“Car Game”

MINOR PROJECT SYNOPSIS (KCA-353)

SUBMITTED TO

DR. A.P.J. ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW(U.P.)

FOR THE PARTIAL FULFILLMENT OF THE DEGREE OF MASTER

IN COMPUTER APPLICATION SESSION (2nd Year)

Under the Guidance of:

Ms. Jyoti Tripathi
Assistant professor
School of IT
IMS-Noida

Submitted by:

Mohit Singh
(2100980140029)

INSTITUTE OF MANAGEMENT STUDIES, NOIDA

PROFORMA FOR APPROVAL OF MCA MINOR PROJECT (KCA-353)

1. Roll No.: 210098014029
2. Name of the student: Mohit Singh
3. E-mail: mohitsingh.mca21060@imsnoida.com
4. Mob. No.: 9718323563
5. Title of the Minor Project: Car game
6. Name of the Guide: Ms. Jyoti Tripathi

For Office Use Only:

Signature of the Mentor

Approved Not Approved Date: ----------

Suggestions (if any): -

1

2

3

4

5

6

7

8

9

10

INDEX

S. No	Particulars	Page No.	Date	T. Sign
1	Cover And Title Page			
2	Synopsis Approval			
3	Index			
4	Acknowledgement			
5	Introduction And Objective of The Project			
6	Feasibility test			
7	Development Process			
8	Game Engine			
9	Screen Shots			
10	Coding			
11	TESTING AND MAINTAINING			
12	H/W And S/W Requirement			
13	Future scope			
14	Limitations of the project			
15	References/Bibliography			

ACKNOWLEDGEMENT

I am very grateful to my Minor project (KCA-353) **Ms. Jyoti Tripathi**, for giving his/her valuable time and constructive guidance in preparing the Synopsis-Minor Project (KCA-353). It would not have been possible to work on this project without his/her kind encouragement and valuable guidance.

DATE: 23/11/2022

SIGNATURE:

CERTIFICATE OF ORIGINALITY

I hereby declare that MCA MINI Project (KCA-353) titled **Car Game** submitted to IT Department, IMS Noida, which is affiliated with **DR. A.P.J. ABDUL KALAM TECHNICAL UNIVERSITY LUCKNOW (U.P.)** for the partial fulfillment of the degree of Master of Computer Application, in Session 2021-2023. This has not previously formed the basis for the award of any other degree, diploma or other title from any other University.

PLACE: IMS NOIDA SECTOR-62

DATE: 23/11/2022

SIGNATURE:

Introduction And Objective of The Project

Introduction:

Developing software applications is a time-consuming process, and with time-consuming processes come high costs. During the last years, several software development methodologies, often known as agile software development, have become widely used by software developers to address this issue. Many different development methodologies can be more or less good, depending of the task and application type.

One of the software development methodologies is the evolutionary software method, which, as the name hints, takes on an evolutionary approach to the problem, and allows the project to evolve through different stages of the project. Our case study will show how well this evolutionary approach worked on our project where I choose to develop an android game. Some requirements for the game were given from the beginning, such as:

3D graphics - The game must contain 3D models, and render these in the game. 3D environments were never a requirement, and platform games with 2D environment could still open up for 3D objects.

Impressive result: - The game result must impress whoever plays the game. It should last long, and make the players come back and play it over and over again.

Graphical effects: - To achieve an impressive result, we would need to add modern graphical effects, such as real-time rendered soft shadows, motion blur, and ambient occlusion.

Working with these requirements, I decided to use Unity as our platform to develop the game. This decision was made with regard to that the platform had many in-built tools and provided a good framework for us to get started with the development as fast as possible.

Objectives:

To have smooth and fun gameplay and have people spend as much time as they can.

Development Process

In a software development project, the resulting product is required to fulfil many different qualities. Examples of such quality requirements are: robustness, availability, maintain- ability, dependability and usability. To meet such varying demands, it is important to base the work on a well-prepared strategy. In software engineering, the term for such a strategy is commonly known as software process, which is built on one or several software process models.

What Is a Software Process Model?

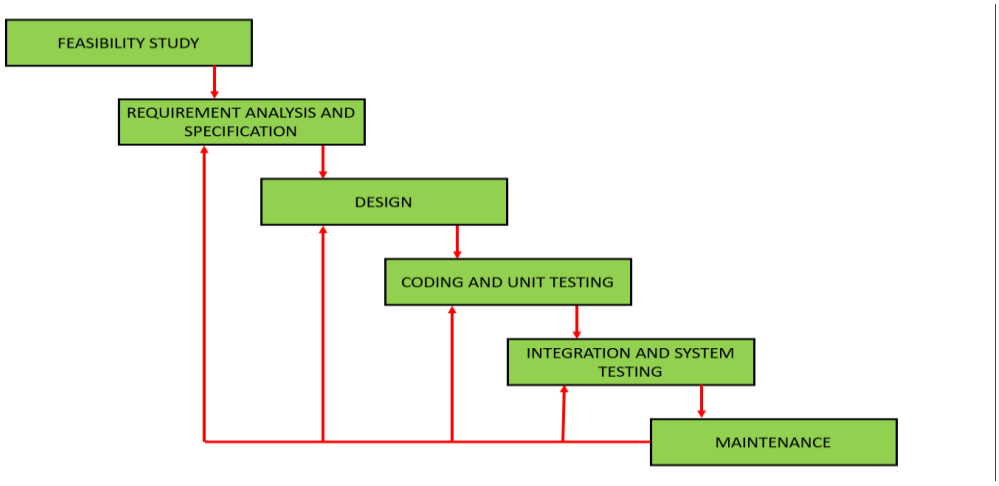
A software process model is a theoretical philosophy that describes the best way of developing software. Based on one or several models, a software process is formed providing guidance on how to operate. A software process model may also be described as an abstract representation of a soft-ware process. The concept of the process model is similar to an abstract java class, which cannot be instantiated, but it can be implemented by another class, thus providing basic guidelines for that other class. A model may for example demand customer involvement, but it does not state exactly how. A process implementing that model should involve the customer in the process' activities, but is free to choose how. There is not only one type of process model, but two. The first one is the most common, and described above. The second type of process model is called a process paradigm, which is a model even more general than an ordinary process model. Such a model does not hold any details on how the activities that lead to the completion of a project should be performed, but what it does hold is basic guidelines of how to develop software, and assumptions about what sort of project could benefit from implementing a particular model. With this in regard, one can conclude that a process paradigm provides a framework that may be adapted to form a process which suits a particular project. There are three major process paradigms that are commonly used today in software engineering practice; the waterfall model, component-based software engineering and evolutionary development.

Iterative Waterfall Model

The waterfall model is recommended for large and complex systems that have a long life-time2. Some systems which carry these attributes are also critical systems.

It is believed that the waterfall model would be an appropriate choice when developing a critical system, since the model emphasizes on thoroughness. The basic concept is to take all the activities and treat them separately. One activity is always followed by another, in the same way water travels down some falls. This description becomes even more obvious when looking at a visualization of the model.

1. **Requirements definition** All requirements on the system are found by talking to system users. Example of requirements can be services, constraints and goals, such as “We want a webpage that colour-blind people can enjoy”.
2. **System and software design** In this activity, the overall architecture of the system is established.
3. **Implementation and unit testing** The software is implemented in units which also are tested.
4. **Integration and system testing** The units are merged together into a complete system. Further testing is required.
5. **Operation and maintenance** The system is delivered to the customer and put into operation. “Bugs” are almost always found, and therefore the system required bug-fixing and maintenance.



Feasibility test

Economical Feasibility

It is very important aspect to be considered while developing a project No money was spent in making of the game. All is available for free.

Technical Feasibility

This included the study of function, performance and constraints that may affect the ability to achieve an acceptable system.

For running the game, you need at least android 7.0 and 2 Gb RAM.

For running unity 64-bit OS, processor i3, window 7 and above is required.

Operational Feasibility

the system is fully GUI based that is very user friendly and all inputs to be taken all self-explanatory. It is easy to understand and use.

Game Engine

Game Framework:

To save time in our development process, we choose to use Unity, when developing our game.

Unity:

Unity is a cross-platform game engine developed by Unity Technologies and used to develop video games for PC, consoles, mobile devices and websites. First announced only for Mac OS, at Apple's Worldwide Developers Conference in 2005, it has since been extended to target more than fifteen platforms. It is now the default software development kit (SDK) for the Wii U.

