Vehicle Physics Simulation

Lewis Bowes

September 2017

Contents

Part I

Analysis

Introduction

1.1 Background

Physical simulation of vehicles is an area that continues to receive a lot of attention in both engineering and the gaming industry. A virtual model of a car allows manufacturers to predict the behaviour of their designs in a safe and reusable environment before starting production [1]. Potential problems can be detected and solved before they become an issue, making the design process more efficient and reliable. From the point of view of the gaming industry, vehicles that respond realistically to player input significantly increase the immersion of games. With an increase in available computing power, long-running titles such as "Need for Speed" (Fig. 1) have developed dramatically over the past couple of decades [2]. Other popular racing games at the moment – for example "iRacing" [3] – use real world vehicle data and precise measurements (mass, tyre parameters, aerodynamic properties etc.) to bring its vehicles' behaviour as close as possible to real life. "BeamNG Drive" [4], another example, uses soft-body simulation to provide destruction physics with very high realism. As seen in [4], BeamNG is remarkably similar to real life given the complexity of the scenarios being simulated.

Figure 1.1: The original "Need for Speed" released in 1994

1.2 Project aim

For this project, I will be making a vehicle physics simulation. The simulation will be in the form of an open-world sandbox, in which the user is free to drive around. While focusing on vehicle physics, I will also

approach this problem from the point of view of a game developer and will aim to produce software that is fun to use, rather than a scientific simulation, that would function as a design tool (Fig 1.2). This perspective presents important considerations that I will keep in mind throughout the design and implementation phases of the project. Firstly, it should be possible to measure the success of the project by its entertainment value. This means that the physics simulation involved does not need to be accurate to a level that would be considered reliable in scientific analysis. Secondly, the the input to the program must be intuitive and easy to use. The user should not be limited by hard-to-use controls or UI. A 'help' window could be added to make program usage easier.

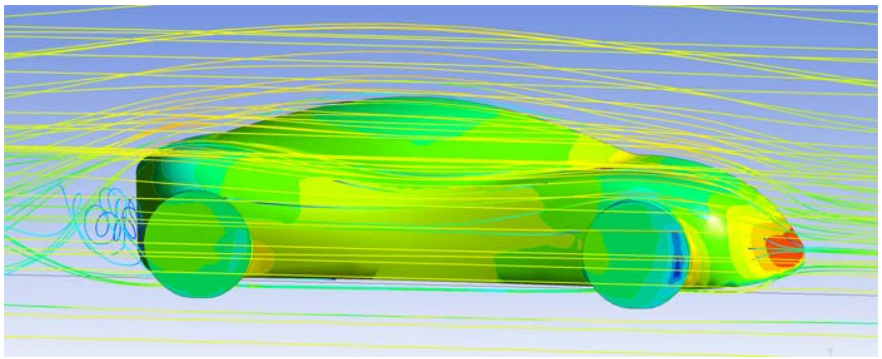


Figure 1.2: [5] The visual output of software used for aerodynamic analysis performed on vehicle models to predict flow behaviour/state around the vehicle and values like the coefficient of drag.

However, I think increasing the accuracy and predictability of the physics simulation (up to a point), *will* increase the overall quality of the application and make it more enjoyable to use. One factor that is known to make video games more fun, is the sense of control that the player is given. This can be attributed in part to immersion: the player can see that their actions have a direct and immediate effect on the outcome of the game, and this outcome is predictable. It could be argued that this is most prevalent in racing games, where the player is continuously interacting with the game (as opposed to turn based games, for example). Immersion and predictability will come with increased realism and as a result, I will aim to make the vehicle physics in my game as realistic as possible, within the time constraints.

It would be possible to dedicate multiple projects to one specific area of vehicle physics simulation, such as mass efficient design, structural simulation, aerodynamic simulation (shown above), tyre-road interaction modelling or performance analysis etc, due to the amount and depth of information available [6]. My project will instead aim to provide a more holistic simulation of a vehicle, covering core elements with simplified physical models.

Research

2.1 Description

The research for this project so far has taken place along with prototyping work. A significant amount of research has involved finding equations needed for simulation of specific components of vehicles. The main equations are outlined in the 'Equations' section below. Most of this research has been in the form of:

- **Research papers**
  For example, [6].
- **Websites**
  The vehicle dynamics Wikipedia page [7] has provided a very useful base on which to expand and research more specific detailed topics. It contains categorised links to these sub-topics, and gives a broad overview of the field. In addition to Wikipedia pages, other websites and articles contain more detailed descriptions of certain areas, such as [8] and the Pacejka magic formula.
- **YouTube videos**
  I have used YouTube videos for explanations of topics. Specifically, the channel "Engineering Explained" [9] has provided high quality descriptions of many subjects like tyre behaviour [10]. In addition to this, YouTube has been helpful for sourcing video of real life car behaviour in a variety of (less common) scenarios, such as drifting/emergency stops. I can use these videos to test the results of my simulation.

Pacejka Magic Formula

During the initial research conducted for this project, I was introduced to new concepts and explanations for concepts I knew existed but did not understand fully. An example of a new topic was the Pacejka Magic Formula for tyre dynamics [11]. Hans B. Pacejka (1934-2017) developed tyre models that simulate the interaction between a tyre and the surface it is moving on. Specifically, his formulae calculate:

- the longitudinal force produced by the tyre (along the direction that the tyre is facing)
- the lateral force produced by the tyre (perpendicular to the tyre's direction)
- the aligning moment of the tyre (a twisting force that rotates the tyre back along its natural direction of travel)

given the:

- Longitudinal slip (ratio) - The longitudinal slip ratio is defined as the ratio between the speed of a point at the contact patch of the tyre and the road (accounting for tyre angular velocity and the velocity of the centre of the tyre relative to the road). It is slip that generates the force for the tyre - without it, cars would not move.

- Lateral slip (angle) - Lateral force, that occurs during cornering, acts perpendicular to the tyres forward axis. It is dependent on lateral slip, which is defined as the angle between the velocity of the centre of the tyre and it's forward axis. While drifting, where the tyre is not aligned with the direction of travel, this angle is greater, but when driving in a straight line this is 0.
- Camber angle of the tyre - This can be described as the angle by which the tyre is tilted away from its upright position. This is positive if the top of the wheel is tilted out, away from the chassis of the car ('toe out') and negative if it tilts into the chassis of the car ('toe in').
- Vertical load on the tyre.

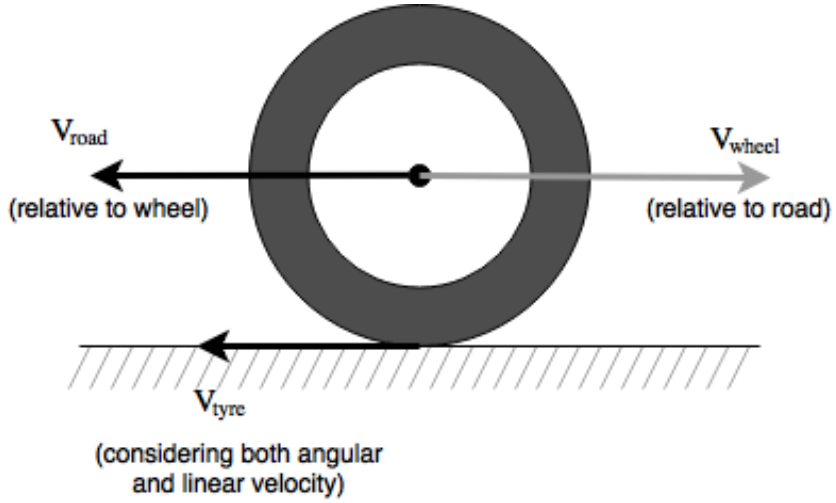


Figure 2.1: The car is braking in this image, as the velocity of the tyre's contact patch is less than the velocity of the centre of the wheel

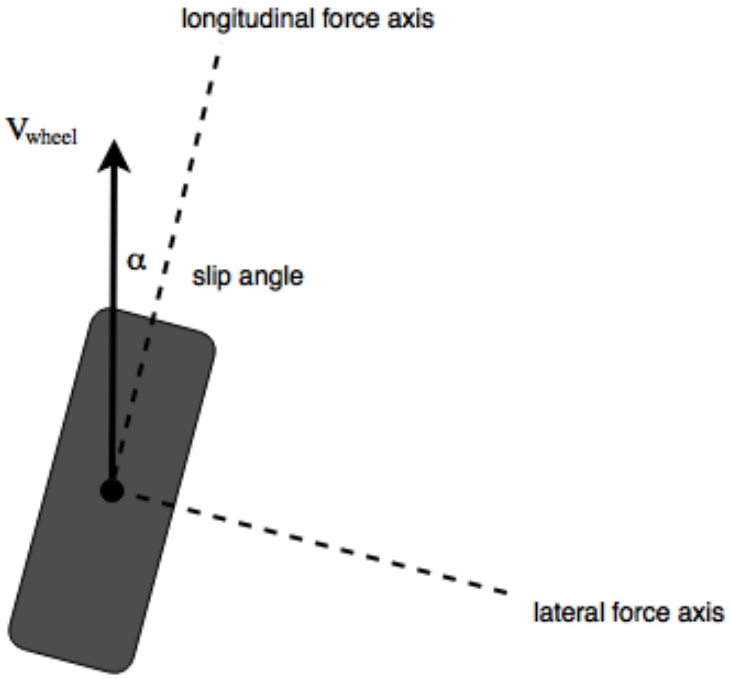


Figure 2.2: A view of a tyre from above, showing the direction in which forces are generated as well as the slip angle.

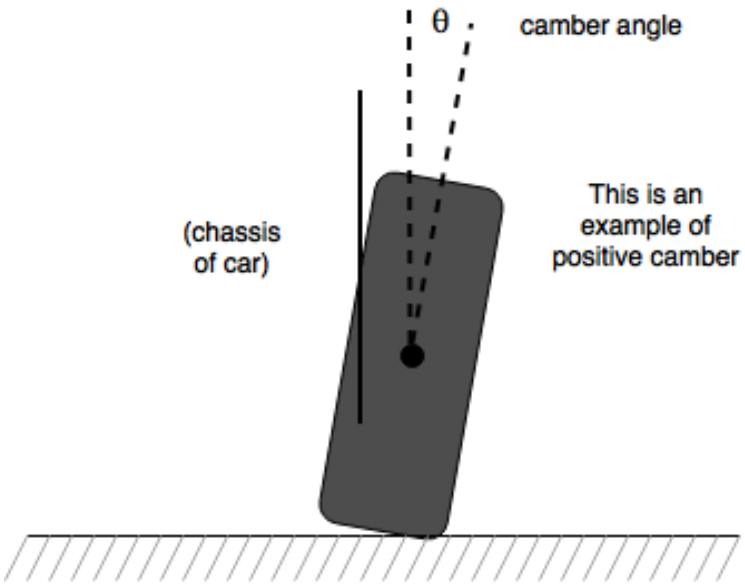


Figure 2.3: This illustrates the camber angle of a tyre.

This forumla is described as 'magic' because it uses a series of non-dimensional parameters. These parameters do not represent any physical quantities, but have been developed through testing to predict tyre behaviour to high accuracy. Different tyres have unique parameters.

So with these values as input, the Pacejka Magic Formula can be considered as a force generator. If the correct values for slip and load can be calculated and passed into a function that computes this formula, then the tyre forces should produce realistic movement of the car.

Equations

From preliminary online research I have collated a list of some of the main equations that will be used within the simulation of the car. The force-based physics section in my game-framework library will be used to support the car's motion through space, with a single total force vector (in world space) and single total torque vector (also in world space) that will be recalculated each update. If this can be passed to the library, it can handle the integration and state updates of the rigid body.

1.

$$\boxed{slip = \frac{wr - v}{v}}$$

Where:

$slip$ = slip ratio

r = wheel radius at tyre contact patch (m)

w = the angular velocity of the wheel along its lateral axis (ms^{-1})

v = the speed of the centre of the wheel (ms^{-1})

The (longitudinal) slip ratio can be calculated using this equation.

2.

$$\boxed{\alpha = -\arctan\left(\frac{V_y}{|V_x|}\right)}$$

Where:

α = slip angle $(degrees)$

V = wheel velocity relative to road surface (ms^{-1})

V_x = longitudinal component of V (ms^{-1})

V_y = lateral component of V (ms^{-1})

This equation is used to calculate the slip angle of the tyre.

3.

$$\boxed{F = -K(L - L_0) - C_{damp} \cdot v}$$

Where:

F = force (N)

k = spring constant (Nm^{-1})

L = current spring length (m)

L_0 = resting spring length (m)

C_{damp} = coefficient of damping $(dimensionless)$

v = compression rate (ms^{-1})

This equation is used to calculate the force generated by a spring. It will be useful for suspension calculations later in the project.

4.

$$\boxed{F_{long} = F_{braking} + F_{drag} + F_{rr}}$$

Where:

F_{long} = the total longitudinal force (N)

$F_{braking}$ = force generated by the tyres (N)

F_{drag} = aerodynamic drag force (N)

F_{rr} = rolling resistance force (N)

The total force acting on the vehicle can be broken down into separate forces.

5.

$$\boxed{F_{rr} = C_{rr} \cdot N}$$

Where:

F_{rr} = rolling resistance force (N)

C_{rr} = coefficient of rolling resistance $(dimensionless)$

N = normal force on tyre (load) (N)

6.

$$\boxed{\Delta N = a \cdot \frac{h}{L} \cdot m}$$

Where:

ΔN = change in load (N)

a = a component of the car's acceleration vector in its local frame (ms^{-2})

h = height of the car's centre of mass above the terrain (m)

L = the distance between the two load transfer locations (m)

m = mass (kg)

This general equation is used to calculate a transfer of load. It will be used for calculation of longitudinal and lateral load transfer. The correct values for a and L must be used for each. The longitudinal version of this equation will use the longitudinal acceleration of the car as a (in its local reference frame) and the wheelbase for L whereas the lateral equivalent will use the lateral acceleration and wheel track. The result is given in Newtons and can be either positive or negative. As this is a transfer of load, $\Delta N_{Rear} = -\Delta N_{Front}$ and vice versa.

7.

$$\boxed{\delta_{max} = \arcsin\left(\frac{L}{\sqrt{(R - T_{front})^2 + L^2}}\right)}$$

Where:

δ_{max} = maximum absolute steering angle $(degrees)$

L = wheelbase of car (m)

T_{front} = axle track (m)

R = minimum turning radius (curb-to-curb) (m)

Accounting for Ackermann steering geometry, I have derived this equation to calculate the maximum absolute steering (or 'deflection') angle for each of the front wheels. It is positive and negative for right and left respectively. During a full-lock turn, the absolute steering angle of the inner tyre will be δ_{max} (that of the outer tyre will be slightly different).

8.

$$\boxed{\tau = r \cdot F}$$

Where:

τ = torque (Nm)

F = force (N)

r = perpendicular displacement from rotation point/axis (m)

This equation will be used when calculating the torques generated on/within the car. For example, the torque on the car produced by the tyres can be calculated with the total tyre force as F and the distance of the tyre contact patch to the centre of mass as r.

9.

$$\boxed{F_{drag} = \frac{1}{2} \cdot \rho \cdot A \cdot C_d \cdot V^2}$$

Where:

F_{drag} = aerodynamic drag force (N)

ρ = air density (kgm^{-3})

A = area of the object facing the air flow (m^{-2})

C_d = coefficient of drag $(dimensionless)$

V = velocity of object (ms^{-1})

The aerodynamic drag force.

2.2 End users

The end users of my project will be friends in my year, specifically Callum Thompson, Harrison Holgate and Oliver Sheldon. They all play PC games so they will be able to provide feedback and suggestions for the sandbox, to improve both immersion and entertainment value. As well as requirements gathering in the form of discussions and group pooling of ideas (given the starting points of 'vehicle physics' and 'racing game') I asked my friends the following questions. All responses are outlined below for each section:

Q: "What features come to mind when someone says 'racing game'? What do you associate with this type of software?"
A:

- A car model
- Drifting car behaviour
- A timing system such as recorded time trials to try to beat
- Multiple camera views - a cinematic camera
- Control guides such as direction arrows, throttle hints like red/amber/green arrows along the track.
- Live data about the car to view while you're driving.
- The ability to choose different cars or customise and save ones that you design (a "garage" for adding updates to your vehicle)
- Controller support
- Virtual (AI) opponents to race against
- A track to drive around
- A damage system and sparks
- Tire "squealing" sound effects

Developing the suggestion of customisability, the following question was used to isolate areas that my end users most wanted to be able to modify. This included features not directly related to the car. To make the program more entertaining, I will make customisability one of its core features. This will add variation to the program. By tuning the parameters of their vehicle (changes to the suspension spring stiffness damping, car mass, tyre parameters etc are all options to be considered) the user is given the freedom to experiment and change the behaviour of their car.

