M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Khazar Mammadli

VEHICLE SIMULATION IN UNITY GAME ENGINE

SUPERVISOR

Dr. László Blázovics

BUDAPEST, 2021

Contents

STUDENT DECLARATION

I, **Khazar Mammadli**, the undersigned, hereby declare that the present BSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 14 May 2021

...…………………………………………….
Khazar Mammadli

Summary

The topic of the thesis is the simulation of a self-driving vehicle in a virtual city in the Unity Framework. The vital components for the Unity Framework required to achieve a sufficient autonomous car are presented. The thesis gives an extensive overview of the currently used techniques in the modern-day automotive sector. It describes the implementation of one of these methods and takes the user through a detailed design process. After the design process, the thesis explains how to achieve a sufficient amount of control by trial and error techniques. The implemented vehicle is tested in urban areas with pedestrians, traffic lights, and countryside highways with different cars.

In the end, a discussion and improvement of the approach have been noted. Assessment of the work has been made based on the observations during the research, development, and testing phases. The main goal of achieving an autonomous car component that can be deployed in several different virtual environments has been met.

1 Introduction

Thanks to recent breakthroughs in Artificial Intelligence, we are confronted with unique and revolutionary technology on a day-to-day basis. Without a doubt, the most exciting of these groundbreaking innovations are self-driving vehicles.

Recently, driverless vehicles have become a popular subject all around the world. Many consider these automobiles to be the most significant breakthrough of the century. The number of countries using these vehicles is increasing on a daily basis. Tesla, Uber, Google, and other megacorporations continuously allocate resources to research this field, hoping to expand the horizons of this future-changing technology even further. These constant advancements in the industry, combined with my insatiable passion for cars, compelled me to delve deeper into the available research. The roots of my interest in this field actually date back to my childhood, the main reason being my father's involvement in this area. As an automobile engineer, he would always inform my brother and me with interesting new facts about the direction that the industry was going towards. I remember watching a movie with my dad when I was nine years old, where the main protagonist had an autonomous car as his trusted friend. At that time, my father assured me that this was not fiction and all of this would become a reality in the near future, and perhaps one day, even I could be working on such a car.

The standard in the automotive industry has changed from only putting emphasis on the speed and reliability of the cars to the incorporation of functionality, safety, and "intelligence" into the products. The strong development in the field of information technology has already created favourable conditions for the production of such machines. Thanks to the collaboration between the automotive industry and IT developers, a lot of work is already underway to create fully autonomous cars. My interest in both fields, together with my childhood memories, led me to choose this topic as my final thesis.

The competition in the automotive industry is fierce. To have the upper hand in this battle, to limit the loss of funds, time and to ensure the safety of the solution extensive testing has to be done before adopting the solution into real-world scenarios. This is where Game Engines such as Unity can help simulating the possible scenarios.

In the thesis, the development of a self-driving vehicle that is suitable for different city environments has been described. In chapter two, the Unity Framework is introduced. Chapter three presents the current methods that are employed in the industry. Chapter four is focused on the implementation of one of the approaches described in chapter three. The final chapter presents possible improvements and discussion on the implementation method.

2 Unity Game Engine

Unity is a cross-platform game engine developed by Unity Technologies. Its first-time debut was at the 2005 Apple Inc. Worldwide Developers Conference with the purpose of generalizing the gaming industry by making it accessible to a larger pool of developers. Although at the time of the release, it was a Mac OS X-exclusive only, over the years, the list of supported platforms has only continued growing. Currently, Unity supports 19 different platforms that include mobile(Android, iOS), desktop(Universal Windows Platform, Mac, Linux), console(Playstation, Nintendo Switch, Xbox), and virtual reality(Playstation VR, Oculus). Along with the gaming component, it has widely become more and more utilized in the fields of engineering, architecture, film, automotive industries. [1]

In the May of 2018, Unity announced its increase in commitment to the Automotive industry and created a separate division that includes experts from world-leading automobile manufacturers such as Volkswagen, Renault, Delphi. In the July of the same year, they have hosted a two-day Auto Tech summit where demonstrations were made showing the current automotive industry trends and the ability of Unity to tackle the challenges that these trends bring with them. [2]

Unity comes bundled with a slew of features that simplify game implementation, including three-dimensional graphics, an integrated Physics engine, a community-managed Asset store, Scripting API with an integrated Visual Studio, and much more.

2.1 Unity Editor Interface

This section introduces components of the Unity Editor Interface that are required for application development. These components are referenced in the subsequent chapters.

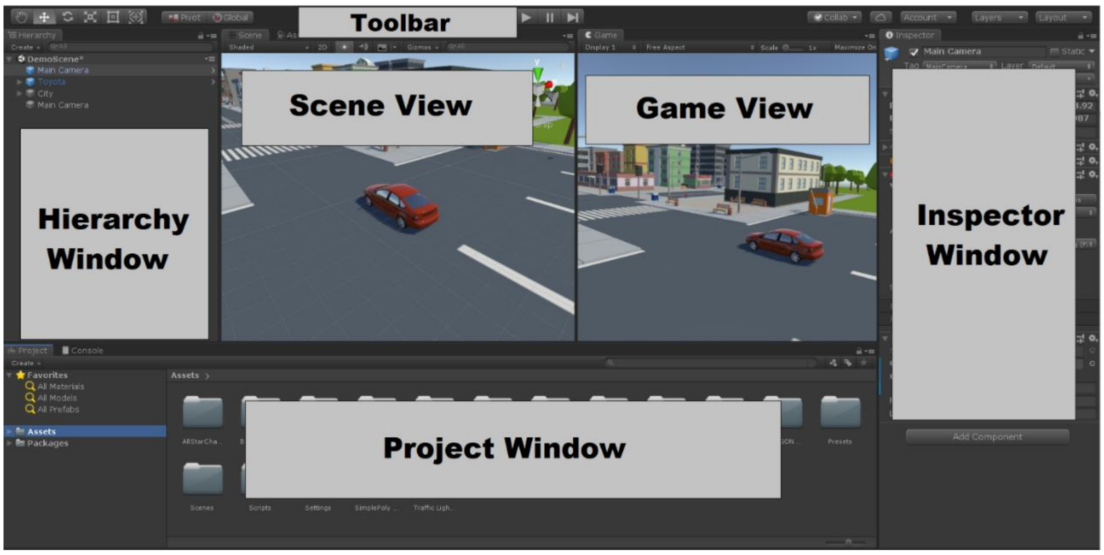


Figure 2.1 User Interface of Unity Editor

- **Project Window:**

The *Project Window* is the main window used for navigation inside the current Unity Project. It contains all of the imported Assets and Project files, such as C# Scripts.

- **Scene View:**

The *Scene View* is the interactive view of the current game that is being created. Any kind of GameObject has to be added to *Scene* before it can be utilized in the application.

- **Game View:**

The *Game View* is the tab where the user can see the final state of the application. It is rendered from the cameras that are placed in the Scene View.

- **Inspector Window:**

The *Inspector Window* displays all of the information about the currently selected GameObject. Properties of the GameObjects can be modified using this window. Component addition to GameObjects is also done through this window.

- **Hierarchy Window:**

The hierarchy window is responsible for displaying all of the GameObjects that are located in the current *Scene*. It contains the *GameObjects* hierarchically, meaning all of the child objects of a GameObject can be found inside that object.

Toolbar:

The *Toolbar* holds the essential working features. On the left side are the buttons for manipulating the current *Scene* view. Play, pause, and step buttons responsible for manipulating the game view are located in the middle of the *Toolbar*. On the right side is the Account button which can be used to access the user's account, followed by the layer visibility menu and editor layout menu used for customizing and saving layouts.

2.2 Prefab System & Asset Store

The *Prefab System* in Unity lets users construct, configure and store a *GameObject* with all of its property values and components defined. This *Prefab Asset* can be used as a template from which new *Prefab Instances* can be created in a different *Scene*. If a *GameObject* is created in one *Scene*, then moved to another *Scene* by copy-paste, any changes made to that *GameObject* will only be kept in the new *Scene*. By creating a *Prefab Asset,* a user has the advantage of avoiding this as any modification made to *Prefab Asset* in one *Scene* is automatically reflected across all of the *Scenes* that utilize this *Prefab Asset.*

One typical usage of *Prefab* would be environmental assets such as a particular type of building that is used numerous times when generating a city scene.

Asset Store is where developers from the Unity community can share their created *Prefab Assets.* It has a wide range of choices ranging from simple, free *prefabs* such as character models, city environments to highly detailed *prefabs* such as Non-Playable-Character with embedded routing algorithms, dynamic city builders.

2.3 Simulation Environment

Regarding simulations, one of the most complex parts is to set up the correct environment. Unity introduces its own *Physics* engine, which takes a tremendous amount of this burden from the developer's shoulders. Without it, the developer would have to create an environment that incorporates basic physics rules applicable to vehicles, such as air resistance, tire friction, and gravity. The *Physics* engine of Unity takes care of all complicated physics rules. It boils down the development of a drivable vehicle to making use of its integrated *Rigidbody, Box Collider, Wheel Collider,* and *Script* components. The following pages provide an overview of these components.

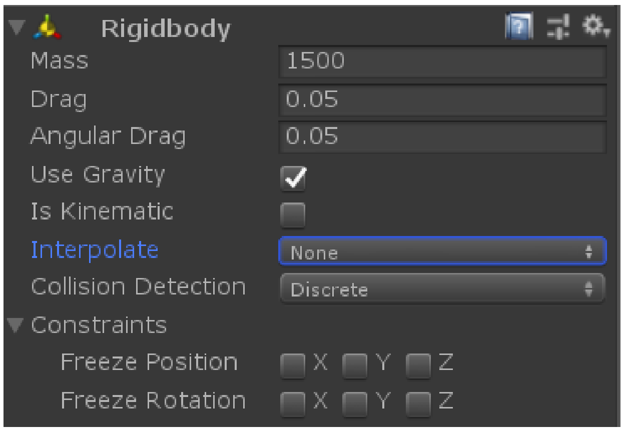
2.3.1 GameObject & Component

GameObject is the simplest entity from in Unity and serves as a container for *Components*. Any object that is placed on the *Scene View* is a *GameObject*. By default, every *GameObject* is generated together with a *Component* called *Transform*.

Components add the necessary functionality to the *GameObjects*. Apart from the default *Transform*, Unity gives access to several in-built components. In the case the desired functionality cannot be guaranteed by the use of these, custom scripts can be implemented, which also act as *Components*

2.3.2 Rigidbody

The *Rigidbody* component of Unity is responsible for making the physics rules of the game engine to be applied to the attached *GameObject*. With the utilization of this component, behaviour that is really similar to the real world can be easily achieved. The *Rigidbody* component exposes several different properties to fine-tune the physics rules applied to the object.


