed. We also have to apply it in the reverse direction just like the brakes by multiplying the drag by the sign of the forward velocity. (Monster, 2003) Having both of these we can calculate the total force on the car F_t :

$$AR = 0.5 * C_d * A_f * \rho_a$$

$$F_d = RR * V_c + AR * V_c^2$$

$$F_t = \tau_w \cdot r_w - F_d$$

Rest of the calculations are pretty straightforward. We use Newton's second law to calculate the car's acceleration a_c from total force and mass m . (Monster, 2003)

$$a_c = F_t \div m$$

We need to however aim the acceleration in the right direction, converting from acceleration in terms of car's direction a_c to acceleration in terms of world coordinates a_w . To achieve that we multiply the longitudinal and lateral acceleration in x and y direction by trigonometric functions based on the car's angle of orientation θ_c . (Monster, 2003)

$$a_{wx} = \cos(\theta_c) \cdot a_{cy} + \sin(\theta_c) \cdot a_{cx}$$

$$a_{wy} = \cos(\theta_c) \cdot a_{cx} - \sin(\theta_c) \cdot a_{cy}$$

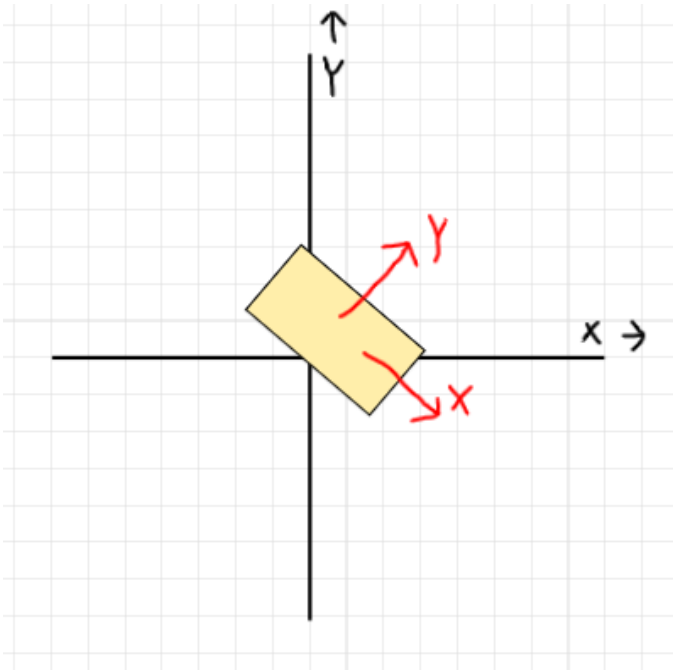


Figure 25

The sketch above showcases the coordinate system of the world in black and car's direction in red.

```
def process(self):
    for ent, (ster, accel, caccel, velo, cvelo, ff, eng, cha) in self.world.get_components(com.Steering, com.Acceleration, com.CarAcceleration, com.Velocity, com.CarVelocity, com.ForwardForce, com.
        # Resistance and total force as well as acceleration

        drag_force_x = -constants.RR_COEFF * cvelo.velV.x - constants.DRAG_COEFF * cvelo.velV.x * abs(cvelo.velV.x);
        drag_force_y = -constants.RR_COEFF * cvelo.velV.y - constants.DRAG_COEFF * cvelo.velV.y * abs(cvelo.velV.y);

        total_force_x = ff.forward_force + drag_force_x
        total_force_y = ff.sideway_force + drag_force_y + math.cos(ster.steer_angle) * (ster.friction_front_left + ster.friction_front_right) + (ster.friction_rear_left + ster.friction_rear_right)
        if(velo.velV.magnitude() < 0.5 and (eng.throttle == 0 or cha.brake > 0.8 or cha.ebrake == 1)):
            total_force_x = 0
            total_force_y = 0

        #print(velo.velV.y)
        caccel.accV.x = total_force_x / cha.mass[0]
        caccel.accV.y = total_force_y / cha.mass[0]

        accel.accV.x = ster.cs * caccel.accV.y + ster.sn * caccel.accV.x
        accel.accV.y = ster.cs * caccel.accV.x - ster.sn * caccel.accV.y
```

Figure 26

From there we can calculate velocity in terms of world's coordinates v_w by multiplying car's acceleration in terms of world's coordinates a_w with delta time. Same goes for position p_w calculation but this time we multiply velocity by delta time. (Monster, 2003)

$$v_w = a_w \cdot dt$$

$$p_w = v_w \cdot dt$$

```
def process(self):
    for ent, (accel, velo, dt) in self.world.get_components(com.Acceleration, com.Velocity, com.DeltaTime):
        # Velocity
        velo.velV.x += accel.accV.x * dt.dt
        velo.velV.y += accel.accV.y * dt.dt
```

Figure 27

```
def process(self):
    for ent, (dt, velo, pos, rect) in self.world.get_components(com.DeltaTime, com.Velocity, com.Position, com.Rect):
        pos.posV.x += velo.velV.x * dt.dt
        pos.posV.y += velo.velV.y * dt.dt
        rect.rect.x = pos.posV.x*constants.SCALE
        rect.rect.y = pos.posV.y*constants.SCALE
```

Figure 28

Steering

Implementation of the steering proved to be less problematic than implementation of the drivetrain. One big problem I encountered was calculating delta time at a wrong place causing my car to spin at extraordinarily high rate. While not strictly related to steering, I did not notice it before because the only place I used it before was for velocity and position. I did not have a point of reference to dictate whether the car was driving too fast or too slow. Due to the fact that driving forward and backwards looked okay, it took me quite some time to find the source of the problem to be delta time, as I did not suspect it to be the cause.

