**ABSTRACT**

The main aim that we have is to create a completely automated car parking/Driving system with minimal human interference in real world as well as in real world games . With the rising population in the world, time is of the essence and hence we need to minimize the time taken by trivial activities such as finding a place to park in a busy place and avoid traffic congestion. We have seen in existing systems sometimes accidents can occur in parking situations by cars going at high speed o caused by frustrated drivers unable to find a parking space for a long period of time. In our project we propose a smart and automated car parking model that will help the user in booking their parking spaces beforehand and the vehicle will be able to park automatically once in the parking zone .The difference between our project of automated car parking systems is we hope to minimize human interaction as much as possible and make both the vehicle and the parking area fitted with sensors that will help us execute a safe and efficient way of parking. Hence, we aim to provide a completely safe and automated experience that is robust and can be implemented in real time and hopefully be implemented as the general norm for parking systems in the future.

With the increase in population, the parking space is swarming with cars, which lead to collision accidents. People wait in long queues to get their cars parked. To solve this issue, we present a simple yet effective method for implementing low-cost self-parking car. The car is planned to be made smarter by data handling and machine learning. We propose to store the information from Google maps in the computer system of the car, also with machine learning and image processing the car learns about different scenarios. The results show how self-parking can be implemented to regular cars and an ample amount of time and money can be saved.

On the other hand we can use this simulation prefab for creating open world driving included games where the others cars except the one the player is operating can be AI driven so that the game works like in real life. For implementing this prefab we just have to train our model as per our need in the game using GAIL (Generative Adversarial Imitation Learning) and Behavioral cloning.

TABLE OF CONTENTS

| **CHAPTER NO.** | **TITLE** | **PAGE NO.** |
| --- | --- | --- |
|  | **ABSTRACT** | 4 |
|  | **LIST OF TABLES** | 5 |
|  | **LIST OF FIGURES** | 6 |
| **1.** | **INTRODUCTION** | 7 |
|  | 1.1 Overview |  |
|  | 1.2 Problem Definition |  |
| **2.** | **SYSTEM ANALYSIS** | 8 |
|  | 2.1 Existing System |  |
|  | 2.2 Proposed System |  |
|  | 2.3 Development Environment |  |
| **3** | **SYSTEM DESIGN** | 9 |
|  | 3.1. UML Diagrams |  |
|  | 3.2 Data Dictionary |  |
|  | 3.3 ER Diagram |  |
|  | 3.4 Data Flow Diagram |  |
| **4.** | **SYSTEM ARCHITECTURE** | 13 |
|  | 4.1 Architecture Overview |  |
|  | 4.2 Module Description |  |
|  | 4.3 Pseudo code / Algorithm Description |  |
| **5.** | **SYSTEM IMPLEMENTATION** | 19 |
|  | 5.1 Client-side Coding |  |
|  | 5.2 Server-side Coding |  |
| **6.** | **SYSTEM TESTING** | 23 |
|  | Test Cases & Reports |  |
| **7.** | **CONCLUSION** | 24 |
|  | Conclusion and Future Enhancements |  |
|  | **APPENDICES** |  |
|  | Sample Screenshots |  |
|  | **REFERENCES** |  |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **FIG NO.** | **FIGURE DESCRIPTION** | **PAGE NO.** |
| 3.1 | **Block Diagram** | 9 |
| 3.2 | **Activity Diagram** | 10 |
| 3.3 | **Class Diagram** | 11 |
| 3.4 | **Use-case Diagram** | 12 |
| 3.5 | **Control-flow Diagrams** | 13 |
| 4.3 | **Rigid-Body** | 16 |
| 4.3 | **Car-Controller** | 17 |
| 4.3 | **Sensors** | 17 |
| 4.3 | **Environment and target** | 18 |
| 6.1 | **Working GIFs** | 24 |

**CHAPTER 1**

**INTRODUCTION**

Automatic parking is an autonomous car-maneuvering system that moves a vehicle from a traffic lane into a parking spot to perform parallel, perpendicular, or angle parking. The automatic parking system aims to enhance the comfort and safety of driving in constrained environments where much attention and experience is required to steer the car. The parking maneuver is achieved by means of coordinated control of the steering angle and speed which takes into account the actual situation in the environment to ensure collision-free motion within the available space.

Though the number of people driving cars have improved drastically over the past few years all over the world, most people struggle at the parking part of driving. There are more chances of people crashing during parking than regular driving. Though there are many people driving good they struggle while parking since it needs more precise judgement of their car’s body and the surroundings. This happens due to the presence of blind spots all over the car which cannot be avoided. This is where the autonomous parking system using machine learning comes into place.

The Autonomous parking system is a great solution for parking cars with ease and safety. This helps in parking the car at very tight spaces without stressing the driver for precision. The driver can stay peacefully while the ai parks the car at the spot using its trained brain and sensors to scan the environment. Many car manufacturers have started to use this technology to make the parking easy. But its only available in the cars that come with a very premium price.