Unity Pro is available for a fee and Unity Personal has no fee; it is available for any use to individuals or companies with less than US$100,000 of annual gross revenue. On March 3, 2015 with the release of Unity 5.0, Unity Technologies made the complete engine available for free including all features, less source code and support. Unity is noted for an ability to target games to multiple platforms.

Five versions of Unity have been released. In 2006 at the 2006 WWDC trade show, Apple, Inc. named Unity as the runner up for its Best Use of Mac OS X Graphics category.

Car Physics:

In this scene, we want to start to affect the player's car through the use of forces and torques (angular forces) rather than position and rotation directly. This has several advantages including more realistic motion, but more importantly, we don't have to multiply movements by Time. The deltaTime when working with physics because forces are already time independent. Once again, start by opening up the Basic Setup scene, then click on Car Control -> 4 Physics -> carcontrol.js and examine it in the Inspector panel.

In this section we will build the first version of a raycast car. A raycast car is a car which slides over the ground. Its wheels don't spin as at drives forward; in this respect it is actually more like a snowmobile than a car. Every frame we apply a force to move it forward. It doesn't move sideways because we will use a Physic Material with

anisotropic friction. Anisotropic friction is a necessary part of a game involving motion because it allows you to have different friction values for forward or sideways motion. When moving sideways we will use a much higher friction value than forward. This will make the car tend to slide forward instead of sideways and follow the front wheels when they are rotated. We'll use a simple, three-step process to build the first version of a raycast car for the player to control. First, we want to create colliders for the car: add a box collider to the car body of Player Car, then add raycast colliders to each of the four wheels. These are added to an object by selecting the object in the Scene view or the Hierarchy panel, and then selecting Component >> Dynamics from the menu bar. A raycast collider simply uses a ray for collision detection instead of a volume shape; i.e., a sphere. Also attach a Rigid Body with a mass of 10 to the Player Car. Second, create a Physic Material and set it up as shown in the following screenshot. Rename it Wheel Material and move it to the 4 Physics directory. Assign this new material to all the 4 wheels using drag and drop.

Basic Camera:

Let's start by first simply getting the main camera positioned above and behind the player car. Open our previous scene's Physics.unity scene. Camera Control Scripts -> 1 Basic Follow Camera and name the file Camera. This scene contains all of the elements and scripts from the previous section of the tutorial. Now attach the Camera.js script to the main camera. The Camera.js script has a distance and height variable that we will be able to modify from the Inspector, but also a "target" variable that we will need to assign to the player's car. Connect this variable to the player car object the same way we connected the wheel control variables to the wheel objects of the car object.

The Smooth Camera.js script is very simple.

1. We set up variables for height, distance and additionally for damping.
2. We calculate both current and wanted rotation and height for the camera.
3. We dampen the height and rotation by using the Lerp function.

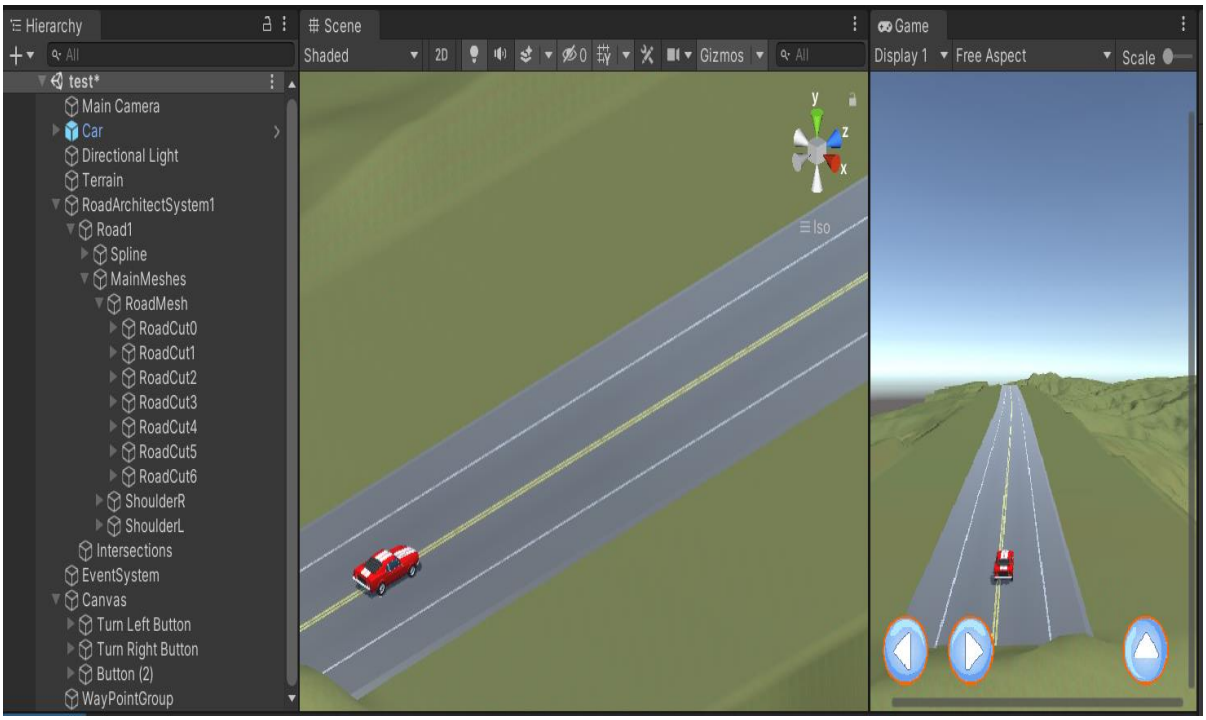
4. We convert our rotation calculation from degrees into radians so that the Quaternion interface understands it.

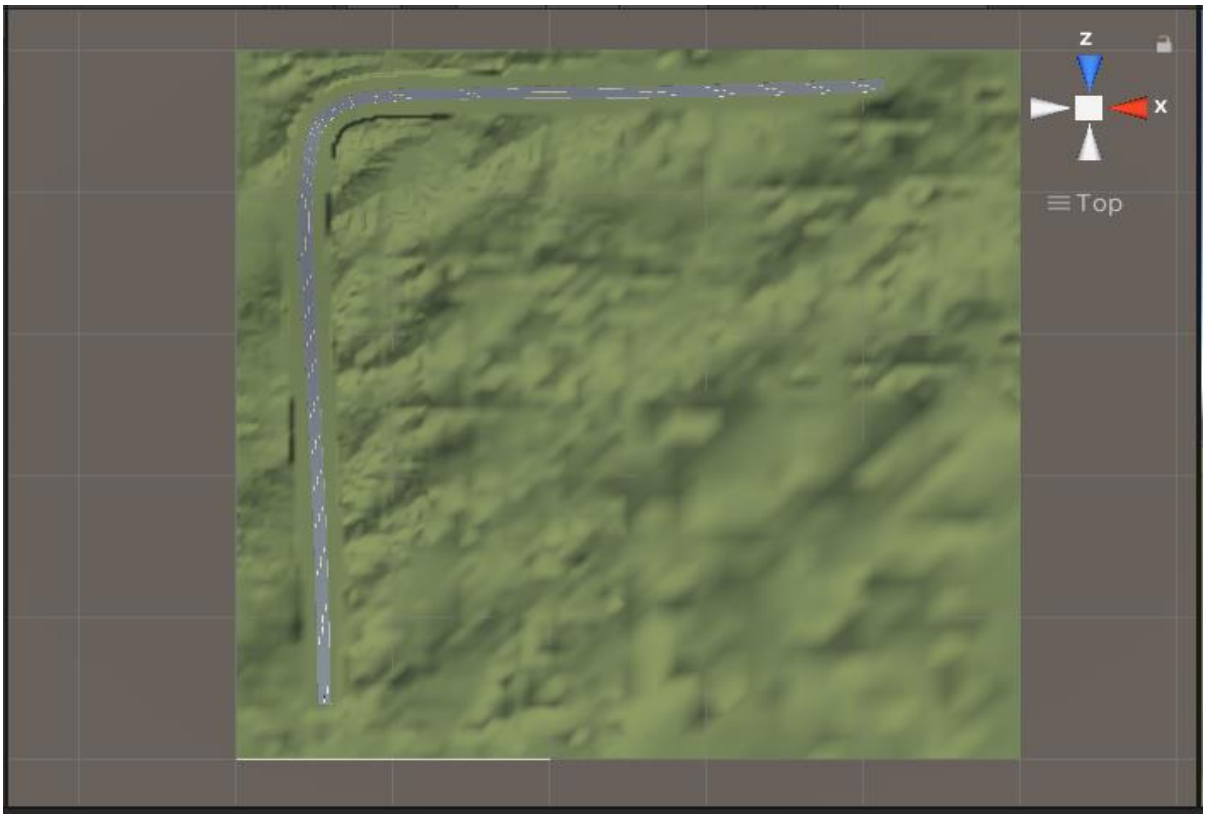
5. In this part we finally position the camera where we want it and point the camera to always look at the target. Run the scene and drive the player car to see the improved camera in action. Notice that we're using specifically the Mathf.LerpAngle to damp the rotation around the player car's vertical (y) axis, and using Mathf. Lerp to damp the height. It also uses some other basic functions built into Unity such as EulerAngles, Time. deltaTime and others. Most of the rest of the script uses basic variables and functions to move the camera with the car. It's now time to add opponent vehicles and program to race around the track against the player. Otherwise known as the "cool stuff."