Firstly I calculate the sin and cos of the car's angle for use in the acceleration calculation as mentioned earlier and velocity calculator now.

I follow the same principles as converting the acceleration in relation to the car, to acceleration in relation to the world. This time I convert from velocity in relation to the world v_w to velocity in relation to the car v_c . (Monster, 2003)

$$v_{cx} = \cos(\theta_c) \cdot v_{wy} + \sin(\theta_c) \cdot v_{wx}$$

$$v_{cy} = \cos(\theta_c) \cdot v_{wx} - \sin(\theta_c) \cdot v_{wy}$$

To calculate weight on each while moving, I assumed that the weight distribution from side to side is the same. Weight transfer affects the grip of each wheel which is very important when racing and acceleration is high.

Stationary weight on each wheel is quite simple to work out using moments. For a car as it is just a distance between axle and the centre of gravity d_x divided by wheelbase WB, times distance to the centre of gravity from axle length divided by axle track AT (0.5 in our case because the we assumed side to side weight distribution is equal), times gravity g times mass. (Monster, 2003)

However when we put motion into it, things get more complex. Before multiplying everything by mass, we need to add or takeaway (depending on the direction of acceleration) lateral acceleration times centre of gravity CG divided by wheelbase, and similarly, add or takeaway acceleration sideways times centre of gravity divided by axle track. (Monster, 2003) Here is a calculation for that:

$$W_w = m \cdot (d_x / WB \cdot 0.5 \cdot g \pm a_{cx} \cdot CG \div WB \pm a_{cy} \cdot CG \div AT)$$

Turning the car also takes place around its axis which is at the centre of gravity. Therefore the yaw rate or angular velocity ω has to be multiplied by the percentage weight distribution WD between the rear and the front to get yaw speed YS. (Monster, 2003)

$$YS_f = WD_f * \omega$$

$$YS_r = -WD_r * \omega$$

One more step before calculating the friction between tyres and ground is calculating slip angles. Slip angle S is an angle between where the tyre is heading and where it is pointing.

We need to calculate sideslip and rotation angle. Sideslip SS is the angle between where the car is heading and where it is pointing, similarity to slip angle. And rotation angle θ_r is my previous yaw speed, which is basically lateral velocity divided by the car's longitudinal velocity. The calculation for front tyres also takes away the steering angle θ_s applied by the player because the tyres do not always point parallel to the car. I joined these two equations in my code as it runs faster. (Monster, 2003) Here are the equation's:

$$SS = \arctan(v_{cy} \div v_{cx})$$

$$\theta_r = \arctan(YS \div v_{cx})$$

$$S_f = \arctan((v_{cy} + YS_f) \div v_{cx}) - v_{cx} \cdot \theta_s$$

$$S_r = \arctan((v_{cy} + YS_r) \div v_{cx})$$

In a real scenario all tires could lock without ABS however that is hard to implement so I simplified it a bit. Only the rear tyres lock and they lose grip progressively as the hand brake is depressed. I did it this way as it is intuitive and helps the car slide well making drifting while using a handbrake much more enjoyable than foot brake drifting. The value of 0.7 is there as that is roughly the slip ratio of a locked tire. (Cho et al., 2007)

Finally we can work out the friction at each corner. Tire grip is the limiting factor of the grip. Therefore we calculate the coefficient of friction itself by multiplying cornering stiffness CS by slip angle and limit it by maximum tyre grip G_t so that it cannot be bigger than it. Then we multiply it by the weight on each tire as more pressure on the surface causes more grip. (Monster, 2003) This all can be put into a simple friction force F_f formula.

