3D Car Physics Simulation Focused on Games

Author: Tiyagora Yetalit

March, 2025

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

T_{wheel} : Torque of any wheel

r_{wheel} : Radius of any wheel

$F_{traction}$: Traction force

V : Linear speed

C_{brake} : Constant brake force

F_{rr} : Rolling resistance

F_{drag} : Aerodynamic drag

C_{drag} : Aerodynamic drag constant

a : Linear acceleration

M : Mass of the car

Δt : Time increase between calls to the physics engine

α_{wheel} : Slip angle of any wheel

ω : Angular velocity

t_f : Front width of the car

δ_{wheel} : Angle of any wheel

$F_{latwheel}$: Lateral force of any wheel

b : Distance from the center of gravity of the front wheels

c : Distance from the center of gravity of the rear wheels

C_{cs} : Roll stiffness constant

T_{net} : Total torque of the car

α : Angular acceleration

I : Moment of inertia

Ω_e : Engine speed

g_k : ratio of the wheel for the current gear

G_0 : Final gear ratio

T_e : Engine torque

η : Transmission loss

TP : Throttle position

h : Height of the center of gravity

G : Gravitational acceleration

ΔW_{long} : Longitudinal weight transfer

ΔW_{lat} : Lateral weight transfer

LIST OF FIGURES

ABSTRACT

Keywords: 3D, Car Physics, Simulation, Game, Unity

In this study, different **3D** and **vehicle dynamics** concepts are discussed in an attempt to simulate car physics. The fact that the software developed is focused on games means that the software and the research conducted have tried to take into account topics that can be useful for other fields.

For a 3D car, **acceleration**, **braking**, **cornering**, **gear shifting**, **weight** for a 3D car **transfer** and **suspension system** are discussed.

The **Unity** game engine was preferred for easier implementation of the research. However, in order to make the study more inclusive, except for collision detection and collision response, the other calculations were done from the scratch. Since collision situations are very extensive and not the focus of this study, the physics engine used is given the final **linear and angular velocity** of the car and an interactive 3D simulation can be achieved.

In this way, game developers can use the parameters of their favorite car to experience a 3D car with the characteristics of a car.

PART 1. INTRODUCTION

In this report, I have shared my research on simulating a car in a 3D environment using the parameters of a car, followed by my software work where the research is applied to games. To simulate the movements of a car, it is necessary to study the vehicle dynamics and the physics phenomena that take place.

1.1. Vehicle Dynamics and Physics

In order to simulate acceleration, braking, braking, cornering, gear shifting, weight transfer and suspension, it is necessary to analyze both how the car works and the physical forces involved in the movement of the car.

In the first chapters, the total force and torque that a car is subjected to when moving is described. Calculating the total force and torque means calculating the linear and angular accelerations of the car and the linear and angular velocity due to these accelerations. The velocity information tells us how much an object will change position and direction in a certain amount of time, and with this information, we can show what position and direction the car has in each frame and make a 3D simulation.

In other chapters, we have talked about the cycles that the car engine makes inside the car and the force of motion that is generated as a result of the rotation of the wheels from the gear system to the wheels. In order to understand the logic of this force and the transmission system, the necessary vehicle dynamics topics have been studied.

1.2. Software Application

Unity game engine was used to transfer the knowledge obtained from the research to a 3D environment with a focus on games. Unity was chosen because I thought that the application could be implemented faster in this environment. The codes were written in C# language supported by Unity. Here, the *Raycasting* technique is used and the final linear and angular velocity of the car for collisions is given to the *RigidBody* component. Users interested in other 3D environments should use their equivalents in that environment instead of Unity's *Raycast* and *RigidBody* techniques. Also, the directions in the equations are set according to Unity's left hand rule:

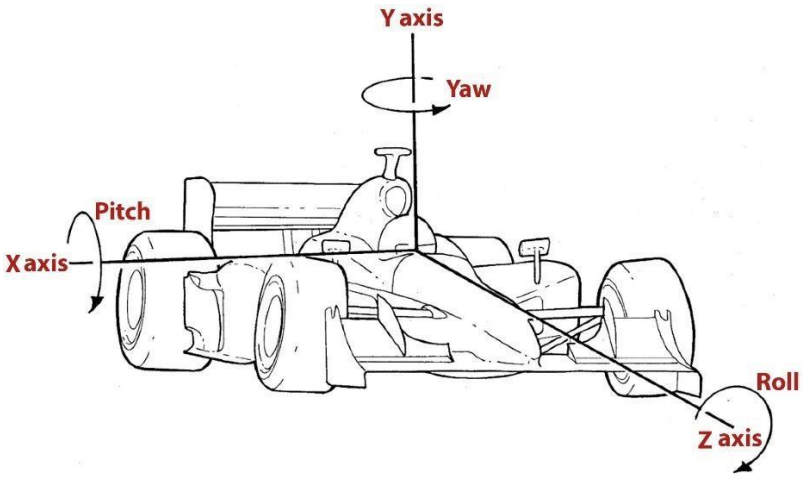


Figure 1.1.1 left hand rule. Taken and modified from [7]

1.3. Similar and Motivational Studies

Much work has been done in this area, mostly commercial; from realistic car games to software for car manufacturers to simulate their own cars. There are also free works such as *Racer v0.9.0*[1].

Figure 1.1.2 Racer simulator

Here, I would like to mention two works that gave direction and motivation to my research: Car Physics for Games by Marco Monster[2] and Development of a car physics engine for games by Punyawee Srisuchat[3].

