3D Traffic Modeling in Unity

Group 2:

Isaiah Martinez
CSUN
Computer Science Department
isaiah.martinez.891@my.csun.edu

Jae Molina
CSUN
Computer Science Department
jae.molina.499@my.csun.edu

Anastasia Naydina
CSUN
Computer Science Department
anastasia.naydina.947@my.csun.edu

05/13/2024

Contents

1 Change History

Version: 0.42
Modifier: Isaiah Martinez
Date: 4/21/24
Description of Change: Finished TomTom API using Python. Made Python Script accessible via Command line.

Version: 0.19
Modifier: Isaiah Martinez
Date: 3/29/24
Description of Change: Added API for TomTom to obtain Images of Traffic Flow. Demo car scene implemented. Pathfinding added.

Version: 0.11
Modifier: Isaiah Martinez, Jae Molina, Anastasia Naydina
Date: 2/26/24
Description of Change: Simple Car model made. Looked at Related Works for process in utilizing Unity for traffic modeling. Looking for additional related works.

Version: 0.09
Modifier: Isaiah Martinez, Jae Molina, Anastasia Naydina
Date: 2/19/24
Description of Change: Discussed High Level Architecture of the project: Unity for modeling, C#/Python for helper script, Python for ML training, and JS for API connectivity. Added template to follow for documentation. Structured git repo directories. First, we will be working on a set amount of locations with small amount of available traffic data. Later, we hope to implement API connectivity to obtain traffic info and map data with more locations.

Version: 0.05
Modifier: Isaiah Martinez, Jae Molina, Anastasia Naydina
Date: 2/12/24
Description of Change: Made Git repository. Looked at scholarly articles for related works. Uploaded sample scholarly article to view. Laid out big ideas for project. Began work on models to be used in Unity.

2 Introduction

This document describes the work put in to create a program that runs in Unity to model traffic in 3D. It allows for users to determine a start position, as well as a desired ending position which will follow traffic rules to ensure a fast and legal guide.

The code is able to be viewed on Github at this Link.

2.1 Background

Traffic congestion is a very common experience for those who live in Los Angeles [1]. In order to address this problem many apps such as Waze and Google Maps were created [2]. Following in the style of these applications, we wanted to create software that follows a similar purpose: to provide a good model for traffic flow and path to a destination with this information.

2.2 Related Works

We were interested in trying to model a similar outcome as outlined in this paper [3]. In order to achieve this, we would utilize Unity, a 3rd party package called EasyRoads [6], MLAgents [5], and an external source for the traffic data. To obtain the traffic data, we would utilize real-time traffic information from TomTom [7]. We initially were hoping to utilize ML Learning Agents to do something similar to what was done in this Youtube video [4]. In the end we were decided against implementing ML Learning Agents for our project, as we would utilize Unity's built-in pathfinding capabilities.

2.3 Stakeholders

The Major Stakeholders are:

1. Users: they want a product that will work by giving accurate and detailed information of traffic. Additionally, they want the data to be useful for determining the route to take from their source destination to the desired ending destination.

2. Developers: they want to have a simplistic model for creating, modifying, and debugging the project across its various languages, scripts, and so forth. Development would be made easy by utilizing professional tools, technologies, and features.

2.4 Document Structure

This document contains a series of 3 main sections:

1. Design Goals: This section outlines the main objectives and guiding principles that drive the design of the software project. It provides a high-level vision of what the software aims to achieve in terms of functionality, usability, performance, and other key aspects.

2. System Behavior: This section describes how the system behaves as a whole and its overall functionality. It should give readers a clear understanding of the system's expected actions and reactions in different scenarios.

3. Logical View: The logical view provides a conceptual model of the software's architecture and its components. This includes how the software is organized and how its different parts work together.

4. Scenario View: The scenario view illustrates how the system behaves in specific use cases or scenarios. It helps to show how the system meets the requirements in practice.

We will explain the project utilizing UML diagrams, User-Interaction Diagrams, Class Diagrams, and other commonly used professional Software Development Models. This is to ensure that out project is explained with the utmost clarity, and conciseness.

2.5 Responsibilities

Isaiah: API connectivity. Obtained the Traffic Data and Map Data through the use of a Python Script and TomTom's developer API. Added ML Agents to the unity project.

