# Gazebo World Generator Using Geographic Data

Rory Scobie scobier@email.arizona.edu
Michael Treiber michaeltreiber@email.arizona.edu
Sahachel Flores sflores1@email.arizona.edu

ECE 573
Software Engineering Concepts
Spring 2021

# Contents

# List of Figures

# List of Tables

# 1 Executive Summary

This project aims to develop a ROS application that procedurally generates a Gazebo world based on geographic OpenStreetMap (OSM) data. This data is collected at runtime using web API's, converted into an SDF file, which then gets transferred into a world file that uses Gazebo to simulate the world. A GUI is used in conjunction with a URDF vehicle model to control the movement and location of this vehicle in the world and send data from the URDF model to ROS. As the vehicle moves around the world, more simulated world chunks will appear and the chunks behind the vehicle will disappear, maintaining a self generating world of chunks.

# 2 Project Overview

Simulating a self-generating world in Gazebo using realistic geographical locations is an interesting application due to the massive expansions it can bring with an autonomous vehicle. If one can set the geographical location of a certain area of interest to study how the vehicle will function in the environment specified has numerous possibilities in improving the quality of simulating autonomous vehicles. This project will generate the environment of real-time data from OSM, run this data into Gazebo using an SDF format in a world file and move a car inside this simulation, which generates more chunks for the car to go to. This project will also project a GUI for the user to change the velocity of the car and the location the car is in. Many users would like to test their vehicles in many different environments but cannot accomplish this, so this project is done to fix that problem. Using OSM, the data from the meshes will be converted into an SDF file, and put into a world file to be launched in Gazebo. Gazebo will then simulate the environment, the URDF vehicle created and any other models one wishes to put into the world. The GUI will then be used to move the car around, and set/change locations of the vehicle, while data in the background is being published in ROS. The vehicle will reach the edge of the world and generate another chunk from OSM, while deleting a portion of the previous area, creating self-generating world chunks for the car to move around in.

# 3 Requirements

## 3.1 B Requirements

1. The user shall have control over the following variables: starting coordinate or preset location from a list, number of chunks to load around car, car movement speed. This is dependent on the creation of a working GUI.

2. The program shall generate the environment with a minimum real-world dimension of 1000x1000m at a time. This is dependent on the creation of a working world file.

3. The program shall base its environments on real-world data. This is dependent on generating OSM meshes and turning these meshes into working SDF files.

4. The generated world shall include world-accurate terrain heights. This is dependent on the data returned from OSM and the Gazebo heightmap model. (relies on B3)

5. The generated world shall include roads with world-accurate positions, connections, and lengths. This is dependent on the conversion of OSM data to SDF files to make accurate roads. (relies on B3)

6. The generated world shall include buildings with world-accurate footprints and heights. This is dependent on the data from OSM and the conversion algorithms used to generate accurate building models. (relies on B3)

7. The program shall fulfill all other requirements in city block, neighborhood, and freeway environments. This is dependent on the vehicle URDF file used, the GUI being used and the generated world used to see all procedures in these areas fulfilled. (Relies on B1,2,3,4,5,7,8,9,10,11,12)

8. The program shall generate a world around the reference point such that the vehicle's sensors won't detect the unloaded world. This is dependent on the URDF file created, and the range of the sensors in the URDF model compared to the generated chunk.

9. The program shall start the simulation from a reference coordinate. This is dependent on the URDF file being the center of the generated world file, set to those starting coordinates.

10. During the simulation the vehicle shall stay on the roads limits. This is dependent on the collision settings for the URDF file and the generated road SDF files in the world file. (relies on B5)

11. The speed of the vehicle shall be constant. This is dependent on the node used to control the velocity and the UI, keeping the velocity at a constant value.

12. The simulation shall be generated on Gazebo. This is dependent on the URDF file, the world generated and the SDF files included in the world. If any of these don't generate then the simulation isn't running properly.

## 3.2 A Requirements

1. The car shall navigate between two GPS coordinates, following the roads where possible. This is dependent on the UI settings getting put into the simulation and the URDF file being updated to have better steering on the roads. (relies on B10)

2. Dynamic speed will change based on the environment. This is dependent on the new models included in the world, and the nodes responsible for keeping the velocity constant. Changing these nodes to make the car stop if there's something in front of it, or speed up if there's nothing in front of it, will ensure dynamic speed.