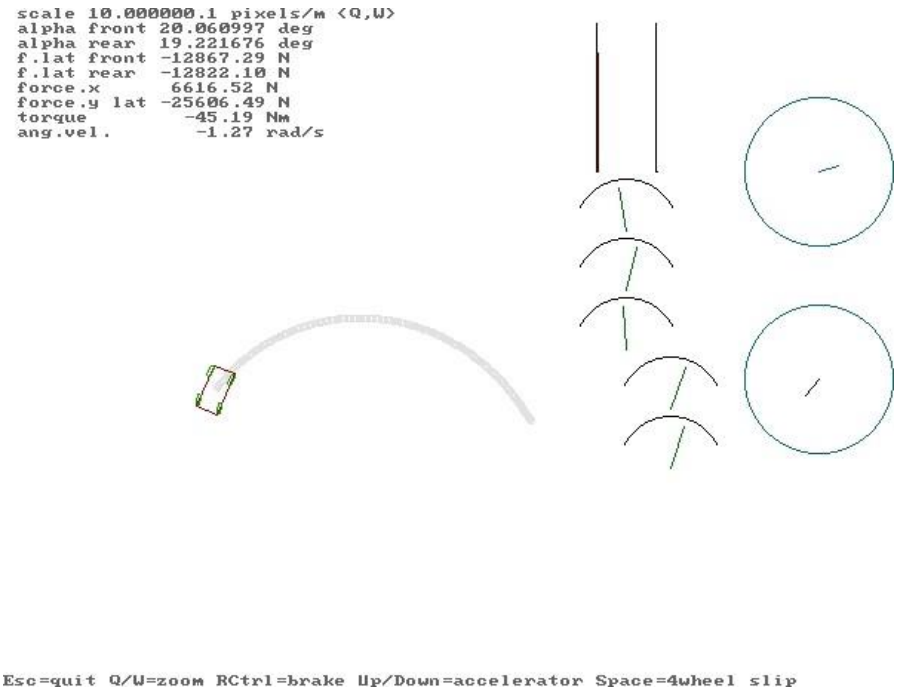


Figure 1.1.3 Marco Monster's 2D car simulation

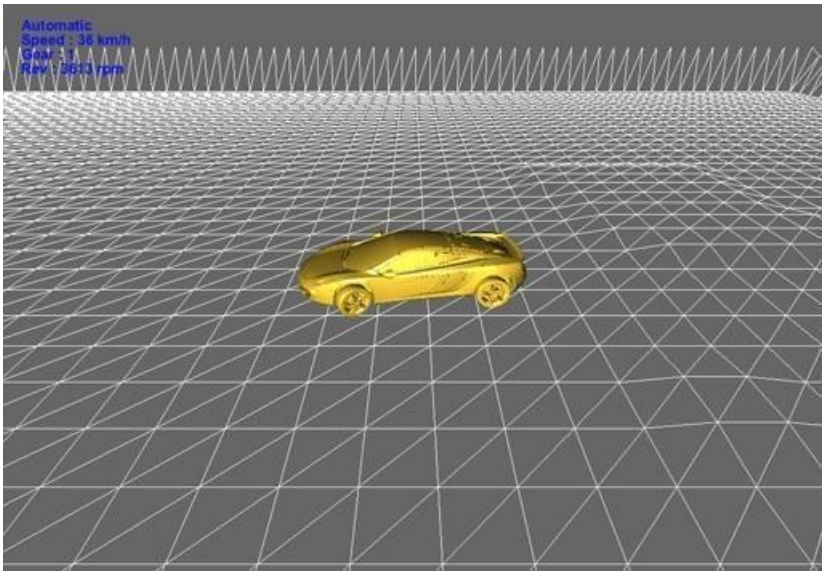


Figure 1.1.4 McLaren scene from Development of a car physics engine for games

1.4. Project Cost

Unity Game Engine License Prices:

- Unity Personal Free of charge
- Unity Plus: from $2,200.00 per year
- Unity Industry: from $4,950.00 per year

Note: Prices for the first quarter of 2025.

CHAPTER 2. FORWARD/BACKWARD MOVEMENT

In this section, the **longitudinal** forces that move a car forward or backward, independent of whether it is cornering or not, are discussed. Here, in order to get realistic results, it is necessary to perform complex analyses due to different situations, such as the physical characteristics of the car tire. For the software in the study, more general motion physics concepts are considered.

2.1. Traction Force

When we think of the forces that make a vehicle move forward/backward, the first thing that comes to mind is the force of the engine. However, the force that moves a car is not directly the force of the engine. In cars, the force provided by the engine is used to turn the wheels (this is discussed in the following sections). As the wheels try to turn, they exert a force in the opposite direction on the ground with which they are in contact. The ground, due to the friction between it and the wheels, exerts a force in the direction of the car movement. This is called **the traction force**. In ideal cases, this force and the force exerted by the wheels are of the same size and parallel to the ground.

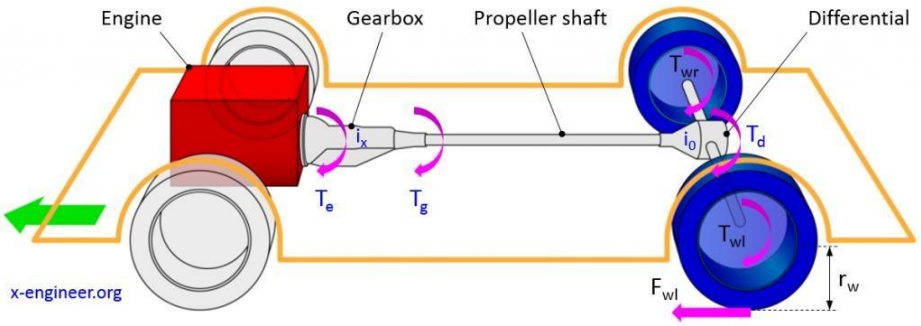


Figure 2.1.1 Rear-wheel drive (RWD) powertrain schematic[4]

After the torque of the engine passes through the different sections, it is distributed to the wheels by the differential. If we assume that the distribution is equal in a RWD car as in Figure 2.1.1, we can calculate the torque of each wheel in this way[4] (Equation 2.1.1):

(2.1.1)

$$T_{left\ wheel} = T_{right\ wheel} = \frac{T_{differential}}{2}$$

Using the formula for torque (Equation 2.1.2):

(2.1.2)

$$T = r \times F = rFsin(\theta)$$

T: torque or rotational force (N.m), r: radius (m), F: force (N), θ: angle between F and r

Since θ=90°, we can calculate the tractive force of each wheel using the following formula (Equation 2.1.3):

$$F_{traction} = T_{wheel} / r_{wheel} \quad \textbf{(2.1.3)}$$

$F_{traction}$: tractive force (N), T_{wheel}: wheel torque (N.m), r_{wheel}: wheel radius (m)

There is a maximum traction force that the wheels can generate. We can calculate by knowing the coefficient of friction (Equation 2.1.4):

(2.1.4)

$$F_{maxtraction} = \mu_t W$$

$F_{maxtraction}$: maximum tractive force (N), μ_t: coefficient of friction, W: weight (N)

If the wheel turns fast enough to exceed this limit, it will start to slip.

2.2. Braking

A simple model can be used for a car brake. When the user presses the brake, instead of a traction force, a force opposite to the direction of motion of the car can be applied (Equation 2.1.5):

(2.1.5)

$$F_{traction} = -sgn(V_z) * C_{brake}$$

$F_{traction}$: traction force (N), V_z: longitudinal speed (m/s), C_{brake}: brake force constant (N)

2.3. Rolling Resistance

Wheels lose energy as they roll on the ground. The most important force that causes this is called **rolling resistance**. Test standards such as **SAE J2452** exist for calculating this force (Equation 2.1.6):

(2.1.6)

$$F_{rr} = P^{\alpha}\ Z^{\beta}\ (a\ +\ bV\ +\ cV^2)$$

Instead of this detailed formula, we can use the formula obtained using values close to the actual rolling coefficients for air-filled tires on dry roads[5] (Equation 2.1.7):

(2.1.7)

$$F_{rr} = -sgn(V)W(0.005 + (10^5\ /\ P)\ (0.01 + 0.0095\ (0.036 * V)^2))$$

F_{rr}: rolling resistance (N), W: weight (N), P: tire pressure (Pa), V: speed (m/s)

2.4. Aerodynamics Drag

The resistance of the air to the motion of the car is called **aerodynamic drag**. The general formula for this force is as follows (Equation 2.1.8):

(2.1.8)

$$F_{drag} = -\frac{1}{2} * C_d * A * \rho * V * |V|$$

F_{drag}: aerodynamic drag (N), C_d: drag coefficient, A: cross-sectional area (m²), ρ: liquid density (kg/m³), V: velocity (m/s)

We can use a simpler form for our software (Equation 2.1.9):

(2.1.9)

$$F_{drag} = -C_{drag} * V * |V|$$

F_{drag}: aerodynamic drag (N), C_{drag}: aerodynamic drag constant, V: speed (m/s)

While rolling resistance is more effective at low speeds, as speed increases, aerodynamic drag becomes more effective and is one of the factors that determine the maximum speed a car can reach.