Jae:

Anastasia:

3 Design Goals

3.1 Purpose

The purpose of this software project is to create a simulation within Unity to recreate real-time traffic scenarios. By focusing on real-time traffic data collection, different pathfinding algorithms, with visualization help of Unity, the project aims to provide users with a platform for comparing simulated traffic behaviors to real-life situations.

3.2 Objectives

Our initial goals for the project included:

1. Real-Time Traffic Data Collection: Collect real-time data from satellite imagery to construct maps for the simulation.

2. Pathfinding: Implementing pathfinding algorithms to simulate realistic vehicle movements and route optimization within the traffic simulation.

3. Rendering in Unity with 3D Objects: Utilizing Unity's to create visual representations of the simulated traffic scenarios.

Were they met? Why/Why not?

Due to the project's complexity and time constraints, some of our initial goals were not fully met

1. Real-Time Creation of Map with Data: While we did integrate real-time data collection mechanisms, such as traffic APIs through TomTom, into the project, the process of dynamically constructing maps within the simulation environment proved more challenging than originally believed. As a result, the maps were often pre-generated and simplified representations rather than real-time creations.

2. Real-Time Comparison to Traffic: Achieving real-time comparison to real-life traffic was a significant challenge. Despite implementing a pathfinding algorithm and system, accurately recreating the complexities of real-world traffic behaviors within the Unity environment proved to be beyond the project's scope within the given timeframe.

3. Different Pathfinding Algorithms: While we implemented several pathfinding algorithms, such as Dijkstra's algorithm and A* search, integrating a wide range of algorithms for comparative analysis was not feasible within the project's constraints.

However the project did accomplish in other main areas

1. Visualization: Overall, the visualization and creation of the different assets for the project in order to show a visually appealing simulation to the user

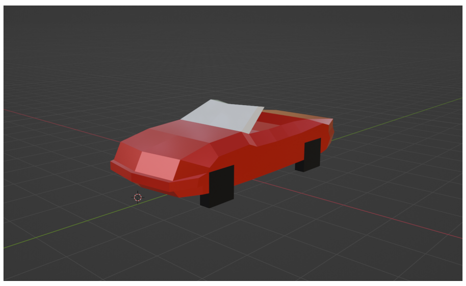


Figure 1: Sports Car rendered in Blender

2. Map Creation: With the use of the TomTom API, the project was able to have a few select scenes of nearby areas and which were digitally recreated.

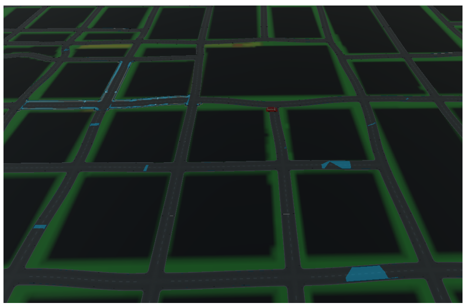


Figure 2: Map rendered in Unity

3. Main Pathfinding: The overall scoring of the pathfinding is interesting enough and accurate to normal behavior of traffic. As more vehicles enter the road, congestion will occur and the overall speed to reach the goal is reduced heavily.

Comparison to Initial Vision Despite falling short, the finished project represents a significant achievement in terms of simulating real-time traffic within a Unity. While compromises were made due to complexity and time constraints, the project still provides valuable insights into traffic simulation and serves as a foundation for future developments to the project.

Below is a sequential diagram for the design of the code and classes running inside Unity. The main relationships are between the UI, which are the classes UIManager and PathSet, and the spawns like SpawningCars, SpawnPF, and PathFind that makes the pathfinding algorithm for the main car object.