WayPoints:

The most common artificial intelligence in a racing game is waypoint navigation by carefully placing waypoints (nodes) in the game environment to move the game-controlled characters between each point. This is a very time consuming and CPU intensive problem. Using the A* algorithm can effectively solve the path finding problem in a static racing game environment; therefore, we present two modified A* algorithm instead of putting waypoints by hand and minimum the lap time. Finally, we propose a more general dynamic algorithm which can solve the random obstacles avoidance problem in a racing game. All the three algorithms are able to find the path for a car racing game and can save the most import resource in game, CPU cycles.

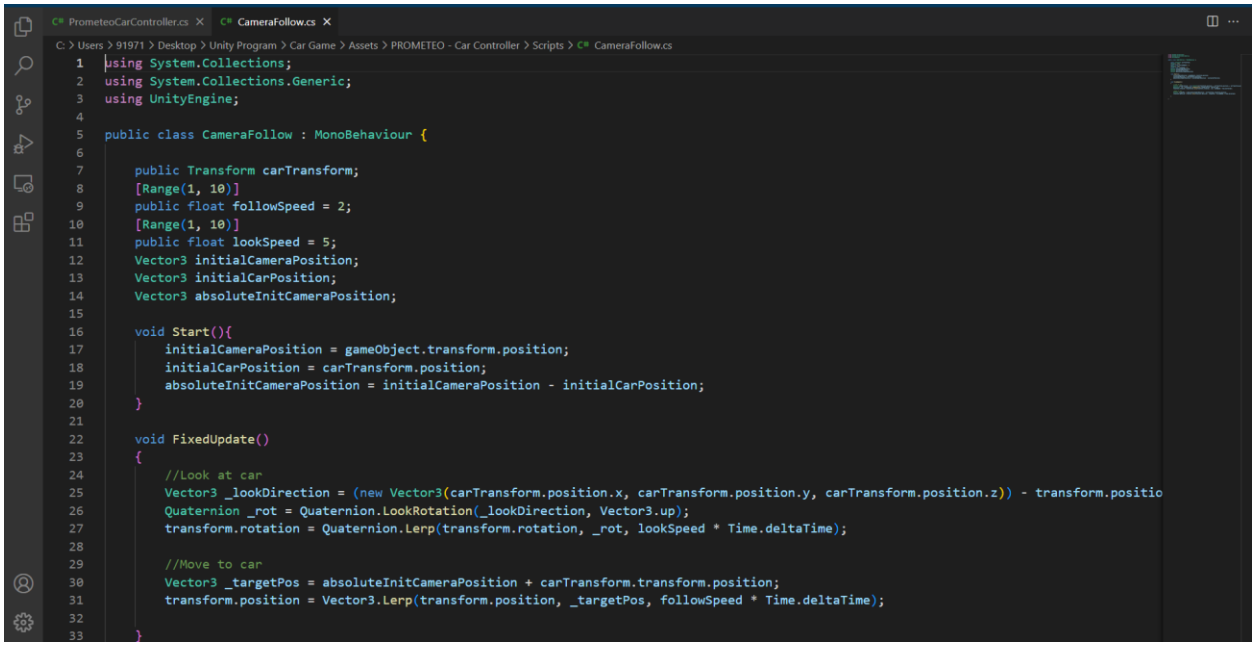
Screen Shots



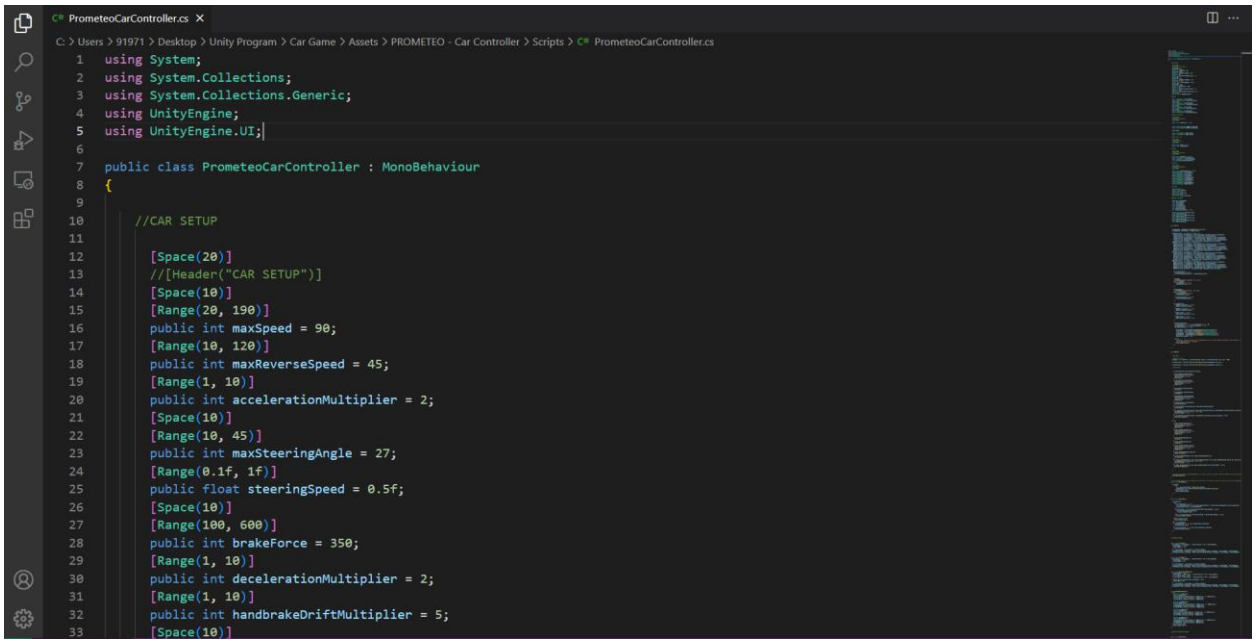


Coding

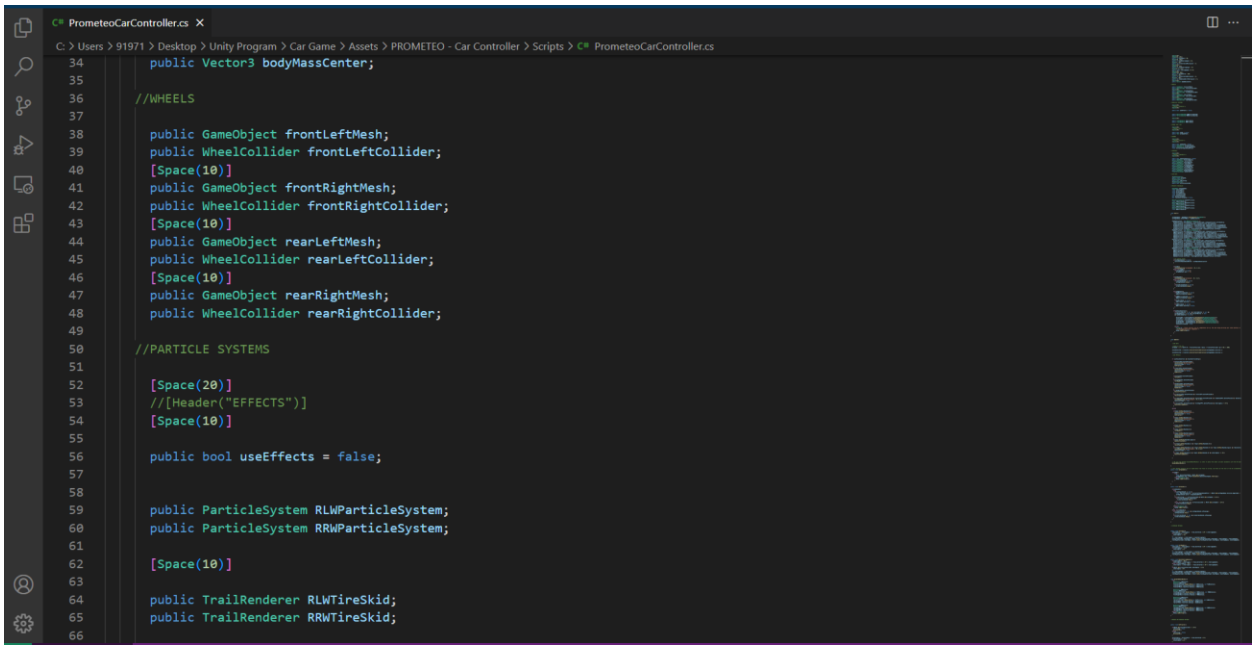
C# - C# is used as C# is the only language that Unity supports natively. And it is easy to use. It supports OOPS concept.

