RAMAPO COLLEGE OF NEW JERSEY

2D Car Racing Game

CMPS 450: Senior Project

Andrew Ferrand

Fall 2023

Table of Contents

Content	Page Number
Introduction	3
Installation Instructions	4 - 8
User Manual	9 - 13
Design	14 - 22
Program	23 - 196
Testing	197 - 198
Conclusion	199 - 200
References	201

Introduction

For my Senior Project, I developed a 2D racing game in Unity using C# scripts. In this game, the player is able to choose from a selection of cars to drive with and tracks to race on. Races are seen from a fixed top-down camera angle that provides a complete view of the race track. These race tracks contain multiple obstacles and boxes with items, which help make the gameplay a bit more interesting.

There are 8 cars in these races with 7 of them being AI opponents that will react to other cars and obstacles on the road. Additionally, the game incorporates a customizable difficulty setting, allowing players to tailor the challenge level of the AI opponents to their preference. The game also has a scoring system where the player will be awarded points based on multiple factors, including the amount of time it takes for them to complete a race, the position they finished in, and the top speed they reached. At the end of the race, the player will be shown the amount of points they have earned based on these factors, which will be added to a total amount. If the player has enough points, then they can be used to purchase unlockables, such as cars and race tracks.

One immediate challenge for this project was learning how to use Unity's game engine as I didn't have much experience using it before, which required me to view tutorials about the basics of Unity and how to use its game objects and UI elements so that I could properly implement the graphics and sound effects to my game. Since this is a racing game, I needed to figure out how to simulate the movement and physics of the cars as they moved around the race tracks and collided with other objects, which was accomplished thanks to Unity's built-in physics engine. I also needed to figure out how to set up the AI in a way so that they properly drove around the race track, avoided obstacles, and so on. This took a good amount of time and research, where I ultimately went with a waypoint node system where the cars would drive to each node in a certain order. The other major challenge was learning C# so that I could create the scripts for this game, which also involved viewing tutorials in order to better understand the programming language. Concurrently, the .NET Environment course has also been greatly beneficial by providing me more opportunities to practice with C# code.

Installation Instructions

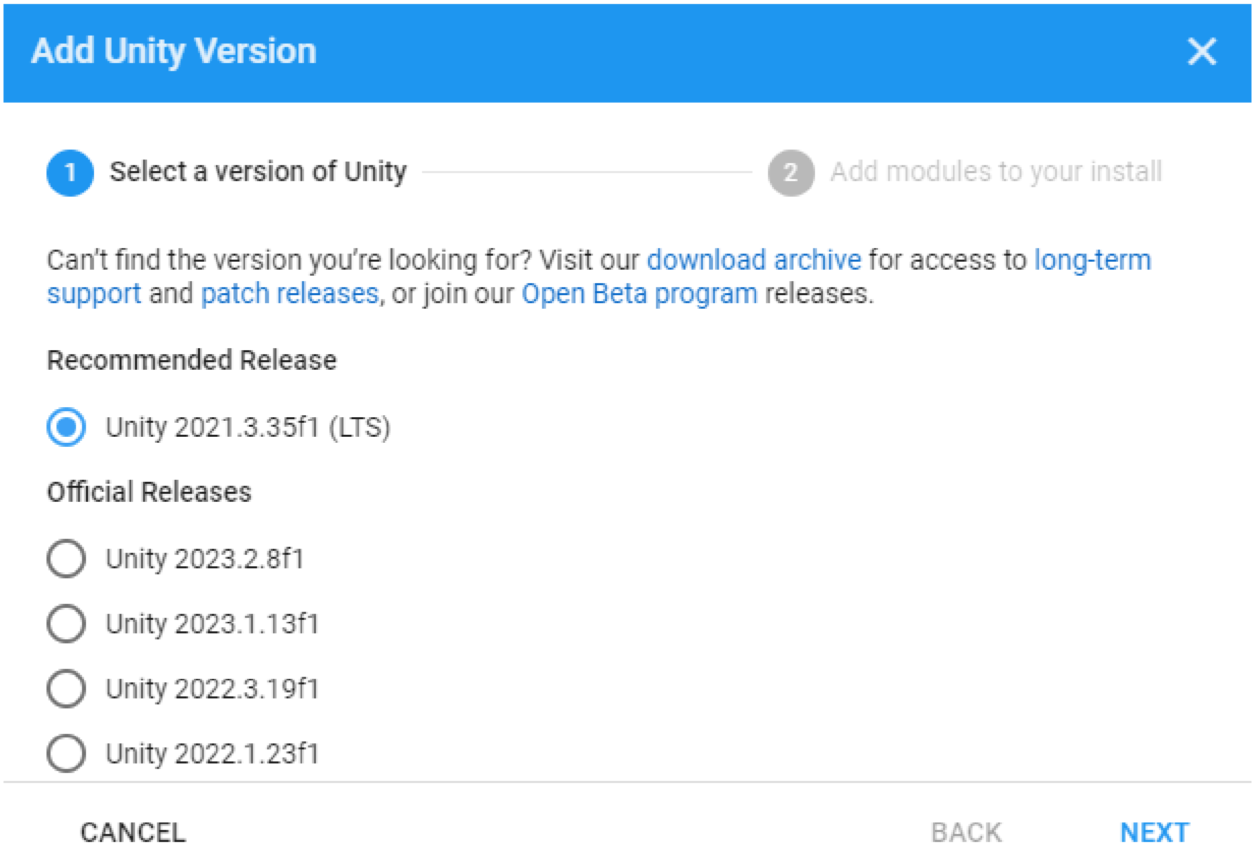
Specifications and Tools Needed:

- Windows Operating System (Windows 10 or higher recommended)
- Unity
- Microsoft Visual Studio 2022 or newer

Setting Up Unity:

Download the Unity Hub on the official website here: https://unity.com/download

Open UnityHubSetup.exe and follow the onscreen instructions. Once it has finished installing, open the Unity Hub application and click on the "Add" or "Install Editor" button in the Installs section. Make sure the recommended release version is selected and click the "Next" button.



Add Unity Version

1 Select a version of Unity — 2 Add modules to your install

Can't find the version you're looking for? Visit our download archive for access to long-term support and patch releases, or join our Open Beta program releases.

Recommended Release

● Unity 2021.3.35f1 (LTS)

Official Releases

○ Unity 2023.2.8f1

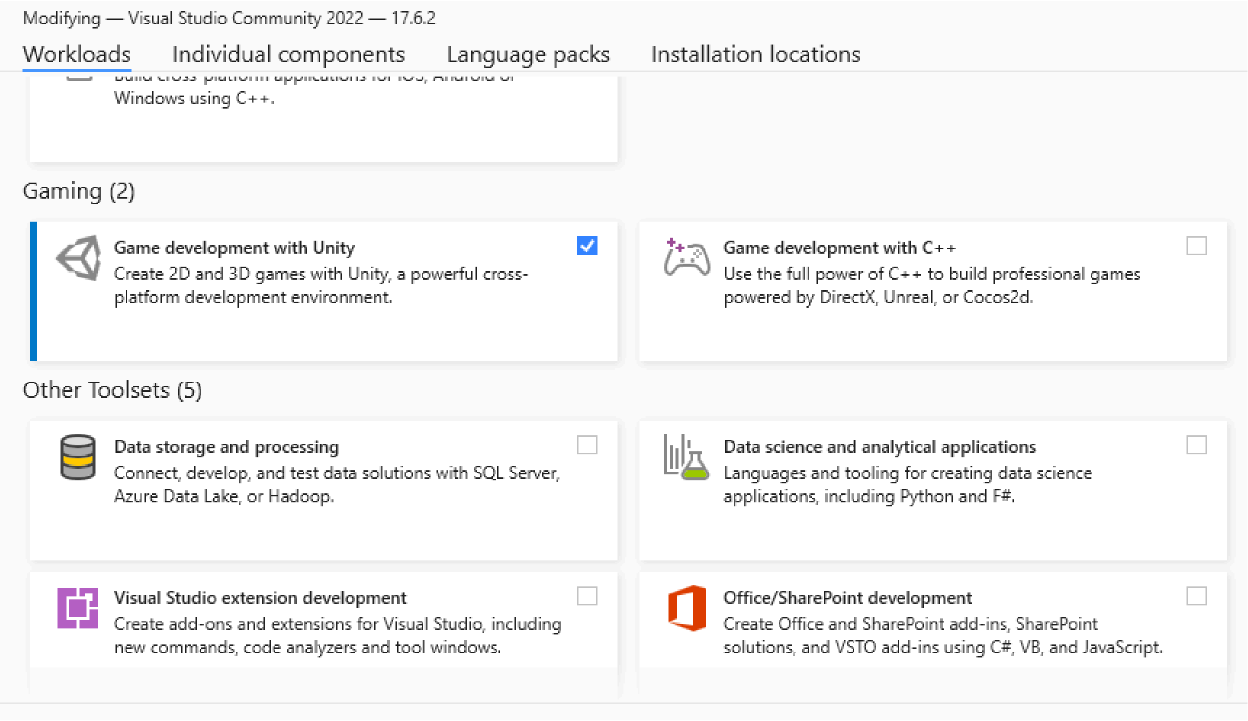
○ Unity 2023.1.13f1

○ Unity 2022.3.19f1

○ Unity 2022.1.23f1

On the next screen, check the box next to the “Microsoft Visual Studio Community 2022” module and click “Done”. After a couple minutes, a pop-up window will appear for the Visual Studio Installer. A full guide for the installer can be found here: https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022

On the Workloads page, scroll down to the Gaming section, check the box for “Game development with Unity”, and click the Install button.



Modifying — Visual Studio Community 2022 — 17.6.2

Workloads | Individual components | Language packs | Installation locations

Build cross-platform applications for iOS, Android or Windows using C++.

Gaming (2)

☑ Game development with Unity
Create 2D and 3D games with Unity, a powerful cross-platform development environment.

☐ Game development with C++
Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Other Toolsets (5)

☐ Data storage and processing
Connect, develop, and test data solutions with SQL Server, Azure Data Lake, or Hadoop.

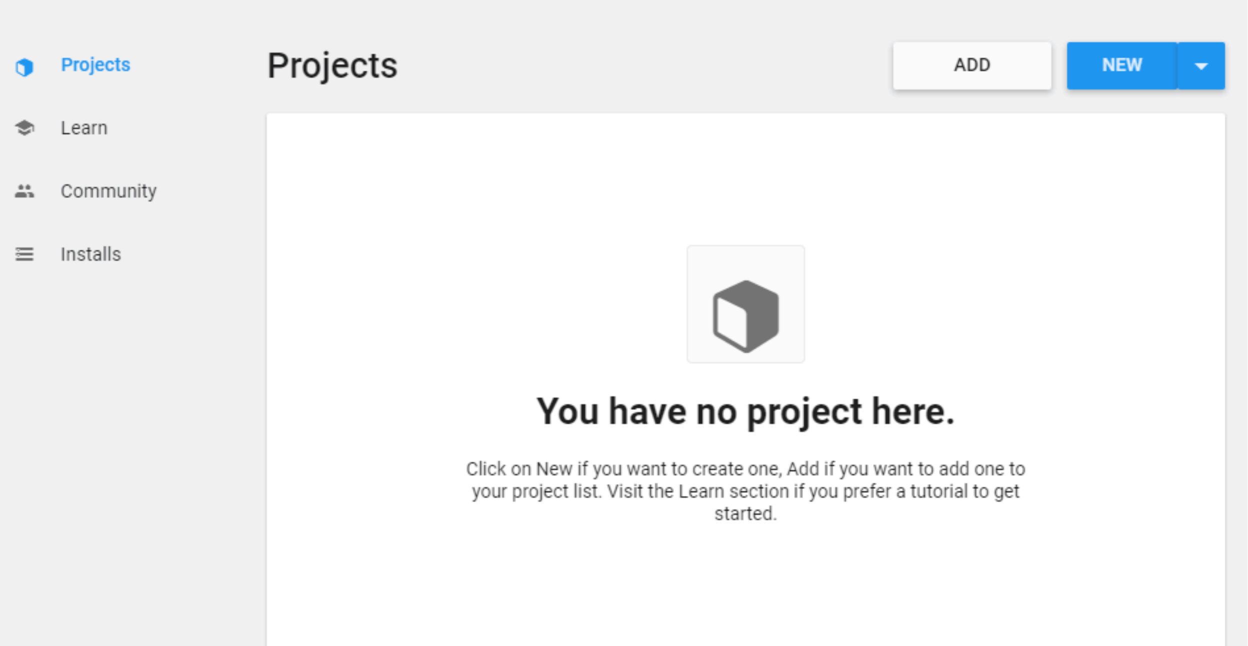
☐ Data science and analytical applications
Languages and tooling for creating data science applications, including Python and F#.

☐ Visual Studio extension development
Create add-ons and extensions for Visual Studio, including new commands, code analyzers and tool windows.

☐ Office/SharePoint development
Create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins using C#, VB, and JavaScript.

Building the Application:

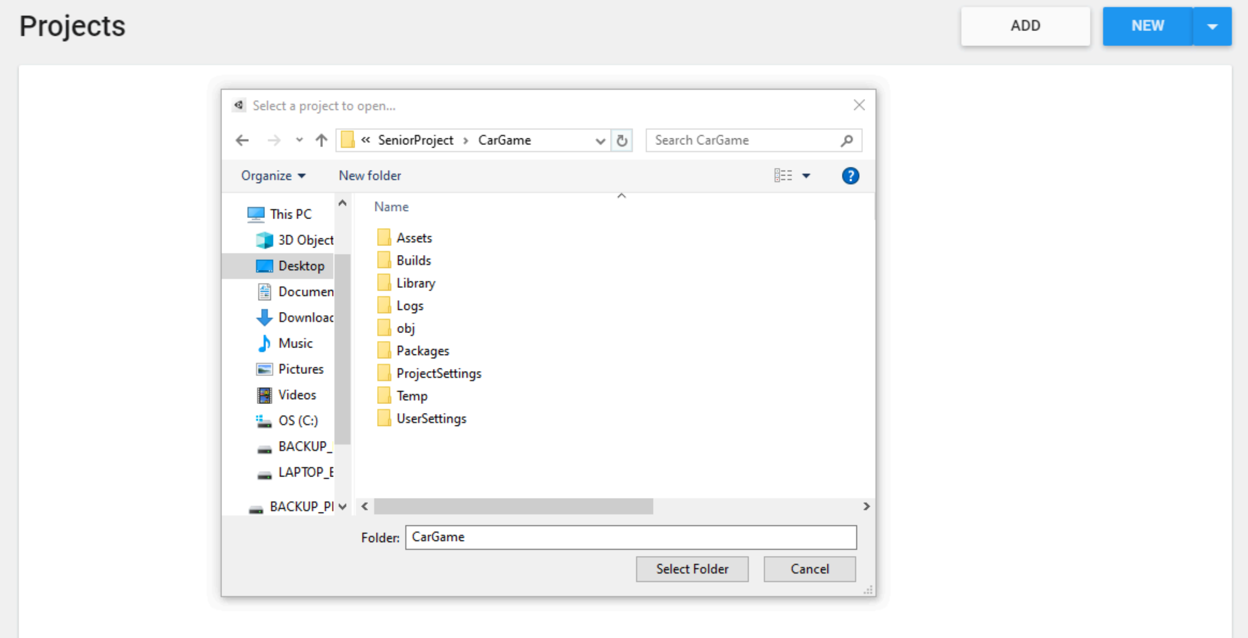
1. Open the Unity Hub and select the “Add” button to open File Explorer



You have no project here.

Click on New if you want to create one, Add if you want to add one to your project list. Visit the Learn section if you prefer a tutorial to get started.

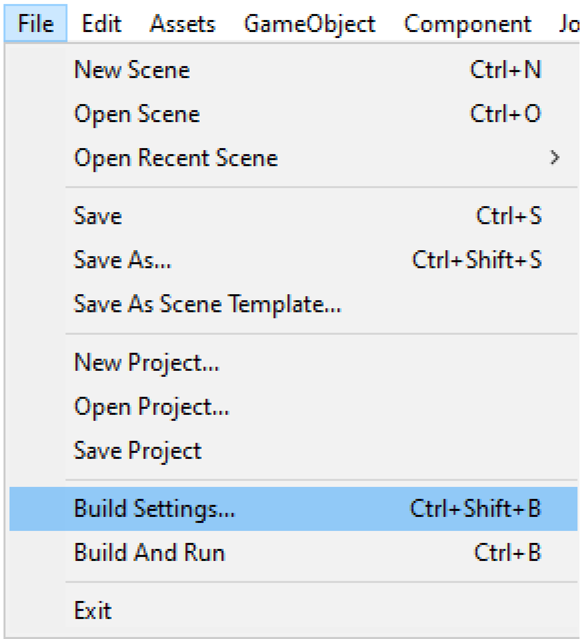
2. In File Explorer, click the “CarGame” folder and hit the “Select Folder” button



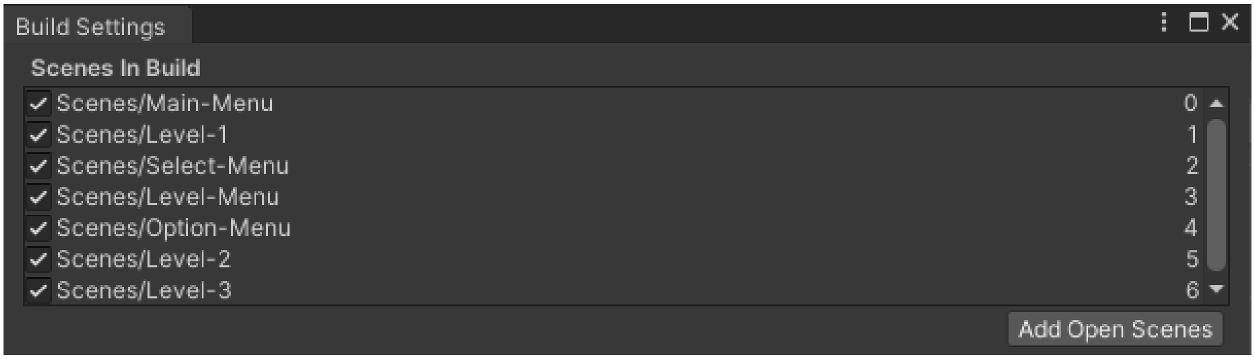
Folder: CarGame

3. After the project has been added to the hub, click it to open the project in Unity Editor. (You may need to install the 2021.3.30f1 version of Unity Editor if a message pops up saying, “Editor version not installed”)

4. Once the project has loaded in the editor, go to “File” > “Build Settings…”, which will open a new window



5. In the “Scenes In Build” section, make sure all of the boxes next to the scenes are checked on

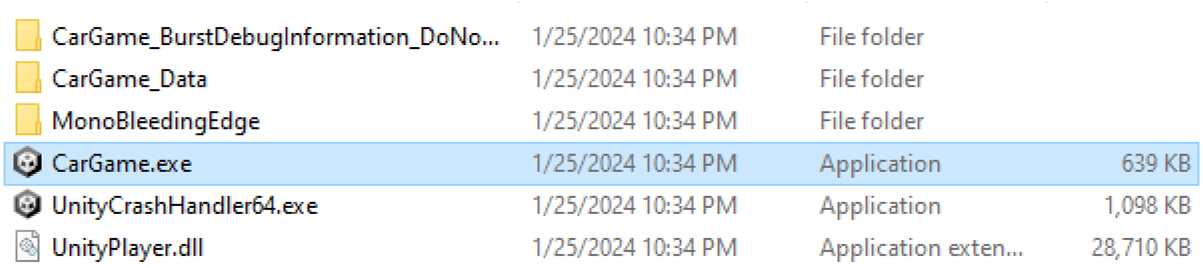

Build Settings

Scenes In Build

☑	Scenes/Main-Menu	0
☑	Scenes/Level-1	1
☑	Scenes/Select-Menu	2
☑	Scenes/Level-Menu	3
☑	Scenes/Option-Menu	4
☑	Scenes/Level-2	5
☑	Scenes/Level-3	6

6. Click “Build” at the bottom of the window to open File Explorer and select the “Builds” folder located in CarGame (It may take a few minutes for the game to build)

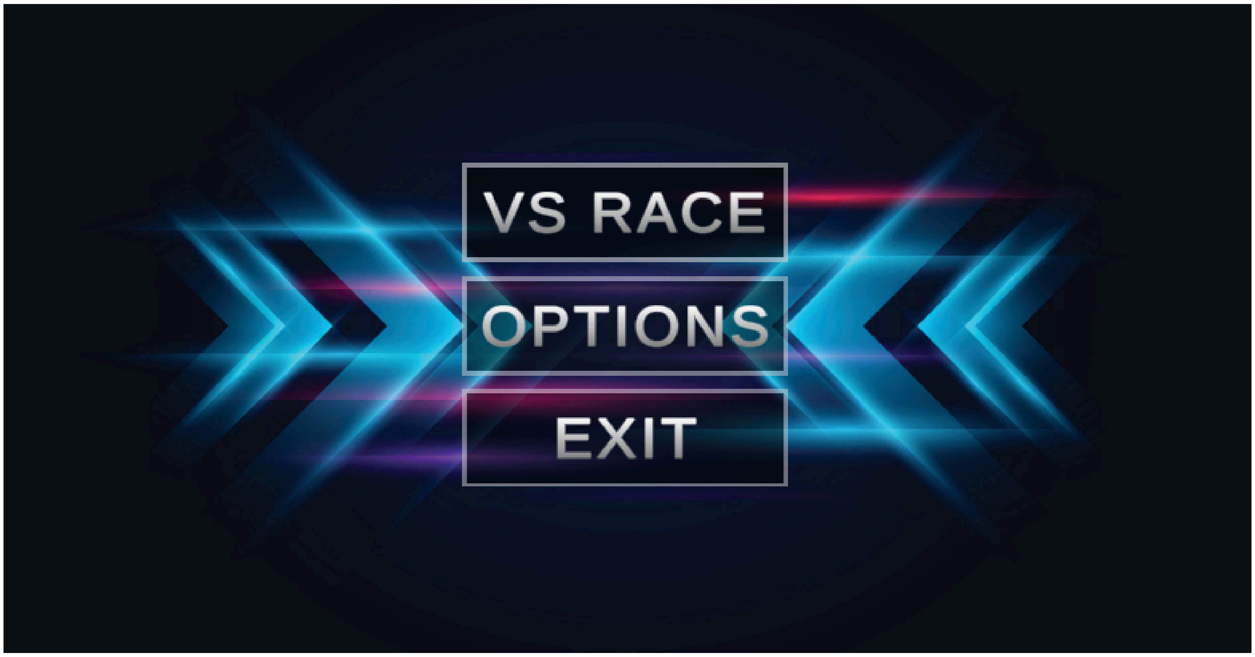
7. After the game has finished building, go into the "Builds" folder and click "CarGame.exe" to start the game



CarGame_BurstDebugInformation_DoNo...	1/25/2024 10:34 PM	File folder	
CarGame_Data	1/25/2024 10:34 PM	File folder	
MonoBleedingEdge	1/25/2024 10:34 PM	File folder	
CarGame.exe	1/25/2024 10:34 PM	Application	639 KB
UnityCrashHandler64.exe	1/25/2024 10:34 PM	Application	1,098 KB
UnityPlayer.dll	1/25/2024 10:34 PM	Application exten...	28,710 KB

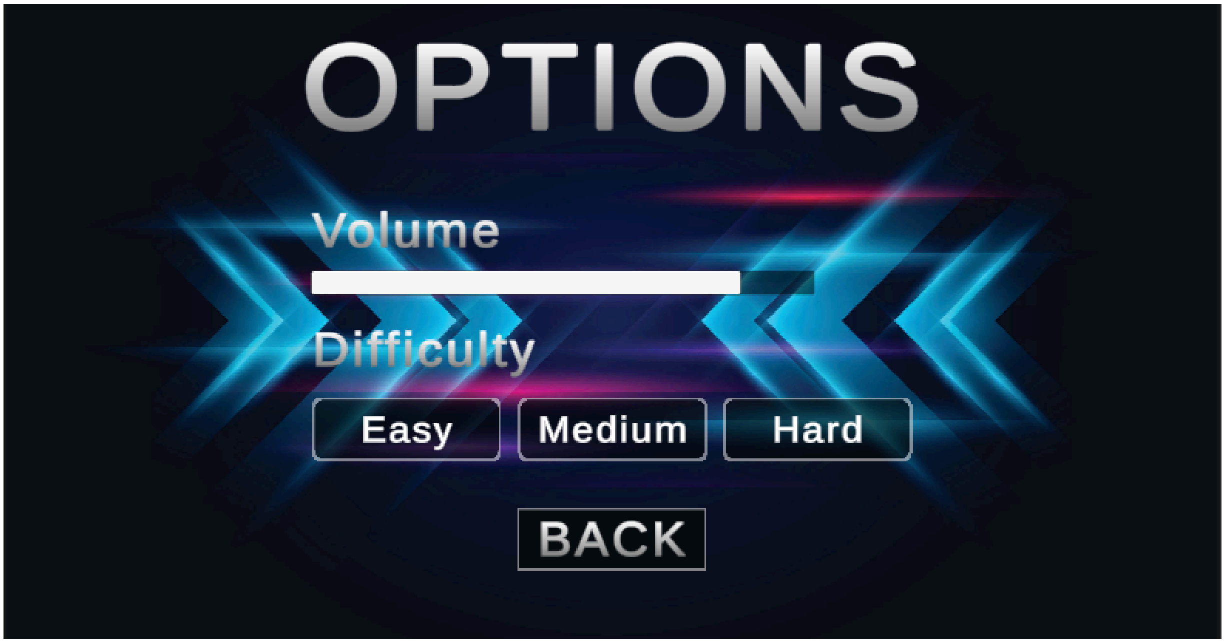
User Manual

Main Menu:



The first menu loaded for the game, which allows the player to either start a race, go to the options menu, or exit the game. Navigation through menus is achieved using the mouse cursor, with selections made by left-clicking on the desired menu options.

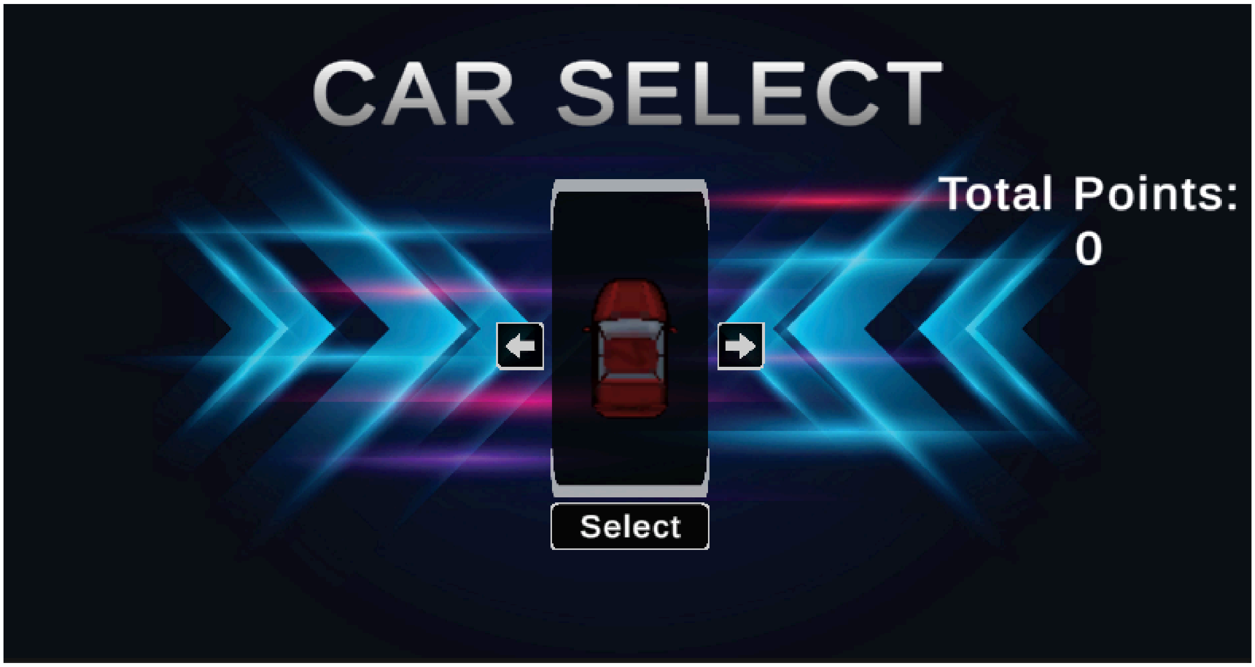
Options:



Volume: Bar that controls the amount of volume for the sound effects in the game.

Difficulty: Allows the player to select the difficulty level of the AI opponents in a race.

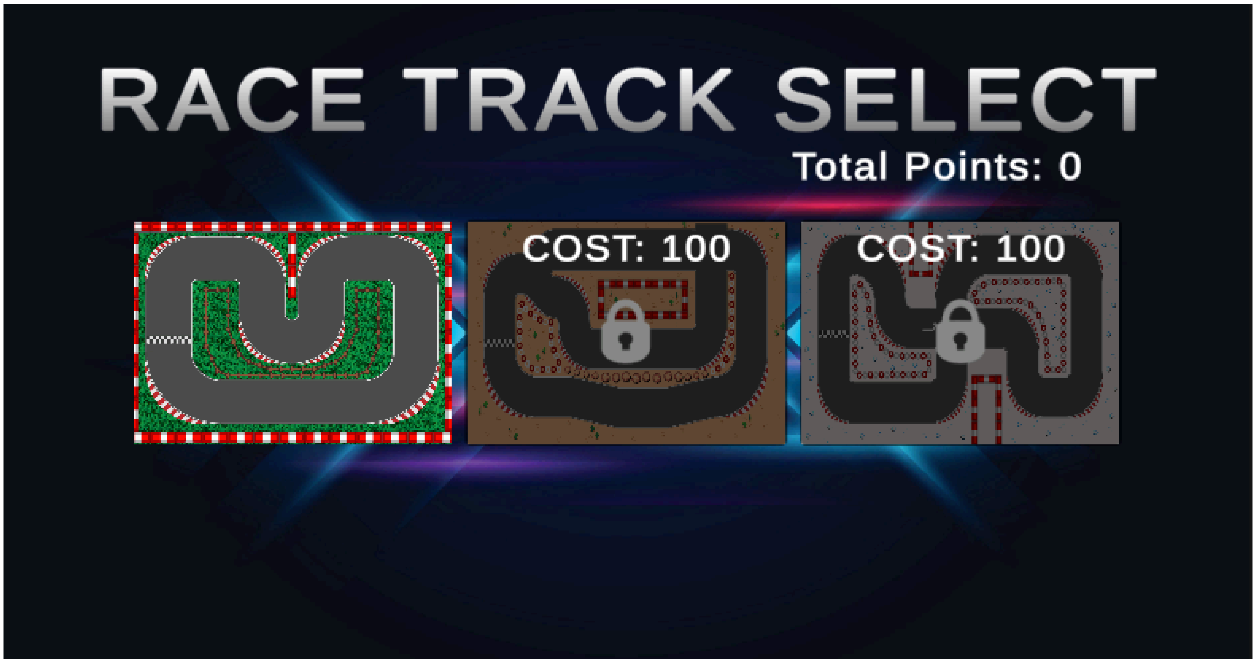
Car Select:



This menu allows the player to choose the car they want to use in a race. The player can click on the arrow buttons to view each car and confirm their choice by clicking on the select button. (The player can also use the left and right arrow keys to move between cars and press the spacebar to select them)

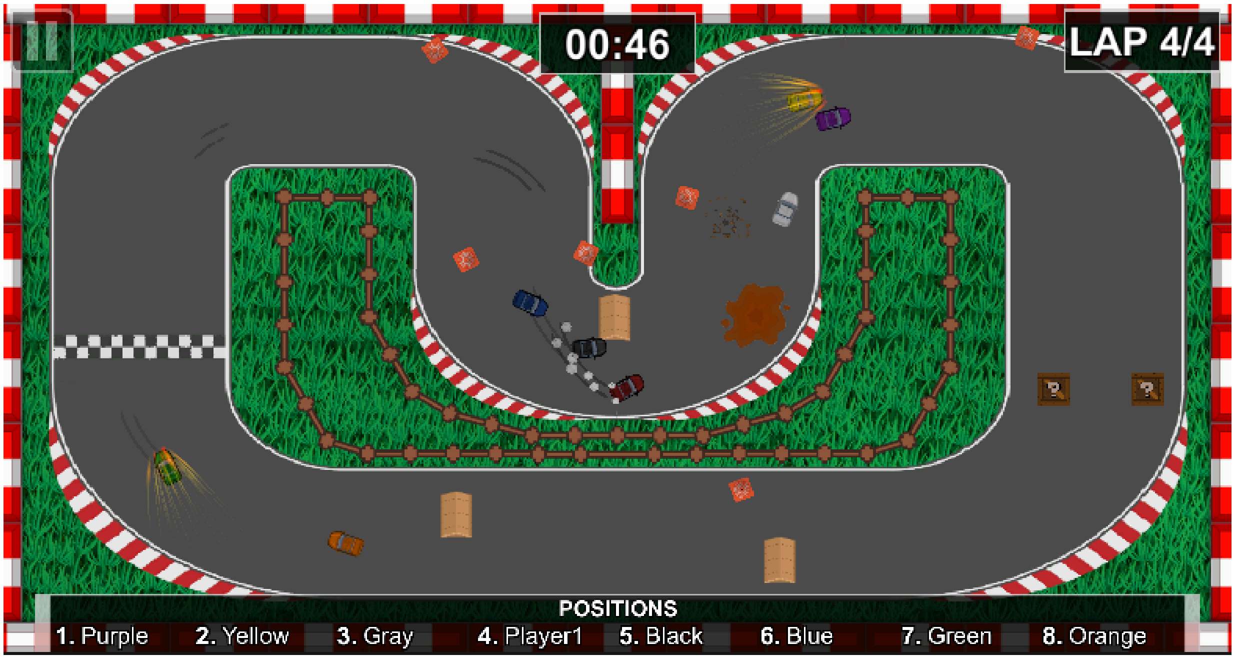
Some of the cars need to be purchased with a certain amount of points. The player can look at their total amount points to see if they have enough to buy another car.

Race Track Select:



This menu allows the player to choose the track they want to race on. They can select the track to race on by clicking on their respective portraits. Only the first track is available to play on at the start as the other tracks need to be purchased. Once the player has enough points, they can unlock other tracks by clicking on their portraits.

In-Game:



Before the start of the race, there will be a short countdown sequence to allow the player to get prepared. Once it has finished, the player will be able to move their car around the track. The player can control their car using either the arrow keys or the WASD keys. The 7 other cars on the track are controlled by AI that will race against each other and the player.

Race tracks contain off-road sections that slow down cars when driven on. The tracks also contain multiple obstacles and item boxes that affect the cars in different ways. When a car touches an item box, it can get one of two items that can be activated by pressing the spacebar:

- Speed Boost: A temporary power up that increases the top speed of the car.
- Mud Puddle: The car drops a mud puddle behind it that slows down any car that comes in contact with it. The puddle disappears after a few seconds.

Timer: Shows the current amount of time.

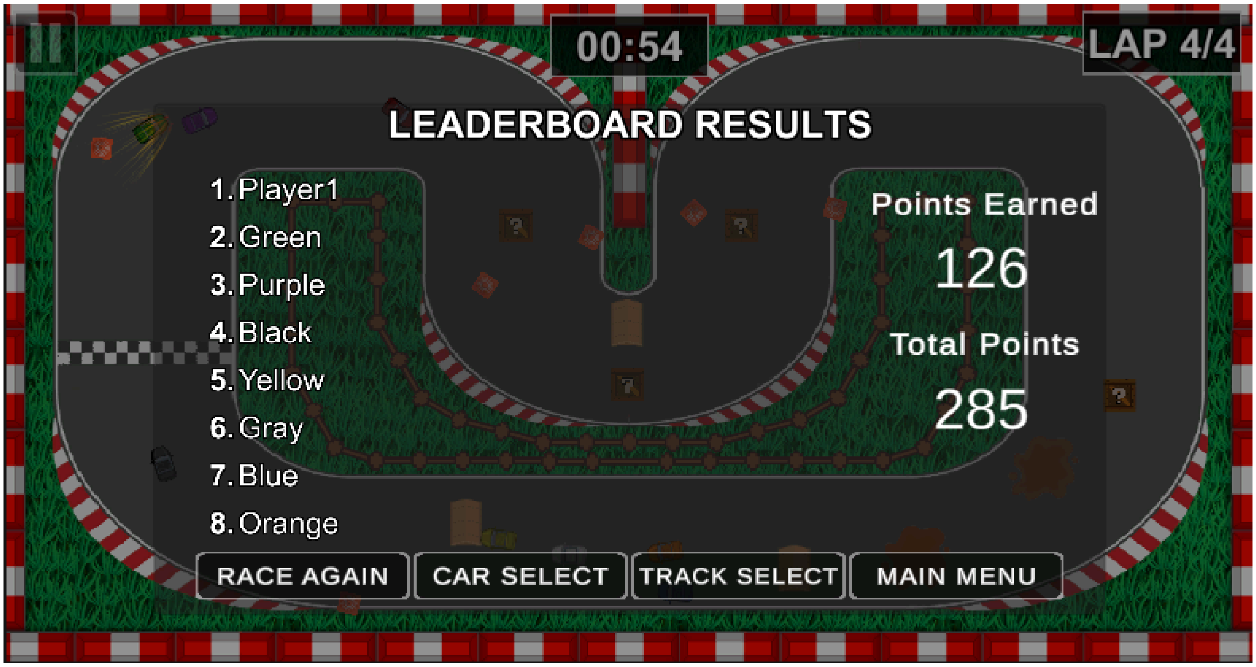
Lap Counter: Shows the amount of laps the player has completed.

Pause Button: Allows the player to pause the game and gives them the option to exit to another menu

Position Counter: Shows the current positions of the cars.

The race will continue until the player has completed all 4 laps of a race, which will bring up the leaderboard results screen.

Leaderboard Results:



LEADERBOARD RESULTS

1. Player1
2. Green
3. Purple
4. Black
5. Yellow
6. Gray
7. Blue
8. Orange

Points Earned

126

Total Points

285

This screen shows the final positions of the cars at the time the player finishes the race. Additionally, it provides information about the player's points. The amount of points the player earns at the end of the race is determined based on multiple factors including the position they finished in, the top speed they reached, and the amount of time to complete the race.

These points are then added to the total amount, which can be used to purchase unlockables.