Rigidbody

Mass	1500
Drag	0.05
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐ X ☐ Y ☐ Z
Freeze Rotation	☐ X ☐ Y ☐ Z

Figure 2.2 Inspector view of a Rigidbody component attached to a GameObject

2.3.2.1 Properties

- **Mass:** Represents the weight of the object that it is attached to (in kilograms)
- **Drag:** The amount of air resistance applied to the object when it is moving by the result of forces. 0 means no air resistance for this object, while infinity makes the object stop immediately.
- **Angular Drag:** The amount of air resistance applied to the object when it is rotating by the result of torque. It is impossible to stop an object from rotating by setting it to infinity.

- **Use Gravity:** If gravity should apply to the object
- **Is Kinematic:** If enabled, the object becomes unaffected by the physics engine and can be moved only by manipulating its Transform object.
- **Interpolate:** Used for smoothing the jerkiness of movement of the objects

  -None – No interpolation

  -Interpolate – Smoothing is done according to the *Transform* of the previous frame

  -Extrapolate – Smoothing is done according to the *Transform* of the next frame
- **Collision Detection:** Prevent fast-moving objects from passing through other objects without detecting collisions.

  -Discrete: Default setting for detecting typical collisions.

  -Continuous: This setting makes the object use the Continuous Collision Detection algorithm. It is mainly used for objects such as bullets, for example, where all collisions need to be detected, as a bullet can travel from one side of a wall to the other side in just a second. If using Discrete collision, this will not be registered.
- **Constraints:** Restrict the motions of the object.

  -Freeze Position: Prevents the object from moving along the selected axes.

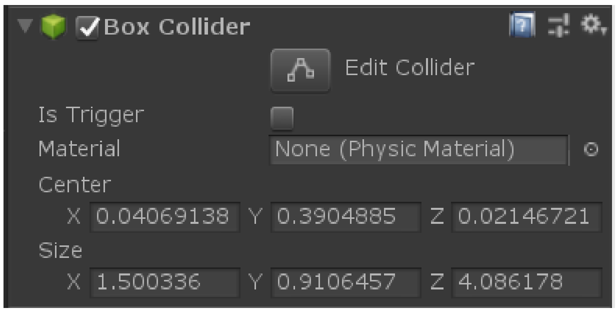
  -Freeze Rotation: Prevents the object from rotating along the selected axes

To manipulate an object with a *Rigidbody* component, scripts need to be utilized. *Rigidbody* can be moved by adding forces to it with *AddForce()* and rotated by adding with *AddTorque().*

2.3.3 Box Collider

A Rigidbody component by itself is not enough to make the physics engine aware of the collisions. *Collider* components need to be added to the object for the physics engine to detect collisions. Unity offers a wide variety of options for *Colliders* ranging from simple and least processor-intensive *Box Collider*, *Sphere Collider*, *Capsule Collider*, which are also referred to as *Primitive Colliders* to more complex, resource-intensive *Colliders* as *Mesh Colliders, Terrain Colliders, Wheel Colliders.*

Mesh colliders are the go-to choice if an object needs to have a detailed collision, while *primitive colliders* are an excellent option for basic simulations due to the low resource cost.


Box Collider

Edit Collider

Is Trigger ☐

Material None (Physic Material)

Center

X 0.04069138 Y 0.3904885 Z 0.02146721

Size

X 1.500336 Y 0.9106457 Z 4.086178

Figure 2.3 Inspector view of a Box Collider component attached to a GameObject

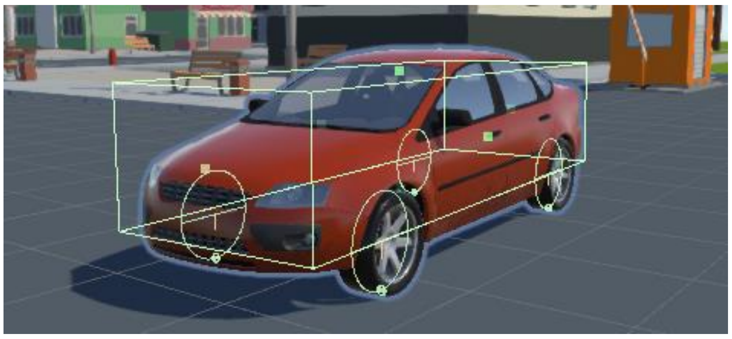


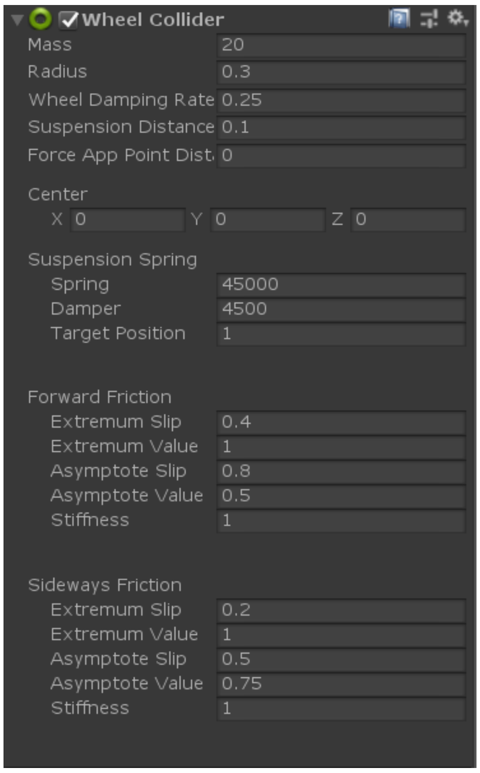
Figure 2.4 Box Collider attached to a car with its edit vertexes enabled

2.3.3.1 Properties

- **IsTrigger:** Tells the physics engine to ignore this *Collider* and use it only for triggering events.
- **Material:** Reference to the *Physics Material*, which determines the *Collider's* behaviour when a collision happens with other *Colliders*.
- **Center:** Position of the *Collider* in the local space of the object that it is attached to.
- **Size:** The size of the *Collider*.

2.3.4 Wheel Collider

The *Wheel Collider* is a type of *Collide*r with a built-in collision detection technique, physics of an actual wheel.[5]



Wheel Collider

Mass	20				
Radius	0.3				
Wheel Damping Rate	0.25				
Suspension Distance	0.1				
Force App Point Dist.	0				
Center					
X	0	Y	0	Z	0
Suspension Spring					
Spring	45000				
Damper	4500				
Target Position	1				
Forward Friction					
Extremum Slip	0.4				
Extremum Value	1				
Asymptote Slip	0.8				
Asymptote Value	0.5				
Stiffness	1				
Sideways Friction					
Extremum Slip	0.2				
Extremum Value	1				
Asymptote Slip	0.5				
Asymptote Value	0.75				
Stiffness	1				

Figure 2.5 Inspector view of a Wheel Collider component attached to a GameObject

2.3.4.1 Properties

- **Mass:** Represents the weight of the wheels in kilograms.
- **Radius:** The radius of the *Wheel Collider* circle.
- **Wheel Damping rate:** The amount of damping applied to the wheel in order to prevent it from oscillating.
- **Suspension Distance:** The maximum distance the suspension can extend. The suspension always extends in downward the direction of the local Y-axis.
- **Force App Point Distance:**

  The location where the forces get applied to the wheel
- **Center:** Center of the wheel object in the local space
- **Suspension spring:**

  -Spring: The force that tries to make the suspension reach the *Target Position*. Increasing the value will result in suspension trying to hit the *Target Position* more quickly.

-Damper: The counteractive force to the spring. The greater the value, the slower the suspension will reach the *Target Position*.

- **Target Position:** The position that suspension tries to get the wheel back into by applying the before mentioned suspension forces.
- **Forwards/Sideways Friction:** Properties of the tire friction when the wheel is rolling in a forward and sideways motion. They are required to adjust the vehicle slip on surfaces.

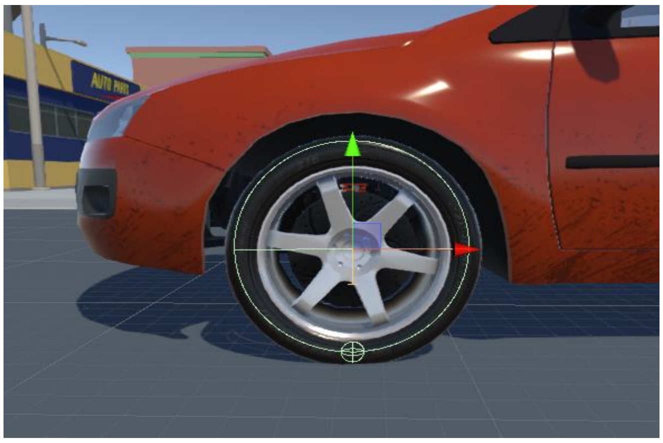


Figure 2.6 Wheel Collider with 0.3 Radius and 0.1 Suspension Distance

3 Research

Without a doubt, the autonomous vehicle field is quite advanced. The development of a fully autonomous car is still a challenge yet to be undertaken by the Autonomous Vehicle community. The interest in this topic is at an all-time high, with constant research being done on the subject, with new papers and hardware emerging on almost a daily basis. To accomplish the end goal of simulating a self-driving car, extensive research on the subject was necessary.

3.1 Operation of a Self-Driving Car

This section presents the basic operational principles of an autonomous vehicle, giving a brief overview of how human-like driving is achieved in modern-day autonomous cars.