2.5. Net Force

Applying the forces mentioned throughout this chapter, we can calculate the net force for the forward/reverse motion of a car as follows (Equation 2.2.1):

(2.2.1)

$$F_{net} = F_{traction} + F_{rr} + F_{drag}$$

Using Newton's 2nd law, we can calculate the linear acceleration of the car (Equation 2.2.2):

(2.2.2)

$$a = \frac{F_{net}}{M}$$

a: linear acceleration (m/s²), M: mass of the car (kg)

With the acceleration information we can calculate the new linear velocity and eventually the new position (Equations 2.2.3 and 2.2.4):

(2.2.3)

$$V = a * \Delta t + V_0$$

(2.2.4)

$$X=V_{world}*\Delta t+X_0$$

V_{world}: new speed relative to world coordinates (m/s), Δt: time increment between calls to physics engine (s), V_0: current speed (m/s), X: new position (m), X_0: current position (m)

The conversion of the new speed into world coordinates is shown at the end of the next section.

If you want to use the information so far to move your car forward or backward, you can (for now) replace T_{wheel} with a fixed value of your choice. You can influence the direction of movement with the sign of the torque.

CHAPTER 3. CORNERING

One of the most influential details of car simulations is the physics of cornering. In this section, the **lateral force** and angular acceleration due to the change in steering angle are discussed. The angular acceleration will show how much the car angle will change.

3.1. Slip Angle

When a car corners, it does not make the same angle change as the steering angle. The difference between these two angles is called the **slip angle**.

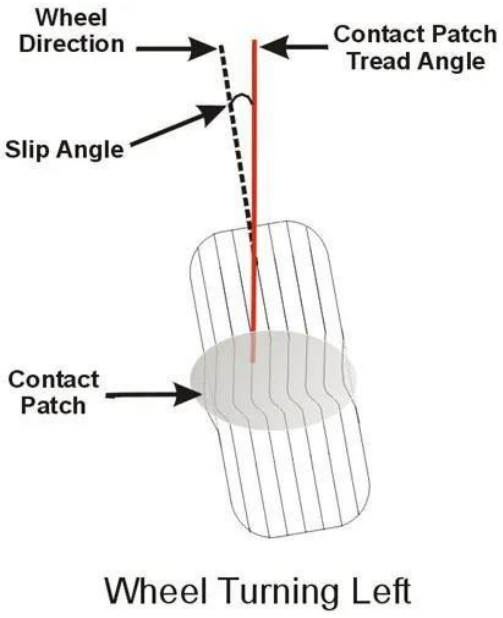


Figure 3.1.1 left-turning wheel[6]

We can calculate the slip angle of each wheel as follows[7] (Equation 3.1.1):

(3.1.1)

$$\alpha_{11} = \delta_{11} * sgn(V_z) - arctan((V_x + b * \omega_y)/(|V_z| - t_f * \frac{\omega_y}{2}))$$

$$\alpha_{12} = \delta_{12} * sgn(V_z) - arctan((V_x + b * \omega_y)/(|V_z| + t_f * \frac{\omega_y}{2}))$$

$$\alpha_{21} = -arctan((V_x - c * \omega_y)/(|V_z| - t_r * \frac{\omega_y}{2}))$$

$$\alpha_{22} = -arctan((V_x - c * \omega_y)/(|V_z| + t_r * \frac{\omega_y}{2}))$$

α_{11}: left front wheel slip angle (rad), δ_{11}: left front wheel angle (rad), V_z: longitudinal velocity (m/s), V_x: lateral velocity (m/s), b: distance of front wheels from center of gravity (m), ω_y: angular velocity of car in y-axis (rad/s), t_f: front width of car (m), α_{12}: right front wheel slip angle (rad), δ_{12}: right front wheel angle (rad), α_{21}: left rear wheel slip angle (rad), c: distance of rear wheels from center of gravity (m), t_r: rear width of car (m), α_{22}: right rear wheel slip angle (rad)

In the demo application, the slip angle information for the front and rear wheels is stored in *Vector2* variables:

```
Vector2 slip_angle_f = Vector2.zero;
Vector2 slip_angle_r = Vector2.zero;

slip_angle_f.x = car.steerAngle * Mathf.Sign(velocity.z) -
 Mathf.Atan((velocity.x + car.angularVelocity.y * car.b) /
 (Mathf.Abs(velocity.z) - car.track * car.angularVelocity.y / 2));

slip_angle_f.y = car.steerAngle * Mathf.Sign(velocity.z) -
 Mathf.Atan((velocity.x + car.angularVelocity.y * car.b) /
 (Mathf.Abs(velocity.z) + car.track * car.angularVelocity.y / 2));

slip_angle_r.x = -Mathf.Atan((velocity.x - car.angularVelocity.y
 * car.c) / (Mathf.Abs(velocity.z) - car.track *
 car.angularVelocity.y / 2));

slip_angle_r.y = -Mathf.Atan((velocity.x - car.angularVelocity.y
 * car.c) / (Mathf.Abs(velocity.z) + car.track *
 car.angularVelocity.y / 2));
```

X components are used for the left wheels and Y components are used for the right wheels.

3.2. Lateral Force

We need to model how much lateral force the slip angle causes. Different models and formulas have been proposed for this. The **Pacejka Magic Formula** is one of the most well-known models. In the figure below, you can see the variation of the lateral force according to the slip angle:

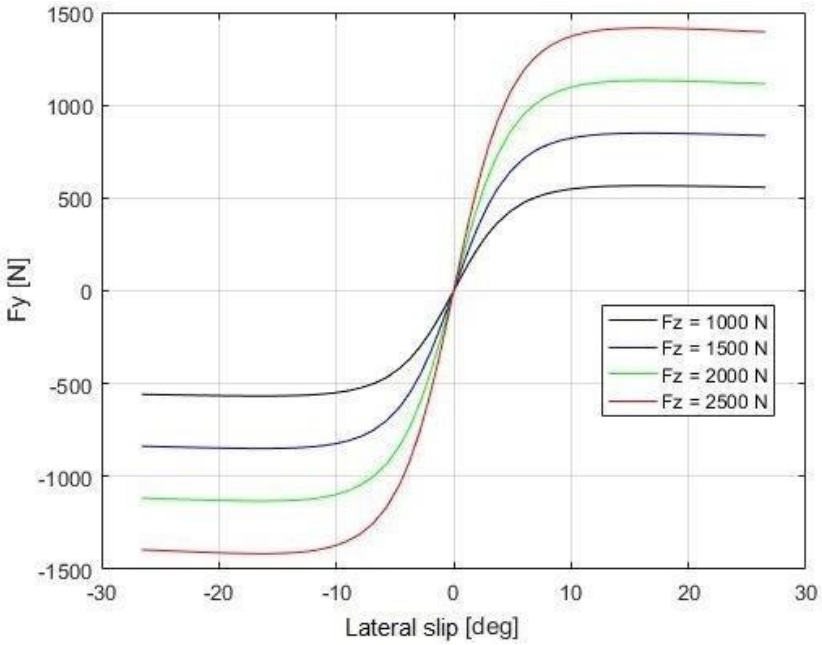


Figure 3.1.2 Lateral force according to slip angle[8]

The Pacejka model has a non-linear curve. But we can use a simpler, linear form. We can assume that the curve is a straight line from zero to the point where its change decreases. We can adjust the slope of this line with a constant called **Cornering Stiffness**. After the point where its variation decreases, we can assume that the curve does not change at all and continues at the same constant value.