Q: "Given the option to customise features of the simulation, which features would you most like to be able to adjust/tweak/experiment with?"
A:

- The speed of the simulation.
- Properties of the car like engine type, mass or aerodynamic parameters. This could be done in a "car creator" or "garage" similar to existing racing games
- Basic environmental properties like visbility (fog density), time of day and weather conditions
- Terrain properties such as shape
- The colour of the car

Q: "What aspects of vehicle physics simulation do you think are most significant?"
A:

- Acceleration
- Braking
- Steering
- Collision
- Suspension
- An environment

Q: "Are there any specific new features that you would like this game to have that others don't?"
A:

- A low-poly graphical style
- Randomly generated environments
- Live data output from the car
- A day night cycle
- A boot and doors that can be opened and closed

Q: "Are there any other features that you think could increase immersion while driving?"
A:

- A player model
- The ability to look around inside the car
- VR support
- Head bobbing camera while the car moves

Based on the discussions I have had with my end users and their responses to the questions above, I have formed the objectives for the project. Given the time constraints, one feature suggested above that I *know* will not make it into the final application is VR support. I do not have a VR headset (or computer that is powerful enough to support one) to use for debugging the application, and MFL does not support binocular rendering.

The remaining feature requests have been grouped to make up the core objectives within the project. They have been decided based on a combination of popularity, significance and implementation time (in that order). Implementation time has been estimated based on the complexity of the feature and whether or not MFL provides any important code needed to implement it. Remaining features will be added based on the same criteria, if time permits.

Objectives

Objective 1

Title:	"Simulate the motion of a 4-wheeled vehicle with 6 degrees of freedom in motion (in real time), under the influence of multiple forces"
Outline:	This objective simply includes the realistic movement of the car. The simulation should involve the calculation of at least the following forces and resulting torques: 1. Weight 2. Aerodynamic drag 3. Normal reaction 4. Tyre traction (both longitudinal and lateral)
Time:	I think this objective will take the longest to complete, because it will involve the most programming. Structuring and implementing a layered hierarchy of subsystems and components will occur continuously throughout the project as more internal areas of the simulation are added (within the car). The structure may need to be modified to support new features or make it easier to navigate and maintain code. This objective will take a number of months.

Objective 2

Title:	"Display the output of the simulation in real time, in a graphical application with a 3D rendered scene"
Outline:	The rendered models of the car should be updated and change position on screen as the program is running. This objective will be met as long as each of the following objects are rendered: 1. A main car chassis model 2. 4 wheels 3. 4 suspension springs 4. A steering wheel 5. A terrain model 6. A skybox
Time:	Similar to objective 1, this will be completed throughout the project and could also take months. The graphical representation of the simulation will be added on top of its internal output, so the car model for example can only be accurately rendered when the cars physics simulation is completed. If this limits progress on this objective, I can work on UI design or creating models while debugging the physics code and return to simulation output when it's fixed.

Objective 3

Title:	"The graphical output of the application should have a low-poly style"
Outline:	The models for the car and the terrain should be made with a low number of triangles to match a low-poly game appearance.
Time:	Low-poly models can take less time to create that those with more polygons. The modelling of the car chassis might take a few hours. Once the terrain's internal data has been generated, the algorithm for converting it to a 3D model might require around a few days of work.

Objective 4

Title:	"Allow the 3D rendered models to be viewed from different perspectives with multiple cameras, as the simulation is running"
Outline:	It should be possible to toggle the currently active camera using the 'C' key on the keyboard. This objective will be met as long as different cameras provide unique views. The following cameras will be implemented: 1. A driver camera from the view of the driver of the car. The mouse will be used to control the direction in which the camera is looking. This must be updated as the user moves the mouse, if not the objective will not be met. 2. A camera that is fixed (relative to the car) on the front right wheel. This camera will not move. 3. A free camera that can move in 3D around the environment to allow the user to select their own camera angles. It will be possible to change the direction of movement using the mouse and move using the keyboard (a possible input key mapping is W, A, S, D, SPACE, SHIFT for forward, left, right, backwards, up, down respectively).
Time:	Depending on the type of software, an in-game camera may logically overlap multiple areas. Fundamentally, they are used to render a scene from different perspectives and are therefore closely tied to the graphical side of an application. In the case of an FPS ("first person shooter") camera that is controlled by the player, movement code will be required. This movement control will require input handling, and toggling between multiple cameras will also be done using user input. Cameras that follow/track/orbit an object may need access to the object's state (e.g position). Due to the fact that cameras may be linked to many areas, this objective might be quite time consuming. In addition to this, my end users may want to tweak camera positions/movement speeds and input key mappings etc. and this will also take time. This objective could take a few weeks, or just over a month.

Objective 5

Title:	"Allow the user to control the motion of the car using the keyboard and mouse"
Outline:	The keyboard will be used to adjust the internal state of the car. This adjustment will result in changes to the magnitude and direction of the friction force applied to the car's wheels. This will include: 1. The steering wheel - direction 2. The accelerator pedal depression - magnitude 3. The brake pedal depression - magnitude The mouse will also be used to navigate the menu system.
Time:	The implementation of this objective is dependent on the internal simulation code, because the car must have an interface to its components in order for them to be controlled. However, once this has been completed, linking these components to the user with key input mapping should be relatively simple. I estimate that this will take a few days, perhaps a week. If the progress of this objective is limited by simulation code, I could spend time working on creating models and testing UI (similar to objective 2) in order to continue making progress.

Objective 6

Title:	"Procedurally generate and store the heights and surface normal vectors of a large, square terrain heightfield"
Outline:	The data for the terrain should be generated during the loading period of the application. The 3D graphical model of the terrain should also be generated here, but will be displayed at run-time. It will be constant size and must not be completely flat. Ideally, it should not contain visible repeating features. The size of the terrain must be large enough to give the user a sense of freedom (around 256 x 256 should be sufficient, giving the user 65536 square metres to move around in).
Time:	This will be a time consuming task. I will have to decide on: • The format in which to store the terrain data • A method of generating the terrain • How to handle edge cases with sampling out-of-bounds data • Specifically how to calculate the normal vectors of a terrain heightfield The shape and style of the terrain could be subject to multiple changes, if the end user decides to tweak it. I will need to design a modular method of generating the terrain if I am to add both a track and rough, off-road, surface. Once this is completed, time will need to be taken to test the interaction between the car and the terrain. Overall, I think that this objective might take between one/two months, or just over.

Objective 7

Title:	"Imprint a track shape into the terrain data"
Outline:	The terrain data should contain a noticeable track feature, that the player can (optionally) follow while driving. It must be a smooth continuous loop that is wide enough to allow one car to travel around at speed. The track shape should contain both straight and rounded sections.
Time:	As mentioned in the previous objective, the track generation will be subject to small adjustment and modifications. The task of creating an algorithm to generate a track will also be relatively complex. This objective might take a few weeks.

Objective 8

Title:	"Display simulation settings and output with a user interface, that can be navigated with the mouse"
Outline:	I will use the windows created by ImGui [12] to house simulation settings e.g time speed and the physical state of the car (such as position, velocity, acceleration etc). This library already provides many options for the customisability of the appearance and functionality of the windows and widgets it creates. It has been developed as an 'immediate mode' GUI, meaning that UI objects can be created and updated anywhere within the rendering cycle, by calling ImGui functions. Development time should be reduced as a result.
Time:	This objective should take less time to complete than others, because ImGui handles handles window creation, rendering and input. The UI code will need to be integrated into the rest of the graphical side of the application, on top of the model layer. It will need to have access to simulation data to display. The relationship between the UI and the internal simulation code will take time to decide. This should ideally take less than 2 weeks.

Part II

Design

PREAMBLE:

- Throughout the Design section, occurrences of identifiers and small, inline code snippets will be **highlighted in blue** . These directly correspond to the identifiers that will be used in the source code. See chapter 3 "Implementation" for the complete project source code.
- Code identifiers that begin with 'm' are class member variables.
- Coordinate reference frames in code are indicated with 'variable_frame', so force_world is a force in world-space etc.
- Coordinate reference frames in diagrams are indicated with the orientation of the principle axes (x, y, z) relative to the base (world) reference frame.
- The 'default' reference frame uses y as the vertical axis, x as right and $-z$ as forward, to match the coordinate frame used by OpenGL and make visualising/debugging easier.

3.1 External Dependencies

For this project, I will be using a library that I have been developing since July 2016. It provides a very basic game-like application framework to speed up the process of creating applications (most suited to games) using modern OpenGL and C++. For the rest of the documentation I will be referring to this library as "MFL" ("My Framework Library").

"Framework"

These are the external dependencies used within MFL, and thus prerequisites for any project that will use it (along with Framework.lib itself).

Internally, MFL requires the following third-party static libraries:

- opengl32.lib – core OpenGL library itself [13]
- glew32s.lib – "extension wrangler" for OpenGL [14]
- glfw3.lib – window creation, event and input handling [15]
- SOIL.lib – "Simple OpenGL Image Loader" is used for OpenGL texture creation [16]

In addition to static libraries, MFL uses the following header-only libraries:

- Dear ImGui – an immediate-mode style GUI library that can be integrated into an existing OpenGL rendering pipeline [12]
- GLM – "OpenGL Mathematics" is a maths based around the maths involved in graphics programming [17]

MFL provides abstractions around some of the basic functionality needed within OpenGL applications. The most important of which are outlined below:

1. *An abstract base class to enforce the structure of the application* - The user creates their own application 'root' class that inherits from the abstract base class, **Application** . They then provide the implementation for 4 pure virtual member functions:
   - **virtual void onLoad() = 0**
   - **virtual void onInputCheck() = 0**

- **virtual void onUpdate() = 0**
- **virtual void onRender() = 0**

Once these functions have all been implemented, the library calls them at appropriately, handling the main game loop of the application. The implementations of these 4 functions will make use of other applications

2. *OpenGL context creation and initialisation* - MFL creates an OpenGL context window that is ready to render whatever data is presented to it later on by a **Renderer** in the **onRender()** method.
3. *Keyboard, mouse and controller input handling* - MFL provides a basic interface for checking input device states, enabled internally by GLFW. Individual key states and/or buttons can be queried along with the mouse position using the functions found in **Input** class.
4. *A resource management system* - The **Application** class contains a single **ResourceSet** instance by default, but the user of the library can create their own anywhere to store their own groups of resources. This class is used for storing any objects in the Framework derived from **Resource**, many of which are listed in the following paragraph.
5. *Abstractions around OpenGL objects* - MFL contains abstractions around OpenGL **Shader**s, multiple **Texture** types, **VertexBuffer**s, **VertexArray**s, **IndexBuffer**s as well as higher level objects like **Model**s, **Mesh**es and a rendering system that combines and handles these lower level components. In my simulation, these will all be used to show the player 3D models of the car they're driving and the environment. They will be able to look around the environment from multiple views using **PerspectiveCamera**s.
6. *GUI* - For the menu system (and debugging) I will be using "ImGui". This provides a simple way to create a wide range of GUI elements for the project. This will be used to create the menu system within the game and help with debugging.
7. *Physics-related code based around rigid-body motion* - I've most recently started working on a physics section of MFL. The **Spring** class was added during the prototyping phase of this project, to later support the addition of suspension.

Evaluation

I have chosen to use MFL over commercially available alternatives because it does not provide anything more than what will be necessary. In addition to this, I've developed it, so the time available for the project can be put directly into researching the vehicle dynamics and visual design of the game rather than trying to learn a new API. Given the time constraints and the resulting complexity of the program, MFL should be sufficient to support my project.

Structure

4.1 High level overview

I will split my the application into two core parts: 'Internal' and 'Visual'.

'Internal' will be used used to describe all code responsible for the logic behind the physics simulation. It will be the back-end of the application. Throughout development, the aim will be to include code in the Internal namespace that could theoretically be run independently from the graphical application. That is, it should be possible to copy the code in this namespace, into e.g. a console application and - with relatively few modifications - still internally run the physics simulation, in real time. By keeping this in mind, I will ensure that the the back-end code remains isolated from the front-end. This will limit coupling between this section and the visual section, which should make both easier to maintain. As a brief summary, this section will include:

- Physics simulation of the **Car**
- Terrain generation (height-field and surface normal data)

The Visual section of the application will be concerned with everything displayed on the screen in the application window. This will include:

- A visual 'shell' wrapped around the output of the Car's physics simulation
  - A model for the Car
  - A model for the Environment
- Multiple cameras used to view these models
- A GUI containing simulation output and controls

Before adding any features, I will ask "Should this be updated per frame or per physics update? Does it increase the complexity of the simulation model or does it change the appearance of the application?".

To allow the visual section of the program to act as a 'shell' on top of the internal simulation, it will store a **reference** to (as opposed to own an instance of) a **Car** object. Each frame, the visual (rendering) code will update the state of its models etc. based on what the reference object's state. It will be an **observer**. A rough outline for the program's structure is shown by the following image:

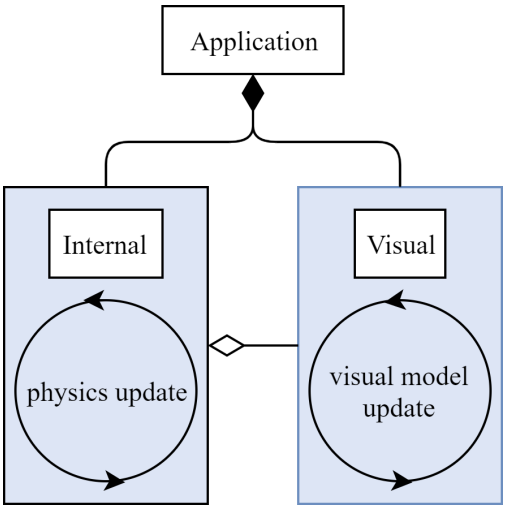


Figure 4.1: Visual stores a reference to the physics simulation output, but this can exist independently from visual.

The internal section of the program can be further split into separate sections: 'Environment' and 'Car'. A static **Environment** class will contain all code that is externally involved in the physics simulation but is not logically part of the car. I will create a separate **Car** class for this code. Both sections above are expanded upon later in the design. The **Environment** will be used during the **Car**'s internal work to update its state. As the **Environment** class will be static, the **Car** does not need to hold or use any reference to it. I will **not update** any state within the **Environment** class as the simulation is running, but terrain data for example, will need to be created at load time. This could be an area for expansion - wind direction and magnitude is an example of an environmental feature that changes over time. The relationship between these two subsections is shown below:

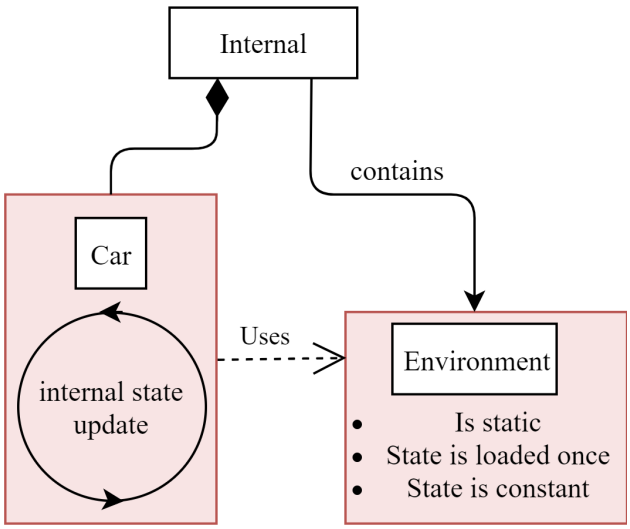


Figure 4.2: Visual stores a reference to the physics simulation output, but this can exist independently from visual.