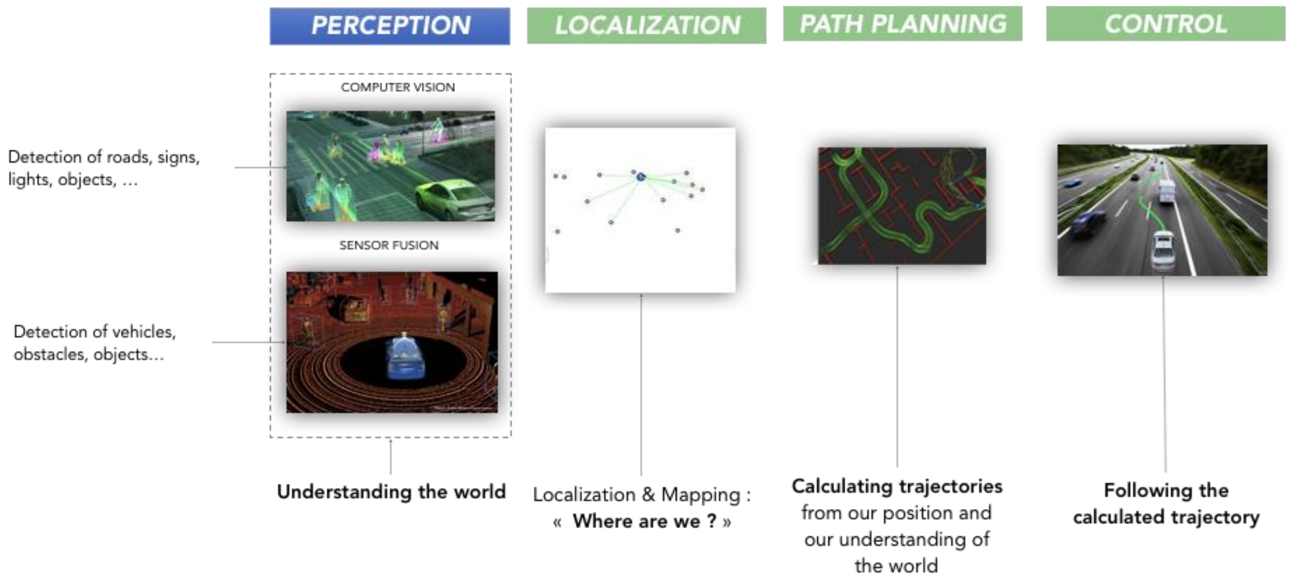

PERCEPTION
LOCALIZATION
PATH PLANNING
CONTROL
COMPUTER VISION
Detection of roads, signs, lights, objects, ...
SENSOR FUSION
Detection of vehicles, obstacles, objects...
Understanding the world
Localization & Mapping : « Where are we ? »
Calculating trajectories from our position and our understanding of the world
Following the calculated trajectory

Figure 3.1: Thought process of a self-driving car[17]

3.1.1 Perception

Perception is the ability to derive applicable information from one's immediate environment. An autonomous vehicle driving on a highway is constantly bombarded with data. From street signs and lane markers to traffic jams, the ability to make fast, intelligent decisions becomes a necessity. The main three primary hardware for making a self-driving car aware of its surroundings are *Camera*, *RADAR*, and *LiDAR* components.[20][21]

The *Cameras* are responsible for producing a digital image of the environment. Autonomous vehicles utilize several *Cameras* in order to stitch together a 360-degree image of their surroundings. Although cameras have precise visuals, they are not without drawbacks. They can distinguish details in their surroundings, but the distances between the car and the objects must still be computed to determine their precise location. Additionally, camera-based sensors have a more challenging time detecting objects in low-light environments such as fog, rain, or nighttime.

RADAR sensors work by transmitting radio waves in pulses. After striking an object, these pulses return to the sensor containing the data about its speed and location. *RADARs* supplement the vehicles *Cameras* in low light conditions, but they are also insufficient when it comes to detecting precisely what kind of object was hit.

LiDAR is a sensor that uses pulsing lasers to measure distances. Lidar enables self-driving cars to have a 3D view of their surroundings. It gives shape and depth to the surrounding cars and pedestrians, as well as the road geography.

The collected data from these sensors is subjected to a method known as *Sensor Fusion*, which involves fusing data from multiple sensors to obtain more accurate information with less uncertainty. This information is then passed into a high-performance artificial intelligence computer, which makes routing decisions.

3.1.2 Localization

The process by which a car determines its location in the world is called *Localization*. To travel autonomously without human-provided assistance, the vehicle must first place itself on a map. Without knowledge of the vehicle’s location within an area or location of the other objects, the vehicle cannot reach its destination or be aware of its direction. A self-driving vehicle that aims to drive from one location to another must be mindful of its global location. The autonomous vehicle must view and localize itself in its immediate environment in relation to static and dynamic obstacles such as trees, houses, other cars, pedestrians, and cyclists. Additionally, the vehicle should be conscious of its position on the route, including its proximity to lane markings, road markings, and curbs. This is necessary to ensure that the vehicle understands when to drive, when to brake, when to change directions, and when to turn.[22]

3.1.3 Behaviour and Path planning

The brain of a self-driving car is the *Path-planning* algorithm. It is the module that attempts to mimic the reasoning and decision-making processes that humans engage in when driving — reading the map, analyzing our environment (other vehicles and pedestrians), and deciding on the optimal course of action based on safety, speed, and traffic laws.

Model-Free Approach:

This approach utilizes control theory and mathematics to calculate inputs and does not rely on training.

Model-Based approach:

This approach is based on putting the car into different circumstances as crossroads, road junctions, high-speed motorways and modeling each possible outcome.

Machine Learning approach:

This methodology is quite unlike the previous. It consists of two phases: training and prediction. Large amounts of vehicle history data is collected to be utilized in the training phase. From the gathered knowledge, the algorithm then tries to predict the next movement

Pathing algorithms:

Autonomous vehicles employ a variety of pathfinding algorithms to decide the most efficient route from their starting point to their destination. *Dijkstra's Shortest Path Algorithm,*[7] *A* Search*[8] algorithms are some of the general examples of these.

3.1.4 Control

The *Control* module is responsible for actuating the vehicle by generating *longitudinal*[9] and *lateral* [10] inputs. The components within this module are called *Controllers.*

Longitudinal control is responsible for managing the vehicle's cruise speed, acceleration, and brake torque. The *longitudinal control* strategy is critical in ensuring the safety and comfort of automotive passengers through automated guidance. Sophisticated longitudinal control enables further fuel savings. Most cars utilize this control method with simple *Cruise Control Systems.*

Lateral control, also known as *steering control,* is in charge of managing the vehicle's turning rate. The aim of *lateral control* is to follow a desired trajectory while maintaining zero *lateral* and *angular* errors. The path can be generated either offline (predefined path) or online (to avoid obstacles, to change lane). Lateral control can also be used for the purposes of lane-keeping.

There are several possible design choices when it comes to selecting a *Controller* for these inputs. The following section does an overlook of these available *Controller* designs.[11]

3.1.5 Bang-Bang Controller

Bang-Bang Controller is the most straightforward and cheapest controller design out of all of the available schemes. It is a simple binary controller that acts based upon the received input signal. This controller does not take error into consideration, producing the same control behaviour with regards to the input signal being above or below the reference value for the desired outcome. This controller is practically useless, as utilizing it in either *longitudinal* or *lateral* control will produce an abysmal riding experience for the passengers due to the constant jerkiness generated by changing acceleration or steering angle to meet the target speed and trajectory. It is also critical to note that rapid changes in behaviour produced by a bang-bang controller can result in a short lifespan of mechanical parts, if not outright failure.

3.1.6 PID Controller

A common among industrial and numerous other control systems that require a continuous modulated control, *Proportional-Derivative-Integral* controller[12] is a feedback-based control loop mechanism. *PID* is the direct result of the combination of *proportiona*l (P), *integra*l (I), and *derivative* (D) controllers, which operate in conjunction in order to create a better and more accurate controller. Due to its many advantages such as basic structure, robust architecture, overall decent control efficiency, and straightforward implementation, it has become a staple in the autonomous vehicle industry, being utilized in many sectors ranging from car *Cruise Control Systems* to autopilots in drones. [15]

For input, the control loop of a *PID controller* repeatedly receives an *error value* $e(t)$, which is the difference between a target setpoint r(t) and a defined process variable. Afterward, a modification is made on the input value according to the proportional, integral, and derivative terms to generate the output. This output is then funneled back to the control loop.

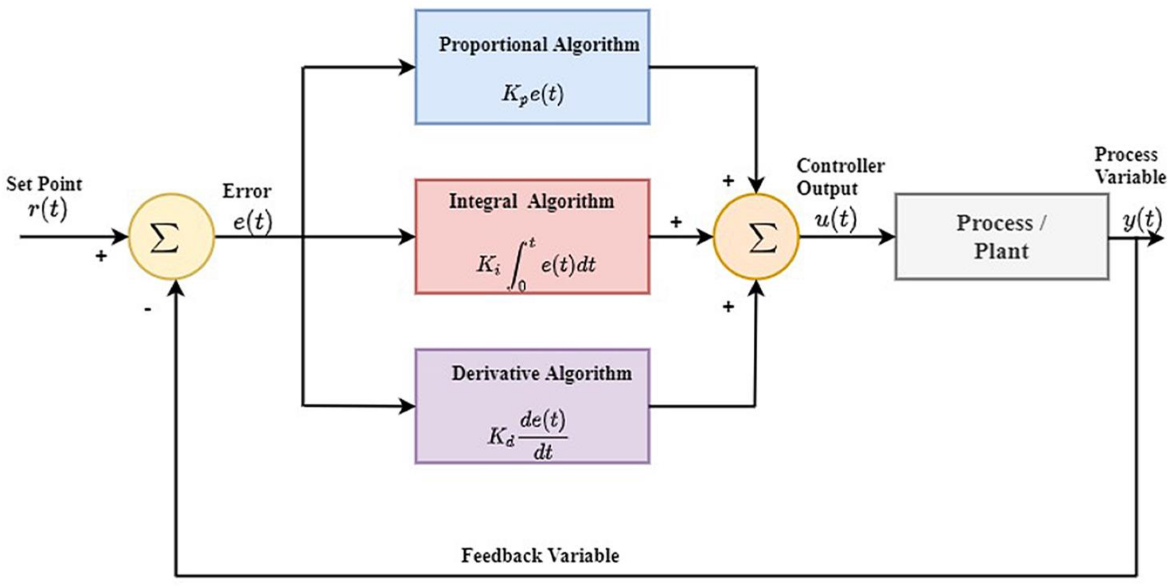


Figure 3.2 A diagram of a PID controller's feedback loop. *r(t)* denotes the target process value, and y(t) denotes the measured process value[14]

The control mechanism of a *PID Controller* can mathematically be represented as

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + \; K_d \frac{de(t)}{dt} \tag{3.1}$$

Where,

$u(t)$ is the output of the controller,

K_p is the proportional gain coefficient, (K_p,

K_i is the integral gain coefficient, (K_i>0),

K_d is the derivative gain coefficient, (K_d>0),

$e(t)$ is the error equal to $r(t) - y(t)$ ($r(t)$ is the Set Point, $y(t)$ is the Process Variable),

t is the time.