$$F_f = [CS \cdot S]_{G_t}^{G_t} \cdot W_w$$

```

def process(self):
    for ent, (ster, velo, cvelo, accel, caccel, cha, dt, eng) in self.world.get_components([com.Steering, com.Velocity, com.CarVelocity, com.Acceleration, com.CarAcceleration, com.Chassis, com.DeltaTime], com.Engine):
        ster.sn = math.sin(ster.heading)
        ster.cs = math.cos(ster.heading)

        cvelo.velV.x = ster.cs * velo.velV.y + ster.sn * velo.velV.x
        cvelo.velV.y = ster.cs * velo.velV.x - ster.sn * velo.velV.y

        weight_front_left = cha.mass[0] * (cha.cg_rear_axle/cha.wheelbase*0.5 * constants.GRAVITY - caccel.accV.x * cha.cg_height[0]/cha.wheelbase - caccel.accV.y * cha.cg_height[0]/(cha.width-cha.wheel_width))
        weight_front_right = cha.mass[0] * (cha.cg_rear_axle/cha.wheelbase*0.5 * constants.GRAVITY - caccel.accV.x * cha.cg_height[0]/cha.wheelbase + caccel.accV.y * cha.cg_height[0]/(cha.width-cha.wheel_width))
        weight_rear_left = cha.mass[0] * (cha.cg_rear_axle/cha.wheelbase*0.5 * constants.GRAVITY + caccel.accV.x * cha.cg_height[0]/cha.wheelbase - caccel.accV.y * cha.cg_height[0]/(cha.width-cha.wheel_width))
        weight_rear_right = cha.mass[0] * (cha.cg_rear_axle/cha.wheelbase*0.5 * constants.GRAVITY + caccel.accV.x * cha.cg_height[0]/cha.wheelbase + caccel.accV.y * cha.cg_height[0]/(cha.width-cha.wheel_width))

        yawSpeedFront = cha.cg_front_axle * ster.yawRate
        yawSpeedRear = -cha.cg_rear_axle * ster.yawRate

        slipAngleFront = math.atan2(cvelo.velV.y + yawSpeedFront, abs(cvelo.velV.x)) - np.sign(cvelo.velV.x) * ster.steer_angle
        print("F * = ster.slipAngleFront")
        slipAngleRear = math.atan2(cvelo.velV.y + yawSpeedRear, abs(cvelo.velV.x))
        print("slipAngleRear")
        tire_grip_front = cha.tire_grip * (1 - cha.ebrake * (1 - 0.7))
        tire_grip_rear = cha.tire_grip * (1 - cha.ebrake * (1 - 0.7))

        ster.friction_front_left = np.clip(-5*slipAngleFront, -tire_grip_front, tire_grip_front) * weight_front_left
        ster.friction_front_right = np.clip(-5*slipAngleFront, -tire_grip_front, tire_grip_front) * weight_front_right
        ster.friction_rear_left = np.clip(-5.2*slipAngleRear, -tire_grip_rear, tire_grip_rear) * weight_rear_left
        ster.friction_rear_right = np.clip(-5.2*slipAngleRear, -tire_grip_rear, tire_grip_rear) * weight_rear_right

```

Figure 29

Finally we can get to actually moving the car around its axis. This part is a rehash of a lot of the drivetrain physics but working with angular motion instead of linear motion. We start with calculating the angular torque τ_a , then angular acceleration α which uses inertia I instead of mass (they are equal in our game), then we increase the angular velocity by acceleration times delta time, and finally we increase the car's angle by the angular velocity time delta time. The only real difference between the equations earlier is how we calculate the torque. (Monster, 2003)

$$\tau_a = F_{ff} \cdot d_f - F_{fr} \cdot d_r$$

$$\alpha = \tau_a \div I$$

$$\omega = \alpha \cdot dt$$

$$\theta_c = \omega \cdot dt$$

I also have a check so that when the car's velocity is low and when certain inputs are being pressed such as no throttle or fully depressed brake, a lot of the variables responsible for movement turn to zero to prevent jerking back and forth.

```

angularTorque = (ster.friction_front_left + ster.friction_front_right) * cha.cg_front_axle - (ster.friction_rear_left + ster.friction_rear_right) * cha.cg_rear_axle;

if(velo.velV.magnitude() < 0.5 and (eng.throttle == 0 or cha.brake > 0.8 or cha.ebrake == 1)):
    cvelo.velV.y = 0
    cvelo.velV.x = 0
    velo.velV.y = 0
    velo.velV.x = 0
    accel.accV.y = 0
    accel.accV.x = 0
    caccel.accV.y = 0
    caccel.accV.x = 0
    angularTorque = 0
    ster.yawRate = 0

angularAccel = angularTorque / cha.inertia

ster.yawRate += angularAccel * dt.dt
if(velo.velV.magnitude() < 1 and ster.steer_angle < 0.05):
    ster.yawRate = 0
ster.heading += ster.yawRate * dt.dt

```

Figure 30

Results and evaluation

Tests

Test case ID: 1		Purpose: Check top speed of a car in each gear and whether the gearbox works as expected.	
Steps: 7		Preconditions: Stock car configuration, map at least 500 tiles long	
Step No:	Procedure	Expected response	Pass/ Fail
1	Accelerate car in first gear until maximum engine RPM	Have a top speed of 56 km/h +- 15%	Pass
2	Accelerate a car in second gear until maximum engine RPM	Have a top speed of 93 km/h +- 15%	Pass
3	Accelerate a car in third gear until maximum engine RPM	Have a top speed of 134 km/h +- 15%	Pass
4	Accelerate a car in fourth gear until maximum engine RPM	Have a top speed of 169 km/h +- 15%	Pass
5	Accelerate a car in fifth gear until maximum engine RPM	Have a top speed of 204 km/h +- 15%	Pass
6	Accelerate a car in sixth gear until maximum engine RPM	Have a top speed of 217 km/h +- 15%	Pass
7	Accelerate a car in reverse for until maximum engine RPM	Have a top speed of xx km/h +- 15%, in reverse	Pass
Comments: All of them passed but very close to the margin error, all of the speeds I recorder were lower than expected other than the top speed which was higher and seemed to agree with the official top speed			

Test case ID: 2		Purpose: Check for car's acceleration	
Steps: 2		Preconditions: Stock car configuration, manual gearbox, map at least 500 tiles long	
Step No:	Procedure	Expected response	Pass/ Fail
1	Accelerate from a stop up to 100 km/h	Accelerate from 0 to 100 km/h in 7.6 seconds +- 15%	Fail
2	Accelerate from a stop up to 200 km/h	Accelerate from 0 to 200 km/h in 34.9 seconds +- 15%	Fail
Comments: Acceleration is faster, however I believe that is because I went with a sequential gearbox and that takes away the time it takes to change gears.			

