Autonomous vehicle simulation for studying driver distraction

Mohammed Bendahmane

Abstract—In the last decade we have seen a lot of interest in the domain of autonomous vehicle systems. While a lot of research is going in to developing these systems and making partial autonomous driving possible, it is also important to consider the risks that come with this technology, specifically driver distraction. We believe that it is important to have a safe environment where we can study the effects that these autonomous systems have on driver attentiveness. It is for that reason that we decided to create a proof-of-concept simulation software that allows us to study driver distraction in partially automated vehicles. The simulation consists of a 3D environment generated using OpenStreetMap data of real world locations and an autonomous vehicle that a user can interact with by choosing a destination for the vehicle to drive to autonomously. We can then introduce specific test scenarios that mimic real world situations where the human driver has to take manual control of the vehicle in order to avoid an accident. These test scenarios allow us to gauge the driver attentiveness and reaction speed when faced with such situations and therefore allowing us to better understand the problem of driver distraction in autonomous vehicles.

Index Terms—Autonomous Vehicles, driver distraction, Godot engine, OpenStreetMap, Partially automated driving systems, simulation.

1 Introduction

With the rising automation of vehicles, a new problem faces vehicle manufacturers, which is driver distraction. In order to further understand this problem and our proposed solution, first we need to classify the different levels of automation.

According to SAE (Society of Automotive Engineers) [1] there are six levels of vehicle automation [2]. These levels range from level 0 where there is no automation to level 5 where the vehicle is fully automated and can completely drive on its own without the need of a human driver to supervise it.

At automation level 1 (driver assistance) the vehicle offers assistance features, but the driver is still the one responsible of monitoring the driving environment. This means that the driver is required to have almost the same level of concentration as they would at level 0.

Level 2 (partial automation) is similar to level 1, but with even more assistance.

Starting from automation level 3 (conditional automation) we move from human monitored environment to the vehicle being able to monitor the environment on its own. However, at this level the vehicle still isn't fully able to handle all driving situation and at specific timely events it may require urgent human intervention.

At automation level 4 (high automation), even if the driver does not react to events that require human intervention, the vehicle still has some safety measures in order to pull over or avoid situations where human input would be required.

- *E-mail de contacto: 1534144@uab.cat*
- *Mención realizada: Ingeniería de Computadores*
- *Tutor: Lluís Ribas Xirgo*
- *Curso: 2022-2023*

Febrero de 2023, Escola d'Enginyeria (UAB)

The two level of automation where we think that this work can contribute the most are levels 2 and 3, where driver spends considerable amounts of time without having to manually drive the vehicle. Which highly increases the chance of human distraction.

This project aims to create an environment where users can experience these levels of vehicle automation in a safe manner and provide researchers with the tools and data necessary to understand and measure the effects of these distractions.

In order to achieve this, we created a 3D world with vehicles and pedestrians roaming the streets. We then test the effects of driver distraction by letting the user drive around in an autonomous vehicle and subjecting them to various situations where they need to take manual control of the vehicle to prevent causing an accident. This test scenarios allows us to generate data relating to the user's interactions, specifically their awareness and reaction speed, which can then be analysed by experts to help them better understand this problem.

2 Objective

The goal is to create a simulation where a user operates an autonomous vehicle that drives from one location to another and at random points in time the vehicle hands the control to the user, which then has to react in the shortest time possible in order to prevent an accident.

These scenarios can test the driver's attentiveness and reaction speed to see if they are able to handle different situations that requires them to take manual control of the vehicle at specific time intervals.

Such a simulation could be used as a tool to study the effects of autonomous vehicles on driver distraction and provide analytical data, that could be used to improve the safety of autonomous vehicle systems.

3 Sub-objectives

Throughout the development of this project, there are a few sub-objectives that we would like to realize in order to achieve our goals.

The first one is having a basic working simulation that allows us to do various test on the users and gather the information that results from these tests. This simulation consists of multiple systems, including a 3D world which we would construct from OpenStreetMap data, a vehicle that can traverse this world both autonomously and manually and a system that allows us to test the user's awareness.

After finishing this part of the project and gathering relevant data from the users. The goal would be to use various techniques in order to analyse this data and if possible, draw some conclusions form it.

Lastly, in order to make the simulation more appealing and immersive to the users. We would focus on adding more details to the world. These details consist of buildings, additional objects, textures, visual effects, etc.

By having these three elements together, we believe that this work could be useful to study the effects of driver distraction in autonomous vehicles and possibly develop new safety measures that reduce accidents caused primarily by the lack of driver attentiveness.

4 State of the art

There are numerous software solutions and research that have been developed over the years in order to attempt to solve all the different individual obstacles that this project is trying to tackle.

In this section we will look at a few of these works, what they have achieved and how some of them can help or inspire our own project.

When it comes to creating a 3D simulated world there are many solutions that have been developed over the years, one of them is CARLA, which was developed at the CVC (Computer Vision Center), Barcelona in collaboration with other entities. This open-source tool offers the ability to train autonomous driving systems, using a virtual 3D simulation which can be fully controlled using the API that they offer alongside many other tools that facilitate the training of such systems [3].