3.1.6.1 Proportional term

The first term in the control algorithm is known as the *Proportional term*. It generates an output value proportional to the current value of the error. Adjusting the proportional response implies multiplying the error value $e(t)$ by a constant K_p, which can be referred to as the proportional gain/benefit constant[12][14]. Mathematically expression of this term is

$$P_{out} = K_p e(t) \tag{3.2}$$

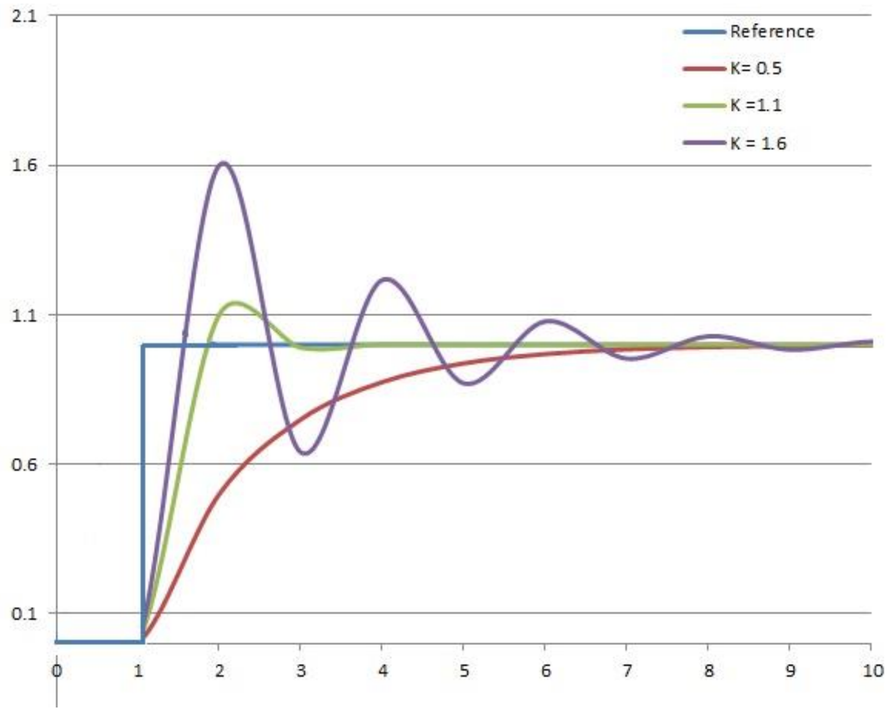


Figure 3.3: Response of Process Value to step change of Set Point vs. time, for values of K_p, K_i and K_d where K_i and K_d are constant[12]

In the above **Figure 3.3,** a graph illustrates the effect of varying K_p values on the response of the system. It can be observed that a sizeable *proportional gain* results in a significant change in the output for a given change in the error. If the *proportional gain* is set to be too high, the system can become unstable and start oscillating around the reference value, sometimes even resulting in the system spinning out of control. This can be seen in **Figure 3.3,** where the reference value is set at 1 and $\boldsymbol{K_p}$ at the value of 1.6. The controller overshoots the reference value by a large margin then slowly oscillates back to match it. By contrast, a low amount of gain yields a small output response to a significant input error, which in return makes the controller react in a less responsive and sensitive manner. By observing the low *proportional gain* $\boldsymbol{K_p}$ which is set to be 0.5, with the reference value of 1 in **Figure 3.3,** the slowness of the response and the significant amount required to reach the target value by the controller can be clearly noted. [12]

3.1.6.2 Integral term

The second term, *Integral term,* is proportional to both the magnitude of the error and to the duration of the error.[12] It is subject to continuous increase over time and is responsible for detecting the errors that go unnoticed by the *Proportional term.*[19] Integral of a *PID Controller* is the summation of the previously calculated errors measured over time which results in a cumulative error offset that was supposed to be adjusted previously.[12] This is then multiplied by K_i, *the integral gain coefficient* and added to the output of the controller. Mathematical representation of the integral term is

$$I_{out} = K_i \int_0^t e(t)dt \tag{3.2}$$

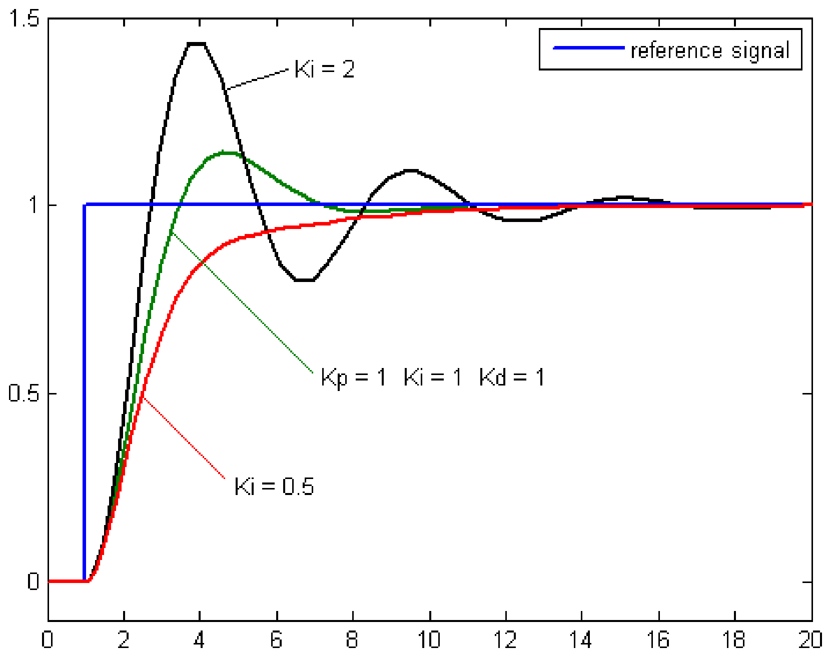


Figure 3.4 Response of Process Value to step change of Set Point vs. time, for values of K_p, K_i and K_d where K_p and K_d are constant[12]

From **Figure 3.4,** it can be observed that if the *integral gain* is set to be too high, it can result in overshooting of the reference value, altering normal controller fluctuations more than required, thus, making the controller unstable. If the *integral gain* is set to be too low, on the other hand, the response takes too much time to get in line with the reference. [12]

3.1.6.3 Derivative Term

The third and last term is the *Derivative term.* Its goal is to flatten the error trajectory into a horizontal line by dampening the applied force, thus reducing overshoot. To put it simpler, it tries to countermeasure the oscillations that happen during the proportional control. The calculation of the derivative is based on determining the slope of the process error over time and multiplying this measured rate of variance by the derivative gain constant K_d. This term can be expressed as

$$D_{out} = K_d \frac{de(t)}{dt} \tag{3.3}$$

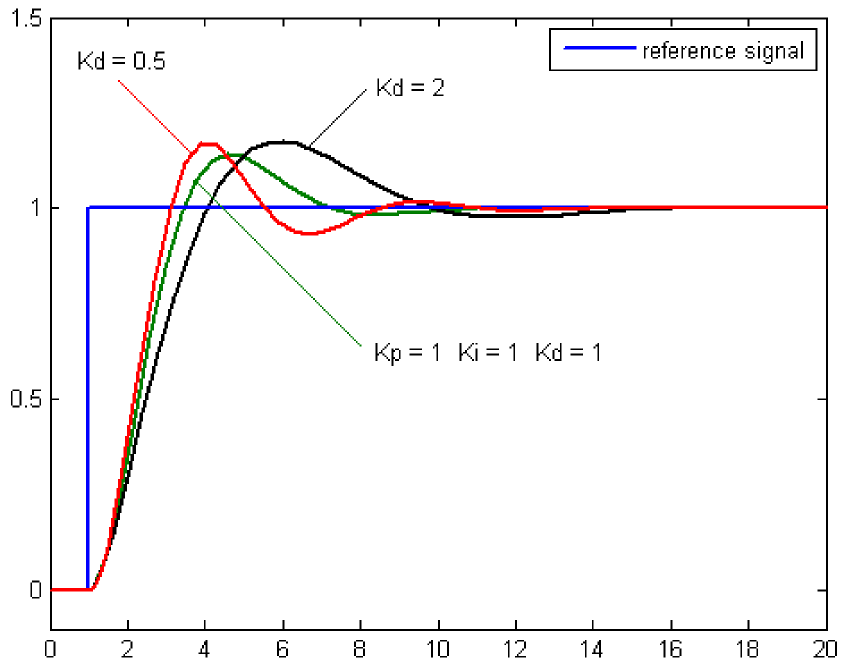


Figure 3.5: Response of Process Value to step change of Set Point vs. time, for values of K_p, K_i and K_d where K_p and K_i are constant[12]

As shown in **Figure 3.5** that if the *derivative gain* is set to be too low, the system will be *underdamped*, meaning it will still continue to oscillate. If vice-versa, the *derivative gain* is set to be too high, the system will be *overdamped*, meaning it will take a significantly larger amount of time than desired to correct for offsets.

3.1.6.4 Fine-Tuning the Control Loop

To be efficient at controlling the system PID controller needs to be specifically *tuned*. *Tuning* is the process of tweaking a control loop's proportional, integral, and derivative gains to an optimal setting to get an appropriate control response. Even though it might seem as simple as just playing around with coefficients, in practice, tuning is the hardest part of PID controller design, especially when targeting several conflicting tasks as high stability and low latency response.[12]

The oldest strategy in the book is, of course, *Manual tuning*. One of the standard techniques is to assign the integral and derivative parameters to zero at the beginning and gradually increase the proportional coefficient until the output of the loop starts oscillating, afterward halving it, to get a "quarter amplitude decay" kind of response. Then integral coefficient needs to be increased to correct for any offset, keeping in mind the sufficient time allotted for the desired response. It should also be taken into account that over exaggerating this parameter will lead to instability of the system. Finally, the derivative coefficient is raised until it can be observed that the loop returns to its reference value in an acceptable amount of time. As with the other parameters, too much derivative coefficient will also pose drawbacks, as an excessive response, which will make the system overshoot.[12]

Fortunately, there a quite a few other proven methods that can be easily adopted to achieve better results in shorter time than *manual tuning*. One of these methods is the *Ziegler-Nichols Method.*

3.1.6.5 Ziegler-Nichols Method

Developed by John G. Ziegler and Nathaniel B. Nichols in the early 1940s, this heuristic method is performed by first setting the *integral* and *derivative* gains to zero. The *proportional* gain is then raised from zero until it approaches the *ultimate gain* $\boldsymbol{K_u}$, where the response of the control loop starts oscillating in a steady and consistent manner. Depending on the type of the controller, the *oscillation period* $\boldsymbol{T_u}$ and *ultimate gain* $\boldsymbol{K_u}$ can be utilizing the table below:[12][13]