I will also subdivide the visual section into multiple smaller components. The **Car** and **Environment** will each have a graphical model to represent them. As the **Environment** will not be updated at

runtime, neither will its model. The **CarModel** however, should be updated and have a changing position and orientation, achieved using the world transform matrix that the **Car** class will produce. It should contain multiple **Framework::Graphics::Mesh** es to display the positions of components like the chassis, wheels etc. The visual section must also handle the creation of the user interface. My framework library (internally ImGui) handles automatic updating of windows. Lastly I will create an abstraction around the different simulation cameras that my end user requested. These will be updated using both user input and the **Car** 's current state (i.e. position and orientation). The reference to the **Car** stored by the visual module can be passed down to these components for use. The diagram below illustrates these relationships:

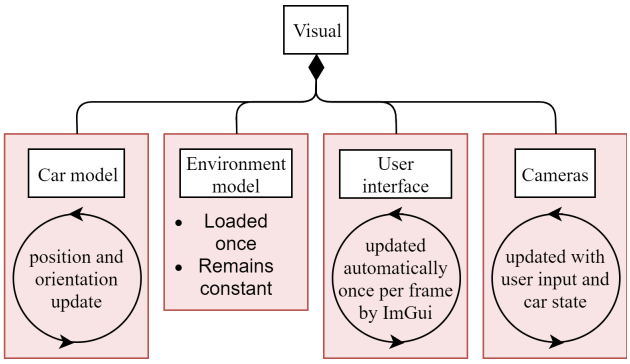


Figure 4.3: Visual stores a reference to the physics simulation output, but this can exist independently from visual.

4.2 Internal

4.2.1 Car

The **Car** class will be the focus of the application. The code responsible for the core elements of the simulation will be connected to this class. As mentioned earlier, I will model this object as a rigid body (meaning linear and angular motion can be calculated separately and combined). The physics section of MFL provides a **RigidBody** base class from which **Car** will inherit. Subclasses of **RigidBody** must implement the following pure virtual functions:

- **glm::dvec3 getForce_world(Framework::Physics::State state, double t)**
- **glm::dvec3 getTorque_world(Framework::Physics::State state, double t)**

One of the primary functions of the **Car** class will be to calculate all the forces and torques acting on the object at any point in time, given the current state of the **RigidBody** and user input. Once these functions have been implemented and return the correct results, the **Car** class must simply call **Framework::RigidBody::integrate(double t, double dt)** once, to advance the state of the simulation by the timestep dt (in seconds). At this point, MFL will take over and update the state of the object at time t to a new state at $t + dt$, using a physics integrator chosen before hand (Euler or Runge-Kutta 4). The state update also includes the calculation of local to world-space transformation matrices for the object. This data can be picked up at a later point by the visual side of the application in order to position and rotate the **Car**'s model correctly in the world.

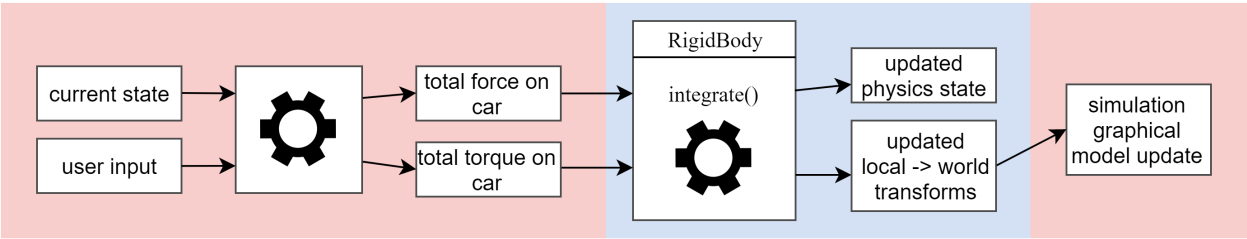


Figure 4.4: The region in blue is handled by MFL and the red areas are to be implemented. The cogs represent examples of significant data processing points. This diagram provides a rough outline of one 'update' operation for the Car object, that will be called continuously in a loop. The **Car** class will be focused on the first red section.

Two forces that can easily be calculated already are gravity, using $F_g = mg$ and $F_{drag} = \frac{1}{2} \cdot \rho \cdot A \cdot C_d \cdot V^2$. I will probably store the gravitational constant value and air density in a separate 'Environment' class. **The remaining forces acting on the vehicle will be generated by the tyres and the suspension**.

Car structure - overview

The real-world model of a car can be intuitively described with subsystems, components and parts etc. Because of this, looking at real-world relationships between components within cars provides a good starting point for the relationships between the corresponding modules of code in the program. For example, the wheels – which could be viewed as 'components' – could be described as having 'parts' such

as tyres, and frames and a connection to an axle. A translation of these relationships into an initial code design might mean that **Wheel** objects contain both **Tyre** and **Frame** objects, and hold some reference to an **Axle**, for example.

This architecture will produce a general tree structure. Lower level classes - leaf nodes - will be owned by classes further up the tree and perform the force calculations. Classes further up the tree will be responsible for supplying the low level classes with required data for their calculations and then provding access to the results, to classes higher up the tree towards the root. The root in this case is the **Car** class, more specifically, the pure virtual functions from the inherited **RigidBody**.

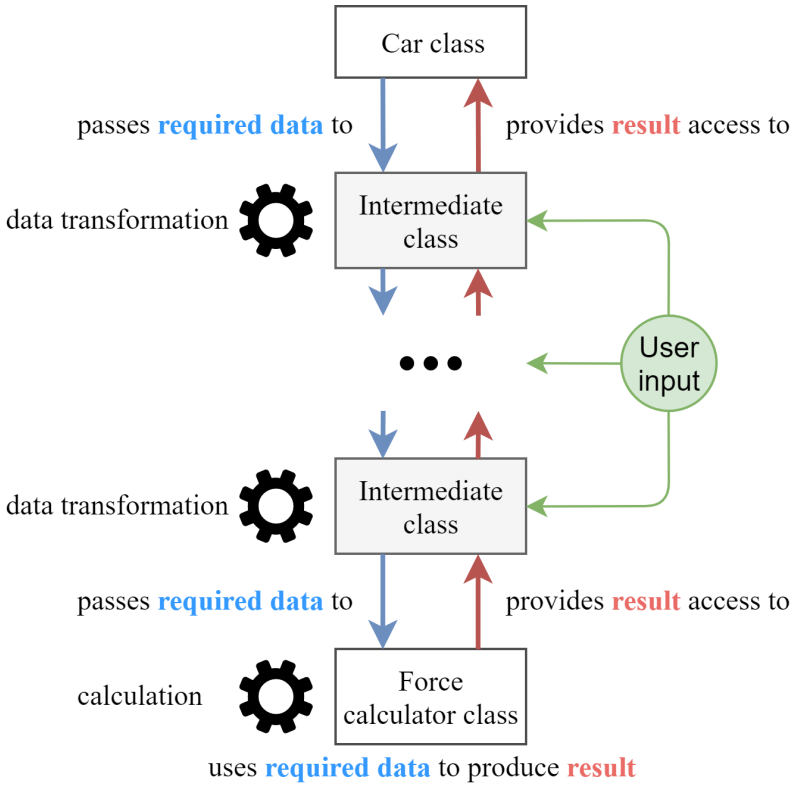


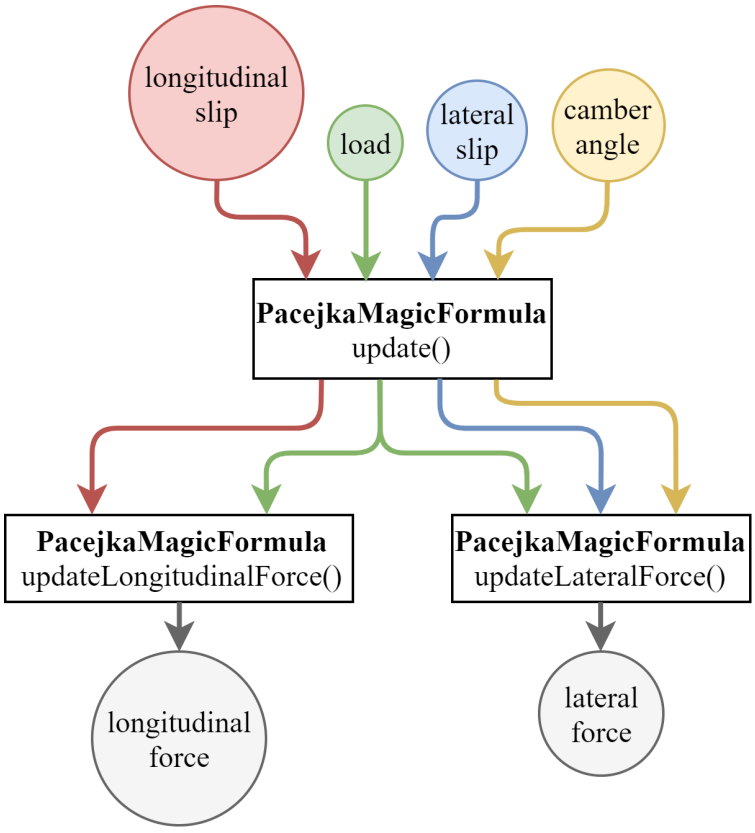
Figure 4.5: One branch of the 'force calculation tree', with the **Car** class as the root. This outlines the way in which data will be produced, transformed and used for calculations.

Pacejka Magic Formula implementation

At the lowest level, the tyre force generation will occur in a **PacejkaMagicFormula** class. A more detailed description of this method can be found in section 2.1. This class should contain:

- All longitudinal force parameters
- All lateral force parameters
- A method of updating both types of force
- A way to access both of these values
- A way of switching tyre types

As input, the **update()** function must take in the vertical load on the tyre, the longitudinal slip, the lateral slip angle in degrees and the camber angle. These values must be given to two separate functions: **updateLongitudinalForce()** and **updateLateralForce()**. [8] provides the implementation for these functions and states that the longitudinal slip must be a percentage (between the values 0 and 100 as opposed to 0.0 and 1.0). The following image illustrates the where they are each sent:



Given the requirements mentioned above, an outline for the **PacejkaMagicFormula** class can be created. The functions **setToRoadTyreParams()** and **setToDriftingTyreParams()** each assign specific values to all tyre parameters. **getLongitudinalForce()** and **getLateralForce()** return the specified force magnitudes. These values will be combined into a total (2D) force vector by a **Tyre** class later on, and then converted into a world-space 3D vector for the final result.

PacejkaMagicFormula
+ [multiple longitudinal params 'b0 -> b13']: float(s) + [multiple lateral params 'a0 -> a17']: float(s) - mLongitudinalForce: double - mLateralForce: double
+ PacejkaMagicFormula(): void + updateForces(): void + setToRoadTyreParams(): void + setToDrifingTyreParams(): void + getLongitudinalForce(): double + getLateralForce(): double - updateLongitudinalForce(double, double): void - updateLateralForce(double, double, double): void

As described by [8], these are the implementations for **update()**, **updateLongitudinalForce()** and **updateLateralForce()** in pseudocode. Variables starting with 'b' and 'a' are the member variable tyre parameters. Since a tyre can only generate force with a normal load and non-zero slip, a check should be made during each of these functions. If either of these conditions is not met, the functions must set their force to 0.0 and return.

```
1: procedure UPDATE(DOUBLE VERTICALLOAD_N, DOUBLE SLIPASPERCENT, DOUBLE SLIPANGLE_DEGS,
   DOUBLE CAMBERANGLE_DEGS)
2:     updateLongitudinalForce(load_kN, slipAsPercent)
3:     updateLateralForce(load_kN, slipAngle_degs, camberAngle_degs)
```

```
1: procedure UPDATELONGITUDINALFORCE(DOUBLE LOAD_KN, DOUBLE SLIPASPERCENT)
2:     if load_kN = 0.0 OR abs(slipAsPercent) = 0.0 then
3:         mLongitudinalForce ← 0.0
4:         return
5:
6:     loadSquared_kN ← load_kN^2
7:
8:     C ← b0
9:     D ← load_kN · (b1 · load_kN + b2)
10:    BCD ← (b0 * loadSquared_kN + b4 · load_kN) · exp(−b5 · load_kN)
11:    B ← BCD/(C * D)
12:    E ← (b6 · loadSquared_kN + b7 · load_kN + b8) · (1.0 − b13 · sign(slipAsPercent + H))
13:    H ← b9 · load_kN + b10
14:    V ← b11 · load_kN + b12
15:    Bx1 ← B · (slipAsPercent + H)
16:
17:    mLongitudinalForce ← D · sin(C · arctan(Bx1 − E · (Bx1 − arctan(Bx1)))) + V
```

```
1: procedure UPDATELATERALFORCE(DOUBLE LOAD_KN, DOUBLE SLIPANGLE_DEGS, DOUBLE CAM-
   BERANGLE_DEGS)
2:     if load_kN = 0.0 OR abs(slipAngle_degs) = 0.0 then
3:         mLateralForce ← 0.0
4:         return
5:
6:     C = a0
7:     D = load_kN · (a1 · load_kN + a2) · (1.0 − a15 · camberAngle_degs^2)
8:     BCD = a3 · sin(arctan(load_kN/a4) · 2.0) · (1.0 − a5 · abs(camberAngle_degs))
9:     B = BCD/(C · D)
10:    E = (a6 · load_kN + a7) · (1.0 − (a16 · camberAngle_degs + a17) · sign(slipAngle_degs + H))
11:    H = a8 · load_kN + a9 + a10 · camberAngle_degs
12:    V = a11 · load_kN + a12 + (a13 · load_kN + a14) · camberAngle_degs · load_kN
13:    Bx1 = B · (slipAngle_degs + H)
14:
15:    mLateralForce = D · sin(C · arctan(Bx1 − E · (Bx1 − arctan(Bx1)))) + V
```

Slip class

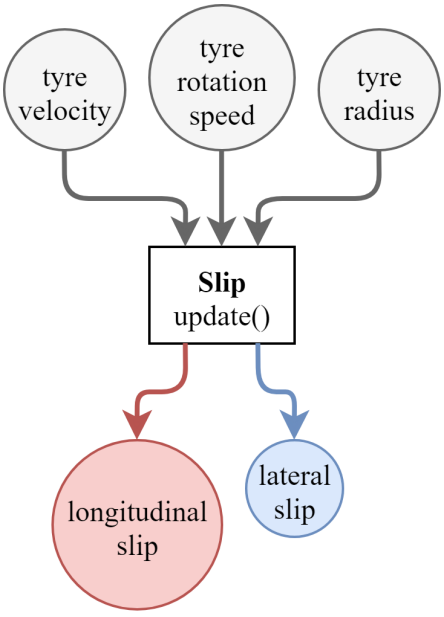
Developing on the requirements of the **PacejkaMagicFormula** class, I will create a separate class called **Slip** that will perform all tyre slip calculations, and have its own data requirements for doing so. This class must contain and update:

- A longitudinal value
- A lateral value (angle)

[6] details how the slip calculation requires:

1. The tyre velocity, given as a 2D vector containing a lateral and a longitudinal component
2. The rotational speed of the tyre in radians per second
3. The radius of the tyre

Using requirements 2 and 3 above, and equations 1 and 2 in section 'Equations', the velocity of the tyre's contact patch can be calculated and the result can be compared to the tyre's velocity vector to produce longitudinal and lateral values. The following diagram illustrates the role of the **Slip** class.

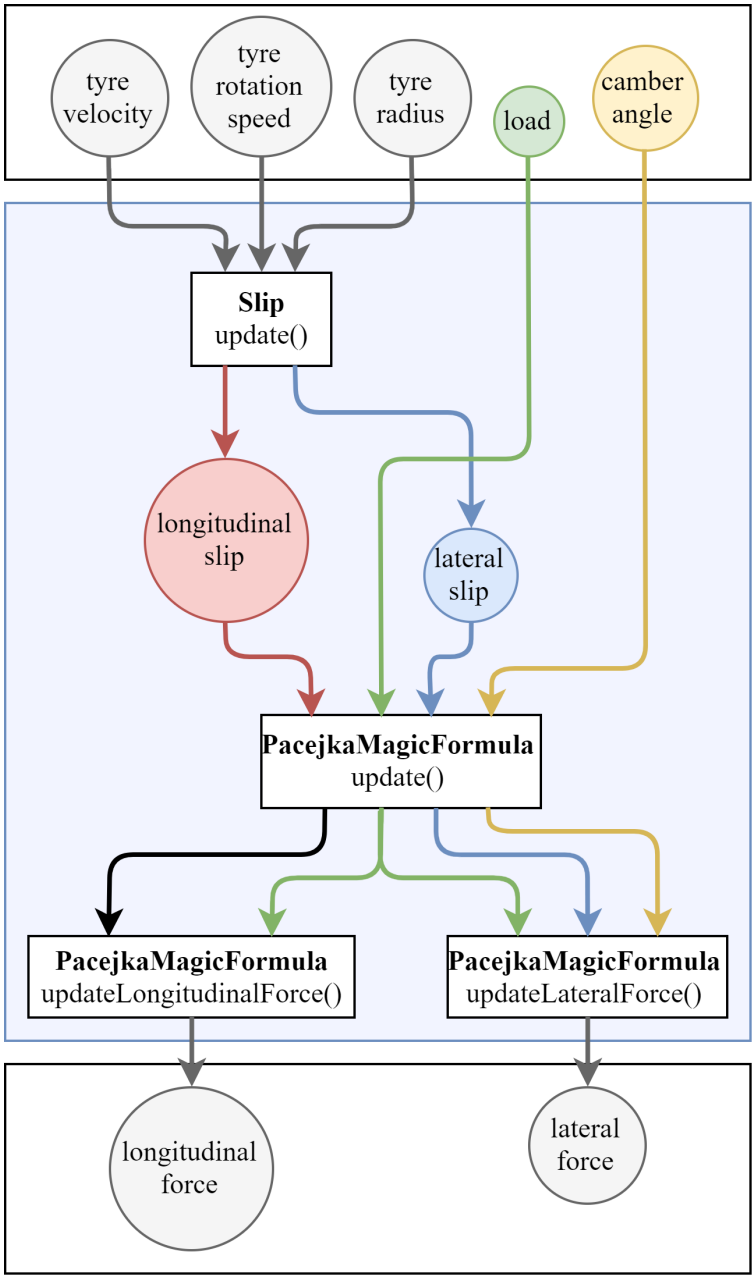


Tyre class

Given that the slip calculation outlined above provides some of the input for the **PacejkaMagicFormula** force calculator, it would make sense to encapsulate both these objects into one class, that updates the **Slip** and passes the results to the **PacejkaMagicFormula**. This class will be the **Tyre**.