The player can also select one of the buttons at the bottom to either play the race again or exit to a different menu.

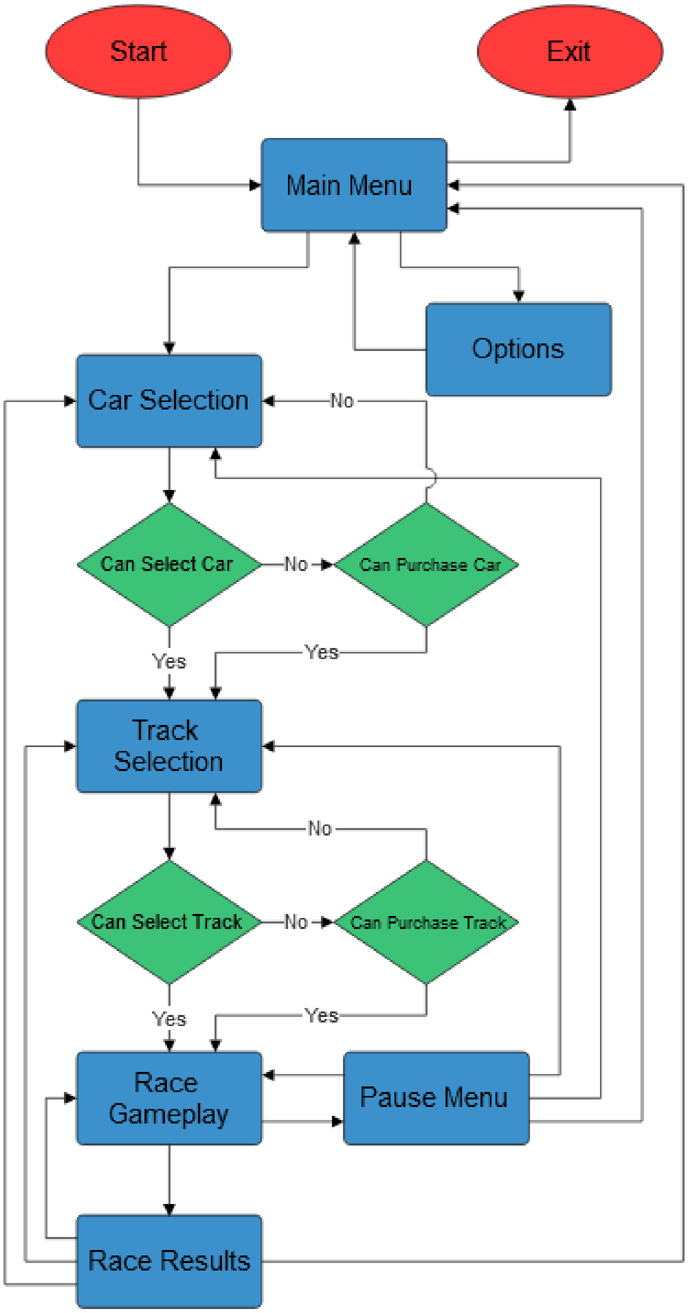
Design

Unity Game Engine serves as the backbone of the game by providing tools for physics simulation, graphics rendering, and UI management. C# Scripts drive the game's logic, including player input processing, car movement, AI behavior, collision handling, scoring, etc. When designing this game, the initial phase focused on establishing the fundamental game mechanics. This involved setting up the Unity environment and developing the physics of car movement, including handling collisions and interactions with race tracks and obstacles.

After establishing the basic mechanics, the next step involved programming the AI for the non-player cars. I opted for a waypoint node system to guide the AI-driven cars around the tracks. I also implemented the A* pathfinding algorithm to further improve the AI and allow the cars to reverse when stuck. The focus then shifted to developing the user interface. This involved creating menus for car and track selection, implementing a points system, and designing a results screen to display post-race information.

The final phase of development was dedicated to adding additional features and refining the game. This included implementing a variety of items with specific behaviors and integrating a system for unlocking new cars and tracks. I also continually tested and refined the game to fix any small bugs and issues I could find.

Flowchart:


Start
Exit
Main Menu
Options
Car Selection
No
Can Select Car
No
Can Purchase Car
Yes
Yes
Track Selection
No
Can Select Track
No
Can Purchase Track
Yes
Yes
Race Gameplay
Pause Menu
Race Results

Data Structures:

Arrays and Lists: Used extensively to manage collections of objects and data elements in a flexible and efficient manner. Examples include:

- Car Data: Arrays of CarData objects were used to store information about different cars available in the game, such as their unique IDs, sprites, prefabs, and costs.
- Lap Counters: A list of LapCounter objects tracked the progress of each car in a race, including laps completed and checkpoints passed.
- Waypoints: Arrays in Waypoint class manage the sequence of nodes that AI cars follow around the track.

GameObjects and Transform: These were used to represent and manipulate objects in the game world.

Each car, item, and track element in the game was represented as a GameObject with a Transform component, allowing for manipulation of their position, rotation, and scale in the game scene.

Dictionary: Provided a way to efficiently map keys to values and useful for quickly accessing data based on a unique key.

- Used in SelectCarMenu for mapping car prefab names to more user-friendly names, aiding in the car selection and UI display processes.

ScriptableObjects: Allowed for the creation of reusable data assets in Unity.

- CarData scriptable objects were used to store data about each car, making it easy to modify and extend car attributes without changing the core game logic.

Animation Curves: Used to create smooth and customizable transitions and movements, especially useful for UI animations and car motion.

- In CarController, Animation Curves controlled the jump dynamics of cars, allowing for fine-tuning of the motion during in-game actions.

Particle Systems: Used for creating dynamic visual effects like smoke, dust, and boosts.

- Particle systems were linked to car behaviors in order to visually indicate actions, such as drifting and item effects.

UI Components: Essential for creating interactive and informative user interfaces.

- Text components displayed race information and scores.
- Image components showed car sprites and track visuals.
- Buttons facilitated user interactions in menus.

Class Descriptions:

AStarNode: Represents a node used in the A* pathfinding algorithm in a grid-based environment. Each node contains information about its position in the grid, its neighbors, and various costs associated with the pathfinding process. The costs include the distance from the start, the estimated distance to the goal, and the total cost. The class also provides methods for calculating these costs based on the AI's current position and destination, as well as a method to reset the node's properties for new pathfinding calculations. Nodes can be marked as obstacles, affecting their traversal in the pathfinding algorithm.

AStarPath: Responsible for implementing the A* pathfinding algorithm in a grid-based environment. It manages the creation and initialization of the grid of AStarNodes, calculates paths for AI navigation, and provides methods for converting between world and grid coordinates. Key features include grid initialization, obstacle detection, and pathfinding.

CarAIHandler: Responsible for controlling the AI behavior of cars in the game environment. It encompasses a range of functionalities including following waypoints or players, obstacle avoidance, dynamic speed adjustment based on skill level, and decision-making during different AI modes. The class integrates various aspects such as raycasting for obstacle detection, waypoint navigation, skill-based speed control, stuck detection, and dynamic pathfinding using

AStar algorithm. It also handles AI responses to environmental factors, like avoiding other cars and using items randomly.

CarController: Manages the physics, control, and behavior of a car in the game environment. It handles fundamental car dynamics such as acceleration, steering, drifting, and jumping, as well as interactions with different surfaces and the use of power-ups and items. This class controls the car's response to player inputs and AI commands,adjusting its behavior based on factors like surface type, obstacles, and current speed. Special features like jumps and item usage are also managed within this class.

CarData: A ScriptableObject class used to store data about cars in the game. It includes information such as a unique identifier for each car, a sprite for the UI representation, a prefab for the car object, and the cost of the car.

CarInputHandler: Responsible for managing player inputs and translating them into actions for the car. It captures and processes inputs such as steering, acceleration, braking, and the use of items. This class serves as the intermediary between the player's input devices and the car's behavior, ensuring that player commands are accurately reflected in the game.

CarJumpData: A simple data container that holds parameters for car jumping mechanics in the game. It defines the scale of jump height and push, which can be adjusted to customize the jumping behavior of cars.

CarParticleHandler: Manages the particle effects associated with a car's movement and actions in the game. It dynamically adjusts the rate and intensity of particle emissions based on the car's current state, such as drifting, braking, or driving on different surfaces.

CarSFXHandler: Manages the sound effects associated with a car's actions and interactions. It controls audio cues for different states and behaviors of the car, such as engine sounds, tire screeching during drifts, collision impacts, and jump landings. The class adjusts sound properties like volume and pitch based on the car's dynamics.

CarSpawn: Responsible for initializing and spawning player cars at the start of a race. It dynamically places cars at designated spawn points according to the players' choices and game settings. The class handles the instantiation of car prefabs, aligning them with players' preferences and assigning control mechanisms based on whether the player is an AI or a human.

CarUIHandler: Responsible for managing the user interface elements related to displaying car information in the game. It controls UI elements like car images and price tags, and manages animations for car selection scenarios.

Checkpoint: Represents a checkpoint or a finish line in the game environment. It holds essential information about each checkpoint, such as whether it is a finish line and its sequential number in the race course.

CountdownUIHandler: Responsible for managing the visual countdown sequence. It controls a UI Text element to display a countdown sequence, indicating the start of the race.

GameManager: Serves an important role in managing the overall game state and player data across different levels and scenes in the game. It operates as a singleton, ensuring only one instance exists throughout the game's lifecycle. This class handles various aspects of gameplay, including tracking the game state, managing race timings, player scores, and AI difficulty levels, as well as keeping records of purchased cars and unlocked tracks. GameManager also provides interfaces for other game components to access and modify game-related data, such as player information, points, and game states.

ItemBox: Represents item boxes in the game, which players can interact with to receive race items like speed boosts or obstacles. This class manages the item box's behavior, including its interactions with cars, the random selection of items, triggering animations, and handling the item box's destruction and respawn. When a car collides with an item box, the box grants a random item to the car and then initiates a destruction animation. The class also coordinates with the ItemBoxSpawn to manage the respawn of item boxes after they are destroyed.

ItemBoxSpawn: Handles the spawning and respawning of item boxes in the game. It controls the placement and timing for the appearance of item boxes, which provide players with race items. This class uses a prefab for the item box, and can initiate a respawn after a set delay, allowing for consistent and timed distribution of item boxes throughout the race.

LapCounter: Responsible for tracking the progress of cars in the game through laps and checkpoints. It keeps count of the number of laps completed, the checkpoints passed, and the time at each checkpoint. This class also manages the logic for determining when a race is finished and updates the UI to reflect the car's current position in the race. It also coordinates with the GameManager to signal the end of the race and manage post-race actions for player-controlled cars.

LapUIHandler: Responsible for managing and updating the lap information displayed on the game's user interface. It controls a Text element within the UI to show the current lap number and the total number of laps in the race.

MainMenu: Responsible for handling user interactions in the main menu of the game. It provides functionality for navigating to different parts of the game such as the car selection menu, options menu, and the functionality to exit the game.

OptionMenu: Manages the in-game options menu, providing functionalities such as adjusting game audio settings and setting the difficulty level for AI opponents. It utilizes Unity's AudioMixer to control the game's master volume.

PauseMenu: Handles the in-game pause functionality, allowing players to halt gameplay and access the pause menu. This class is responsible for toggling the visibility of the pause menu, adjusting the game's time scale to pause or resume the game, and providing options to navigate to different scenes.

PlayerInfo: Designed to store and manage information about both the human player and AI. It holds data such as player number, name, the unique ID of the selected car, AI status, and performance metrics like last race position, top speed, race completion time, and score. This class ensures that each player's data is encapsulated and managed efficiently, allowing for easy access and modification throughout the game.

PositionHandler: Responsible for tracking and updating the race positions of all cars in the game. It utilizes the LapCounter components attached to each car to determine their current positions based on the number of checkpoints passed and the time at the last checkpoint. This class orchestrates the sorting of cars based on their progress and updates the UI with the current positions using the PositionUIHandler.

PositionItemInfo: Responsible for managing the display of race position and car name information in a user interface element. It primarily interacts with text components from the Unity UI framework to update and reflect the current race position and the corresponding car's name.

PositionUIHandler: Responsible for managing the display of race positions for each car in the game. It handles the creation and updating of UI elements that represent each car's position in the race. This class uses either a vertical or horizontal layout to display the position information, based on the configuration.

RaceItemTypes: An abstract base class for items that can be used in the game. It defines a common interface for all race items, requiring the implementation of a Use method, which specifies how each item affects the car or the race when used.

ResultUIHandler: Responsible for managing the display of race results in the game. It controls a canvas that shows the player's score for the most recent race, their total accumulated score, and provides options to navigate to different menus.

SelectCarMenu: Manages the car selection interface in the game. This class handles UI interactions for car selection, including spawning car sprites, updating UI elements based on car purchase status, and navigating between different cars. It also manages the buying of cars using points and transitions the player to the level selection menu after car selection.

SelectTrackMenu: Manages the track selection interface in the game. It allows players to view, unlock, and select tracks for racing. This class also handles UI interactions for track selection, including displaying total points and updating the state of unlock buttons for each track.

Startup: Contains initialization logic that runs when the game loads, prior to the first scene being loaded. Its primary function is to instantiate a set of predefined GameObjects that are required to be present from the very beginning of the game.

Surface: Represents different types of surfaces that can be encountered in the game, such as road, grass, sand, etc. Each surface type has a different effect on the cars' speed.

SurfaceHandler: Responsible for detecting and managing the type of surface a car is driving on in the game. It uses collision detection to determine the surface underneath the car and updates the current surface type accordingly.

TimeUIHandler: Responsible for managing the display of race time in the game's user interface. It continuously updates a UI text element to show the elapsed time in minutes and seconds format. This class also uses a coroutine to efficiently update the time display at regular intervals, ensuring the time shown is current and accurate.

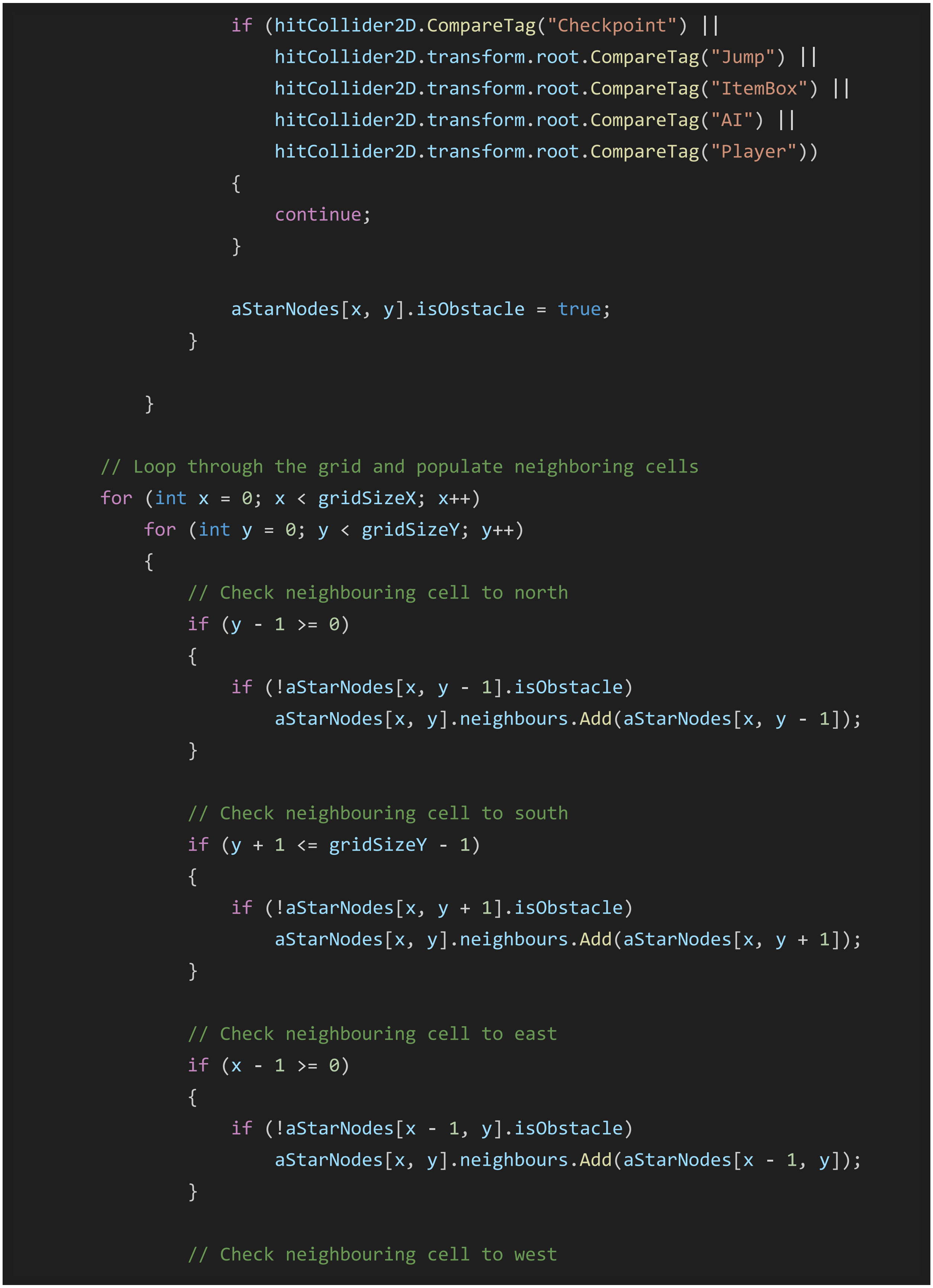
TrailHandler: Responsible for managing the visual trail effects for the cars. The class listens to the car's drifting and braking status, and activates or deactivates the trail emission accordingly.

Waypoint: Used for guiding AI-controlled cars along the track. Each waypoint represents a position on the track that AI cars aim to reach, effectively forming a path for them to follow.

PROGRAM

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

/**/
/*
    This class represents a node used in the A* pathfinding algorithm in a
grid-based environment.
    Each node contains information about its position in the grid, its neighbors,
and various costs
    associated with the pathfinding process. The costs include the distance from
the start,
    the estimated distance to the goal, and the total cost. The class also provides
    methods for calculating these costs based on the AI's current position and
destination,
    as well as a method to reset the node's properties for new pathfinding
calculations.
    Nodes can be marked as obstacles, affecting their traversal in the pathfinding
algorithm.
*/
/**/
public class AStarNode
{
    public Vector2Int gridPosition;

    public List<AStarNode> neighbours = new List<AStarNode>();

    public bool isObstacle = false;

    public int gCostDistanceFromStart = 0;

    public int hCostDistanceFromGoal = 0;

    public int fCostTotal = 0;

    public int pickedOrder = 0;

    bool isCostCalculated = false;
    /**/
    /*
    AStarNode::AStarNode(Vector2Int gridPosition_) AStarNode::AStarNode(Vector2Int
gridPosition_)

    NAME
        AStarNode - Constructor for the AStarNode class.

    SYNOPSIS
        AStarNode(Vector2Int gridPosition_);
            gridPosition_    --> The position of the node in the grid.

    DESCRIPTION
        This constructor initializes an AStarNode instance. It sets the grid
position of the node,
        which is used in the A* pathfinding algorithm to track the node's location
on the grid.

    RETURNS
        An instance of AStarNode.
    */
    /**/
    public AStarNode(Vector2Int gridPosition_)
    {
        gridPosition = gridPosition_;
    }

    /**/
    /*
    AStarNode::CalculateCostsForNode() AStarNode::CalculateCostsForNode()

    NAME
        AStarNode::CalculateCostsForNode - calculates costs for this node.

    SYNOPSIS
        void CalculateCostsForNode(Vector2Int aiPosition, Vector2Int
aiDestination);
            aiPosition       --> The current position of the AI.
            aiDestination    --> The destination position of the AI.
    DESCRIPTION
        This method calculates the costs (gCost, hCost, and fCost) for the node.
        If costs are already calculated, the method returns early.

    RETURNS
        Nothing.
    */
    /**/
    public void CalculateCostsForNode(Vector2Int aiPosition, Vector2Int aiDestination)
    {
        if (isCostCalculated)
            return;

        // Calculate the distance from the AI's current position to this node.
        gCostDistanceFromStart = Mathf.Abs(gridPosition.x - aiPosition.x) + Mathf.Abs(gridPosition.y - aiPosition.y);

        // Calculate the estimated distance from this node to the AI's destination.
        hCostDistanceFromGoal = Mathf.Abs(gridPosition.x - aiDestination.x) + Mathf.Abs(gridPosition.y - aiDestination.y);

        fCostTotal = gCostDistanceFromStart + hCostDistanceFromGoal;

        isCostCalculated = true;
    }

    /**/
    /*
    AStarNode::Reset() AStarNode::Reset()

    NAME
        AStarNode::Reset - resets the node's cost and order properties.

    SYNOPSIS
        void Reset();

    DESCRIPTION
        This method resets the node's properties, including costs and picked order.
        It is used to prepare the node for a new pathfinding calculation.

    RETURNS
        Nothing.
    */
    /**/
    public void Reset()
    {
        isCostCalculated = false;
        pickedOrder = 0;
        gCostDistanceFromStart = 0;
        hCostDistanceFromGoal = 0;
        fCostTotal = 0;
    }
}
```

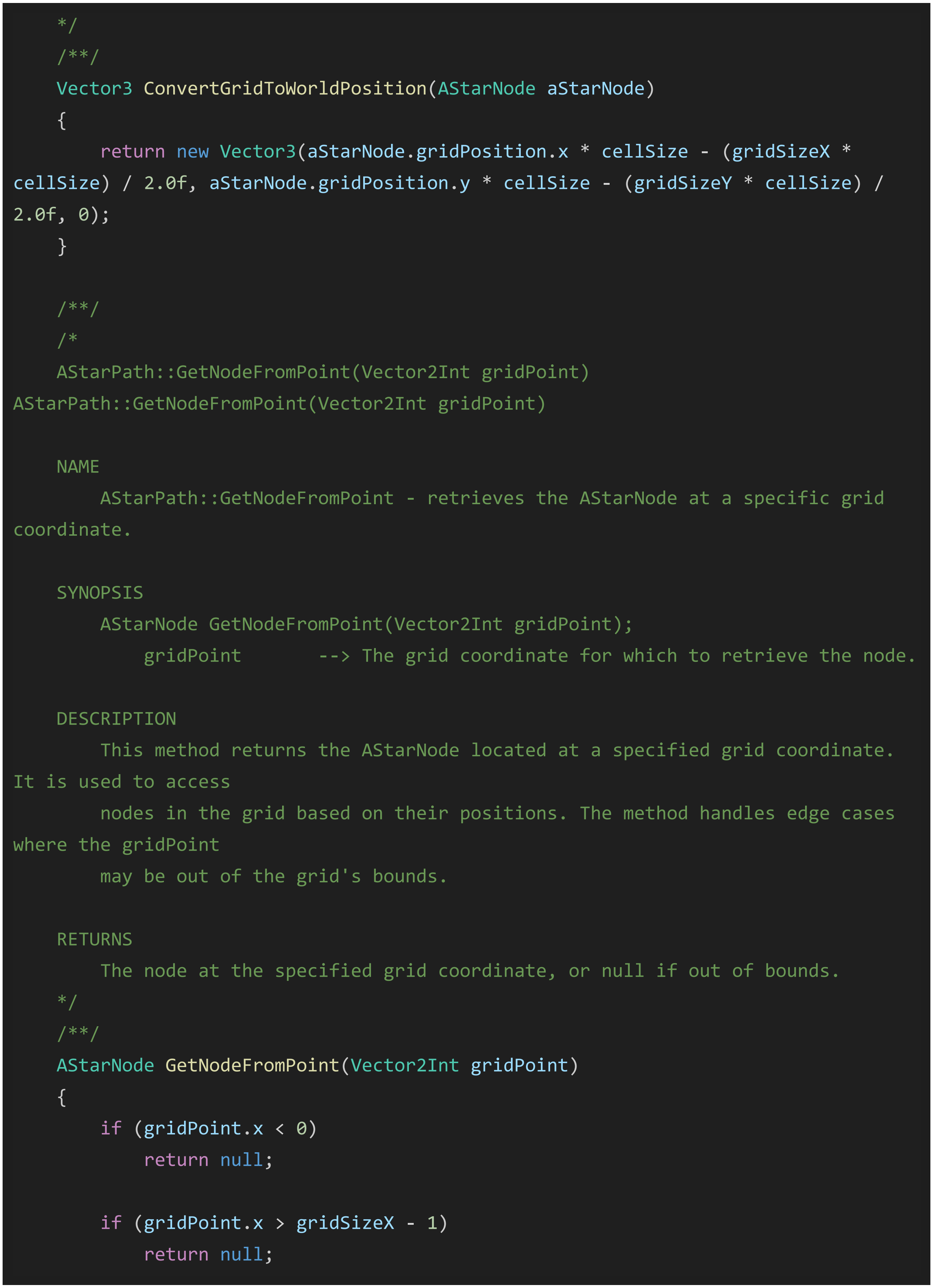
```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using System.Linq;
using UnityEditor;
using Unity.VisualScripting;

/**/
/*
    This class is responsible for implementing the A* pathfinding algorithm in a
grid-based environment.
    It manages the creation and initialization of the grid of AStarNodes,
calculates paths for AI navigation,
    and provides methods for converting between world and grid coordinates. Key
features include grid
    initialization, obstacle detection, and pathfinding.
*/
/**/
public class AStarPath : MonoBehaviour
{
    int gridSizeX = 60;
    int gridSizeY = 30;
    float cellSize = 1;

    AStarNode[,] aStarNodes;

    AStarNode startNode;

    List<AStarNode> nodesToCheck = new List<AStarNode>();
    List<AStarNode> nodesChecked = new List<AStarNode>();

    List<Vector2> aiPath = new List<Vector2>();

    /**/
    /*
    AStarPath::Start() AStarPath::Start()

    NAME
        AStarPath::Start - Initialization method called before the first frame update.

    SYNOPSIS
        void Start();

    DESCRIPTION
        This method is responsible for initializing the A* pathfinding algorithm. It calls the CreateGrid method
        to initialize and populate the grid with AStarNodes, and subsequently calls the FindPath method with
        a predefined destination to demonstrate pathfinding.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        CreateGrid();

        FindPath(new Vector2(32, 17));
    }
    /**/
    /*
    AStarPath::CreateGrid() AStarPath::CreateGrid()

    NAME
        AStarPath::CreateGrid - initializes and populates the grid with AStarNodes.

    SYNOPSIS
        void CreateGrid();

    DESCRIPTION
        This method creates a grid of AStarNode instances, each representing a cell
in the pathfinding grid.
        It initializes each node, assigns its position, and determines if it's an
obstacle based on colliders in the scene.

    RETURNS
        Nothing.
    */
    /**/
    void CreateGrid()
    {
        aStarNodes = new AStarNode[gridSizeX, gridSizeY];

        // Create the grid of nodes and check for obstacles
        for (int x = 0; x < gridSizeX; x++)
            for (int y = 0; y < gridSizeY; y++)
            {
                aStarNodes[x, y] = new AStarNode(new Vector2Int(x, y));

                Vector3 worldPosition = ConvertGridToWorldPosition(aStarNodes[x,
y]);

                Collider2D hitCollider2D = Physics2D.OverlapCircle(worldPosition,
cellSize / 2.0f);

                // Mark objects as obstacles unless thay have certain tags
                if (hitCollider2D != null)
                {
```



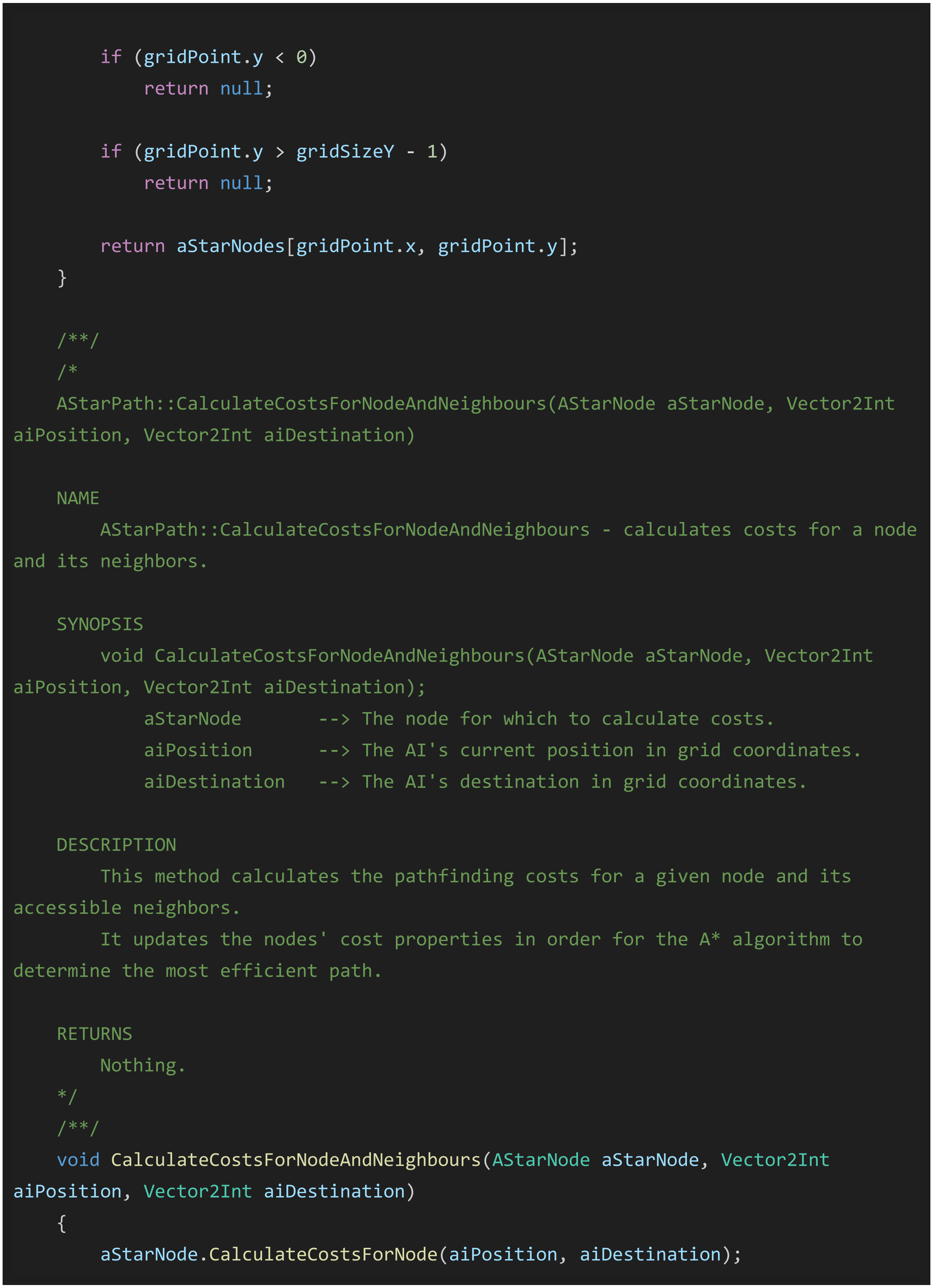
```
                if (hitCollider2D.CompareTag("Checkpoint") ||
                    hitCollider2D.transform.root.CompareTag("Jump") ||
                    hitCollider2D.transform.root.CompareTag("ItemBox") ||
                    hitCollider2D.transform.root.CompareTag("AI") ||
                    hitCollider2D.transform.root.CompareTag("Player"))
                {
                    continue;
                }

                aStarNodes[x, y].isObstacle = true;
            }

        }

    // Loop through the grid and populate neighboring cells
    for (int x = 0; x < gridSizeX; x++)
        for (int y = 0; y < gridSizeY; y++)
        {
            // Check neighbouring cell to north
            if (y - 1 >= 0)
            {
                if (!aStarNodes[x, y - 1].isObstacle)
                    aStarNodes[x, y].neighbours.Add(aStarNodes[x, y - 1]);
            }

            // Check neighbouring cell to south
            if (y + 1 <= gridSizeY - 1)
            {
                if (!aStarNodes[x, y + 1].isObstacle)
                    aStarNodes[x, y].neighbours.Add(aStarNodes[x, y + 1]);
            }

            // Check neighbouring cell to east
            if (x - 1 >= 0)
            {
                if (!aStarNodes[x - 1, y].isObstacle)
                    aStarNodes[x, y].neighbours.Add(aStarNodes[x - 1, y]);
            }

            // Check neighbouring cell to west
```

```
                if (x + 1 <= gridSizeX - 1)
                {
                    if (!aStarNodes[x + 1, y].isObstacle)
                        aStarNodes[x, y].neighbours.Add(aStarNodes[x + 1, y]);
                }
            }
    }

    /**/
    /*
    AStarPath::FindPath(Vector2 destination) AStarPath::FindPath(Vector2 
destination)

    NAME
        AStarPath::FindPath - calculates a path from the current position to the 
specified destination.

    SYNOPSIS
        List<Vector2> FindPath(Vector2 destination);
                destination     --> The target position for pathfinding.

    DESCRIPTION
        This method initiates the pathfinding process from the object's current 
position to the specified destination.
        It uses the A* algorithm to find the shortest path, considering obstacles 
and grid boundaries.
        The method returns null if no path is found.

    RETURNS
        List<Vector2> aiPath: A list of Vector2 points representing the calculated 
path.
    */
    /**/
    public List<Vector2> FindPath(Vector2 destination)
    {
        if (aStarNodes == null)
            return null;

        Reset();
        // Convert the destination from world to grid position
        Vector2Int destinationGridPoint = ConvertWorldToGridPoint(destination);
        Vector2Int currentPositionGridPoint =
ConvertWorldToGridPoint(transform.position);

        // Calculate the costs for the first node by starting the algorithm
        startNode = GetNodeFromPoint(currentPositionGridPoint);

        AStarNode currentNode = startNode;

        bool isDoneFindingPath = false;
        int pickedOrder = 1;

        // Loop until a path is found
        while (!isDoneFindingPath)
        {
            nodesToCheck.Remove(currentNode);

            currentNode.pickedOrder = pickedOrder;

            pickedOrder++;

            nodesChecked.Add(currentNode);

            if (currentNode.gridPosition == destinationGridPoint)
            {
                isDoneFindingPath = true;
                break;
            }

            CalculateCostsForNodeAndNeighbours(currentNode,
currentPositionGridPoint, destinationGridPoint);

            foreach (AStarNode neighbourNode in currentNode.neighbours)
            {
                if (nodesChecked.Contains(neighbourNode))
                    continue;

                if (nodesToCheck.Contains(neighbourNode))
                    continue;
                nodesToCheck.Add(neighbourNode);
            }

            nodesToCheck = nodesToCheck.OrderBy(x => x.fCostTotal).ThenBy(x =>
x.hCostDistanceFromGoal).ToList();

            if (nodesToCheck.Count == 0)
            {
                Debug.LogWarning($"No solutions left to check for
{transform.name}");
                return null;
            }
            else
            {
                currentNode = nodesToCheck[0];
            }
        }

        aiPath = CreatePathForAI(currentPositionGridPoint);

        return aiPath;
    }

    /**/
    /*
    AStarPath::ConvertWorldToGridPoint(Vector2 position)
AStarPath::ConvertWorldToGridPoint(Vector2 position)

    NAME
        AStarPath::ConvertWorldToGridPoint - converts a world position to a grid
coordinate.

    SYNOPSIS
        Vector2Int ConvertWorldToGridPoint(Vector2 position);
            position        --> The world position to be converted.

    DESCRIPTION
        This method converts a position in the world space (e.g., Unity's Vector2
coordinates) to a corresponding
```

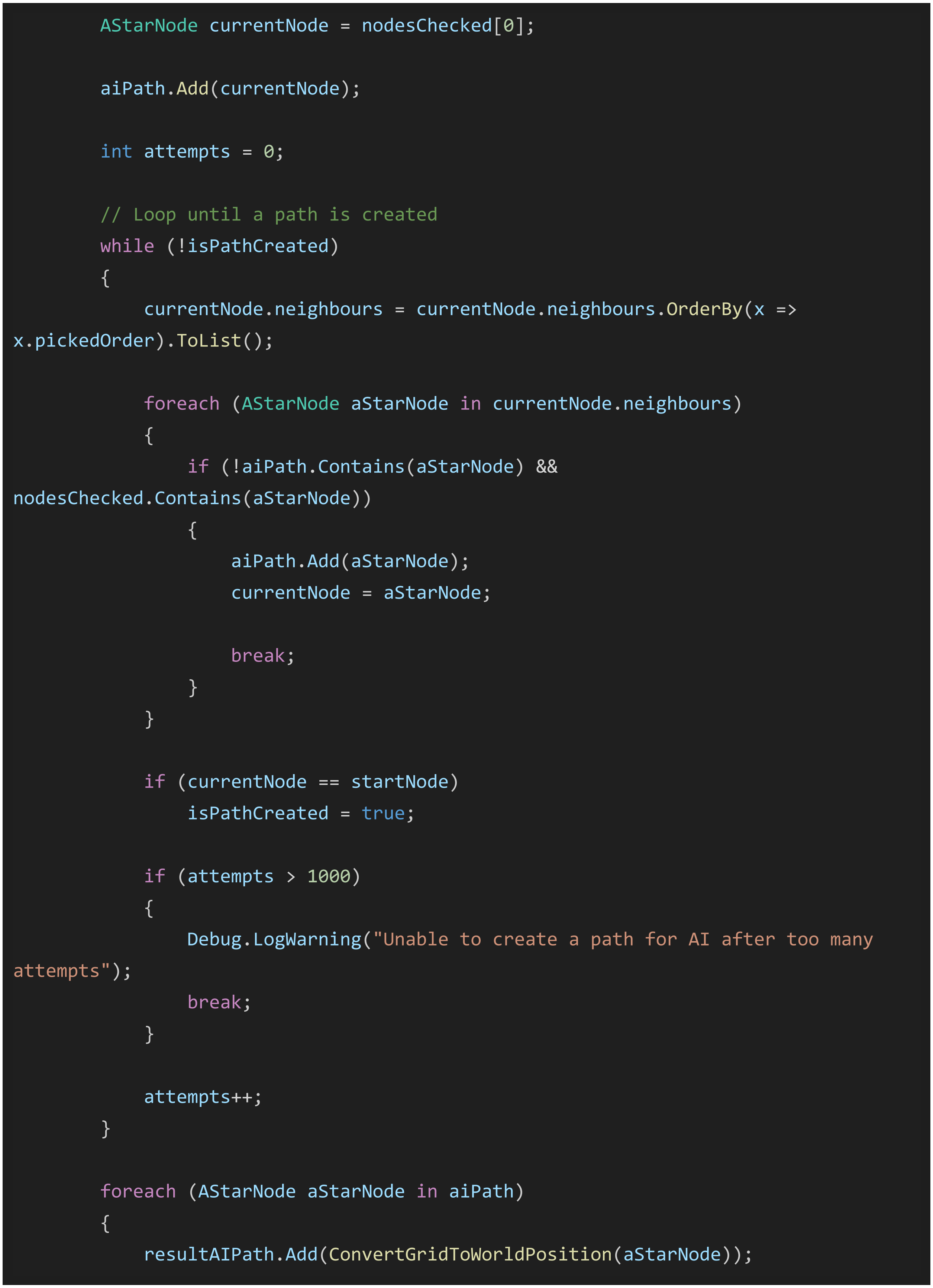
```
        point in the grid coordinate system used for pathfinding. It is essential
for translating game objects' positions
        into the grid-based context of the A* algorithm.