Control Type	K_p	T_i	T_d	K_i	K_d
P	$0.5K_u$	–	–	–	–
PI	$0.45K_u$	$0.8T_u$	–	$0.54K_u/T_u$	–
PD	$0.8K_u$	–	$0.125T_u$	–	$0.1K_uT_u$
Classic PID	$0.6K_u$	$0.5T_u$	$0.125T_u$	$1.2K_u/T_u$	$0.075K_uT_u$
Pessen Integral Rule	$0.7K_u$	$0.4T_u$	$0.15T_u$	$1.75K_u/T_u$	$0.105K_uT_u$
Some overshoot	$0.33K_u$	$0.5T_u$	$0.33T_u$	$0.66K_u/T_u$	$0.11K_uT_u$
No overshoot	$0.2K_u$	$0.5T_u$	$0.33T_u$	$0.4K_u/T_u$	$0.066K_uT_u$

The *ultimate gain* $\boldsymbol{K_u}$ can be expressed as 1/*M*, where *M* is the amplitude ratio. This results in

$$K_i = K_p/T_i \tag{3.4}$$

$$K_d = K_p T_p \tag{3.5}$$

Substituting these into the control function defined at 3.1, we get the output formula of

$$u(t) = K_p e(t) + \frac{K_p}{T_i}\int_0^t e(t)dt + \ K_p T_p \frac{de(t)}{dt} \tag{3.6}$$

3.1.6.6 Pseudocode of a PID controller

The pseudocode for implementing a PID controller is as follows [12]

```
previous_error := 0
integral := 0

loop:
    error := setpoint − measured_value
    proportional := error;
    integral := integral + error × dt
    derivative := (error − previous_error) / dt
    output := Kp × proportional + Ki × integral + Kd × derivative
    previous_error := error
    wait(dt)
    goto loop
```

3.1.7 Model Based and Learning Based Controllers

Model-based controllers make use of one or more types of mathematical models of the controlled system. Further categorization of these controllers can be done based on model complexity and the specific methods used to produce the control commands. [11][18]

Kinematic Controllers use mathematical formulas and geometry to establish the movement and trajectory of the vehicle. These types of controllers do not take into account internal and external forces such as slip, air resistance, gravity and thus are often used in low-speed scenarios where estimation of system dynamics is feasible.

Dynamic Controllers, on the other hand, are based on system dynamics and employ detailed motion models of the supervised system. Thus, these types of controllers outperform the *Kinematic Controllers* in terms of raw control, but this comes at a high price of having to do extensive computations involving precise models at each time step.

Model Predictive Controllers forecast the system's potential states (up to a finite receding horizon) and evaluate the optimal control operation at each time point by numerically solving a bounded optimization problem (basically treating the control problem as an optimization problem, a technique known as optimal control). Seeing as model predictive controllers directly manage motion restrictions, the optimization issue is constrained, which is another factor for their success. While this can seem like the ideal control methodology, if not implemented with heavy thought put into them, they can cause undesirable control delay due to the high cost of having to solve an online optimization problem at each time point.

Learning-based Controllers employ Machine Learning and Artificial Intelligence. These controllers utilize deep learning, neural networks and, unlike the controllers discussed previously, do not require an in-depth knowledge of the system. These controllers have the potential to generalize over a variety of similar situations, making them suitable for slight variations in driving. Drawbacks to these controllers are the exceptionally high cost of the training process and gathering data for this purpose. There is also the factor of unpredictability that these controllers impose, as it is unknown how they will react to a scenario that was previously not encountered.

3.2 Conclusion

The primary requirements before starting the implementation phase were simple implementation and easy adaptation for different scenarios.

After doing extensive research on the available techniques, I’ve decided to proceed with the PID Controller for the control strategy of the car.

The bang-bang control strategy was too error-prone and did not provide the necessary outcome. On the other hand, though AI and Model-based approaches seemed like a good choice, but they are significantly harder to implement, and in most cases, they also utilize some form of PID Controller’s as well.

After deciding on using the PID as my control technique, I had to choose an appropriate tuning method to utilize it effectively. *Manual tuning* required too much expertise and was too time-consuming. The same difficulties arose when trying to implement different tuning strategies such as *Twiddle Algorithm*[23] for auto-tuning. Thus, I’ve decided to go with the proven technique of Ziegler- Nichols.

4 Implementation

4.1 Simulation Vehicle

The first step in implementation was choosing the simulation vehicle. For this purpose, I have decided to go for a **mid-sized sedan**. An average mass for a mid-sized sedan is 1524 kilograms. I have rounded this up to **1500 kg** for the simulation. The next step was to choose the **acceleration** of the vehicle. With keeping both inner city and outer highways in mind, I have chosen a constant acceleration of **4** $\boldsymbol{m/sec^2}$. For the steering angle, I've decided to choose **40 degrees**, as this is the typical steering angle for civilian vehicles. For motor force, an average value of **200** horsepower was selected. The force needed to deaccelerate the vehicle by a constant deacceleration of previously decided **4** $\boldsymbol{m/sec^2}$ was calculated by

$$\boldsymbol{F = m * a} \tag{4.1}$$

Where $\boldsymbol{m}$ is the mass of the vehicle, and $\boldsymbol{a}$ is the acceleration.

Substituting **1500 kg** and **4** $\boldsymbol{m/sec^2}$**,** we get **6000** $\boldsymbol{Nm}$**.** To convert this into break torque, we can then multiply it by the radius of our *WheelCollider*, which in my case is **0.3 meters**(See **Figure 2.5**). As such, the final decision for the break torque was **1800** $\boldsymbol{Nm}$. Another thing to take into consideration for the simulation was finding out how early it is recommended to start steering for a safe turn before reaching a point. This value fluctuated between 3-7 meters, so I've decided to choose the middle ground of **5 meters.**

With these values in mind and strategy for the lateral and longitudinal control decided, the implementation phase started.

4.2 Scenarios

The second phase of Implementation was defining the necessary scenarios that an Autonomous car can find itself in.

4.2.1 Empty roads

Inside a city or on the outskirts of it, there is always the possibility of roads being empty. In this scenario, the autonomous vehicle has the freedom of choice if it would like to accelerate or brake, or change direction. The only factors that need to be taken into consideration in this phase are the traffic rules and speed limits. Generally, speed limits in the inner city are around **35-40 km/h**, while on the highways, the range tends to be between **60-120 km/h**.

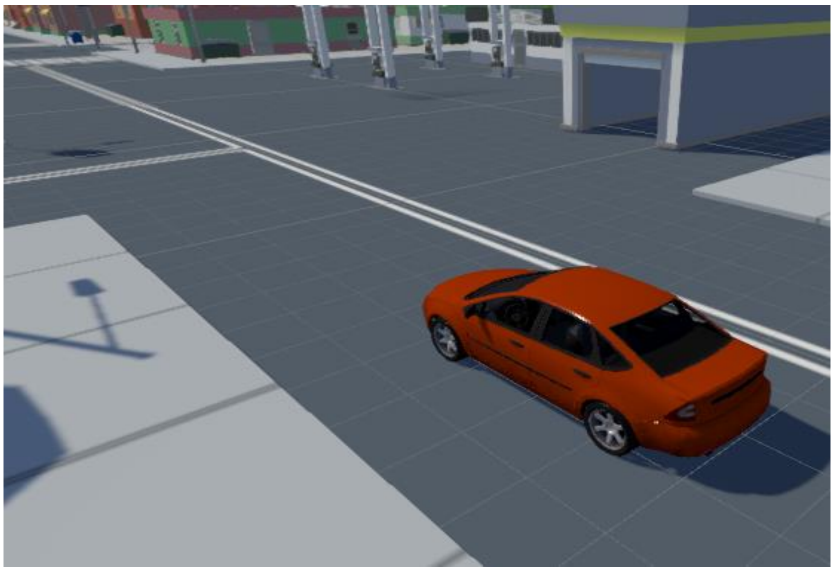


Figure 4.1 Visualization of the Empty road scenario

4.2.2 Other Vehicles

In a city environment, an autonomous car will always be in close proximity of other vehicles. A definitive strategy was needed to be decided on how the car is going to act when coming in contact with other vehicles. Drivers in the real world always maintain a safe distance in between them to have time to perform evasive manoeuvres in case the vehicle in front decides to break suddenly. Nevertheless, if this distance is unreasonably high, it can lead to traffic jams and unwanted road accidents. A rule of thumb for keeping a safe distance is known as the *Two-Second Rule*.[24] To achieve optimal performance in relation to the world, I have decided to use this rule as a basis for control.

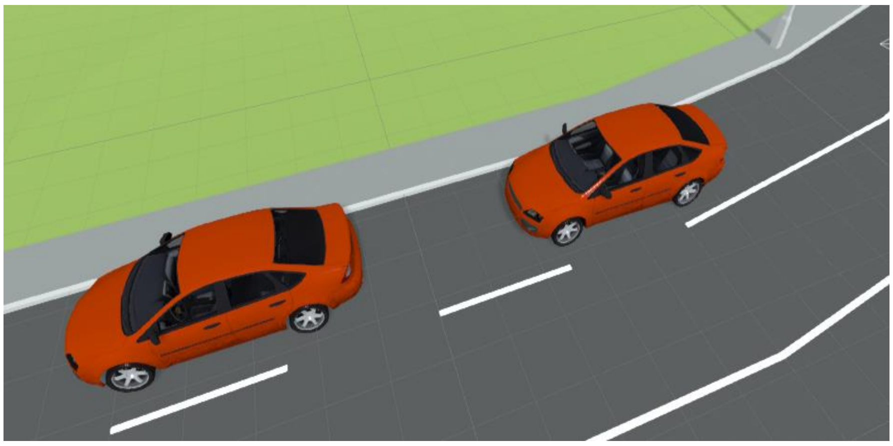


Figure 4.2 Visualization of the Scenario with Two-Second Rule

4.2.3 Pedestrian Crosswalk

Another valid scenario that needed to be taken into consideration for an autonomous vehicle is the buzzing action of pedestrians that are going through crosswalks. An autonomous car should be able to sense these pedestrians from a far away. If it sees that a collision is imminent, if the movement continues with the current speed, it should initiate the breaks while also considering the scenario in the previous section.