As the **Tyre** class will contain a **Slip** instance and a **PacejkaMagicFormula** instance, it must be responsible for providing both with the correct data. The output of the slip calculations can be used (along with other variables that must be passed to the **Tyre** from the next layer up) as input to the force calculations.

This is illustrated in the following diagram, with input at the top and output at the bottom. The blue region represents the code encapsulated within the **Tyre** class.



WheelInterface class

The final vehicle will have 4 **Wheel**s, 4 **Suspension** springs, 4 **Brake**s and 2 **Axle**s. The per-wheel objects could all be contained within the Car class separately as 12 objects, but since there are the same number of each object type, it would be better to create a separate class containing one instance of each and store 4 of them. This will be called a **WheelInterface**. The **WheelInterface** class will be significant in the application, because it will contribute the remaining forces needed for the car to move realistically (after gravity and drag mentioned in 4.1.1).

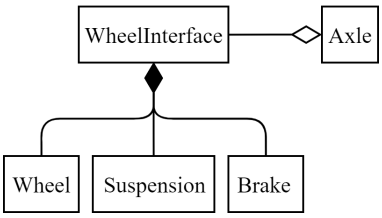


Figure 4.6: This diagram illustrates the relationship between the objects mentioned above.

This class will need to have:

- **Wheel**, **Suspension** and **Brake** object instances
- a reference to an **Axle** object
- an **update()** member function
- a fixed position in car space
- a function to get the total force that it generates

Each call to **update()** must pass the required data to each of these components, calling their update() functions, before totalling up the force that they all generate - a familiar pattern so far.

These components will require some information about the basic physical state of the **Car** object in order to calculate their forces. A few examples that come to mind:

- The **Suspension** will require the velocity of **WheelInterface**'s position on the **Car**, in order to simulate damping.
- The **Wheel** will need to have access to its own velocity in car space, in order to calculate the force generated by the Tyres.
- The **Car**'s local-to-world rotational transform will be needed in order to convert direction vectors from car space to world space and vice versa.

Overall, this class will be more simple than others because it will fundamentally just be passing data along from the **Car** object to individual component objects (**Suspension**, **Wheel**s etc. mentioned earlier). Some data processing might be required in order to get this information into the correct form (spatial conversions), but at a higher level, it will just be passing this data further down the structure. The **WheelInterface** will primarily be created for **abstraction**.

To simplify the calculation of a total force vector, each of these components will calculate their own force directly in world space (given any information they require), so that the **WheelInterface** will not need to have any knowledge of the component's unique a) ways of generating force or b) coordinate spaces. The alternative would be to allow them to each calculate a local force and have the **WheelInterface** class do

all the work needed for world space conversion during the calculation of the total. The first option will reduce coupling the most.

Suspension class

Out of the **Wheel** class and **Suspension** class, the **Suspension** will be easier to implement. Continuing the similar pattern encountered earlier, this object will also have an **update()** member function and will also produce a force in world space. As input to the **update()** function, this class must be given

- The vertical component of the road's velocity, in car space. The road is moving upwards relative to the car.
- A terrain overlap value. Given the total radius of the wheel frame + tyre, the world-space position of a **WheelInterface** and the world-space height of the terrain, this value can be calculated. This will take place in the **WheelInterface** class and the result will be passed into the **Suspension** class.
- A world-space line of action of the **Suspension**. As a simplification, this will be the terrain normal directly under the **WheelInterface**'s world-space position. It would also make sense to also calculate this in the **WheelInterface** class and pass it into the **Suspension**.

> **NOTE:** The frequent use of 'world-space' here, is a demonstration of the point mentioned in the section 'WheelInterface' that individual components will calculate their own forces in world space, rather than have the **WheelInterface** do this.

A **Suspension** object will own a **Framework::Physics::Spring**, implemented in the prototyping phase of this project. The Spring class contains the following member variables:

- mK
- $mDampingCoefficient$
- $mRestLength$

The **Spring** takes in a current length (m) and a compression rate (ms^{-1}) and calculates a force magnitude (N) using $F = -K(L - L_0) - C_{damp} \cdot v$. The resulting force vector is created by normalising the line of action vector passed in, and scaling it by the magnitude. Pseudocode for this calculation can be seen below. A check is made to make sure that if the wheel is not overlapping the terrain, there is no force generated.

```
1: procedure SUSPENSION::UPDATE(DOUBLE VERTICALROADVELOCITY_CAR, DOUBLE TERRAINOVER-
   LAP, VEC3 LINEOFACTION_WORLD)
2:     if terrainOverlap ≠ 0.0 then
3:         mSpring.update(−terrainOverlap, verticalRoadVelocity_car)
4:         mForce_world ← normalize(lineOfAction_world) ∗ mSpring.getForce()
5:     else
6:         mSpring.update(−terrainOverlap, 0.0)
7:         mForce_world ← vec3(0.0)
```

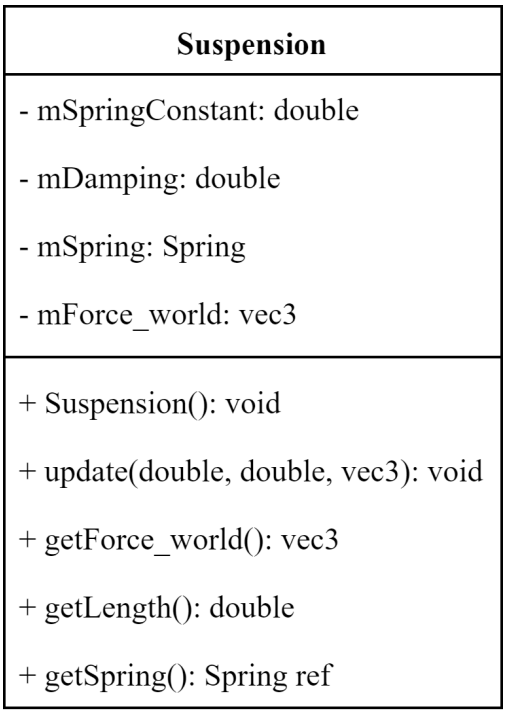


Figure 4.7: A potential structure for the Suspension class.

4.2.2 External influences

In order to make the simulation interesting, the car must interact with an external environment that will influence its motion. The car will act under the influence of the following forces:

- A gravitational force
- An aerodynamic drag force
- 4 spring forces generated by wheel collision with terrain

The first two forces listed here can be calculated with relative ease (in a highly simplified model) given environmental parameters and the car's current state. However, the last class of forces will be more complex to simulate. The section of the program focusing on the car's external influences will be dedicated to providing the information required for it to update its suspension (specifically, terrain surface normal vectors and heights).

Environment - overview

To encapsulate the items mentioned above, I will create an **Environment** class. This will need to hold the gravitational acceleration constant, 9.80665 ms^{-2}, as well as the sea level air density of earth for aerodynamic drag calculation, 1.225 kgm^{-3}. These variables can be public and static to allow easy access to them throughout the code, by simply #including "Environment.h". The **Environment** class will also need to contain information about the shape of the terrain the car is on. I will abstract this information away into a separate class called **Terrain**. **Environment** can then also hold a static instance of **Terrain**. This will mean that duplicated instances of the **Environment** and **Terrain** don't need to be passed around to wherever they are needed, or stored by another class. Looking at the requirements for this class so far, a class diagram can be formed:

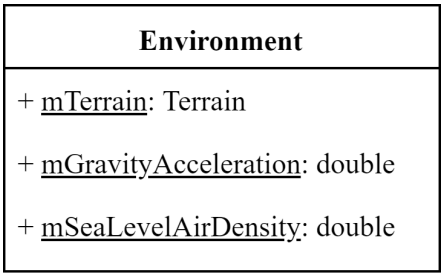


Figure 4.8: A potential structure for the Environment class.

Terrain class

The car's collision with the terrain will be an important part of the application. Randomly generated terrain is a core objective for this project.

This class will contain:

- A variable (unsigned short) called **mSize** to store the size of the terrain. This will be the number of discrete height points along one side of the square. If the height field has a resolution of 1m, then a (256 x 256) terrain would give the player 65025 square metres to drive around in.
- A buffer for storing the terrain's height data called **mHeights**. This will be a (one dimensional) std::vector<double> of size **mSize**^{2}.
- A buffer for storing the terrain's normal data called **mNormals**. This will be a (one dimensional) std::vector<glm::vec3>, of size $2 \cdot ($ **mSize** $- 1)^2$. A value for **mSize** of n (i.e n sample points along an edge) corresponds to $(n-1)$ subsections of the edge, therefore an n x n grid of points corresponds to a grid of $(n-1)$ x $(n-1)$ squares. Broken into 2 trianglular faces, each square can have 2 unique normal vectors.
- Functions for getting the terrain heights and normal vectors at given (x, z) positions.
- A function for getting the terrain's surface type at a given (x, z) position.

> **NOTE:** Whilst the terrain is physically 2 dimensional, I will use 1 dimensional buffers to store its data, to be more cache-friendly. This will require slightly more code for indexing.

The terrain will be modelled as a height-field. Height-field terrain generation is achieved by starting with a perfectly regular, square, horizontal grid of points, and adjusting the heights of the points. The x and z components of the points' positions remain fixed in line with the the regular grid, but by changing the y positions of the points, a variety of terrain features can be created ranging in size from small irregularities in the shape of the terrain such as ditches, to large rolling hills or cliffs.

Generating the terrain in this way allows for a layered approach that will be very useful for implementing specific features such as the track shape, as this cannot be entirely random. The fact that the heightfield is a fixed regular square grid along the xz plane will make retriving data more simple. A generation 'layer' can be defined by a single function that takes as input, a reference to a std::vector<double>. The function will update the contents of the buffer with the addition of a layer of heights. Using multiple layers that each add to the previous layer in a different way will create varied and visually appealing terrain for the car to collide with.

This additive function for generating terrain can be described by the following pseudo code:

```
1: procedure GENERATEHEIGHTDATA
2:     generationLayers.add(Track)
3:     generationLayers.add(Hills)
4:     generationLayers.add(AnotherLayer)
5:
6:     for all layer in generationLayers do
7:         layer.run(heights)
```

The function *layer.run*() is explained in 'TerrainGenerationLayer' below.

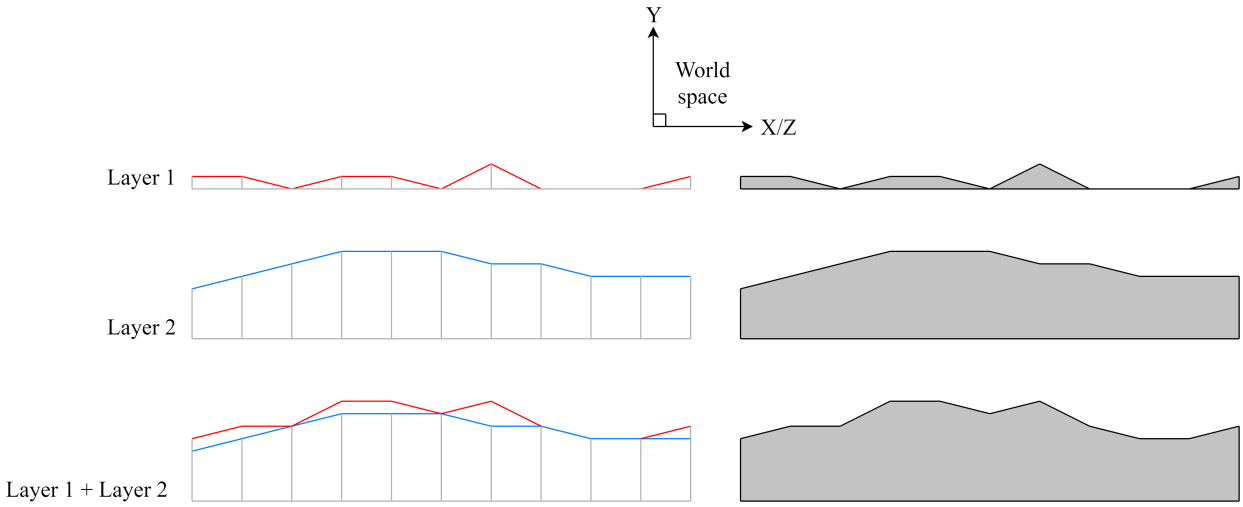


Figure 4.9: A 2D visual explanation of the process of layering terrain.

All heights in the terrain heights buffer **mHeights** will be set to 0.0 to begin with, and subsequent layers will be added to this. This algorithm requires iterating over each terrain generation layer, which in turn requires iterating over each height sample point across the terrain. As a result, I will only call this function once at load time. However after the buffer is full, retrieving the terrain height at any given (x, z) coordinate will be much quicker than recalculating it.

The terrain height data will be stored for the duration of the program, to increase performance. With just rough terrain, it would be feasible to recalculate the ground height at any given point whenever it's needed, but the generation of the track shape will make this algorithm significantly more time consuming, so this will be performed once, up front and saved. Given the size of one side of the terrain in sample points, it is possible to calculate the amount of memory in bytes that will be required to store this data, using $M = 4 \cdot S^2$ where M = memory requirement (*bytes*) and S = the size of one side of the terrain (*samples*). Using a (256 x 256) terrain again, this would mean that the terrain's **height** data would use roughly 26 kB of memory.

The diagram below explains how the height sample points will be arranged in a 1D buffer in memory.