For creating a 3D world specifically using OSM data there is projects like OSM Buildings which offers a layer of buildings rendered in 3D using data obtained from OpenStreetMap [4]. These buildings contain all the basic metadata relating to their properties, which can be extracted and used for various applications.

On the topic of extracting data from OpenStreetMaps, one popular tool is the Overpass API [5] which allows users to query their database in order to extract OSM data in order to be used for all kinds of projects. The main difference between Overpass API and the main OSM API, is that it is optimized for reading data rather than editing. It allows us to query great amounts of data in a relatively short time, which could help with creating the 3D environment of the simulation in our project.

One important aspect of our project is creating a tool that helps study driver distraction. We can find many studies that try to tackle this same problem, one example is the work presented in [6], which is a study that consists of an experiment where they put thirty-seven participants in a vehicle simulator under different conditions including partially automated and manual driving, with or without non-driving related task. Their objective was to examine how non-driving related task affects driving performance in partially automated vehicles.

There are many organizations investigating whether current manufacturers are taking the problem of driver distraction into account when designing their autonomous driving systems, such as developing driver monitoring systems (DMS) that detect when the driver is not paying attention using cameras, sensors and other devices. One of these organizations is the Insurance Institute for Highway Safety (IIHS) [7], which developed a rating system [8] that evaluates the safety of autonomous driving systems based on how good they are at identifying driver distraction, how they alert drivers once the distraction has been identified and also whether they employ any safety measures when the driver does not react to the alerts.

5 Tools and research

One of the most important tools that will be used over the course of this project is OSM (OpenStreetMap) in order to extract all the necessary data for the 3D world that we want to create. OSM provides us with a ".osm" file which has the same structure as an XML file. This file has different tags that describe all the data it contains, some of this data is not relevant to this project, which means that it is necessary to learn what all these different tags mean. This information can be learned from the OSM wiki [9], which provides detailed explanations for any data that a ".osm" file might contain.

As for the 3D engine, we use the Godot engine, which is an open-source game engine that provides us with all the necessary tools to create a 3D environment and scripting capabilities for all the functionality that our simulation needs.

Although this engine is relatively new compared to other similar engines, it has extensive documentation [10], that provides us with all the necessary information to learn how it works.

Coding in Godot will be done with GDScript, which is Godot's main scripting language. It is an easy to use but powerful scripting language that allows us to do all kinds of 3D manipulation and implement all the functionality that the simulation might need.

6 Tasks

Before we started the project, first we needed to get familiar with all the different tools that we would be using, especially OpenStreetMap and the Godot engine.

This requires reading all the relevant documentation and developing a small-scale project inside the Godot engine that allowed us to test the limits of the engine and get familiar with the structure of the ".osm" files used to generate the 3D environment.

After getting familiar with the environment and all the tools needed for the project and in order to achieve the previously mentioned objectives, the following tasks have been carried out.

For creating the 3D environment, we used real world data gathered by OpenStreetMap. This data describes a 2D map and needs to be parsed in order to extract all the relevant information, which then would be manipulated and used to generate all the different 3D geometry of the environment.

The geometry consists of 3D meshes of roads and buildings which can be placed in the 3D world according to the coordinates given in the OSM data.

In order for the vehicle to drive autonomously from one destination to another, we needed a node graph that represents all the different roads. Which can be used to apply a pathfinding algorithm.

The challenge here was to take all the road information that we parsed from the OSM data and transform it into a node graph.

After that we used a pathfinding algorithm that allows the autonomous vehicle to traverse the world node by node in order to reach a chosen destination.

In the real world this would not be possible, since driving requires more than just following a set of nodes, but for the purpose of this project we used pathfinding on a node graph to simulate autonomous driving.

An important element of the simulation is the autonomous vehicle which is not only capable of following the path provided by the pathfinding system, but also, we needed the user to be able to control it manually in order to achieve the simulation's purpose. Which means that we had to provide all the necessary controls for the user to interact with the vehicle and drive it manually.

Another important aspect is for the user to be able to choose a location for the vehicle to drive to, autonomously. For this we decided to create a 2D top view of the world which would serve the same purpose as what a GPS device would in a real vehicle.

In order to test and evaluate the driver's attentiveness, we wanted to create multiple scenarios that mimic real life situations where the user has to manually control the vehicle in order to avoid causing an accident. These scenarios allow us to generate the necessary data needed to analyse the user's attentiveness and reaction speed.

Finally, in order to have a more polished application we added a graphical user interface which provides the users with all the relevant information that they need.

In the appendix A, we detail the planning and time frames of these tasks.

7 Development

The development of the features of this project requires many techniques, the most important among them are researching and finding the requirements of these features before starting to work on them, after that comes the design stage which allows us to decide what kind of data structures and coding patterns to use and then finally comes the coding stage where most of the development time is spent.