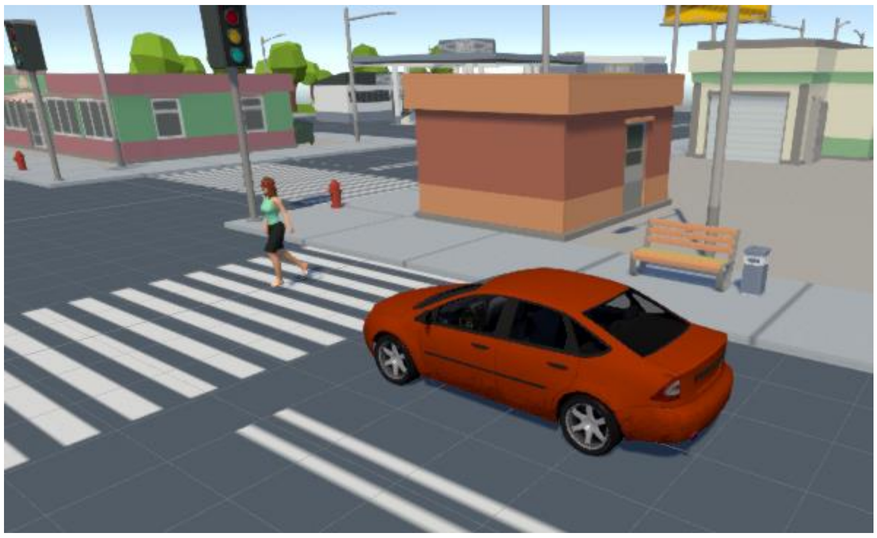


Figure 4.3 Visualization of the Pedestrian Scenario

4.2.4 Traffic Lights

The last scenario for the autonomous vehicle in a virtual city would be traffic road regulations, such as traffic lights. The car needs to be able to check ahead for and measure if it is going to cross at a red light if it continues to maintain the velocity at which it is moving. In the case of green light, it should continue maintaining its speed and, if the traffic rules allow for it, accelerate.

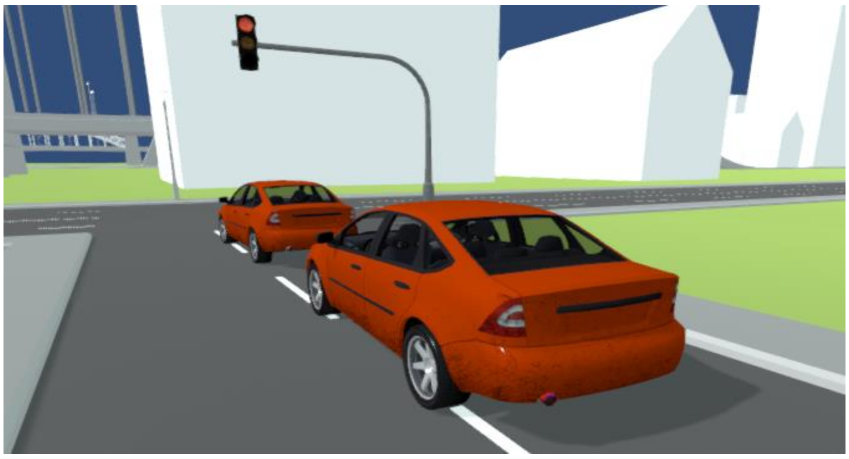


Figure 4.4 Visualization of the Traffic Light Red Scenario

4.3 Generation of route

To experience the before-mentioned scenarios, an autonomous car first needs to have a route. A route in the simulation world is a set of points that the car needs to go through in order to reach the destination point. For defining the route, empty *GameObjects* can be placed on the *Scene*. *GameObject* is the simplest entity in Unity. When created, it will always come with a *Transform* component attached to it. This component can be accessed to get the location of the entity.

To create the route I have introduced an empty *GameObject* called *Path*. This *Path* object holds other empty *GameObjects* called *Waypoints* in the as children. Each *Waypoint* also holds a separate component, with one property called *Speed Limit*. The *Speed limit* is utilized by the vehicle's *Cruise Control* system in order to alter the velocity when the need arises.

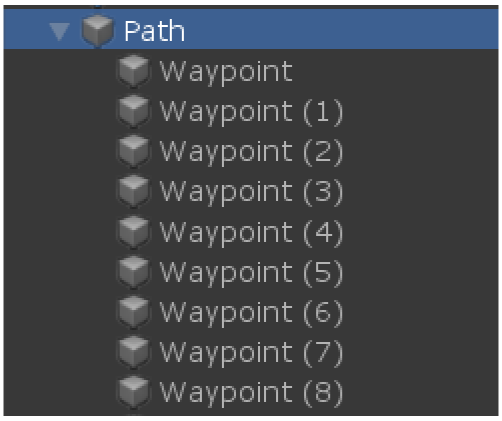
```
Transform[] pathTransforms = GetComponentsInChildren<Transform>();

    waypoints = new List<Transform>();

    for (int i = 0; i < pathTransforms.Length; i++) {
        if (pathTransforms[i] != transform) {
            waypoints.Add(pathTransforms[i]);
        }

    }
```

The only thing that needs to be taken care of is the exclusion of the *Path* object's t*ransform* from the list of the *Waypoint transforms*.



Path
Waypoint
Waypoint (1)
Waypoint (2)
Waypoint (3)
Waypoint (4)
Waypoint (5)
Waypoint (6)
Waypoint (7)
Waypoint (8)

Figure 4.5 Path object in the hierarchy window

Script of the waypoint object

```
public class Waypoint : MonoBehaviour
{
    //Speed limit that can be assigned to each waypoint to be utilized as a target
speed in the Cruise Control PID Controller
    public float speedLimit;

}
```

For better visualization of the trajectory, `OnDrawGizmos()` function can be used to represent the waypoints as spheres and draw lines in between them.

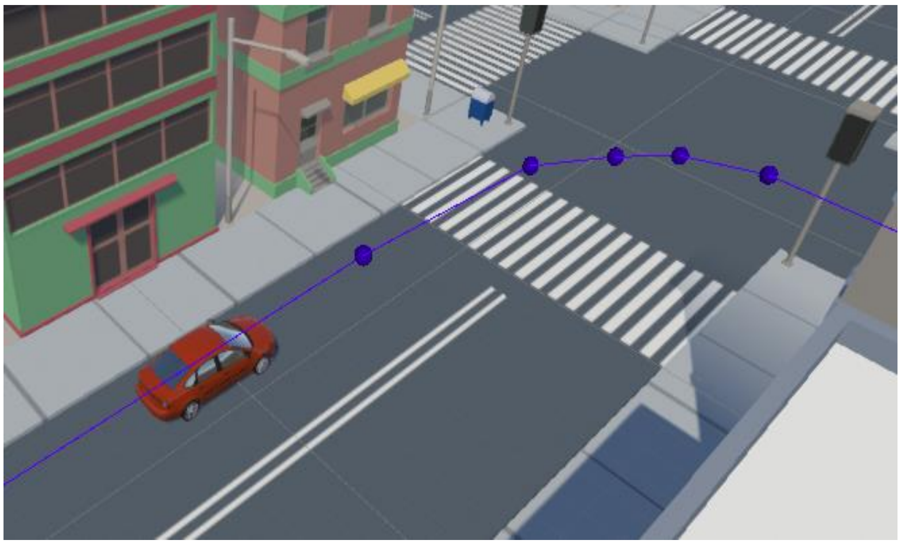


Figure 4.6 A car following a set trajectory. Spheres represent the waypoints.

4.4 Implementation of the Scenarios

4.4.1 Empty road with Cruise control

After the path has been defined, the next step was to implement the longitudinal inputs of the vehicle to imitate *Cruise Control*. If implemented without *Cruise Control*, the car would differentiate between the speeds too rapidly, which would result in unwanted behaviour for the passengers. The Cruise Control system works by imitating a throttle value between -1 and +1. If the current velocity of the car is lower than the required target speed, the throttle factor becomes +1, sending the car instruction to increase its speed. If the target speed before reaching the next waypoint should be reduced, the factor becomes -1 telling the vehicle to initiate its breaks, to slowly deaccelerate to meet the speed requirement.

Using the pseudocode described in section 3.1.6.6, a simple Proportional control algorithm was implemented for mimicking this control.

```
 public class SpeedPIDController : MonoBehaviour
{
    float speedDifference_old;
    float speedDifference_sum;

    public float proportionalCoefficent;

    public float GetCruiseControlThrottle(float speedDifference) {

         float output = 0f;
       output = proportionalCoefficent * speedDifference;

        speedDifference_old = speedDifference;

        return output;
    }
```

The controller receives the speed difference between the `speedLimit` variable defined in the *Waypoint* entity and the car’s velocity at every frame. It then multiplies this value with the proportional coefficient to produce the throttle factor. Another thing that needs to be taken care of is that this value might be too large, resulting in unwanted jitter for the car. To limit this, `Mathf.Clamp()`[6] has been used, which clamps the response gotten from the P controller between -1 and +1. It then multiplies this value with the motor force if the car needs to accelerate or brake torque if the car needs to deaccelerate.

```
private void CalculateCruiseControlThrottle()
{

        float currentVelocity = 3.6f * car.velocity.magnitude;
        float speedDifference = m_targetSpeed - currentVelocity;
 m_cruiseControlThrottle=speedPIDController.GetCruiseControlThrottle(speedDifference);

        m_cruiseControlThrottle = Mathf.Clamp(m_cruiseControlThrottle, -1, 1);

        if (m_cruiseControlThrottle > 0)
        {
            m_isAccelerating = true;

        }
        if (m_cruiseControlThrottle < 0)
        {
            m_isAccelerating = false;
            m_cruiseControlThrottle *= -1;
        }

  }
```

For tuning of the P controller, the manual turning method described in **section 3.1.6.4** yielded satisfactory results.

K_p	Observations
1	Too much motor force/ brake torque for a small error
0.5	Lesser force/brake torque, but still greater than desired
0.1	Perfect fit,thus the final solution

4.4.1.1 Lateral Control

After the success achieved with the *longitudinal* control, the next move was to introduce *lateral* control, as the self-driving vehicle should be tested away from the city until it is proven to be somewhat reliable on the road. The initial approach was the implementation of a *Bang-Bang* controller described in **section 3.1.5**. Although this logic was implemented with a few lines of code, it was not sufficient due to several factors:

The car was unable to gain any speed as the wheels would constantly switch from -40 degree angle to +40 degree angle. This also made the car stick to the route and perform the turns at either too much or too little of an angle. To get around this, a PID controller had to be adopted for calculating the *Cross Track Error* at every frame and modify it accordingly for the movement.

Cross Track Error, simply put, is the difference between the car's current location and the desired position, which is the point nearest to the car on a line connecting two waypoints. To visualize this, see **figure 4.7** below.