Test case ID: 3		Purpose: Check for maximum sustained lateral g force	
Steps: 2		Preconditions: Stock car configuration	
Step No:	Procedure	Expected response	Pass/ Fail
1	Turn max to the right and accelerate just until the car starts to slip, hold that speed	Achieve a sustained lateral g force of 0.90 g	
2	Turn max to the left and accelerate just until the car starts to slip, hold that speed	Achieve a sustained lateral g force of 0.90 g	
Comments: I did not implement a way to measure a lateral g force therefore I could not run this test			

Test case ID: 4		Purpose: Check if the getting drift points is consistent	
Steps: 3		Preconditions: Stock car configuration	
Step No:	Procedure	Expected response	Pass/ Fail
1	Accelerate to top speed in 1st gear, turn all the way left then press handbrake, repeat 5 times to calculate average points	Get a consistent amount of points, within +/-10% of each other	Pass
2	Accelerate to top speed in 1st gear, turn all the way left then press handbrake, repeat 5 times to calculate average points		
3	Just drift around until getting 100,000 points	Managed to get 100,000 points	Pass
Comments: Works as expected			

Test case ID: 5		Purpose: Check for memory leaks	
Steps: 1		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Measure the memory usage, run the game for 10 minutes, measure the memory usage again	The memory usage at the end should be no bigger than 15% than at the start	Fail
Comments: Memory usage went up tremendously to the point where the game took up 4GB of RAM after 10 minutes starting at 750MB			

Test case ID: 6		Purpose: Check going between menus	
Steps: 3		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Go between to game and back to main menu	Menu menu should look and function exactly the same	Fail
2	Go to tuning menu and back to main menu	Main menu should look and function exactly the same	Pass
3	Go to game and then back to main menu and then tuning menu	Tuning menu should look exactly the same	Fail
Comments: Tuning menu is not accessible, because the button disappears, after the game state is run.			

Test case ID: 7		Purpose: Check if tuning settings work	
Steps: 2		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Go to tuning menu and adjust settings drastically...	The numbers display correctly and sliders slide properly	Pass
2	...and then back to main menu and then to game	The car is affected by the tuning settings	Pass
Comments: Works perfectly!			

Test case ID: 8		Purpose: Clutch	
Steps: 2		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Rev up the engine to 7500 RPM with car in gear and clutch disengaged	The car does not move	Pass
2	Engage the clutch and let off the throttle	The car accelerates quickly to a speed of roughly 15 km/h +- 10 km/h	Fail
Comments: Engine does not send a pulse to the wheels through the clutch			

Physics

I am really happy with the physics implementation as it touches upon a lot of aspects that are definitely essential when drifting and most of them work as can be seen from the tests.

Top speed in each gear is on the very margin of error I allowed which is not amazing but I suspect it might be to do with the rolling and/or air resistance. They all come up short and last gear is limited by air resistance and this makes me think that another cause of this might be the engine does not really propel the car at idle like a real car would so the overall rev limiter is actually 700 RPM lower.

Acceleration is substantially higher than real acceleration. I believe however that this problem is not directly with the physics calculations themselves but with the inputs. The lack of clutch and just the fact that the user is using a gamepad where pressing and releasing a button takes 0.1 sec compared to a real car where changing gears takes way more time. Accounting for the fact that vertical (eg. 1st to 2nd) shift time takes 0.5sec and horizontal (eg. 2nd to 3rd) shift time takes 1 sec, the final acceleration values would be well within 10% of the real acceleration values.

I could not measure the grip through maximum lateral force because I did not implement a way of measuring it.

One thing I wish I ended up making work is the fully manual gearbox with a clutch as I feel like that is very important to drifting.

The implementation is rather simplistic however quite immersive for what it is. It serves as an amazing basis to build up with other features as specific parts, such as advanced tyre grip physics or engine model, can be just plugged in without modifying the rest of it.

Engine

The sliders work amazing though for many different variables with no hardcoded values.

The UI is not working properly as the button goes missing after going back to the main menu from a game. I tried to locate the bug but I could not in such a short amount of time therefore I have no idea what the cause could be.

I am also not entirely happy with how I went about changing states. The idea was right but the execution missed the mark. Instead of adding and removing the processors, I will need to add and remove processors and entities when changing states. This fixes the problem I where I had to put a state variable for a button as the buttons were displayed on both states because they both used the same processor. This also possibly frees up memory so the game gets better performance.

Rendering systems are not modular at all either because I did not have time to refactor them, so there is one for a car and one for objects, and another one for tilemap.