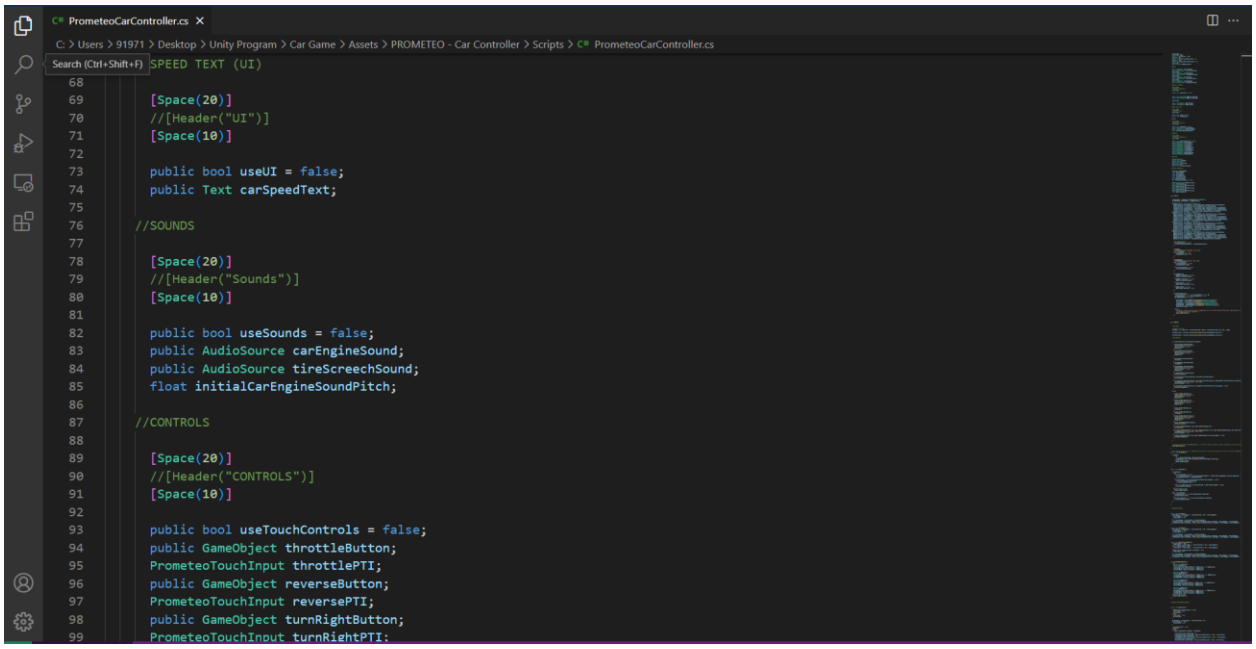


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraFollow : MonoBehaviour {

    public Transform carTransform;
    [Range(1, 10)]
    public float followSpeed = 2;
    [Range(1, 10)]
    public float lookSpeed = 5;
    Vector3 initialCameraPosition;
    Vector3 initialCarPosition;
    Vector3 absoluteInitCameraPosition;

    void Start(){
        initialCameraPosition = gameObject.transform.position;
        initialCarPosition = carTransform.position;
        absoluteInitCameraPosition = initialCameraPosition - initialCarPosition;
    }

    void FixedUpdate()
    {
        //Look at car
        Vector3 _lookDirection = (new Vector3(carTransform.position.x, carTransform.position.y, carTransform.position.z)) - transform.positio
        Quaternion _rot = Quaternion.LookRotation(_lookDirection, Vector3.up);
        transform.rotation = Quaternion.Lerp(transform.rotation, _rot, lookSpeed * Time.deltaTime);

        //Move to car
        Vector3 _targetPos = absoluteInitCameraPosition + carTransform.transform.position;
        transform.position = Vector3.Lerp(transform.position, _targetPos, followSpeed * Time.deltaTime);

    }
```



```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PrometeoCarController : MonoBehaviour
{