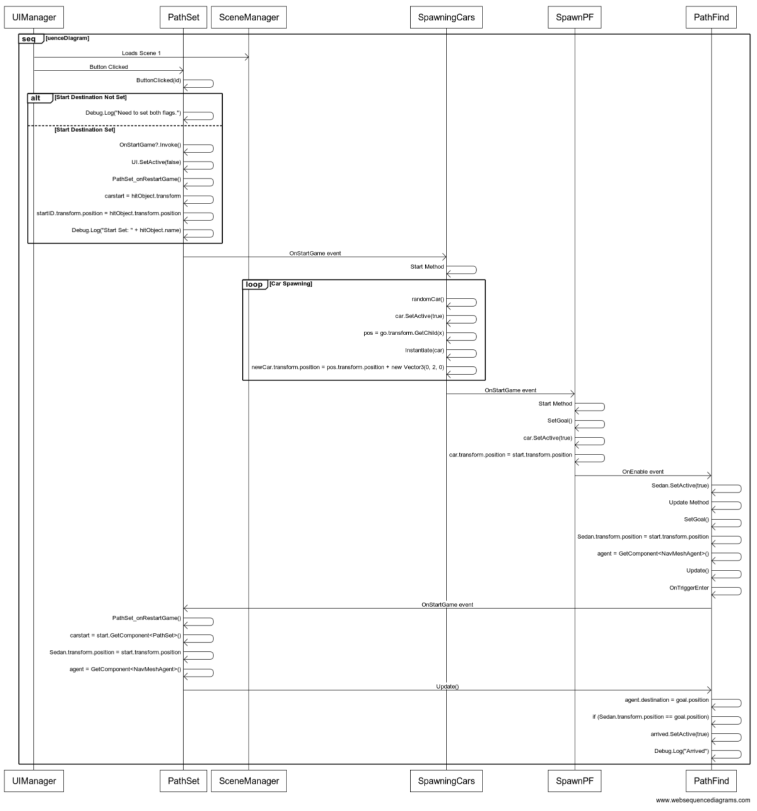

UIManager
PathSet
SceneManager
SpawningCars
SpawnPF
PathFind
seq [uenceDiagram]
Loads Scene 1
Button Clicked
ButtonClicked(id)
alt [Start Destination Not Set]
Debug.Log("Need to set both flags.")
[Start Destination Set]
OnStartGame?.Invoke()
UI.SetActive(false)
PathSet_onRestartGame()
carstart = hitObject.transform
startID.transform.position = hitObject.transform.position
Debug.Log("Start Set: " + hitObject.name)
OnStartGame event
Start Method
loop [Car Spawning]
randomCar()
car.SetActive(true)
pos = go.transform.GetChild(x)
Instantiate(car)
newCar.transform.position = pos.transform.position + new Vector3(0, 2, 0)
OnStartGame event
Start Method
SetGoal()
car.SetActive(true)
car.transform.position = start.transform.position
OnEnable event
Sedan.SetActive(true)
Update Method
SetGoal()
Sedan.transform.position = start.transform.position
agent = GetComponent<NavMeshAgent>()
Update()
OnTriggerEnter
OnStartGame event
PathSet_onRestartGame()
carstart = start.GetComponent<PathSet>()
Sedan.transform.position = start.transform.position
agent = GetComponent<NavMeshAgent>()
Update()
agent.destination = goal.position
if (Sedan.transform.position == goal.position)
arrived.SetActive(true)
Debug.Log("Arrived")
UIManager
PathSet
SceneManager
SpawningCars
SpawnPF
PathFind
www.websequencediagrams.com

Figure 3: Sequential class diagram.

4 System Behavior

This section describes how the system behaves as a whole and its overall functionality. It should give readers a clear understanding of the system's expected actions and reactions in different scenarios.

4.1 General Overview

The project is a navigation simulation where a user can place a starting position and ending position on a premade map based on a real life location (streets of city of Northridge) and watch the player's object car navigate from the starting position to the end with pathfinding. The player's car will have to navigate with other cars on the map that are going to their own locations to simulate traffic. We are able to pull a live-time image of current traffic and the streets through the TomTom API, which we used to build the premade map and have in the program. The cars were modelled and there are 5 different types of cars that can spawn.

4.2 Key Features and Functionality

The program starts with a main menu, a Canvas with the title text and a button for the Northridge map scene. We can expand this menu with adding more map scenes for different areas and streets, but initially we wanted to use the TomTom API to pull a live image representing traffic that would be loaded onto the map. The user would input any location they want, and we would have a python code run through the image returned by the TomTom API for that location and make the 3d road scene for that location. Unfortunately, the issue we ran into here was that there is no distinction between highways and streets on the image, and there was no information on any traffic parts like street lights, stop signs, etc. Instead, we decided to have these presets and build the 3d map ourselves.