    RETURNS
        Vector2Int gridPoint: The grid coordinate corresponding to the given world
position.
    */
    /**/
    Vector2Int ConvertWorldToGridPoint(Vector2 position)
    {
        Vector2Int gridPoint = new Vector2Int(Mathf.RoundToInt(position.x /
cellSize + gridSizeX / 2.0f), Mathf.RoundToInt(position.y / cellSize + gridSizeY /
2.0f));

        return gridPoint;
    }

    /**/
    /*
    AStarPath::ConvertGridToWorldPosition(AStarNode aStarNode)
AStarPath::ConvertGridToWorldPosition(AStarNode aStarNode)

    NAME
        AStarPath::ConvertGridToWorldPosition - converts a grid coordinate to a
world position.

    SYNOPSIS
        Vector3 ConvertGridToWorldPosition(AStarNode aStarNode);
            aStarNode       --> The AStarNode whose grid position is to be
converted.

    DESCRIPTION
        This method converts a grid coordinate from an AStarNode back to a world
space position.
        It is useful for mapping the pathfinding results to actual positions in the
game world.

    RETURNS
        The world space position corresponding to the AStarNode's grid position.
```



```
    */
    /**/
    Vector3 ConvertGridToWorldPosition(AStarNode aStarNode)
    {
        return new Vector3(aStarNode.gridPosition.x * cellSize - (gridSizeX * cellSize) / 2.0f, aStarNode.gridPosition.y * cellSize - (gridSizeY * cellSize) / 2.0f, 0);
    }

    /**/
    /*
    AStarPath::GetNodeFromPoint(Vector2Int gridPoint)
AStarPath::GetNodeFromPoint(Vector2Int gridPoint)

    NAME
        AStarPath::GetNodeFromPoint - retrieves the AStarNode at a specific grid coordinate.

    SYNOPSIS
        AStarNode GetNodeFromPoint(Vector2Int gridPoint);
            gridPoint       --> The grid coordinate for which to retrieve the node.

    DESCRIPTION
        This method returns the AStarNode located at a specified grid coordinate. It is used to access
        nodes in the grid based on their positions. The method handles edge cases where the gridPoint
        may be out of the grid's bounds.

    RETURNS
        The node at the specified grid coordinate, or null if out of bounds.
    */
    /**/
    AStarNode GetNodeFromPoint(Vector2Int gridPoint)
    {
        if (gridPoint.x < 0)
            return null;

        if (gridPoint.x > gridSizeX - 1)
            return null;
```

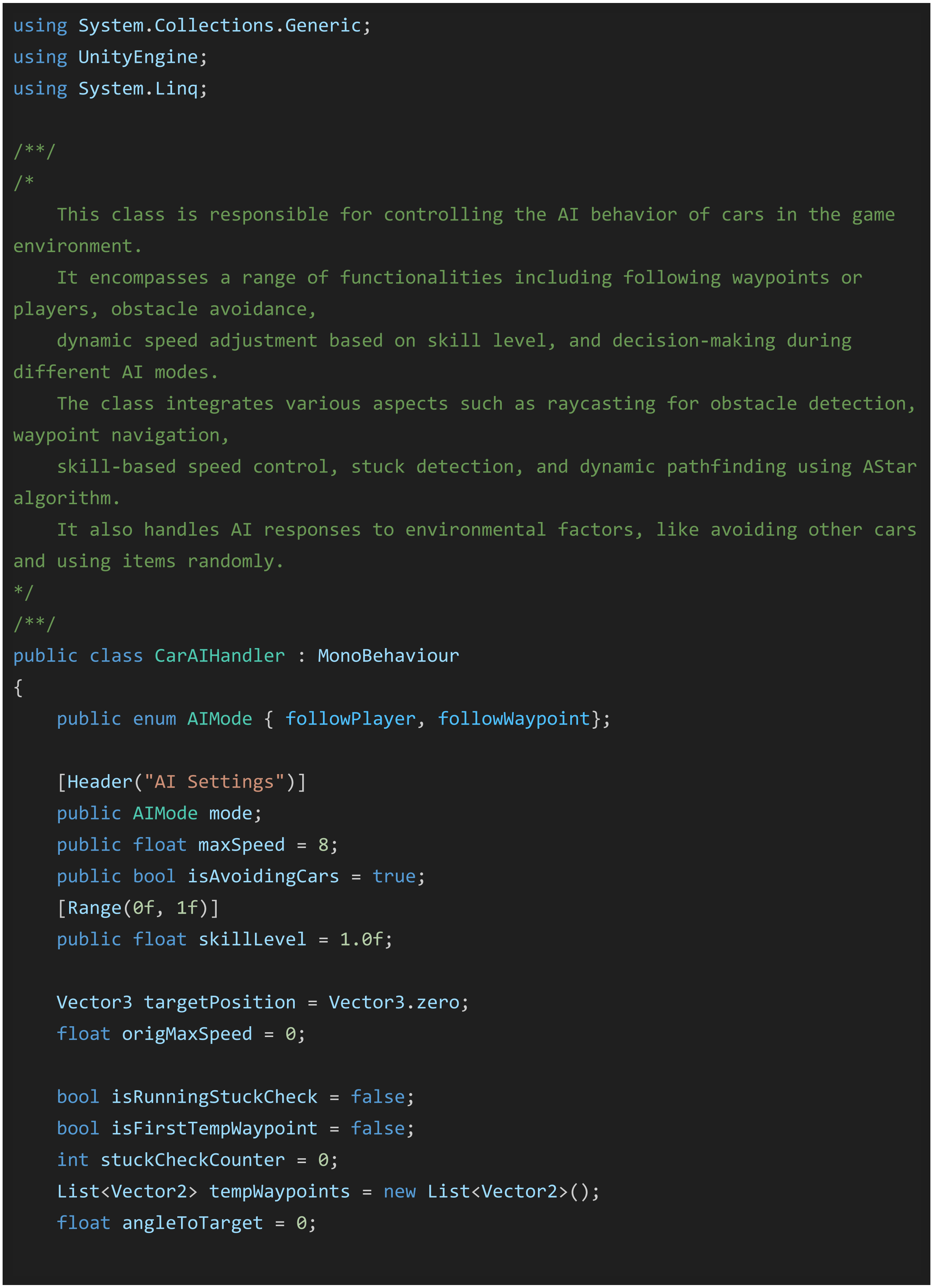


```
        if (gridPoint.y < 0)
            return null;

        if (gridPoint.y > gridSizeY - 1)
            return null;

        return aStarNodes[gridPoint.x, gridPoint.y];
    }

    /**/
    /*
    AStarPath::CalculateCostsForNodeAndNeighbours(AStarNode aStarNode, Vector2Int
aiPosition, Vector2Int aiDestination)

    NAME
        AStarPath::CalculateCostsForNodeAndNeighbours - calculates costs for a node
and its neighbors.

    SYNOPSIS
        void CalculateCostsForNodeAndNeighbours(AStarNode aStarNode, Vector2Int
aiPosition, Vector2Int aiDestination);
            aStarNode       --> The node for which to calculate costs.
            aiPosition      --> The AI's current position in grid coordinates.
            aiDestination   --> The AI's destination in grid coordinates.

    DESCRIPTION
        This method calculates the pathfinding costs for a given node and its
accessible neighbors.
        It updates the nodes' cost properties in order for the A* algorithm to
determine the most efficient path.

    RETURNS
        Nothing.
    */
    /**/
    void CalculateCostsForNodeAndNeighbours(AStarNode aStarNode, Vector2Int
aiPosition, Vector2Int aiDestination)
    {
        aStarNode.CalculateCostsForNode(aiPosition, aiDestination);
        foreach (AStarNode neighbourNode in aStarNode.neighbours)
        {
            neighbourNode.CalculateCostsForNode(aiPosition, aiDestination);
        }
    }

    /**/
    /*
    AStarPath::CreatePathForAI(Vector2Int currentPositionGridPoint)
AStarPath::CreatePathForAI(Vector2Int currentPositionGridPoint)

    NAME
        AStarPath::CreatePathForAI - creates the final path for AI navigation.

    SYNOPSIS
        List<Vector2> CreatePathForAI(Vector2Int currentPositionGridPoint);
            currentPositionGridPoint --> The current position of the AI in grid
coordinates.

    DESCRIPTION
        After the pathfinding process is complete, this method is used to backtrack
from the destination to the start,
        creating a list of waypoints that represent the path for the AI to follow.
The method ensures that the path is
        created efficiently and handles potential issues in path creation.

    RETURNS
        List<Vector2> resultAIPath: A list of waypoints representing the AI's path.
    */
    /**/
    List<Vector2> CreatePathForAI(Vector2Int currentPositionGridPoint)
    {
        List<Vector2> resultAIPath = new List<Vector2>();
        List<AStarNode> aiPath = new List<AStarNode>();

        nodesChecked.Reverse();

        bool isPathCreated = false;
```



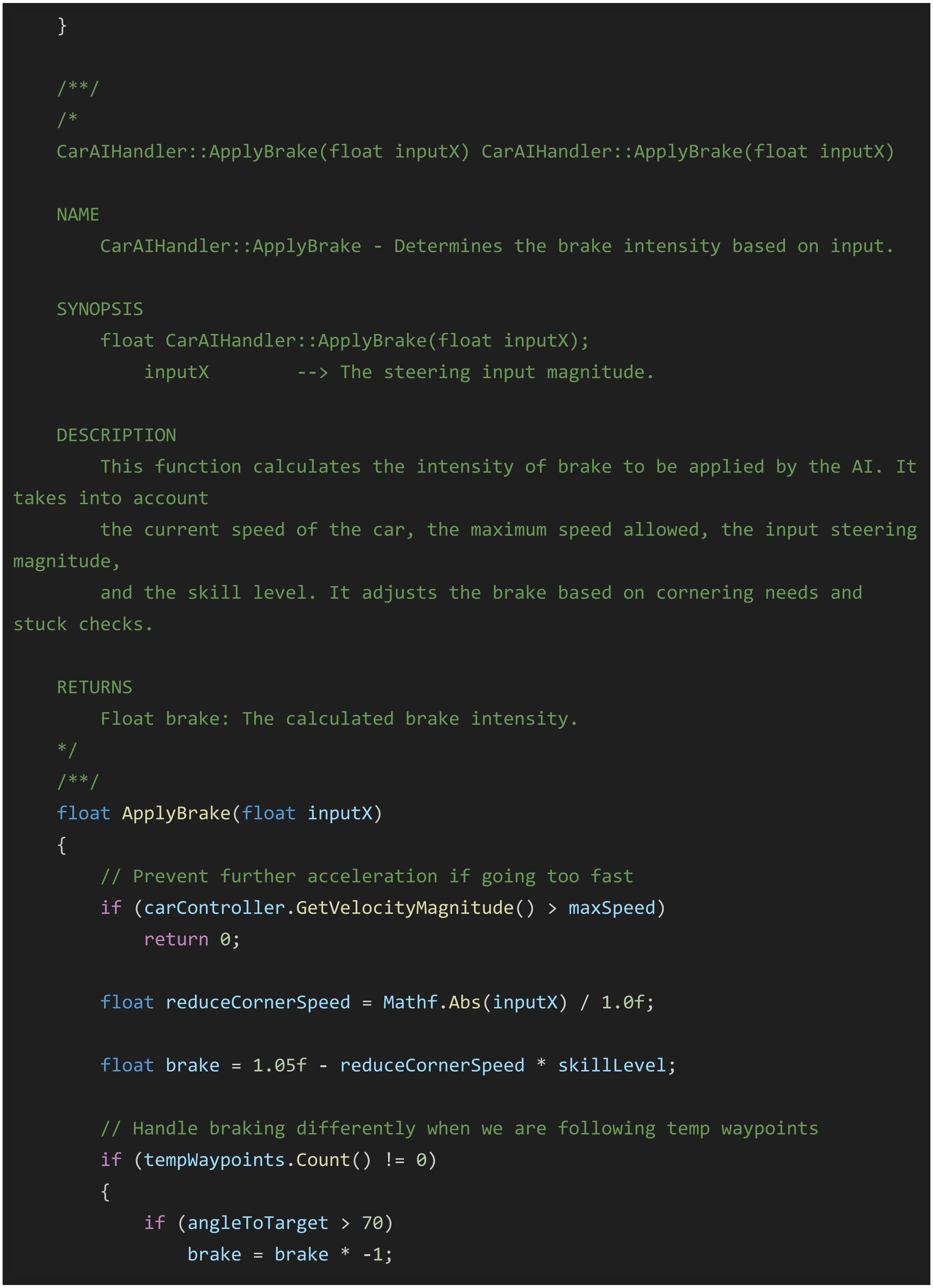
```
        AStarNode currentNode = nodesChecked[0];

        aiPath.Add(currentNode);

        int attempts = 0;

        // Loop until a path is created
        while (!isPathCreated)
        {
            currentNode.neighbours = currentNode.neighbours.OrderBy(x =>
x.pickedOrder).ToList();

            foreach (AStarNode aStarNode in currentNode.neighbours)
            {
                if (!aiPath.Contains(aStarNode) &&
nodesChecked.Contains(aStarNode))
                {
                    aiPath.Add(aStarNode);
                    currentNode = aStarNode;

                    break;
                }
            }

            if (currentNode == startNode)
                isPathCreated = true;

            if (attempts > 1000)
            {
                Debug.LogWarning("Unable to create a path for AI after too many
attempts");
                break;
            }

            attempts++;
        }

        foreach (AStarNode aStarNode in aiPath)
        {
            resultAIPath.Add(ConvertGridToWorldPosition(aStarNode));
        }

        resultAIPath.Reverse();

        return resultAIPath;
    }

    /**/
    /*
    AStarPath::Reset() AStarPath::Reset()

    NAME
        AStarPath::Reset - resets the pathfinding data for a new calculation.

    SYNOPSIS
        void Reset();

    DESCRIPTION
        This method clears the lists of nodes to check and checked nodes, and
resets each node in the grid
        in order to prepare for a new path calculation.

    RETURNS
        Nothing.
    */
    /**/
    private void Reset()
    {
        nodesToCheck.Clear();
        nodesChecked.Clear();
        aiPath.Clear();

        for (int x = 0; x < gridSizeX; x++)
            for (int y = 0; y < gridSizeY; y++)
                aStarNodes[x, y].Reset();
    }
}
```

```
using System.Collections;
```

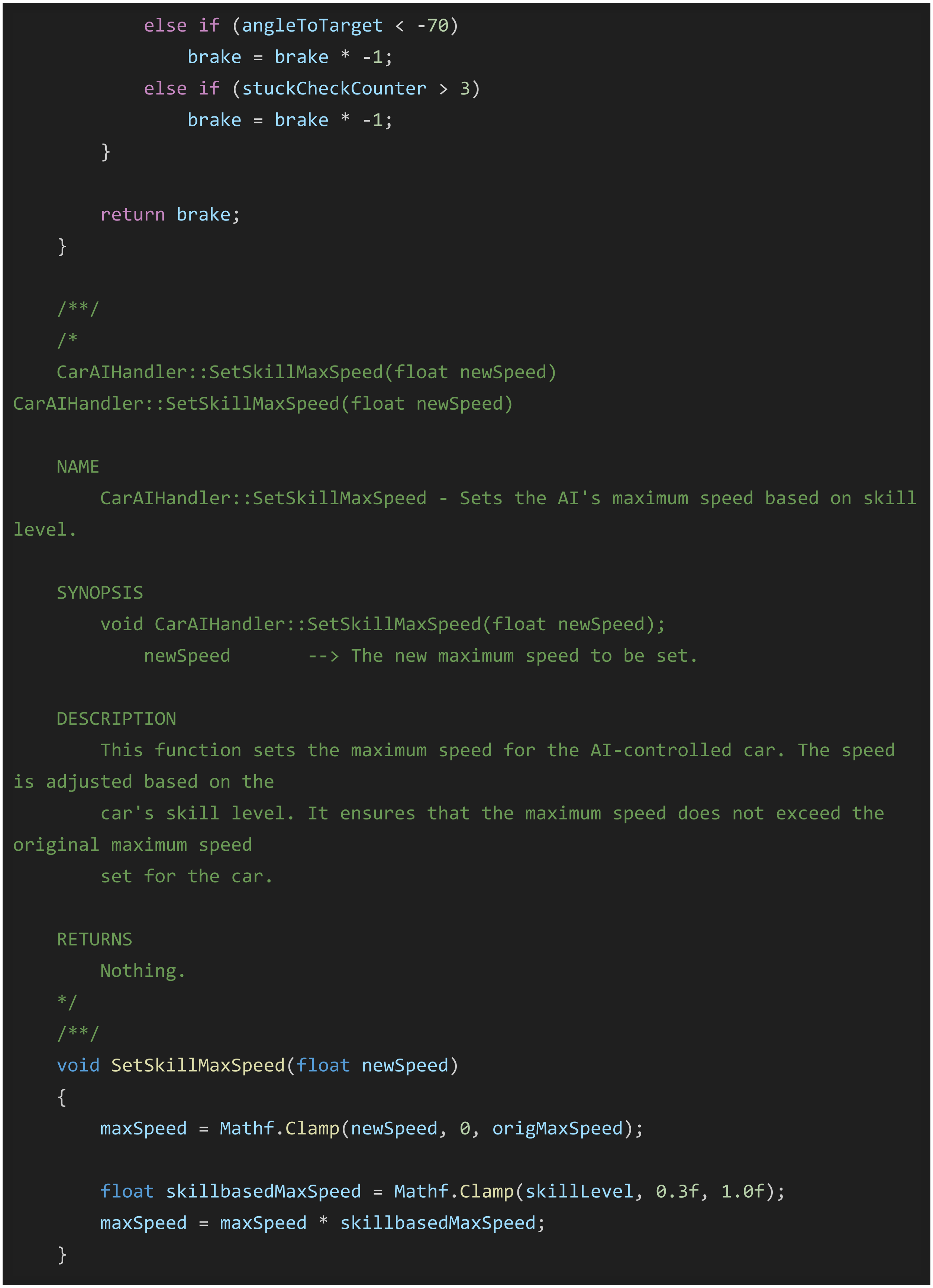


```csharp
using System.Collections.Generic;
using UnityEngine;
using System.Linq;

/**/
/*
    This class is responsible for controlling the AI behavior of cars in the game environment.
    It encompasses a range of functionalities including following waypoints or players, obstacle avoidance,
    dynamic speed adjustment based on skill level, and decision-making during different AI modes.
    The class integrates various aspects such as raycasting for obstacle detection, waypoint navigation,
    skill-based speed control, stuck detection, and dynamic pathfinding using AStar algorithm.
    It also handles AI responses to environmental factors, like avoiding other cars and using items randomly.
*/
/**/
public class CarAIHandler : MonoBehaviour
{
    public enum AIMode { followPlayer, followWaypoint};

    [Header("AI Settings")]
    public AIMode mode;
    public float maxSpeed = 8;
    public bool isAvoidingCars = true;
    [Range(0f, 1f)]
    public float skillLevel = 1.0f;

    Vector3 targetPosition = Vector3.zero;
    float origMaxSpeed = 0;

    bool isRunningStuckCheck = false;
    bool isFirstTempWaypoint = false;
    int stuckCheckCounter = 0;
    List<Vector2> tempWaypoints = new List<Vector2>();
    float angleToTarget = 0;
    private Vector2 avoidanceVectorLerped = Vector2.zero;

    Waypoint currentWaypoint = null;
    Waypoint previousWaypoint = null;
    Waypoint[] allWaypoints;

    PolygonCollider2D polygonCollider2D;

    CarController carController;
    AStarPath aStarLite;

    /**/
    /*
    CarAIHandler::Awake() CarAIHandler::Awake()

    NAME
        CarAIHandler::Awake - Initializes components and settings for AI-controlled
car.

    SYNOPSIS
        void CarAIHandler::Awake();

    DESCRIPTION
        This function is called when the script instance is being loaded. It
initializes the car controller,
        waypoints, and other components like PolygonCollider2D and AStarPath. It
also sets the original
        maximum speed and adjusts the skill level based on the GameManager's AI
difficulty setting.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        carController = GetComponent<CarController>();
        allWaypoints = FindObjectsOfType<Waypoint>();

        polygonCollider2D = GetComponentInChildren<PolygonCollider2D>();
        origMaxSpeed = maxSpeed;

        aStarLite = GetComponent<AStarPath>();

        skillLevel = GameManager.instance != null ?
GameManager.instance.AIDifficulty : 1.0f;
    }

    /**/
    /*
    CarAIHandler::Start() CarAIHandler::Start()

    NAME
        CarAIHandler::Start - Sets the initial maximum speed of the AI-controlled
car.

    SYNOPSIS
        void CarAIHandler::Start();

    DESCRIPTION
        This function sets the initial maximum speed of the AI-controlled car based
on its skill level.
        This is used to adjust the car's behavior at the start of the game scene.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        SetSkillMaxSpeed(maxSpeed);
    }

    /**/
    /*
    CarAIHandler::FixedUpdate() CarAIHandler::FixedUpdate()

    NAME
        CarAIHandler::FixedUpdate - Main AI behavior logic for obstacle detection
```

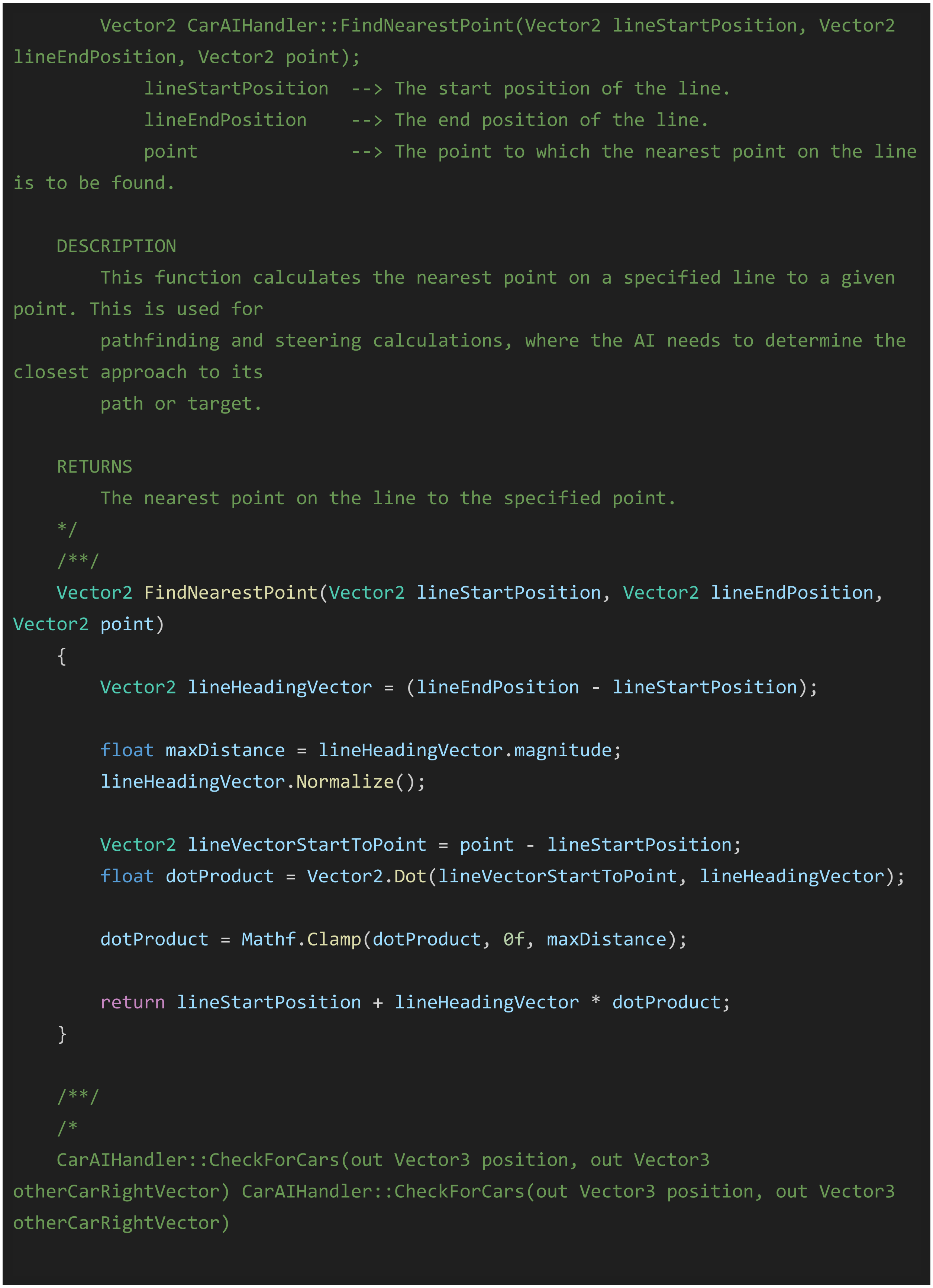
```
and movement.

    SYNOPSIS
        void CarAIHandler::FixedUpdate();

    DESCRIPTION
        This function is called at a fixed interval and contains the main logic for
the AI behavior.
        It manages obstacle detection, waypoint following, input vector
adjustments, and stuck check
        routines. It also handles item usage based on random probability.

    RETURNS
        Nothing.
    */
    /**/
    void FixedUpdate()
    {
        if (GameManager.instance.GetGameState() == GameStates.countdown)
            return;

        Vector2 inputVector = Vector2.zero;

        if (IsObstacleAhead(out RaycastHit2D hit, 10f)) // 5f is the detection
distance, adjust as needed
        {
            // Logic to avoid the obstacle
            AvoidObstacle(hit, ref inputVector);
        }
        else
        {
            // Regular AI behavior when no obstacle is detected
            switch (mode)
            {
                case AIMode.followWaypoint:
                    if (tempWaypoints.Count == 0)
                        FollowWaypoint();
                    else FollowTempWayPoints();
                    break;
            }
            inputVector.x = TurnToTarget();
            inputVector.y = ApplyBrake(inputVector.x);
        }

        // Checks if the car is stuck if the AI is unable to gain any speed.
        if (carController.GetVelocityMagnitude() < 0.5f && Mathf.Abs(inputVector.y)
> 0.01f && !isRunningStuckCheck)
            StartCoroutine(StuckCheckCO());

        // Check if car is still stuck after reversing. If not, drive forward.
        if (stuckCheckCounter >= 4 && !isRunningStuckCheck)
            StartCoroutine(StuckCheckCO());

        if (Random.value < 0.01f && carController.currentItem != null)
        {
            carController.currentItem.Use(carController);
            carController.currentItem = null;
        }

        carController.SetInputVector(inputVector);
    }

    /**/
    /*
    CarAIHandler::FollowWaypoint() CarAIHandler::FollowWaypoint()

    NAME
        CarAIHandler::FollowWaypoint - Logic for following a waypoint in the game.

    SYNOPSIS
        void CarAIHandler::FollowWaypoint();

    DESCRIPTION
        This function manages the AI's behavior when it is set to follow waypoints.
It determines the target
        position based on the current and next waypoints, adjusts the car's speed
and direction towards the
        target, and switches waypoints once the current one is reached.
    RETURNS
        Nothing.
    */
    /**/
    void FollowWaypoint()
    {
        // Pick the closest waypoint if none are set.
        if (currentWaypoint == null)
        {
            currentWaypoint = FindClosestWaypoint();
            previousWaypoint = currentWaypoint;
        }

        // Set the target on the waypoint's position
        if (currentWaypoint != null)
        {
            targetPosition = currentWaypoint.transform.position;

            float distToWaypoint = (targetPosition - transform.position).magnitude;

            if (distToWaypoint > 10)
            {
                Vector3 nearestPointOnLine =
FindNearestPoint(previousWaypoint.transform.position,
currentWaypoint.transform.position, transform.position);

                float segments = distToWaypoint / 20.0f;

                targetPosition = (targetPosition + nearestPointOnLine * segments) /
(segments + 1);
            }

            // Check if close enough to consider whether the waypoint has been
reached
            if (distToWaypoint <= currentWaypoint.minDistance)
            {
                if (currentWaypoint.maxSpeed > 0)
                    SetSkillMaxSpeed(currentWaypoint.maxSpeed);
                else
                    SetSkillMaxSpeed(1000);
                previousWaypoint = currentWaypoint;

                currentWaypoint = currentWaypoint.nextWaypoint[Random.Range(0,
currentWaypoint.nextWaypoint.Length)];
            }
        }
    }

    /**/
    /*
    CarAIHandler::FollowTempWayPoints() CarAIHandler::FollowTempWayPoints()

    NAME
        CarAIHandler::FollowTempWayPoints - Follows temporary waypoints generated
during stuck checks.

    SYNOPSIS
        void CarAIHandler::FollowTempWayPoints();

    DESCRIPTION
        This function is used when the AI car is following a set of temporary
waypoints, usually generated
        after a stuck check. It guides the car along these waypoints until it
reaches the target position or
        resumes its normal waypoint following behavior.

    RETURNS
        Nothing.
    */
    /**/
    void FollowTempWayPoints()
    {
        targetPosition = tempWaypoints[0];

        float distanceToWayPoint = (targetPosition - transform.position).magnitude;

        SetSkillMaxSpeed(5);

        float minDistanceToWaypoint = 1.5f;
        if (!isFirstTempWaypoint)
            minDistanceToWaypoint = 3.0f;

        if (distanceToWayPoint <= minDistanceToWaypoint)
        {
            tempWaypoints.RemoveAt(0);
            isFirstTempWaypoint = false;
        }
    }

    /**/
    /*
    CarAIHandler::FindClosestWaypoint() CarAIHandler::FindClosestWaypoint()

    NAME
        CarAIHandler::FindClosestWaypoint - Finds the closest waypoint to the AI
car.

    SYNOPSIS
        Waypoint CarAIHandler::FindClosestWaypoint();

    DESCRIPTION
        This function searches through all available waypoints and returns the one
that is closest to the
        AI car's current position. It's used to determine the next target waypoint
for the AI to follow.

    RETURNS
        The closest waypoint to the AI car.
    */
    /**/
    Waypoint FindClosestWaypoint()
    {
        return allWaypoints
            .OrderBy(t => Vector3.Distance(transform.position,
t.transform.position))
            .FirstOrDefault();
    }
    /**/
    /*
    CarAIHandler::TurnToTarget() CarAIHandler::TurnToTarget()

    NAME
        CarAIHandler::TurnToTarget - Calculates the steering amount towards the target.

    SYNOPSIS
        float CarAIHandler::TurnToTarget();

    DESCRIPTION
        This function calculates the amount of steering needed for the AI to turn towards its current target.
        It considers the current direction of the car and the position of the target, adjusting the steering
        to align the car towards the target.

    RETURNS
        Float steerAmount: The amount of steering required to turn towards the target.
    */
    /**/
    float TurnToTarget()
    {
        Vector2 vectorToTarget = targetPosition - transform.position;
        vectorToTarget.Normalize();

        if (isAvoidingCars)
            AvoidCars(vectorToTarget, out vectorToTarget);

        // Calculate an angle towards the target
        angleToTarget = Vector2.SignedAngle(transform.up, vectorToTarget);
        angleToTarget *= -1;

        float steerAmount = angleToTarget / 45.0f;

        steerAmount = Mathf.Clamp(steerAmount, -1.0f, 1.0f);

        return steerAmount;
```



```
    }

    /**/
    /*
    CarAIHandler::ApplyBrake(float inputX) CarAIHandler::ApplyBrake(float inputX)

    NAME
        CarAIHandler::ApplyBrake - Determines the brake intensity based on input.

    SYNOPSIS
        float CarAIHandler::ApplyBrake(float inputX);
            inputX        --> The steering input magnitude.

    DESCRIPTION
        This function calculates the intensity of brake to be applied by the AI. It
takes into account
        the current speed of the car, the maximum speed allowed, the input steering
magnitude,
        and the skill level. It adjusts the brake based on cornering needs and
stuck checks.

    RETURNS
        Float brake: The calculated brake intensity.
    */
    /**/
    float ApplyBrake(float inputX)
    {
        // Prevent further acceleration if going too fast
        if (carController.GetVelocityMagnitude() > maxSpeed)
            return 0;

        float reduceCornerSpeed = Mathf.Abs(inputX) / 1.0f;

        float brake = 1.05f - reduceCornerSpeed * skillLevel;

        // Handle braking differently when we are following temp waypoints
        if (tempWaypoints.Count() != 0)
        {
            if (angleToTarget > 70)
                brake = brake * -1;
```



```
            else if (angleToTarget < -70)
                brake = brake * -1;
            else if (stuckCheckCounter > 3)
                brake = brake * -1;
        }

        return brake;
    }

    /**/
    /*
    CarAIHandler::SetSkillMaxSpeed(float newSpeed)
CarAIHandler::SetSkillMaxSpeed(float newSpeed)

    NAME
        CarAIHandler::SetSkillMaxSpeed - Sets the AI's maximum speed based on skill
level.

    SYNOPSIS
        void CarAIHandler::SetSkillMaxSpeed(float newSpeed);
            newSpeed       --> The new maximum speed to be set.

    DESCRIPTION
        This function sets the maximum speed for the AI-controlled car. The speed
is adjusted based on the
        car's skill level. It ensures that the maximum speed does not exceed the
original maximum speed
        set for the car.

    RETURNS
        Nothing.
    */
    /**/
    void SetSkillMaxSpeed(float newSpeed)
    {
        maxSpeed = Mathf.Clamp(newSpeed, 0, origMaxSpeed);

        float skillbasedMaxSpeed = Mathf.Clamp(skillLevel, 0.3f, 1.0f);
        maxSpeed = maxSpeed * skillbasedMaxSpeed;
    }
    /**/
    /*
    CarAIHandler::SetSkillLevel(float skillLevel) CarAIHandler::SetSkillLevel(float 
skillLevel)

    NAME
        CarAIHandler::SetSkillLevel - Sets the skill level of the AI.

    SYNOPSIS
        void CarAIHandler::SetSkillLevel(float skillLevel);
            skillLevel    --> The skill level to be set for the AI.

    DESCRIPTION
        This function sets the skill level of the AI, which influences its driving 
behavior and decision-making.
        A higher skill level typically results in more aggressive and efficient 
driving behavior.

    RETURNS
        Nothing.
    */
    /**/
    public void SetSkillLevel(float skillLevel)
    {
        this.skillLevel = skillLevel;
        SetSkillMaxSpeed(maxSpeed); // Update max speed based on new skill level
    }

    /**/
    /*
    CarAIHandler::FindNearestPoint(Vector2 lineStartPosition, Vector2 
lineEndPosition, Vector2 point) CarAIHandler::FindNearestPoint(Vector2 
lineStartPosition, Vector2 lineEndPosition, Vector2 point)

    NAME
        CarAIHandler::FindNearestPoint - Finds the nearest point on a line to a 
given point.

    SYNOPSIS
```



```
        Vector2 CarAIHandler::FindNearestPoint(Vector2 lineStartPosition, Vector2
lineEndPosition, Vector2 point);
            lineStartPosition  --> The start position of the line.
            lineEndPosition    --> The end position of the line.
            point              --> The point to which the nearest point on the line
is to be found.

    DESCRIPTION
        This function calculates the nearest point on a specified line to a given
point. This is used for
        pathfinding and steering calculations, where the AI needs to determine the
closest approach to its
        path or target.

    RETURNS
        The nearest point on the line to the specified point.
    */
    /**/
    Vector2 FindNearestPoint(Vector2 lineStartPosition, Vector2 lineEndPosition,
Vector2 point)
    {
        Vector2 lineHeadingVector = (lineEndPosition - lineStartPosition);

        float maxDistance = lineHeadingVector.magnitude;
        lineHeadingVector.Normalize();

        Vector2 lineVectorStartToPoint = point - lineStartPosition;
        float dotProduct = Vector2.Dot(lineVectorStartToPoint, lineHeadingVector);

        dotProduct = Mathf.Clamp(dotProduct, 0f, maxDistance);

        return lineStartPosition + lineHeadingVector * dotProduct;
    }

    /**/
    /*
    CarAIHandler::CheckForCars(out Vector3 position, out Vector3
otherCarRightVector) CarAIHandler::CheckForCars(out Vector3 position, out Vector3
otherCarRightVector)
```

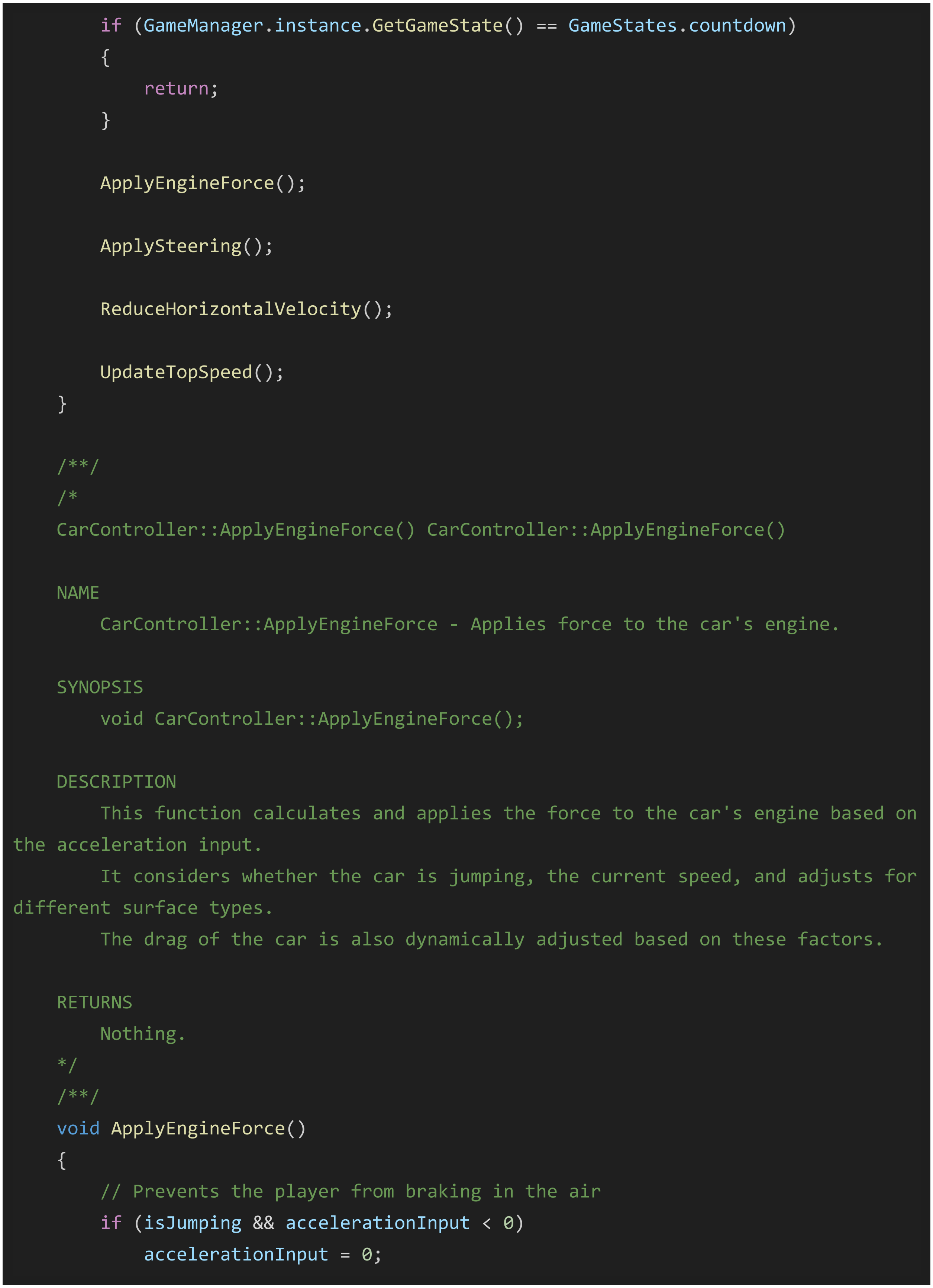
```
    NAME
        CarAIHandler::CheckForCars - Checks for the presence of other cars nearby.