    //CAR SETUP

      [Space(20)]
      //[Header("CAR SETUP")]
      [Space(10)]
      [Range(20, 190)]
      public int maxSpeed = 90;
      [Range(10, 120)]
      public int maxReverseSpeed = 45;
      [Range(1, 10)]
      public int accelerationMultiplier = 2;
      [Space(10)]
      [Range(10, 45)]
      public int maxSteeringAngle = 27;
      [Range(0.1f, 1f)]
      public float steeringSpeed = 0.5f;
      [Space(10)]
      [Range(100, 600)]
      public int brakeForce = 350;
      [Range(1, 10)]
      public int decelerationMultiplier = 2;
      [Range(1, 10)]
      public int handbrakeDriftMultiplier = 5;
      [Space(10)]
```


PrometeoCarController.cs
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs
public Vector3 bodyMassCenter;
//WHEELS
public GameObject frontLeftMesh;
public WheelCollider frontLeftCollider;
[Space(10)]
public GameObject frontRightMesh;
public WheelCollider frontRightCollider;
[Space(10)]
public GameObject rearLeftMesh;
public WheelCollider rearLeftCollider;
[Space(10)]
public GameObject rearRightMesh;
public WheelCollider rearRightCollider;
//PARTICLE SYSTEMS
[Space(20)]
//[Header("EFFECTS")]
[Space(10)]
public bool useEffects = false;
public ParticleSystem RLWParticleSystem;
public ParticleSystem RRWParticleSystem;
[Space(10)]
public TrailRenderer RLWTireSkid;
public TrailRenderer RRWTireSkid;


PrometeoCarController.cs
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs
Search (Ctrl+Shift+F)
SPEED TEXT (UI)
[Space(20)]
//[Header("UI")]
[Space(10)]
public bool useUI = false;
public Text carSpeedText;
//SOUNDS
[Space(20)]
//[Header("Sounds")]
[Space(10)]
public bool useSounds = false;
public AudioSource carEngineSound;
public AudioSource tireScreechSound;
float initialCarEngineSoundPitch;
//CONTROLS
[Space(20)]
//[Header("CONTROLS")]
[Space(10)]
public bool useTouchControls = false;
public GameObject throttleButton;
PrometeoTouchInput throttlePTI;
public GameObject reverseButton;
PrometeoTouchInput reversePTI;
public GameObject turnRightButton;
PrometeoTouchInput turnRightPTI;

PrometeoCarController.cs

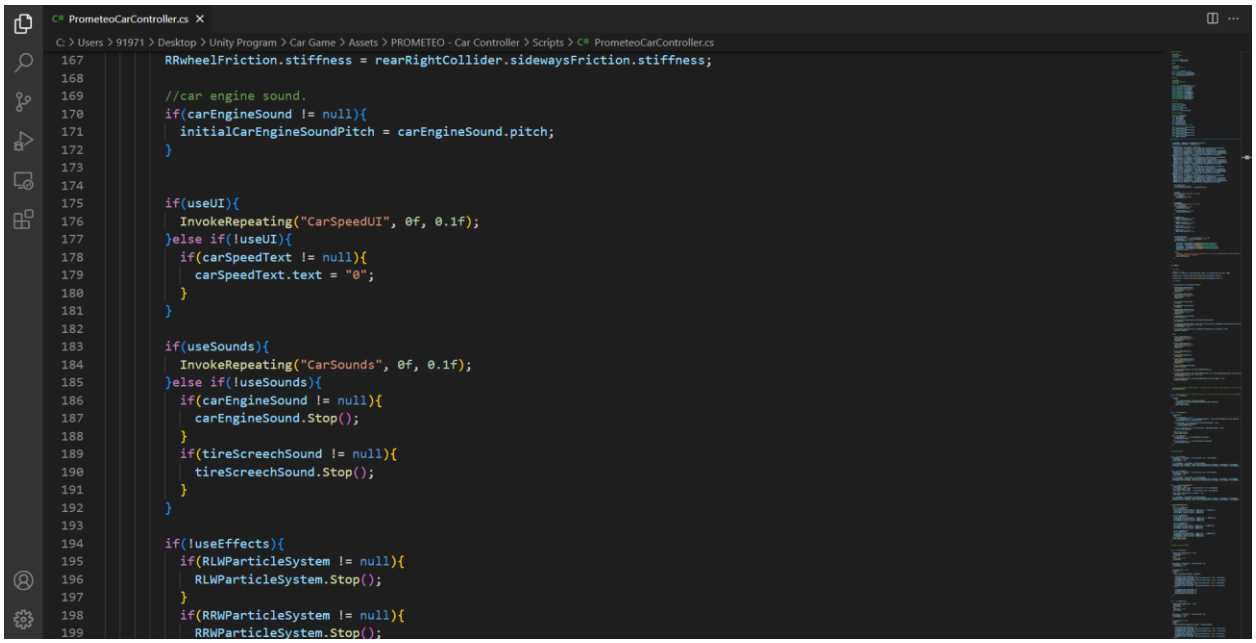
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs

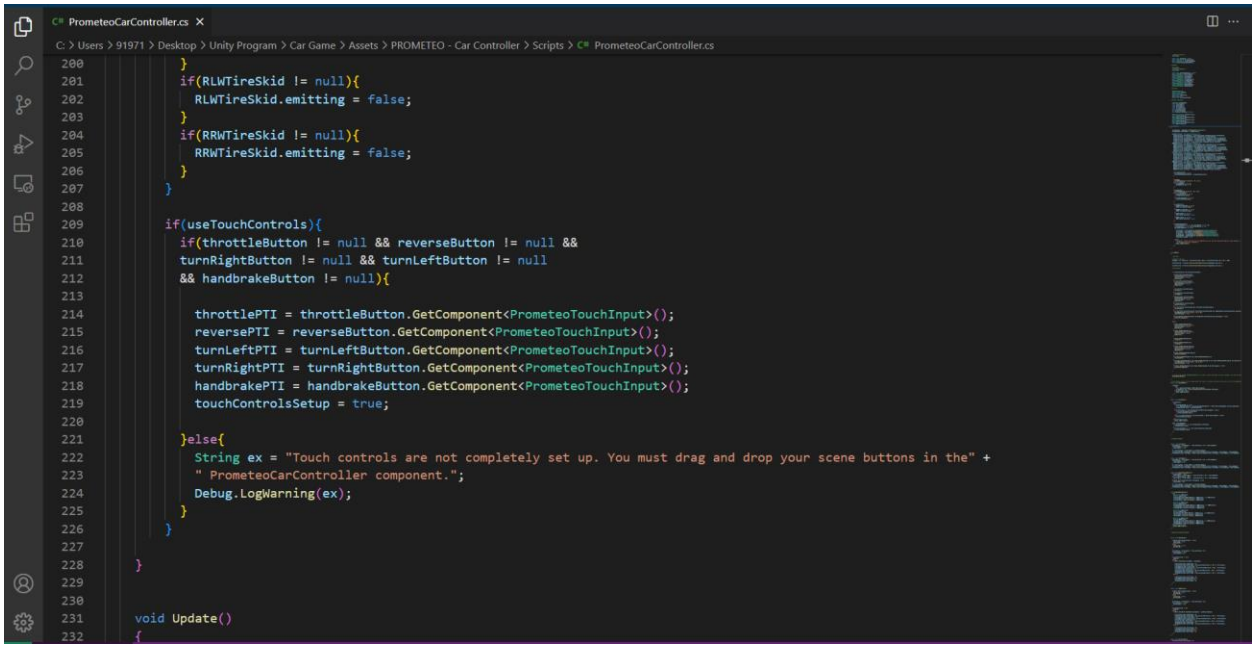
```
100     public GameObject turnLeftButton;
101     PrometeoTouchInput turnLeftPTI;
102     public GameObject handbrakeButton;
103     PrometeoTouchInput handbrakePTI;
104
105   //CAR DATA
106
107     [HideInInspector]
108     public float carSpeed;
109     [HideInInspector]
110     public bool isDrifting;
111     [HideInInspector]
112     public bool isTractionLocked;
113
114   //PRIVATE VARIABLES
115
116     Rigidbody carRigidbody;
117     float steeringAxis;
118     float throttleAxis;
119     float driftingAxis;
120     float localVelocityZ;
121     float localVelocityX;
122     bool deceleratingCar;
123     bool touchControlsSetup = false;
124
125     WheelFrictionCurve FLwheelFriction;
126     float FLWextremumSlip;
127     WheelFrictionCurve FRwheelFriction;
128     float FRWextremumSlip;
129     WheelFrictionCurve RLwheelFriction;
130     float RLWextremumSlip;
131     WheelFrictionCurve RRwheelFriction;
132     float RRWextremumSlip;
```

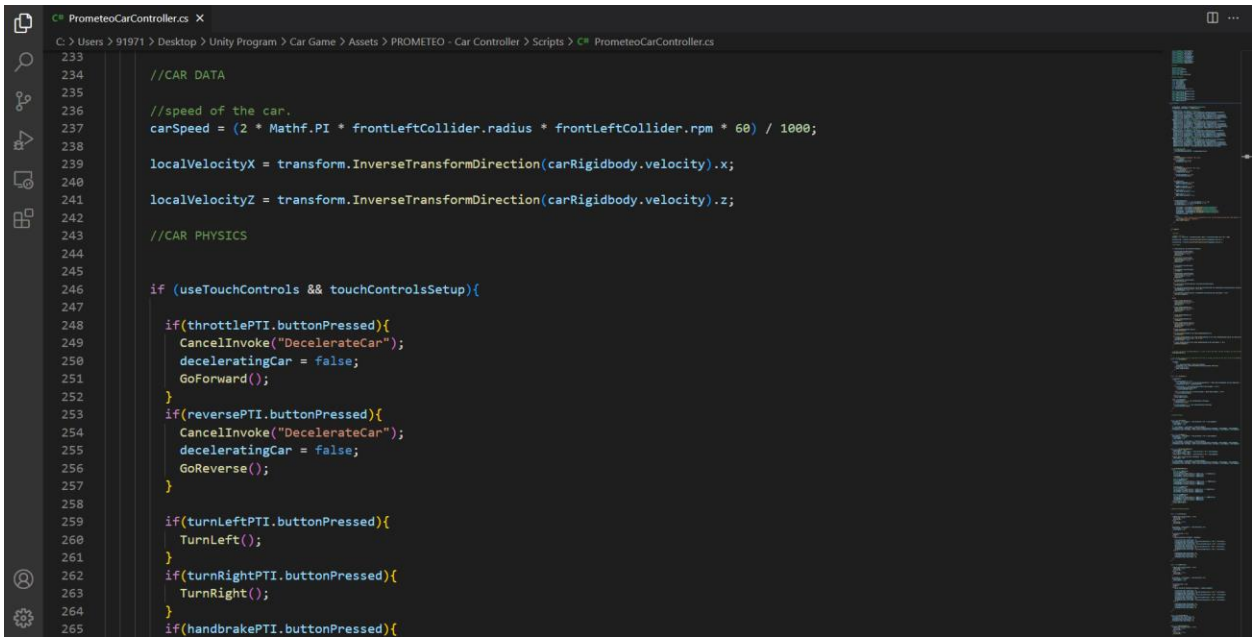
PrometeoCarController.cs

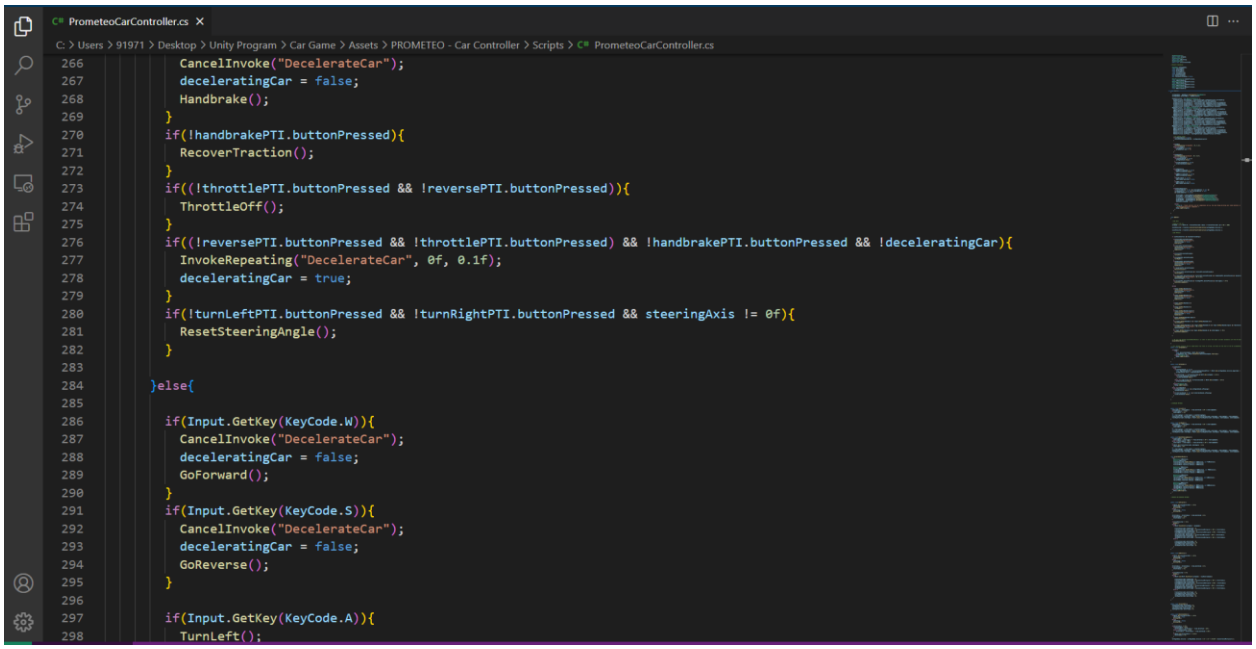
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs

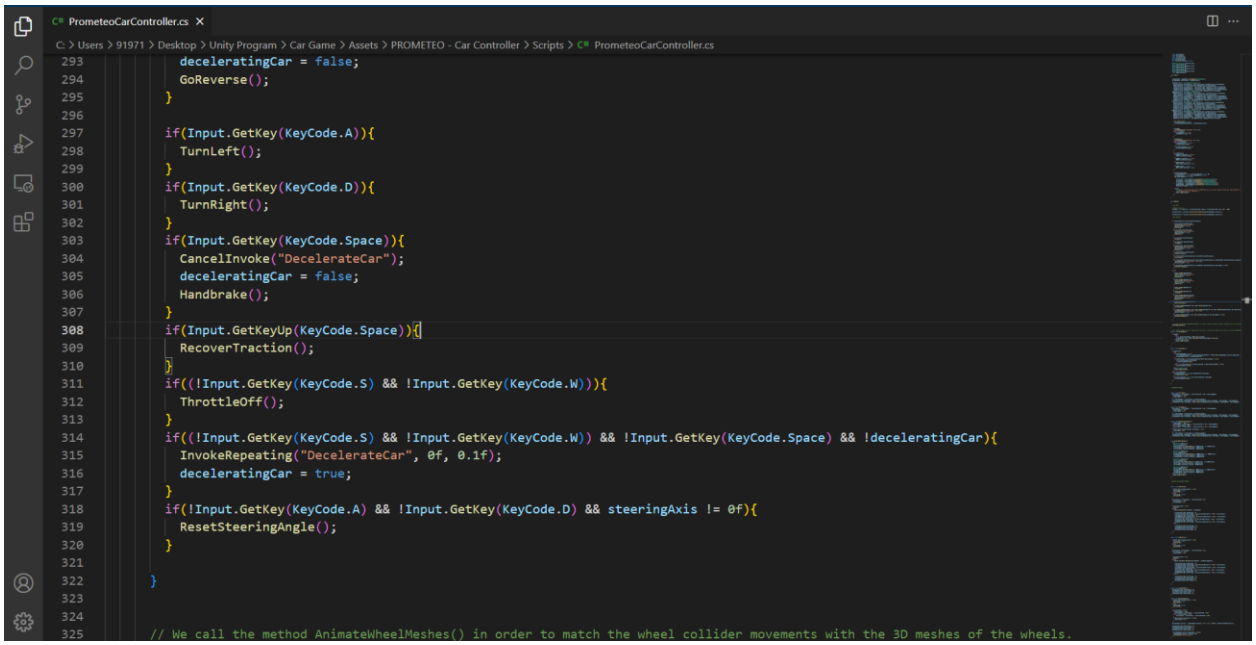
```
134   void Start()
135   {
136
137     carRigidbody = gameObject.GetComponent<Rigidbody>();
138     carRigidbody.centerOfMass = bodyMassCenter;
139
140     FLwheelFriction = new WheelFrictionCurve ();
141       FLwheelFriction.extremumSlip = frontLeftCollider.sidewaysFriction.extremumSlip;
142       FLWextremumSlip = frontLeftCollider.sidewaysFriction.extremumSlip;
143       FLwheelFriction.extremumValue = frontLeftCollider.sidewaysFriction.extremumValue;
144       FLwheelFriction.asymptoteSlip = frontLeftCollider.sidewaysFriction.asymptoteSlip;
145       FLwheelFriction.asymptoteValue = frontLeftCollider.sidewaysFriction.asymptoteValue;
146       FLwheelFriction.stiffness = frontLeftCollider.sidewaysFriction.stiffness;
147     FRwheelFriction = new WheelFrictionCurve ();
148       FRwheelFriction.extremumSlip = frontRightCollider.sidewaysFriction.extremumSlip;
149       FRWextremumSlip = frontRightCollider.sidewaysFriction.extremumSlip;
150       FRwheelFriction.extremumValue = frontRightCollider.sidewaysFriction.extremumValue;
151       FRwheelFriction.asymptoteSlip = frontRightCollider.sidewaysFriction.asymptoteSlip;