3. The quality of generating the world around the car will improve. This is dependent on the models used in the world file. Seeing if these models have an improved version (.dae file) is worth checking out. (relies on B3,4,5)

4. The program will Generate obstacles along/on the road, such as people, cars, streetlights, etc. This is dependent on the new models being used in the generated world file. (relies on B5)

5. The program shall generate a scenic environment. (water, trees, building aesthetics, cacti, etc.) This is dependent on the new models being used in the generated world file. (relies on B5)

6. The generated world shall have common traffic control such as stop lights, crosswalks, and stop signs. This is dependent on the new models being used in the generated world file, and the URDF file being updated to account for these models used for traffic control. (relies on B5)

7. The user shall control the car's movement with a keyboard. This is dependent on the GUI having key binding events to account for keyboard presses.

# 4 Application Analysis

The use cases, sequence diagram, state model, and prototype of the UI is showed below. Figure 8 displays the UI prototype which has the option to insert a latitude, longitude, and the quality of the simulation. The UI was developed using PyQt5 which is similar to Ros-UI, qrt. Figure 1 displays the state model of the UI, the state model has the following behavior. When UI is initialized, it first goes to the idle state where we set the output variables of latitude, longitude, quality of graphics, and constant speed to default values. Additional to this, we check for error messages and print them to the user if they are found. Then we wait for the start button to be pressed. If neither of the input options is not modified by the user, then the default predetermined options are maintained. However, if any of the input options changes, inputDetect will be one and thus we move to the checkInput state. In this state, all possible inputs are checked by transitioning to 3 different states when the input variables are analyzed. If the variables are within the acceptable range, we pass the test and transition to the next state, however if one of the variables does not passed the test, we go to the error state. In the error state we print a message to the user where we inform that an invalid input was detected and wait for next inputs to perform the analysis. If all values are within a desired range, then, it

transitions to the ready state where it clears the message and wait for the start button. If an input is detected while in this state, it repeats the input analysis check. Once the start button is pressed, we transition to the simulation state where the simulation begins. In this state, we publish the input variables to ROS and subscribe to the location and velocity topics. Then, the UI displays the dynamic location of the vehicle and the velocity. To exit simulation state the stop button needs to be pressed, which causes the transition to the idle state. Figure Figure 1 displays the use case for the gui. The user can select start or stop the simulation.5 illustrates the interaction of the user, UI, ROS, and Gazebo. When the user presses the start button, the gui analyses the inputs provided by the user then sent this information to ROS which then sent it to Gazebo. When the user presses the stop button, ROS and Gazebo are terminated. Figure 6 illustrates the interaction of the user, UI, and ROS when the user enters an input on the latitude and longitude input boxed. Figure 7 illustrates the interaction of the user, UI, and ROS when the user selects one of the quality options