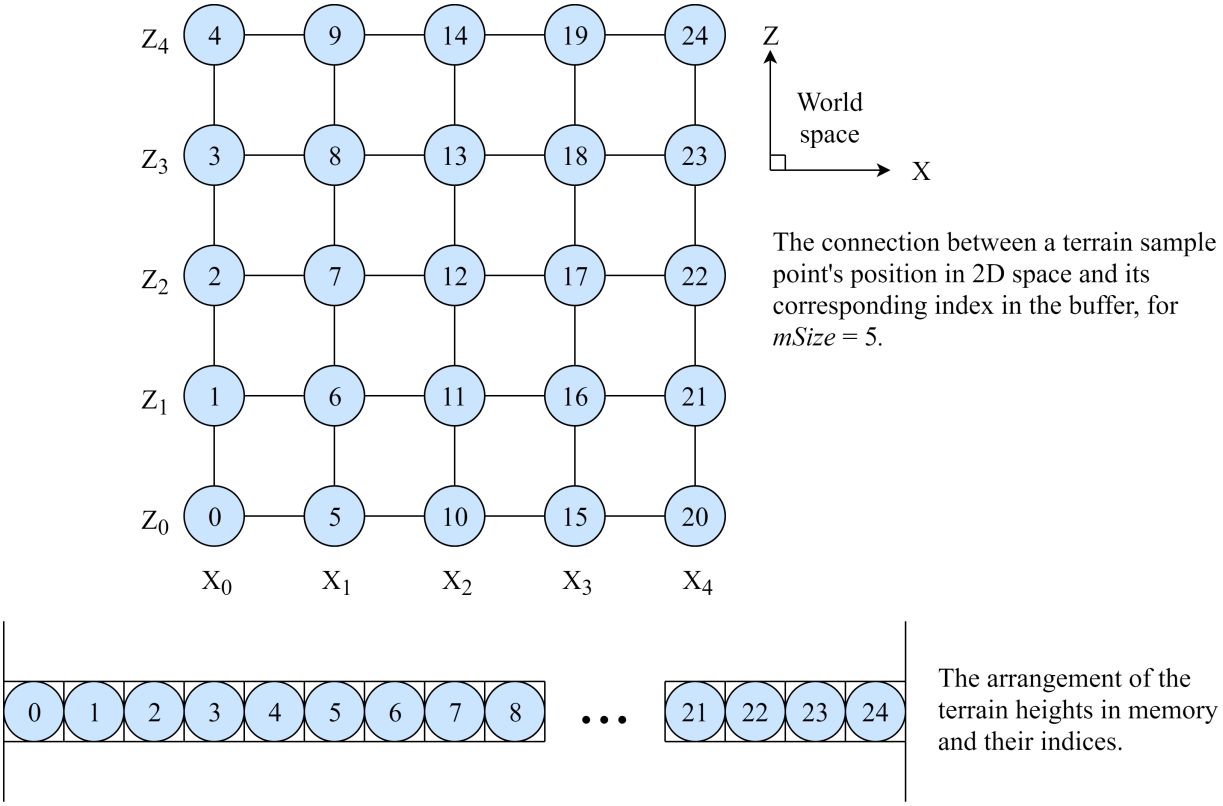


Figure 4.10: Heights arranged in 2D space vs memory

Given an x index and a z index, (X_n and Z_n in the image above), and a size s of the terrain in samples points, it is possible to calculate the 1D index to access the corresponding height value, i, in the buffer using:

$i = s \cdot X_n + Z_n$

The reverse is also possible. A single 1D index can be split into two indices representing its x and z indices, were the buffer a 2D array using:

$x = \lfloor \frac{i}{s} \rfloor$

and then

$z = i - s \cdot x$

This mapping between 1D indexing and 2D indexing will be important during the implementation of terrain height sampling.

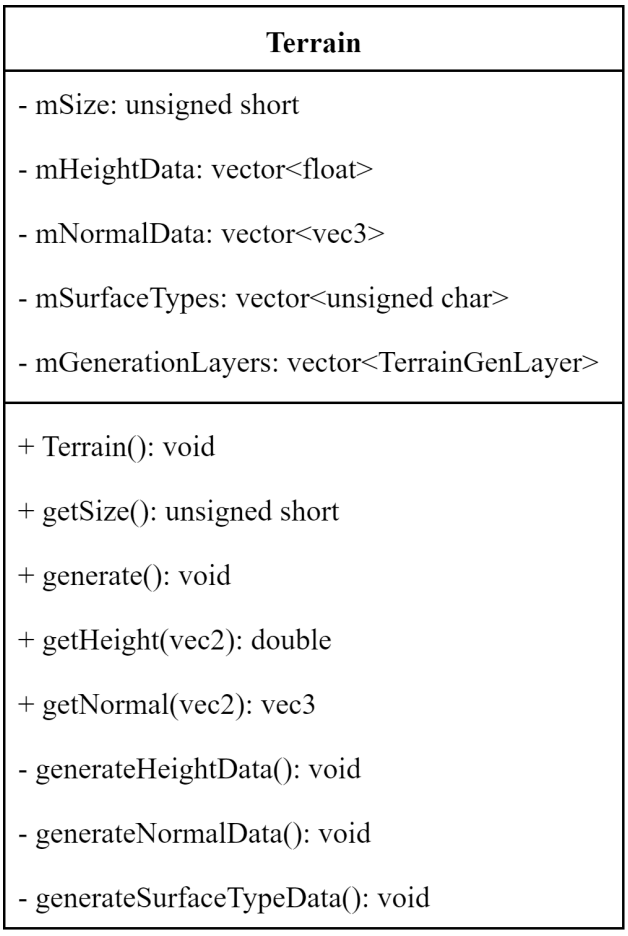


Figure 4.11: Terrain class diagram

TerrainGenerationLayer class

A terrain generation layer could be encapsulated by an object. This would enable them to be stored by the `Terrain` object. I can make `TerrainGenerationLayer` an abstract base class that forces the implementation of a `run(std::vector<double> heights)` function, for example. Layers would then become modules that could be swapped-out and changed independently etc. Using this approach, the `Terrain` object will own an std::vector of layers. When the `Track` object is implemented, it will inherit from `TerrainGenerationLayer` and provide its own function for modifying the terrain's heights. See the sections entitled 'Track generation' and 'Hill generation' for specific examples of these functions.

> **NOTE:** Surface normal data will not be created by individual layers since it depends on the height of the terrain. Each layer will add its own modification to the terrain heights and then the surface normal vectors will be calculated in **one pass at the end**, once all layers have acted.

This class must have:

- A pure virtual function called `runHeights(std::vector<double> previousLayerHeights)` that modifies the previous layer's heights
- A pure virtual function called `runSurfaceTypes(std::vector<unsigned char> previousLayerSurfaceTypes)` that modifies the previous layer's surface types

Using the requirements mentioned above, a class diagram can be created for `TerrainGenLayer`:

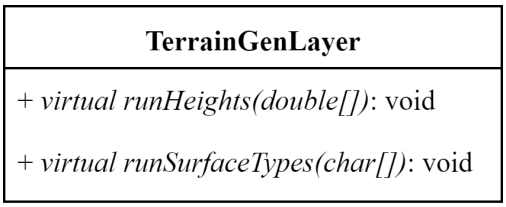


Figure 4.12: TerrainGenLayer class diagram

RoughGround class

The first **TerrainGenLayer** that will be used, will generate rough, bumpy terrain. I will use two layers of Perlin noise to achieve this. The first will form lower frequency features like hills and the second will add smaller higher frequency mounds of earth. By combining the two, the player will have a detailed terrain surface to drive around.

> **NOTE:** The following images have been added after implementation.

Developing on the 2D explanation of terrain layering in fig. 4.6, these screenshots show this effect in the final application. An orthographic camera has been positioned above the terrain to create this view.

Figure 4.13: Left + middle = right

The following pseudocode outlines the **RoughGround**'s implementation of **runHeights()**:

The edge length of the terrain can be found easily given that the terrain is a square. An iteration is performed over each point in the heights buffer. For each point, two Perlin noise values are calculated using different frequencies and amplitudes. These two values are added together to get a total and this is assigned to the correct index in the buffer.

```
1: procedure runHeights(double[] previousLayerHeights)
2:     terrainSize ← √(previousLayerHeights.size())
3:     halfTerrainSize ← 0.5 · terrainSize
4:
5:     for x ← −halfTerrainSize to halfTerrainSize do
6:         for z ← −halfTerrainSize to halfTerrainSize do
7:             currentHeightIndex = (x + halfTerrainSize) ∗ terrainSize + (z + halfTerrainSize)
8:
9:             perlinX ← x/16.0
10:            perlinZ ← z/16.0
11:
12:            total ← 0.0
13:                                                          ▷ Low frequency features
14:            amp ← 8
15:            freq ← 0.1
16:            total ← total − |octavePerlin(perlinX ∗ freq, perlinZ ∗ freq, 3, 1, 1) ∗ amp|
17:                                                          ▷ High frequency features
18:            amp ← 0.5
19:            freq ← 1
20:            total ← total − multiFractalRidged(perlinX ∗ freq, perlinZ ∗ freq, 2, 1, 1) ∗ amp
21:
22:            previousLayerHeights[(x+halfTerrainSize)∗terrainSize+(z+halfTerrainSize)] ← total
23:
```

Track class

The second **TerrainGenLayer** that I will implement, will carve a race track shape into the terrain. This class will be called the **Track**. Just like **RoughGround**, the **Track** will take the current height buffer of the terrain (this will be the output of **RoughGround**) and add to it.

The track itself will be a continous loop (starts and ends at the same place), with a constant width. The shape of this loop will be defined with a function that takes in a value between 0.0 and 1.0 (inclusive) and returns an angle in degrees between 0 and 360. The value passed in represents a distance along the loop from the starting point, with 0.0 being the start and 1.0 being the end (also the start). The angle returned will represent the angle between the tangent to the loop at the input position, and the tangent to the loop at the start.

For example: For a circular track, this function is simply a line between (0, 0) and (1, 360). The angle of the tangent to the circle increases linearly with the distance round the loop. Travelling eg. 40% of the way round a circle will mean that you have rotated 40% of 360 (144) degrees. Using this angle look-up method, complex continous track shapes can be defined. It's important to note that this method only defines the **shape** of a track. The position, rotation and scale of this shape is ignored - something that will have to be considered when generating the track within the terrain's square.

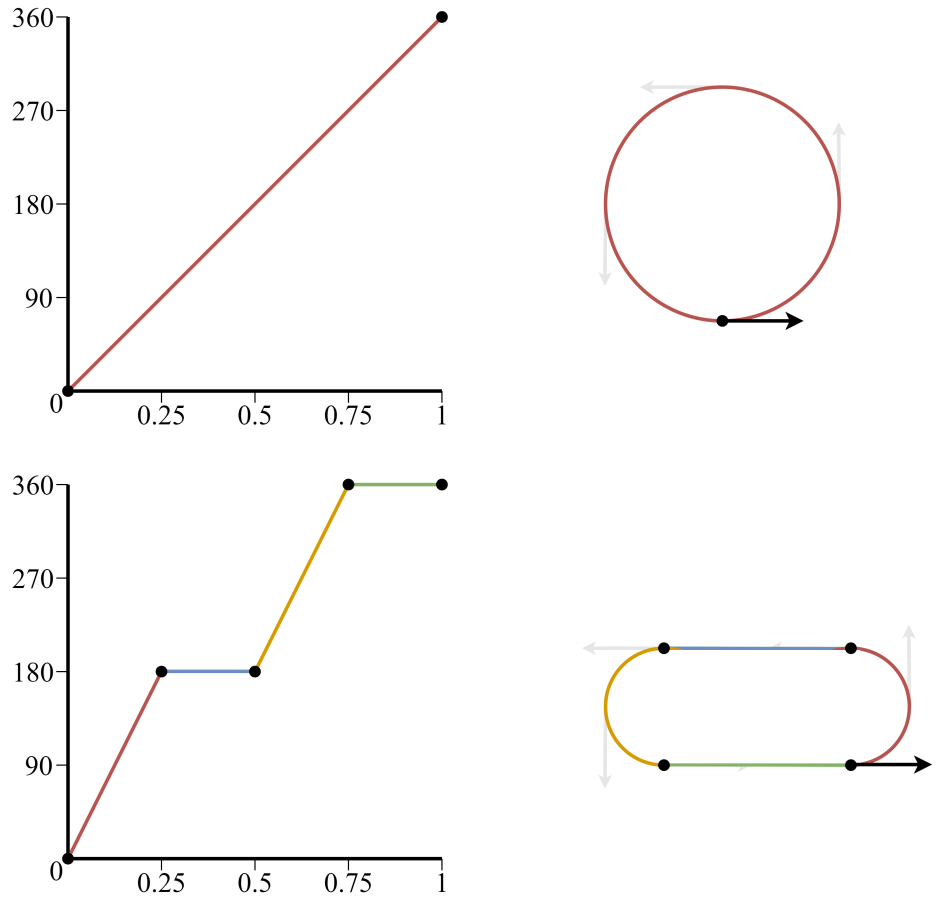


Figure 4.14: Percentage-angle graphs for two track shapes

A variety of track shapes can be made by simply combining common fractions (halves and quarters) of circles of differing radii, with straight lines. This also gives the tracks a good balance between straight regions in which to build up speed and bends that require braking and steering control. This design allows for tracks that appear complex to be defined relatively easily.

I will store the track shape as a vector of points in percent-angle space. On the diagrams shown above, the coordinates of the black circles will be stored. For samples that lie between these points, linear interpolation can be used.

> **NOTE:** Tangents with angles of 0 degrees and 360 degrees are identical. This occurs because the track is a loop. I will have to consider this during implementation.

To cover the shape of the track, I will break it into a fixed number of discrete samples. By sampling the track's current tangent angle at each one of these points, a position tracker can be moved along the tracks tangent. Doing this continuously over many samples, the position tracker will 'walk' around the track shape and eventually return (very close) to the start position. This will work as long as the steps are a constant size. The greater the number of steps, the greater the accuracy and the closer the position tracker will be to the start position after travelling round the whole shape. The number of samples will be tuned to be just accurate enough. In pseudocode, the algorithm is as follows.

```
1: numSamples ← 4000
2: positionTracker ← (0, 0)
3: startDirection ← (1, 0)
4: currentDirection ← startDirection
5:
6: for i ← 0 to numSamples do
7:     currentAngle ← lookUpAngleAt(i/numSamples)
8:     currentDirection ← rotate(startDirection, currentAngle)
9:     positionTracker ← positionTracker + currentDirection
```

positionTracker and *startDirection* have been given values of $(0, 0)$ and $(1, 0)$ for simplicity and the step size in space is 1, but for the track to be created within the bounds of the terrain square these 3 parameters must be calculated. The tracks overall size must be maximised given a starting position, starting direction, step size, width and a padding value (a distance around the terrain's border that must be clear of the track).

Pilot run

The starting position, starting direction and step size depend on the track's width, the terrrain padding value and **the tracks overall shape**. This means that an initial iteration must be completed, in order to reach these values. They can then be used in the final version that will perform the height data addition. I will refer to this separate initial iteration as a 'pilot' run. Details on the pilot run are outlined below:

1. Firstly, using the terrain dimensions, the terrain padding value and the track's width it is possible to calculate the maximum bounding square that the track's loop can occupy. If its minimum bounding rectangle - "MBR" - is larger than this result, the track cannot fit neatly on the terrain. This is illustrated by the following diagram:

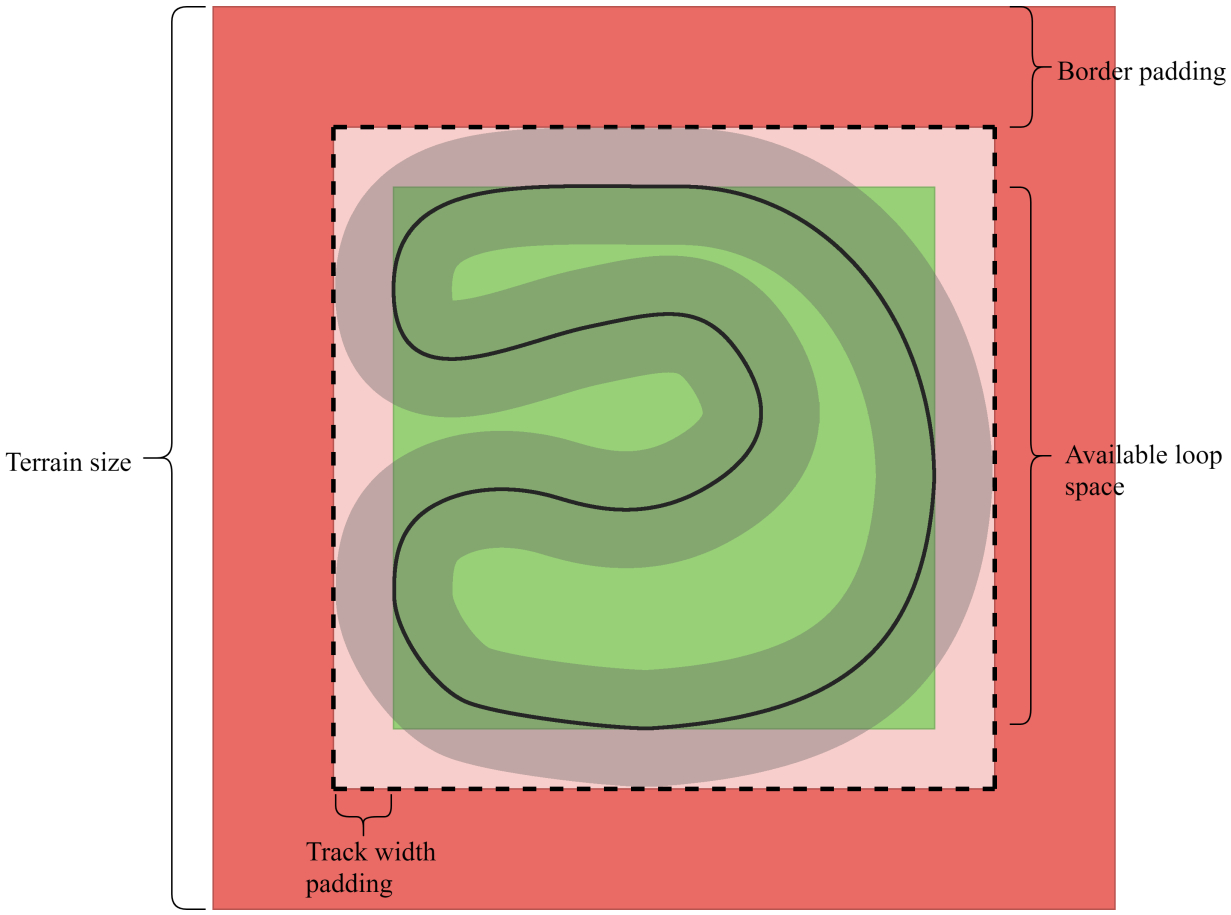


Figure 4.15: The maximum available space for the track loop to fit within the terrain

2. Next, the MBR of the loop in 'pilot' space (not to scale and using a step size of 1) must be obtained, using the pseudocode above. At each iteration, the current x and y components of the position tracker should be checked against $maxX$, $minX$, $maxY$ and $minY$ values. If the position lies outside the current MBR defined by these values, then they values are adjusted to include this point (this is similar to finding the minimum/maximum values in a list, only in 2D). Once this step is complete, $maxX$, $minX$, $maxY$ and $minY$ will define the MBR of the pilot track's loop. This is the green box in the diagram above - the loop must fit inside this box.