In this way, to calculate the lateral force, we first multiply the slip angle by the cornering stiffness constant (Equation 3.1.2):

(3.1.2)

$$\mu_{wheel} = \alpha_{wheel} * C_{cs}$$

α: slip angle (rad), C_{cs}: cornering stiffness constant (1/rad)

We limit the value we obtained to the maximum friction constant of the wheel:

```
Mathf.Clamp(μ_wheel, -maxGrip, maxGrip)
```

We multiply the result by the weight on the wheel (Equation 3.1.3):

(3.1.3)

$$F_{lat\ wheel} = \mu_{wheel} * W_{wheel}$$

$F_{latwheel}$: lateral force of the wheel (N), W_{wheel}: weight on the wheel (N)

3.3. Linear and Angular Acceleration

Including the longitudinal force calculated in the previous section, we can calculate the final net force as follows[7] (Equation 3.1.4):

(3.1.4)

$$F_z = F_{traction} + F_{rr_z} + F_{drag_z} - F_{lat\ 11}\ sin(\delta_{11}) - F_{lat\ 12}\ sin(\delta_{12})$$

$$F_x = F_{rr_x} + F_{drag_x} + F_{lat\ 11}\ cos(\delta_{11}) + F_{lat\ 12}\ cos(\delta_{12}) + F_{lat\ 21} + F_{lat\ 22}$$

F_z: net longitudinal force (N), F_x: net lateral force (N)

Due to the lateral force on the wheels, not only does the car move sideways, but also its angle in the y-axis changes. To calculate the angle of change, we can first calculate the torque of the car in the y-axis in this way[7] (Equation 3.1.5):

(3.1.5)

$$T_y = (F_{lat\ 11}\ cos(\delta_{11}) + F_{lat\ 12}\ cos(\delta_{12})) * b - (F_{lat\ 21} + F_{lat\ 22}) * c + ((F_{lat\ 11}\ sin(\delta_{11}) - F_{lat\ 12}\ sin(\delta_{12})) * t_f/2$$

T_y: torque of the car in the y-axis (N.m), b: distance of the front wheels from the center of gravity (m), t_f: front width of the car (m), c: distance of the rear wheels from the center of gravity (m)

To calculate angular acceleration, we can use the application of Newton's 2nd law to rotational motion (Equation 3.1.6):

(3.1.6)

$$T_{net} = I\alpha$$

T_{net}: total torque of the car (N.m), α: angular acceleration (rad/s²), I: inertia of the car moment (kg.m²)

The moment of inertia of the car can be obtained either from readily available data or by yourself. For simplicity, we can compare the car to a rectangular prism and calculate it as follows (Equation 3.1.7):

(3.1.7)

$$I = \begin{bmatrix} \frac{1}{12}m(h^2 + l^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2 + l^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2 + h^2) \end{bmatrix}$$

m: mass of the car (kg), h: height (m), l: length (m), w: width (m)

Here, considering the values on the diagonal, we will consider the moment of inertia as a 3D vector and calculate the angular acceleration (Equation 3.1.8):

(3.1.8)

$$\vec{\alpha} = (T_{net_x} / I_x, T_{net_y} / I_y, T_{net_z} / I_z)$$

$\vec{\alpha}$: angular acceleration vector (rad/s²)

We can calculate the angular velocity in a similar way to the linear velocity (Equation 3.1.9):

(3.1.9)

$$\omega = \alpha * \Delta t + \omega_0$$

ω: new angular velocity (rad/s), Δt: time increment between calls to the physics engine (s), ω_0: current angular velocity (rad/s)

As shown earlier (Equations 2.2.2 and 2.2.3), the linear acceleration and velocity are also calculated.

The linear and angular velocity we obtain can be given to *the RigidBody* as follows:

```
Vector3 velocity_world = car_rotation * velocity;
rigidBody.linearVelocity = velocity_world;
rigidBody.angularVelocity = angular_velocity;
```

RigidBody receives the new velocity information and runs its own collision engine. Here, due to collision situations that may arise, other negative or positive velocities may be added to the linear and angular velocity of the RigidBody, and the final velocity values may differ from what we expect.

CHAPTER 4. VEHICLE DYNAMICS

In the previous chapters, the focus was more on physics. In this section, we focus more on the inside of a car and try to explain vehicle dynamics such as the engine, transmission system and suspension system.

4.1. Engine Speed and Gear System

The number of rotations/revolutions made by the crankshaft in the engine of the car in 1 minute is called the engine speed. This rotation turns the wheels starting from the engine and going all the way to the wheels and causes torque (T_{wheel}) on the wheels.

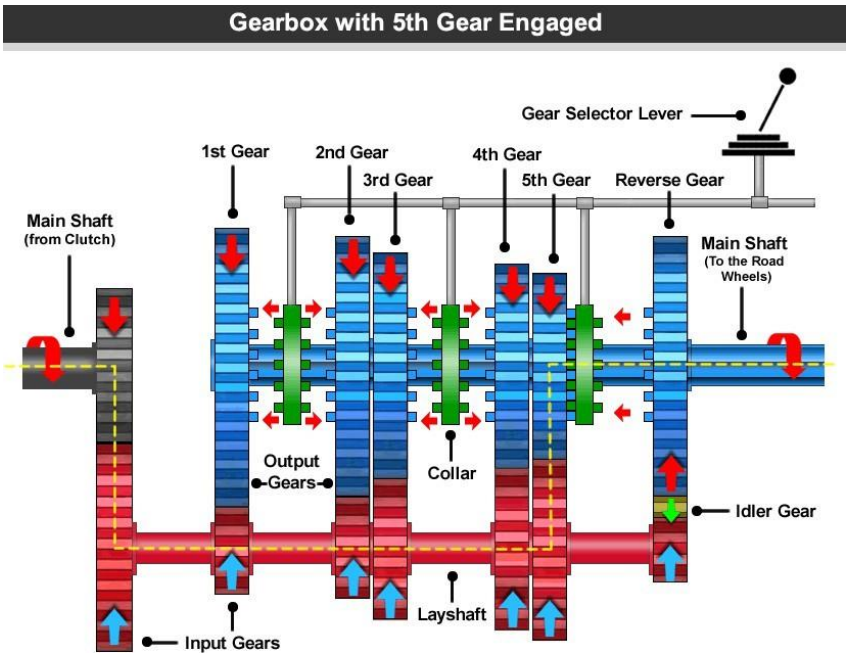


Figure 4.1.1 Wheels in a gearbox[9]

Assuming that the car is on the ground and the wheels do not slip, we can calculate the engine speed as follows (Equation 4.1.1):

(4.1.1)

$$\Omega_e = V_z * 60 * g_k * G_0 \,/\, (2\pi * r_{wheel})$$

Ω_e: engine speed (rpm), V_z: longitudinal speed (m/s), g_k: ratio of the wheel to the current gear, G_0: final gear ratio, r_{wheel}: wheel radius (m)

Engine speed has a certain **redline**. The higher the speed, the closer the engine gets to this number. Once the engine reaches this redline, the car will not be able to gain any more speed. Here, in order to reduce the engine speed, it is necessary to shift to a higher gear.

As can be seen in Figure 4.1.1, in a higher gear, the ratio of the wheel, i.e. g_k, is smaller. Therefore the engine speed is reduced.