    SYNOPSIS
        bool CarAIHandler::CheckForCars(out Vector3 position, out Vector3 
otherCarRightVector);
            position              --> Out parameter to store the position of the 
detected car.
            otherCarRightVector   --> Out parameter to store the right vector of 
the detected car.

    DESCRIPTION
        This function performs a check to see if there are other cars in close 
proximity to the AI car.
        It uses raycasting to detect other cars and, if found, provides their 
position and orientation.
        This is used for collision avoidance and pathfinding.

    RETURNS
        True if another car is detected, False otherwise.
    */
    /**/
    bool CheckForCars(out Vector3 position, out Vector3 otherCarRightVector)
    {
        polygonCollider2D.enabled = false;

        RaycastHit2D raycastHit2D = Physics2D.CircleCast(transform.position + 
transform.up * 0.5f, 0.5f, transform.up, 8, 1 << LayerMask.NameToLayer("Car"));

        polygonCollider2D.enabled = true;

        if (raycastHit2D.collider != null)
        {
            position = raycastHit2D.collider.transform.position;
            otherCarRightVector = raycastHit2D.collider.transform.right;
            return true;
        }

        position = Vector3.zero;
        otherCarRightVector = Vector3.zero;
        return false;
    }

    /**/
    /*
    CarAIHandler::AvoidCars(Vector2 vectorToTarget, out Vector2 newVectorToTarget)
CarAIHandler::AvoidCars(Vector2 vectorToTarget, out Vector2 newVectorToTarget)

    NAME
        CarAIHandler::AvoidCars - Adjusts the car's path to avoid other cars.

    SYNOPSIS
        void CarAIHandler::AvoidCars(Vector2 vectorToTarget, out Vector2
newVectorToTarget);
            vectorToTarget        --> The current vector towards the target.
            newVectorToTarget     --> Out parameter for the adjusted vector after
avoiding other cars.

    DESCRIPTION
        This function modifies the car's current path to avoid collisions with
other cars. It calculates an
        avoidance vector and combines it with the current path vector to steer the
AI car away from other vehicles.

    RETURNS
        Nothing.
    */
    /**/
    void AvoidCars(Vector2 vectorToTarget, out Vector2 newVectorToTarget)
    {
        if (CheckForCars(out Vector3 otherCarPosition, out Vector3
otherCarRightVector))
        {
            Vector2 avoidanceVector = Vector2.zero;

            avoidanceVector = Vector2.Reflect((otherCarPosition -
transform.position).normalized, otherCarRightVector);

            float distanceToTarget = (targetPosition -
transform.position).magnitude;

            float driveToTargetInfluence = 6.0f / distanceToTarget;

            driveToTargetInfluence = Mathf.Clamp(driveToTargetInfluence, 0.30f,
1.0f);

            float avoidanceInfluence = 1.0f - driveToTargetInfluence;

            newVectorToTarget = vectorToTarget * driveToTargetInfluence +
avoidanceVector * avoidanceInfluence;
            newVectorToTarget.Normalize();

            return;
        }

        newVectorToTarget = vectorToTarget;
    }

    /**/
    /*
    CarAIHandler::IsObstacleAhead(out RaycastHit2D hit, float detectionDistance)
CarAIHandler::IsObstacleAhead(out RaycastHit2D hit, float detectionDistance)

    NAME
        CarAIHandler::IsObstacleAhead - Checks for obstacles ahead of the car.

    SYNOPSIS
        bool CarAIHandler::IsObstacleAhead(out RaycastHit2D hit, float
detectionDistance);
            hit                  --> RaycastHit2D object to store information about
the obstacle hit.
            detectionDistance    --> Distance within which to check for obstacles.

    DESCRIPTION
        This function uses raycasting to detect if there are any obstacles in the
car's path within a specified
        distance. It helps in determining whether the car needs to take evasive
action to avoid a collision.
```

```
    RETURNS
        True if an obstacle is detected, False otherwise.
    */
    /**/
    bool IsObstacleAhead(out RaycastHit2D hit, float detectionDistance)
    {
        hit = Physics2D.Raycast(transform.position, transform.up,
detectionDistance, LayerMask.GetMask("Obstacle"));
        return hit.collider != null;
    }

    /**/
    /*
    CarAIHandler::AvoidObstacle(RaycastHit2D hit, ref Vector2 inputVector)
CarAIHandler::AvoidObstacle(RaycastHit2D hit, ref Vector2 inputVector)

    NAME
        CarAIHandler::AvoidObstacle - Manages the car's response to detected
obstacles.

    SYNOPSIS
        void CarAIHandler::AvoidObstacle(RaycastHit2D hit, ref Vector2
inputVector);
            hit            --> The RaycastHit2D object containing information about
the detected obstacle.
            inputVector    --> The current input vector for the car, to be adjusted
for obstacle avoidance.

    DESCRIPTION
        This function adjusts the car's input vector to avoid an obstacle detected
in its path. It calculates
        an avoidance vector based on the obstacle's position and orientation, and
smoothly transitions the
        car's current direction to this new vector to steer clear of the obstacle.

    RETURNS
        Nothing.
    */
    /**/
    void AvoidObstacle(RaycastHit2D hit, ref Vector2 inputVector)
    {
        Vector2 directionToObstacle = hit.point - (Vector2)transform.position;
        Vector2 avoidanceVector = Vector2.Reflect(directionToObstacle.normalized, hit.normal);

        avoidanceVectorLerped = Vector2.Lerp(avoidanceVectorLerped, avoidanceVector, Time.fixedDeltaTime * 4);

        float angle = Vector2.SignedAngle(transform.up, avoidanceVectorLerped);
        inputVector.x = Mathf.Clamp(-angle / 45.0f, -1.0f, 1.0f);
    }

    /**/
    /*
    CarAIHandler::StuckCheckCO() CarAIHandler::StuckCheckCO()

    NAME
        CarAIHandler::StuckCheckCO - Coroutine for checking if the AI car is stuck.

    SYNOPSIS
        IEnumerator CarAIHandler::StuckCheckCO();

    DESCRIPTION
        This coroutine is executed to determine if the AI-controlled car is stuck. It checks the car's position
        over a period of time to see if there has been significant movement. If the car is deemed to be stuck,
        it calculates a new path using the AStar algorithm to navigate around the obstacle.

    RETURNS
        IEnumerator that yields execution for a set duration and then performs checks and actions based on the AI car's movement.
    */
    /**/
    IEnumerator StuckCheckCO()
    {
        Vector3 initialStuckPosition = transform.position;

        isRunningStuckCheck = true;
        yield return new WaitForSeconds(0.7f);

        // If the car has not moved after waiting, then its stuck
        if ((transform.position - initialStuckPosition).sqrMagnitude < 3)
        {
            tempWaypoints = aStarLite.FindPath(currentWaypoint.transform.position);

            if (tempWaypoints == null)
                tempWaypoints = new List<Vector2>();

            stuckCheckCounter++;

            isFirstTempWaypoint = true;
        }
        else stuckCheckCounter = 0;

        isRunningStuckCheck = false;
    }
}
```

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

/**/
/*
    The CarController class manages the physics, control, and behavior of a car in
the game environment. It handles
    fundamental car dynamics such as acceleration, steering, drifting, and jumping,
as well as interactions with different
    surfaces and the use of power-ups and items. This class controls the car's
response to player inputs and AI commands,
    adjusting its behavior based on factors like surface type, obstacles, and
current speed. Special features like
    jumps and item usage are also managed within this class.
*/
/**/
public class CarController : MonoBehaviour
```

```
{
    [Header("Car Settings")]
    public float driftFactor = 0.93f;
    public float accelerationFactor = 6.0f;
    public float turnFactor = 3.5f;
    public float maxSpeed = 8;

    [Header("Sprite Settings")]
    public SpriteRenderer carSpriteRenderer;
    public SpriteRenderer carShadowRenderer;

    [Header("Jump Settings")]
    public AnimationCurve jumpCurve;

    [Header("Item Settings")]
    public GameObject mudPuddlePrefab;

    [Header("Boost Settings")]
    public float boostSpeed = 10f;
    public float boostDuration = 5f;

    public ParticleSystem speedBoostParticles;

    public float originalMaxSpeed;
    private bool isBoosting;

    float accelerationInput = 0;
    float steeringInput = 0;
    float rotationAngle = 0;
    float velocityVsUp = 0;
    bool isJumping = false;
    private float topSpeed = 0f;

    public RaceItem currentItem;

    Rigidbody2D carRigidbody2D;
    Collider2D carCollider;
    SurfaceHandler surfaceHandler;
    CarSFXHandler carSFXHandler;
    /**/
    /*
    CarController::TopSpeed

    NAME
        CarController::TopSpeed - Gets the top speed achieved by the car.

    DESCRIPTION
        This property returns the highest speed that the car has achieved. Used for tracking performance
        and gameplay elements that depend on the car's speed.

    RETURNS
        Float topSpeed: The top speed of the car.
    */
    /**/
    public float TopSpeed
    {
        get { return topSpeed; }
    }

    /**/
    /*
    CarController::AssignItem(RaceItem item) CarController::AssignItem(RaceItem item)

    NAME
        CarController::AssignItem - Assigns a race item to the car.

    SYNOPSIS
        void CarController::AssignItem(RaceItem item);
            item    --> The race item to be assigned to the car.

    DESCRIPTION
        This function assigns a race item to the car.

    RETURNS
        Nothing.
    */
    /**/
    public void AssignItem(RaceItem item)
    {
        currentItem = item;
    }

    /**/
    /*
    CarController::Awake() CarController::Awake()

    NAME
        CarController::Awake - Initializes components for the car controller.

    SYNOPSIS
        void CarController::Awake();

    DESCRIPTION
        This function is called when the script instance is being loaded. It
initializes the Rigidbody2D,
        Colliders, SurfaceHandler, and CarSFXHandler components attached to the
car. It also sets the
        original maximum speed of the car.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        carRigidbody2D = GetComponent<Rigidbody2D>();
        carCollider = GetComponentInChildren<Collider2D>();
        surfaceHandler = GetComponent<SurfaceHandler>();
        carSFXHandler = GetComponent<CarSFXHandler>();
        originalMaxSpeed = maxSpeed;
    }

    /**/
    /*
    CarController::Start() CarController::Start()

    NAME
```

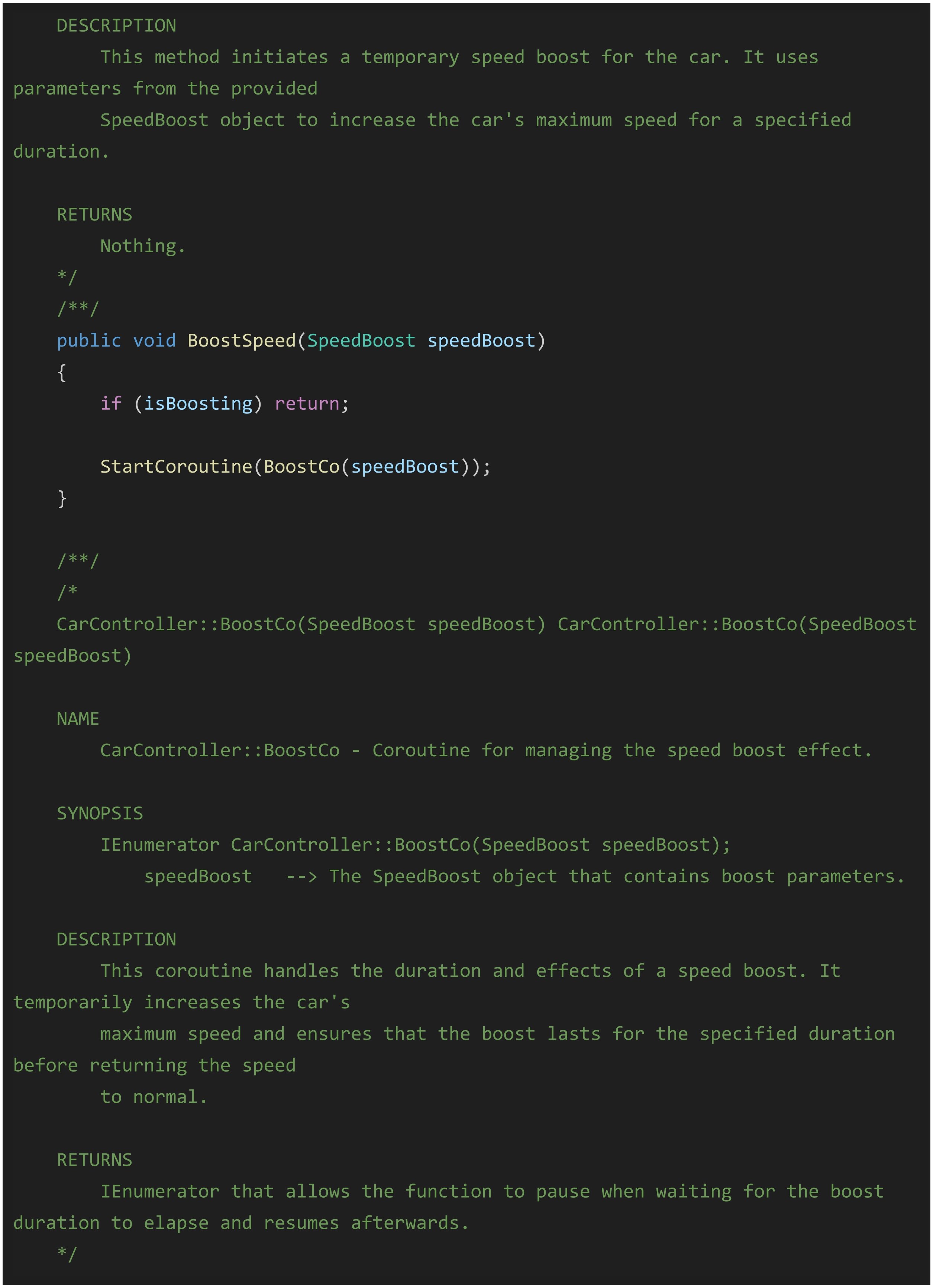
```
        CarController::Start - Initialization at the start of the scene.

    SYNOPSIS
        void CarController::Start();

    DESCRIPTION
        This function initializes the car's rotation angle based on its current
orientation.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        rotationAngle = transform.rotation.eulerAngles.z;
    }

    /**/
    /*
    CarController::FixedUpdate() CarController::FixedUpdate()

    NAME
        CarController::FixedUpdate - Handles the physics updates for the car.

    SYNOPSIS
        void CarController::FixedUpdate();

    DESCRIPTION
        This method is called every fixed framerate frame and is used for handling
physics-based updates.
        It includes applying engine force, steering, and managing horizontal
velocity.

    RETURNS
        Nothing.
    */
    /**/
    void FixedUpdate()
    {
```



```
        if (GameManager.instance.GetGameState() == GameStates.countdown)
        {
            return;
        }

        ApplyEngineForce();

        ApplySteering();

        ReduceHorizontalVelocity();

        UpdateTopSpeed();
    }

    /**/
    /*
    CarController::ApplyEngineForce() CarController::ApplyEngineForce()

    NAME
        CarController::ApplyEngineForce - Applies force to the car's engine.

    SYNOPSIS
        void CarController::ApplyEngineForce();

    DESCRIPTION
        This function calculates and applies the force to the car's engine based on the acceleration input.
        It considers whether the car is jumping, the current speed, and adjusts for different surface types.
        The drag of the car is also dynamically adjusted based on these factors.

    RETURNS
        Nothing.
    */
    /**/
    void ApplyEngineForce()
    {
        // Prevents the player from braking in the air
        if (isJumping && accelerationInput < 0)
            accelerationInput = 0;
        // Calculate how much we are going forward in terms of velocity's direction
        velocityVsUp = Vector2.Dot(transform.up, carRigidbody2D.velocity);

        // Limit to max speed of car
        if (velocityVsUp > maxSpeed && accelerationInput > 0)
            return;

        // Limit to 50% of max speed when driving in reverse
        if (velocityVsUp < -maxSpeed * 0.5f && accelerationInput < 0)
            return;

        // Limit the speed the car goes in any direction while accelerating
        if (carRigidbody2D.velocity.sqrMagnitude > maxSpeed * maxSpeed &&
accelerationInput > 0 && !isJumping)
            return;

        // Apply drag if there is no input
        if (accelerationInput == 0)
        {
            carRigidbody2D.drag = Mathf.Lerp(carRigidbody2D.drag, 3.0f,
Time.fixedDeltaTime * 3);
        }
        else
        {
            carRigidbody2D.drag = Mathf.Lerp(carRigidbody2D.drag, 0,
Time.fixedDeltaTime * 10);
        }

        // Apply a certain amount of drag depending on the surface being driven on
when not using a boost item
        if (!isBoosting)
        {
            switch (GetSurface())
            {
                case Surface.SurfaceTypes.Grass:
                    carRigidbody2D.drag = Mathf.Lerp(carRigidbody2D.drag, 10.0f,
Time.fixedDeltaTime * 3);
                    break;
                case Surface.SurfaceTypes.Sand:
                    carRigidbody2D.drag = Mathf.Lerp(carRigidbody2D.drag, 9.0f,
Time.fixedDeltaTime * 3);
                    break;

                case Surface.SurfaceTypes.Mud:
                    carRigidbody2D.drag = Mathf.Lerp(carRigidbody2D.drag, 10.0f,
Time.fixedDeltaTime * 3);
                    break;
            }
        }

        // Create a force for the engine
        Vector2 engineForceVector = transform.up * accelerationInput *
accelerationFactor;

        // Applies force and pushes the car forward
        carRigidbody2D.AddForce(engineForceVector, ForceMode2D.Force);

    }

    /**/
    /*
    CarController::ApplySteering() CarController::ApplySteering()

    NAME
        CarController::ApplySteering - Manages the steering of the car.

    SYNOPSIS
        void CarController::ApplySteering();

    DESCRIPTION
        This function handles the car's steering. It calculates the steering angle
based on user input
        and the current speed of the car. The function ensures that steering is
more effective at higher
        speeds and less pronounced when the car is moving slowly.

    RETURNS
        Nothing.
    */
    /**/
    void ApplySteering()
    {
        // Limit car's ability to turn when moving slowly
        float minSpeedBeforeTurn = (carRigidbody2D.velocity.magnitude / 8);
        minSpeedBeforeTurn = Mathf.Clamp01(minSpeedBeforeTurn);

        rotationAngle -= steeringInput * turnFactor * minSpeedBeforeTurn;

        carRigidbody2D.MoveRotation(rotationAngle);
    }

    /**/
    /*
    CarController::ReduceHorizontalVelocity()
CarController::ReduceHorizontalVelocity()

    NAME
        CarController::ReduceHorizontalVelocity - Reduces unwanted sideways
movement.

    SYNOPSIS
        void CarController::ReduceHorizontalVelocity();

    DESCRIPTION
        This function reduces the car's sideways velocity to prevent unrealistic
sliding during movement.
        It uses the car's current direction and drift factor to adjust the velocity
and maintain more realistic driving physics.

    RETURNS
        Nothing.
    */
    /**/
    void ReduceHorizontalVelocity()
    {
        Vector2 forwardVelocity = transform.up *
Vector2.Dot(carRigidbody2D.velocity, transform.up);
        Vector2 rightVelocity = transform.right *
Vector2.Dot(carRigidbody2D.velocity, transform.right);

        float currentDriftFactor = driftFactor;

        switch (GetSurface())
        {
            case Surface.SurfaceTypes.Grass:
                currentDriftFactor *= 1.05f;
                break;
        }

        carRigidbody2D.velocity = forwardVelocity + rightVelocity *
currentDriftFactor;
    }

    /**/
    /*
    CarController::GetHorizontalVelocity() CarController::GetHorizontalVelocity()

    NAME
        CarController::GetHorizontalVelocity - Retrieves the car's sideways
movement speed.

    SYNOPSIS
        float CarController::GetHorizontalVelocity();

    DESCRIPTION
        This function calculates and returns the sideways velocity of the car. It
determines the car's drifting behavior
        and is used in physics calculations related to car handling and tire
screeching.

    RETURNS
        The car's sideways velocity.
    */
    /**/
    float GetHorizontalVelocity()
    {
        return Vector2.Dot(transform.right, carRigidbody2D.velocity);
    }
    /**/
    /*
    CarController::IsTireDrifting(out float horizontalVelocity, out bool isBraking)
CarController::IsTireDrifting(out float horizontalVelocity, out bool isBraking)

    NAME
        CarController::IsTireDrifting - Checks if the car's tires are drifting.

    SYNOPSIS
        bool CarController::IsTireDrifting(out float horizontalVelocity, out bool
isBraking);
            horizontalVelocity   --> Out parameter to store the horizontal
velocity.
            isBraking         --> Out parameter to indicate if the car is braking.

    DESCRIPTION
        This function determines whether the car's tires are drifting based on its
horizontal velocity and
        braking status. It calculates if the car is screeching due to drifting or
braking, which is
        used for triggering audio or visual effects associated with tire
screeching.

    RETURNS
        True if the tires are drifting, False otherwise.
    */
    /**/
    public bool IsTireDrifting(out float horizontalVelocity, out bool isBraking)
    {
        horizontalVelocity = GetHorizontalVelocity();
        isBraking = false;

        if (isJumping)
            return false;

        // Checks if the player is moving forward and hitting the brakes.
        if (accelerationInput < 0 && velocityVsUp > 0)
        {
            isBraking = true;
            return true;
        }

        // If there is a lot of side movement, then the tires should be screeching
        if (Mathf.Abs(GetHorizontalVelocity()) > 4.0f)
            return true;

        return false;
    }

    /**/
    /*
    CarController::UpdateTopSpeed() CarController::UpdateTopSpeed()

    NAME
        CarController::UpdateTopSpeed - Updates the record of the car's top speed.

    SYNOPSIS
        void CarController::UpdateTopSpeed();

    DESCRIPTION
        This function updates the car's top speed record. It checks the current
speed and, if it's higher than
        the previously recorded top speed, updates the record.

    RETURNS
        Nothing.
    */
    /**/
    private void UpdateTopSpeed()
    {
        float currentSpeed = GetVelocityMagnitude();
        if (currentSpeed > topSpeed)
        {
            topSpeed = currentSpeed;
        }
    }

    /**/
    /*
    CarController::SetInputVector(Vector2 inputVector)
CarController::SetInputVector(Vector2 inputVector)

    NAME
        CarController::SetInputVector - Sets the input vector for car control.

    SYNOPSIS
        void CarController::SetInputVector(Vector2 inputVector);
            inputVector    --> The input vector for steering and acceleration.

    DESCRIPTION
        This method is used to set the steering and acceleration inputs for the car
based on player controls.
        The input vector contains values for both steering (x-axis) and
acceleration (y-axis), which are
        used to control the car's movement and direction.

    RETURNS
        Nothing.
    */
    /**/
    public void SetInputVector(Vector2 inputVector)
    {
        steeringInput = inputVector.x;
        accelerationInput = inputVector.y;
    }

    /**/
    /*
    CarController::GetVelocityMagnitude() CarController::GetVelocityMagnitude()

    NAME
        CarController::GetVelocityMagnitude - Retrieves the car's current velocity
magnitude.

    SYNOPSIS
        float CarController::GetVelocityMagnitude();

    DESCRIPTION
        This function calculates and returns the magnitude of the car's current
```

```
velocity.

    RETURNS
        The magnitude of the car's current velocity.
    */
    /**/
    public float GetVelocityMagnitude()
    {
        return carRigidbody2D.velocity.magnitude;
    }

    /**/
    /*
    CarController::GetSurface() CarController::GetSurface()

    NAME
        CarController::GetSurface - Retrieves the type of surface the car is
currently on.

    SYNOPSIS
        Surface.SurfaceTypes CarController::GetSurface();

    DESCRIPTION
        This method returns the type of surface that the car is currently driving
on, such as grass, sand, or mud.
        It is utilized to adjust the car's handling and physics according to
different surface types.

    RETURNS
        The type of surface the car is on.
    */
    /**/
    public Surface.SurfaceTypes GetSurface()
    {
        return surfaceHandler.GetCurrentSurface();
    }

    /**/
    /*
    CarController::Jump(float jumpHeightScale, float jumpPushScale)
```

```
CarController::Jump(float jumpHeightScale, float jumpPushScale)

    NAME
        CarController::Jump - Initiates a jump for the car with given parameters.

    SYNOPSIS
        void CarController::Jump(float jumpHeightScale, float jumpPushScale);
            jumpHeightScale   --> The scale of the jump height.
            jumpPushScale     --> The scale of the forward push during the jump.

    DESCRIPTION
        This function triggers a jumping mechanic for the car. It uses provided
scale factors to determine
        the height and forward momentum of the jump.

    RETURNS
        Nothing.
    */
    /**/
    public void Jump(float jumpHeightScale, float jumpPushScale)
    {
        if (!isJumping)
        {
            StartCoroutine(JumpCo(jumpHeightScale, jumpPushScale));
        }
    }

    /**/
    /*
    CarController::JumpCo(float jumpHeightScale, float jumpPushScale)
CarController::JumpCo(float jumpHeightScale, float jumpPushScale)

    NAME
        CarController::JumpCo - Coroutine for handling the car's jump behavior.

    SYNOPSIS
        IEnumerator CarController::JumpCo(float jumpHeightScale, float
jumpPushScale);
            jumpHeightScale   --> The scale of the jump height.
            jumpPushScale     --> The scale of the forward push during the jump.
    DESCRIPTION
        This coroutine manages the detailed behavior of the car's jump. It includes
the jump's animation,
        collision handling, and the effects of the jump on the car's physics. The
coroutine uses scale factors
        for height and push to vary the jump's characteristics.

    RETURNS
        IEnumerator that yields the execution at various points, such as waiting
for the jump to complete, and resumes afterwards.
    */
    /**/
    private IEnumerator JumpCo(float jumpHeightScale, float jumpPushScale)
    {
        isJumping = true;

        float jumpStartTime = Time.time;
        float jumpDuration = carRigidbody2D.velocity.magnitude * 0.05f;

        jumpHeightScale = jumpHeightScale * carRigidbody2D.velocity.magnitude *
0.05f;
        jumpHeightScale = Mathf.Clamp(jumpHeightScale, 0.0f, 1.0f);

        carCollider.enabled = false;

        carSFXHandler.PlayJumpSfx();

        carSpriteRenderer.sortingLayerName = "Midair";
        carShadowRenderer.sortingLayerName = "Midair";

        carRigidbody2D.AddForce(carRigidbody2D.velocity.normalized * jumpPushScale
* 0.1f, ForceMode2D.Impulse);

        // Update the sprite renders of the car and its shadow while jumping
        while (isJumping)
        {
            float jumpCompletedPercentage = (Time.time - jumpStartTime) /
jumpDuration;
            jumpCompletedPercentage = Mathf.Clamp01(jumpCompletedPercentage);
            carSpriteRenderer.transform.localScale = Vector3.one + Vector3.one *
jumpCurve.Evaluate(jumpCompletedPercentage) * jumpHeightScale;

            carShadowRenderer.transform.localScale =
carSpriteRenderer.transform.localScale * 0.75f;

            carShadowRenderer.transform.localPosition = new Vector3(1, -1, 0.0f) *
3 * jumpCurve.Evaluate(jumpCompletedPercentage) * jumpHeightScale;

            if (jumpCompletedPercentage == 1.0f)
                break;

            yield return null;
        }

        carCollider.enabled = false;

        // Do not check for collisions with triggers
        ContactFilter2D contactFilter2D = new ContactFilter2D();
        contactFilter2D.useTriggers = false;

        Collider2D[] hitResults = new Collider2D[2];

        int numberOfHitObjects = Physics2D.OverlapCircle(transform.position, 1.5f,
contactFilter2D, hitResults);

        carCollider.enabled = true;

        // Check if landing is ok based on whether no objects were hit
        if (numberOfHitObjects != 0)
        {
            isJumping = false;

            Jump(0.1f, 0.3f);
        }
        else
        {
            carSpriteRenderer.transform.localScale = Vector3.one;
            carShadowRenderer.transform.localPosition = Vector3.zero;
            carShadowRenderer.transform.localScale =
carSpriteRenderer.transform.localScale;

            carCollider.enabled = true;

            carSpriteRenderer.sortingLayerName = "Car";
            carShadowRenderer.sortingLayerName = "Car";

            isJumping = false;
        }

        carSpriteRenderer.transform.localScale = Vector3.one;

        carShadowRenderer.transform.localPosition = Vector3.zero;
        carShadowRenderer.transform.localScale =
carSpriteRenderer.transform.localScale;

        if (jumpHeightScale > 0.2f)
        {
            carSFXHandler.PlayLandingSfx();
        }

        carCollider.enabled = true;

        isJumping = false;
    }

    /**/
    /*
    CarController::BoostSpeed(SpeedBoost speedBoost)
CarController::BoostSpeed(SpeedBoost speedBoost)

    NAME
        CarController::BoostSpeed - Activates a speed boost for the car.

    SYNOPSIS
        void CarController::BoostSpeed(SpeedBoost speedBoost);
            speedBoost    --> The SpeedBoost object that contains boost parameters.
```



```
    DESCRIPTION
        This method initiates a temporary speed boost for the car. It uses
parameters from the provided
        SpeedBoost object to increase the car's maximum speed for a specified
duration.

    RETURNS
        Nothing.
    */
    /**/
    public void BoostSpeed(SpeedBoost speedBoost)
    {
        if (isBoosting) return;

        StartCoroutine(BoostCo(speedBoost));
    }

    /**/
    /*
    CarController::BoostCo(SpeedBoost speedBoost) CarController::BoostCo(SpeedBoost
speedBoost)

    NAME
        CarController::BoostCo - Coroutine for managing the speed boost effect.

    SYNOPSIS
        IEnumerator CarController::BoostCo(SpeedBoost speedBoost);
            speedBoost   --> The SpeedBoost object that contains boost parameters.

    DESCRIPTION
        This coroutine handles the duration and effects of a speed boost. It
temporarily increases the car's
        maximum speed and ensures that the boost lasts for the specified duration
before returning the speed
        to normal.

    RETURNS
        IEnumerator that allows the function to pause when waiting for the boost
duration to elapse and resumes afterwards.
    */
    /**/
    private IEnumerator BoostCo(SpeedBoost speedBoost)
    {
        isBoosting = true;
        maxSpeed = boostSpeed;

        yield return new WaitForSeconds(boostDuration);

        speedBoost.EndBoost(this);
        isBoosting = false;
    }

    /**/
    /*
    CarController::DropMudPuddle(float duration) CarController::DropMudPuddle(float
duration)

    NAME
        CarController::DropMudPuddle - Drops a mud puddle on the track.

    SYNOPSIS
        void CarController::DropMudPuddle(float duration);
            duration    --> Duration for which the mud puddle remains on the track.

    DESCRIPTION
        This function creates a mud puddle at the car's current position. The mud
puddle persists for a specified
        duration and can affect other cars' speed.

    RETURNS
        Nothing.
    */
    /**/
    public void DropMudPuddle(float duration)
    {
        GameObject mudPuddle = Instantiate(mudPuddlePrefab, transform.position,
Quaternion.identity);

        StartCoroutine(RemoveMudPuddleAfterTime(mudPuddle, duration));
    }
    /**/
    /*
    CarController::RemoveMudPuddleAfterTime(GameObject mudPuddle, float duration)
CarController::RemoveMudPuddleAfterTime(GameObject mudPuddle, float duration)

    NAME
        CarController::RemoveMudPuddleAfterTime - Removes a mud puddle after a specified duration.

    SYNOPSIS
        IEnumerator CarController::RemoveMudPuddleAfterTime(GameObject mudPuddle, float duration);
            mudPuddle   --> The mud puddle GameObject to be removed.
            duration    --> The duration after which the mud puddle will be removed.

    DESCRIPTION
        This coroutine waits for a specified duration before removing a mud puddle from the track.

    RETURNS
        IEnumerator that temporarily halts execution for the duration of the mud puddle's presence.
    */
    /**/
    private IEnumerator RemoveMudPuddleAfterTime(GameObject mudPuddle, float duration)
    {
        yield return new WaitForSeconds(duration);

        Destroy(mudPuddle);
    }

    /**/
    /*
    CarController::OnTriggerEnter2D(Collider2D collider2d)
CarController::OnTriggerEnter2D(Collider2D collider2d)

    NAME
```

```
        CarController::OnTriggerEnter2D - Handles trigger collision events.

    SYNOPSIS
        void CarController::OnTriggerEnter2D(Collider2D collider2d);
            collider2d    --> The Collider2D component of the object the car has collided with.

    DESCRIPTION
        This method is called when the car enters a trigger collider. When the car hits a jump trigger,
        it executes a jump with parameters defined by the trigger.

    RETURNS
        Nothing.
    */
    /**/
    void OnTriggerEnter2D(Collider2D collider2d)
    {
        if (collider2d.CompareTag("Jump"))
        {
            CarJumpData jumpData = collider2d.GetComponent<CarJumpData>();
            Jump(jumpData.jumpHeightScale, jumpData.jumpPushScale);
        }
    }
}
```

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

[CreateAssetMenu(fileName = "New Car Data", menuName = "Car Data", order = 51)]

/**/
/*
    CarData is a ScriptableObject class used to store data about cars in the game. It includes information
    such as a unique identifier for each car, a sprite for the UI representation, a prefab for the car object,
    and the cost of the car.
```

```
*/
/**/
public class CarData : ScriptableObject
{
    [SerializeField]
    private int carUniqueID = 0;

    [SerializeField]
    private Sprite carUISprite;

    [SerializeField]
    private GameObject carPrefab;

    [SerializeField]
    private int cost;

    /**/
    /*
    CarData::CarUniqueID CarData::CarUniqueID

    NAME
        CarData::CarUniqueID - Getter for the car's unique identifier.

    SYNOPSIS
        int CarUniqueID

    DESCRIPTION
        Provides read-only access to the car's unique identifier, which is used to
distinguish
        different cars within the game.

    RETURNS
        Int carUniqueID: The unique identifier of the car.
    */
    /**/
    public int CarUniqueID
    {
        get { return carUniqueID; }
    }
    /**/
    /*
    CarData::CarUISprite CarData::CarUISprite

    NAME
        CarData::CarUISprite - Getter for the car's UI sprite.

    SYNOPSIS
        Sprite CarUISprite

    DESCRIPTION
        Provides read-only access to the Sprite object representing the car in the
game's user interface.
        This sprite is utilized for displaying the car the menu.

    RETURNS
        Sprite carUISprite: The sprite used for the car's UI representation.
    */
    /**/
    public Sprite CarUISprite
    {
        get { return carUISprite; }
    }

    /**/
    /*
    CarData::CarPrefab CarData::CarPrefab

    NAME
        CarData::CarPrefab - Getter for the car's prefab.

    SYNOPSIS
        GameObject CarPrefab

    DESCRIPTION
        Provides read-only access to the GameObject prefab of the car, which is
instantiated in the game
        to create a playable version of the car.

    RETURNS
```

```
        GameObject carPrefab: The prefab representing the car in the game.
    */
    /**/
    public GameObject CarPrefab
    {
        get { return carPrefab; }
    }

    /**/
    /*
    CarData::Cost CarData::Cost

    NAME
        CarData::Cost - Getter for the cost of the car.

    SYNOPSIS
        int Cost

    DESCRIPTION
        Provides read-only access to the cost of the car, used in in-game transactions like purchasing cars.