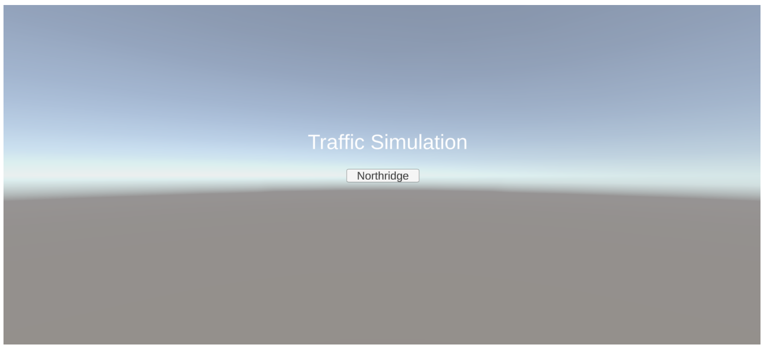


Figure 4: The main menu.

Once the user clicks on the Northridge button, the program switches to the Northridge scene that we made. Here, the user can choose a starting point for the car and an ending point for the car. This is achieved by the buttons "Start Marker" and "End Marker" on the side. Clicking on the Start Marker button then at a road on the map will place a green orb that represents the starting position. Click on the End Marker button then at a road on the map wll place a red orb that represents the ending position.

When the user places both markers, then they can click the "Run" button to spawn their car on the road and start the simulation. Once the program starts

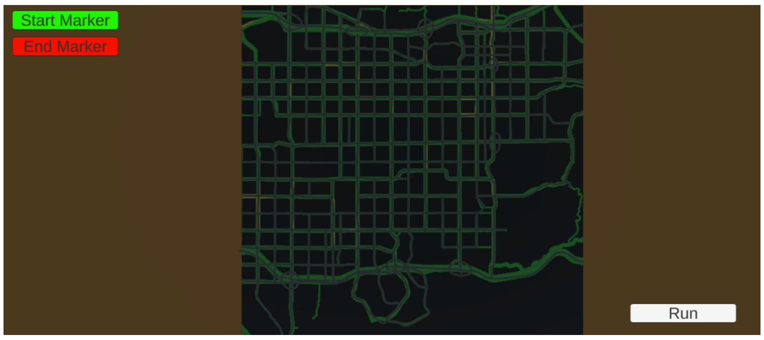


Figure 5: The initial map and buttons "Start marker", "End Marker", and "Run".

running, the pathfinding class inside the car kicks in once it turns active at the event the users clicks run and the pathfinding will lock towards the positon of the End Position that the user made. The car will navigate towards the end position which it considers its goal, then stop there. Along the way, there can be car obstacles. All of the cars have a Nav Mesh Agent attached to them, and the cars that aren't the one the users chooses a start and end position for move towards randomized locations. When they reach that location, the goal position randomizes again and they move towards that one next. The spawned cars move on forever until the user stops the game.The cars are able to move along the roads because of the AI and Pathfinding Unity package that allows the roads to be "baked", which lets the car objects know which surfaces they can use pathfinding on and which they can't. Surfaces that can be traversed on need to be marked as static, hence all the road objects are static objects.

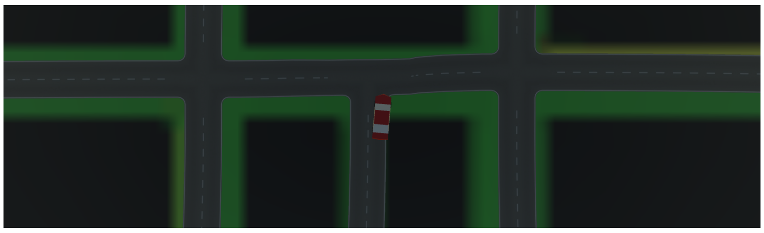


Figure 6: The camera zooms in and follows the main car as it navigates to the end position.