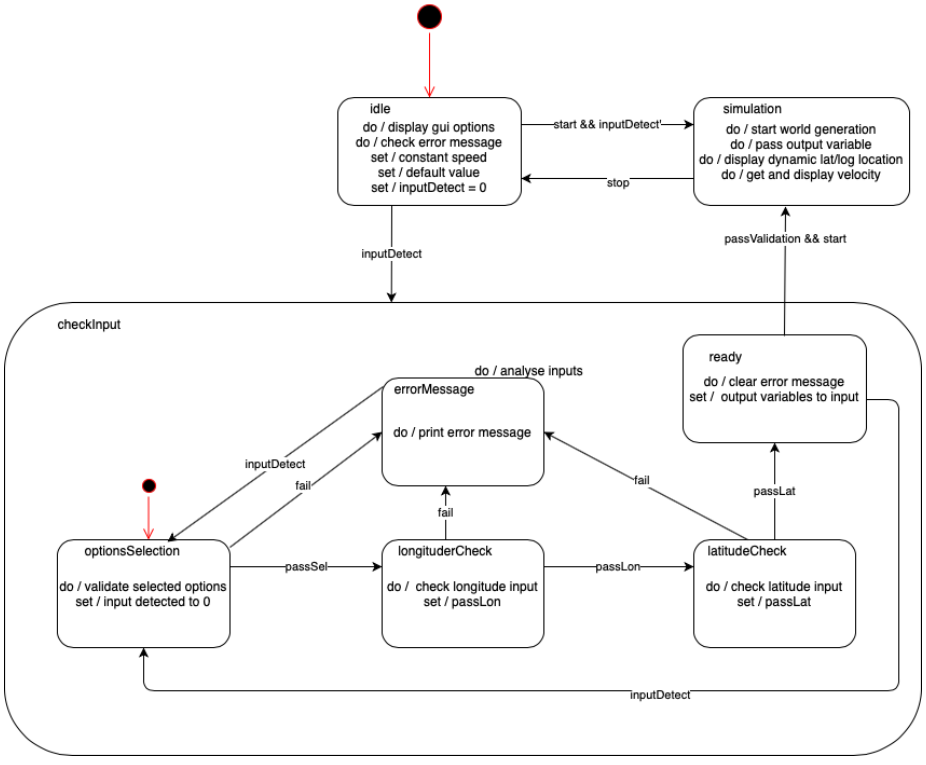


Figure 1: State model representing the UI behavior

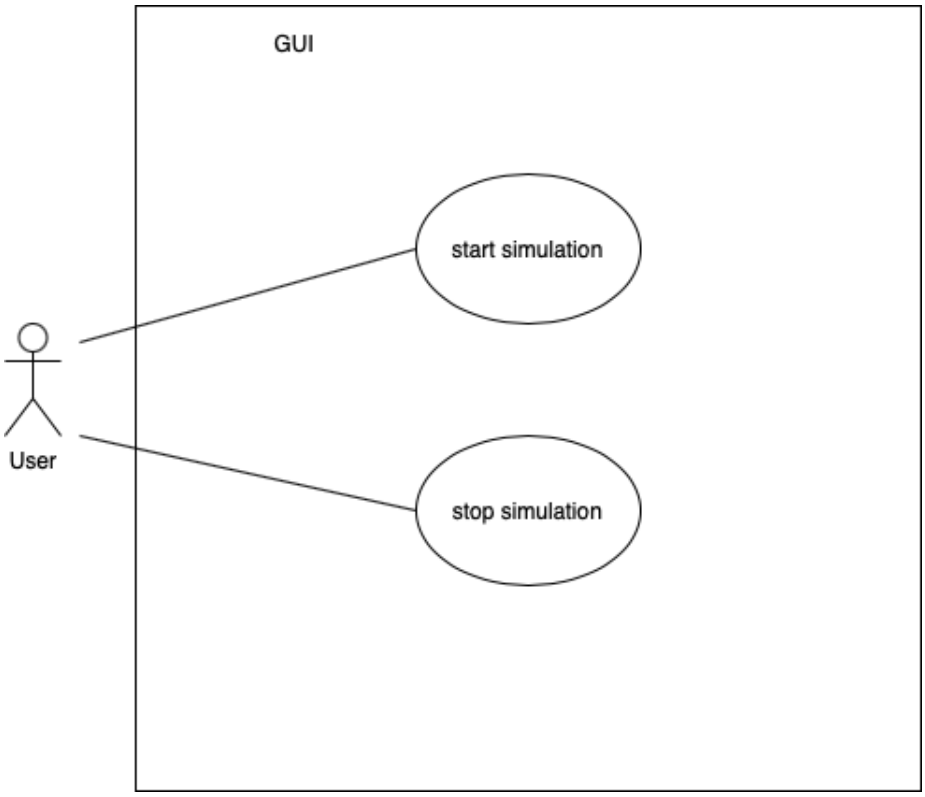


Figure 2: Use case diagram for the user quality option

# 5 Domain Analysis

Interactions of the portions of the design that are not seen are shown in the figures that are listed here. Figure 9 is a general representation of the three main nodes that will be used in this design. It shows the process used for the nodes publishing and subscribing to each other as well as talking to other built-in ROS nodes, such as rqt, ros_control and the Gazebo node that integrates with ROS. The arrows represent the publishing of a topic, where the arrow tips are publishing, and the connections are subscribing to one another. Figure 10 is an Activity Diagram that shows a general view of what is happening in the Gazebo simulation, showing how the user would enter information into the GUI, and how the simulated world would react to that, giving the user many options when simulating the design. Figure 11 is an Activity Diagram that shows a breakdown of the Generation Node, and how it communicates with the Simulation Node. It also shows the process of receiving data from OSM and creating STL files from it, via the algorithms listed in the Important Algorithms section. Figure 12 is an Activity Diagram that shows a more in depth view of the Simulation Node