    RETURNS
        Int cost: The cost of the car.
    */
    /**/
    public int Cost
    {
        get { return cost; }
    }
}
```

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

/**/
/*
    This class is responsible for managing player inputs and translating them into
```

```
actions for the car.
    It captures and processes inputs such as steering, acceleration, braking, and
the use of items.
    This class serves as the intermediary between the player's input devices and
the car's behavior,
    ensuring that player commands are accurately reflected in the game.
*/
/**/
public class CarInputHandler : MonoBehaviour
{
    CarController carController;

    /**/
    /*
    CarInputHandler::Awake() CarInputHandler::Awake()

    NAME
        CarInputHandler::Awake - Initializes the CarInputHandler.

    SYNOPSIS
        void CarInputHandler::Awake();

    DESCRIPTION
        This function initializes the CarController component attached to the same
GameObject.

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        carController = GetComponent<CarController>();
    }

    /**/
    /*
    CarInputHandler::Update() CarInputHandler::Update()

    NAME
```

```
        CarInputHandler::Update - Handles per-frame input processing.

    SYNOPSIS
        void CarInputHandler::Update();

    DESCRIPTION
        This method is called once per frame and handles the processing of player
inputs. It reads the
        horizontal and vertical axis inputs, and triggers the use of the car's
current item if the space
        key is pressed. The input is then passed to the car controller for handling
car movement.

    RETURNS
        Nothing.
    */
    /**/
    void Update()
    {
        Vector2 inputVector = Vector2.zero;

        inputVector.x = Input.GetAxis("Horizontal");
        inputVector.y = Input.GetAxis("Vertical");

        if (Input.GetKeyDown(KeyCode.Space) && carController.currentItem != null)
        {
            carController.currentItem.Use(carController);
            carController.currentItem = null;
        }

        carController.SetInputVector(inputVector);
    }
}
```

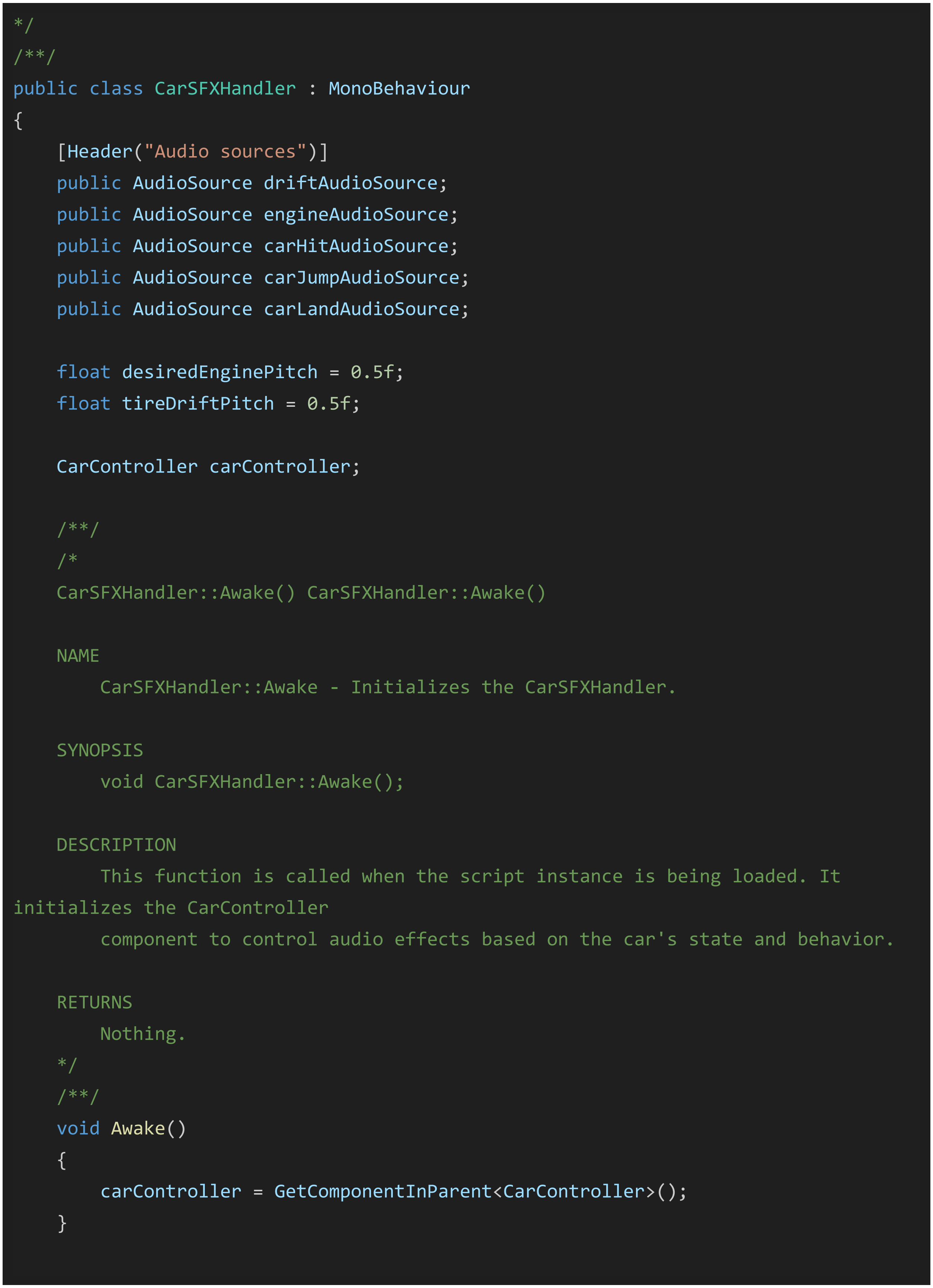
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/**/
```

```
/*
    This class is a simple data container that holds parameters for car jumping mechanics in the game.
    It defines the scale of jump height and push, which can be adjusted to customize the jumping behavior of cars.
*/
/**/
public class CarJumpData : MonoBehaviour
{
    [Header("Jump Settings")]

    public float jumpHeightScale = 1.0f;
    public float jumpPushScale = 1.0f;
}
```

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

/**/
/*
    This class manages the particle effects associated with a car's movement and actions in the game.
    It dynamically adjusts the rate and intensity of particle emissions based on the car's current state,
    such as drifting, braking, or driving on different surfaces.
*/
/**/
public class CarParticleHandler : MonoBehaviour
{
    float particleEmissionRate = 0;

    CarController carController;
    ParticleSystem particleSystemSmoke;
    ParticleSystem.EmissionModule particleEM;

    /**/
    /*
    CarParticleHandler::Awake() CarParticleHandler::Awake()
    NAME
        CarParticleHandler::Awake - Initializes the CarParticleHandler.

    SYNOPSIS
        void CarParticleHandler::Awake();

    DESCRIPTION
        This function initializes the CarController and ParticleSystem components.
It also retrieves and configures
        the emission module of the particle system to initially emit no particles.

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        carController = GetComponentInParent<CarController>();

        particleSystemSmoke = GetComponent<ParticleSystem>();

        // Get the emission component and set it to zero emission.
        particleEM = particleSystemSmoke.emission;
        particleEM.rateOverTime = 0;
    }

    /**/
    /*
    CarParticleHandler::Update() CarParticleHandler::Update()

    NAME
        CarParticleHandler::Update - Updates the particle effects each frame.

    SYNOPSIS
        void CarParticleHandler::Update();

    DESCRIPTION
        This method is called once per frame and handles the emission of particles
based on the car's movement.
```

```
        It gradually reduces the emission rate over time and increases it when the
car is drifting or braking.
        The emission rate is proportional to the car's horizontal velocity.

    RETURNS
        Nothing.
    */
    /**/
    void Update()
    {
        particleEmissionRate = Mathf.Lerp(particleEmissionRate, 0, Time.deltaTime *
5);
        particleEM.rateOverTime = particleEmissionRate;


        if (carController.IsTireDrifting(out float horizontalVelocity, out bool
isBraking))
        {
            if (isBraking)
                particleEmissionRate = 30;

            else particleEmissionRate = Mathf.Abs(horizontalVelocity) * 2;
        }
    }
}
```

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

/**/
/*
    This class manages the sound effects associated with a car's actions and
interactions. It controls audio cues
    for different states and behaviors of the car, such as engine sounds, tire
screeching during drifts, collision impacts,
    and jump landings. The class adjusts sound properties like volume and pitch
based on the car's dynamics.
```



```
*/
/**/
public class CarSFXHandler : MonoBehaviour
{
    [Header("Audio sources")]
    public AudioSource driftAudioSource;
    public AudioSource engineAudioSource;
    public AudioSource carHitAudioSource;
    public AudioSource carJumpAudioSource;
    public AudioSource carLandAudioSource;

    float desiredEnginePitch = 0.5f;
    float tireDriftPitch = 0.5f;

    CarController carController;

    /**/
    /*
    CarSFXHandler::Awake() CarSFXHandler::Awake()

    NAME
        CarSFXHandler::Awake - Initializes the CarSFXHandler.

    SYNOPSIS
        void CarSFXHandler::Awake();

    DESCRIPTION
        This function is called when the script instance is being loaded. It
initializes the CarController
        component to control audio effects based on the car's state and behavior.

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        carController = GetComponentInParent<CarController>();
    }
    /**/
    /*
    CarSFXHandler::Update() CarSFXHandler::Update()

    NAME
        CarSFXHandler::Update - Updates the car's sound effects each frame.

    SYNOPSIS
        void CarSFXHandler::Update();

    DESCRIPTION
        This method is called once per frame and handles the updating of the car's sound effects. It manages
        the engine sound effects and tire drift sounds based on the car's speed, drift status, and braking behavior.

    RETURNS
        Nothing.
    */
    /**/
    void Update()
    {
        UpdateEngineSFX();
        UpdateTireDriftSFX();
    }

    /**/
    /*
    CarSFXHandler::UpdateEngineSFX() CarSFXHandler::UpdateEngineSFX()

    NAME
        CarSFXHandler::UpdateEngineSFX - Manages the engine sound effects.

    SYNOPSIS
        void CarSFXHandler::UpdateEngineSFX();

    DESCRIPTION
        This method adjusts the engine sound effects based on the car's current velocity.
        It modifies both the volume and pitch of the engine sound to reflect
```

```
changes in the car's speed.

    RETURNS
        Nothing.
    */
    /**/
    void UpdateEngineSFX()
    {
        float velocityMagnitude = carController.GetVelocityMagnitude();

        // Increase the engine volume as the car goes faster
        float desiredEngineVolume = velocityMagnitude * 0.05f;

        desiredEngineVolume = Mathf.Clamp(desiredEngineVolume, 0.2f, 1.0f);

        engineAudioSource.volume = Mathf.Lerp(engineAudioSource.volume,
desiredEngineVolume, Time.deltaTime * 10);

        desiredEnginePitch = velocityMagnitude * 0.2f;
        desiredEnginePitch = Mathf.Clamp(desiredEnginePitch, 0.5f, 2f);
        engineAudioSource.pitch = Mathf.Lerp(engineAudioSource.pitch,
desiredEnginePitch, Time.deltaTime * 1.5f);
    }

    /**/
    /*
    CarSFXHandler::UpdateTireDriftSFX() CarSFXHandler::UpdateTireDriftSFX()

    NAME
        CarSFXHandler::UpdateTireDriftSFX - Manages the tire drifting sound
effects.

    SYNOPSIS
        void CarSFXHandler::UpdateTireDriftSFX();

    DESCRIPTION
        This method adjusts the tire drifting sound effects based on whether the
car is
        drifting or braking. It changes the volume and pitch of the drifting sounds
to reflect the
```

```
        intensity of the car's lateral movements.

    RETURNS
        Nothing.
    */
    /**/
    void UpdateTireDriftSFX()
    {
        // Handle tire screeching SFX
        if (carController.IsTireDrifting(out float lateralVelocity, out bool
isBraking))
        {
            // If the car is braking, then change the volume and pitch of the tire
screech
            if (isBraking)
            {
                driftAudioSource.volume = Mathf.Lerp(driftAudioSource.volume, 1.0f,
Time.deltaTime * 10);
                tireDriftPitch = Mathf.Lerp(tireDriftPitch, 0.5f, Time.deltaTime *
10);
            }
            else
            {
                driftAudioSource.volume = Mathf.Abs(lateralVelocity) * 0.05f;
                tireDriftPitch = Mathf.Abs(lateralVelocity) * 0.1f;
            }
        }
        // Fade out the tire screech SFX if we are not screeching
        else driftAudioSource.volume = Mathf.Lerp(driftAudioSource.volume, 0,
Time.deltaTime * 10);
    }

    /**/
    /*
    CarSFXHandler::OnCollisionEnter2D(Collision2D collision2D)
CarSFXHandler::OnCollisionEnter2D(Collision2D collision2D)

    NAME
        CarSFXHandler::OnCollisionEnter2D - Handles collision sound effects.
    SYNOPSIS
        void CarSFXHandler::OnCollisionEnter2D(Collision2D collision2D);
            collision2D    --> Collision data from the 2D physics engine.

    DESCRIPTION
        This method is triggered when the car enters a collision. It plays a sound
effect based on the
        intensity of the collision, adjusting the pitch and volume to reflect the
force of impact.

    RETURNS
        Nothing.
    */
    /**/
    void OnCollisionEnter2D(Collision2D collision2D)
    {
        float relativeVelocity = collision2D.relativeVelocity.magnitude;

        float volume = relativeVelocity * 0.1f;

        carHitAudioSource.pitch = Random.Range(0.95f, 1.05f);
        carHitAudioSource.volume = volume;

        if (!carHitAudioSource.isPlaying)
            carHitAudioSource.Play();
    }

    /**/
    /*
    CarSFXHandler::PlayJumpSfx() CarSFXHandler::PlayJumpSfx()

    NAME
        CarSFXHandler::PlayJumpSfx - Plays the jump sound effect.

    SYNOPSIS
        void CarSFXHandler::PlayJumpSfx();

    DESCRIPTION
        This method triggers the sound effect associated with the car's jump
action.
```

```
    RETURNS
        Nothing.
    */
    /**/
    public void PlayJumpSfx()
    {
        carJumpAudioSource.Play();
    }

    /**/
    /*
    CarSFXHandler::PlayLandingSfx() CarSFXHandler::PlayLandingSfx()

    NAME
        CarSFXHandler::PlayLandingSfx - Plays the sound effect associated with the car landing.

    SYNOPSIS
        void CarSFXHandler::PlayLandingSfx();

    DESCRIPTION
        This method triggers the sound effect for when the car lands after a jump.

    RETURNS
        Nothing.
    */
    /**/
    public void PlayLandingSfx()
    {
        carLandAudioSource.Play();
    }
}
```

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

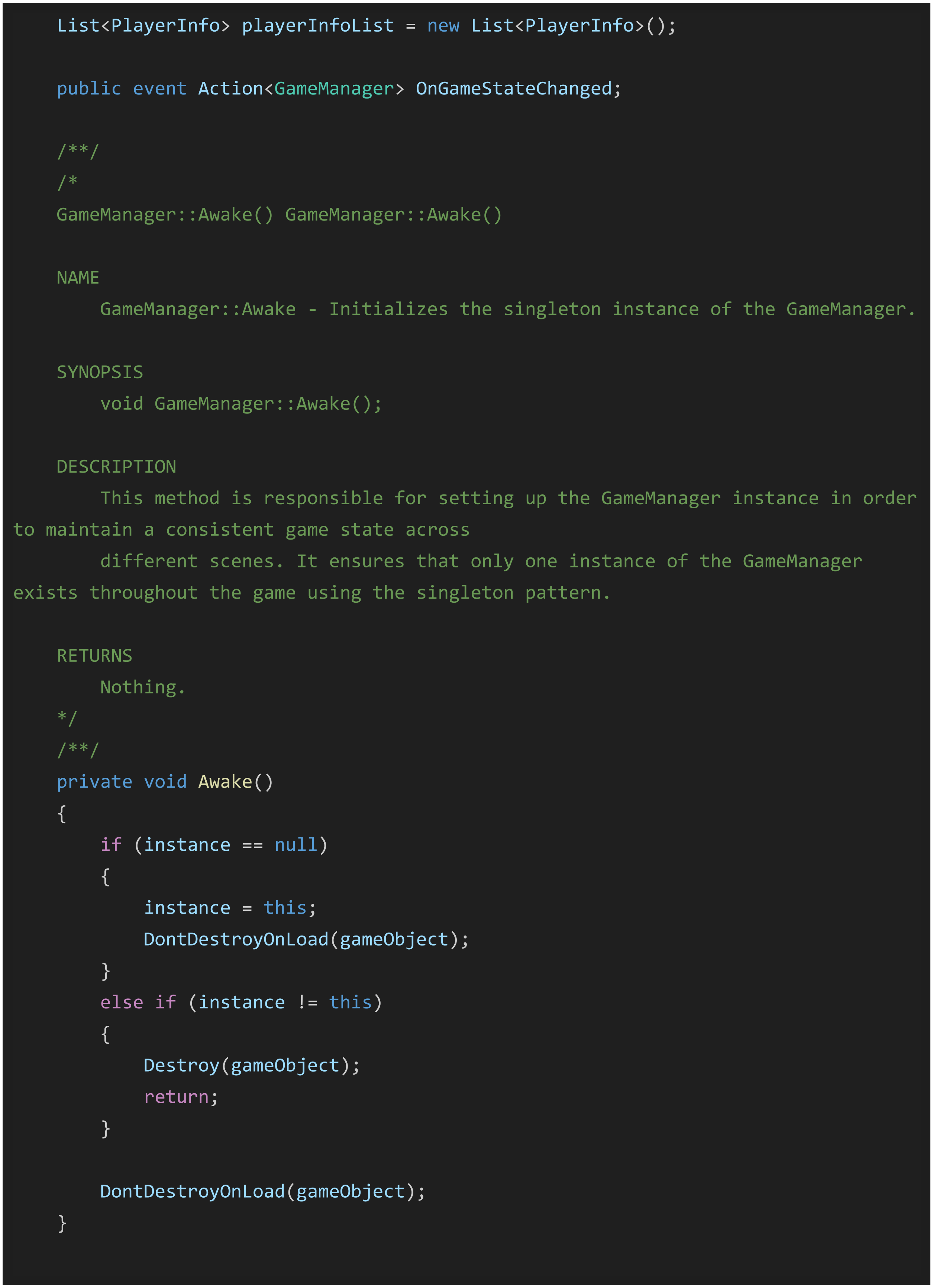
```
/**/
/*
    This class is responsible for initializing and spawning player cars at the
start of a race. It dynamically
    places cars at designated spawn points according to the players' choices and
game settings. The class handles
    the instantiation of car prefabs, aligning them with players' preferences and
assigning control mechanisms based on
    whether the player is an AI or a human.
*/
/**/
public class CarSpawns : MonoBehaviour
{
    /**/
    /*
    CarSpawns::Start() CarSpawns::Start()

    NAME
        CarSpawns::Start - Initializes and spawns player cars at the beginning of
the game.

    SYNOPSIS
        void CarSpawns::Start();

    DESCRIPTION
        This function locates all available spawn points and assigns cars to
players based on their selected preferences.
        It instantiates car prefabs at the spawn points, adjusting their settings
for AI or player control based on the GameManager's
        player information.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        GameObject[] spawnPoints = GameObject.FindGameObjectsWithTag("SpawnPoint");

        CarData[] allCarData = Resources.LoadAll<CarData>("CarData/");
        List<PlayerInfo> playerInfoList = new List<PlayerInfo>(GameManager.instance.GetPlayerList());

        for (int i = 0; i < spawnPoints.Length; i++)
        {
            Transform spawnPoint = spawnPoints[i].transform;

            if (playerInfoList.Count == 0)
            {
                return;
            }

            PlayerInfo playerInfo = playerInfoList[0];

            int selectedCarID = playerInfo.carUniqueID;

            foreach (CarData carData in allCarData)
            {
                if (carData.CarUniqueID == selectedCarID)
                {
                    GameObject car = Instantiate(carData.CarPrefab, spawnPoint.position, spawnPoint.rotation);

                    car.name = playerInfo.name;

                    if (playerInfo.isAI)
                    {
                        car.GetComponent<CarInputHandler>().enabled = false;
                        car.tag = "AI";
                    }
                    else
                    {
                        car.GetComponent<CarAIHandler>().enabled = false;
                        car.GetComponent<AStarPath>().enabled = false;
                        car.tag = "Player";
                    }

                    break;
                }
            }

            playerInfoList.Remove(playerInfo);
        }
    }
}
```

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

/**/
/*
    This class is responsible for managing the user interface elements related to
displaying car information in the game.
    It controls UI elements like car images and price tags, and manages animations
for car selection scenarios.
*/
/**/
public class CarUIHandler : MonoBehaviour
{
    [Header("Car Details")]
    public Image carImage;
    public TMP_Text carPriceText;

    Animator animator = null;

    /**/
    /*
    CarUIHandler::Awake() CarUIHandler::Awake()

    NAME
        CarUIHandler::Awake - Initializes the CarUIHandler component.

    SYNOPSIS
        void CarUIHandler::Awake();
    DESCRIPTION
        This method initializes the CarUIHandler by finding and storing the
Animator component within the child objects.
        The Animator is used for handling UI animations related to car selection.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        animator = GetComponentInChildren<Animator>();
    }

    /**/
    /*
    CarUIHandler::SetupCar(CarData carData, bool isCarPurchased)
CarUIHandler::SetupCar(CarData carData, bool isCarPurchased)

    NAME
        CarUIHandler::SetupCar - Configures the UI elements for a specific car.

    SYNOPSIS
        public void SetupCar(CarData carData, bool isCarPurchased);
            carData          --> The data object containing details about the car.
            isCarPurchased   --> Flag indicating whether the car is already
purchased.

    DESCRIPTION
        This method is used to configure the UI elements, such as car image and
price, based on the provided car data.
        It updates the UI to reflect whether the car is already purchased or still
available for purchase.

    RETURNS
        Nothing.
    */
    /**/
    public void SetupCar(CarData carData, bool isCarPurchased)
    {
        carImage.sprite = carData.CarUISprite;

        if (!isCarPurchased)
        {
            carPriceText.text = carData.Cost.ToString() + " Points";
            carPriceText.gameObject.SetActive(true);
        }
        else
        {
            carPriceText.gameObject.SetActive(false); // Hide price text for
purchased cars
        }
    }

    /**/
    /*
    CarUIHandler::StartCarEnterAnim(bool isEnterOnRight)
CarUIHandler::StartCarEnterAnim(bool isEnterOnRight)

    NAME
        CarUIHandler::StartCarEnterAnim - Starts the car entry animation.

    SYNOPSIS
        public void StartCarEnterAnim(bool isEnterOnRight);
            isEnterOnRight   --> Flag indicating the direction of the entry
animation.

    DESCRIPTION
        This method triggers the car's entry animation into the UI, based on the
specified direction.

    RETURNS
        Nothing.
    */
    /**/
    public void StartCarEnterAnim(bool isEnterOnRight)
    {
        if (isEnterOnRight)
        {
            animator.Play("Car UI Enter L");
        }
        else
        {
            animator.Play("Car UI Enter R");
        }
    }

    /**/
    /*
    CarUIHandler::StartCarExitAnim(bool isExitOnRight)
CarUIHandler::StartCarExitAnim(bool isExitOnRight)

    NAME
        CarUIHandler::StartCarExitAnim - Starts the car exit animation.

    SYNOPSIS
        public void StartCarExitAnim(bool isExitOnRight);
            isExitOnRight    --> Flag indicating the direction of the exit
animation.

    DESCRIPTION
        This method triggers the car's exit animation from the UI, based on the
specified direction.

    RETURNS
        Nothing.
    */
    /**/
    public void StartCarExitAnim(bool isExitOnRight)
    {
        if (isExitOnRight)
        {
            animator.Play("Car UI Exit L");
        }
        else
        {
            animator.Play("Car UI Exit R");
        }
    }
    /**/
    /*
    CarUIHandler::OnCarExitAnimComplete() CarUIHandler::OnCarExitAnimComplete()

    NAME
        CarUIHandler::OnCarExitAnimComplete - Handles the completion of the car exit animation.

    SYNOPSIS
        public void OnCarExitAnimComplete();

    DESCRIPTION
        This method is called when the car's exit animation is complete.

    RETURNS
        Nothing.
    */
    /**/
    public void OnCarExitAnimComplete()
    {
        Destroy(gameObject);
    }
}
```

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

/**/
/*
    The Checkpoint class represents a checkpoint or a finish line in the game environment. It holds essential
    information about each checkpoint, such as whether it is a finish line and its sequential number in the race course.
*/
/**/
public class Checkpoint : MonoBehaviour
{
    public bool isFinishLine = false;
    public int checkpointNum = 1;
}
```

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

/**/
/*
    This class is responsible for managing the visual countdown sequence. It 
controls a UI Text element
    to display a countdown sequence, indicating the start of the race.
*/
/**/
public class CountdownUIHandler : MonoBehaviour
{
    public Text countdownText;

    /**/
    /*
    CountdownUIHandler::Awake() CountdownUIHandler::Awake()

    NAME
        CountdownUIHandler::Awake - Initializes the CountdownUIHandler component.

    SYNOPSIS
        void CountdownUIHandler::Awake();

    DESCRIPTION
        This method initializes the CountdownUIHandler by setting the countdown 
text to an empty string.

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        countdownText.text = "";
    }

    /**/
    /*
    CountdownUIHandler::Start() CountdownUIHandler::Start()

    NAME
        CountdownUIHandler::Start - Begins the countdown sequence.

    SYNOPSIS
        void CountdownUIHandler::Start();

    DESCRIPTION
        This method starts a coroutine (CountdownCO) to manage the countdown
sequence for the race start.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        StartCoroutine(CountdownCO());
    }

    /**/
    /*
    CountdownUIHandler::CountdownCO() CountdownUIHandler::CountdownCO()

    NAME
        CountdownUIHandler::CountdownCO - Coroutine for handling the countdown
sequence.

    SYNOPSIS
        IEnumerator CountdownUIHandler::CountdownCO();

    DESCRIPTION
        This coroutine handles the countdown sequence for the start of the race.
After showing the countdown,
        it notifies the GameManager to start the race and then deactivates the countdown UI.

    RETURNS
        IEnumerator that temporarily halts execution for the duration of each part of the countdown.
    */
    /**/
    IEnumerator CountdownCO()
    {
        int counter = 3;
        yield return new WaitForSeconds(2.3f);

        while (true)
        {
            if (counter != 0)
            {
                countdownText.text = counter.ToString();
            }
            else
            {
                countdownText.text = "GO!";

                GameManager.instance.OnRaceStart();

                break;
            }

            counter--;
            yield return new WaitForSeconds(1);
        }

        yield return new WaitForSeconds(0.5f);

        gameObject.SetActive(false);
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using System;
using System.IO;
using System.Linq;

/**/
/*
    The GameManager class is a central component in managing the overall game state and player data across different levels
    and scenes in the game. It operates as a singleton, ensuring only one instance exists throughout the game's lifecycle.
    This class handles various aspects of gameplay, including tracking the game state, managing race timings, player scores,
    and AI difficulty levels, as well as keeping records of purchased cars and unlocked tracks.
    GameManager also provides interfaces for other game components to access and modify game-related data, such as player
    information, points, and game states.
*/
/**/
public enum GameStates { countdown, running, raceOver };

public class GameManager : MonoBehaviour
{
    public static GameManager instance = null;

    GameStates gameState = GameStates.countdown;

    float raceStartedTime = 0;
    float raceFinishedTime = 0;
    private float aiDifficulty = 1.0f;
    private int lastRaceScore = 0;
    public int totalPoints = 0;

    private HashSet<int> purchasedCarIDs = new HashSet<int>();

    private HashSet<int> unlockedTracks = new HashSet<int>();
```



```
    List<PlayerInfo> playerInfoList = new List<PlayerInfo>();

    public event Action<GameManager> OnGameStateChanged;

    /**/
    /*
    GameManager::Awake() GameManager::Awake()

    NAME
        GameManager::Awake - Initializes the singleton instance of the GameManager.

    SYNOPSIS
        void GameManager::Awake();

    DESCRIPTION
        This method is responsible for setting up the GameManager instance in order to maintain a consistent game state across
        different scenes. It ensures that only one instance of the GameManager exists throughout the game using the singleton pattern.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else if (instance != this)
        {
            Destroy(gameObject);
            return;
        }

        DontDestroyOnLoad(gameObject);
    }
    /**/
    /*
    GameManager::Start() GameManager::Start()

    NAME
        GameManager::Start - Initializes player information at the start.

    SYNOPSIS
        void GameManager::Start();

    DESCRIPTION
        This method initializes the player information list with default values for
testing and initial gameplay setup.
        This method is adapted to dynamically set player information based on game
settings

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        playerInfoList.Add(new PlayerInfo(1, "Player1", 0, false));
    }

    /**/
    /*
    GameManager::LevelStart() GameManager::LevelStart()

    NAME
        GameManager::LevelStart - Prepares the game state for a new level.

    SYNOPSIS
        void GameManager::LevelStart();

    DESCRIPTION
        This method is called to initialize the game state at the beginning of a
new level or race.
        It sets the game state to 'countdown', indicating the preparation phase
before the race begins.
```

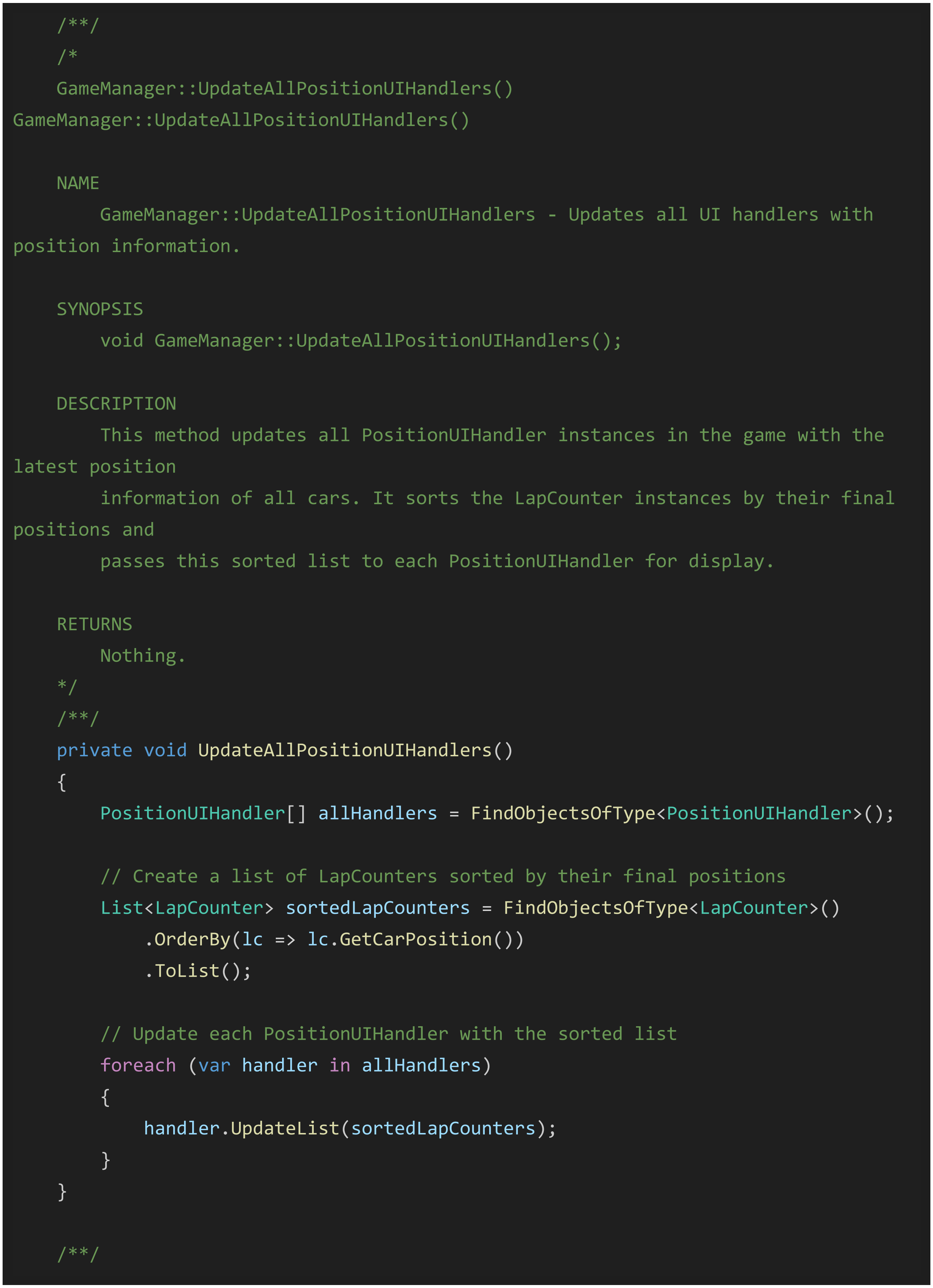
```
    RETURNS
        Nothing.
    */
    /**/
    void LevelStart()
    {
        gameState = GameStates.countdown;
    }

    /**/
    /*
    GameManager::GetGameState() GameManager::GetGameState()

    NAME
        GameManager::GetGameState - Retrieves the current game state.

    SYNOPSIS
        GameStates GameManager::GetGameState();

    DESCRIPTION
        This method returns the current state of the game, such as countdown, running, or race over.
        It is used throughout the game to determine the current phase of gameplay and to make decisions
        based on the game's progress.

    RETURNS
        The current state of the game.
    */
    /**/
    public GameStates GetGameState()
    {
        return gameState;
    }

    /**/
    /*
    GameManager::ChangeGameState(GameStates newGameState) GameManager::ChangeGameState(GameStates newGameState)
    NAME
        GameManager::ChangeGameState - Changes the game state to a new state.

    SYNOPSIS
        void GameManager::ChangeGameState(GameStates newGameState);
            newGameState   --> The new state to set the game to.

    DESCRIPTION
        This method changes the game's state to the specified new state. It
triggers the OnGameStateChanged event
        if the state has been changed.

    RETURNS
        Nothing.
    */
    /**/
    void ChangeGameState(GameStates newGameState)
    {
        if (gameState != newGameState)
        {
            gameState = newGameState;

            OnGameStateChanged?.Invoke(this);
        }
    }

    /**/
    /*
    GameManager::OnEnable() GameManager::OnEnable()

    NAME
        GameManager::OnEnable - Event registration when the script is enabled.

    SYNOPSIS
        void GameManager::OnEnable();

    DESCRIPTION
        This method registers the GameManager to listen for the 'sceneLoaded' event
from the SceneManager.
```

```
        It is used to respond to scene load events, allowing it to initialize or
reset game states appropriately.

    RETURNS
        Nothing.
    */
    /**/
    private void OnEnable()
    {
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

    /**/
    /*
    GameManager::AddPoints(int points) GameManager::AddPoints(int points)

    NAME
        GameManager::AddPoints - Adds points to the player's total score.

    SYNOPSIS
        void GameManager::AddPoints(int points);
            points   --> The number of points to add to the total score.

    DESCRIPTION
        This method increases the player's total points by the specified amount.

    RETURNS
        Nothing.
    */
    /**/
    public void AddPoints(int points)
    {
        totalPoints += points;
    }

    /**/
    /*
    GameManager::GetRaceTime() GameManager::GetRaceTime()

    NAME
```

```
        GameManager::GetRaceTime - Calculates the elapsed time of the current race.

    SYNOPSIS
        float GameManager::GetRaceTime();

    DESCRIPTION
        This method calculates and returns the elapsed time since the race started.
The time calculation
        depends on the current game state: during the countdown or race over, it
returns 0, and during the
        race, it calculates the time since the race started.

    RETURNS
        The elapsed time of the current race in seconds.
    */
    /**/
    public float GetRaceTime()
    {
        if (gameState == GameStates.countdown)
        {
            return 0;
        }
        else if (gameState == GameStates.raceOver)
        {
            return raceFinishedTime - raceStartedTime;
        }
        else
        {
            return Time.time - raceStartedTime;
        }
    }

    /**/
    /*
    GameManager::AIDifficulty GameManager::AIDifficulty

    NAME
        GameManager::AIDifficulty - Property for getting and setting AI difficulty.

    SYNOPSIS
```

```
        float GameManager::AIDifficulty

    DESCRIPTION
        This property allows getting and setting the difficulty level of AI players
in the game.
        The difficulty value is clamped between 0.0 (easiest) and 1.0 (hardest) to
ensure it remains
        within a valid range.

    RETURNS
        Float aiDifficulty: The AI difficulty level.
    */
    /**/
    public float AIDifficulty
    {
        get { return aiDifficulty; }
        set { aiDifficulty = Mathf.Clamp(value, 0.0f, 1.0f); }
    }

    /**/
    /*
    GameManager::AddPlayerToList(int playerNum, string name, int carUniqueID, bool
isAI) GameManager::AddPlayerToList(int playerNum, string name, int carUniqueID,
bool isAI)

    NAME
        GameManager::AddPlayerToList - Adds a player to the game's player list.

    SYNOPSIS
        void GameManager::AddPlayerToList(int playerNum, string name, int
carUniqueID, bool isAI);
            playerNum     --> The number identifying the player.
            name          --> The name of the player.
            carUniqueID   --> The unique ID of the player's chosen car.
            isAI          --> Indicates whether the player is an AI.

    DESCRIPTION
        This method adds a new player to the game's player list, with details such
as player number,
        name, selected car ID, and whether the player is an AI. This information is
```

```
used throughout the
        game to manage players' data and states.

    RETURNS
        Nothing.
    */
    /**/
    public void AddPlayerToList(int playerNum, string name, int carUniqueID, bool isAI)
    {
        playerInfoList.Add(new PlayerInfo(playerNum, name, carUniqueID, isAI));
    }

    /**/
    /*
    GameManager::ClearPlayerList() GameManager::ClearPlayerList()

    NAME
        GameManager::ClearPlayerList - Clears the list of player information.

    SYNOPSIS
        void GameManager::ClearPlayerList();

    DESCRIPTION
        This method clears all entries in the player information list. It's typically used during game
        initialization or when resetting the game state to ensure that outdated player data is removed
        before starting a new game or level.

    RETURNS
        Nothing.
    */
    /**/
    public void ClearPlayerList()
    {
        playerInfoList.Clear();
    }

    /**/
    /*
    GameManager::GetPlayerList() GameManager::GetPlayerList()

    NAME
        GameManager::GetPlayerList - Retrieves the list of player information.

    SYNOPSIS
        List<PlayerInfo> GameManager::GetPlayerList();

    DESCRIPTION
        This method returns the current list of players in the game, including
their details like player
        number, name, car ID, and AI status.

    RETURNS
        The list of players currently in the game.
    */
    /**/
    public List<PlayerInfo> GetPlayerList()
    {
        return playerInfoList;
    }

    /**/
    /*
    GameManager::OnRaceStart() GameManager::OnRaceStart()

    NAME
        GameManager::OnRaceStart - Handles the beginning of a race.

    SYNOPSIS
        void GameManager::OnRaceStart();

    DESCRIPTION
        This method is called to mark the start of a race. It records the start
time of the race and
        manages the transition to the 'running' game state.

    RETURNS
        Nothing.
    */
    /**/
    public void OnRaceStart()
    {
        raceStartedTime = Time.time;

        ChangeGameState(GameStates.running);
    }

    /**/
    /*
    GameManager::UpdatePlayerTopSpeed() GameManager::UpdatePlayerTopSpeed()

    NAME
        GameManager::UpdatePlayerTopSpeed - Updates the top speed record for the
player.

    SYNOPSIS
        void GameManager::UpdatePlayerTopSpeed();

    DESCRIPTION
        This method updates the top speed achieved by the player during the race.
It retrieves the
        CarController component of the player's car to access the top speed and
updates the corresponding
        player's information in the playerInfoList.

    RETURNS
        Nothing.
    */
    /**/
    public void UpdatePlayerTopSpeed()
    {
        CarController playerCarController =
GameObject.FindGameObjectWithTag("Player").GetComponent<CarController>();
        if (playerCarController != null)
        {
            PlayerInfo playerInfo = playerInfoList.Find(p => p.playerNum == 1); //
Assuming playerNum == 1 is the player
            if (playerInfo != null)
            {
                playerInfo.topSpeed = playerCarController.TopSpeed;
            }
        }
    }

    /**/
    /*
    GameManager::UpdatePlayerRacePosition(int playerNum, int position) GameManager::UpdatePlayerRacePosition(int playerNum, int position)

    NAME
        GameManager::UpdatePlayerRacePosition - Updates the race position of a specific player.

    SYNOPSIS
        void GameManager::UpdatePlayerRacePosition(int playerNum, int position);
            playerNum   --> The number identifying the player.
            position    --> The player's position in the race.

    DESCRIPTION
        This method updates the race position of a player identified by playerNum. It is called typically
        at the end of the race to record the player's final position.

    RETURNS
        Nothing.
    */
    /**/
    public void UpdatePlayerRacePosition(int playerNum, int position)
    {
        PlayerInfo playerInfo = playerInfoList.Find(p => p.playerNum == playerNum);
        if (playerInfo != null)
        {
            playerInfo.lastRacePosition = position;
        }
    }

    /**/
    /*
    GameManager::OnRaceFinish() GameManager::OnRaceFinish()

    NAME
        GameManager::OnRaceFinish - Handles the end of a race.

    SYNOPSIS
        void GameManager::OnRaceFinish();

    DESCRIPTION
        This method is called when a race is finished. It records the finish time,
updates the player's top
        speed, calculates and updates the player's race position, and changes the
game state to 'raceOver'.

    RETURNS
        Nothing.
    */
    /**/
    public void OnRaceFinish()
    {
        raceFinishedTime = Time.time;

        UpdatePlayerTopSpeed();

        LapCounter playerLapCounter =
FindObjectsOfType<LapCounter>().FirstOrDefault(lc => lc.CompareTag("Player"));
        if (playerLapCounter != null)
        {
            UpdatePlayerRacePosition(1, playerLapCounter.GetCarPosition());
        }

        lastRaceScore = CalculatePlayerScore();
        totalPoints += lastRaceScore;

        ChangeGameState(GameStates.raceOver);

        Debug.Log("Race Finished. Updating Position UI Handlers.");
        UpdateAllPositionUIHandlers();
    }
```



```
    /**/
    /*
    GameManager::UpdateAllPositionUIHandlers()
GameManager::UpdateAllPositionUIHandlers()

    NAME
        GameManager::UpdateAllPositionUIHandlers - Updates all UI handlers with
position information.

    SYNOPSIS
        void GameManager::UpdateAllPositionUIHandlers();

    DESCRIPTION
        This method updates all PositionUIHandler instances in the game with the
latest position
        information of all cars. It sorts the LapCounter instances by their final
positions and
        passes this sorted list to each PositionUIHandler for display.

    RETURNS
        Nothing.
    */
    /**/
    private void UpdateAllPositionUIHandlers()
    {
        PositionUIHandler[] allHandlers = FindObjectsOfType<PositionUIHandler>();

        // Create a list of LapCounters sorted by their final positions
        List<LapCounter> sortedLapCounters = FindObjectsOfType<LapCounter>()
            .OrderBy(lc => lc.GetCarPosition())
            .ToList();

        // Update each PositionUIHandler with the sorted list
        foreach (var handler in allHandlers)
        {
            handler.UpdateList(sortedLapCounters);
        }
    }

    /**/
    /*
    GameManager::GetLastRaceScore() GameManager::GetLastRaceScore()

    NAME
        GameManager::GetLastRaceScore - Retrieves the score from the last completed race.

    SYNOPSIS
        int GameManager::GetLastRaceScore();

    DESCRIPTION
        This method returns the score achieved by the player in the most recently completed race.
        It is used for displaying post-race results and for any calculations that depend on the
        player's performance in the last race.

    RETURNS
        Int lastRaceScore: The score from the last race.
    */
    /**/
    public int GetLastRaceScore()
    {
        return lastRaceScore;
    }

    /**/
    /*
    GameManager::CalculatePlayerScore() GameManager::CalculatePlayerScore()

    NAME
        GameManager::CalculatePlayerScore - Calculates the player's score based on race performance.

    SYNOPSIS
        private int GameManager::CalculatePlayerScore();

    DESCRIPTION
        This private method calculates the player's score based on various factors like race time,
        position, and top speed. It is typically called at the end of a race to determine the
        player's score for that race, which can then be used for updating the total points.

    RETURNS
        The calculated score for the player.
    */
    /**/
    private int CalculatePlayerScore()
    {
        // Retrieve the player's info
        PlayerInfo playerInfo = playerInfoList.Find(p => p.playerNum == 1);
        Debug.Log("Player's final position: " + playerInfo.lastRacePosition);

        float raceTime = GetRaceTime();
        int timeScore = Mathf.Max(0, 100 - (int)raceTime);
        int positionScore = (8 - playerInfo.lastRacePosition) * 10;
        int speedScore = (int)playerInfo.topSpeed * 1;

        Debug.Log($"Time Score: {timeScore}, Position Score: {positionScore}, Speed Score: {speedScore}");

        return timeScore + positionScore + speedScore;
    }

    /**/
    /*
    GameManager::MarkCarAsPurchased(int carID) GameManager::MarkCarAsPurchased(int carID)

    NAME
        GameManager::MarkCarAsPurchased - Records the purchase of a car.

    SYNOPSIS
        void GameManager::MarkCarAsPurchased(int carID);
            carID   --> The unique identifier of the car that was purchased.

    DESCRIPTION
        This method marks a car as purchased by adding its unique identifier to the
```

```
purchasedCarIDs set.
        It is used to track which cars have been purchased by the player.

    RETURNS
        Nothing.
    */
    /**/
    public void MarkCarAsPurchased(int carID)
    {
        purchasedCarIDs.Add(carID);
    }

    /**/
    /*
    GameManager::IsCarPurchased(int carID) GameManager::IsCarPurchased(int carID)

    NAME
        GameManager::IsCarPurchased - Checks if a car has been purchased.

    SYNOPSIS
        bool GameManager::IsCarPurchased(int carID);
            carID   --> The unique identifier of the car.

    DESCRIPTION
        This method checks if a car, identified by its unique ID, has been
purchased by the player.
        It returns a boolean value indicating whether the car is in the set of
purchasedCarIDs.

    RETURNS
        True if the car has been purchased, False otherwise.
    */
    /**/
    public bool IsCarPurchased(int carID)
    {
        return purchasedCarIDs.Contains(carID);
    }

    /**/
    /*
    GameManager::IsTrackUnlocked(int trackID) GameManager::IsTrackUnlocked(int
trackID)

    NAME
        GameManager::IsTrackUnlocked - Checks if a track is unlocked.

    SYNOPSIS
        bool GameManager::IsTrackUnlocked(int trackID);
            trackID   --> The unique identifier of the track.

    DESCRIPTION
        This method checks whether a particular track, identified by its unique ID,
has been unlocked
        by the player. It returns a boolean value indicating the unlocked status of
the track.

    RETURNS
        True if the track is unlocked, False otherwise.
    */
    /**/
    public bool IsTrackUnlocked(int trackID)
    {
        return unlockedTracks.Contains(trackID);
    }

    /**/
    /*
    GameManager::UnlockTrack(int trackID) GameManager::UnlockTrack(int trackID)

    NAME
        GameManager::UnlockTrack - Unlocks a track for the player.

    SYNOPSIS
        void GameManager::UnlockTrack(int trackID);
            trackID   --> The unique identifier of the track to be unlocked.

    DESCRIPTION
        This method unlocks a track, identified by its unique ID, for the player.
It adds the track ID
        to the set of unlockedTracks, and deducts points from the player's total as
```

```
a cost for unlocking it.

    RETURNS
        Nothing.
    */
    /**/
    public void UnlockTrack(int trackID)
    {
        if (!unlockedTracks.Contains(trackID))
        {
            unlockedTracks.Add(trackID);
            totalPoints -= 100;
        }
    }

    /**/
    /*
    GameManager::OnSceneLoaded(Scene scene, LoadSceneMode mode)
GameManager::OnSceneLoaded(Scene scene, LoadSceneMode mode)

    NAME
        GameManager::OnSceneLoaded - Responds to scene loading events.

    SYNOPSIS
        void GameManager::OnSceneLoaded(Scene scene, LoadSceneMode mode);
            scene   --> The loaded scene.
            mode    --> The mode in which the scene was loaded.

    DESCRIPTION
        This method is used to trigger the LevelStart method, ensuring that the
game state is set correctly when a new race begins.

    RETURNS
        Nothing.
    */
    /**/
    void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        LevelStart();
    }
```

```
}
```