The steering input is contained within the main game loop. it is harder to modify the values and it is likely slower than a proper implementation. I tried implementing input as a system however it seemed like the input it was getting was very patchy. One time it would work and other times it would not. The current implementation works great from a user standpoint and is very responsive and there is no ghosting.

The engine is mostly just a framework with not many transferable features right now. It is also not very modular. This will require rewriting some aspects of it when writing more features, especially if they involve more than one car. Because it is not big though, I could make it into a basic framework over a weekend that would allow me to make games of any type while taking care of inputs, rendering, changing game states etc.

Other aspects

The drift points system works as expected and longer, faster drifts award more points. Testing this part of the would require.

Particles came out looking great. The performance is adequate with how many of them are there. One small thing is that they follow the middle of the screen however that is not very perceptible in our game unless driving at a huge speed. The bigger problem is that they are not modular at all. If I was to need different particles I would need to create a separate system and change a lot of hard coded variables instead of just passing arguments into a function.

I feel like 2D was a good idea from a programming standpoint as it allowed me to program more features without spending lots of times on making geometry and assets work. However it made the game not as immersive as I would ideally want it to be, especially considering it is a simulator.

I think the gamepad controls work really well and aid the immersion with the analog inputs. When playing on a keyboard I do not feel as much control over the car which causes me to change my inputs, eg. instead of holding a trigger at 50% to keep RPMs the same, I need to tap the W key which breaks the immersion and makes the inputs more twitchy.

Looking back at the timeline

	19-Feb	26-Feb	05-Mar	12-Mar	19-Mar	26-Mar	02-Apr	23-Apr	30-Apr	07-May	14-May
Review Meetings											
Report											
Boilerplate code											
Gamepad support											
Powertrain physics											
Suspension physics											
Tuning											
Gameplay elements and level design											
Polishing											
Testing & bugfixing											

Figure 31

Looking back at the timeline, a lot of things went differently. Starting off with the report, I ended up not writing it continuously, but in 3 chunks. This was mostly because I wanted to get started on the project quickly and make a good basis for the rest of the features. After I did that I sat down and wrote “Introduction”, “Background” and “Design” sections of the report. Then I did more coding and when I had the general idea of where the project was going to end up, I went over the stuff I already wrote and added “Future work” section. Then after I finished programming I wrote “Implementation”, “Results”, “Conclusion” and “Reflection” sections.

Initially I was hoping to write down the implementation as I was writing it however that proved difficult due to how fast everything was changing. That would require frequent rewriting of some sections which would be a waste of time. I also wanted to have specification and design done before starting the majority of the work, however I felt a lack of experience in that area and needed to start shaping the project a bit before getting a good idea of how it will end up.

I expected some roadblocks when writing the physics however they were more severe than I thought. If it was not for them I the physics system would be much fleshed out.

Another aspect I did not take into account at all before was refactoring. I did the majority of it near the end of the project however I feel like it would have been more productive if I spent one day a week refactoring what I wrote in the past 4 days.

Refleciton

There are many new considerations that I did not see before when it comes to planning out such a big project. Next time I plan a big project I would allow myself some time at the end of each week for refactoring and a few days extra before the deadline. I also would not be as ambitious with report writing as I expected to be while the whole project is going at such a quick pace.

Another thing many do not consider is allowing yourself breaks. There were some weekends where I was stuck on a problem and worked them out through the weekend however other times I came up with a new solution on Monday even though I had time off on the weekend. This seems counterproductive but I found that a lot of learning is done when your brain is resting and it allows you to destress which slows down work.

One more takeaway is that certain algorithms work on paper, in isolation or in someone else's project, does not mean they will work straight away in your own project, especially when adding new features that interact with those.

Something I did not realize before is the importance of refactoring. All the projects I have wrote before were short and after writing them I did not have to worry about it anymore. Aside from the fact I want to develop this project further, just over the short period of 10 weeks I noticed the increasing technical debt. I tried to keep my code clean but sometimes the "prototypes" were left in production however as time went on I had to end up rewriting and refactoring some parts to make them work with other aspects of the program.

This is also the first time I wrote a performance critical application. I only had to optimize my particles as they were causing low framerate but it gave me another insight into creating real time applications. This taught me how to identify performance critical parts of code.

I also did not instantly realize I had a memory leak. I am glad I thought of testing it as it allowed me to realize it is there and it is the highest priority fix now. I have not learnt how to find a source of a memory leak but I am planning to really soon.

I have also never wrote a proper documentation before and while the report touches upon only the most important aspects of my game, it gave me a chance to expand those skills so that it would come easier in future.

Finally, I found out that writing a game from the ground up instead of using an already made game engine reinforces good coding practices through requiring me to make decoupled code and just comprehending all of that code requires it to be clean. It allows for more freedom in implementation and most importantly, teaches the basics of game development and just computer science in general, much better than creating a game in an engine would.

Conclusion

The aim of this project was to create a 2D driving simulator game with a focus on drifting. I feel like I achieved that however I came up short with a few smaller things that could make this game better however they mostly related to gameplay aspects which were secondary to the physics. The game met most of the test I set out for it which further strengthens my opinion. I believe it is close to being a minimum viable product. it is quite realistic and mildly immersive but I think that is because 2 dimensional graphics cannot “sell” the experience as much.