For a car with an automatic transmission system, the maximum speed it can reach in each gear is calculated according to a maximum engine speed value you set yourself. When it reaches these speeds, it has to shift to a higher or lower gear, depending on the gear it is currently in. The maximum speed each gear can reach (Equation 4.1.2):

(4.1.2)

$$V_{max} = \Omega_{e\,max} * 2\pi * r_{wheel} \,/\, (60 * g_k * G)$$

V_{max}: maximum longitudinal speed (m/s), $\Omega_{e\,max}$: maximum engine speed (rpm)

4.2. Engine Torque

The torque value caused by engine speed is called **engine torque**. As explained earlier, this torque passes through different impellers and reaches the wheels. Here, we can use the torque **curve** of the car we are simulating to calculate the engine torque depending on the engine speed. As an example, you can see the torque graph of a 2004 Porsche Boxster S car:

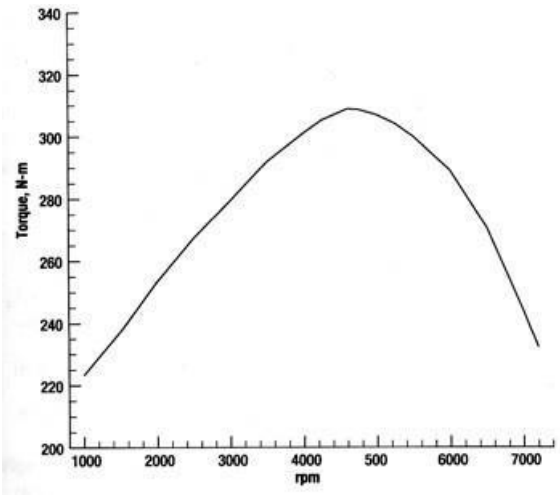


Figure 4.1.2 2004 Boxster S torque graph[3]

We can simplify the graph here to a few lines:

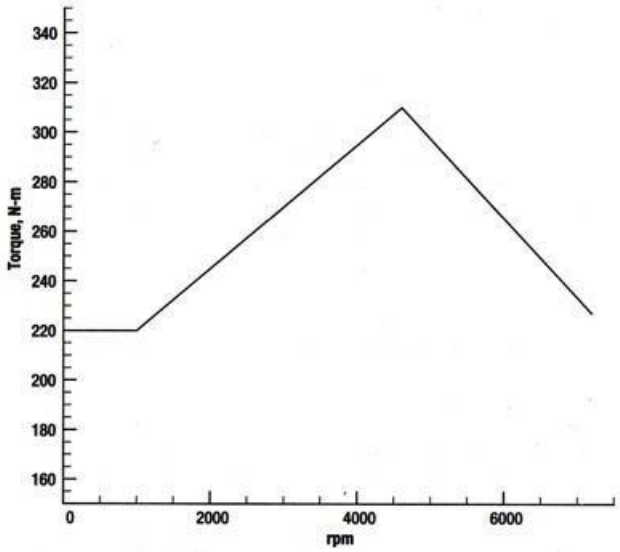


Figure 4.1.3 Simplified 2004 Boxster S torque plot[3]

Finally, once we have our motor torque value, we can calculate the T_{wheel} value as follows (Equation 4.1.3):

(4.1.3)

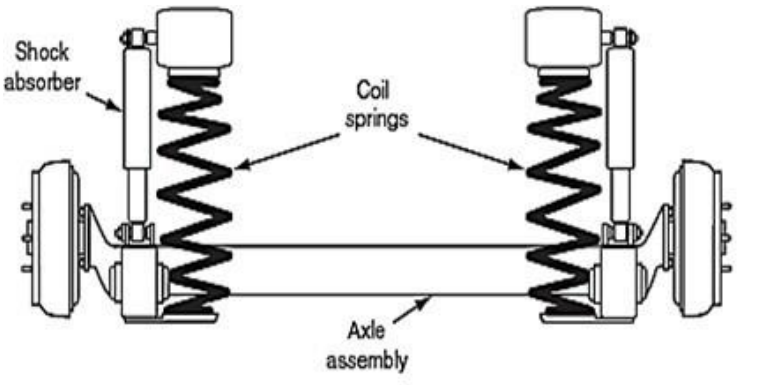
$$T_{wheel} = T_e * g_k * G_0 * (1 - \eta) * TP$$

T_e: engine torque (N.m), g_k: gear ratio of the current gear, G_0: final gear ratio, η: drivetrain loss, TP: throttle position

η and TP values should be between 0 and 1. Transmission loss represents the torque loss of the engine torque until it reaches the wheels.

4.3. Suspension System

For car suspension, we can design a simple system consisting o f a spring and a damper/shock absorber:



Semi-independent rear suspension system with coil springs and shock absorbers mounted separately.

Figure 4.1.4 Suspension with spring and shock absorber[10]

The change in height experienced by the wheels causes tension or compression of the spring. According to the change of the spring length, we can calculate the resulting force by Hooke's law (Equation 4.1.4):

(4.1.4)

$$F_s = k*\Delta x$$

F_s: spring force (N), k: spring stiffness (N/m), Δx: change in length (m)

We can apply the resulting force to the car from the point where the spring starts. But for a good driving experience, we also need a shock absorber mechanism to counteract the harsh forces that can occur.

We want the shock absorber to apply a force that depends on the speed of the car in the vertical direction at the point where the suspension force is applied. The higher the velocity of this part of the car chassis, the stronger the shock absorber will resist. Consequently, we can calculate the shock absorber force as follows (Equation 4.1.5):

(4.1.5)

$$F_d = c * (\vec{V}_{point} \cdot \vec{\mathrm{U}}_{point})$$

F_d: shock absorber force (N), c: shock absorber stiffness (N.s/m), $\vec{V}_{point}$: velocity vector of the point (m/s), $\vec{\mathrm{U}}_{point}$: unit vector indicating the upward direction of the point

The product between two vectors is **Dot Product.** The reason for this operation is that we are trying to find the velocity perpendicular to the car.
We can calculate the final suspension force as follows (Equation 4.1.6):

(4.1.6)

$$F = F_s - F_d$$

For the suspension system, the *Raycasting* technique, common in 3D environments, can be used. Using one ray for each wheel, we can find the distance from the start of the spring to the ground. Here, parameters like the **initial length** and the **maximum length variation** of the spring need to be determined beforehand.

By adding the initial length, the maximum length change and the radius of the wheel, you can determine **the maximum length of the ray**. If it does not hit any ground during this distance, then the wheel has no contact with the ground and you do not need to apply any suspension force.
If it hits the ground, the distance to the point of impact minus the radius of the wheel will be the length of the spring. According to the length here, you can calculate the force exerted by the spring with Equation 4.1.4.

4.4. Weight Transfer

When a car is moving, due to acceleration, it has a different weight distribution from the initial state. When this distribution is unbalanced, the car starts to lean towards the sides where the weight is greater.

There are two types of weight transfer: longitudinal and lateral weight transfer. Longitudinal weight transfer occurs when the car accelerates forward or rearward: (arrows indicate the direction of weight change)

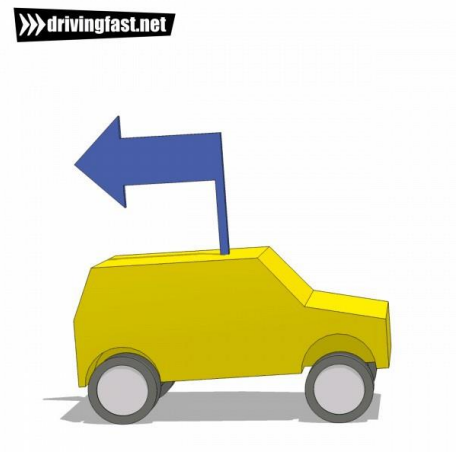


Figure 4.1.5 Forward acceleration[11]

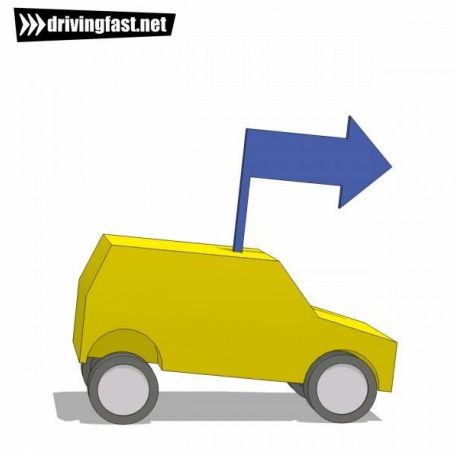


Figure 4.1.6 backward acceleration[11]