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

/**/
/*
    This class represents item boxes in the game, which players can interact with to receive race items like
    speed boosts or obstacles. This class manages the item box's behavior, including its interactions with cars, the
    random selection of items, triggering animations, and handling the item box's destruction and respawn. When a car
    collides with an item box, the box grants a random item to the car and then initiates a destruction animation.
    The class also coordinates with the ItemBoxSpawn to manage the respawn of item boxes after they are destroyed.
*/
/**/
public class ItemBox : MonoBehaviour
{
    private Animator animator;
    private bool isDestroyed = false;
    public float destructionDelay = 2f;

    /**/
    /*
    ItemBox::Awake() ItemBox::Awake()

    NAME
        ItemBox::Awake - Initializes the ItemBox component.

    SYNOPSIS
        void ItemBox::Awake();

    DESCRIPTION
        This method initializes the Animator component which is used to handle animations for the item box.
    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        animator = GetComponent<Animator>();
    }

    /**/
    /*
    ItemBox::OnTriggerEnter2D(Collider2D collider)
ItemBox::OnTriggerEnter2D(Collider2D collider)

    NAME
        ItemBox::OnTriggerEnter2D - Handles the interaction when an object enters
its trigger.

    SYNOPSIS
        void ItemBox::OnTriggerEnter2D(Collider2D collider);
            collider   --> The collider of the object that entered the trigger.

    DESCRIPTION
        This method is triggered when an object enters the item box's trigger
collider. If the item box
        is not already destroyed, it checks if the collider belongs to a
CarController. If so, it assigns
        a random race item to the car, triggers the destruction animation, and
notifies the ItemBoxSpawn
        for a respawn if necessary.

    RETURNS
        Nothing.
    */
    /**/
    void OnTriggerEnter2D(Collider2D collider)
    {
        if (!isDestroyed)
        {
            CarController carController = 
collider.GetComponentInParent<CarController>();
            if (carController != null)
            {
                RaceItem item = GetRandomItem();
                carController.AssignItem(item);
                TriggerDestructionAnimation();

                ItemBoxSpawn spawner = GetComponentInParent<ItemBoxSpawn>();
                if (spawner != null)
                {
                    spawner.RespawnItemBox();
                }
            }
        }
    }

    /**/
    /*
    ItemBox::TriggerDestructionAnimation() ItemBox::TriggerDestructionAnimation()

    NAME
        ItemBox::TriggerDestructionAnimation - Triggers the destruction animation 
of the item box.

    SYNOPSIS
        void ItemBox::TriggerDestructionAnimation();

    DESCRIPTION
        This method is responsible for triggering the destruction animation of the 
item box. It marks
        the item box as destroyed to prevent further interactions and starts a 
coroutine to destroy the
        item box game object after a delay, allowing the animation to complete.

    RETURNS
        Nothing.
    */
    /**/
    void TriggerDestructionAnimation()
    {
        isDestroyed = true;
        animator.SetTrigger("Destroy");
        StartCoroutine(DestroyAfterDelay());
    }

    /**/
    /*
    ItemBox::DestroyAfterDelay() ItemBox::DestroyAfterDelay()

    NAME
        ItemBox::DestroyAfterDelay - Coroutine to destroy the item box after a
delay.

    SYNOPSIS
        IEnumerator ItemBox::DestroyAfterDelay();

    DESCRIPTION
        This coroutine waits for a specified delay, then destroys the item box game
object. This delay
        allows the destruction animation to complete before the object is removed
from the scene.

    RETURNS
        IEnumerator that allows the function to pause when waiting for the duration
of destructionDelay to elapse and resumes afterwards.
    */
    /**/
    IEnumerator DestroyAfterDelay()
    {
        yield return new WaitForSeconds(destructionDelay);
        Destroy(gameObject);
    }

    /**/
    /*
    ItemBox::GetRandomItem() ItemBox::GetRandomItem()

    NAME
        ItemBox::GetRandomItem - Selects a random race item.
    SYNOPSIS
        RaceItem ItemBox::GetRandomItem();

    DESCRIPTION
        This method randomly selects and returns a race item from a predefined set
of items.

    RETURNS
        The randomly selected race item.
    */
    /**/
    RaceItem GetRandomItem()
    {
        int randomIndex = Random.Range(0, 2);
        switch (randomIndex)
        {
            case 0:
                return new SpeedBoost();
            case 1:
                return new MudPuddle();
            default:
                return new SpeedBoost();
        }
    }
}
```

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

/**/
/*
    This class handles the spawning and respawning of item boxes in the game. It
controls the placement and timing for
    the appearance of item boxes, which provide players with race items. This class
uses a prefab for the item box,
    and can initiate a respawn after a set delay, allowing for consistent and timed
distribution of item boxes
```

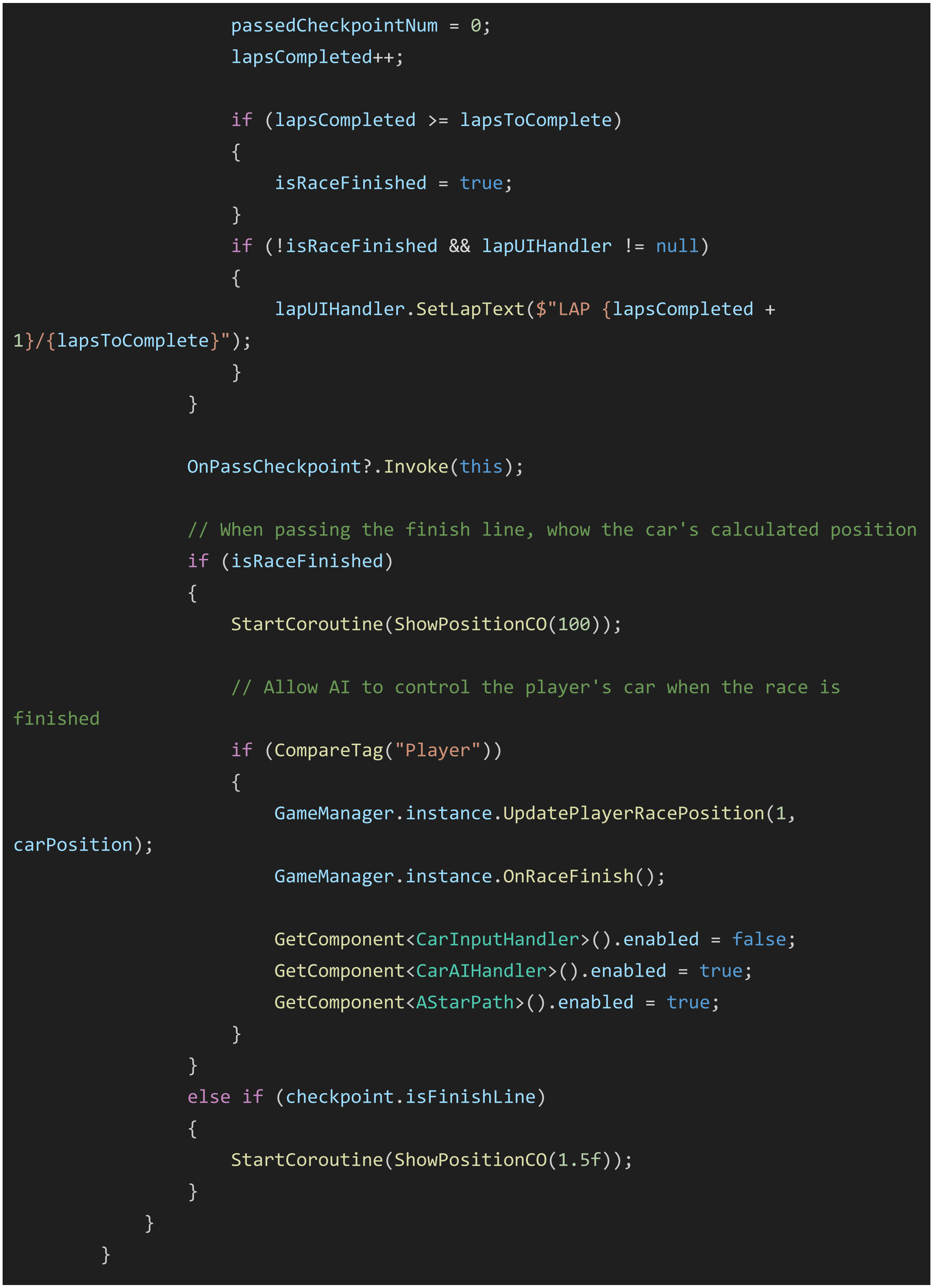
```
    throughout the race.
*/
/**/
public class ItemBoxSpawn : MonoBehaviour
{
    public GameObject itemBoxPrefab;
    public float respawnTime = 5f;

    /**/
    /*
    ItemBoxSpawn::Start() ItemBoxSpawn::Start()

    NAME
        ItemBoxSpawn::Start - Initializes the item box spawn.

    SYNOPSIS
        void ItemBoxSpawn::Start();

    DESCRIPTION
        This method calls the SpawnItemBox method to initially spawn an item box at the location of this GameObject.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        SpawnItemBox();
    }

    /**/
    /*
    ItemBoxSpawn::SpawnItemBox() ItemBoxSpawn::SpawnItemBox()

    NAME
        ItemBoxSpawn::SpawnItemBox - Spawns an item box.

    SYNOPSIS
        void ItemBoxSpawn::SpawnItemBox();
    DESCRIPTION
        This method instantiates an item box at the spawn location. It is used both for initial item box
        spawning and for respawning item boxes after they have been destroyed.

    RETURNS
        Nothing.
    */
    /**/
    void SpawnItemBox()
    {
        Instantiate(itemBoxPrefab, transform.position, transform.rotation, transform);
    }

    /**/
    /*
    ItemBoxSpawn::RespawnItemBox() ItemBoxSpawn::RespawnItemBox()

    NAME
        ItemBoxSpawn::RespawnItemBox - Initiates the respawn of an item box.

    SYNOPSIS
        void ItemBoxSpawn::RespawnItemBox();

    DESCRIPTION
        This method starts the RespawnCoroutine to respawn an item box after a defined delay.

    RETURNS
        Nothing.
    */
    /**/
    public void RespawnItemBox()
    {
        StartCoroutine(RespawnCoroutine());
    }

    /**/
    /*
    ItemBoxSpawn::RespawnCoroutine() ItemBoxSpawn::RespawnCoroutine()

    NAME
        ItemBoxSpawn::RespawnCoroutine - Coroutine for respawning an item box.

    SYNOPSIS
        IEnumerator ItemBoxSpawn::RespawnCoroutine();

    DESCRIPTION
        This coroutine waits for a specified amount of time defined by respawnTime,
then calls
        SpawnItemBox to respawn an item box.

    RETURNS
        IEnumerator that temporarily halts execution for the duration of
respawnTime.
    */
    /**/
    private IEnumerator RespawnCoroutine()
    {
        yield return new WaitForSeconds(respawnTime);
        SpawnItemBox();
    }
}
```

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

/**/
/*
    This class is responsible for tracking the progress of cars in the game through
laps and checkpoints.
    It keeps count of the number of laps completed, the checkpoints passed, and the
time at each checkpoint.
    This class also manages the logic for determining when a race is finished and
```

```
updates the UI to reflect
    the car's current position in the race. It also coordinates with the
GameManager to signal the end of
    the race and manage post-race actions for player-controlled cars.
*/
/**/
public class LapCounter : MonoBehaviour
{
    int passedCheckpointNum = 0;
    float timeAtLastPassedCheckpoint = 0;

    int numOfPassedCheckpoints = 0;

    int lapsCompleted = 0;
    const int lapsToComplete = 4;

    bool isRaceFinished = false;

    int carPosition = 0;

    public Text carPositionText;

    bool isHideRoutineRunning = false;
    float hideUIDelayTime;

    LapUIHandler lapUIHandler;

    public event Action<LapCounter> OnPassCheckpoint;

    /**/
    /*
    LapCounter::Start() LapCounter::Start()

    NAME
        LapCounter::Start - Initialization of the LapCounter component.

    SYNOPSIS
        void LapCounter::Start();

    DESCRIPTION
```

```
        This method initializes the LapCounter for the player by setting up the lap
UI text and preparing other relevant lap information.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        if (CompareTag("Player"))
        {
            lapUIHandler = FindObjectOfType<LapUIHandler>();
            lapUIHandler.SetLapText($"LAP {lapsCompleted + 1}/{lapsToComplete}");
        }
    }

    /**/
    /*
    LapCounter::SetCarPosition(int position) LapCounter::SetCarPosition(int
position)

    NAME
        LapCounter::SetCarPosition - Sets the position of the car in the race.

    SYNOPSIS
        void LapCounter::SetCarPosition(int position);
            position   --> The race position of the car.

    DESCRIPTION
        This method sets the race position of the car.

    RETURNS
        Nothing.
    */
    /**/
    public void SetCarPosition(int position)
    {
        carPosition = position;
    }
    /**/
    /*
    LapCounter::GetCarPosition() LapCounter::GetCarPosition()

    NAME
        LapCounter::GetCarPosition - Retrieves the race position of the car.

    SYNOPSIS
        int LapCounter::GetCarPosition();

    DESCRIPTION
        This method returns the current race position of the car.

    RETURNS
        Int carPosition: The current race position of the car.
    */
    /**/
    public int GetCarPosition()
    {
        return carPosition;
    }

    /**/
    /*
    LapCounter::GetNumberOfCheckpointsPassed() LapCounter::GetNumberOfCheckpointsPassed()

    NAME
        LapCounter::GetNumberOfCheckpointsPassed - Retrieves the number of checkpoints passed.

    SYNOPSIS
        int LapCounter::GetNumberOfCheckpointsPassed();

    DESCRIPTION
        This method returns the total number of checkpoints passed by the car during the race.

    RETURNS
        Int numOfPassedCheckpoints: The total number of checkpoints passed.
    */
    /**/
    public int GetNumberOfCheckpointsPassed()
    {
        return numOfPassedCheckpoints;
    }

    /**/
    /*
    LapCounter::GetTimeAtLastCheckPoint() LapCounter::GetTimeAtLastCheckPoint()

    NAME
        LapCounter::GetTimeAtLastCheckPoint - Retrieves the time at the last passed
checkpoint.

    SYNOPSIS
        float LapCounter::GetTimeAtLastCheckPoint();

    DESCRIPTION
        This method returns the time (in seconds) recorded when the car last passed
a checkpoint.

    RETURNS
        Float timeAtLastPassedCheckpoint: The time at the last checkpoint the car
passed.
    */
    /**/
    public float GetTimeAtLastCheckPoint()
    {
        return timeAtLastPassedCheckpoint;
    }

    /**/
    /*
    LapCounter::ShowPositionCO(float delayUntilHidePosition)
LapCounter::ShowPositionCO(float delayUntilHidePosition)

    NAME
        LapCounter::ShowPositionCO - Coroutine to display and hide car position UI.
    SYNOPSIS
        IEnumerator LapCounter::ShowPositionCO(float delayUntilHidePosition);
            delayUntilHidePosition   --> The delay in seconds before hiding the
position UI.

    DESCRIPTION
        This coroutine displays the car's current race position on the UI and hides
it after a specified delay.
        It is used to provide the player with feedback about their current position
during the race.

    RETURNS
        IEnumerator that temporarily halts execution for the duration of
hideUIDelayTime.
    */
    /**/
    IEnumerator ShowPositionCO(float delayUntilHidePosition)
    {
        hideUIDelayTime += delayUntilHidePosition;

        if (carPositionText == null) yield break;

        carPositionText.text = carPosition.ToString();

        carPositionText.gameObject.SetActive(true);

        if (!isHideRoutineRunning)
        {
            isHideRoutineRunning = true;

            yield return new WaitForSeconds(hideUIDelayTime);
            carPositionText.gameObject.SetActive(false);

            isHideRoutineRunning = false;
        }
    }

    /**/
    /*
    LapCounter::OnTriggerEnter2D(Collider2D collider2D)
```

```
LapCounter::OnTriggerEnter2D(Collider2D collider2D)

    NAME
        LapCounter::OnTriggerEnter2D - Handles the car passing through a
checkpoint.

    SYNOPSIS
        void LapCounter::OnTriggerEnter2D(Collider2D collider2D);
            collider2D   --> The collider of the checkpoint.

    DESCRIPTION
        This method is called when the car enters the trigger collider of a
checkpoint or the finish line.
        It updates the lap and checkpoint information, triggers UI updates, and
handles race completion logic.

    RETURNS
        Nothing.
    */
    /**/
    private void OnTriggerEnter2D(Collider2D collider2D)
    {
        if (collider2D.CompareTag("Checkpoint"))
        {
            if (isRaceFinished)
                return;

            Checkpoint checkpoint = collider2D.GetComponent<Checkpoint>();

            // Checks that the car is passing the checkpoints in the correct order
            if (passedCheckpointNum + 1 == checkpoint.checkpointNum)
            {
                passedCheckpointNum = checkpoint.checkpointNum;

                numOfPassedCheckpoints++;

                timeAtLastPassedCheckpoint = Time.time;

                if (checkpoint.isFinishLine)
                {
```



```
                    passedCheckpointNum = 0;
                    lapsCompleted++;

                    if (lapsCompleted >= lapsToComplete)
                    {
                        isRaceFinished = true;
                    }
                    if (!isRaceFinished && lapUIHandler != null)
                    {
                        lapUIHandler.SetLapText($"LAP {lapsCompleted +
1}/{lapsToComplete}");
                    }
                }

                OnPassCheckpoint?.Invoke(this);

                // When passing the finish line, whow the car's calculated position
                if (isRaceFinished)
                {
                    StartCoroutine(ShowPositionCO(100));

                    // Allow AI to control the player's car when the race is
finished
                    if (CompareTag("Player"))
                    {
                        GameManager.instance.UpdatePlayerRacePosition(1,
carPosition);

                        GameManager.instance.OnRaceFinish();

                        GetComponent<CarInputHandler>().enabled = false;
                        GetComponent<CarAIHandler>().enabled = true;
                        GetComponent<AStarPath>().enabled = true;
                    }
                }
                else if (checkpoint.isFinishLine)
                {
                    StartCoroutine(ShowPositionCO(1.5f));
                }
            }
        }
```

```
    }
}
```

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

/**/
/*
    This class is responsible for managing and updating the lap information displayed on the game's user interface.
    It controls a Text element within the UI to show the current lap number and the total number of laps in the race.
*/
/**/
public class LapUIHandler : MonoBehaviour
{
    Text lapText;

    /**/
    /*
    LapUIHandler::Awake() LapUIHandler::Awake()

    NAME
        LapUIHandler::Awake - Initializes the LapUIHandler component.

    SYNOPSIS
        void LapUIHandler::Awake();

    DESCRIPTION
        This method initializes the LapUIHandler by finding and storing the Text component attached to the same GameObject.
        This Text component is used to display lap information to the player during the race.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        lapText = GetComponent<Text>();
    }

    /**/
    /*
    LapUIHandler::SetLapText(string text) LapUIHandler::SetLapText(string text)

    NAME
        LapUIHandler::SetLapText - Updates the lap text display.

    SYNOPSIS
        public void SetLapText(string text);
            text    --> The string to be displayed in the lap text UI.

    DESCRIPTION
        This method updates the lap text UI with the provided string. It's called
to display the current lap
        and the total number of laps.

    RETURNS
        Nothing.
    */
    /**/
    public void SetLapText(string text)
    {
        lapText.text = text;
    }
}
```

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

/**/
/*
```

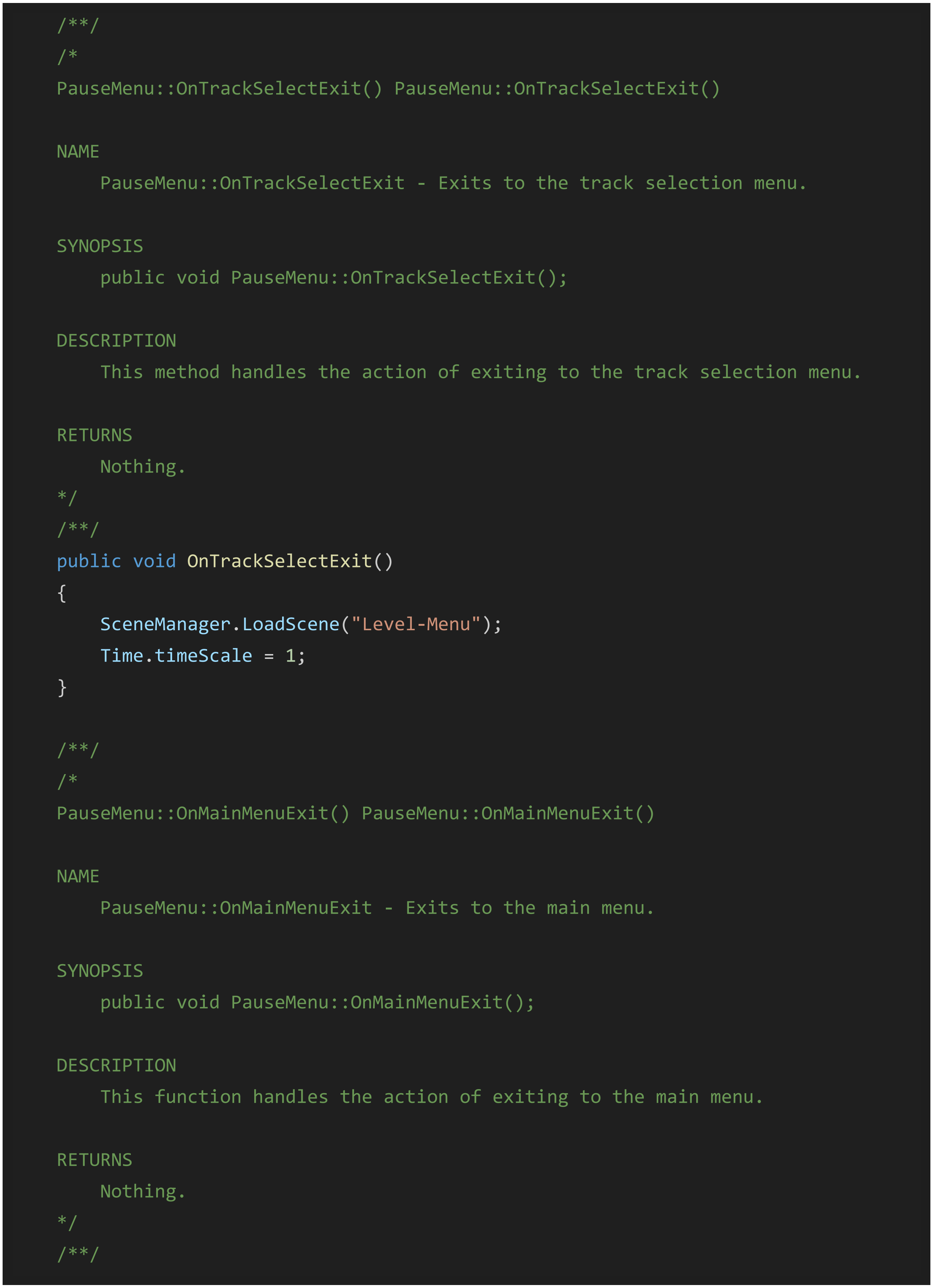
```
    This class is responsible for handling user interactions in the main menu of
the game.
    It provides functionality for navigating to different parts of the game such as
the car selection menu,
    options menu, and the functionality to exit the game.
*/
/**/
public class MainMenu : MonoBehaviour
{
    /**/
    /*
    MainMenu::PlayGame() MainMenu::PlayGame()

    NAME
        MainMenu::PlayGame - Loads the car selection menu.

    SYNOPSIS
        public void MainMenu::PlayGame();

    DESCRIPTION
        This function is triggered by a UI event to load the car selection menu.

    RETURNS
        Nothing.
    */
    /**/
    public void PlayGame()
    {
        SceneManager.LoadScene("Select-Menu");
    }

    /**/
    /*
    MainMenu::SelectOptions() MainMenu::SelectOptions()

    NAME
        MainMenu::SelectOptions - Loads the options menu.

    SYNOPSIS
        public void MainMenu::SelectOptions();
    DESCRIPTION
        This function is called to load the options menu scene and changes the
current scene to the options menu.

    RETURNS
        Nothing.
    */
    /**/
    public void SelectOptions()
    {
        SceneManager.LoadScene("Option-Menu");
    }

    /**/
    /*
    MainMenu::SelectMainMenu() MainMenu::SelectMainMenu()

    NAME
        MainMenu::SelectMainMenu - Returns to the main menu.

    SYNOPSIS
        public void MainMenu::SelectMainMenu();

    DESCRIPTION
        This function is used to return to the main menu scene. It changes the
current scene back to the main menu.

    RETURNS
        Nothing.
    */
    /**/
    public void SelectMainMenu()
    {
        SceneManager.LoadScene("Main-Menu");
    }

    /**/
    /*
    MainMenu::ExitGame() MainMenu::ExitGame()
    NAME
        MainMenu::ExitGame - Exits the game application.

    SYNOPSIS
        public void MainMenu::ExitGame();

    DESCRIPTION
        This function is responsible for quitting the game application.

    RETURNS
        Nothing.
    */
    /**/
    public void ExitGame()
    {
        Application.Quit();
    }
}
```

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

/**/
/*
    This class manages the in-game options menu, providing functionalities such as
adjusting
    game audio settings and setting the difficulty level for AI opponents. It
utilizes Unity's
    AudioMixer to control the game's master volume.
*/
/**/
public class OptionMenu : MonoBehaviour
{
    public AudioMixer audioMixer;

    const string MASTER_VOLUME = "MasterVolume";
    /**/
    /*
    OptionMenu::SetVolume(float volume) OptionMenu::SetVolume(float volume)

    NAME
        OptionMenu::SetVolume - Sets the game's master volume.

    SYNOPSIS
        public void OptionMenu::SetVolume(float volume);
            volume   --> The volume level to set, typically between a min and max
range.

    DESCRIPTION
        This method adjusts the master volume of the game's audio mixer.

    RETURNS
        Nothing.
    */
    /**/
    public void SetVolume(float volume)
    {
        audioMixer.SetFloat(MASTER_VOLUME, volume);
    }

    /**/
    /*
    OptionMenu::GetVolume() OptionMenu::GetVolume()

    NAME
        OptionMenu::GetVolume - Retrieves the current master volume setting.

    SYNOPSIS
        public float OptionMenu::GetVolume();

    DESCRIPTION
        This method fetches the current setting of the master volume from the audio
mixer.

    RETURNS
```

```
        Float volume: The current master volume level.
    */
    /**/
    public float GetVolume()
    {
        float volume;
        bool result = audioMixer.GetFloat(MASTER_VOLUME, out volume);

        // If the volume cannot be fetched, it defaults to 0
        if (result)
        {
            return volume;
        }
        else
        {
            return 0;
        }
    }

    /**/
    /*
    OptionMenu::SetAIDifficulty(string difficulty)
OptionMenu::SetAIDifficulty(string difficulty)

    NAME
        OptionMenu::SetAIDifficulty - Configures the difficulty level for AI opponents.

    SYNOPSIS
        public void OptionMenu::SetAIDifficulty(string difficulty);
            difficulty   --> A string indicating the desired difficulty level ("easy", "medium", "hard").

    DESCRIPTION
        This method is used to set the difficulty level of AI opponents in the game.

    RETURNS
        Nothing.
    */
    /**/
    public void SetAIDifficulty(string difficulty)
    {
        float skillLevel = 0.9f;

        switch (difficulty.ToLower())
        {
            case "easy":
                skillLevel = 0.8f;
                break;
            case "medium":
                skillLevel = 0.9f;
                break;
            case "hard":
                skillLevel = 1.0f;
                break;
        }

        if (GameManager.instance != null)
        {
            GameManager.instance.AIDifficulty = skillLevel;
        }
    }
}
```

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

/**/
/*
    This class handles the in-game pause functionality, allowing players to halt
gameplay and
    access the pause menu. This class is responsible for toggling the visibility of
the pause menu, adjusting
    the game's time scale to pause or resume the game, and providing options to
navigate to different scenes.
*/
```

```
/**/
public class PauseMenu : MonoBehaviour
{
    [SerializeField]
    GameObject pauseMenu;

    /**/
    /*
    PauseMenu::OnPause() PauseMenu::OnPause()

    NAME
        PauseMenu::OnPause - Activates the pause menu and pauses the game.

    SYNOPSIS
        public void PauseMenu::OnPause();

    DESCRIPTION
        This function is triggered to pause the game. It activates the pause menu
UI and sets the
        game's time scale to 0, which pauses all game actions.

    RETURNS
        Nothing.
    */
    /**/
    public void OnPause()
    {
        pauseMenu.SetActive(true);
        Time.timeScale = 0;
    }

    /**/
    /*
    PauseMenu::OnResume() PauseMenu::OnResume()

    NAME
        PauseMenu::OnResume - Resumes the game from a paused state.

    SYNOPSIS
        public void PauseMenu::OnResume();
    DESCRIPTION
        This method resumes the game from a paused state. It deactivates the pause
menu UI and sets
        the game's time scale back to 1, allowing the game to continue running as
normal.

    RETURNS
        Nothing.
    */
    /**/
    public void OnResume()
    {
        pauseMenu.SetActive(false);
        Time.timeScale = 1;
    }

    /**/
    /*
    PauseMenu::OnCarSelectExit() PauseMenu::OnCarSelectExit()

    NAME
        PauseMenu::OnCarSelectExit - Exits to the car selection menu.

    SYNOPSIS
        public void PauseMenu::OnCarSelectExit();

    DESCRIPTION
        This function handles the action of exiting to the car selection menu.