3. Now that the maximum size and pilot size of the track's loop have been calculated, they can be compared to produce a scaling factor to scale the pilot track to fit inside the terrain. The longest edge of the pilot bounding rectangle is used for this scaling factor. The equation to calculate this value is

$$SF = \frac{maxSize}{max(pilotDimensions.x, pilotDimensions.y)}$$

If the pilot track is too large, then the scaling factor will be less than 1 and if it's too small, the scaling factor will be greater than 1. This value is the step size required for a track to fit perfectly in the terrain.

4. So far, the scale of the track is known but the starting position and direction must be determined before the main iteration can take place. As the pilot run's starting position is (0, 0) it is possible to keep track of how far up/down/left/right the loop extends (this could be done as part of the MBR calculation to minimise code). For example, a loop might occupy space between -10 and 20 on the x axis and -5 and 18 on the y axis. These values allow the pilot track's starting position to be localised within its MBR, which in turn allows a starting position to be calculated for the main run.

Modifying height data

The main iteration can now take place. The algorithm is an extension of the pseudocode on page 41, and uses these new calculated values. During each iteration (step along the loop), instead of just updating a minimum bounding rectangle, the terrain must be modified. Rounding the position tracker's coordinates down, means that it will be possible to index the terrain height buffer passed in to `runHeights()`, as this buffer only contains height points at integer coordinates. By simply subtracting from the height at this index, the loop (**a line**) can be traced into the terrain.

But this ignores track width - the track must be wider than 1m! To make the track wider, instead of just modifying one height index on the loop, a circle of points around this position can be modified. The circle's diameter will be the track width. Iterating over a square of points surrounding the target index and using pythagoras' theorem allows points inside a given circular range to be identified and modified. The following diagram shows illustrates this:

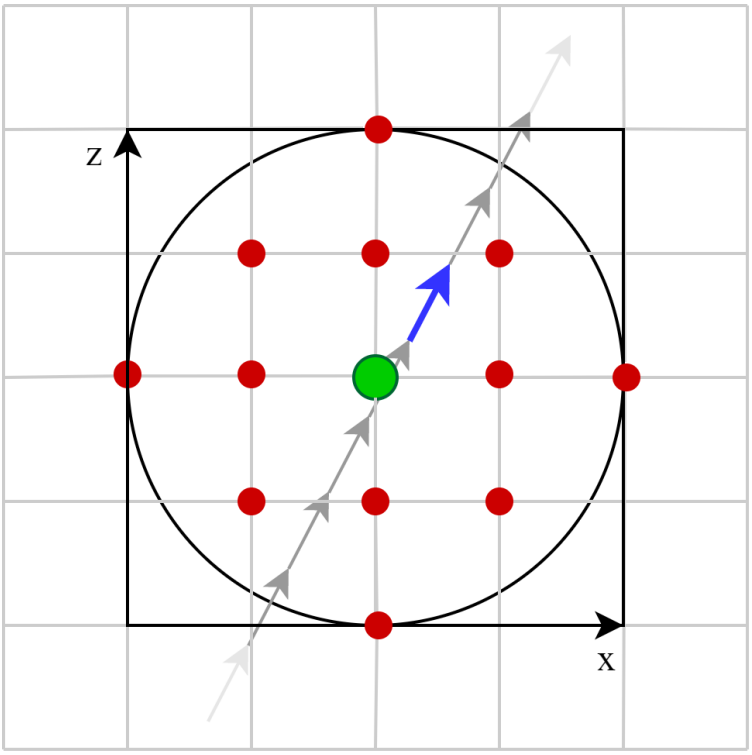


Figure 4.16: Detecting points within a circular range

Explanation The blue arrow represents the end of this iteration's step. Rounding the coordinates at the arrow head down to an integer coordinate takes us to the point denoted by the green dot. This point is the centre of the imprinted circle. Next a separate iteration takes place over the points surrounding the centre (green), shown by the black box and the x and z axes. If $\sqrt{x^2 + z^2} > \frac{terrainWidth}{2}$ then the point is skipped. Otherwise, the height of this point is adjusted. Points that satisfy this equation are highlighted by red dots. They can be lowered by a cubic function of their distance to the centre to give the track a 'bowl' like shape, as opposed to a walled trench. This process should take place every iteration. A optimization check can be made to make sure that the current point has not already been used in order to prevent the same sub-iteration occuring unnecessarily.

4.3 Visual

UILayer class

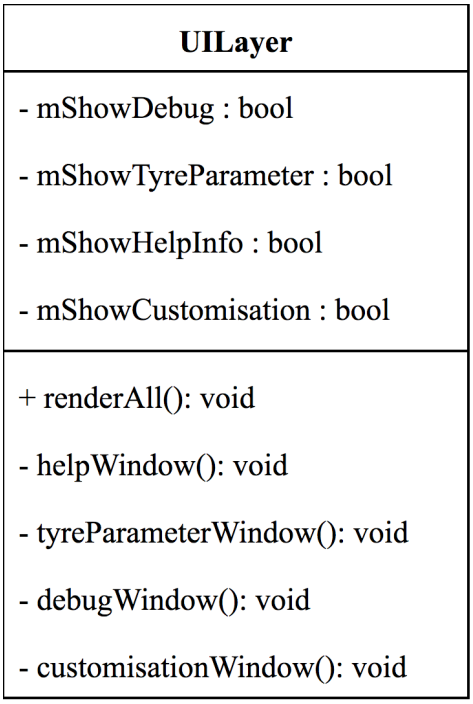
The user interface of this application will be structured around ImGui windows. I will create a main 'control panel' that provides access to simulation controls/settings as well as visibility options for other windows. ImGui was written to - among other things - make debugging easier, and faster. Calls to ImGui functions can conveniently be made at **any point** during the rendering cycle in a program. I will aim to encapsulate all this code into one class, and ultimately one function call. When this is called, all user interface elements will be submitted for drawing in one place - they are just layered on top of what has already been rendered. Code for separate windows can be placed in separate member functions within this class (called '`UILayer`') and called with one main 'renderAll' member function.

I currently plan to add the following windows:

- A main control panel containing settings
- A help window
- A debug view window to display detailed live telemetry needed for debugging
- A tyre parameter window
- A car customisation window

All windows apart from the main control panel will be optionally excluded from rendering (as opposed to simply minimised). The rendering state of each window can be represented by a series of boolean member variables in the `UILayer` class. Before making each window's function call, a check can be made to see whether its corresponding visibilty variable is set to true or not, to determine whether the call should be executed. A section of the main control panel can be dedicated to activating/deactivating these windows, using checkboxes.

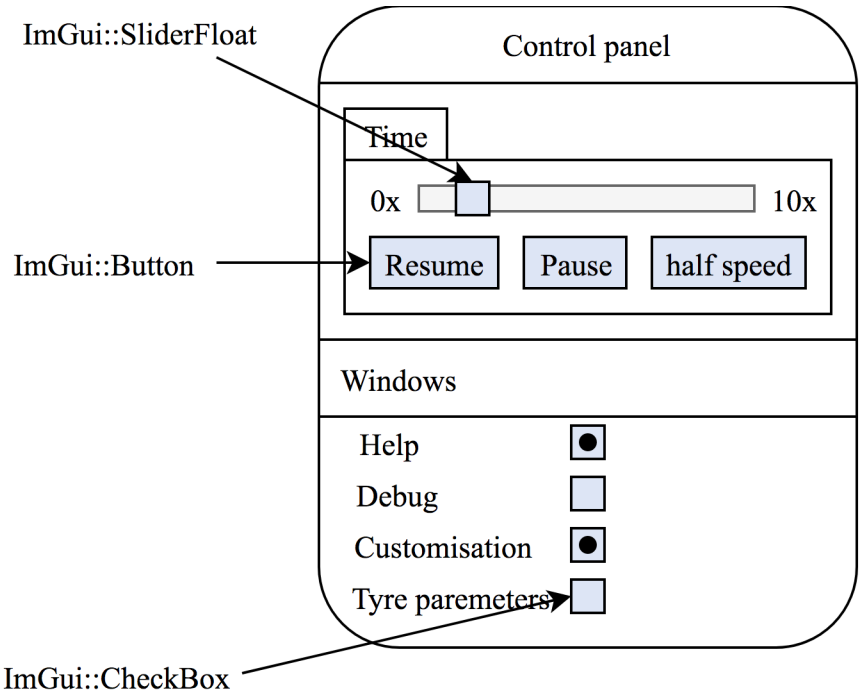
A class diagram can be created given **UILayer** 's current responsibilities:



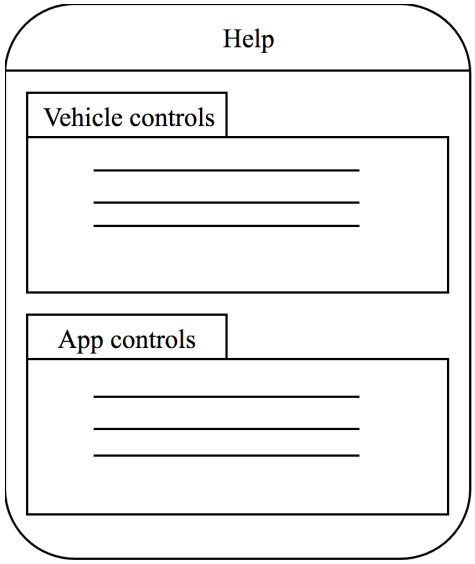
User interface designs

Based on my knowledge of the features that can be included in ImGui windows, I have created the following window designs:

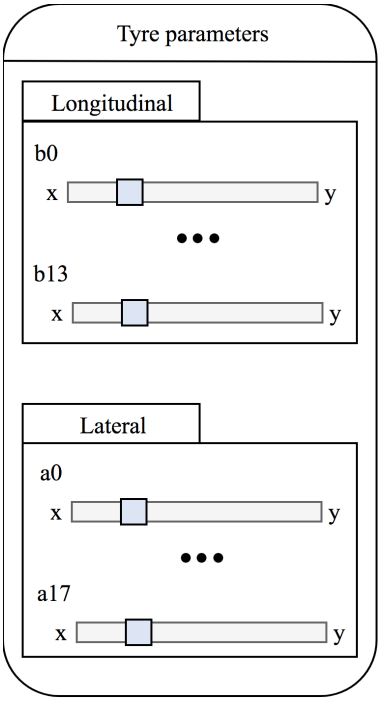
This is a basic outline for the main control panel of the application. A slider can be used for adjusting the simulation speed and button's can be used for resetting it to commonly used values (normal speed, half speed, or paused - a speed of 0). Checkboxes in this panel can be used to toggle the visibility of the child windows.



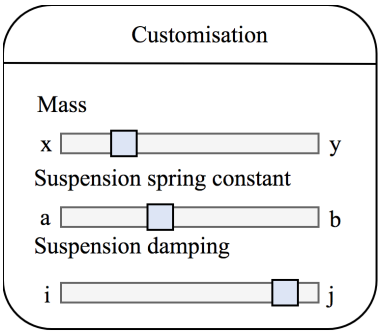
The help window will use two sections as shown by the diagram below. The first will be focused on all vehicle-specific controls, and the second will provide a guide to using the complete application.



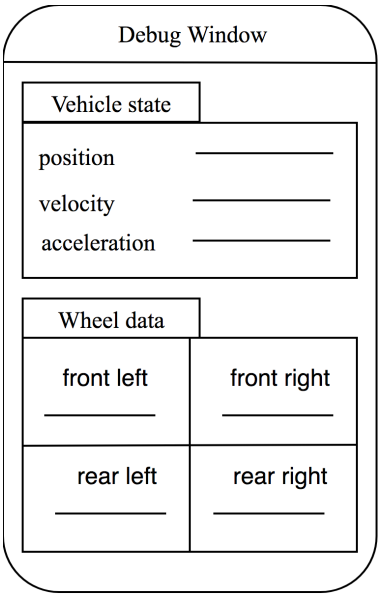
I will also break the tyre parameter window into two sections for longitudinal and lateral parameters. When modifying the behaviour of a tyre, it is more intuitive to use sliders to get a feel for what effect specific parameters have on the tyre. The following diagram outlines a possible structure for this window.



Sliders will also be used for the more general customisation of physical properties of the car such as mass, or suspension properties like the spring/damping constants.



A debug window will be used to output the live telemetry data being generated by the simulation. This will be broken up into two sections. The first will display an overview of the rigid body's state, and the second will display wheel data. This section will be broken into quadrants that map to one wheel each.



Part III

Testing

Outline

The testing for this project has mainly taken place during the implementation. As many features of the simulation depend on classes at the base of the object hierarchy, by testing these features, features that depend on them are also tested.

For example, the **Tyre** force generation takes place at the bottom of the object hierarchy (see section 4.1.1). If there is an error in this area of the program, the error is propagated upwards so that the **Wheel** does not produce the correct total force, and then the **Car** does not calculate the correct torque or move realistically. Considering areas of code as leaf nodes in a tree structure, if an error is seen at a higher level in the program, it can be traced by eliminating branches of the tree, by replacing calculated values with hard-coded values that are known to be correct. The majority of testing during implementation involved this process of elimination, as well as careful incremental checks of features just after they had been implemented (the tyre force generation should be confirmed to be working correctly before implementing the Wheel class for example).

4.3.1 Broad testing

Many features of this application could be tested with the same tools/using the same methods. This section details some of the more broad testing that has taken place.

Time speed control

One of the most frequently used features that made testing easier, was the ability to adjust the speed of the simulation. The **Framework::Physics::RigidBody::update()** function takes in an update delta and by multiplying this by a **mSimulationSpeed** variable, it was possible to slow down and pause the simulation to move the free camera into a better position, or check values at a specific time for example.

Figure 4.17: This is the ImGui element used to control the simulation speed.

For example, while testing the wheel collision and resulting reponse from the suspension, the simulation moves too quickly at real time to be able to check these processes. Since this particular feature was taking a significant amount of time to complete, I added a button labelled 'suspension demo' to quickly reset the car 5 metres above the ground, as well as randomise its orientation. This provided a wide range of scenarios with which to test newly added code. I made use of the time speed slider to test these features by:

1. Pausing the simulation (time speed = 0)
2. Pressing the 'suspension demo' button
3. Positioning the **FPVCamera** beneath the vehicle, looking towards one of the wheels
4. Gradually increasing the time speed until the wheel was about to collide with the terrain
5. Reducing the time speed again to a much slower value to observe the collision

This is just one example, but the ability to change the speed of the simulation has been very useful throughout the development of many features.

Console output

During the early phase of the project, I made use of the console to output data. Printing a value each update and then pausing the simulation at a point of interest, meant that the (very recent) history of this value could also be seen. Again, this is not a specific test, but was used in many places in this project.

Orthographic cameras

Whilst the final output of the simulation uses perspective cameras, there is a better option available for debugging. Orthographic cameras use orthographic projections. This type of projection does not change the sizes of objects as their distances to the camera change. By aligning the car with the principal world-space axes and setting the direction vector of an orthographic camera to point down one of the axes, the car can be viewed perfectly from one side - it appears 2D. This was very useful for debugging the collision of the wheels with the terrain, as well as load transfer.