In this section we go over the development of all the components that make up this project and we discuss how they were achieved and some of the problems that we faced and how we got over them.

7.1 Parsing OSM data

In order to extract data from OSM first we need to download a ".osm" file that contains the data of the region that we want to parse. There are multiple ways to download this file. One of the popular ways of doing it, is using the Overpass API which allows for up to 300 MB of uncompressed downloads.

Another way of downloading the region file is by using the OSM website directly and choosing the region using the provided interface. Although this method is limited by only allowing us to download regions with up to 50.000 nodes. For the purpose of our project this is enough, thus we will be using this method.

After downloading the OSM file, which is structured like an XML file, the next step is to find the nodes that make up the elements that we are interested in. For now, these objects are paths and buildings.

In the OSM file we can find different kinds of tags, the most important ones being "node" and "way". The *node* components are the points that make up all the different elements (paths, buildings, etc.). These nodes have three important descriptors, the ID, the latitude and the longitude of the node.

The *way* components represent the elements themselves and for that reason, *ways* are composed of one or more nodes. In addition to that each way has a unique ID and many tags that describe what specific element it represents and it's details.

To extract the paths and buildings we look for the "building" and "highway" tags respectively. Additionally, we can find other tags that better describe these elements, like the number of levels a building has or the type of path which could range from just a sidewalk to a motorway with multiple lanes.

After parsing the whole file, and extracting all the paths and buildings, the next step is going through all the nodes that represent these elements and converting the latitude and longitude data to XY coordinates that we can use to represent these nodes inside the engine or specifically inside the scene where we are going to generate the 3D world.

7.2 Generating 3D meshes

Once we have parsed the necessary data that describes the paths and buildings, we have to use this data to make 3D meshes that allow us to visually represent these elements.

To create these meshes we use the *ArrayMesh* class from the Godot engine's library. This class allows us to create a mesh using an array. This array contains many attributes that we can set, but for now the attribute that we will be using the *ARRAY_VERTEX* which can be set by providing an array of three-dimensional vectors where each one of these vectors represents the XYZ coordinates of a vertex.

After generating the mesh of a single element (path or building) we can then make an object out of this mesh that we can later add to our 3D scene to be visualised.

In order to generate the mesh, we first need to create the vertex array that we will later be passing to the *ArrayMesh* class. This array is made out of vertices which need to be structured in a way that each three consecutive vertices make up a triangle, which means that the final mesh will be made up by these triangles. The challenge lies in dividing the shape of the building or path that we want to create into these triangles.

In the following sections we describe how we divide the shape of these elements into triangles that can later be used to generate the final mesh.

7.2.1 Buildings

In the case of buildings, we use the data that we extracted from the OSM file to find out which nodes make up this building, after that we can make a polygon out of these nodes that represents the area that a specific building occupies.

Once we have this polygon the next step is deciding where to put the walls and the roof of the building. In this case the easiest way to do this, is by creating a wall between each two consecutive nodes of the polygon, then making the roof the same shape as the polygon that represents this building.

After dividing the building into these smaller elements (walls and roof) we then need to divide these elements further in order to obtain the triangles that make up the whole mesh.

In the case of the walls, it's easy, since walls are basically rectangles in this case, we can divide these rectangles diagonally in order to obtain two triangular segments for each wall.

For the roof we need to find a way to divide a polygon into triangles. If all the polygons were convex then this would be easy but for some buildings that's not true, so we need to find a more complex algorithm that can deal with concave shapes. There are many algorithms that can achieve this, some more complex than others, but in our case, Godot comes with a useful class called *Geometry* which provides us with a method that can do this for us.

Finally, after we have divided a building's shape into triangles, we can pass our vertex array to the *ArrayMesh* class which creates a mesh object that can be instantiated and added to our 3D scene.

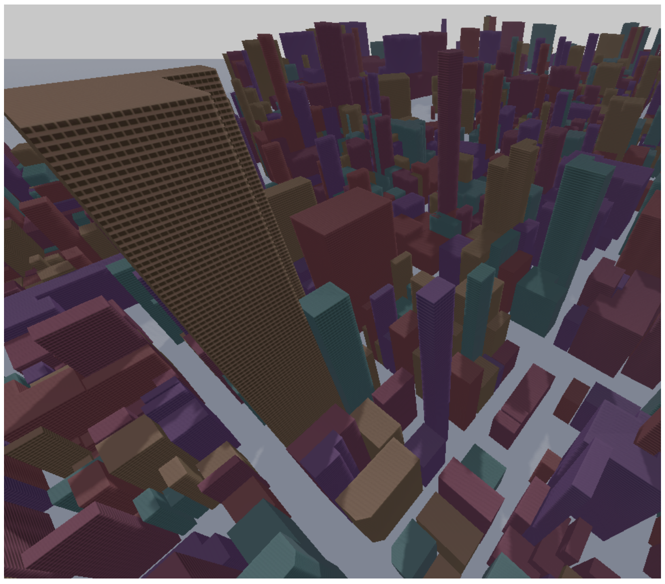


Fig. 1. Building meshes

7.2.2 Pathways

Similarly, to the buildings, we need to divide our paths into simpler segments that can then be divided further into triangles. In this case we take a path which is made up by consecutive nodes and divided into rectangular segments where each segment represents a straight path from one node to the next. Just like we did before we can divide these rectangular segments into two triangles by dividing our rectangles diagonally.