Though our project is just a simulation. It works like a real time environment with all the physics working same as that of our world. So, the same idea can be applied in real life too. This implementation can be used in modern game development applications. Since, this is an ai trained model this will give a lot more reality and reduce the coding part needed for each and every use case.

But the prefab that we have created in our project directly in any game that is being developed with driving as and with a bit of training it will work fine and efficient.

**CHAPTER 2**

**SYSTEM ANALYSIS**

**2.1 Existing System:**

The Existing systems for Self-parking cars are implemented using the combinations of camera, radar, ultrasonic and LiDAR sensors. These are complex systems which leads to a very premium cost. Tesla is a very well-known car manufacturer that uses both Lidar and Camera to detect the surroundings.

The Games created nowadays mostly use waypoint system in which the vehicle follows a certain path throughout map till it reaches it destination and all the obstacles that come in the way can’t be avoided. This makes most games glitchy and unrealistic. For the vehicle to travel in the selected pathway there’s a lot of code needed to be written for many cases that may happen in the game. This also leads to bugs when a specific case is missed while programming a game.

**2.2 Proposed System:**

We can use the same parking sensors used in our normal cars along with cameras to detect the spot and park accordingly by the same idea used in the simulation.

We have used reinforcement learning technique for training the ai. Reinforcement Learning (RL) is a direct approach to learn from interactions with an environment in order to achieve a defined goal. In this context, the learner and decision maker is referred to as the agent, whereas the part it is interacting with is called environment. The interaction performs in a continuous form so that the agent selects actions at each time step t, the environment responds to them and presents new situations to the agent in form of a state Sₜ₊₁. Responding to the agent’s feedback, the environment returns rewards Rₜ₊₁ in the form of a numerical scalar value. The agent seeks to maximize rewards over time. we can use the same prefab and codes for implementing this car in any game that includes driving. All we have to do is modify the reward system and train it as per our new environment.

**2.3 Development Environment:**

The Development Environment Used is Unity Version 2019.4.10f1. Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. The engine has since been gradually extended to support a variety of desktop, mobile, console and virtual reality platforms. It is particularly popular for iOS and Android mobile game development and used for games such as Pokémon Go, Monument Valley, Call of Duty: Mobile, Beat Saber and Cuphead. It is cited to be easy to use for beginner developers and is popular for Indie game development.

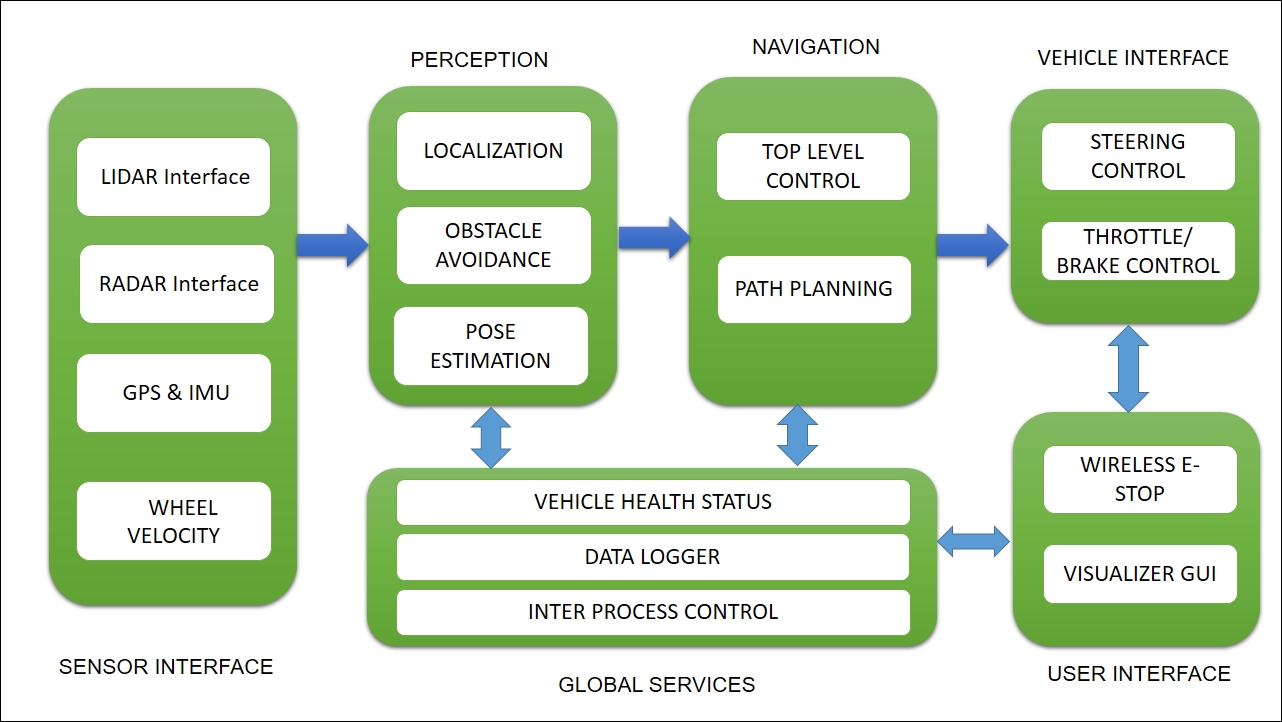
The engine can be used to create three-dimensional (3D) and two-dimensional (2D) games, as well as interactive simulations and other experiences. The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering and construction.