The program uses the AI and Pathfinding packages in Unity, which need to be seperately downloaded. The UI Manager class handles the scene switch from the main menu with the maps, then how the user can choose a starting position and ending position as discussed earlier. Here, the program takes in the button id of the button clicked, which are ids for the starting point, ending point, and run button. The run button will not start unless both the start and end destinations have been set by the user, there are no default positions that can be run if the user doesn't put in anything. The UI has the class PathSet within it as well, where it will take in the position of where the user set the

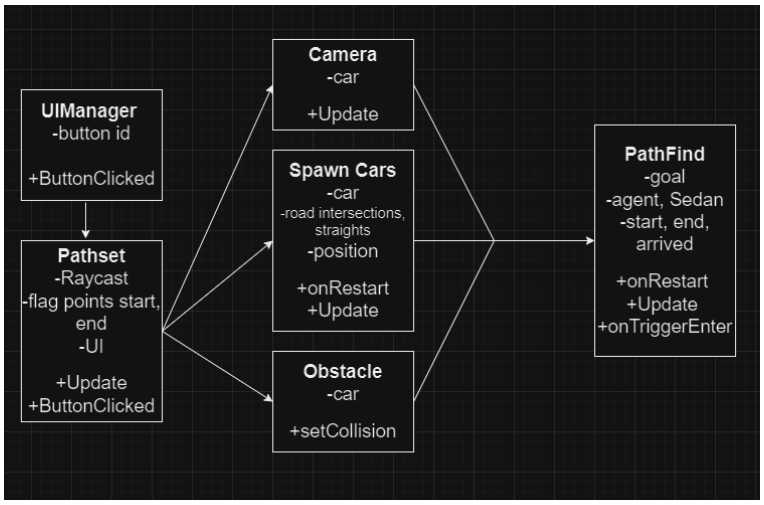


Figure 7: A UML Diagram for thec classes working inside the program.

start position and the end position considered "flag points start, end" from a raycast sent from the user's click onto a road. The road's position is sent back from the raycast, then the flag point is set there. Whichever flag point it is is determined by the button id from the UIManager.

Once the positions have been set and the user chooses to run the program, three classes run simultaneously to set the scene. Firstly, the camera object is set to follow the car object with an offset, where update will move the camera onto the car with the offset to make it follow it the whole way. Secondly, the spawn cars class instantiates prefabs of cars onto the map. How the code handles this is setting an array of the different type of car prefabs that a random range of 0-4 chooses from, then that car is instantiated onto a random position of a road intersection. Initially, we wanted cars to have the option to spawn onto either a straight road or on an intersection (hence the road intersections, straights) however the straight roads that are built into the map have a local position of (0,0,0). Taking in the object's position would just return (0,0,0), then all cars would spawn from that point of the map globally regardless of if randomizing which road straight they spawn from has been successfully initialized. This is why instead we are spawning cars from just road intersections, as road intersections have global transform positions attached to them instead of the local positions the road straights have. There was a lot of learning about parent and child relationships with this project. The onRestart represents the event that the pathFind will wait for the cue for. The obstacle class set the nav mesh agents on all the spawned cars that they can recognize eachother as collisions or obstacles that no car can phase through one another. After those classes have run, then the onRestart event is called the the PathFind finally runs.

The Pathfind takes in the end position or goal that has been set by the user, and initializes a nav mesh agent to the Sedan that the main player navigates. It also takes in the start position where the Sedan will start from, the end position where the Sedan will go towards, and the arrived that is the end position that the Sedan will stop when it gets to its destination. At Update, the agent's destination moves towards the end position working with pathfinding. The onTriggerEnter should make the Sedan recognize that it has reached it's destination and let a "You've arrived" text pop up, but the position most likely isn't set correctly for the Sedan to recognize it's arrived which is a continuing issue that can be fixed, just ran out of time. The user's Sedan is conscious of other cars and avoids collisions because every car is set with a nav mesh agent (to which, human error is also another component of traffic that should be consideredin a simulation) and moves around them or drives behind them slowly at a distance.

Improvements we can make to the system could be some way to organize the positions better that they can be easier taken, have an option for the user to choose positions outside of the road objects and return the nearest road object to the marker, and make the navigation a little more restrictive that cars don't drive in the middle of the road.