With this we can generate all the triangles that make up a path and then use those triangles to generate the final mesh, just like we did with the buildings.

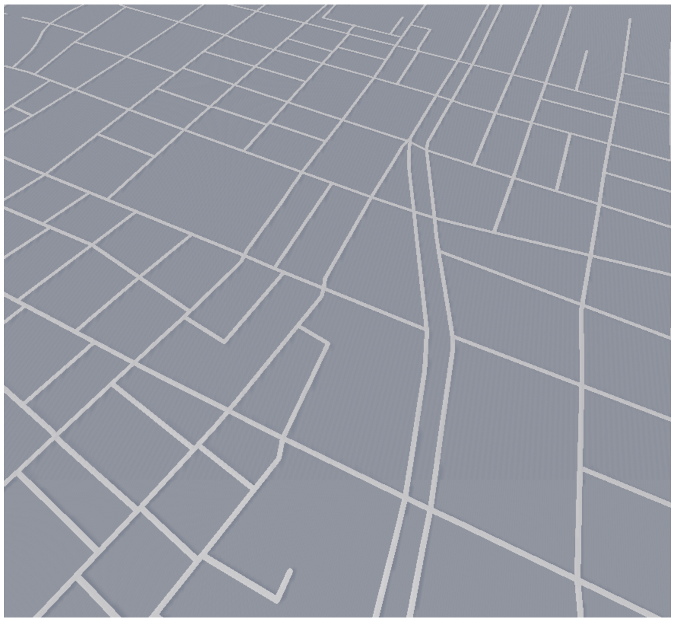


Fig. 2. Road meshes

7.2.3 Final result

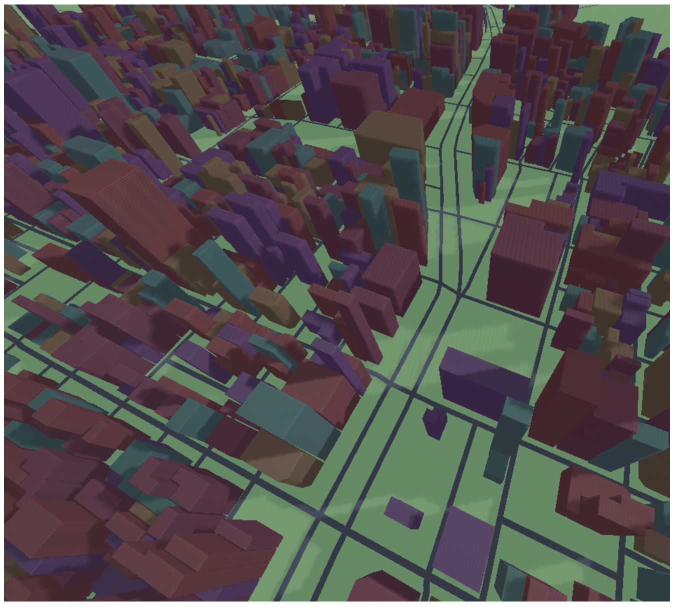


Fig. 3. Final result

Combining both the building and pathway meshes we get a basic 3D world that we can then keep adding details to.

We can see the final result in figure 3.

7.3 Node graph

Since one of our final goals for this project is to have a vehicle that can drive autonomously in our 3D world, we need a find a way for this vehicle to navigate the different roads.

One of the simpler ways to achieve this is by using a pathfinding algorithm which uses a node graph in order to find or approximate the closest path from one node to another. Before we can get into the pathfinding itself, first we need create the node graph.

After parsing the OSM file, we obtained all the different paths that make up a region, now the goal is to take these paths and represent them as nodes inside the node graph and additionally connect all the nodes that represent connected paths and calculate the distance between these nodes which can later be used in the heuristics function of the pathfinding algorithm.

Before we start creating the node graph however we first need to filter out all the non drivable paths, that way we end up with only the roads that our vehicle can drive through. Fortunately as we explained earlier the OSM file gives us a tag that describes the type of the road, this being the "highway" tag.

In the OSM wiki we can find a section that lists all the values that the *highway* tag can have. By examining these values, we can determine which types of highways are not drivable, and with that we can filter out all the non drivable paths and start creating our node graph.

7.4 Pathfinding

Now that we have a node graph that describes all the roads and how they are connected, the next step is to implement a pathfinding algorithm that allows us to find or approximate a path from one node to another.

Before the implementation phase, we looked at different algorithms that could be used, the most popular among them being DFS (Depth first search), BFS (Breadth first search), Dijkstra and A* algorithms.