**CHAPTER 3**

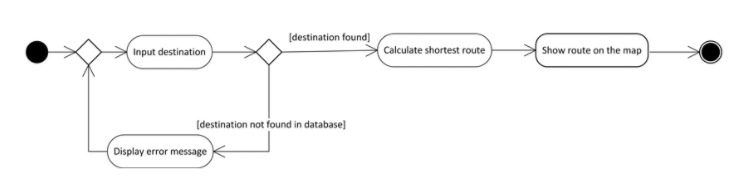
**SYSTEM DESIGN**

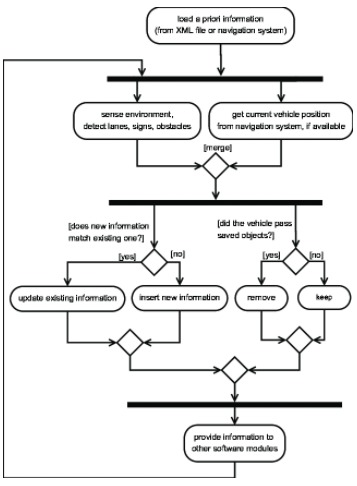
**3.1. UML Diagrams:**

This diagram represents how this algorithm can be implemented for real life cars.

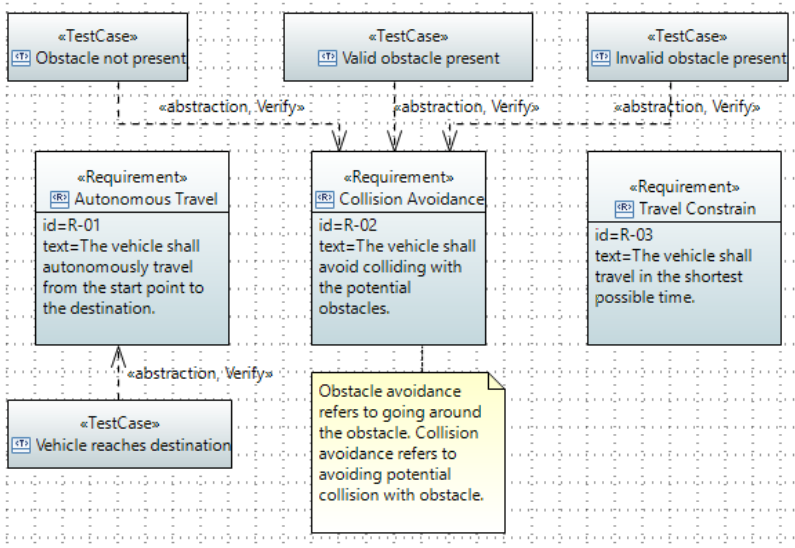


**3.2. Activity Diagrams:**

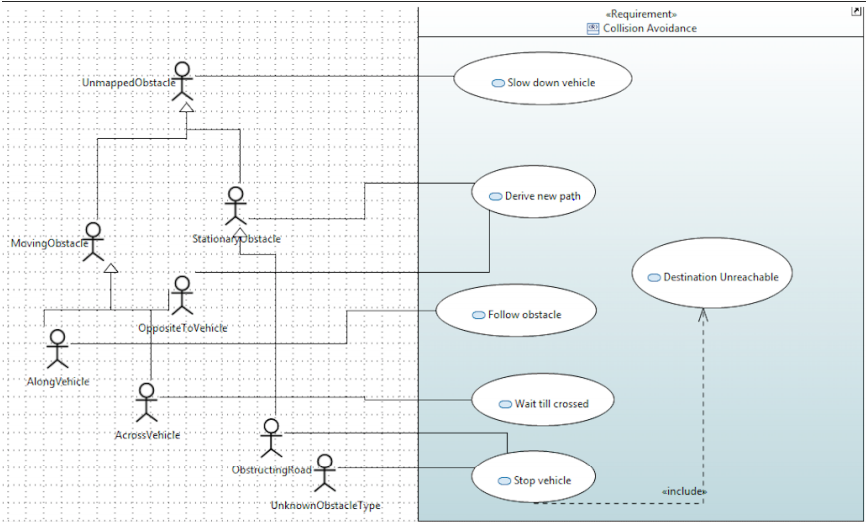


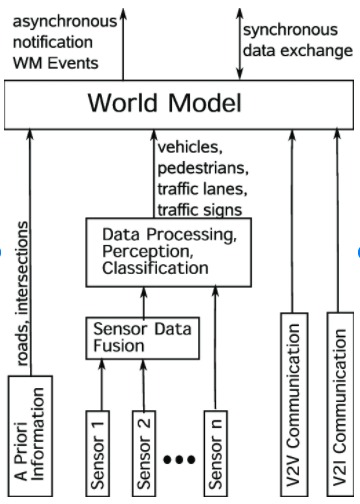


**3.3. Class Diagram:**

****

**3.4. Use-case Diagram:**

****

 **3.5. Control-flow Diagram:**

**CHAPTER 4**

**SYSTEM ARCHITECTURE**

**4.1 Architecture Overview:**

We have seen autonomous vehicles doing trials on the road. But that’s the last phase. Training an AI for autonomous vehicles goes roughly in 3 major phases:

1. Simulation / Supervised Learning — Using Computer or Lab
2. Controlled real world environment learning — Pre-built within campus
3. Real world learning — Hit the road

What we are going to see here is the first phase,Simulation learning where the AI will be trained by physics simulation in a computer model. In this phase, everything exists as data and software. The trained AI model will only produce signals like how much throttle, brake and steering is needed as output. Nothing is linked to any Hardware of physical car. It is in the second phase where the output from AI will be passed to hardware.

**4.2 Module Description:**

**ML-Agents:**

We have used ML-Agents module in which most of the ML training modules are written in Python using a vast variety of ML frameworks available. Unity ML-Agents does make use of Python and TensorFlow under the hood, but it provides a level of abstraction for Unity developers. ML-Agents is a framework using which you can train your models without even writing a single line of Python code.

**CineMachine:**

Cinemachine is a suite of modules for operating the Unity camera. Cinemachine solves the complex mathematics and logic of tracking targets, composing, blending, and cutting between shots. It is designed to significantly reduce the number of time-consuming manual manipulations and script revisions that take place during development.

The procedural nature of these modules makes Cinemachine bug-resistant. When you make adjustments—for example, change an animation, vehicle speed, terrain, or other GameObjects in your Scene—Cinemachine dynamically adjusts its behavior to make the best shot. There is no need, for example, to re-write camera scripts just because a character turns left instead of right.