152       FRwheelFriction.asymptoteValue = frontRightCollider.sidewaysFriction.asymptoteValue;
153       FRwheelFriction.stiffness = frontRightCollider.sidewaysFriction.stiffness;
154     RLwheelFriction = new WheelFrictionCurve ();
155       RLwheelFriction.extremumSlip = rearLeftCollider.sidewaysFriction.extremumSlip;
156       RLWextremumSlip = rearLeftCollider.sidewaysFriction.extremumSlip;
157       RLwheelFriction.extremumValue = rearLeftCollider.sidewaysFriction.extremumValue;
158       RLwheelFriction.asymptoteSlip = rearLeftCollider.sidewaysFriction.asymptoteSlip;
159       RLwheelFriction.asymptoteValue = rearLeftCollider.sidewaysFriction.asymptoteValue;
160       RLwheelFriction.stiffness = rearLeftCollider.sidewaysFriction.stiffness;
161     RRwheelFriction = new WheelFrictionCurve ();
162       RRwheelFriction.extremumSlip = rearRightCollider.sidewaysFriction.extremumSlip;
163       RRWextremumSlip = rearRightCollider.sidewaysFriction.extremumSlip;
164       RRwheelFriction.extremumValue = rearRightCollider.sidewaysFriction.extremumValue;
165       RRwheelFriction.asymptoteSlip = rearRightCollider.sidewaysFriction.asymptoteSlip;
166       RRwheelFriction.asymptoteValue = rearRightCollider.sidewaysFriction.asymptoteValue;
```


PrometeoCarController.cs
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs
RRwheelFriction.stiffness = rearRightCollider.sidewaysFriction.stiffness;
//car engine sound.
if(carEngineSound != null){
initialCarEngineSoundPitch = carEngineSound.pitch;
}
if(useUI){
InvokeRepeating("CarSpeedUI", 0f, 0.1f);
}else if(!useUI){
if(carSpeedText != null){
carSpeedText.text = "0";
}
}
if(useSounds){
InvokeRepeating("CarSounds", 0f, 0.1f);
}else if(!useSounds){
if(carEngineSound != null){
carEngineSound.Stop();
}
if(tireScreechSound != null){
tireScreechSound.Stop();
}
}
if(!useEffects){
if(RLWParticleSystem != null){
RLWParticleSystem.Stop();
}
if(RRWParticleSystem != null){
RRWParticleSystem.Stop();


PrometeoCarController.cs
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs
}
if(RLWTireSkid != null){
RLWTireSkid.emitting = false;
}
if(RRWTireSkid != null){
RRWTireSkid.emitting = false;
}
}
if(useTouchControls){
if(throttleButton != null && reverseButton != null &&
turnRightButton != null && turnLeftButton != null
&& handbrakeButton != null){
throttlePTI = throttleButton.GetComponent<PrometeoTouchInput>();
reversePTI = reverseButton.GetComponent<PrometeoTouchInput>();
turnLeftPTI = turnLeftButton.GetComponent<PrometeoTouchInput>();
turnRightPTI = turnRightButton.GetComponent<PrometeoTouchInput>();
handbrakePTI = handbrakeButton.GetComponent<PrometeoTouchInput>();
touchControlsSetup = true;
}else{
String ex = "Touch controls are not completely set up. You must drag and drop your scene buttons in the" +
" PrometeoCarController component.";
Debug.LogWarning(ex);
}
}
}
void Update()
{


PrometeoCarController.cs
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs
//CAR DATA
//speed of the car.
carSpeed = (2 * Mathf.PI * frontLeftCollider.radius * frontLeftCollider.rpm * 60) / 1000;
localVelocityX = transform.InverseTransformDirection(carRigidbody.velocity).x;
localVelocityZ = transform.InverseTransformDirection(carRigidbody.velocity).z;
//CAR PHYSICS
if (useTouchControls && touchControlsSetup){
if(throttlePTI.buttonPressed){
CancelInvoke("DecelerateCar");
deceleratingCar = false;
GoForward();
}
if(reversePTI.buttonPressed){
CancelInvoke("DecelerateCar");
deceleratingCar = false;
GoReverse();
}
if(turnLeftPTI.buttonPressed){
TurnLeft();
}
if(turnRightPTI.buttonPressed){
TurnRight();
}
if(handbrakePTI.buttonPressed){

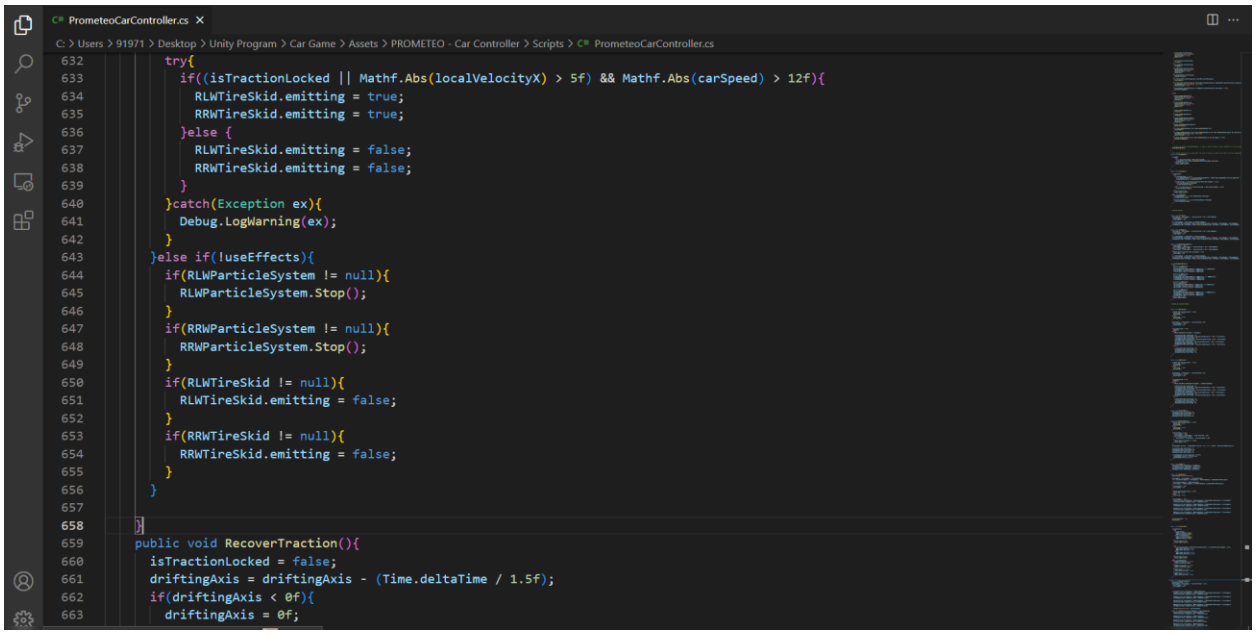

PrometeoCarController.cs
C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs
CancelInvoke("DecelerateCar");
deceleratingCar = false;
Handbrake();
}
if(!handbrakePTI.buttonPressed){
RecoverTraction();
}
if((!throttlePTI.buttonPressed && !reversePTI.buttonPressed)){
ThrottleOff();
}
if((!reversePTI.buttonPressed && !throttlePTI.buttonPressed) && !handbrakePTI.buttonPressed && !deceleratingCar){
InvokeRepeating("DecelerateCar", 0f, 0.1f);
deceleratingCar = true;
}
if(!turnLeftPTI.buttonPressed && !turnRightPTI.buttonPressed && steeringAxis != 0f){
ResetSteeringAngle();
}
}else{
if(Input.GetKey(KeyCode.W)){
CancelInvoke("DecelerateCar");
deceleratingCar = false;
GoForward();
}
if(Input.GetKey(KeyCode.S)){
CancelInvoke("DecelerateCar");
deceleratingCar = false;
GoReverse();
}
if(Input.GetKey(KeyCode.A)){
TurnLeft();



PrometeoCarController.cs

C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs

```
293             decelerateCar = false;
294             GoReverse();
295           }
296
297           if(Input.GetKey(KeyCode.A)){
298             TurnLeft();
299           }
300           if(Input.GetKey(KeyCode.D)){
301             TurnRight();
302           }
303           if(Input.GetKey(KeyCode.Space)){
304             CancelInvoke("DecelerateCar");
305             decelerateCar = false;
306             Handbrake();
307           }
308           if(Input.GetKeyUp(KeyCode.Space)){
309             RecoverTraction();
310           }
311           if((!Input.GetKey(KeyCode.S) && !Input.GetKey(KeyCode.W))){
312             ThrottleOff();
313           }
314           if((!Input.GetKey(KeyCode.S) && !Input.GetKey(KeyCode.W)) && !Input.GetKey(KeyCode.Space) && !decelerateCar){
315             InvokeRepeating("DecelerateCar", 0f, 0.1f);
316             decelerateCar = true;
317           }
318           if(!Input.GetKey(KeyCode.A) && !Input.GetKey(KeyCode.D) && steeringAxis != 0f){
319             ResetSteeringAngle();
320           }
321
322         }
323
324
325       // We call the method AnimateWheelMeshes() in order to match the wheel collider movements with the 3D meshes of the wheels.
```



PrometeoCarController.cs

C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs

```
632           try{
633             if((isTractionLocked || Mathf.Abs(localVelocityX) > 5f) && Mathf.Abs(carSpeed) > 12f){
634               RLWTireSkid.emitting = true;
635               RRWTireSkid.emitting = true;
636             }else {
637               RLWTireSkid.emitting = false;
638               RRWTireSkid.emitting = false;
639             }
640           }catch(Exception ex){
641             Debug.LogWarning(ex);
642           }
643         }else if(!useEffects){
644           if(RLWParticleSystem != null){
645             RLWParticleSystem.Stop();
646           }
647           if(RRWParticleSystem != null){
648             RRWParticleSystem.Stop();
649           }
650           if(RLWTireSkid != null){
651             RLWTireSkid.emitting = false;
652           }
653           if(RRWTireSkid != null){
654             RRWTireSkid.emitting = false;
655           }
656         }
657
658       }
659       public void RecoverTraction(){
660         isTractionLocked = false;
661         driftingAxis = driftingAxis - (Time.deltaTime / 1.5f);
662         if(driftingAxis < 0f){
663           driftingAxis = 0f;
```

PrometeoCarController.cs

C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoCarController.cs