After careful consideration we figured out that the A* algorithm [11] would be the best choice in this case, not only it is accurate enough for our project, but it also has good performance for our real-time application, it's simple to implement and is widely used in game engines like Godot.

Since the A* algorithm is very popular in video game development, the Godot engine already comes with an implementation that we can access using the *AStar* class that comes with its default library.

However, we can't use this class directly because the ID of the nodes that it uses has a limit of 32 bits meanwhile the ID of the nodes that we extracted from the OSM files are 64 bits. In order to solve this issue, we implemented wrapper methods that wrap the main *AStar* class methods that we will be using. These wrapper methods do the conversion between our original 64-bit IDs and the new 32-bit IDs that we will be using with the *AStar* class. This way we can still use the node graph that we created earlier while also being able to use the *AStar* class alongside it.

7.5 Creating the vehicle

In this step of the project's development, we need to make a vehicle that the user can interact with in order to fulfil the project's goals.

We can divide the vehicle into three components which can be developed separately. First, we have the vehicle physics which determine how it should move when interacted with. Then after that we have to provide an interface for the user to control it. Lastly, we have to develop a simple AI that allows the vehicle to drive autonomously when given a path to follow.

Furthermore, the user control component needs to work seamlessly with the AI component in order to allow the user to switch between manual and autonomous driving modes in a smooth manner.

7.5.1 Vehicle movement

This is the physics component of the vehicle. Fortunately, the Godot API already comes with many classes that use the physics engine in order to implement all the useful features a physics body might need, like collision, mass, gravity, friction, etc.

One important class that we use to implement the vehicle physics and movement is the *RigidBody* class or specifically its sub-class known as *VehicleBody*. This class on top of providing us with an implementation of a physics object that reacts to collision and physical forces it also comes with an implementation for the wheels of the vehicle.

The wheels can be divided into two groups, front wheels and rear wheels. The former are used for steering while the latter are used for accelerating and braking. Each of the two groups of wheels comes with a set of variables that can be accessed and manipulated in order achieve the desired movement.

For rear wheels we can use the *engine_force* and *brake* variables which values can be greater or equal to 0.

The acceleration which is represented by the *engine_force* variable is calculated by using two variables that control the maximum RPM (Revolutions Per Minute) of the wheels and the maximum torque that can be applied by the engine to make the wheels revolve faster. These two variables are used in the following formula to determine how much torque (*engine_force*) needs to be applied in each cycle of Godot's physics engine:

```
engine_force = throttle x maxTorque x
(1 - (currentRPM / maxRPM))
```

In this formula the user input affects the throttle variable which in turn determines the acceleration of the vehicle.

Finally, the steering of the vehicle is controlled using the *steering* variable of the front wheels which can range from -1 to +1. This variable can be directly controlled by the user input but we apply some additional interpolation in order to make turning feel smoother and make it easier to control for the user.

Once we calculate these three variables, we can then use them to manipulate the internal variables of the *VehicleBody*. The *engine_force* determines how fast the back wheels of the vehicle revolve which in turn affects the movement of the vehicle, making it speed up in the direction of its forward vector, in this case the forward vector is the direction that the vehicle is facing. The *brake* variable is similar, but instead of moving the vehicle forward it applies an opposing force that causes it to slow down and stop. Finally, the steering controls the angle of the two front wheels which combined with the forward movement causes the vehicle to gradually change the direction it is facing and thus changing the direction of its forward vector.

7.5.2 Vehicle user control

Here we provide an interface for the user to interact with the vehicle. For now, the control interface consists the WASD and SPACE keys of the keyboard, although this can be expanded upon in order to handle joysticks, steering wheels and other types of control devices.

7.5.3 Vehicle AI

This is a simple AI system that allows the vehicle to traverse the node graph in order to reach a specific target node. This is achieved using the previously created node graph and the pathfinding modules.

The way this simple AI system works is by constantly moving the vehicle forward while steering it towards the next node in our path. We also apply breaking when the vehicle is turning in order to make sharper turns.

The challenge here lies in applying the right amount of steering in order to reach the nodes of the path that leads us to our target destination. This can be calculated using vector math, specifically by taking the Y component of the resulting vector of the cross product between the forward vector of the vehicle and the target direction vector, this Y component gives us the amount of steering that needs to be applied in order for the vehicle to turn in the direction of the target node and thus move towards it.

```
steering = (forwardVector x targetVector).y
```

Calculating the forward vector and the target vector requires three inputs. First, we need the current linear velocity of the vehicle which is represented by a three-dimensional vector which holds the value of the XYZ components of the linear velocity, in other words it tells us how much the vehicle is moving in each axis. The second input is the current position of the vehicle in 3D space. Lastly, we need the position of the next target node in our path. We can obtain the forward vector of the vehicle by normalizing the linear velocity vector. After that we calculate the target vector by calculating the difference between the target node position and the current vehicle position and normalizing the resulting vector.