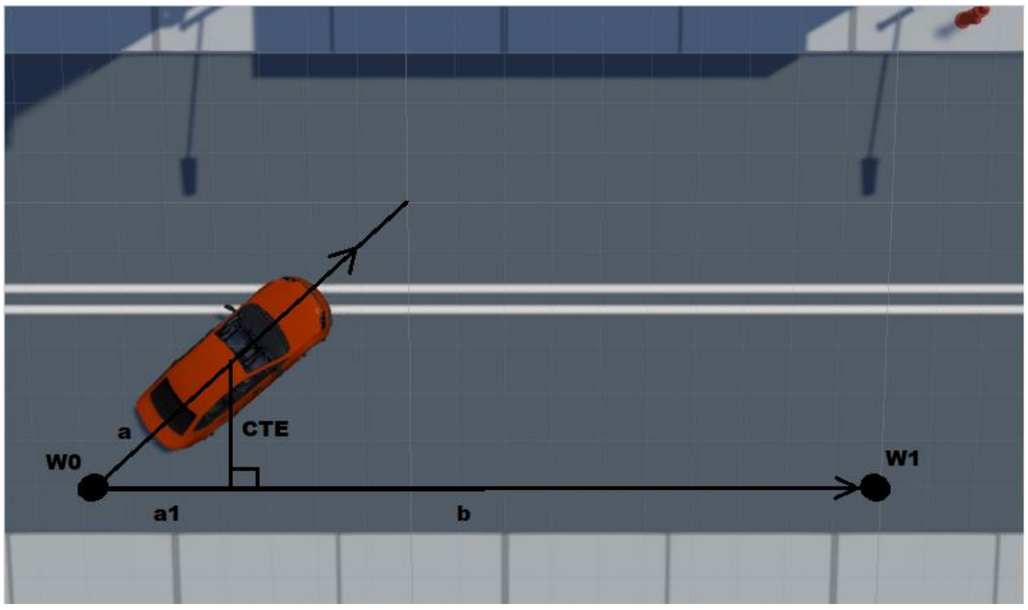


Figure 4.7 Visualization of Cross Track Error when the car is moving in the direction of vector b

Assume that the vehicle is moving from ***Waypoint 0*** to ***Waypoint 1***. Vector $\boldsymbol{a}$ is the vector that goes from the start node ***W0*** and passes through the middle of the car. The vector $\boldsymbol{b}$ is the vector connecting the starting point ***W0*** and target point ***W1*** and is in the direction of the movement. Vector $\boldsymbol{a_1}$ is the *projection*[25] of vector $\boldsymbol{a}$ onto the vector ***b. Cross Track Error*** is the vector rejection**,** in other terms projection of the vector $\boldsymbol{a}$ on the to hyperplane orthogonal to vector ***b.*** The rule implies that sum of the rejection and projection of the vector $\boldsymbol{a}$ should be equal to ***a.*** By using this the vector rejection can be described as

$$\boldsymbol{a_2} = \boldsymbol{a} - \boldsymbol{a_1} \tag{4.2}$$

The vector $\boldsymbol{a_1}$ can be described as a scalar projection with the following formula

$$\boldsymbol{a_1} = \frac{\boldsymbol{a} * \boldsymbol{b} * \boldsymbol{b}}{\|\boldsymbol{b}\|\|\boldsymbol{b}\|} = \frac{\boldsymbol{a} * \boldsymbol{b} * \boldsymbol{b}}{\boldsymbol{b} * \boldsymbol{b}} \tag{4.3}$$

By substituting **4.3** into **4.2,** we get the following expression of

$$a_2 = a - \frac{a * b * b}{b * b} \tag{4.4}$$

Bu utilizing this formula, a code can be generated to calculate the Cross Track Error easily.

```
    public static float GetCrossTrackError(Vector3 carPosition, Vector3 waypointFrom, 
Vector3 waypointTo)
    {

        Vector3 a = carPosition - waypointFrom;

        Vector3 b = waypointTo - waypointFrom;

        Vector3 projection = b*((a.x * b.x + a.y * b.y + a.z * b.z) / (b.x * b.x + b.y * b.y 
+ b.z * b.z));

        float error = (a-projection).magnitude;

        return error;
    }
```

Now that have the error calculation is achieved, it can be passed to the PID controller to tune the error to a suitable degree to use it for steering.

Using the pseudocode in **section 3.1.6.6**, the following PID controller was generated

```
public class SteeringAnglePIDController : MonoBehaviour
{
    //Utilizing Ziegler Nichols algorithm for fine tuning this PID as a Classic PID
    float CrossTrackError_old = 0f;
    float CrossTrackError_sum = 0f;

    //PID parameters
    public float K_Ultimate = 0f;
    public float T_Ultimate = 0f;

    public float GetSteeringAngle(float CrossTrackError)
    {
        //The final steering angle
        float output = 0f;

        //P
        output = 0.6f * K_Ultimate * CrossTrackError; // 0.6* K_Ultimate

        //I
        CrossTrackError_sum += Time.fixedDeltaTime * CrossTrackError;
        output += (1.2f* K_Ultimate / T_Ultimate) * CrossTrackError_sum;
        // 1.2 K_Ultimate / T_Ultimate


        //D
        float d_dt_CTE = (CrossTrackError - CrossTrackError_old) / Time.fixedDeltaTime;
        output += (3f * p K_Ultimate * T_Ultimate)/40 * d_dt_CTE;
       //0.075 * K_Ultimate * T_ultimate

        CrossTrackError_old = CrossTrackError;


        return output;
}
```

After creating the PID controller, it had to be tuned explicitly by entering two parameters: the *ultimate gain* and the *oscillation period* using the Ziegler Nichols method. The table for the method is given in **section 3.1.6.5**.

The observations and the tuning process can be seen from the below table

K_u	T_u	**Observations**
0.5	**-**	Small, unstable oscillations
0.75	-	Small, unstable oscillations
1.5	-	Slightly increased oscillation but still fairly unstable
3	-	Bigger oscillations but still unstable
4.5	-	Better than previously achieved oscillations, but still not fit
13.5	-	Swift response, coupled with aggressive oscillations
12	-	Still a fast response with unwantedly aggressive oscillations
7	-	Point of stability
7	3	Unstable oscillation
7	12	Better but still unstable
7	27	Too slow of a response
7	21	All around good choice, and final solution

Same as before with the *Cruise Control* P controller, the lateral PID controller also can run into issues of the output value being greater than the range of the steering angle. To solve this, the same Unity method `Mathf.Clamp()` can be used to clamp the value between – and + maximum steering angle (in this case, 40 degrees).

4.4.2 Sensors

The previous scenario did not require any interaction of the autonomous vehicle with the environment around it. However, the other three possible scenarios discussed all have some form of stimuli that needs detection. In **section 3.1.1,** the perception module that employs several sensors of the car was described. To implement these sensors, one can use the *Collider* objects with *isTrigger* flag set to true to send a signal when it interacts with the *Collider* of the vehicle. Although computationally cheap, this would prove too simplistic to undertake the dynamic movements of the self-driving vehicle.

Another possible and adopted solution is the using inbuilt feature of the Unity Physics Engine called *RayCasting*. This casts a ray from the given origin, in my case the front of the car, in a chosen direction, the front-facing direction of the car in this case, and in predefined maximum distance. If used in a limited amount, *RayCasting* is exceptionally performance friendly.

A total number of five sensors are utilized, three pointing in the forward direction and two pointing at an angle to cover the possible blind spots of the car.

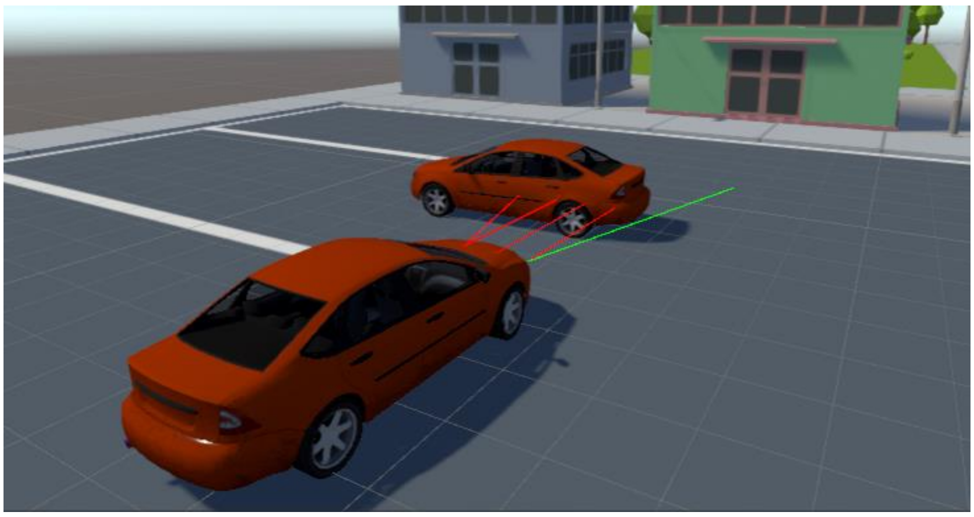


Figure 4.8 Ray Casting as sensors. Red colour defines hit indication with an object, and green colour defines no collision

4.4.3 Vehicles on Route

As mentioned in **4.2.2,** the first thing to interpret was the *Two-Second Rule* to maintain a safe distance between the vehicles. This rule states that between two vehicles, there has to be a gap of at least two seconds. Sometimes this recommended distance can vary to three or four seconds depending on the situation. In other words, the distance has to be equal to the amount of ground that the car can cover in two seconds. To calculate the safe driving distance, the velocity of the car can be multiplied by the amount of time(for the *Two-Second* rule with 2). The resulting value then can be substituted for the length of the sensors, making them optimal for both high and low-speed scenarios.

4.4.4 Pedestrian Crosswalk

Although the sensors worked reasonably well for detecting other cars, an issue arose when implementing the same logic for the pedestrians. The vehicle needs to sense the difference between the objects on the route, as with the initial implementation of the sensor when seeing a pedestrian, the car would act like it is another vehicle, trying to match the walking speed of the pedestrian. To solve this, a separate *BoxCollider* has been placed onto the pedestrian crosswalks. Using Unity's *Layering system*, which is in charge of deciding on which rays will hit which colliders, a *tag* can be set to the *BoxCollider* of the crosswalk to differentiate it from other objects. The civilian character also has a *tag* assigned to it, so the *Pedestrian Crosswalk* can differentiate between the cars and pedestrians that go across the crosswalk.