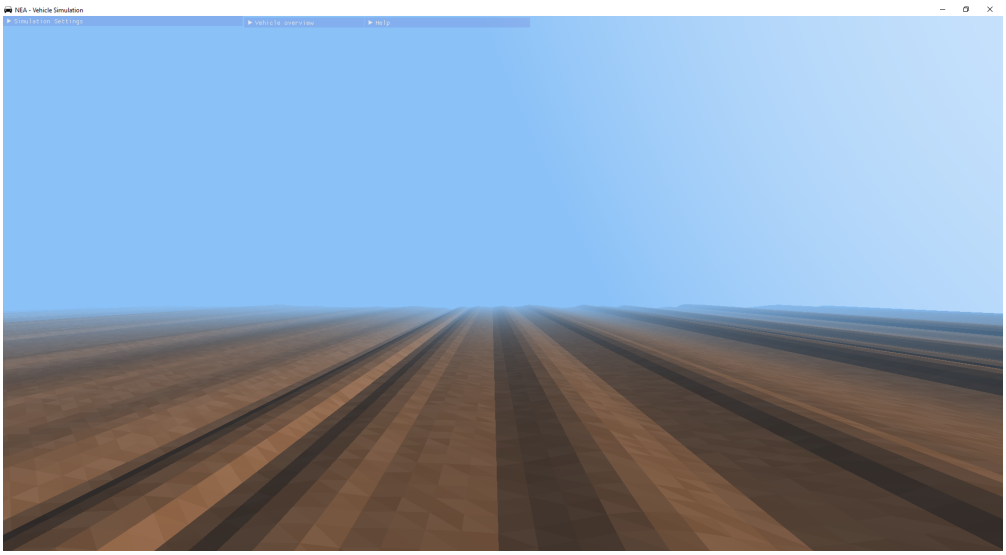


Figure 4.18: By ignoring the Z coordinates of the positions on the terrain during its generation, the terrain's features will only vary along the X axis.

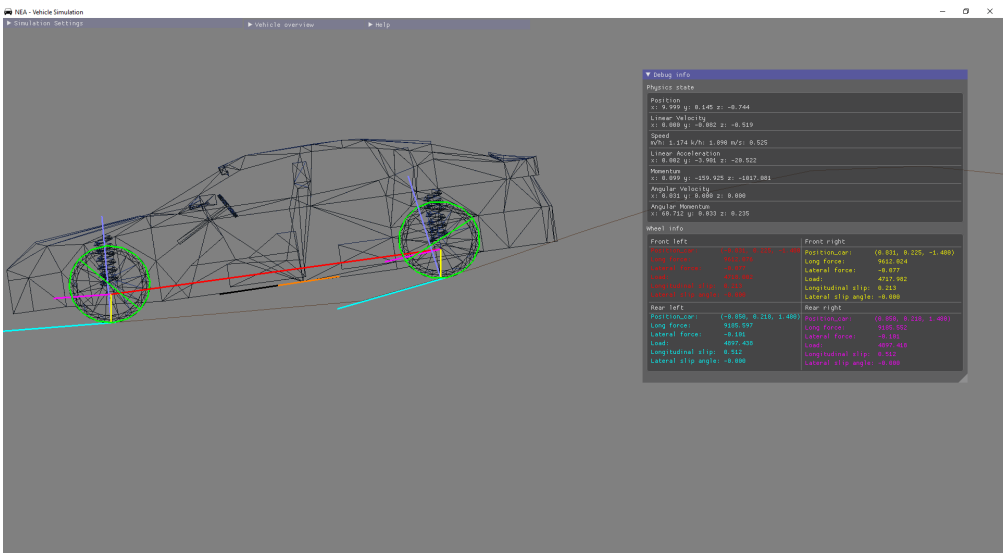


Figure 4.19: Using the terrain shape above, turning on wireframe rendering and viewing the car directly along the Z axis with an orthographic camera means that wheel collision and suspension response can be tested in 2D.

DebugCarModel

In addition to the **GameCarModel**, I have created a debug model of the car to visualise many different areas of the simlation simultaneously. It involves both a new rendered model overlay of the Car object to display vectors and the position of the car and an ImGui window to display values.

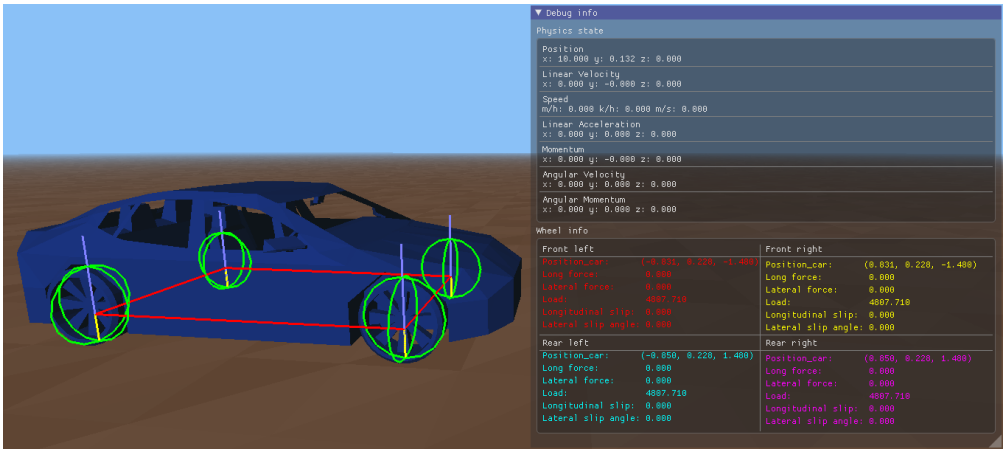


Figure 4.20: The DebugCarModel overlay when the Car is at rest.

Different colours on the model are used to outline different components of the vehicle. The following indicators reference the Car's overall state:

- A red rectangular frame to display the position and orientation of the Car. This is green when colliding with the terrain and red when not.
- A black vector originating at the Car's centre showing its linear velocity
- A black vector originating at the Car's centre showing its angular velocity
- An orange vector originating at the Car's centre showing the drag force acting on it

In addition to these, there are per-wheel indicators used to display the state of individual wheels. Each wheel has:

- A spherical wireframe outline. There are two planes used to assemble the sphere in this model, as opposed to just 1 in order to show two axes of rotation (steering angle and rolling motion). This is green when the Wheel is colliding with the terrain and red when not.
- A normalised (length 1), purple, terrain normal vector originating from the point directly beneath the WheelInterface's world-space position (on the terrain's surface at this point). This has been used to check that the terrain's internal normal data matches that of the graphical model.
- A yellow line joining the WheelInterface's world space position, to the terrain surface directly beneath it. This has been used to check that the terrain's internal height data matches that of the graphical model.
- A magenta vector, originating from the WheelInterface's world-space position representing its current world-space velocity.
- A cyan normalised vector of length 1m, originating from the tyre contact patch on the terrain, that represents the direction of the total force generated by the tyres. This vector is aligned at a tangent to the terrains surface at the contact patch.

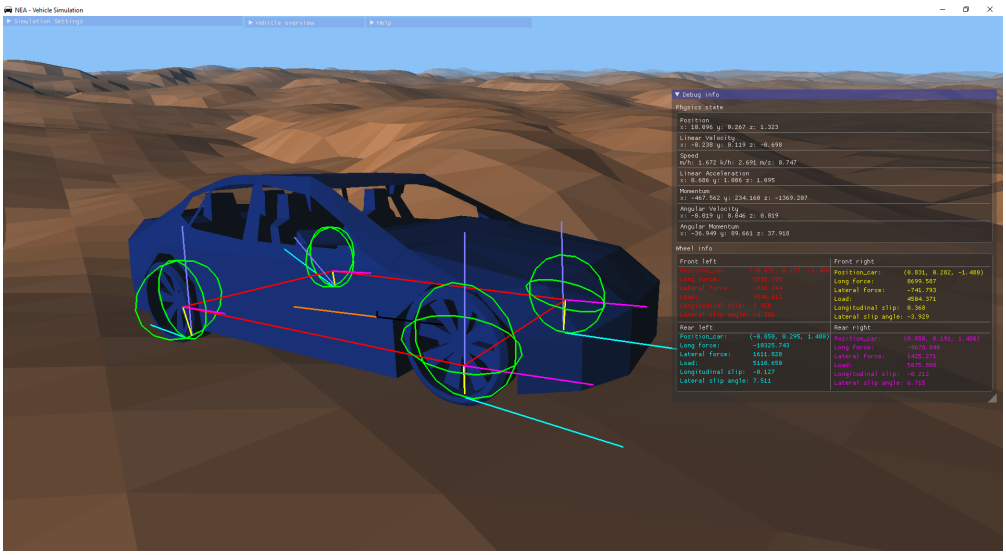


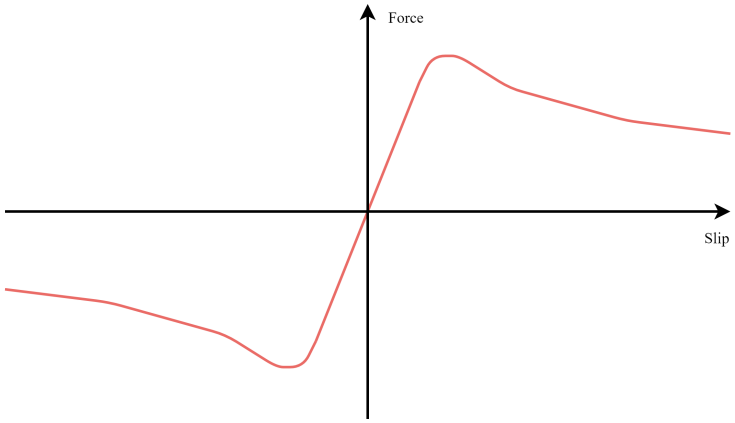
Figure 4.21: All debugging indicators can be seen in this image.

4.3.2 Specific testing - overview

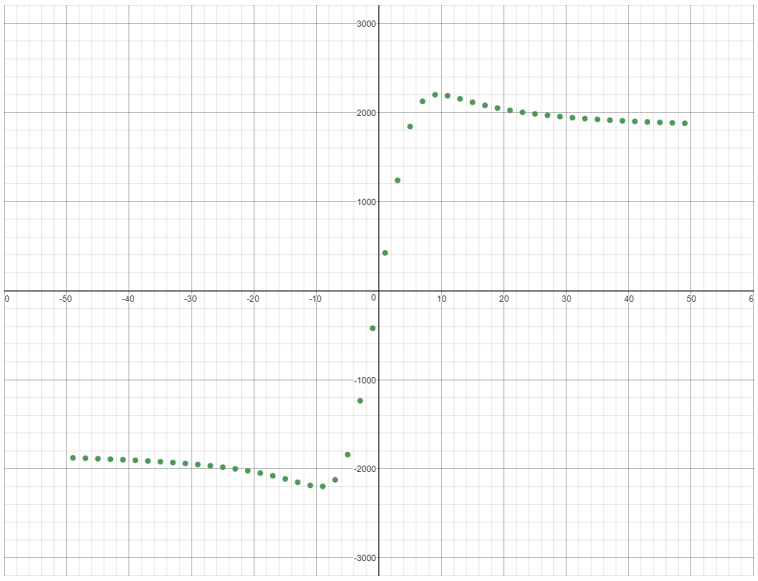
The section below describes two instances where specifically making sure to test certain critical components was important.

Tyre force curves

The tyre force generation is a critical part of this application. Explained in section 'Pacejka Magic Formula Implementation', a force value (either lateral or longitudinal) is a function of with a slip (amongst other parameters). This function forms a distinctive curve, the general shape of which is shown below:



In order to test that the results of the force generator were accurate, I stored the results of several slip samples and plotted them on a graph using [18]. The result can be seen below:



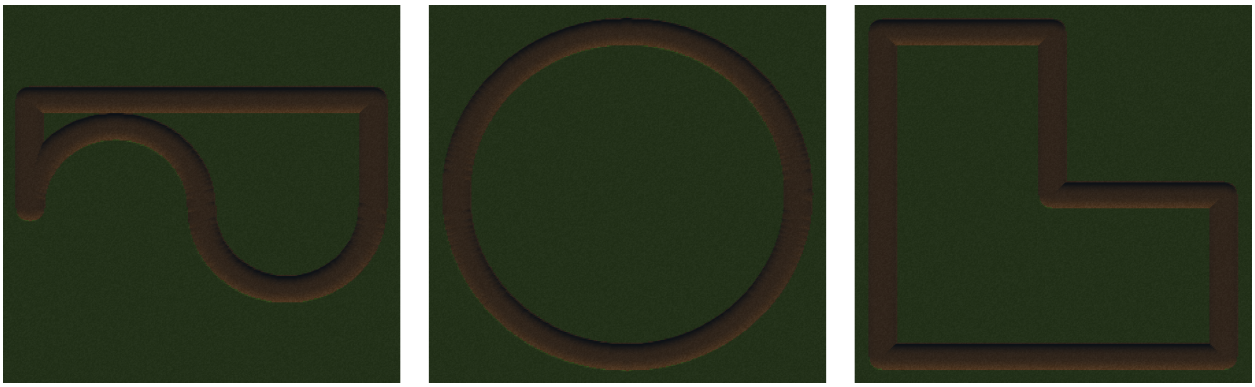
This result (along with others using different tyre parameters) shows that the force calculation is correct. Slip values can have been tested using the **DebugCarModel** 's ImGui panel that displays all slip values in real time. Slowing down the simulation and driving forward shows the slip increase dramatically before slowly returning to 0 as the contact patch velocity matches that of the road beneath the tyre.

Track generation testing

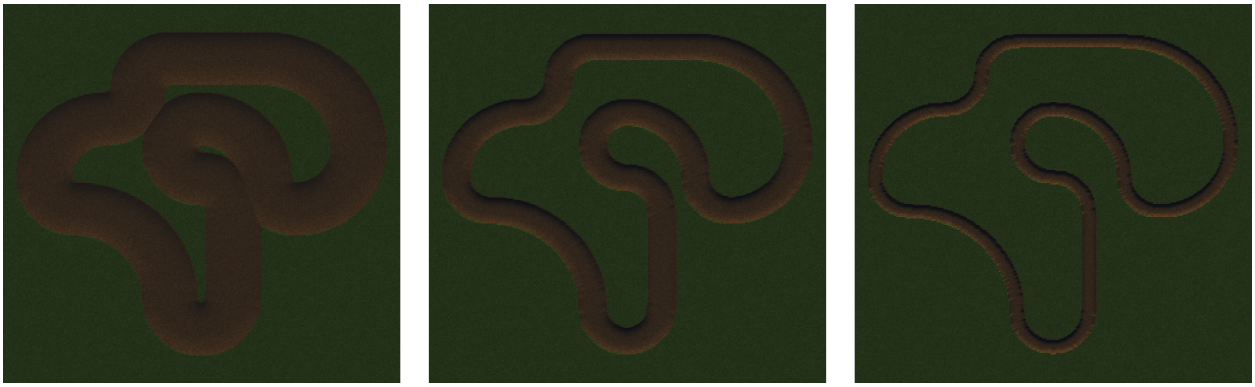
Multiple track generation test have been performed because of the algorithm's relative complexity. The 4 main variables that act in the generation of the track are:

- The track's shape function
- The track's width
- The terrain border padding value
- The number of samples of the shape function that are taken

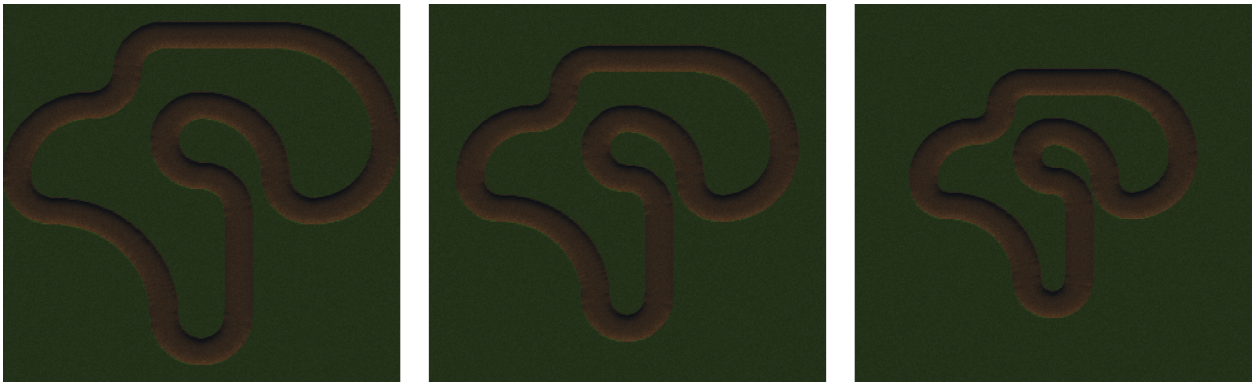
The stored percent-angle points in the track shape function can be changed to create different shapes. In each of the following cases below, the track is continuous and fits the terrain as expected showing that the shape function and angle look-up method works correctly.