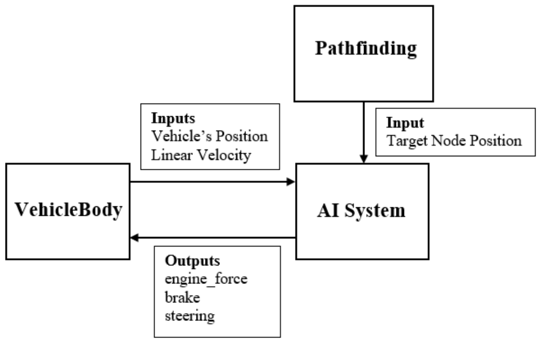


Fig. 4. AI system input/output diagram

A challenge that we faced while developing this AI component, is making the vehicle drive in the right side of the road instead of in the middle like it would do by default when following the path given to it by the pathfinding module.

This was solved by offsetting the position of all the nodes of the path that we are following by the normal vector of each path segment. The length of this normal vector can be changed in order to determine how far to the right we want the vehicle to drive, which is important when we have roads with varying width.

7.5.4 Vehicle visuals

Finally after creating all the important systems of the vehicle we added a 3D model [12] to give it some visuals as shown in the following figure.

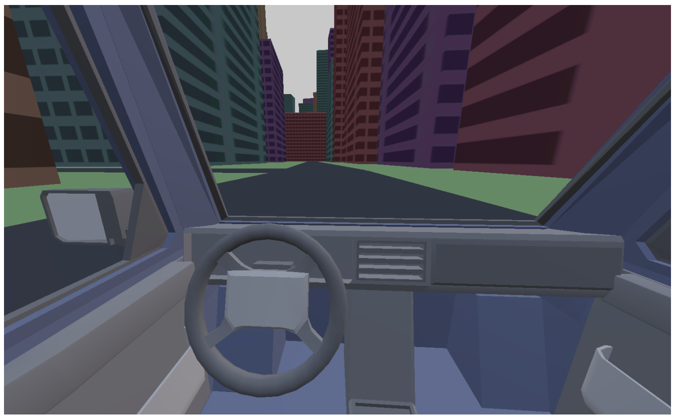


Fig. 5. Vehicle visuals

7.6 GPS system

The GPS system consists of a 2D view of the world which tries to emulate a real GPS by allowing the users to view their position on the map and choose a destination for the vehicle to drive to autonomously.

There are many ways to achieve the 2D view of the world, the simplest way that we came up with is to set up a camera on top of the 3D world and make it look down at it. This way we can see all the roads and buildings as if they were on a 2D map. In addition to that we scaled down the height of all building in order to give it a more 2D feel and preventing tall buildings from obstructing the view in some cases.

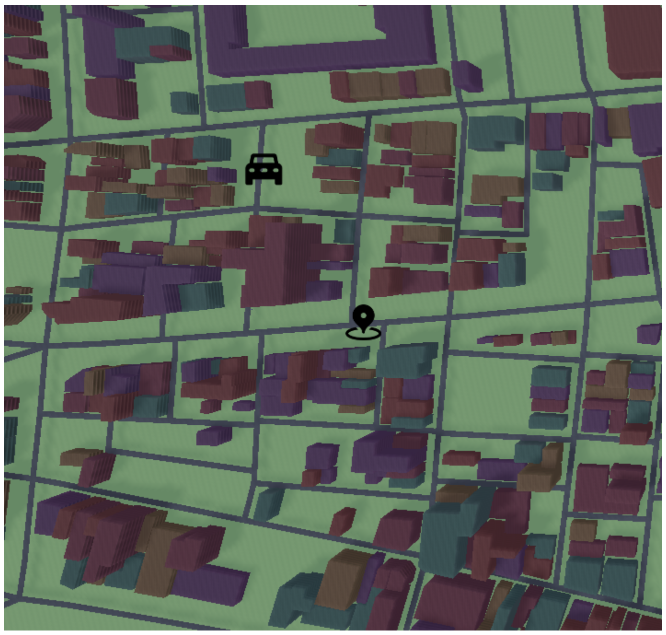


Fig. 6. GPS 2D view

After the user chooses a destination, we need a way to guide them along the path in case they decide to drive manually instead of letting the vehicle drive autonomously, a simple solution is adding a guiding arrow that points them towards the right direction.

7.7 Non-user-controlled vehicles

At this step, we have generated a basic world with buildings and interconnected roads. We can take the vehicle that we developed for the user to interact with the simulation and use it to populate the world with non-user-controlled vehicles in order to make the simulation more interactive by providing one of the challenges that allows us to tests the user's attention and reaction speed.

In order to populate the simulation with these non-user-controlled vehicles we can use the already developed node graph which allows us to procedurally place vehicles in many different nodes. However, so far we are only able to place a few dozen vehicles due to performance limitations, which we will solve later.

Even though the vehicle that we developed earlier has the ability to navigate it's way along a path, it cannot however choose it's own path based on where it is in the world, since that is the user's job. In this case we do need to give these non-user-controlled vehicles a way to keep choosing new paths every time they reach a destination.