Choosing ECS gave the code some structure which saved me some time further down the line as it increased prototyping speed and decreased the amount of iterations of each module I had to do due to the way it retrieves variables.

The engine, world building and some of the functions could be reused into creating a very basic real game engine that I could develop other games on. This would cut down time it takes to setup universal thing such as window, controls or rendering.

Most importantly, this project was an amazing learning experience and taught me so many things I have only heard about but have not had a chance to experience. From refactoring, through week long roadblocks I had to overcome myself, to working with performance critical application on this scale for the first time.

Future Work

This is a really good prototype and I would like to carry on working on it. I believe that with some extra work it could become a fully fledged game for anyone to play.

A super simple feature that I have not thought of would be to save the top scores, car setups etc. as currently the game just starts up fresh each time.

Another simple thing for a full release would be graphical and sound assets. I would like something original that is at least semi-realistic and improves the immersion. The fact that I am using a tileset is a bit of a problem when creating tracks too as it limits the possibilities. A free flowing way to create tracks would be really interesting and it would even allow me to replicate famous tracks such as the ones at Ebisu or Kansai/Suzuka circuit. An in game track editor would be amazing and it would allow the players to create circuits for endless hours of play.

The game would obviously need more tracks and possibly some type of career mode or progression. Highest scores are fun but they get old quickly, especially without a multiplayer leaderboard, which is a good idea. Carrying on with that idea, introduction of drift battles or tandems would provide hours of fun however that would require a ton of work and should be left till the end.

I am really glad with how the physics came out and they do not need much work in their current state, just some fine tuning of some constants. Although there are many more physics features I could add such as more advanced suspension and real engine power calculations based on the engine physics instead of getting the values from torque table. This would expand car tuning to include modifying the engine and more advanced suspension tuning as well. Fixing the clutch would also be a priority. Cosmetic modification would be needed to make this game more enjoyable as individuality is a huge part of drifting. Especially if I was to introduce multiplayer mode.

An interesting addition from a programming perspective would be AI. I believe that from a player perspective multiplayer would have been better due to the fact that you can have real, competitive people to play against. However AI would be really interesting to implement due to the interesting rules of drifting compared to traditional racing. A machine learning project like this could be a good idea for a masters dissertation.

A lot of these changes would require refactoring the current codebase to make some aspects of it more modular and allow bigger range of inputs and outputs.

As the game grows I hope that PyGame's performance does not hinder me however PyGame 2 is slowly being updated to port over more of SDL2 functionality and the performance is improving. I also have not optimized the game at all aside of the particle system so there is definitely room for improvement. However if that was indeed the case I would plan to either move the project to C++ or Rust as I am interested in learning Rust.

Appendix A - tests

Test case ID: 1		Purpose: Check top speed of a car in each gear and whether the gearbox works as expected.	
Steps: 7		Preconditions: Stock car configuration, map at least 500 tiles long	
Step No:	Procedure	Expected response	Pass/ Fail
1	Accelerate car in first gear until maximum engine RPM	Have a top speed of 56 km/h +- 15%	
2	Accelerate a car in second gear until maximum engine RPM	Have a top speed of 93 km/h +- 15%	
3	Accelerate a car in third gear until maximum engine RPM	Have a top speed of 134 km/h +- 15%	
4	Accelerate a car in fourth gear until maximum engine RPM	Have a top speed of 169 km/h +- 15%	
5	Accelerate a car in fifth gear until maximum engine RPM	Have a top speed of 204 km/h +- 15%	
6	Accelerate a car in sixth gear until maximum engine RPM	Have a top speed of 217 km/h +- 15%	
7	Accelerate a car in reverse for until maximum engine RPM	Have a top speed of xx km/h +- 15%, in reverse	
Comments:			

Test case ID: 2		Purpose: Check for car's acceleration	
Steps: 2		Preconditions: Stock car configuration, manual gearbox, map at least 500 tiles long	
Step No:	Procedure	Expected response	Pass/ Fail
1	Accelerate from a stop up to 100 km/h	Accelerate from 0 to 100 km/h in 7.6 seconds +- 15%	
2	Accelerate from a stop up to 200 km/h	Accelerate from 0 to 200 km/h in 34.9 seconds +- 15%	
Comments:			

Test case ID: 3		Purpose: Check for maximum sustained lateral g force	
Steps: 2		Preconditions: Stock car configuration	
Step No:	Procedure	Expected response	Pass/ Fail
1	Turn max to the right and accelerate just until the car starts to slip, hold that speed	Achieve a sustained lateral g force of 0.90 g	
2	Turn max to the left and accelerate just until the car starts to slip, hold that speed	Achieve a sustained lateral g force of 0.90 g	
Comments:			

Test case ID: 4		Purpose: Check if the getting drift points is consistent	
Steps: 3		Preconditions: Stock car configuration	
Step No:	Procedure	Expected response	Pass/ Fail
1	Accelerate to top speed in 1st gear, turn all the way left then press handbrake, repeat 5 times to calculate average points	Get a consistent amount of points, within +/-10% of each other	
2	Accelerate to top speed in 1st gear, turn all the way left then press handbrake, repeat 5 times to calculate average points		
3	Just drift around until getting 100,000 points	Managed to get 100,000 points	
Comments:			