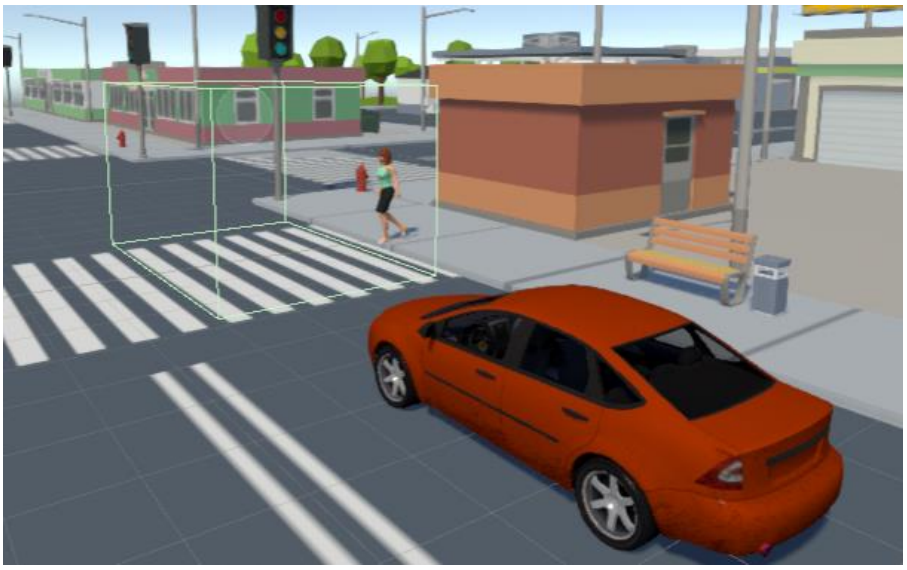
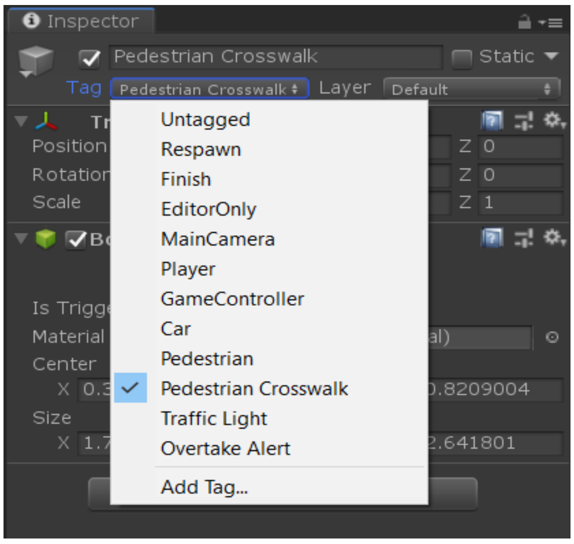


Figure 4.9 Pedestrian Crosswalk



Pedestrian Crosswalk
Tag: Pedestrian Crosswalk
Layer: Default

Figure 4.10 Setting Pedestrian crossing tag

When the car sensors hit the collider of the *Pedestrian Crosswalk*, the entity gets assigned to the car. Whenever a civilian enter this *BoxCollider*, a flag is set, informing the car about the imminent collision, thus, telling it to initiate the breaks.

4.4.5 Traffic Lights

After seeing the initial success with the *Pedestrian Crosswalk*, the same algorithm was utilized for the *Traffic Lights*. Sensors, upon having an impact with the *Traffic Light collider*, assign the entity to the car using the *tag* of the object. The car then checks to see the current situation of the light and produces the necessary control output according to it.

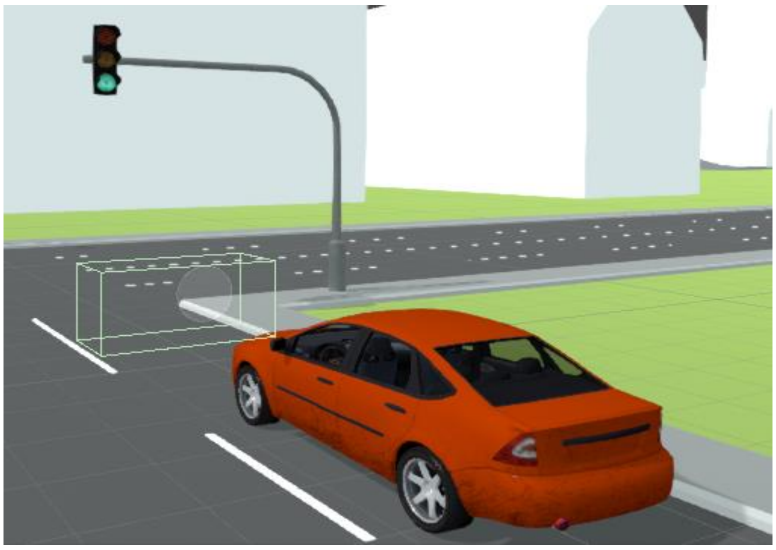


Figure 4.11 Detection of the Traffic Light

5 Discussion & Improvements

During the project development and after the finalization, the were many remarks made:

- With the current implementation, the cars are unable to overtake each other while going on a straight road. An attempt at solving this issue was made via creating a copy of the trajectory of the car with a slight right offset. The overtaken car had box colliders attached to the sides, which were trigger objects to determine if the car has overtaken or not. During the testing phase, the solution did not turn out to be optimal and would sometimes cause road accidents. Thus it was left out.
- Although the PID controller tuning is sufficient for most cases, in very high-speed scenarios, it can be prone to control failure. More sophisticated methods of tuning, like *auto-tuning*, might be able to achieve a better result in such cases.
- The car's perception module does not consider the time frame between the changes of traffic lights but acts only by their current states. A better solution would be to include measurements, so the vehicle would continue accelerating if calculations we're pointing to the light remaining green when the car comes in close contact.

Overall, the vehicle is suitable to be graded with a Level 3 Autonomation on the SAE Levels of Automation scale.[26]

References

[1] Wikipedia: *Unity (game engine)*, https://en.wikipedia.org/wiki/Unity_(game_engine) (revision 12:35, 2021 April 15th)

[2] Unity: *Unity Technologies Increases Commitment to Automotive with Dedicated Team, Industry Bundle, and Two-Day Unity AutoTech Summit*, https://unity.com/our-company/newsroom/unity-technologies-increases-commitment-automotive-dedicated-team-industry (2018)

[3] Unity Manual: *Rigidbody*, https://docs.unity3d.com/Manual/Rigidbody.html (2020.3 LTS)

[4] Unity Technologies: *Unity - Manual: Box Collider*, https://docs.unity3d.com/Manual/BoxCollider.html (2020.3 LTS)

[5] Unity Technologies: *Unity - Manual: Wheel Collider*, https://docs.unity3d.com/Manual/WheelCollider.html (2020.3 LTS)

[6] Unity Technologies: *Unity - Manual: Mathf.Clamp*, https://docs.unity3d.com/ScriptReference/Mathf.html (2020.3 LTS)

[7] Wikipedia: *Dijkstra's Algorithm* https://en.wikipedia.org/wiki/Dijkstra_algorithm (revision 17:50, 2021 March 29th)

[8] Wikipedia: *A* search algorithm*: https://en.wikipedia.org/wiki/A*_search_algorithm (revision 19:28, 2021 April 13th)

[9] Rachid ATTIA, Rodolfo ORJUELA, Michel Basset: *Longitudinal Control for Automated Vehicle Guidance*, IFAC Proceedings Volumes, Volume 45, Issue 30, Pages 65-71, ISSN 1474-6670, ISBN 9783902823168 (2012)

[10] Salvador Dominguez-Quijada, Alan Ali, Gaëtan Garcia, Philippe Martinet: *Comparison of lateral controllers for autonomous vehicle: experimental results*. International IEEE Conference on Intelligent Transportation Systems, Nov 2016, Rio de Janeiro, Brazil.

[11] Samak, Chinmay & Samak, Tanmay & Kandhasamy, Sivanathan: *Control Strategies for Autonomous Vehicles*. https://www.researchgate.net/publication/345989566_Control_Strategies_for_Autonomous_Vehicles (2020)

[12] Wikipedia: *PID Controller*, https://en.wikipedia.org/PID_controller (revision 02:20, 2021 April 29th)

[13] Wikipedia: *Ziegler-Nichols method*, https://en.wikipedia.org/wiki/Ziegler-Nichols_method (revision 21:49, 2021 January 14th)

[14] Borase, R.P., Maghade, D.K., Sondkar, S.Y. et al. *A review of PID control, tuning methods and applications*. Int. J. Dynam. Control 9, 818–827. https://doi.org/10.1007/s40435-020-00665-4 (2021)

[15] Farag, Wael: *Track Maneuvering using PID Control for Self-Driving Cars.* Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering). 12. 10.2174/2352096512666190118161122. https://www.researchgate.net/publication/331475267_Track_Maneuvering_using_PID_Control_for_Self-Driving_Cars (2019)

[16] Araki M: *PID Control*, Control Systems, Robotics, and Automation, Volume II https://www.eolss.net/Sample-Chapters/C18/E6-43-03-03.pdf

[17] Jeremy Cohen: *AI… And the vehicle went autonomous,* https://towardsdatascience.com/sensor-fusion (2018 January 24th)

[18] Jeremy Cohen: *Control Command in Self-Driving Cars*, https://towardsdatascience.com/the-final-step-control (2018 August 27th)

[19] Bogdan Djukic: *Non-AI approach for steering self-driving car* https://towardsdatascience.com/non-ai-approach-for-steering-self-driving-car (2017 July 6th)

[20] Yeong DJ, Velasco-Hernandez G, Barry J, Walsh J. : *Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review. Sensors* https://doi.org/10.3390/s21062140 (2021 March 18th)

[21] NVIDIA: *NVIDIA Blogs: How Does a Self-Driving Car See?* https://blogs.nvidia.com/blog/2019/04/15/ (2019 April 5th)

[22] Woo, A., Fidan, B., & Melek, W. W.: *Localization for Autonomous Driving. Handbook of Position Location*: Pages 1051–1087. (2019)

[23] Martin Thoma: *The Twiddle Algorithm*, https://martin-thoma.com/twiddle/ (2014 September 6th)

[24] Wikipedia: *Two-second rule*, https://en.wikipedia.org/wiki/Two-second_rule (revision 15:53, 2021 January 19th)

[25] Wikipedia: *Vector Projection*, https://en.wikipedia.org/wiki/Vector_projection (revision 03:10, March 18th)

[26] SAE International: *SAE J3016 Levels of Driving Automation,* https://www.sae.org/news/press-room/2018/12/sae-international-releases-updated-visual-chart-for-its-%E2%80%9Clevels-of-driving-automation%E2%80%9D-standard-for-self-driving-vehicles (2018)

Annex