Something that we didn't consider earlier when developing the vehicle's AI is the possibility to encounter obstacles in the way while following a certain path. Now that we have more than one vehicle roaming the map, we need to develop a system that detects obstacles and react to them. This system however will only be available to the non-user-controlled vehicles, since the user is supposed to take manual control of the vehicle when such situations arise.

In Godot we simulate a proximity sensor by creating a box shaped area that allows us to detect when entities enter or exit this area. By placing this box in front of the vehicle we can then wait for specific objects to trigger the sensor, like other vehicles for example. We can then brake or use any other method to avoid whatever obstacle we want to avoid. In our case we made the vehicles slow down when detecting other non-user-controlled vehicles and completely brake when detecting the user's vehicle that way we can prevent accidents that are cause by the AI vehicles rather than the user.

7.8 Pedestrians

In addition to simulating vehicles roaming the generated world we decided to also have pedestrians, which further gives us new opportunities to test user awareness.

To populate the simulation with these entities we use the same method used in the previous section to place vehicles with a slight modification. Since pedestrians can't be placed on the road, we need offset their position after placing them on a node that represents a point in a road. We also have the choice between placing them on the right or left sidewalk, which we randomize in order to get a more organic looking placement.

Of course, these pedestrian entities like the vehicles they need to move in order to make a more interactive simulation.

Similarly, to what we did with the vehicle pathfinding we use the nodes graph to implement their movement. However this time we took a different approach, instead of generating an entire path from their origin to a certain destination we opted of a more random movement in order to give them a different feel compared to the vehicle entities. With this method we can make them roam around a single location which prevents them from leaving certain locations empty while having clusters of pedestrians in others. Instead, we have a more even distribution.

7.9 Chunk system

After creating a method to populate the map with vehicles and pedestrians, we run into a common problem that projects of this type run into which is performance.

With a few entities the simulation works fine, but that's not enough for what we are trying to achieve, which is having thousands of these entities all over the map.

Fortunately we don't need all of these entities to be active at the same time, since our simulation revolves around the user we only need to worry about what the user can see.

Which leads us to a very common method to solve these kinds of problems, which is a chunk system that allows to only activate the entities that are in proximity of the player.

There are many ways to implement this type of system. In our case we opted for using dictionaries. We can use the chunks coordinates as keys for the dictionary and for the values we use a class that holds a list of all the entities that are inside this chunk in addition to a variable that tells us whether this chunk is active or not.

At the start of the simulation, we can begin populating this dictionary with all the entities that exist. In order to know in which, chunk an entity belongs, we just need to have a variable that tells us the width and the height of each chunk, after that we can figure out where to put each entity by simply dividing it's coordinates by the width/height of the chunks.

The next step is to load and unload chunks based on how far they are from the user coordinates. Every time we unload a chunk, we deactivate all the entities that it holds, which not only stops them from executing all code logic, but also stops them from rendering to save on graphical performance in addition to CPU performance.

Loading requires the same operations but in reverse, meaning that we make the entities visible to the user and resume their execution.

We wrote this chunk system to be flexible, which means in addition to using it for pedestrian and vehicle entities we can use it for any other entities that we would like in the future and still keep a relatively good performance for a simulation like this to work.

7.10 User interaction

In order to measure the user's attentiveness and reaction speed, we need to provide them with certain situations that they can interact with in order to generate data that measures these interactions.

These interactions consist of scenarios that mimic real world situations where the driver has to take manual control of the vehicle in order to prevent an accident.

Currently we have other non-user-controlled vehicles that roam around the 3D world and additionally they can serve as one of these obstacles that can test whether the user is paying attention and whether they can react fast enough in order to avoid a potential collision with another vehicle.

Another one of these situations or obstacles that we developed is the pedestrians, which in addition to their usual behaviour of walking on the sidewalk, we made it so that sometimes a pedestrian decides to cross the street in front of the user's vehicle, which also requires manual input in order to avoid.

Now that we have a few situations that test the user awareness and reaction reaction speed, we can start collecting the relevant data that these interactions generate.

When the user chooses a destination and the vehicle starts driving autonomously to that location, at specific points of that journey we can activate one of these test scenarios and start timing how long it takes for the user to start manually controlling the vehicle and also whether they managed to avoid this dangerous situation or not. Finally, when the user arrives at their chosen destination, we can use the previous data to calculate the final score for that particular journey, which we can then display to the user along with the data of each individual interaction.

8 Conclusion

Solving the problem of driver distraction in autonomous vehicles could be a great hurdle to overcome for vehicle manufacturers and safety organizations. A lot of research is needed to understand these distractions and how they affect human awareness.

We believe that having a safe simulated environment for researchers to gather the relevant data for their work could prove very important to find real life solutions.

For this reason, we developed this software, which allows us to generate 3D virtual environments using real world data gathered by OpenStreetMap and use these environments in order to test the attentiveness of users by putting them in the driver seat of a simulated autonomous vehicle.