**Rider Editor:**

**​**The JetBrains Rider editor package integrates support for the JetBrains Rider .NET Integrated Development Environment (IDE), into the Unity Editor. This package provides an end-point for Rider to call different Unity APIs and to generate .csproj and .sln files, which Rider uses to implement support for Unity in its plug-in.

This package ensures that IDE features like autocomplete suggestions and flagging dependency conflicts work in Rider. It uses .cproj and .sln files which store information about your project such as:

* Versioning information
* Build files
* Platform requirements
* Web server or database settings

Not all code in Unity is directly visible to code editors, particularly when using packages. This is because packages don’t provide their own .csproj files, and Unity doesn’t create them for installed packages by default. This means that IDE features like autocomplete suggestions and flagging dependency conflicts do not work with code in these packages. The purpose of this package is to produce the .csproj files that make these features possible by default when you use Rider.

**Test Framework:**

The Unity Test Framework (UTF) enables Unity users to test their code in both Edit Mode and Play Mode, and also on target platforms such as Standalone, Android, iOS, etc.

This package provides a standard test framework for users of Unity and developers at Unity so that both benefit from the same features and can write tests the same way.

UTF uses a Unity integration of NUnit library, which is an open-source unit testing library for .Net languages.

**TextMeshPro:**

TextMeshPro is the ultimate text solution for Unity. It’s the perfect replacement for Unity’s UI Text and the legacy Text Mesh.

Powerful and easy to use, TextMeshPro (also known as TMP) uses Advanced Text Rendering techniques along with a set of custom shaders; delivering substantial visual quality improvements while giving users incredible flexibility when it comes to text styling and texturing.

TextMeshPro provides Improved Control over text formatting and layout with features like character, word, line and paragraph spacing, kerning, justified text, Links, over 30 Rich Text Tags available, support for Multi Font & Sprites, Custom Styles and more.

Great performance. Since the geometry created by TextMeshPro uses two triangles per character just like Unity’s text components, this improved visual quality and flexibility comes at no additional performance cost.

**Timeline:**

Use Unity's Timeline to create cinematic content, game-play sequences, audio sequences, and complex particle effects.

Each cut-scene, cinematic, or game-play sequence that you create with Unity's Timeline consists of a Timeline Asset and a Timeline instance. The Timeline window creates and modifies Timeline Assets and Timeline instances simultaneously.

The Timeline Overview section includes details on the relationship between the Timeline window, Timeline Assets, and Timeline instances.