5 Logical View

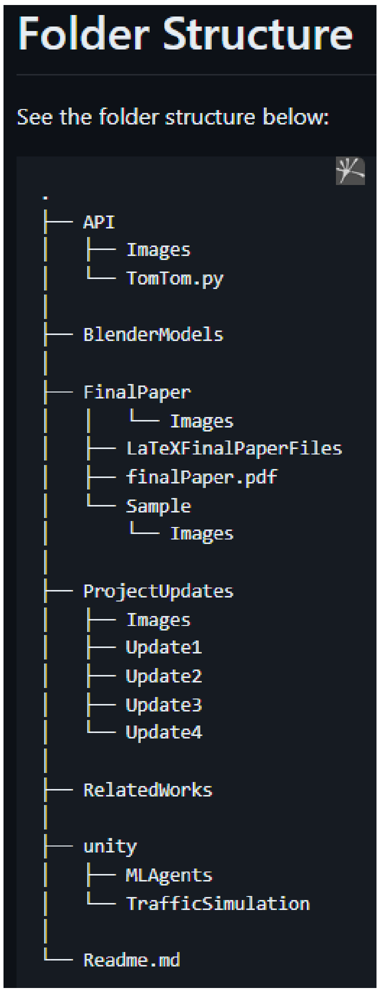
The logical view provides a conceptual model of the software's architecture and its components. This includes how the software is organized and how its different parts work together. This will be the largest section of the document.

5.1 High-Level Design (Architecture)

Another diagram be a layered image for each section of the project from unity to connecting to the python script, pathfinding, etc. each layer should be explained as well.

5.1.1 TomTom.py

Below is a class diagram of the TomTom API.



Folder Structure

See the folder structure below:

```
.
├── API
│   ├── Images
│   └── TomTom.py
│
├── BlenderModels
│
├── FinalPaper
│   │   └── Images
│   ├── LaTeXFinalPaperFiles
│   ├── finalPaper.pdf
│   └── Sample
│       └── Images
│
├── ProjectUpdates
│   ├── Images
│   ├── Update1
│   ├── Update2
│   ├── Update3
│   └── Update4
│
├── RelatedWorks
│
├── unity
│   ├── MLAgents
│   └── TrafficSimulation
│
└── Readme.md
```

Figure 8: Folder Structure of the project.

5.2 Detailed Class Design

should show a list of all functions and classes used to create the project
all classes/functions should have: -Inputs -Purpose -Output

5.2.1 TomTom.py

- extractIMG
  - **Inputs**: (*String*) HTML link, (*Boolean*) Save the image if true
  - **Purpose**: given the HTML link, connects to the TomTom API to obtain the desired image. If instructed, will also save the image in the local folder: './API/Images'
  - **Output**: N/A
- boundChecker

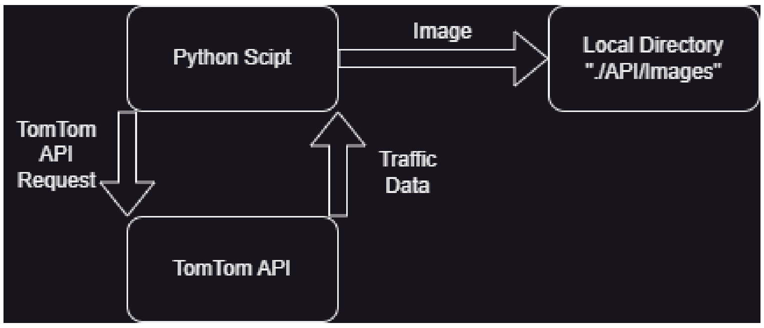


Figure 9: Hierarchy of the script.

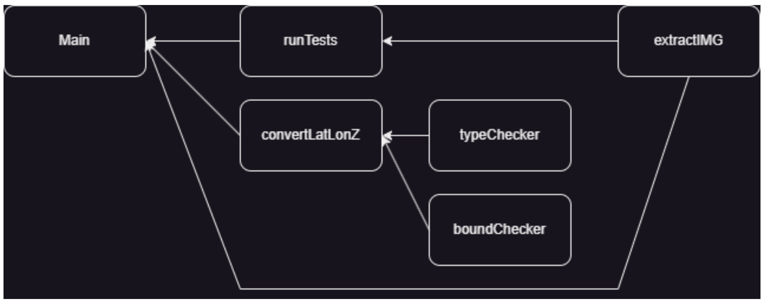


Figure 10: Hierarchy of the script.