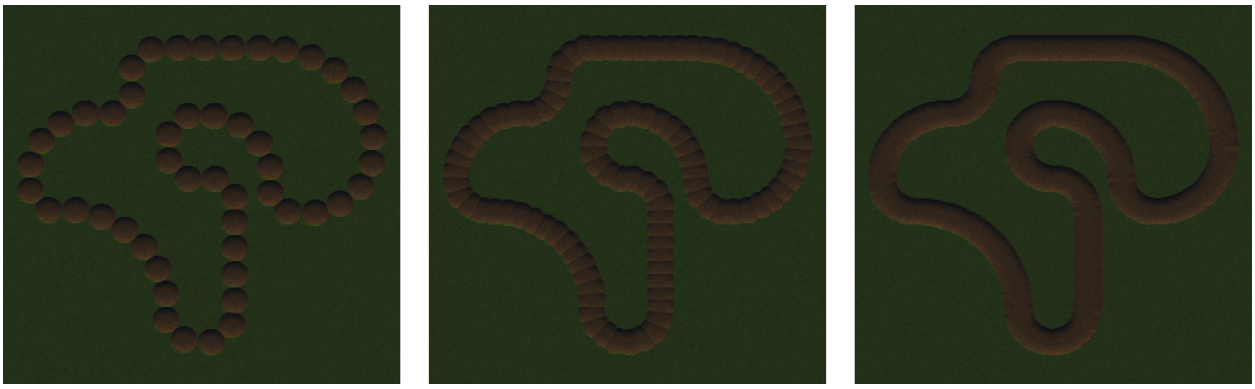
The track width has also been checked to make sure that - whatever this value - the shape still fits within the terrain and the border padding is not affected. From left to right the values used were 40m, 20m and 10m.



The next parameter to be tested is the terrain border padding. This is the depth of an area around the terrain's border that will not be affected by track generation. From left to right, the values used were 0m (the track is generated right up to the edge of the terrain), 20m and 40m. A value of 10m was chosen for the release version.



Lastly, the number of discrete shape samples had to be determined. Too few and the track is not smooth, too many and unnecessary computation is taking place. The image on the left shows the result when 50 samples are taken. This is clearly too few. The middle image - using 100 samples - is an improvement but the shape still lacks sharp edges. Through experimenting with different values, the track shape's edge becomes well defined at around 1600 samples. The image on the right shows 2000 samples.



4.3.3 Effective testing

During a mid-development discussion and demonstration with one of my end users, a severe bug was found. They noticed that if the up arrow (accelerate) was held down while the engine mode was toggled (from forward to reverse) and then released, the car model would disappear completely. This bug has a direct link to objective 2 and if not fixed, would mean that objective 2 could not be met as the car model was no longer being rendered.

At first glance it appeared that the car's disappearance was linked to input-related code, because it happened as soon as the up arrow key was released, but by using the time slider mentioned earlier, more information could be obtained. Slowing down the simulation and reproducing the bug showed that the back wheels disappeared initially when the reverse key was released, and then the whole vehicle disappeared soon after. Since the motion of the car depends on the tyre force, this bug looked like an issue with code linked to the tyres. The **DebugCarModel** 's ImGui window - also mentioned above - showed that that the tyre's longitudinal slip became equal to infinity (displayed as 'inf') while the bug occured. This narrowed down the possible location of the bug to the **Slip** class.

> **NOTE:** The narrowing-down of possible error locations is a demonstration of the point mentioned earlier about the tree-like dependency structure of the simulation: if there's an apparent issue with both the tyres and the car, given that the car depends heavily on the tyres, the **Tyre** class should be checked first.

Since the **Slip** class is small, it was easy to see that there was a division-by-zero occuring when the tyre's changed direction. This meant that the tyre force was instantaneously incorrect, which in turn meant the tyre's angular velocity became infinite. Incorrect forces were propagated up to the **Car** , which went on to produce erroneous transformation matrices. When these matrices were used to transform the **GameCarModel** and **DebugCarModel** it caused them to disappear.

Part IV

Evaluation

4.4 Reflection

4.5 Meeting objectives

Objective	**Notes**
"Simulate the motion of a 4-wheeled vehicle with 6 degrees of freedom in motion (in real time), under the influence of multiple forces"	The car has 4 wheels, can both move and rotate along/around the x,y and z axes and moves in real time. The forces acting on the car are: gravity, aerodynamic drag, 4 tyre forces and 4 spring forces.
"Display the output of the simulation in real time, in a graphical application with a 3D rendered scene"	My program uses modern OpenGL to render: A model of a car (comprised of a main body mesh, a steering wheel mesh, 4 wheel meshes and 4 spring meshes), a heightfield and a skybox.
"The graphical output of the application should have a low-poly style"	The terrain mesh automatically takes on a low-poly style due to the resolution of the heightfield and flat shading. The car model was created with an intentionally small number of triangles.
"Allow the 3D rendered models to be viewed from different perspectives with multiple cameras, as the simulation is running"	The simulation can be viewed from a) a free camera that the user can fly around b) a camera tied to the movement of the car, positioned where the driver's head would be and c) a camera facing the front right wheel on the car. The user can switch between any of these cameras at any time while the application is running.
"Allow the user to control the motion of the car using the keyboard and mouse"	The keyboard can be used to steer the car and apply the accelerator/brake pedals. The mouse can be used to interact with the UI, which in turn has an effect on the car's motion.
"Procedurally generate and store the heights, surface normal vectors and surface types, of a large, square terrain heightfield"	A 256x256 heightfield is generated at load-time and stored for the duration of the program's execution. It contains the three buffers mentioned above, and they are all fully populated.
"Imprint a track shape into the terrain data"	The terrain's height and normal vector buffer contains a noticeable track shape containing both straights and bends.
"Display simulation settings and output with a user interface, that can be navigated with the mouse"	The user has access to multiple ImGui windows that display: a main navigation/settings panel, a vehicle overview, output data, a customisation panel, tyre parameters and a help menu.

4.6 End user feedback

To assess how well my solution met the needs of my end users, I created a Google Survey that asked the following questions:

- "What do you like about this application? (if anything)"
- "What do you dislike about this application? (if anything)"
- "Were you able to use this application easily? If not, please outline why it was difficult to use."
- "Which area(s) do you think need the most improvement? (if any)"
- "Are there any features you would like to see added in a future version? If so, what are they?"
- "Do you have any other comments?"

The 3 responses all indicate that the project was an overall success, and also provide good feature/improvement suggestions for future development. They can be seen below:

"What do you like about this application? (if anything)"

The graphics are simplistic and nice looking. The driving is satisfying to play with, and you can tell uses proper physics (each wheel obviously has independent suspension so acts realistically).

I like the graphics, they are very pretty. The physics are very precise and I always feel I am in control of the car, and when I fail in driving I know it was my fault and not the games. The controls were clearly displayed and the G.U.I. was intuitive and easy to use. There were lots of settings which I could use to test all aspects of the game, for example the suspension demo.

The fidelity of the physics simulation of the car's suspension system is one of most satisfying parts of this game. Seeing the springs compress and based on a given force for each wheel, which can be affected by the elevation of ground each wheel is placed on, is quite visually appealing. I find the small details like how the rotational displacement of the steering wheel maps to position of the car's wheels, as it would in the real-world, to be a rather delightful. Furthermore, I found the ability to change the vehicles colour easy to use. I really enjoyed being able to change simulation parameters with the tyres and ability to change playback speed.

"What do you dislike about this application? (if anything)"
Not much to do, would be better if there was some sort of race against AI or other players feature.
Nothing.
No. Nothing to dislike.

Oliver's response touches on a good point, and one that I plan to address in the future. Due to the time constraints of this project, the number of features and 'things to do' is limited. Adding the option for a competitive element to the program would make it more entertaining. To begin with, recording the user's progress along the track from the starting position and timing their return would allow them to try to beat there own personal-best race times.

A staring point for the implementation of this feature might be a function that - with knowledge of the track's shape and scale - could take in a 2D position on the terrain and return a value between 0.0 and 1.0 to represent a fraction along the track. This function could return -1.0 for example, if the car moves off-road. This function could use a 'snake' of bounding spheres that cover the full track shape, positioned at load-time, however this might be the most performant solution. Each sphere would be checked to see if it contained the car's position. It could be optimised with some sort of spatial partioning system like an octree to reduce the number of bounding spheres that need to be checked. This would be the most difficult aspect of the feature, but once complete would allow the user to be timed as they drive round the track.

Implementing virtual opponents would be a significant feature and would depend on the progress tracking mentioned above (for determining the current vehicle in 1st place etc.).

"Were you able to use this application easily? If not
Yes it was very easy to get running, but the UI could be more built into the game for more immersion. For example having an actual speedometer in the car rather than a box telling me how fast I'm going.
No.
Overall, I did find the simulation very intuitive, and when I didn't know how to use it, I found the "help" menu to quite useful.

Oliver's reponse here about an integrated vehicle UI is a very good suggestion. Racing games traditionally have speedometers, perhaps in one lower corner of the screen. This makes this the car's speed easy to find and read. If this were to be implemented in the future, I could render a 2D textured quad (square) in the lower right, for example, that represented a speedometer. Then by rotating a narrow 2D black triangle clockwise, the speed could be indicated.

"Which area(s) do you think need the most improvement? (if any)"

Things to do, such as multiple maps, a menu screen, playing against AI or friends.

When the user stops accelerating, but still moving, when turning, the turning is stiff and the car does not turn much. Apart from this the steering works perfectly.

Each section of the simulation has been created to an exceptional standard and requires no improvement.

The tyre issue described in the 2nd response is unique to the tyre parameters in the default 'drifting' mode, that have been (experimentally) set to cause the car to drift very easily. The instantaneous torque provided to the rear wheels causes their angular velocity to rapidly increase, which also affects how easily the car enters a drifting state.

"Are there any features you would like to see added in a future version? If so

Sounds of your car's engine and tyre screeching when you turn would make the game more fun to play. Multiple cars in one game each controlled by a player or AI would make the game more fun.

Timer and checkpoints to do time trials, perhaps against an A.I.

The ability to rebind the car control keys, or offer different default presets for users that prefer "wasd" over the arrow keys. Also, it would be nice to see the current test track develop into a more race-like scenario, with checkpoints and a start and finish line.

"Do you have any other comments?"

10/10

Approved: _____

Approved: _____

Approved: _____

4.7 Future improvements/modifications

There are a few features that I would like add to my solution, if time was available. They are mostly linked to the simplicity of the simulation's physical model of the car, but there are additional features that could be added elsewhere.

1. Inertia and centre of mass calculation

Currently, the car is modelled as a point-mass rigid body. Its moment of inertia (MOI) represents that of a point mass and the centre of mass (Cm) of the vehicle is static and located at the model's origin (this is unrealistically low). To improve this, the car's major components (such as its chassis, wheels, axles and engines) could be given separate local MOI and masses. Since they each have a local position relative to the origin of the car, a total MOI could be calculated using the parallel axis theorem, as well as a combined Cm. These would all be new values for the user to customise. During the simulation itself, the motion of the car would be more realistic as it could be rolled easier along its lateral axis, for example.

2. More detailed engine simulation

At the momen, the single `TorqueGenerator` on the car produces a torque instantly as soon as the accelerator pedal is pressed (similar to an electric motor). This torque value eventually approaches 0 as the generator's RPM approaches a certain arbitrary threshold. This is an extremely simplified model of a car's engine! The relationship between torque and RPM is far more complex and depends on a variety of other factors including gear transmission ratios. Different engines have different performance capabilities/limits and huge number of physical properties that could all be included in the simulation. The model used currently is very basic.

3. Improved wheel collision

The wheel collision model is also simplistic in my application. A wheel's collision is modelled using a distance constraint between two points. Once the X and Z coordinates of a wheel have been calculated in world space, the height of the terrain at this exact position (directly beneath this point) is used in the collision algorithm. This is not realistic, as a wheel is a) not spherical and b) capable of contacting the terrain at multiple points or hitting a curb horizontally for example. A ray-casting model or cylindrical collision volume could be used for more accurate wheel collision behaviour.

4. Full rigid body collision detection and response

Following on from the point about using a collision volume for the wheels, using collision volumes and complete collision simulation for the entire car would significantly increase realism. The car is currently modelled as a point mass, so the chassis of the car does not collide with anything. My game framework library does not currently support this feature.

5. More options for customisability

I think there are a few more features that the user could cusomise in an improved version of my application. These could be both simulation and visual features. The option to customise key mappings and mouse sensitivity would be good. The ability to change geometrical properties of the car and have the model display these changes would also be good features to add.

6. An improved car model

The `GameCarModel` uses an OpenGL shader that gives the mesh a single solid colour. The model could be greatly improved by - at least - using different colours for the wheels, steering wheel and suspension springs. It could be textured. It would be relatively easy to add more animated components, such as moving brake pads or opening doors for example.

7. Optimised terrain rendering

This area specifically could be improved significantly. All triangles in the terrain mesh are currently rendered and - whilst this is not a severe problem in my current version - there is a noticeable effect on the FPS. A better approach would involve separating the mesh into square chunks that can be rendered individually. By performing frustum culling on all chunks in a single pass before rendering, the number of triangles being drawn can be reduced (there is no reason to draw anything that the user cannot see). Using this approach (potentially in combination with a level-of-detail system) the terrain size could also be increased.

8. Improved terrain generation

I think there is potential for an improved terrain generation system in a couple of ways. Firstly, each `TerrainGenLayer` does not currently make use of its knowledge about the previous layer, i.e. the terrain's height buffer is **not** inspected in any way before a `TerrainGenLayer` modifies it. Layers are effectively 'pasted' on top of each other. By iterating over the previous layer to assemble information about its topography, the next layer could contain many more subtle features. For example, the rough ground could become smoother around the track's edges or the track could follow the natural shape of the ground. The track could also bank around corners or include jumps over other terrain features.

I plan to continue this project after A level, and aim to implement some of the features above.

References

[1] Optimizing Vehicle Design with Simulation.
https://insidehpc.com/2017/03/optimizing-vehicle-design-simulation/ (viewed Sep 2017)

[2] Evolution of Need For Speed Graphics (1994 - 2015) | PC | ULTRA.
https://www.youtube.com/watch?v=lx5h6jJ2srY (viewed Sep 2017)

[3] iRacing's vehicle modelling. https://www.iracing.com/car-technology/ (viewed Sep 2017)

[4] BeamNG.drive VS Real Life (Physics Damage Comparison).
https://www.youtube.com/watch?v=jCbz4gqJgYg (viewed Sep 2017)

[5] CAE Driven Aerodynamic Analysis for Car Body Design.
https://www.hitechcae.com/blog/cae-driven-aerodynamic-analysis-for-car-body-design/
(viewed Sep 2017)

[6] Using the PAC2002Tire Model.
http://mech.unibg.it/ lorenzi/VDS/Matlab/Tire/tire_models_pac2002.pdf (viewed Sep 2017)

[7] Vehicle dynamics - Wikipedia. https://en.wikipedia.org/wiki/Vehicle_dynamics (viewed Sep 2017)

[8] Pacejka '94 parameters explained { a comprehensive guide.
http://www.edy.es/dev/docs/pacejka-94-parameters-explained-a-comprehensive-guide/ (viewed Oct 2017)

[9] Engineering Explained - YouTube.
https://www.youtube.com/channel/UClqhvGmHcvWL9w3R48t9QXQ (viewed Oct 2017)

[10] What is Drifting - Explained https://www.youtube.com/watch?v=w7mgEp_zMXQ (viewed Dec 2017)

[11] The Pacejka "Magic Formula" tire models - Wikipedia.
https://en.wikipedia.org/wiki/Hans_B._PacejkaThe_Pacejka_%22Magic_Formula%22_tire_models
(viewed Sep 2017)

[12] ocornut - GitHub. ImGui. https://github.com/ocornut/imgui

[13] OpenGL. https://www.opengl.org/

[14] GLEW. http://glew.sourceforge.net/

[15] GLFW. http://www.glfw.org/

[16] SOIL. http://www.lonesock.net/soil.html

[17] GLM. https://glm.g-truc.net/0.9.8/index.html

[18] Desmos https://www.desmos.com/calculator (viewed March 2018)