Lateral weight transfer occurs when the car is cornering:

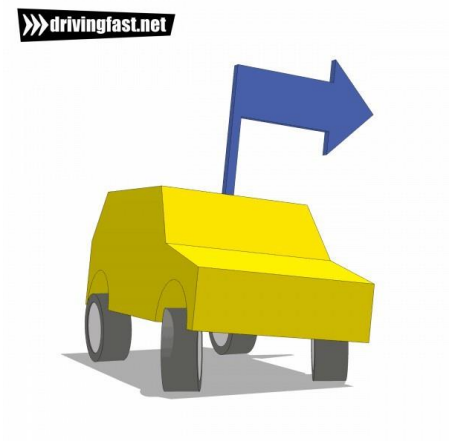


Figure 4.1.7 Lateral acceleration[11]

First, we need to calculate the weight on each wheel in the initial state:

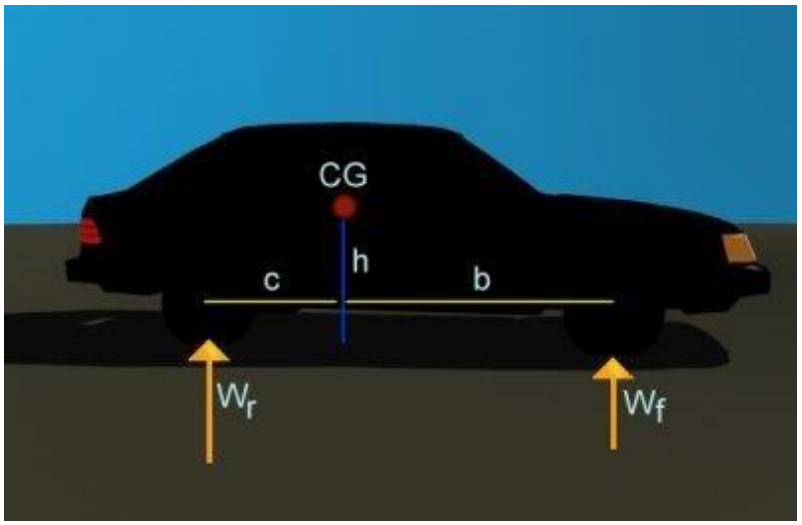


Figure 4.1.8 weight distribution[2]

The weight on each wheel can be calculated as follows (Equation 4.1.7):

(4.1.7)

$$W_f = c/(b+c) * W/2$$
$$W_r = b/(b+c) * W/2$$

W_f: weight of a front wheel (N), c: distance of the rear wheels from the center of gravity (m), b: distance of the front wheels from the center of gravity (m), W: weight of the car (N), W_r: weight of one rear wheel (N)

We can then apply the longitudinal weight transfer to the weight of each wheel (Equation 4.1.8):

(4.1.8)

$$\Delta W_{long} = \frac{h}{b+c} * M * a_z$$
$$W_f \mathrel{-}= \Delta W_{long} \,/\, 2$$
$$W_r \mathrel{+}= \Delta W_{long} \,/\, 2$$

h: height of the center of gravity (m), M: mass of the car (kg), a_z: longitudinal acceleration (m/s²)

Finally, we can apply the lateral weight transfer (Equation 4.1.9):

(4.1.9)

$$\Delta W_{lat_r} = (a_x \,/\, G) * (W_{r_1} + W_{r_2}) * h \,/\, t$$
$$\Delta W_{lat_f} = (a_x \,/\, G) * (W_{f_1} + W_{f_2}) * h \,/\, t$$

$$W_{r_1} \mathrel{+}= \Delta W_{lat_r}$$
$$W_{r_2} \mathrel{-}= \Delta W_{lat_r}$$

$$W_{f_1} += \Delta W_{lat_f}$$
$$W_{f_2} -= \Delta W_{lat_f}$$

a_x: lateral acceleration (m/s²), G: acceleration of gravity (m/s²), W_{r1}: weight of the rear left wheel (N), W_{r_2}: weight of the rear right wheel (N), t: width of the car (m), W_{f_1}: weight of the front left wheel (N), W_{f2}: weight of front right wheel (N)

We can apply the final weight of each wheel as a force to represent the force of gravity on the car. We should always set the forces downwards and relative to the points where the wheels are located.

In the demo, a *float* array is used to hold the weight of each wheel:

```
float[] load = new float[4]; // FL, FR, RL, RR
load[0] = load[1] = (car.c / (car.b + car.c)) * car.mass * 9.8f / 2;
load[2] = load[3] = (car.b / (car.b + car.c)) * car.mass * 9.8f / 2;
```

Once the weight transfer calculations are done, we can set a maximum value to limit the longitudinal and lateral weight transfers:

```
float Wt_acc = car.transform.position.y / (car.b + car.c) *
               car.mass * acc.z;

Wt_acc = Mathf.Clamp(Wt_acc, -max_long_wt, max_long_wt);

load[0] -= Wt_acc / 2;
load[1] -= Wt_acc / 2;
load[2] += Wt_acc / 2;
load[3] += Wt_acc / 2;

float Wt_lat_r = (acc.x / 9.8f) * (load[2] + load[3]) *
                 car.transform.position.y / car.track;

float Wt_lat_f = (acc.x / 9.8f) * (load[0] + load[1]) *
                 car.transform.position.y / car.track;

Wt_lat_r = Mathf.Clamp(Wt_lat_r, -max_lat_wt, max_lat_wt);

Wt_lat_f = Mathf.Clamp(Wt_lat_f, -max_lat_wt, max_lat_wt);

load[0] += Wt_lat_f;
load[1] -= Wt_lat_f;
load[2] += Wt_lat_r;
load[3] -= Wt_lat_r;
```

max_long_wt and *max_lat_wt* determine the maximum longitudinal and lateral weight transfers respectively.

CHAPTER 5. APPLICATION

In this section, a demo application in Unity environment and C# language is presented using the theoretical knowledge described. Here, in order to improve portability to other 3D environments, written utility functions and information that can help the transition from theory to practice can be useful.

Figure 5.1.1 demo application

In the scene, ready-made 3D models were used[12][13].

5.1. Overview

The overall structure of the application is shown in the UML diagram below:

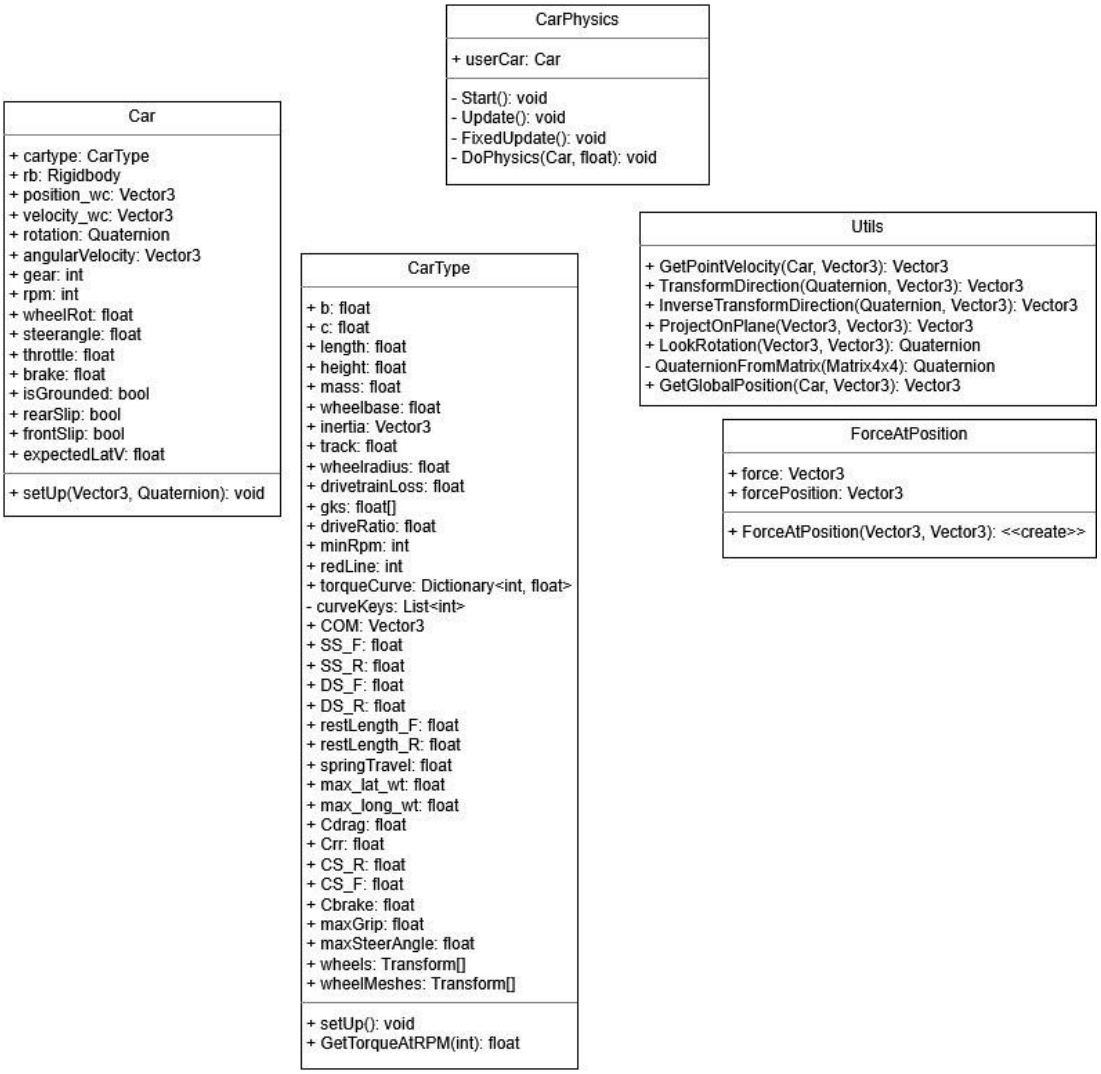


Figure 5.1.2 UML diagram

- *CarPhysics:* It is the main part of the designed physics engine. Most physics calculations and operations of other parts are managed here.
- *Car*: Represents each car object at runtime. It focuses on the dynamic information of the car.
- *CarType*: Holds information about the structure of each car. It represents the car in an abstract way.
- *Utils*: Helper functions and structures are located here.
- *ForceAtPosition*: Holds information about the forces to be applied. It can be seen as part of *Utils*.

5.2. Utility Functions

- *TransformDirection*: Finding the world coordinate equivalent of a vector relative to the local coordinate of the car:

```
public static Vector3 TransformDirection(Quaternion rotation,
Vector3 localDirection)
{
```

```
    // Apply the object's rotation to the local direction
    return rotation * localDirection;
}
```

- *InverseTransformDirection*: Finding the equivalent of a vector relative to the world coordinate in the local coordinate of the car:

```
public static Vector3 InverseTransformDirection(Quaternion rotation, V
ector3 worldDirection)
{
    // Apply the inverse of the object's rotation to the world
    // direction
    return Quaternion.Inverse(rotation) * worldDirection;
}
```

- *GetGlobalPosition*: Find the world coordinate equivalent of a position relative to the local coordinate of the car:

```
public static Vector3 GetGlobalPosition(Car car,
Vector3 LocalPosition)
{
    // Add car's world position to get the global position
    return car.position_wc + TransformDirection(car.rotation,
    LocalPosition);
}
```

- *GetPointVelocity*: Find the velocity of a point relative to a car:

```
public static Vector3 GetPointVelocity(Car car, Vector3 point)
{
    // Get the position vector relative to the object's center of
    // mass
    Vector3 r = point - GetGlobalPosition(car, car.cartype.COM);
    // Calculate the point velocity
    return car.velocity_wc + Vector3.Cross(car.angularVelocity, r);
}
```

- *ProjectOnPlane*: The projection of a vector onto a plane:

```
public static Vector3 ProjectOnPlane(Vector3 v, Vector3 normal)
{
    // Make sure the normal is not zero to avoid division by zero
    if (normal == Vector3.zero)
    {
        return Vector3.zero;
    }
    // Normalize the normal to ensure we are projecting onto a unit
    // normal
    normal.Normalize();
    // Compute the projection of v onto the normal
    float dotProduct = Vector3.Dot(v, normal);
    // Subtract the projection from v to get the component of v on
    // the plane
    Vector3 projection = v - dotProduct * normal;
    return projection;
}
```

5.3. Details About Forces

To manage the forces to be applied, a list structure can be used:

```
List<Utils.ForceAtPosition> forces= new List<Utils.ForceAtPosition>();
```

The ForceAtPosition structure holds the vector of the force and the point at which it is applied. If a force is applied at a point different from the center of mass of the car, it can cause angular acceleration. To calculate the torque here, a loop like the one below can be used:

```
foreach (var forceInfo in forces)
{
    // Calculate the position of force application relative to the
    // center of mass
    Vector3 forceOffset = forceInfo.forcePosition -
    Utils.GetGlobalPosition(car, car.cartype.COM);
    // Add the force to the total force
    totalForce += forceInfo.force;
    // Add the torque (r x F) to the total torque
    totalTorque += Vector3.Cross(forceOffset, forceInfo.force) *
    Mathf.Deg2Rad;
}
```

This is particularly t h e case when applying **gravitational** and **suspension** forces. For the application of these forces, the following operations can be performed:

```
Quaternion hit_RL = Quaternion.identity;
Quaternion hit_RR = Quaternion.identity;
int groundedWheels = 0;
for (int i = 0; i < 4; i++)
{
    forces.Add(new Utils.ForceAtPosition(
    -Vector3.up * load[i], rayPoints[i]));

    RaycastHit hit;

    float maxLength;
    if (i < 2)
    {
        maxLength = car.cartype.restLength_F +
                    car.cartype.springTravel;
    }
    else
    {
        maxLength = car.cartype.restLength_R +
                    car.cartype.springTravel;
    }

    Vector3 pointUp = car.rotation * Vector3.up;
    if (Physics.Raycast(rayPoints[i],
        -pointUp, out hit, maxLength + car.cartype.wheelradius))
    {
        groundedWheels++;
        car.cartype.wheels[i].position = rayPoints[i] + pointUp *
```

```
            (-hit.distance + car.cartype.wheelradius);

            float currentSpringLength = hit.distance -
                                        car.cartype.wheelradius;

            float springVelocity = Vector3.Dot(Utils.GetPointVelocity(car,
            if (i < 2)
            {
                float springCompression = car.cartype.restLength_F -
                                          currentSpringLength;
                float dampForce = car.cartype.DS_F * springVelocity;
                float springForce = car.cartype.SS_F * springCompression;

                forces.Add(new Utils.ForceAtPosition(pointUp *
                (springForce - dampForce), rayPoints[i]));
            }
            else
            {
                Vector3 forwardOnSurface = Utils.ProjectOnPlane(
                car.rotation * Vector3.forward, hit.normal).normalized;
                if (i == 2)
                    hit_RL = Utils.LookRotation(forwardOnSurface,
                             hit.normal);
                else
                    hit_RR = Utils.LookRotation(forwardOnSurface,
                             hit.normal);

                float springCompression = car.cartype.restLength_R -
                                          currentSpringLength;
                float dampForce = car.cartype.DS_R * springVelocity;
                float springForce = car.cartype.SS_R * springCompression;

                forces.Add(new Utils.ForceAtPosition(pointUp *
                (springForce - dampForce), rayPoints[i]));
            }
        }
        else
        {
            car.cartype.wheels[i].position = rayPoints[i] + pointUp *
                                             (-maxLength);
        }
    }
```

Apart from the gravitational and suspension forces, there are also operations for **the traction force**. Since the traction force must be applied parallel to the ground, the angle of the ground at which the rear wheels are located is calculated from the rays emitted for the suspension and inserted into the *hit_RL* and *hit_RR* variables.