  - **Inputs**: (*Float*) Latitude, (*Float*) Longitude, (*Int*) Zoom level, (*String*) Style of image, (*Boolean*) Verbose option. If true prints to console about checking bounds for given inputs
  - **Purpose**: Ensures proper bounds on the given inputs
  - **Output**: (*Boolean*) True means that the inputs are properly within the bounds

- typeChecker
  - **Inputs**: (*Float*) Latitude, (*Float*) Longitude, (*Int*) Zoom level, (*String*) Style of image, (*Boolean*) Verbose option. If true prints to console about checking types for given inputs
  - **Purpose**: Checks the types for all parameters given
  - **Output**: (*Boolean*) True means that the inputs are of the proper type

- convertLatLonZ
  - **Inputs**: (*Float*) Latitude, (*Float*) Longitude, (*Int*) Zoom level, (*String*) Style of image, (*Boolean*) Verbose option. If true prints to console about converting parameters to proper format
  - **Purpose**: Determines the coordinates to use according to the zoom level. This information is needed when accessing the API to obtain the correct location information [8].

    - **Output**: (*Tuple*) Tuple of 3 elements: X, Y, Zoom level

- heightCalc
    - **Inputs**: (*Int*) Red value of pixel, (*Int*) Green value of pixel, (*Int*) Blue value of pixel
    - **Purpose**: Determines the height of a given pixel given the RGB values of the image. This formula is from TomTom [9]. Unused in the final product
    - **Output**: (*Float*) Height of the region described by the pixel as feet above sea level

- runTests
    - **Inputs**: (*Boolean*) Save the images if true, (*Boolean*) Verbose option. Print to the console about each step
    - **Purpose**: Will run a series of tests with default locations to show the different options available for this script
    - **Output**: N/A

- main
    - **Inputs**: (*String*) Latitude, (*String*) Longitude, (*String*) Zoom Level, (*String*) Style of Image to produce, (*Boolean*) Save image created if true, (*String*) Run demo options to showcase all options from this script if true, (*String*) Verbose. If true, describes the steps occurring in the program as they occur
    - **Purpose**: Parses the given arguments, runs through each of the above described functions to achieve the goal of obtaining the image with the given arguments
    - **Output**: N/A

6 Scenario View

The scenario view illustrates how the system behaves in specific use cases or scenarios. It helps to show how the system meets the requirements in practice.

6.1 Use Cases

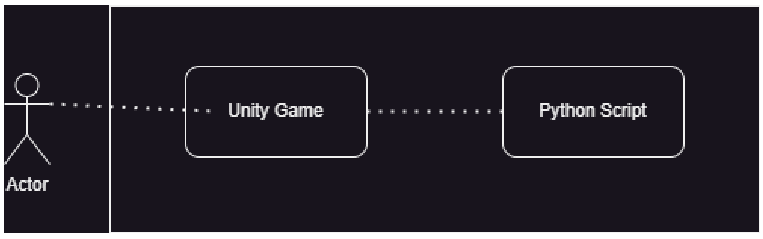


Figure 11: Use Case View.

References

[1] A. L. CNN Illustrations by Natalie Leung, "Los Angeles' traffic problem in graphics," CNN, Feb. 27, 2018. Link

[2] "The 7 Best Traffic Apps of 2023," Lifewire. Link

[3] Teo Niemirepo, J. Toivonen, Marko Viitanen, and J. Vanne, "Open-Source CiThruS Simulation Environment for Real-Time 360-Degree Traffic Imaging," Nov. 2019, doi: https://doi.org/10.1109/iccve45908.2019.8965242.

[4] "Training an unbeatable AI in Trackmania," www.youtube.com. Youtube Link

[5] U. Technologies, "Make a more engaging game w/ ML-Agents — Machine learning bots for game development — Reinforcement learning — Unity," unity.com. Link

[6] "EasyRoads3D Free v3 — 3D Characters — Unity Asset Store," asset-store.unity.com. Unity Store Link

[7] "Traffic APIs," TomTom. Link.

[8] "Zoom Levels and Tile Grid — Map Display API," developer.tomtom.com. Link.

[9] "Hillshade Tile — Map Display API," developer.tomtom.com. Link.