Test case ID: 5		Purpose: Check for memory leaks	
Steps: 1		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Measure the memory usage, run the game for 10 minutes, measure the memory usage again	The memory usage at the end should be no bigger than 15% than at the start	
Comments:			

Test case ID: 6		Purpose: Check going between menus	
Steps: 3		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Go between to game and back to main menu	Menu menu should look and function exactly the same	
2	Go to tuning menu and back to main menu	Main menu should look and function exactly the same	
3	Go to game and then back to main menu and then tuning menu	Tuning menu should look exactly the same	
Comments:			

Test case ID: 7		Purpose: Check if tuning settings work	
Steps: 2		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Go to tuning menu and adjust settings drastically...	The numbers display correctly and sliders slide properly	
2	...and then back to main menu and then to game	The car is affected by the tuning settings	
Comments:			

Test case ID: 8		Purpose: Clutch	
Steps: 2		Preconditions: None	
Step No:	Procedure	Expected response	Pass/ Fail
1	Rev up the engine to 7500 RPM with car in gear and clutch disengaged	The car does not move	
2	Engage the clutch and let off the throttle	The car accelerates quickly to a speed of roughly 15 km/h +- 10 km/h	
Comments:			

Appendix B - glossary of physics terms

Subscript c denotes variable in terms of car direction

Subscript w denotes variable in terms of world coordinates

Subscript f denotes front axle

Subscript r denotes rear axle

dt = Delta time

RPM_e = Engines RPM

v_c = Cars velocity

r_w = Wheel radius

GR = Gear ratio \cdot (current gear ration \cdot differential ratio)

Rad/s to RPM = $60 \div (2 \cdot \pi)$

RPM to Rad/s = $(2 \cdot \pi) \div 60$

τ_w = Torque at wheels

τ_e = Torque at engine

t = Throttle position

fb = Foot brake input

F_{fb} = Max foot brake force

hb = Hand brake input

F_{hb} = Max hand brake force

F_{tb} = Total braking force

F_w = Force at the wheels

F_t = Total force on the car

RR = Rolling resistance coefficient

Cd = Coefficient of drag

Af = Frontal area of the car

ρ_a = Air density

AR = Air resistance coefficient = $0.5 \cdot Cd \cdot Af \cdot \rho_a$

F_d = Total force of drag = $RR \cdot Vc + AR \cdot Vc^2$

a_c = Car's acceleration

m = Car's mass

θ_c = Car's angle(orientation)