5.4. Friction Model

Implementing a friction model for raycast-based wheels can be done in simple or very complex ways. In the demo application, a simple friction model on the final linear velocity is implemented:

```
velocity = Utils.InverseTransformDirection(car.rotation,
           car.rb.linearVelocity);
if ((car.brake > 0 && Mathf.Abs(velocity.z) <= groundedWheels / 4 *
    car.cartype.maxGrip * 9.8f * delta_t) ||
    Mathf.Abs(velocity.z) <= groundedWheels / 4 * car.cartype.Crr *
    9.8f * 0.008f * delta_t)
{
    velocity.z = 0;
}
if (velocity.x > 0)
{
    if (velocity.x - (car.expectedLatV + latVelocity) > 0)
    {
        if (velocity.x -
            (car.expectedLatV + latVelocity) >= groundedWheels / 4 *
            car.cartype.maxGrip * 9.8f * delta_t)
        {
            velocity.x -=
            groundedWheels / 4 * car.cartype.maxGrip * 9.8f * delta_t;
        }
        else
        {
            velocity.x -= velocity.x -
                          (car.expectedLatV + latVelocity);
        }
    }
}
else if (velocity.x < 0)
{
    if (velocity.x - (car.expectedLatV + latVelocity) < 0)
    {
        if (velocity.x -
            (car.expectedLatV + latVelocity) <= groundedWheels / 4 *
            - car.cartype.maxGrip * 9.8f * delta_t)
        {
            velocity.x += groundedWheels / 4 * car.cartype.maxGrip * 9
                          .8f * delta_t;
        }
        else
        {
            velocity.x -= velocity.x -
                         (car.expectedLatV + latVelocity);
        }
    }
}
car.rb.linearVelocity = Utils.TransformDirection(car.rotation,
                        velocity);
car.expectedLatV = velocity.x;
car.wheelRot = velocity.z / car.cartype.wheelradius;
```

Here, first, the checks for the longitudinal velocity take place. If the car is braking, we assume that the wheels on the ground are no longer rolling and stop the car if they do not have enough speed to overcome the friction force.

If it is not braking, we want them to have enough speed to overcome the rolling resistance.

For lateral velocity the situation will be slightly different. If an extra force is applied to the car that accelerates it other than the cornering, similar to the method in the "Lateral Force" section of Chapter 3, we assume that a **static frictional** force is generated sufficient to counteract this force and ignore the resulting extra speed. If the friction force has reached its maximum value, we assume that this maximum value is consistently applied as **kinetic friction** and subtract this value from the extra speed.

Finally, we put the value of the lateral velocity we expect in the next call into the *expectedLatV* variable.

You can also calculate the angular velocity of the wheel by dividing the longitudinal velocity by the radius of the wheel (*wheelRot* variable) to rotate the car wheel models according to the final longitudinal velocity.

CHAPTER 6. CONCLUSIONS AND RECOMMENDATIONS

Throughout the study, different physics and vehicle dynamics topics were addressed to simulate car physics. As a result, using real data of a car, acceleration, braking, cornering, gear shifting, weight transfer and suspension system states were simulated in such a way that they were characteristic of the car.
For example, in the demo application, a Ford Mustang racing car was simulated. The handling of the car was more stable on flat roads, but on a mountainous road, it struggled. Here, with wider tires and a suspension system, the car was able to traverse the roads more easily. However, it did not provide the same comfort on flat roads.

For this study to be used in a computer game, extra adjustments must be made depending on the expected outcome. For example, one of the techniques used in games is to move the center of mass below the car when the car is not on the ground. In this way, a better car experience can be provided to the user.
Also, improvements can be made to the applied forces. In particular, instead of the **simple friction model** used for the wheels, a more realistic model can be developed.
In the suspension system, instead of the *Raycast* technique, *Multi Raycast* or *Shapecast* techniques can be used. These techniques can give better results, but you should also be aware of their impact on the game's performance.

You can access the files of the demo application here: **https://github.com/yetalit**

SOURCES

[1] Racer. 2014. Dolphinity Racer - Car and Racing Simulator. Access link: http://www.racer.nl, Accessed: 12.03.2025.

[2] Monster, M. 2003. Car Physics for Games. Access link: https://www.asawicki.info/Mirror/Car%20Physics%20for%20Games/Car%20Physics%20for%20Games.html, Accessed: 28.02.2025.

[3] Srisuchat, P. 2012. Development of a car physics engine for games. Access link: https://nccastaff.bournemouth.ac.uk/jmacey/MastersProject/MSc 12/Srisuchat, Accessed: 02.03.2025.

[4] X-Engineer Engineering Tutorials. Access Link: https://x-engineer.org/calculate-wheel-torque-engine/, Accessed: 03.04.2025.

[5] The Engineering ToolBox. Resources, Tools and Basic Information for Engineering and Design of Technical Applications. Access link: https://www.engineeringtoolbox.com/rolling-friction-resistance- d_1303.html, Accessed: 04.04.2025.

[6] Suspension Secrets. Access link: https://suspensionsecrets.co.uk/tyre- slip-angle/, Accessed: 05.04.2025.

[7] Romañá, A. B. 2020. Analysis and design of a lateral force estimator for a Formula Student vehicle. Access link: https://upcommons.upc.edu/bitstream/handle/2117/332374/tfg-blazquezromana.pdf, Accessed: 05.04.2025.

[8] Casadio, A. 2018. MathWorks Blogs. Custom Vehicle Modeling using Simscape Language. Access link: https://blogs.mathworks.com/student-lounge/2018/02/28/vehicle-modeling- simscape-language/, Accessed: 06.04.2025.

[9] Learn Driving. Learning to drive guides, advice and tutorials. Access link: https://learndriving.tips/learning-to-drive/how-to-change-gear-in-manual-car/how-manual-car-gears-work/, Accessed 21.04.2025.

[10] Skedbooks - Automobile Engineering. Semi-independent Rear Suspension Systems. Access link: https://skedbooks.com/books/automobile-engineering/semi-independent-rear-suspension-systems/, Accessed 28.04.2025.

[11] Drivingfast. The best driving guide on the web. Access link: https://drivingfast.net/weight-transfers/, Accessed: 29.04.2025.

[12] Ysn Studio. Race Car package. Access link: https://assetstore.unity.com/packages/3d/vehicles/race-car-package-141690, Accessed: 02.05.2025.

[13] CRAFTED DIGITALS. Mountain race tracks. Access link: https://assetstore.unity.com/packages/3d/environments/landscapes/mountain-race-tracks-110408, Accessed: 03.05.2025.