    RETURNS
        Nothing.
    */
    /**/
    public void OnCarSelectExit()
    {
        SceneManager.LoadScene("Select-Menu");
        Time.timeScale = 1;
    }
```



```
/**/
/*
PauseMenu::OnTrackSelectExit() PauseMenu::OnTrackSelectExit()

NAME
    PauseMenu::OnTrackSelectExit - Exits to the track selection menu.

SYNOPSIS
    public void PauseMenu::OnTrackSelectExit();

DESCRIPTION
    This method handles the action of exiting to the track selection menu.

RETURNS
    Nothing.
*/
/**/
public void OnTrackSelectExit()
{
    SceneManager.LoadScene("Level-Menu");
    Time.timeScale = 1;
}

/**/
/*
PauseMenu::OnMainMenuExit() PauseMenu::OnMainMenuExit()

NAME
    PauseMenu::OnMainMenuExit - Exits to the main menu.

SYNOPSIS
    public void PauseMenu::OnMainMenuExit();

DESCRIPTION
    This function handles the action of exiting to the main menu.

RETURNS
    Nothing.
*/
/**/
```

```
    public void OnMainMenuExit()
    {
        SceneManager.LoadScene("Main-Menu");
        Time.timeScale = 1;
    }
}
```

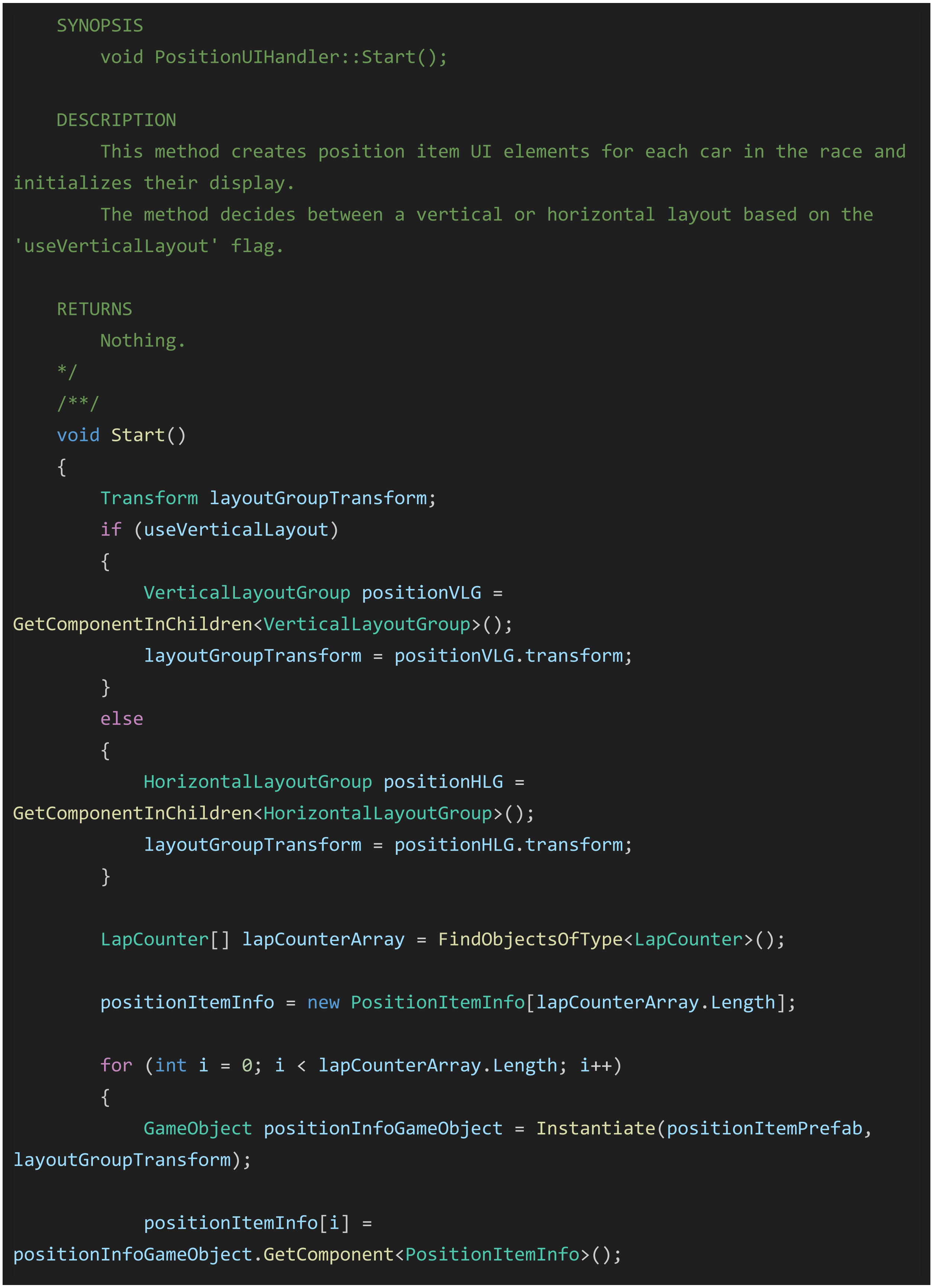
```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

/**/
/*
    This class is responsible for tracking and updating the race positions of all cars in the game.
    It utilizes the LapCounter components attached to each car to determine their current positions based
    on the number of checkpoints passed and the time at the last checkpoint. This class orchestrates the
    sorting of cars based on their progress and updates the UI with the current positions using the PositionUIHandler.
*/
/**/
public class PositionHandler : MonoBehaviour
{
    PositionUIHandler positionUIHandler;

    public List<LapCounter> carLapCounters = new List<LapCounter>();

    /**/
    /*
    PositionHandler::Start() PositionHandler::Start()

    NAME
        PositionHandler::Start - Initializes the PositionHandler component.

    SYNOPSIS
        void PositionHandler::Start();
    DESCRIPTION
        This method initializes the PositionHandler by finding all LapCounter
components in the scene, storing them in a list,
        and setting up event subscriptions for checkpoint passage. Additionally, it
initializes the PositionUIHandler
        to update the UI with the car positions.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        // Get all the lap counters in a scen and stroe them in a list
        LapCounter[] carLapCounterArray = FindObjectsOfType<LapCounter>();
        carLapCounters = carLapCounterArray.ToList<LapCounter>();

        foreach (LapCounter lapCounters in carLapCounters)
            lapCounters.OnPassCheckpoint += OnPassCheckpoint;

        positionUIHandler = FindObjectOfType<PositionUIHandler>();

        positionUIHandler.UpdateList(carLapCounters);
    }

    /**/
    /*
    PositionHandler::OnPassCheckpoint(LapCounter carLapCounter)
PositionHandler::OnPassCheckpoint(LapCounter carLapCounter)

    NAME
        PositionHandler::OnPassCheckpoint - Updates positions when a car passes a
checkpoint.

    SYNOPSIS
        void PositionHandler::OnPassCheckpoint(LapCounter carLapCounter);
            carLapCounter   --> The LapCounter of the car that passed the
checkpoint.
```

```
    DESCRIPTION
        This method is called when a car passes a checkpoint. It sorts the list of
LapCounters based
        on the number of checkpoints passed and the time at the last checkpoint,
updating the race
        positions of all cars. It also updates the PositionUIHandler to reflect
these changes.

    RETURNS
        Nothing.
    */
    /**/
    void OnPassCheckpoint(LapCounter carLapCounter)
    {
        // Sort the car's positon first based on how many checkpoints they have
passed, then sort on time
        carLapCounters = carLapCounters.OrderByDescending(s =>
s.GetNumberOfCheckpointsPassed()).ThenBy(s =>
s.GetTimeAtLastCheckPoint()).ToList();

        int carPosition = carLapCounters.IndexOf(carLapCounter) + 1;

        carLapCounter.SetCarPosition(carPosition);

        positionUIHandler.UpdateList(carLapCounters);
    }
}
```

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

/**/
/*
    This class is responsible for managing the display of race position and car
name information
    in a user interface element. It primarily interacts with text components from
the Unity UI framework
```

```
    to update and reflect the current race position and the corresponding car's name.
*/
/**/
public class PositionItemInfo : MonoBehaviour
{
    public Text positionText;
    public Text carNameText;

    /**/
    /*
    PositionItemInfo::SetPositionText(string newPosition)
PositionItemInfo::SetPositionText(string newPosition)

    NAME
        PositionItemInfo::SetPositionText - Updates the position display text.

    SYNOPSIS
        public void PositionItemInfo::SetPositionText(string newPosition);
            newPosition  --> The new position to be displayed.

    DESCRIPTION
        This function updates the text displaying the position in the race.

    RETURNS
        Nothing.
    */
    /**/
    public void SetPositionText(string newPosition)
    {
        positionText.text = newPosition;
    }

    /**/
    /*
    PositionItemInfo::SetCarName(string newCarName)
PositionItemInfo::SetCarName(string newCarName)

    NAME
        PositionItemInfo::SetCarName - Updates the car name display text.
    SYNOPSIS
        public void PositionItemInfo::SetCarName(string newCarName);
            newCarName  --> The new car name to be displayed.

    DESCRIPTION
        This method is used to update the text displaying the name of the car.

    RETURNS
        Nothing.
    */
    /**/
    public void SetCarName(string newCarName)
    {
        carNameText.text = newCarName;
    }
}
```

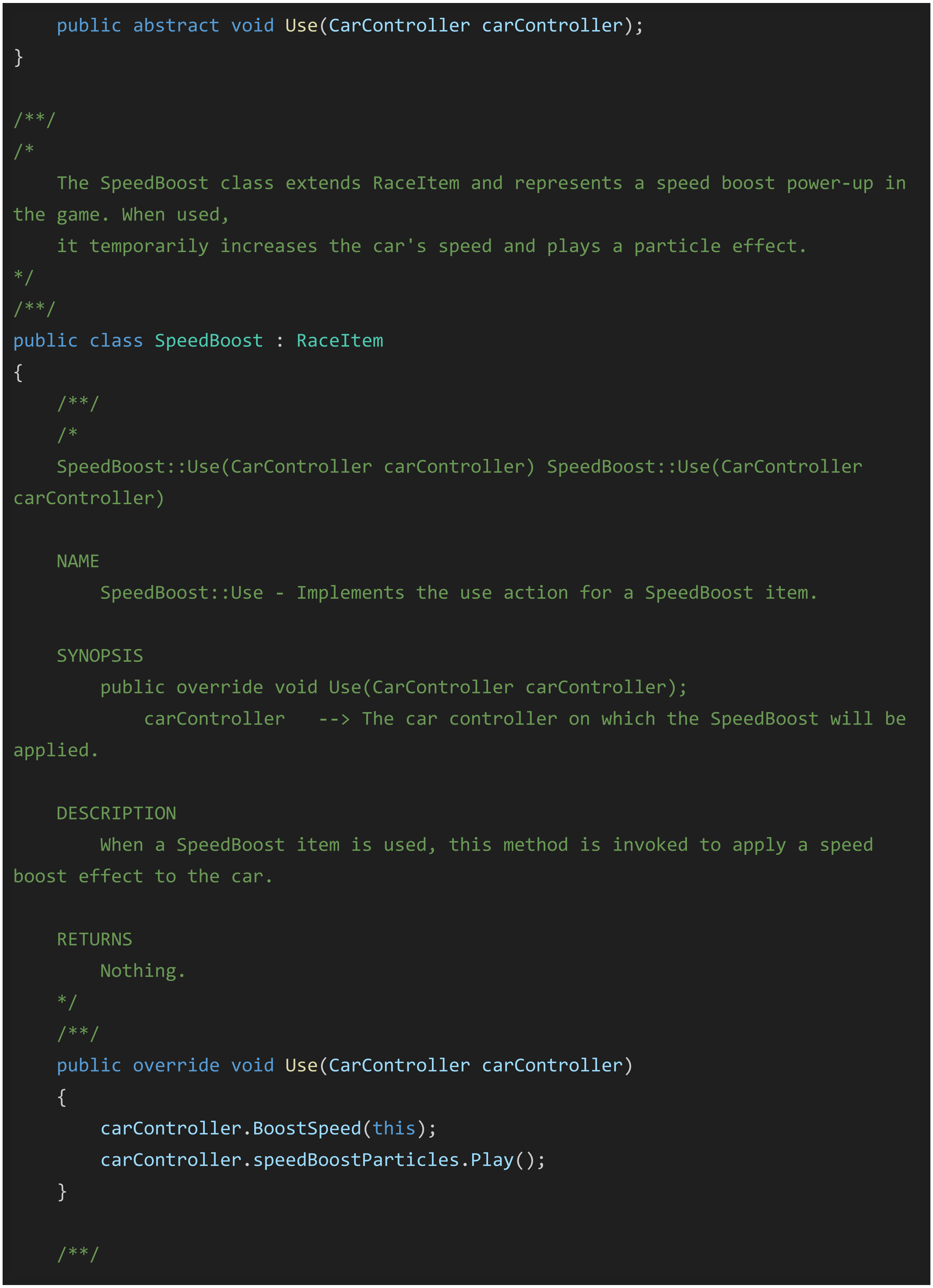
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

/**/
/*
    The PositionUIHandler class is responsible for managing the display of race
positions for each
    car in the game. It handles the creation and updating of UI elements that
represent each car's
    position in the race. This class uses either a vertical or horizontal layout to
display the position information,
    based on the configuration.
*/
/**/
public class PositionUIHandler : MonoBehaviour
{
    public GameObject positionItemPrefab;

    PositionItemInfo[] positionItemInfo;
    Canvas canvas;

    bool isInitialized = false;

    public bool useVerticalLayout = true;

    /**/
    /*
    PositionUIHandler::Awake() PositionUIHandler::Awake()

    NAME
        PositionUIHandler::Awake - Initializes the Position UI Handler.

    SYNOPSIS
        void PositionUIHandler::Awake();

    DESCRIPTION
        This method initializes the canvas and determines the layout type (vertical or horizontal).

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        canvas = GetComponent<Canvas>();
        canvas.enabled = !useVerticalLayout;

        GameManager.instance.OnGameStateChanged += OnGameStateChanged;
    }

    /**/
    /*
    PositionUIHandler::Start() PositionUIHandler::Start()

    NAME
        PositionUIHandler::Start - Prepares position item UI elements.
```



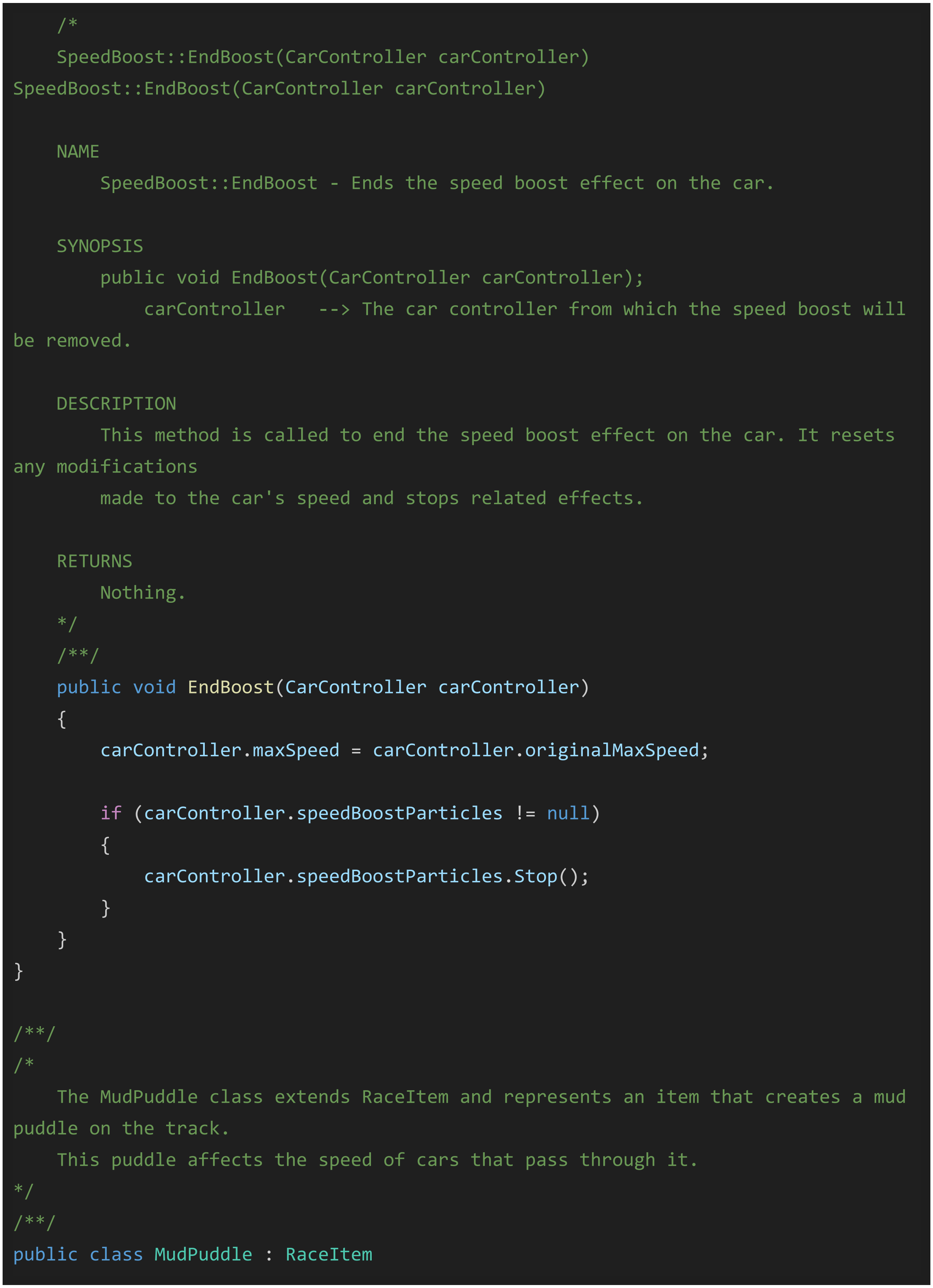
```
    SYNOPSIS
        void PositionUIHandler::Start();

    DESCRIPTION
        This method creates position item UI elements for each car in the race and
initializes their display.
        The method decides between a vertical or horizontal layout based on the
'useVerticalLayout' flag.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        Transform layoutGroupTransform;
        if (useVerticalLayout)
        {
            VerticalLayoutGroup positionVLG =
GetComponentInChildren<VerticalLayoutGroup>();
            layoutGroupTransform = positionVLG.transform;
        }
        else
        {
            HorizontalLayoutGroup positionHLG =
GetComponentInChildren<HorizontalLayoutGroup>();
            layoutGroupTransform = positionHLG.transform;
        }

        LapCounter[] lapCounterArray = FindObjectsOfType<LapCounter>();

        positionItemInfo = new PositionItemInfo[lapCounterArray.Length];

        for (int i = 0; i < lapCounterArray.Length; i++)
        {
            GameObject positionInfoGameObject = Instantiate(positionItemPrefab,
layoutGroupTransform);

            positionItemInfo[i] =
positionInfoGameObject.GetComponent<PositionItemInfo>();
```

```
            positionItemInfo[i].SetPositionText($"{i + 1}.");
        }

        Canvas.ForceUpdateCanvases();

        isInitialized = true;
    }

    /**/
    /*
    PositionUIHandler::UpdateList(List<LapCounter> lapCounters)
PositionUIHandler::UpdateList(List<LapCounter> lapCounters)

    NAME
        PositionUIHandler::UpdateList - Updates the position UI list.

    SYNOPSIS
        public void PositionUIHandler::UpdateList(List<LapCounter> lapCounters);
            lapCounters    --> List of LapCounter objects representing cars in the race.

    DESCRIPTION
        This method updates the list of position UI elements to reflect the current positions of each car in the race.

    RETURNS
        Nothing.
    */
    /**/
    public void UpdateList(List<LapCounter> lapCounters)
    {
        if (!isInitialized)
        {
            return;
        }

        for (int i = 0; i < lapCounters.Count; i++)
        {
            positionItemInfo[i].SetCarName(lapCounters[i].gameObject.name);
            Debug.Log($"Updating car {i}: {lapCounters[i].gameObject.name}");
        }
    }

    /**/
    /*
    PositionUIHandler::OnGameStateChanged(GameManager gameManager)
PositionUIHandler::OnGameStateChanged(GameManager gameManager)

    NAME
        PositionUIHandler::OnGameStateChanged - Responds to changes in the game
state.

    SYNOPSIS
        void PositionUIHandler::OnGameStateChanged(GameManager gameManager);

    DESCRIPTION
        This method is invoked when there is a change in the game state. It is
responsible for
        enabling or disabling the canvas based on the game state and the layout
type.

    RETURNS
        Nothing.
    */
    /**/
    void OnGameStateChanged(GameManager gameManager)
    {
        if (GameManager.instance.GetGameState() == GameStates.raceOver)
        {
            if (!useVerticalLayout)
            {
                canvas.enabled = false;
            }
            else
            {
                canvas.enabled = true;
            }
        }
    }
    /**/
    /*
    PositionUIHandler::OnDestroy() PositionUIHandler::OnDestroy()

    NAME
        PositionUIHandler::OnDestroy - Cleans up before the object is destroyed.

    SYNOPSIS
        void PositionUIHandler::OnDestroy();

    DESCRIPTION
        This method is called when the script object is being destroyed.

    RETURNS
        Nothing.
    */
    /**/
    void OnDestroy()
    {
        GameManager.instance.OnGameStateChanged -= OnGameStateChanged;
    }
}
```

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

/**/
/*
    This class is an abstract base class for items that can be used in the game. It
defines a common
    interface for all race items, requiring the implementation of a Use method,
which specifies how each item
    affects the car or the race when used.
*/
/**/
public abstract class RaceItem
{
```



```
    public abstract void Use(CarController carController);
}

/**/
/*
    The SpeedBoost class extends RaceItem and represents a speed boost power-up in
the game. When used,
    it temporarily increases the car's speed and plays a particle effect.
*/
/**/
public class SpeedBoost : RaceItem
{
    /**/
    /*
    SpeedBoost::Use(CarController carController) SpeedBoost::Use(CarController
carController)

    NAME
        SpeedBoost::Use - Implements the use action for a SpeedBoost item.

    SYNOPSIS
        public override void Use(CarController carController);
            carController   --> The car controller on which the SpeedBoost will be
applied.

    DESCRIPTION
        When a SpeedBoost item is used, this method is invoked to apply a speed
boost effect to the car.

    RETURNS
        Nothing.
    */
    /**/
    public override void Use(CarController carController)
    {
        carController.BoostSpeed(this);
        carController.speedBoostParticles.Play();
    }

    /**/
```



```
    /*
    SpeedBoost::EndBoost(CarController carController)
SpeedBoost::EndBoost(CarController carController)

    NAME
        SpeedBoost::EndBoost - Ends the speed boost effect on the car.

    SYNOPSIS
        public void EndBoost(CarController carController);
            carController   --> The car controller from which the speed boost will
be removed.

    DESCRIPTION
        This method is called to end the speed boost effect on the car. It resets
any modifications
        made to the car's speed and stops related effects.

    RETURNS
        Nothing.
    */
    /**/
    public void EndBoost(CarController carController)
    {
        carController.maxSpeed = carController.originalMaxSpeed;

        if (carController.speedBoostParticles != null)
        {
            carController.speedBoostParticles.Stop();
        }
    }
}

/**/
/*
    The MudPuddle class extends RaceItem and represents an item that creates a mud
puddle on the track.
    This puddle affects the speed of cars that pass through it.
*/
/**/
public class MudPuddle : RaceItem
```

```
{
    public float duration = 5f;

    /**/
    /*
    MudPuddle::Use(CarController carController) MudPuddle::Use(CarController carController)

    NAME
        MudPuddle::Use - Implements the use action for a MudPuddle item.

    SYNOPSIS
        public override void Use(CarController carController);
            carController   --> The car controller on which the MudPuddle will be applied.

    DESCRIPTION
        When a MudPuddle item is used, this method is invoked to create a mud puddle effect on the race track.

    RETURNS
        Nothing.
    */
    /**/
    public override void Use(CarController carController)
    {
        carController.DropMudPuddle(duration);
    }
}
```

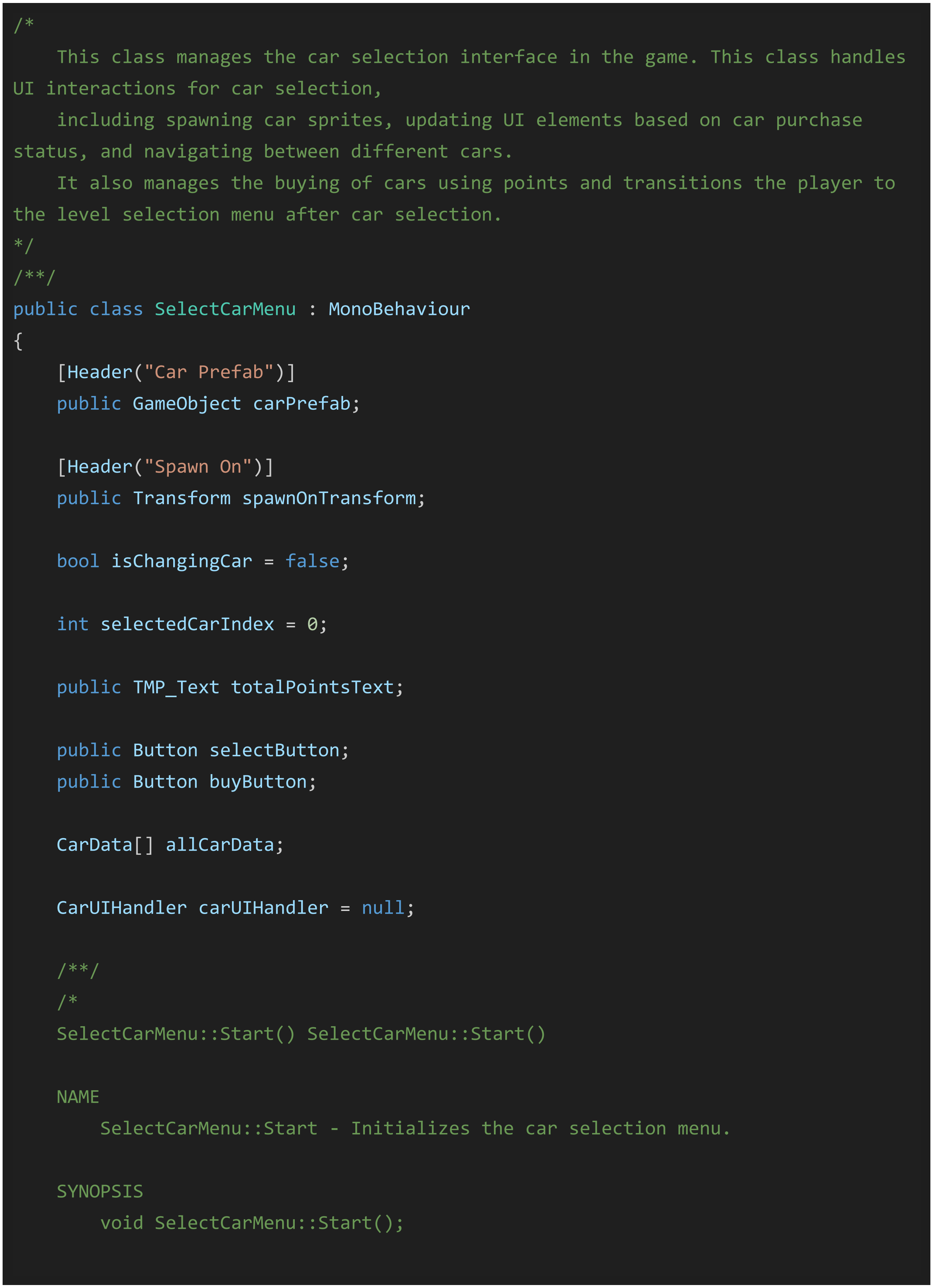
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;

/**/
/*
```

```
    This class is responsible for managing the display of race results in the game.
It controls a canvas
    that shows the player's score for the most recent race, their total accumulated
score, and provides options
    to navigate to different menus.
*/
/**/
public class ResultUIHandler : MonoBehaviour
{
    Canvas canvas;

    public TMP_Text scoreText;
    public TMP_Text totalScoreText;

    /**/
    /*
    ResultUIHandler::Awake() ResultUIHandler::Awake()

    NAME
        ResultUIHandler::Awake - Initializes the Result UI Handler.

    SYNOPSIS
        private void ResultUIHandler::Awake();

    DESCRIPTION
        This function initializes the canvas component and prepares the UI handler
to display the race results
        correctly once the race is over.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        canvas = GetComponent<Canvas>();

        canvas.enabled = false;

        GameManager.instance.OnGameStateChanged += OnGameStateChanged;
    }

    /**/
    /*
    ResultUIHandler::OnRaceAgain() ResultUIHandler::OnRaceAgain()

    NAME
        ResultUIHandler::OnRaceAgain - Reloads the current race scene.

    SYNOPSIS
        public void ResultUIHandler::OnRaceAgain();

    DESCRIPTION
        This method is called when the player chooses to race again. It reloads the
current active scene,
        restarting the race.

    RETURNS
        Nothing.
    */
    /**/
    public void OnRaceAgain()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    /**/
    /*
    ResultUIHandler::OnCarSelectExit() ResultUIHandler::OnCarSelectExit()

    NAME
        ResultUIHandler::OnCarSelectExit - Navigates to the car selection menu.

    SYNOPSIS
        public void ResultUIHandler::OnCarSelectExit();

    DESCRIPTION
        This method is called when the player decides to exit to the car selection
menu after a race.
```

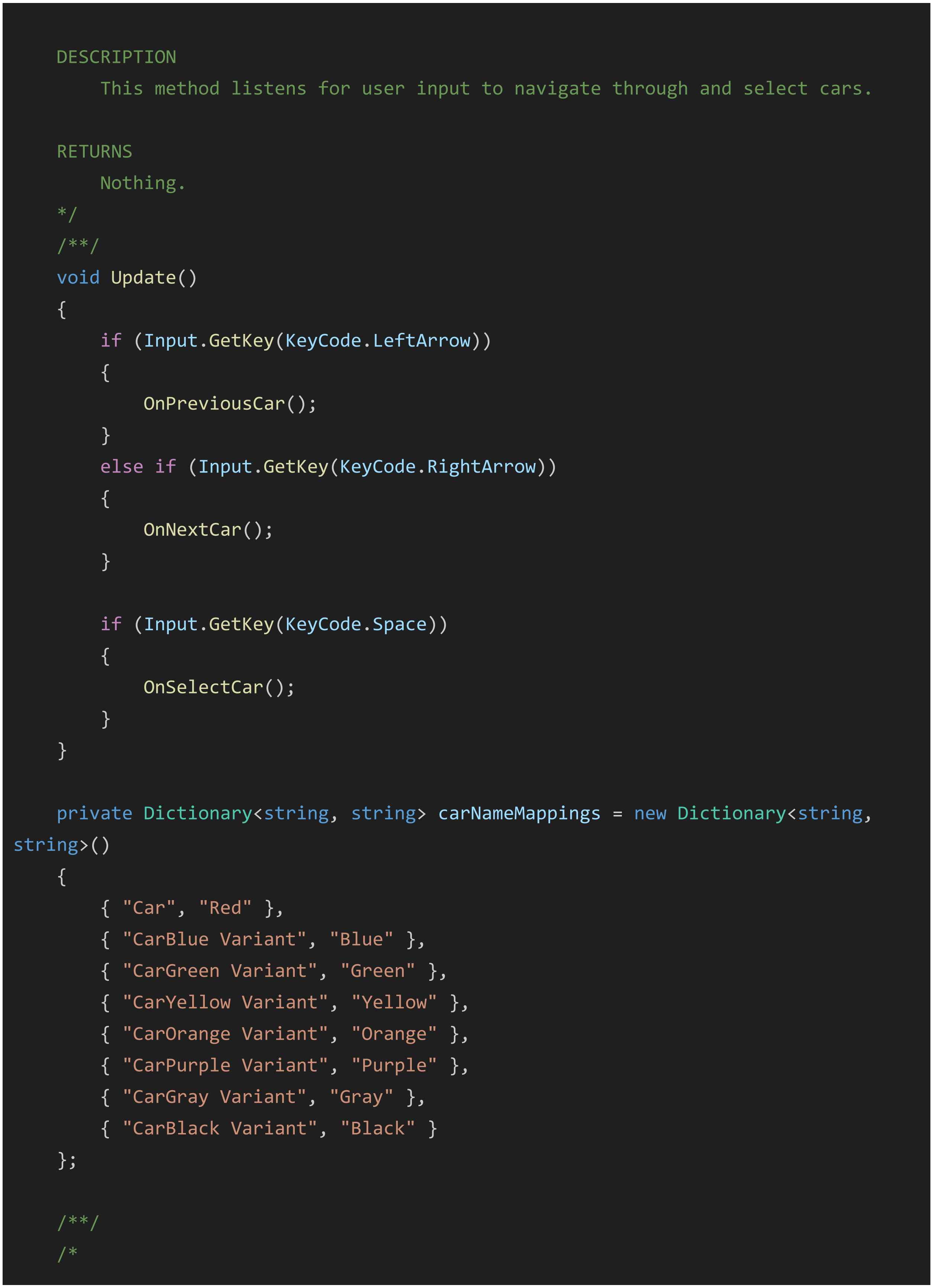
```
    RETURNS
        Nothing.
    */
    /**/
    public void OnCarSelectExit()
    {
        SceneManager.LoadScene("Select-Menu");
    }

    /**/
    /*
    ResultUIHandler::OnTrackSelectExit() ResultUIHandler::OnTrackSelectExit()

    NAME
        ResultUIHandler::OnTrackSelectExit - Navigates to the track selection menu.

    SYNOPSIS
        public void ResultUIHandler::OnTrackSelectExit();

    DESCRIPTION
        This function is invoked when the player decides to go to the track
selection menu.

    RETURNS
        Nothing.
    */
    /**/
    public void OnTrackSelectExit()
    {
        SceneManager.LoadScene("Level-Menu");
    }

    /**/
    /*
    ResultUIHandler::OnMainMenuExit() ResultUIHandler::OnMainMenuExit()

    NAME
        ResultUIHandler::OnMainMenuExit - Returns to the main menu.

    SYNOPSIS
```

```
        public void ResultUIHandler::OnMainMenuExit();

    DESCRIPTION
        This method is triggered when the player decides to return to the main
menu.

    RETURNS
        Nothing.
    */
    /**/
    public void OnMainMenuExit()
    {
        SceneManager.LoadScene("Main-Menu");
    }

    /**/
    /*
    ResultUIHandler::DisplayRaceScore(int raceScore)
ResultUIHandler::DisplayRaceScore(int raceScore)

    NAME
        ResultUIHandler::DisplayRaceScore - Displays the score of the most recent
race.

    SYNOPSIS
        public void ResultUIHandler::DisplayRaceScore(int raceScore);
            raceScore    --> The score achieved in the most recent race.

    DESCRIPTION
        This method updates the score display to show the score earned in the most
recent race.

    RETURNS
        Nothing.
    */
    /**/
    public void DisplayRaceScore(int raceScore)
    {
        if (scoreText != null)
            scoreText.text = raceScore.ToString();
    }

    /**/
    /*
    ResultUIHandler::DisplayTotalScore() ResultUIHandler::DisplayTotalScore()

    NAME
        ResultUIHandler::DisplayTotalScore - Displays the player's total accumulated score.

    SYNOPSIS
        public void ResultUIHandler::DisplayTotalScore();

    DESCRIPTION
        This function retrieves and displays the total points accumulated by the player over multiple races.

    RETURNS
        Nothing.
    */
    /**/
    public void DisplayTotalScore()
    {
        if (totalScoreText != null)
        {
            int totalScore = GameManager.instance.totalPoints;
            totalScoreText.text = totalScore.ToString();
        }
    }

    /**/
    /*
    ResultUIHandler::ShowMenuCO() ResultUIHandler::ShowMenuCO()

    NAME
        ResultUIHandler::ShowMenuCO - Coroutine to display the result menu.

    SYNOPSIS
        IEnumerator ResultUIHandler::ShowMenuCO();
    DESCRIPTION
        This coroutine waits for a set duration before enabling the canvas to 
display the race results.

    RETURNS
        IEnumerator that temporarily halts execution for a second.
    */
    /**/
    IEnumerator ShowMenuCO()
    {
        yield return new WaitForSeconds(1);

        canvas.enabled = true;
    }

    /**/
    /*
    ResultUIHandler::OnGameStateChanged(GameManager gameManager)
ResultUIHandler::OnGameStateChanged(GameManager gameManager)

    NAME
        ResultUIHandler::OnGameStateChanged - Handles changes in the game state.

    SYNOPSIS
        void ResultUIHandler::OnGameStateChanged(GameManager gameManager);
            gameManager    --> Reference to the GameManager instance.

    DESCRIPTION
        This method responds to changes in the game's state. When the race is over, 
it displays the race score
        and the total score, and initiates the result menu display coroutine.

    RETURNS
        Nothing.
    */
    /**/
    void OnGameStateChanged(GameManager gameManager)
    {
        if (GameManager.instance.GetGameState() == GameStates.raceOver)
        {
            DisplayRaceScore(gameManager.GetLastRaceScore());
            DisplayTotalScore();
            StartCoroutine(ShowMenuCO());
        }
    }

    /**/
    /*
    ResultUIHandler::OnDestroy() ResultUIHandler::OnDestroy()

    NAME
        ResultUIHandler::OnDestroy - Cleanup when the object is destroyed.

    SYNOPSIS
        void ResultUIHandler::OnDestroy();

    DESCRIPTION
        This method ensures that the event handler is properly removed when the object is destroyed.

    RETURNS
        Nothing.
    */
    /**/
    void OnDestroy()
    {
        GameManager.instance.OnGameStateChanged -= OnGameStateChanged;
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

/**/
```



```
/*
    This class manages the car selection interface in the game. This class handles UI interactions for car selection,
    including spawning car sprites, updating UI elements based on car purchase status, and navigating between different cars.
    It also manages the buying of cars using points and transitions the player to the level selection menu after car selection.
*/
/**/
public class SelectCarMenu : MonoBehaviour
{
    [Header("Car Prefab")]
    public GameObject carPrefab;

    [Header("Spawn On")]
    public Transform spawnOnTransform;

    bool isChangingCar = false;

    int selectedCarIndex = 0;

    public TMP_Text totalPointsText;

    public Button selectButton;
    public Button buyButton;

    CarData[] allCarData;

    CarUIHandler carUIHandler = null;

    /**/
    /*
    SelectCarMenu::Start() SelectCarMenu::Start()

    NAME
        SelectCarMenu::Start - Initializes the car selection menu.