The Using Timeline section shows how to create Timeline Assets and Timeline instances, record basic animation, animate humanoids, and use other Timeline features.

**Unity UI:**

Unity UI is a UI toolkit for developing user interfaces for games and applications. It is a Game Object-based UI system that uses Components and the Game View to arrange, position, and style user interfaces.​You cannot use Unity UI to create or change user interfaces in the Unity Editor.

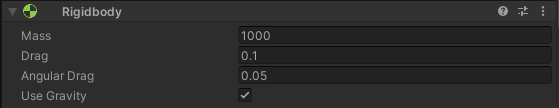
This documentation describes Unity UI features such as creating a Canvas, positioning and animating elements, defining user interactions, and sizing layouts automatically.

**Visual Studio Code Editor:**

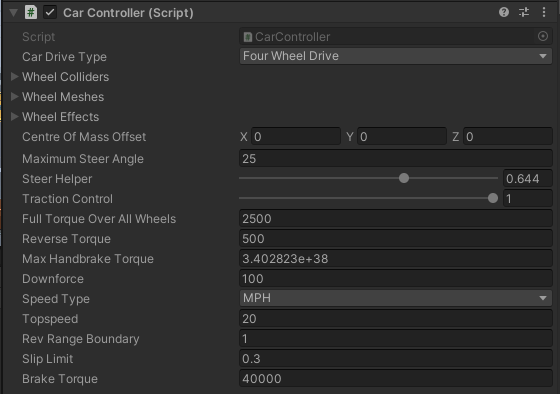
Code editor integration for supporting Visual Studio Code as code editor for unity. Adds support for generating csproj files for intellisense purposes, auto discovery of installations, etc.

**4.3 Pseudo code / Algorithm Description:**

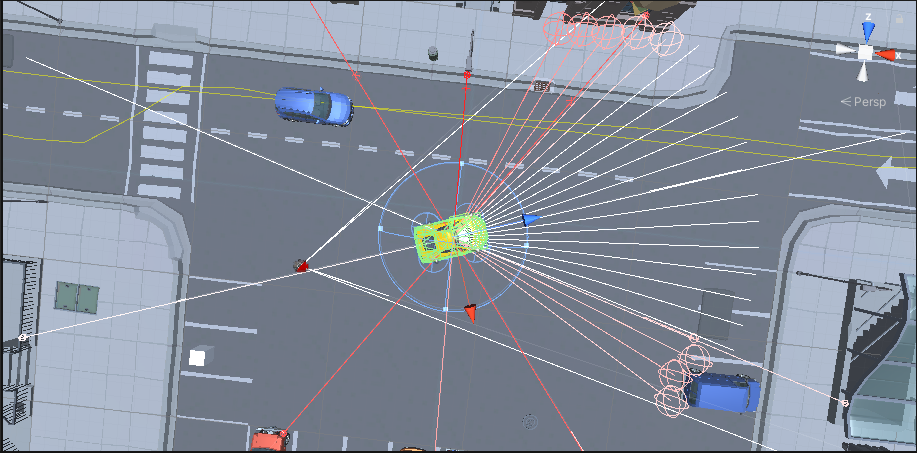
This simulation has 2 key parts, One is the “Car” which we are going to refer as agent going forward and the other is the environment against which we are training the car. In this project, the car element is created as a prefab and attached with a Rigid body physics element. The car that we are simulating is set with the following values.

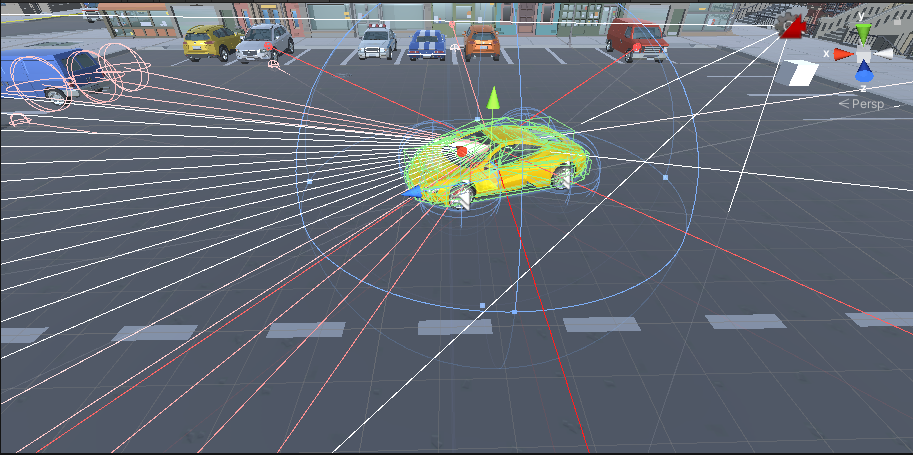


* A total body mass of about 1000kg
* A Car controller with Four wheel drive and Traction Control.
* Brake on all 4 wheels
* Motor Torque of 2500
* Brake/Reverse Torque of 40000 and
* Max. steering angle of 25 degrees