```
667         if(FLwheelFriction.extremumSlip > FLWextremumSlip){
668           FLwheelFriction.extremumSlip = FLWextremumSlip * handbrakeDriftMultiplier * driftingAxis;
669           frontLeftCollider.sidewaysFriction = FLwheelFriction;
670
671           FRwheelFriction.extremumSlip = FRWextremumSlip * handbrakeDriftMultiplier * driftingAxis;
672           frontRightCollider.sidewaysFriction = FRwheelFriction;
673
674           RLwheelFriction.extremumSlip = RLWextremumSlip * handbrakeDriftMultiplier * driftingAxis;
675           rearLeftCollider.sidewaysFriction = RLwheelFriction;
676
677           RRwheelFriction.extremumSlip = RRWextremumSlip * handbrakeDriftMultiplier * driftingAxis;
678           rearRightCollider.sidewaysFriction = RRwheelFriction;
679
680           Invoke("RecoverTraction", Time.deltaTime);
681
682         }else if (FLwheelFriction.extremumSlip < FLWextremumSlip){
683           FLwheelFriction.extremumSlip = FLWextremumSlip;
684           frontLeftCollider.sidewaysFriction = FLwheelFriction;
685
686           FRwheelFriction.extremumSlip = FRWextremumSlip;
687           frontRightCollider.sidewaysFriction = FRwheelFriction;
688
689           RLwheelFriction.extremumSlip = RLWextremumSlip;
690           rearLeftCollider.sidewaysFriction = RLwheelFriction;
691
692           RRwheelFriction.extremumSlip = RRWextremumSlip;
693           rearRightCollider.sidewaysFriction = RRwheelFriction;
694
695           driftingAxis = 0f;
696         }
697       }
698
699   }
```

PrometeoTouchInput.cs

C: > Users > 91971 > Desktop > Unity Program > Car Game > Assets > PROMETEO - Car Controller > Scripts > PrometeoTouchInput.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class PrometeoTouchInput : MonoBehaviour
6   {
7
8       public bool changeScaleOnPressed = false;
9       [HideInInspector]
10      public bool buttonPressed = false;
11      RectTransform rectTransform;
12      Vector3 initialScale;
13      float scaleDownMultiplier = 0.85f;
14
15      void Start(){
16        rectTransform = GetComponent<RectTransform>();
17        initialScale = rectTransform.localScale;
18      }
19
20      public void ButtonDown(){
21        buttonPressed = true;
22        if(changeScaleOnPressed){
23          rectTransform.localScale = initialScale * scaleDownMultiplier;
24        }
25      }
26
27      public void ButtonUp(){
28        buttonPressed = false;
29        if(changeScaleOnPressed){
30          rectTransform.localScale = initialScale;
31        }
32      }
33
34  }
35
```

TESTING AND MAINTAINING

way of performing typical software maintenance, with patches distributed via networks. Unlike computer software, video games could not release patches and new versions of the game for most of the industry's lifespan, making the process of software testing and maintenance in the gaming industry an interesting and relatively unexplored topic. Game developers had to use what little maintenance methods they had at their disposal, such as sequels that contained many corrections and gameplay additions that should have been included in the original game. Testing has occurred since video games were first made, but true maintenance in the gaming industry is still a new, constantly evolving process. To understand testing and maintenance in the video game industry completely, one must examine current testing and maintenance procedures, how these testing and maintenance practices have evolved since the gaming industry began, and how their evolution has negatively impacted video game design today.

Testing in General

Testing is a necessary step in any software development cycle. To understand how testing is done in the video game industry, one must examine what software testing is and how testing is done in general, then look into how testing video games is best accomplished. Testing can be defined as "evaluating software by observing its execution". Large software projects contain highly complex source code that can often give unpredictable results when the code is finally executed. Testing helps developers see if their code works properly and correct any errors that may occur before the product is put into implementation. Testing, however, does not simply involve running an executable file and performing a few random tests to determine the output. A developer must carefully plan out the testing phase to ensure that the maximum number of glitches and potential risks are accounted for and corrected with the tests.

Functional Testing

Two main methods exist to determine the test cases for a particular project: functional testing and structural testing. With functional testing, the tester cannot see how a particular piece of software is implemented. He or she only knows which inputs map to which outputs. Functional testing is also known as black box testing, which comes from the fact that the tester cannot see the specific implementation of the software. This method of testing is based on the idea that any piece of software can be considered a function that maps input values in the function's domain to output values in the function's range. When a designer is using functional testing to create test cases, the task
becomes human centered. The designer must examine the requirements of a program, what tasks the software should perform and when the software should perform these tasks, and develop a system of tests that encompass all the requirements of the software. Pezze and Young stated that the "core of functional test case design is partitioning the possible behaviors of the program into a finite number of homogeneous classes," where each of these classes can ultimately be determined to be correct or incorrect.

Structural Testing

Structural testing is often referred to as white box testing. In structural testing, test cases are developed based upon how the code in a program is written. Unlike functional testing, where the implementation of the program is like a black box, structural testing makes the tester aware of how the code is written and the test is thus more like a white box because implementation is known in addition to appropriate mappings of inputs to outputs. One of the biggest strengths structural testing has over functional testing lies in the area of test coverage metrics, which is the ability to measure how much of a specific program is tested. Using structural testing, a designer can look at the source code of a program and write a test case that tests a specific function of the program or maybe a small module of code. For example, a two-dimensional platforming game, something

similar to Super Mario or Sonic, may contain a function that causes the player controlled character to jump. This function may perform multiple tasks, such as controlling how high the character jumps, making sure the character does not fall through the ground, checking if the character made any collisions with other sprites on the screen, or a myriad of other tasks. Using functional testing, multiple tests would be required to test the jumping function. One test may check if the player can jump when a button ispressed, another may be used to see if collisions between the player and enemies are detected while the player is jumping, and a third test may ensure that the player does not fall through various areas of the ground. These tests would not be able to be combined since the tester is unaware of how jumping is implemented in the game. If a jump function caused the player to jump and collision detection was handled outside the jump function, testing for collision outside of the jumping test would be necessary. In short, the nature of the jump function determines how many test cases are necessary.

Maintenance In General

After the testing phase is completed, the software product is released and the longest phase in the software development cycle begins: the maintenance phase. As with testing, in order to understand how maintenance is used in the video game industry, one must first examine how maintenance is used in the computer software industry in general. Maintenance in the software development industry can be defined as “all the actions that
are needed to keep software in such a running order that it achieves all its objectives from the beginning until the end of the usage”. The maintenance phase is an important part of the software development cycle. On average, two thirds of a product’s total cost is spent on maintenance.

H/W And S/W Requirement

Hardware used: -

- **Intel CORE i5 9th Gen** is used as a processor because it is fast, reliable and stable and we can run our pc for longtime. By using this processor, we can keep on developing our project without any worries.
- **Ram 8 GB** is used as it will provide fast reading and writing capabilities and will in turn support in processing**.**
- Operating system- **Windows 10** is used as the operating system as it is stable and supports more features and is more user friendly.

Hardware requirement: -

- 2 GB RAM

Software used: -

- **Unity** software is used to developed the project because its UI is clean and is vary stable. A lot of indie game developer uses Unity so there are a lot of content out in the internet about Unity. Unity can be used to make all shorts of games unlike Unreal which heavily focuses on very realistic AAA game.
- **C#** is used as C# is the only language that Unity supports natively. And it is easy to use. It supports OOPS concept.

Software requirement: -

- Android 7.0+ ‘Nougat’ (API 24)

Future scope

- Implementing Multiplayer.
- Improving UI
- Adding new stages and cars
- Making it more fun to play
- Improving control
- Implementing FPV
- Adding gyroscope sensor and Joystick

Limitations of the project

- Not a Multiplayer Game.
- Very few maps and Cars
- UI is not attractive.
- Control needs to be improved
- No 1st person perspective

Reference

- **Unity: -** https://assetstore.unity.com/
  https://unity.com/how-to/beginner-video-game-resources
- **YouTube: -** https://www.youtube.com/@Brackeys
  https://www.youtube.com/watch?v=RgomLumqwCk&t=1s
  https://www.youtube.com/watch?v=fbcJjZInt-A
  and many more.