    SYNOPSIS
        void SelectCarMenu::Start();

    DESCRIPTION
        This method loads all car data, marks free cars as purchased, updates the
display of total points,
        adds a listener to the buy button, and starts the coroutine to spawn the
first car.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        allCarData = Resources.LoadAll<CarData>("CarData/");

        foreach (var carData in allCarData)
        {
            if (carData.Cost == 0 &&
!GameManager.instance.IsCarPurchased(carData.CarUniqueID))
            {
                GameManager.instance.MarkCarAsPurchased(carData.CarUniqueID);
            }
        }

        UpdateTotalPointsDisplay();

        buyButton.onClick.AddListener(OnBuyButtonClicked);

        StartCoroutine(SpawnCarCO(true));
    }

    /**/
    /*
    SelectCarMenu::Update() SelectCarMenu::Update()

    NAME
        SelectCarMenu::Update - Handles real-time input during the car selection
process.

    SYNOPSIS
        void SelectCarMenu::Update();
```



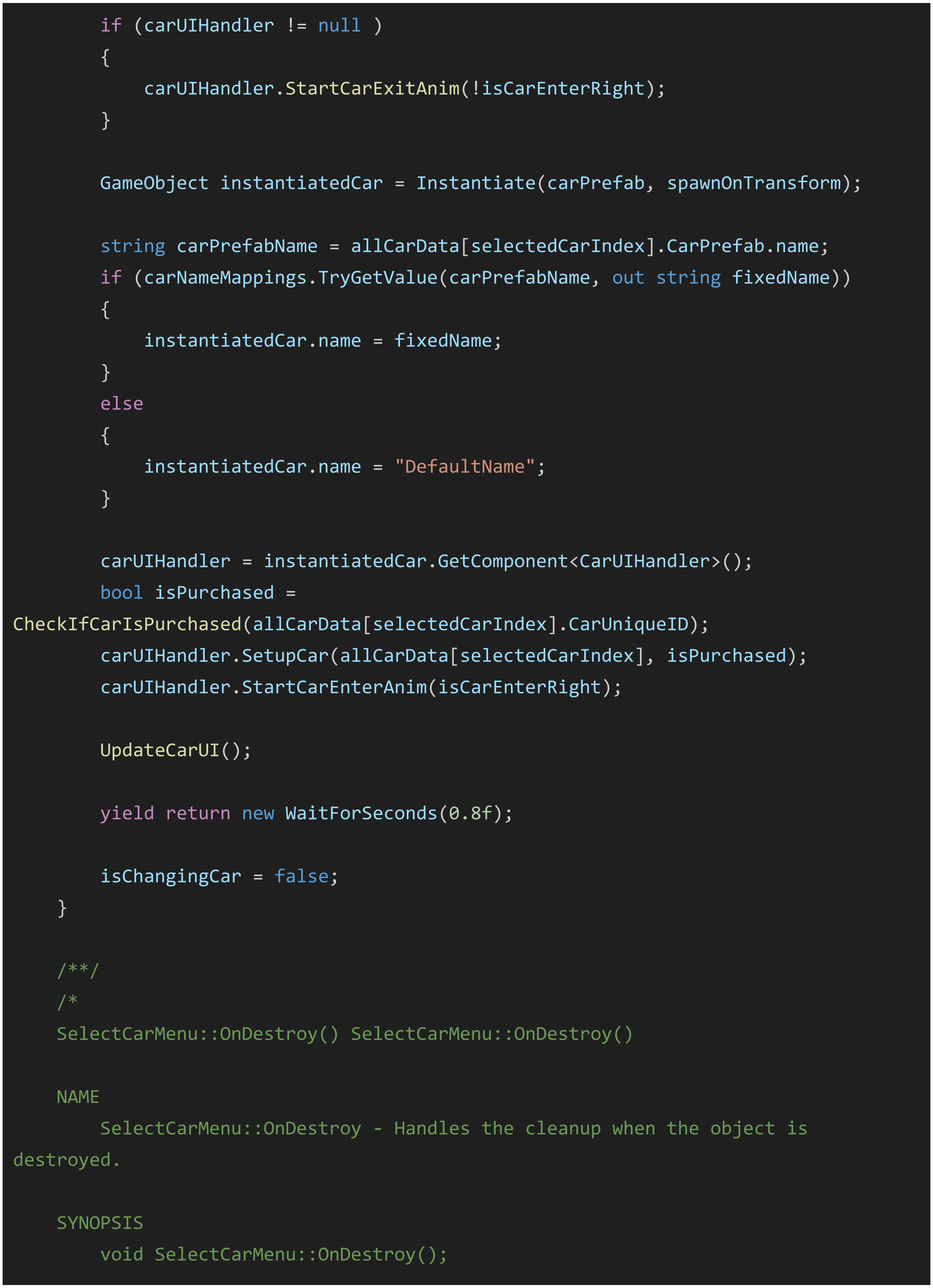
```
    DESCRIPTION
        This method listens for user input to navigate through and select cars.

    RETURNS
        Nothing.
    */
    /**/
    void Update()
    {
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            OnPreviousCar();
        }
        else if (Input.GetKey(KeyCode.RightArrow))
        {
            OnNextCar();
        }

        if (Input.GetKey(KeyCode.Space))
        {
            OnSelectCar();
        }
    }

    private Dictionary<string, string> carNameMappings = new Dictionary<string, string>()
    {
        { "Car", "Red" },
        { "CarBlue Variant", "Blue" },
        { "CarGreen Variant", "Green" },
        { "CarYellow Variant", "Yellow" },
        { "CarOrange Variant", "Orange" },
        { "CarPurple Variant", "Purple" },
        { "CarGray Variant", "Gray" },
        { "CarBlack Variant", "Black" }
    };

    /**/
    /*
    SelectCarMenu::UpdateCarUI() SelectCarMenu::UpdateCarUI()

    NAME
        SelectCarMenu::UpdateCarUI - Updates the UI based on the car's purchase
status.

    SYNOPSIS
        void SelectCarMenu::UpdateCarUI();

    DESCRIPTION
        This method updates the car selection UI elements based on whether the
currently selected car
        has been purchased.

    RETURNS
        Nothing.
    */
    /**/
    private void UpdateCarUI()
    {
        CarData selectedCarData = allCarData[selectedCarIndex];
        bool isCarPurchased =
GameManager.instance.IsCarPurchased(selectedCarData.CarUniqueID);

        // Show the buy button only if the car has not been purchased
        buyButton.gameObject.SetActive(!isCarPurchased);

        // Show the select button only if the car has been purchased
        selectButton.gameObject.SetActive(isCarPurchased);
    }

    /**/
    /*
    SelectCarMenu::OnPreviousCar() SelectCarMenu::OnPreviousCar()

    NAME
        SelectCarMenu::OnPreviousCar - Handles the action of selecting the previous
car in the car selection menu.

    SYNOPSIS
```

```
        void SelectCarMenu::OnPreviousCar();

    DESCRIPTION
        This method is triggered when the player wants to view the previous car in the car selection menu.
        It decrements the selectedCarIndex, ensuring it wraps around if it goes below zero. The method
        then calls SpawnCarCO coroutine to handle the car changing animation and updates the car UI accordingly.

    RETURNS
        Nothing.
    */
    /**/
    public void OnPreviousCar()
    {
        if (isChangingCar)
        {
            return;
        }

        selectedCarIndex--;

        if (selectedCarIndex < 0)
        {
            selectedCarIndex = allCarData.Length - 1;
        }

        StartCoroutine(SpawnCarCO(true));

        UpdateCarUI();
    }

    /**/
    /*
    SelectCarMenu::OnNextCar() SelectCarMenu::OnNextCar()

    NAME
        SelectCarMenu::OnNextCar - Navigates to the next car in the selection menu.
    SYNOPSIS
        void SelectCarMenu::OnNextCar();

    DESCRIPTION
        This method is called to navigate to the next car in the car selection
menu. It increments
        the selectedCarIndex and loops back to zero if it exceeds the total number
of cars.
        It also invokes a coroutine for the car changing animation and updates the
UI accordingly.

    RETURNS
        Nothing.
    */
    /**/
    public void OnNextCar()
    {
        if (isChangingCar)
        {
            return;
        }

        selectedCarIndex++;

        if (selectedCarIndex > allCarData.Length - 1)
        {
            selectedCarIndex = 0;
        }

        StartCoroutine(SpawnCarCO(false));

        UpdateCarUI();
    }

    /**/
    /*
    SelectCarMenu::OnSelectCar() SelectCarMenu::OnSelectCar()

    NAME
        SelectCarMenu::OnSelectCar - Finalizes the car selection process and loads
```

```
the level selection menu.

    SYNOPSIS
        void SelectCarMenu::OnSelectCar();

    DESCRIPTION
        This method finalizes the car selection by the player and prepares the game
for the next stage.
        It clears the existing player list in the GameManager, adds the selected
player and AI players with
        their respective cars, and transitions to the level selection scene.

    RETURNS
        Nothing.
    */
    /**/
    public void OnSelectCar()
    {
        GameManager.instance.ClearPlayerList();

        GameManager.instance.AddPlayerToList(1, "Player1",
allCarData[selectedCarIndex].CarUniqueID, false);

        List<CarData> uniqueCars = new List<CarData>(allCarData);

        // Remove the car that player has selected
        uniqueCars.Remove(allCarData[selectedCarIndex]);

        string[] names = { "Red", "Blue", "Green", "Yellow", "Orange", "Purple",
"Gray", "Black" };
        List<string> uniqueNames = names.ToList<string>();

        foreach (CarData aiCarData in uniqueCars)
        {
            string carPrefabName = aiCarData.CarPrefab.name;

            // Use the carNameMappings dictionary to get a specific name for each
AI car
            if (carNameMappings.TryGetValue(carPrefabName, out string
aiDriverName))
```

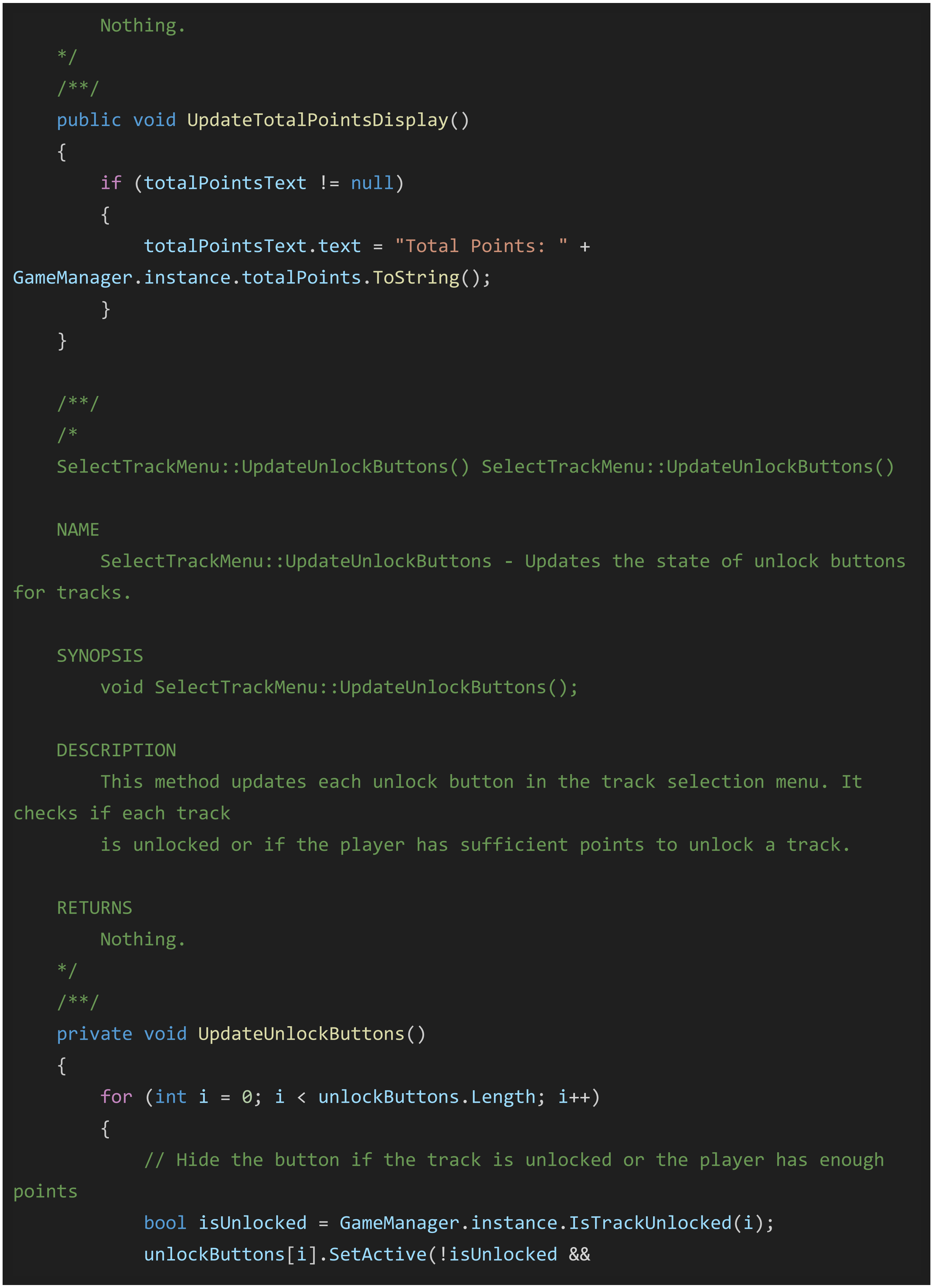
```
            {
                GameManager.instance.AddPlayerToList(aiCarData.CarUniqueID,
aiDriverName, aiCarData.CarUniqueID, true);
            }
            else
            {
                // Default name if no mapping is found
                GameManager.instance.AddPlayerToList(aiCarData.CarUniqueID,
"DefaultAIName", aiCarData.CarUniqueID, true);
            }
        }

        SceneManager.LoadScene("Level-Menu");
    }

    /**/
    /*
    SelectCarMenu::UpdateTotalPointsDisplay()
SelectCarMenu::UpdateTotalPointsDisplay()

    NAME
        SelectCarMenu::UpdateTotalPointsDisplay - Updates the display of total
points in the UI.

    SYNOPSIS
        void SelectCarMenu::UpdateTotalPointsDisplay();

    DESCRIPTION
        This method updates the total points text display in the car selection menu
to show the current point balance.

    RETURNS
        Nothing.
    */
    /**/
    private void UpdateTotalPointsDisplay()
    {
        if (totalPointsText != null)
        {
            totalPointsText.text = "Total Points: " +
GameManager.instance.totalPoints.ToString();
        }
    }

    /**/
    /*
    SelectCarMenu::OnBuyButtonClicked() SelectCarMenu::OnBuyButtonClicked()

    NAME
        SelectCarMenu::OnBuyButtonClicked - Handles the 'Buy Car' button click
event.

    SYNOPSIS
        void SelectCarMenu::OnBuyButtonClicked();

    DESCRIPTION
        This method is invoked when the 'Buy Car' button is clicked. It checks if
the player has enough points to
        purchase the selected car. If sufficient points are available, the car's
cost is deducted from the player's
        total points, and the car is marked as purchased.

    RETURNS
        Nothing.
    */
    /**/
    private void OnBuyButtonClicked()
    {
        CarData selectedCarData = allCarData[selectedCarIndex];
        int carCost = selectedCarData.Cost;

        // Check if the player has enough points
        if (GameManager.instance.totalPoints >= carCost)
        {
            GameManager.instance.totalPoints -= carCost;

            GameManager.instance.MarkCarAsPurchased(selectedCarData.CarUniqueID);

            // Update the UI to reflect the new points total and purchase status
            UpdateTotalPointsDisplay();
            UpdateCarPurchaseUI();
        }
        else
        {
            Debug.Log("Not enough points to buy this car");
        }
        UpdateCarUI();
    }

    /**/
    /*
    SelectCarMenu::UpdateCarPurchaseUI() SelectCarMenu::UpdateCarPurchaseUI()

    NAME
        SelectCarMenu::UpdateCarPurchaseUI - Updates the purchase UI for the
selected car.

    SYNOPSIS
        void SelectCarMenu::UpdateCarPurchaseUI();

    DESCRIPTION
        This method updates the UI elements related to the car purchase.

    RETURNS
        Nothing.
    */
    /**/
    private void UpdateCarPurchaseUI()
    {
        CarData selectedCarData = allCarData[selectedCarIndex];
        bool canAffordCar = GameManager.instance.totalPoints >=
selectedCarData.Cost;
        bool isCarPurchased =
GameManager.instance.IsCarPurchased(selectedCarData.CarUniqueID);

        buyButton.gameObject.SetActive(canAffordCar && !isCarPurchased);
    }

    /**/
    /*
    SelectCarMenu::CanAffordSelectedCar() SelectCarMenu::CanAffordSelectedCar()

    NAME
        SelectCarMenu::CanAffordSelectedCar - Checks if the player can afford the
selected car.

    SYNOPSIS
        bool SelectCarMenu::CanAffordSelectedCar();

    DESCRIPTION
        This method returns a boolean value indicating whether the player has
enough points to purchase the currently
        selected car based on its cost.

    RETURNS
        True if the player can afford the car, False otherwise.
    */
    /**/
    private bool CanAffordSelectedCar()
    {
        int carCost = allCarData[selectedCarIndex].Cost;
        return GameManager.instance.totalPoints >= carCost;
    }

    /**/
    /*
    SelectCarMenu::CheckIfCarIsPurchased(int carID)
SelectCarMenu::CheckIfCarIsPurchased(int carID)

    NAME
        SelectCarMenu::CheckIfCarIsPurchased - Verifies if a car is already
purchased.

    SYNOPSIS
        bool SelectCarMenu::CheckIfCarIsPurchased(int carID);
            carID   --> The unique ID of the car to check.

    DESCRIPTION
        This method checks if a car with the specified unique ID has already been
purchased by the player.
```

```
    RETURNS
        True if the car is purchased, False otherwise.
    */
    /**/
    private bool CheckIfCarIsPurchased(int carID)
    {
        return GameManager.instance.IsCarPurchased(carID);
    }


    /**/
    /*
    SelectCarMenu::SpawnCarCO() SelectCarMenu::SpawnCarCO()

    NAME
        SelectCarMenu::SpawnCarCO - Coroutine for spawning and displaying cars in
the car selection menu.

    SYNOPSIS
        IEnumerator SelectCarMenu::SpawnCarCO(bool isCarEnterRight);
            isCarEnterRight   --> Indicates if the car should enter from the right
side.

    DESCRIPTION
        This coroutine manages the car spawning process in the car selection menu.
It instantiates the car prefab,
        assigns a name based on the car data, and manages the car's entrance
animation to allow for a smooth transition
        between car selections.

    RETURNS
        IEnumerator that temporarily halts execution to properly time the spawn
animation.
    */
    /**/
    IEnumerator SpawnCarCO(bool isCarEnterRight)
    {
        isChangingCar = true;
```



```
        if (carUIHandler != null )
        {
            carUIHandler.StartCarExitAnim(!isCarEnterRight);
        }

        GameObject instantiatedCar = Instantiate(carPrefab, spawnOnTransform);

        string carPrefabName = allCarData[selectedCarIndex].CarPrefab.name;
        if (carNameMappings.TryGetValue(carPrefabName, out string fixedName))
        {
            instantiatedCar.name = fixedName;
        }
        else
        {
            instantiatedCar.name = "DefaultName";
        }

        carUIHandler = instantiatedCar.GetComponent<CarUIHandler>();
        bool isPurchased =
CheckIfCarIsPurchased(allCarData[selectedCarIndex].CarUniqueID);
        carUIHandler.SetupCar(allCarData[selectedCarIndex], isPurchased);
        carUIHandler.StartCarEnterAnim(isCarEnterRight);

        UpdateCarUI();

        yield return new WaitForSeconds(0.8f);

        isChangingCar = false;
    }

    /**/
    /*
    SelectCarMenu::OnDestroy() SelectCarMenu::OnDestroy()

    NAME
        SelectCarMenu::OnDestroy - Handles the cleanup when the object is
destroyed.

    SYNOPSIS
        void SelectCarMenu::OnDestroy();
    DESCRIPTION
        This method is called when the SelectCarMenu object is being destroyed.

    RETURNS
        Nothing.
    */
    /**/
    private void OnDestroy()
    {
        buyButton.onClick.RemoveListener(OnBuyButtonClicked);
    }
}
```

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

/**/
/*
    This class manages the track selection interface in the game. It allows players to view, unlock,
    and select tracks for racing. This class also handles UI interactions for track selection,
    including displaying total points and updating the state of unlock buttons for each track.
*/
/**/
public class SelectTrackMenu : MonoBehaviour
{
    public TMP_Text totalPointsText;

    public GameObject[] unlockButtons;

    /**/
    /*
    SelectTrackMenu::Start() SelectTrackMenu::Start()

    NAME
        SelectTrackMenu::Start - Initializes the track selection menu.

    SYNOPSIS
        void SelectTrackMenu::Start();

    DESCRIPTION
        This method initializes the track selection menu by updating the total
points display and
        the state of the unlock buttons for each track.

    RETURNS
        Nothing.
    */
    /**/
    private void Start()
    {
        UpdateTotalPointsDisplay();
        UpdateUnlockButtons();
    }

    /**/
    /*
    SelectTrackMenu::UpdateTotalPointsDisplay()
SelectTrackMenu::UpdateTotalPointsDisplay()

    NAME
        SelectTrackMenu::UpdateTotalPointsDisplay - Updates the display of total
points.

    SYNOPSIS
        void SelectTrackMenu::UpdateTotalPointsDisplay();

    DESCRIPTION
        This method updates the text display showing the total points available to
the player.

    RETURNS
```



```
        Nothing.
    */
    /**/
    public void UpdateTotalPointsDisplay()
    {
        if (totalPointsText != null)
        {
            totalPointsText.text = "Total Points: " +
GameManager.instance.totalPoints.ToString();
        }
    }

    /**/
    /*
    SelectTrackMenu::UpdateUnlockButtons() SelectTrackMenu::UpdateUnlockButtons()

    NAME
        SelectTrackMenu::UpdateUnlockButtons - Updates the state of unlock buttons
for tracks.

    SYNOPSIS
        void SelectTrackMenu::UpdateUnlockButtons();

    DESCRIPTION
        This method updates each unlock button in the track selection menu. It
checks if each track
        is unlocked or if the player has sufficient points to unlock a track.

    RETURNS
        Nothing.
    */
    /**/
    private void UpdateUnlockButtons()
    {
        for (int i = 0; i < unlockButtons.Length; i++)
        {
            // Hide the button if the track is unlocked or the player has enough
points
            bool isUnlocked = GameManager.instance.IsTrackUnlocked(i);
            unlockButtons[i].SetActive(!isUnlocked &&
GameManager.instance.totalPoints < 1000);
        }
    }

    /**/
    /*
    SelectTrackMenu::UnlockTrack(int trackID) SelectTrackMenu::UnlockTrack(int trackID)

    NAME
        SelectTrackMenu::UnlockTrack - Unlocks a specified track.

    SYNOPSIS
        void SelectTrackMenu::UnlockTrack(int trackID);
            trackID   --> The ID of the track to unlock.

    DESCRIPTION
        This method is called to unlock a specific track. It checks if the player has enough points
        to unlock the track. If sufficient points are available, the track is unlocked via the GameManager,
        and the UI is updated to reflect the new unlock state and total points.

    RETURNS
        Nothing.
    */
    /**/
    public void UnlockTrack(int trackID)
    {
        if (GameManager.instance.totalPoints >= 100)
        {
            GameManager.instance.UnlockTrack(trackID);
            UpdateUnlockButtons();
            UpdateTotalPointsDisplay();
        }
        else
        {
            Debug.Log("You need at least 100 points to unlock this track.");
        }
    }
    /**/
    /*
    SelectTrackMenu::StartTrack(int trackID) SelectTrackMenu::StartTrack(int trackID)

    NAME
        SelectTrackMenu::StartTrack - Initiates the loading of a selected track.

    SYNOPSIS
        void SelectTrackMenu::StartTrack(int trackID);
            trackID   --> The ID of the track to be loaded.

    DESCRIPTION
        This method is responsible for starting a race on the selected track. It constructs the scene name
        based on the provided track ID and then uses the SceneManager to load the corresponding level scene.

    RETURNS
        Nothing.
    */
    /**/
    public void StartTrack(int trackID)
    {
        string trackName = "Level-" + trackID;

        SceneManager.LoadScene(trackName);
    }
}
```

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

/**/
/*
    This class contains initialization logic that runs when the game loads, prior to the first scene being loaded.
    Its primary function is to instantiate a set of predefined GameObjects that are required to be present from the very
    beginning of the game.
*/
/**/
public class Startup
{
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]

    /*
    Startup::InstantiatePrefabs() Startup::InstantiatePrefabs()

    NAME
        Startup::InstantiatePrefabs - Instantiates prefabs when the game loads.

    SYNOPSIS
        [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
        public static void InstantiatePrefabs();

    DESCRIPTION
        This static method loads and instantiates a set of predefined GameObjects from the "InstantiateOnLoad" resources folder.

    RETURNS
        Nothing.
    */
    public static void InstantiatePrefabs()
    {
        GameObject[] prefabsToInstantiate = Resources.LoadAll<GameObject>("InstantiateOnLoad/");

        foreach (GameObject pref in prefabsToInstantiate)
        {
            GameObject.Instantiate(pref);
        }
    }
}
```

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

/**/
/*
    This class represents different types of surfaces that can be encountered in the game, such as road, grass, sand, etc.
    Each surface type has a different effect on the cars' speed.
*/
/**/
public class Surface : MonoBehaviour
{
    public enum SurfaceTypes { Road, Grass, Sand, Mud };

    [Header("Surface")]
    public SurfaceTypes surfaceType;

}
```

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

/**/
/*
    This class is responsible for detecting and managing the type of surface a car is driving on in the game.
    It uses physics collision detection to determine the surface underneath the car and updates the current surface type accordingly.
*/
/**/
public class SurfaceHandler : MonoBehaviour
{
    [Header("Surface Detection")]
    public LayerMask surfaceLayer;

    Collider2D[] surfaceCollidersHit = new Collider2D[10];
    Vector3 lastSurfacePosition = Vector3.one * 10000;

    Surface.SurfaceTypes onSurface = Surface.SurfaceTypes.Road;

    Collider2D carCollider;

    /**/
    /*
    SurfaceHandler::Awake() SurfaceHandler::Awake()

    NAME
        SurfaceHandler::Awake - Initializes the SurfaceHandler component.

    SYNOPSIS
        void SurfaceHandler::Awake();

    DESCRIPTION
        This method initializes the SurfaceHandler by finding and storing the
Collider2D component of the car.

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        carCollider = GetComponentInChildren<Collider2D>();
    }

    /**/
    /*
    SurfaceHandler::Update() SurfaceHandler::Update()

    NAME
        SurfaceHandler::Update - Updates the car's surface detection.

    SYNOPSIS
        void SurfaceHandler::Update();

    DESCRIPTION
        This method checks the car's position and detects the surface type the car
is currently on.
```

```
        If the car has moved sufficiently, it updates the 'onSurface' variable to
reflect the new surface type.

    RETURNS
        Nothing.
    */
    /**/
    void Update()
    {
        if ((transform.position - lastSurfacePosition).sqrMagnitude < 0.75f)
            return;

        ContactFilter2D contactFilter2D = new ContactFilter2D();
        contactFilter2D.layerMask = surfaceLayer;
        contactFilter2D.useLayerMask = true;
        contactFilter2D.useTriggers = true;

        int numOfHits = Physics2D.OverlapCollider(carCollider, contactFilter2D,
surfaceCollidersHit);

        float lastSurfaceValue = -1000;

        for (int i = 0; i < numOfHits; i++)
        {
            Surface surface = surfaceCollidersHit[i].GetComponent<Surface>();

            if (surface.transform.position.z > lastSurfaceValue)
            {
                onSurface = surface.surfaceType;
                lastSurfaceValue = surface.transform.position.z;
            }
        }

        if (numOfHits == 0)
            onSurface = Surface.SurfaceTypes.Road;

        lastSurfacePosition = transform.position;
    }

    /**/
    /*
    SurfaceHandler::GetCurrentSurface() SurfaceHandler::GetCurrentSurface()

    NAME
        SurfaceHandler::GetCurrentSurface - Retrieves the current surface type.

    SYNOPSIS
        public Surface.SurfaceTypes GetCurrentSurface();

    DESCRIPTION
        This method returns the type of surface the car is currently on. It is used
to adjust various
        aspects of the car's behavior, such as handling and sound effects,
depending on the surface type.

    RETURNS
        The current surface type the car is on.
    */
    /**/
    public Surface.SurfaceTypes GetCurrentSurface()
    {
        return onSurface;
    }
}
```

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

/**/
/*
    This class is responsible for managing the display of race time in the game's
user interface.
    It continuously updates a UI text element to show the elapsed time in minutes
and seconds format.
    This class also uses a coroutine to efficiently update the time display at
regular intervals,
    ensuring the time shown is current and accurate.
```

```
*/
/**/
public class TimeUIHandler : MonoBehaviour
{
    Text timeText;

    float lastRaceTimeUpdate = 0;

    /**/
    /*
    TimeUIHandler::Awake() TimeUIHandler::Awake()

    NAME
        TimeUIHandler::Awake - Initializes the TimeUIHandler component.

    SYNOPSIS
        void TimeUIHandler::Awake();

    DESCRIPTION
        This method is responsible for initializing the TimeUIHandler component, primarily by obtaining
        a reference to the Text component which will be used to display the race time on the UI.

    RETURNS
        Nothing.
    */
    /**/
    private void Awake()
    {
        timeText = GetComponent<Text>();
    }

    /**/
    /*
    TimeUIHandler::Start() TimeUIHandler::Start()

    NAME
        TimeUIHandler::Start - Starts the coroutine for updating the race time display.
    SYNOPSIS
        void TimeUIHandler::Start();

    DESCRIPTION
        This method triggers the start of the coroutine UpdateTimeCO(), which continually updates the race time displayed on the UI.

    RETURNS
        Nothing.
    */
    /**/
    void Start()
    {
        StartCoroutine(UpdateTimeCO());
    }

    /**/
    /*
    TimeUIHandler::UpdateTimeCO() TimeUIHandler::UpdateTimeCO()

    NAME
        TimeUIHandler::UpdateTimeCO - Continuously updates the race time display.

    SYNOPSIS
        IEnumerator TimeUIHandler::UpdateTimeCO();

    DESCRIPTION
        This coroutine is responsible for regularly updating the race time display in the UI.
        It queries the current race time from the GameManager and updates the text component to
        reflect the elapsed minutes and seconds.

    RETURNS
        IEnumerator that temporarily halts execution to properly update the timer text.
    */
    /**/
    IEnumerator UpdateTimeCO()
    {
        while (true)
        {
            float raceTime = GameManager.instance.GetRaceTime();

            if (lastRaceTimeUpdate != raceTime)
            {
                int raceTimeMinutes = (int)Mathf.Floor(raceTime / 60);
                int raceTimeSeconds = (int)Mathf.Floor(raceTime % 60);

                timeText.text =
$"{raceTimeMinutes.ToString("00")}:{raceTimeSeconds.ToString("00")}";

                lastRaceTimeUpdate = raceTime;
            }

            yield return new WaitForSeconds(0.1f);
        }
    }
}
```

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

/**/
/*
    The TrailHandler class is responsible for managing the visual trail effects for
the cars.
    The class listens to the car's drifting and braking status, and activates or
deactivates the trail emission accordingly.
*/
/**/
public class TrailHandler : MonoBehaviour
{
    CarController carController;
    TrailRenderer trailRenderer;

    /**/
    /*
    TrailHandler::Awake() TrailHandler::Awake()

    NAME
        TrailHandler::Awake - Initializes the TrailHandler component.

    SYNOPSIS
        void TrailHandler::Awake();

    DESCRIPTION
        This method initializes the TrailHandler by finding and storing the
CarController and TrailRenderer components.
        The TrailRenderer is set to not emit at the start, and will be activated
when certain conditions in the Update method
        are met, such as when the car is drifting.

    RETURNS
        Nothing.
    */
    /**/
    void Awake()
    {
        carController = GetComponentInParent<CarController>();

        trailRenderer = GetComponent<TrailRenderer>();

        trailRenderer.emitting = false;
    }

    /**/
    /*
    TrailHandler::Update() TrailHandler::Update()

    NAME
        TrailHandler::Update - Updates the trail emission based on car's behavior.

    SYNOPSIS
        void TrailHandler::Update();

    DESCRIPTION
```

```
        This method checks if the car is currently drifting or braking using the
CarController's IsTireDrifting method.
        If the car is drifting, it activates the trail renderer to emit trails,
otherwise it stops the emission.

    RETURNS
        Nothing.
    */
    /**/
    void Update()
    {
        if (carController.IsTireDrifting(out float lateralVelocity, out bool
isBraking))
        {
            trailRenderer.emitting = true;
        }
        else
        {
            trailRenderer.emitting = false;
        }
    }
}
```

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

/**/
/*
    This class is a key component for guiding AI-controlled cars along the track.
Each waypoint represents
    a position on the track that AI cars aim to reach, effectively forming a path
for them to follow.
    Waypoints can influence AI behavior by specifying parameters like the maximum
speed AI cars should aim for
    when approaching a waypoint, and the minimum distance at which the waypoint is
considered reached.
*/
```

```
/**/
public class Waypoint : MonoBehaviour
{
    [Header("Waypoints")]
    public float maxSpeed = 0;

    public float minDistance = 5;

    public Waypoint[] nextWaypoint;
}
```

TESTING

Game Initialization and Loading:

- Test game startup, ensuring all initial screens load correctly.
- Verify loading of different game scenes, including menus, car selection, and track selection.
- Check for proper initialization of game settings (e.g., default AI difficulty, volume settings).

Car and Track Selection:

- Validate the functionality of car selection, ensuring all available cars can be selected and viewed.
- Test the unlocking mechanism for cars and tracks, ensuring point deduction and unlocking are functioning.
- Confirm that locked cars and tracks cannot be selected until unlocked.

User Interface:

- Test navigation through various menus (main menu, options, car selection, track selection).
- Validate the functionality of UI elements like menu buttons, volume slider, etc.
- Check for visual consistency and proper display of UI elements across different screens.

In-Game Functionality:

- **Racing Mechanics:** Test car movement, handling, item usage, and collision responses.
- **Race Progression:** Ensure correct lap counting, checkpoint functioning, and accurate race completion.
- **AI Behavior:** Test AI cars for proper navigation, item usage, and reaction to the player and environment.

Game Performance:

- Monitor game performance for any lag, glitches, or crashes, particularly during races with multiple AI opponents.

Post-Race Processes:

- Verify accurate scoring and point allocation based on race results.
- Test the display of post-race results, ensuring all relevant information is correctly shown.

Error Handling and Messaging:

- Test the game's response to various error conditions using debug logs.

Feedback Loop:

- Allow other family members to play the game and gain feedback on their experience with it to gain insights on possible improvements and bugs.

Conclusion

When I was deciding on what to do for my Senior Project, I knew I wanted to do something that would be fun to develop. This has led me to think back on some childhood games I have played, such as Mario Kart and Kirby Air Ride, which ultimately inspired me to develop a racing game. I also figured that a racing game shouldn't be too challenging or time consuming to create, but I would learn not too long after that wasn't the case. Nevertheless, I still enjoyed working on this project and the experience has helped me to gain a better understanding of Unity and learn a lot more about C# programming. I had to put a ton of hours into researching and experimenting with the various features and functions they had to offer. The project also helped me learn a lot more about the game development process and how time consuming it can really be. I gained a deeper appreciation for the intricacies of designing and implementing game mechanics, and the importance of balancing gameplay elements.

Despite the enjoyment, there are some improvements that I wish I could make to the development of this project. One improvement would have been to do more research before I began working on the project as I underestimated the difficulty of creating a racing game like this. While I did create a flowchart to plan the game's menus and processes, having a more detailed diagram would have been a big help. The project also took much longer to develop than I had originally thought due to the large amount of classes and assets that I needed to create and manage for the game to function. This is more evident when looking at the wishlist of features that I wasn't able to incorporate into the project, which included:

- Online multiplayer functionality where multiple players can connect and race each other.
- Alternative game modes aside from regular races, such as time trials, elimination races, capture the flag, etc.
- Unique attributes for each car with varying speeds and handling capabilities.
- The ability to purchase upgrades for your cars, such as increased top speeds, faster turns, longer speed boosts, etc.
- A save feature that can allow players to save their progress

I also wish that I had done more pre-planning for the project. Oftentimes, I didn't have all the assets I needed, such as sprites, while I was creating the game, which required me to halt development and search online for the free asset I needed based on the task that I was working on at the moment, which could sometimes take a little while.

Overall, this project has given me the chance to experience a journey of discovery and learning, filled with challenges and rewarding achievements. It has not only advanced my technical skills, but also provided valuable lessons in game design, project management, and problem-solving. The completion of this project marks a significant milestone in my academic and professional development and has helped to prepare me for future endeavors in my career.

REFERENCES

1. “Unity User Manual 2022.3 (LTS).” *Unity*, docs.unity3d.com/Manual/UnityManual.html.

2. Coding in Flow. “Build a 2D Platformer Game in Unity | Unity Beginner Tutorial.” *YouTube*, YouTube, www.youtube.com/playlist?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U.

3. Pandemonium. “Unity 2D Platformer for Complete Beginners.” *YouTube*, YouTube, www.youtube.com/playlist?list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV.

4. “Waypoints.” *Unity Learn*, learn.unity.com/tutorial/waypoints.

5. “Get Started with the A* Pathfinding Project.” *A* Pathfinding Project: Main Page*, arongranberg.com/astar/docs_dev/index.php.

6. “C# Docs - Get Started, Tutorials, Reference.” *C# Docs - Get Started, Tutorials, Reference. | Microsoft Learn*, learn.microsoft.com/en-us/dotnet/csharp/.

7. Vegetarian Zombie. “Beginning C# with Unity.” *YouTube*, YouTube, www.youtube.com/playlist?list=PLFgjYYTq6xyhtVK6VzLiFe3pmBu-XSNlX.

8. Miller, Victor. “CMPS 357 .Net Environment.” *Ramapo College of New Jersey*, pages.ramapo.edu/~vmiller/DotNET/index.htm.