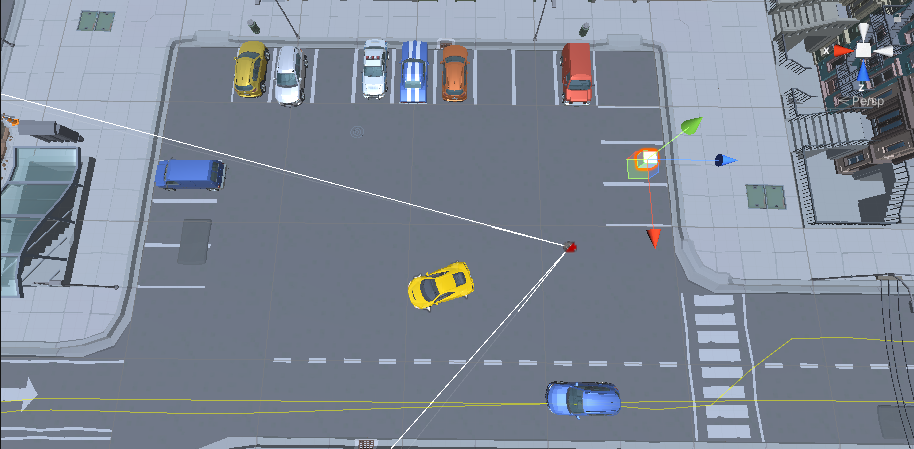


* Ray cast sensors in all four directions of the car. The ray cast sensors in real-world would be the LIDAR like vision sensors





**Reinforcement Learning**

Unity’s ML-agents use Reinforcement learning (Markov Decision Process) for training the agents. So we need a mechanism to let the agent know when it has properly occupied a parking slot. So the parking lot is setup with a trigger collider i.e., the cube in the parking spot.

* The agent will receive a penalty if it runs into other cars or runs on the platform etc....
* At the beginning of each training episode, reset the parking lot to normal if the ai has moved any cars and objects in the environment and place the agent in a random location in the parking lot.
* There are 3 signals received from the AI, first one is the amount of steering. second is the amount of throttle and third is the amount of brake applied for each simulation step.
* The state of the simulation is collected in CollectObservations, The velocity alignment with the nearest parking lot is calculated and given as a reward. It should be noted that the ray cast sensor values will get automatically added into the observation at each step.
* There is an accident detection routine which will give a penalty if the car runs into any barrier, tree or another car. The penalty value chosen should be bigger enough such that the agent doesn’t repeat it too often and small enough such that the agent is allowed to explore. Larger penalty value doesn’t allow the agent to learn.
* When the agent successfully enters a parking lot, this trigger collider is called and a Jackpot reward is given to the agent. A bonus value is calculated based on how aligned the agent is to the parking lot. A minimum bonus if the car is parked facing the wall, maximum bonus if it is parked facing the road.

**CHAPTER 5**

**SYSTEM IMPLEMENTATION**

At the beginning of each training episode, reset the parking lot occupancy to a different pattern and place the agent in a random location in the parking lot. The code used is shown below.

public override void OnEpisodeBegin()

{

HasCarUsedIt = true;

m\_Driving = true;

//transform.position = spawnPosition.position + new Vector3(+3, 0, Random.Range(-0f, +0f));

//transform.rotation = carrotationn;

int index = cspawnlist.Count;

int index2 = gspawnlist.Count;

int index3 = Plist.Count;

int randomIndex = Random.Range(0, index);

transform.position = cspawnlist[randomIndex] + new Vector3(+3, 0, Random.Range(-0f, +0f));

transform.rotation = Quaternion.Euler(0, Random.Range(0, 360), 0);

int randomIndex2 = Random.Range(0, index2);

//m\_Target.position = gspawnlist[randomIndex2];

//m\_Target.rotation = gspawnrlist[randomIndex2];

int randomIndex3 = Random.Range(0, index3);

Target = randomIndex3;

for (i = 0; i < Plist.Count; i++)

{

if (i == randomIndex3)

Plist[i].gameObject.SetActive(true);

else if (i != randomIndex3) {

Plist[i].gameObject.SetActive(false);

//CancelInvoke();

}

}

m\_Rigidbody.isKinematic = true;

m\_Rigidbody.isKinematic = false;

EnvironmentReset();

}

There are 3 signals received from the AI, first one is the amount of steering. second is the amount of throttle and third is the amount of brake applied for each simulation step. But brake and accelerator are obtained by just one variable named accel. If accel is between 0 and 1 it acts as accelerator whereas if its between -1 and 0 it acts brake.

public override void OnActionReceived(ActionBuffers Actions)