W_w = Weight on a specific wheel while moving

d_x = Distance to a specific axle from centre of gravity

WB = Wheelbase

AT = Axle track

g = gravity

CG = centre of gravity height

WD = percentage weight distribution

YS = Yaw speed

ω = Angular velocity

S = Slip angle

θ_s = Steer angle

SS = Side slip

CS = Cornering stiffness

$G_t =$ Grip of a tire

$F_{ff} =$ Friction force of front axle

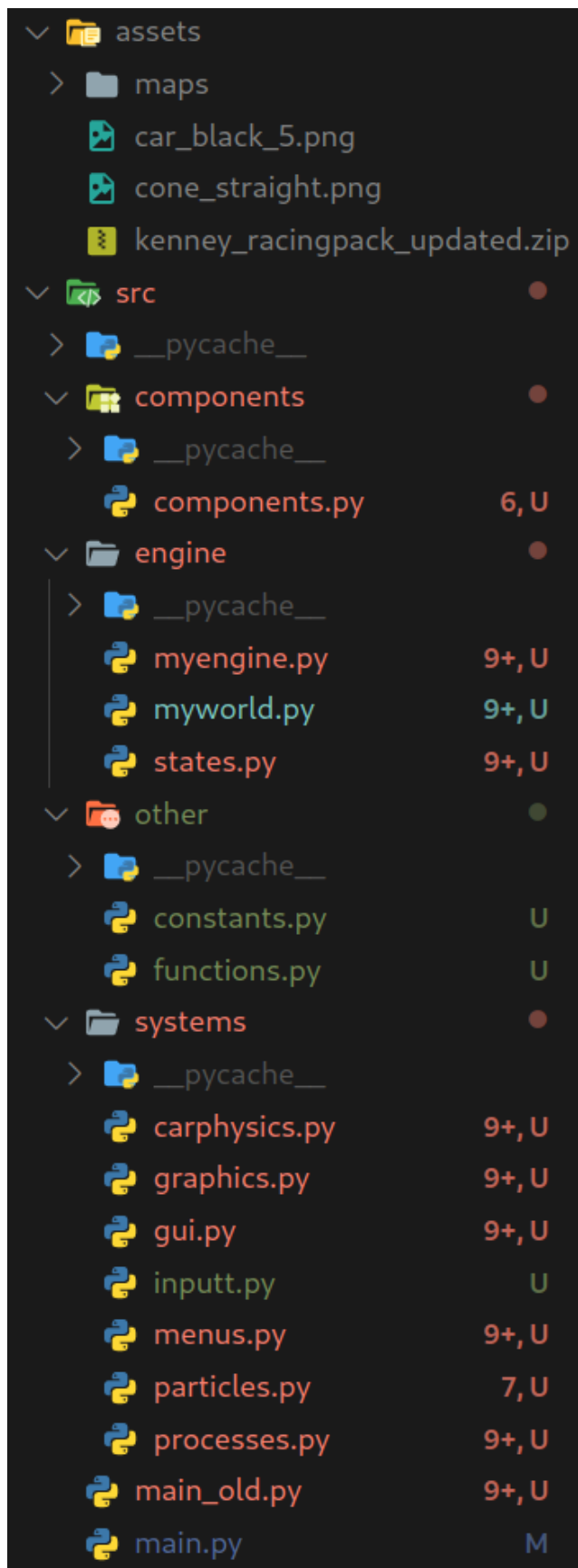
$F_{fr} =$ Friction force of rear axle

$\tau_a =$ Angular torque

$I =$ Inertia

$\alpha =$ Angular acceleration

Appendix C - file structure



References

- Adam. (2007). *Entity Systems are the future of MMOG development*. T-machine. Retrieved 05 14, 2021, from <http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/>
- The British Drift Championship. (n.d.). *Just How IS Drifting Scored?* Retrieved 05 14, 2021, from <https://www.thebritishdriftchampionship.co.uk/just-how-is-drifting-scored/>
- CarX Technologies, LLC. (2017). *CarX Drift Racing Online*. Retrieved 05 14, 2021, from <https://carx-online.com/>
- Cho, J. R., Choi, J.-H., Lee, H., Woo, J.-S., & Yoo, W.-S. (2007). *Braking Distance Prediction by Hydroplaning Analysis of 3-D Patterned Tire Model*. ResearchGate. Retrieved 05 14, 2021, from https://www.researchgate.net/publication/245525713_Braking_Distance_Prediction_by_Hydroplaning_Analysis_of_3-D_Patterned_Tire_Model
- DD Performance Research LLC. (2013). *FRS & BRZ Dyno Compilation dyno testing results*. DD Performance Research LLC. <https://www.ddperformanceresearch.com/forum/viewtopic.php?f=2&t=95>

Dennis Jarvis (Director). (1995). *Jeremy Clarkson's Motorworld* (Season 1, Episode 1) [TV series episode].

https://www.youtube.com/watch?v=3_loR6nm4dU

Descartes, R. (1637). *Discourse on the Method*.

eran0004. (2015-2019). *Programming a driving simulator, how hard can it be?*

GTPlanet. Retrieved 05 14, 2021, from

<https://www.gtplanet.net/forum/threads/programming-a-driving-simulator-how-hard-can-it-be.324556/>

Funselektor Labs Inc. (2015). *Absolute Drift*. Retrieved 05 14, 2021, from

<http://absolutedrift.com/>

GTChannel, Tsuchiya, K., & de Vera, R.J. (2010). *Drifting 101 featuring the Drift*

King Keiichi Tsuchiya - GTChannel. YouTube. Retrieved 05 14, 2021, from

<https://www.youtube.com/watch?v=eLDejTKH388>

Kavanagh, J. (2012). *2013 Scion FR-S: Dyno Tested*. Edmunds.

<https://www.edmunds.com/car-reviews/track-tests/2013-scion-fr-s-dyno-tested.html>

Kenney. (2019). *Racing Pack*. Kenney. Retrieved 05 14, 2021, from

<https://www.kenney.nl/assets/racing-pack>

Lear, S. (2021). *Drifting and Its Rapid Growth in Popularity*. Retrieved 05 14,

2021, from

<https://grassrootsmotorsports.com/articles/drifting-and-its-rapid-growth-popularity/>

Monster, M. (2003). *Car Physics for Games*. Retrieved 05 14, 2021, from <https://asawicki.info/Mirror/Car%20Physics%20for%20Games/Car%20Physics%20for%20Games.html>

Newton, I. (1736). *Method of Fluxions*.

PyGame. (2021). *PyGame documentation*. PyGame. Retrieved 05 14, 2021, from <https://www.pygame.org/docs/>

Tsuchiya, K. (n.d.). *Keiichi Tsuchiya - Toyota AE86 Drift lessons*. YouTube. Retrieved 05 14, 2021, from https://www.youtube.com/watch?v=S_CY5KIFV4I

Bibliography

"DaFluffyPotato", C. (2018-2021). *PyGame Tutorial Series*. YouTube. Retrieved 05 14, 2021, from <https://www.youtube.com/watch?v=xxRhvyZXd8I&list=PLX5fBCkxJmm1fPSqgn9gyR3qih8yYLvMj>

Best Motoring. (n.d.). *Best Motoring videos*. YouTube. Retrieved 05 14, 2021, from <https://www.youtube.com/channel/UCrmsruqPgCIs4PImiZs0t9w>

Donut Media. (2018-2019). *Science Garage*. Retrieved 14, 05, from

<https://www.youtube.com/playlist?list=PLFI907chpCa7R39I29VY5DygA8RgBQZBf>