This work is by no means meant to be a software simulator ready to be used for professional applications. It is more of proof-of-concept which intends to show how such an environment can be useful to study driver distraction in autonomous vehicles.

This project could be expanded in the future using cameras, sensors and other tools, to not only measure driver reaction speed but also study driver behaviours such as body and eye movements, heart rate, speech, among many others. This kind of tracking might allow us to detect when and what causes these distractions, how to better alert the driver when detected and what other alternative safety measures can be implemented.

References

[1] SAE (Society of Automotive Engineers).
[https://en.wikipedia.org/wiki/SAE_International]

[2] Levels of automation.
[https://web.archive.org/web/20180701034327/https://cdn.oemoffhighway.com/files/base/acbm/ooh/document/2016/03/automated_driving.pdf]

[3] Alexey Dosovitskiy and German Ros and Felipe Codevilla and Antonio Lopez and Vladlen Koltun, An Open Urban Driving Simulator, Proceedings of the 1st Annual Conference on Robot Learning, 1-16, 2017
https://carla.org/

[4] 3D layer of buildings.
[https://osmbuildings.org/]

[5] An API for read-only queries of OSM data
[http://overpass-api.de/]

[6] Noa Zangi, Rawan Srour-Zreik, Dana Ridel, Hadas Chassidim, Avinoam Borowsky, Driver distraction and its effects on partially automated driving performance: A driving simulator study among young-experienced drivers, Accident Analysis & Prevention, Volume 166, 2022, 106565, ISSN 0001-4575.
https://www.sciencedirect.com/science/article/pii/S000145752200001X

[7] Insurance Institute for Highway Safety (IIHS)
[https://www.iihs.org/about-us]

[8] IIHS creates safeguard ratings for partial automation
[https://www.iihs.org/news/detail/iihs-creates-safeguard-ratings-for-partial-automation]

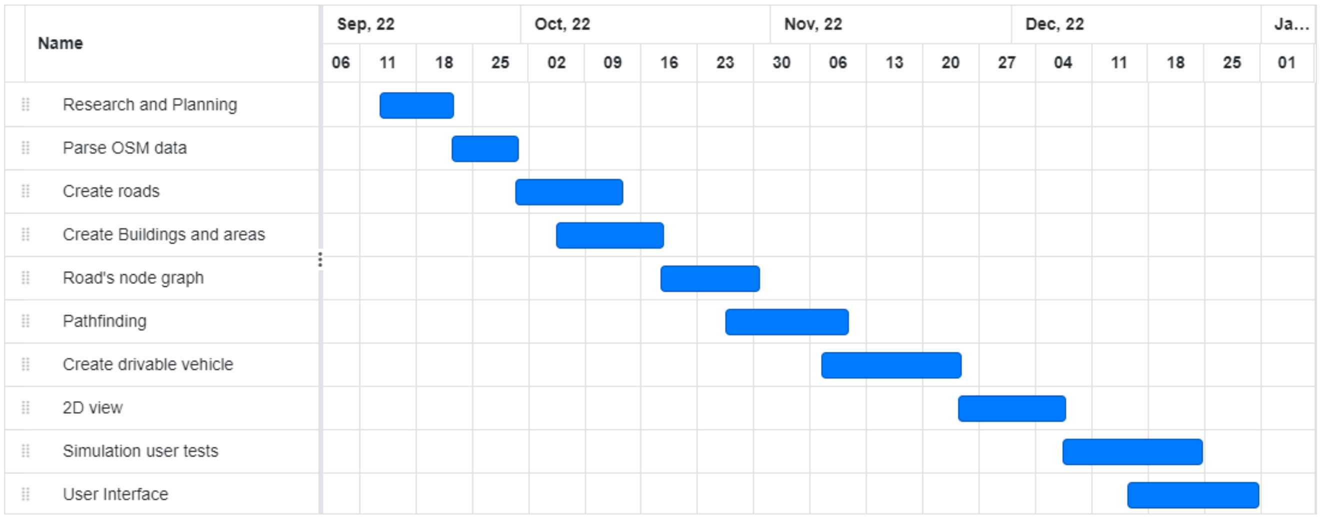
[9] The OpenStreetMap wiki.
[https://wiki.openstreetmap.org/wiki/Main_Page]

[10] Godot Version 3.5 documentation. Juan Linietsky, Ariel Manzur and the Godot community (CC-BY 3.0). Revision 7299355d.
[https://docs.godotengine.org/en/stable/index.html]

[11] A* pathfinding algorithm section from the Wikipedia.
[https://en.wikipedia.org/wiki/A*_search_algorithm]

[12] Vehicle 3D model.
[https://sketchfab.com/3d-models/low-poly-small-car-ebe7c5e98a7448b5abb2eaf0cb22b766]

APPENDIX A
PLANNING GANTT CHART


Name
Sep, 22
Oct, 22
Nov, 22
Dec, 22
Ja...
Research and Planning
Parse OSM data
Create roads
Create Buildings and areas
Road's node graph
Pathfinding
Create drivable vehicle
2D view
Simulation user tests
User Interface