{

{

steer = Actions.ContinuousActions[0];

accel = Actions.ContinuousActions[1];

float brake = Actions.ContinuousActions[2];

// our target position starts off as the 'real' target position

Vector3 offsetTargetPos = m\_Target.position;

// calculate the local-relative position of the target, to steer towards

Vector3 localTarget = transform.InverseTransformPoint(offsetTargetPos);

dist1 = Vector3.Distance(transform.position, m\_Target.position);

//Debug.Log(dist1);

m\_CarController.Move(steer, accel, accel, 0f);

// if appropriate, stop driving when we're close enough to the target.

if (m\_StopWhenTargetReached && localTarget.magnitude < m\_ReachTargetThreshold)

{

//AddReward(0.1f);

//m\_Driving = false;

EndEpisode();

}

}

AddReward(-1.0f / MaxStep);

}

There is a negative reward with respect to the time taken by the ai to park which helps in speeding up the parking process.

The state of the simulation is collected in CollectObservations, The velocity alignment with the nearest parking lot is calculated and given as a reward. It should be noted that the ray cast sensor values will get automatically added into the observation at each step.

public override void CollectObservations(VectorSensor sensor)

{

sensor.AddObservation(transform.localPosition);

sensor.AddObservation(transform.rotation);

sensor.AddObservation(GameObject.Find("Parkingspot").transform.position);

sensor.AddObservation(m\_CarController.CurrentSpeed);

Vector3 dirToTarget = (m\_Target.transform.position - transform.position).normalized;

sensor.AddObservation(transform.position.normalized);

sensor.AddObservation(

this.transform.InverseTransformPoint(m\_Target.transform.position));

sensor.AddObservation(transform.InverseTransformDirection(dirToTarget));

sensor.AddObservation(transform.forward);

sensor.AddObservation(transform.right);

sensor.AddObservation(StepCount / MaxStep);

float velocityAlignment = Vector3.Dot(dirToTarget, m\_Rigidbody.velocity);

AddReward(0.001f \* velocityAlignment);

}

Apart from the above, there is an accident detection routine which will give a penalty if the car runs into any barrier, tree or another car. The penalty value chosen should be bigger enough such that the agent doesn’t repeat it too often and small enough such that the agent is allowed to explore. Larger penalty value doesn’t allow the agent to learn.

void OnCollisionEnter(Collision collider)

{

if (collider.transform.tag.ToLower() == "player" || collider.transform.tag.ToLower() == "buildings")

{

string objectName = gameObject.name;

//Debug.Log(objectName);

TakeAwayPoints();

}

if (collider.transform.tag.ToLower() == "platform" || collider.transform.tag.ToLower() == "cars" || collider.transform.tag.ToLower() == "barrier" || collider.transform.tag.ToLower() == "poles" || collider.transform.tag.ToLower() == "bench" || collider.transform.tag.ToLower() == "tree")

{

AddReward(-0.1f);

}

}

When the agent successfully enters a parking lot, this trigger collider is called and a Jackpot reward is given to the agent. A bonus value is calculated based on how aligned the agent is to the parking lot. A minimum bonus if the car is parked facing the wall, maximum bonus if it is parked facing the road.

if (((transform.rotation.eulerAngles.y <= (m\_Target.transform.rotation.eulerAngles.y + 10)) && (transform.rotation.eulerAngles.y >= (m\_Target.transform.rotation.eulerAngles.y - 10))) || !enforceGoalMinRotation)

{

if (((transform.position.x <= m\_Target.position.x + 0.5) && (transform.position.x >= m\_Target.position.x - 0.5)) && ((transform.position.z <= m\_Target.position.z + 0.5) && (transform.position.z >= m\_Target.position.z - 0.5)))

{

Debug.Log("hit");

GivePoints(goalReward, true);

}

HasCarUsedIt = false;

//GivePoints(goalReward, true);

}

else if (((transform.rotation.eulerAngles.y <= (m\_Target.transform.rotation.eulerAngles.y - 170)) && (transform.rotation.eulerAngles.y >= (m\_Target.transform.rotation.eulerAngles.y - 180))) || ((transform.rotation.eulerAngles.y <= (m\_Target.transform.rotation.eulerAngles.y + 170)) && (transform.rotation.eulerAngles.y >= (m\_Target.transform.rotation.eulerAngles.y + 180))) || !enforceGoalMinRotation)

{

if (((transform.position.x <= m\_Target.position.x + 0.5) && (transform.position.x >= m\_Target.position.x - 0.5)) && ((transform.position.z <= m\_Target.position.z + 0.5) && (transform.position.z >= m\_Target.position.z - 0.5)))

{

Debug.Log("hit");

AddReward(1.0f);

GivePoints(goalReward, true);

}

HasCarUsedIt = false;

//GivePoints(goalReward, true);

}

The Training config yaml is setup such that we have 2 hidden layers with 128 nodes per layer. And we run a training session of 10 million cycles.

behaviors:

CarBehaviour:

trainer\_type: ppo

hyperparameters:

batch\_size: 128

buffer\_size: 8192

learning\_rate: 0.0003

beta: 0.01

epsilon: 0.2

lambd: 0.95

num\_epoch: 3

learning\_rate\_schedule: linear

network\_settings:

normalize: false

hidden\_units: 128

num\_layers: 2

reward\_signals:

extrinsic:

gamma: 0.99

strength: 1.0

gail:

strength: 0.5

demo\_path: D:\MachineLearning\SelfDriving\demo\Selfdrive.demo

behavioral\_cloning:

strength: 0.5

demo\_path: D:\MachineLearning\SelfDriving\demo\Selfdrive.demo

time\_horizon: 64

max\_steps: 10.0e6

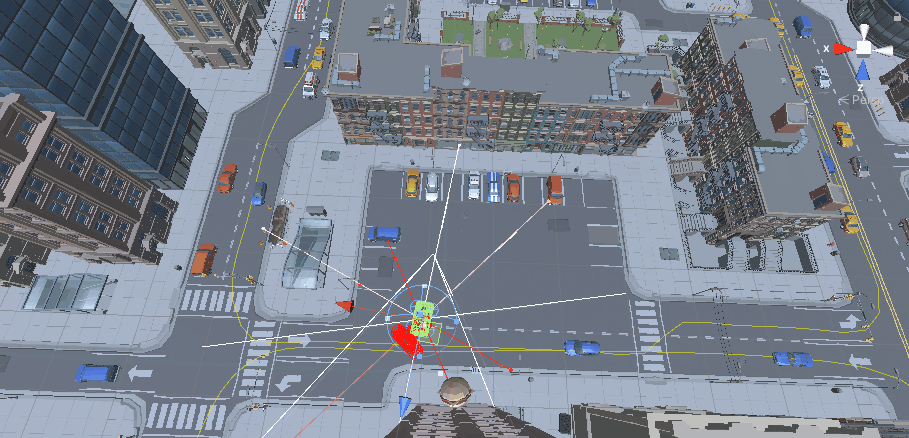
summary\_freq: 5000

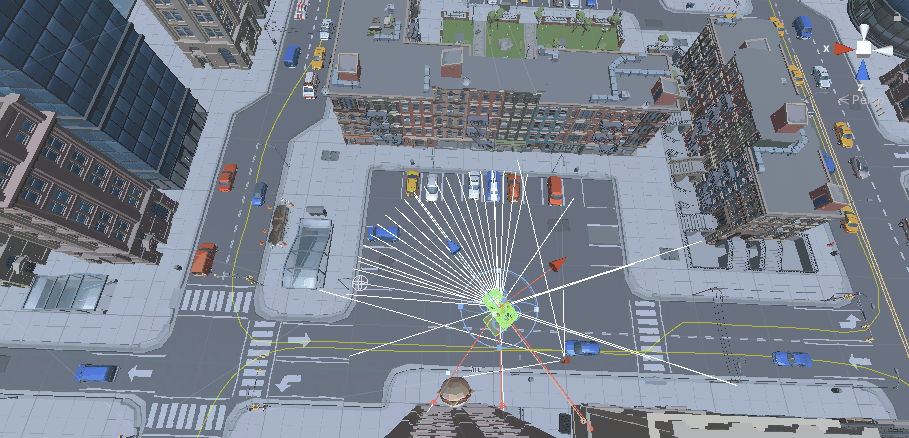
threaded: true

**CHAPTER 6**

**SYSTEM TESTING**

During the training it is found that the agent doesn’t learn properly even after 5 million cycles if we always place the agent at the entry of the parking lot. So randomly placing the agent near to an empty parking lot increased its chances of collecting a reward and there by increased the variance in learning. The simulation manager is altered to do this randomization 50% of the time. The rest of the time the agent is placed in the parking entry.





**CHAPTER 7**

**CONCLUSION**

Thus the self-parking car simulation is done and made working properly in Unity Api. This same prefab developed using reinforcement learning to implement self driving cars in any game developed and this idea can be used in self parking cars in real life using this idea. Time saving is going to play a major role in the upcoming years and this self parking system will help in saving a lot of it. This has the potential to provide confidence to even amateur drivers while parking at tight spots. Using this prefab in game development will help us to save a lot of time and labor put into programming.

References:

1. <https://medium.com/xrpractices/autonomous-car-parking-using-ml-agents-d780a366fe46>
2. <https://www.codemonkey.com/>
3. <https://towardsdatascience.com/reinforcement-learning-a-self-driving-car-ai-in-unity-60b0e7a10d9e>
4. <https://bhargav265.github.io/Simulation_of_self_driving_car/>
5. https://www.youtube.com/watch?v=pdE3fXRoyjI