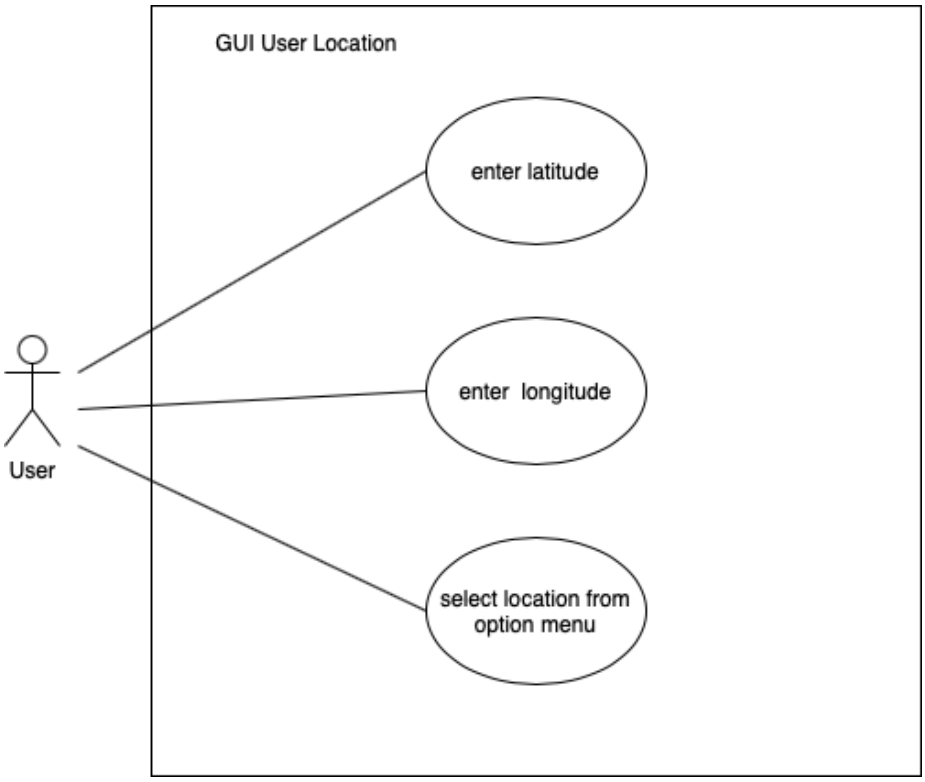


Figure 3: Use case diagram for the user setting options

working with both the UI Node and Generation Node. It calls to the Generation Node to collect an SDF file based on the initial UI Node response from the user on initial location. The Generation Node complies by collecting OSM data and turning this mesh data into STL files which are compacted into an SDF file which is returned to the Simulation Node. If the Simulation Node doesn't receive this, or gets an error message, it will try again. This SDF file gets put into a world file along with the vehicle URDF file and if that doesn't launch in Gazebo, then the Simulation Node will go back to querying the Generation Node. If the world file successfully generates, then the Simulation Node communicates with the UI Node to control the generated URDF vehicle file. It is communicating with the Gazebo node to show the simulation working with the inputs given, and can put in more models, or chunks of the world dynamically with the movement of the vehicle. It can also set different locations to go to or change locations in the world via UI Node coordinates entered. Figure 13 shows a communication path of the three nodes in a Sequence Diagram. The communication path shows the UI Node giving a UI event to the Simulation Node, which can move the vehicle, while the Simulation Node is asking the Generation Node for SDF files for the next chunk. The Generation Node complies and gives an SDF file using

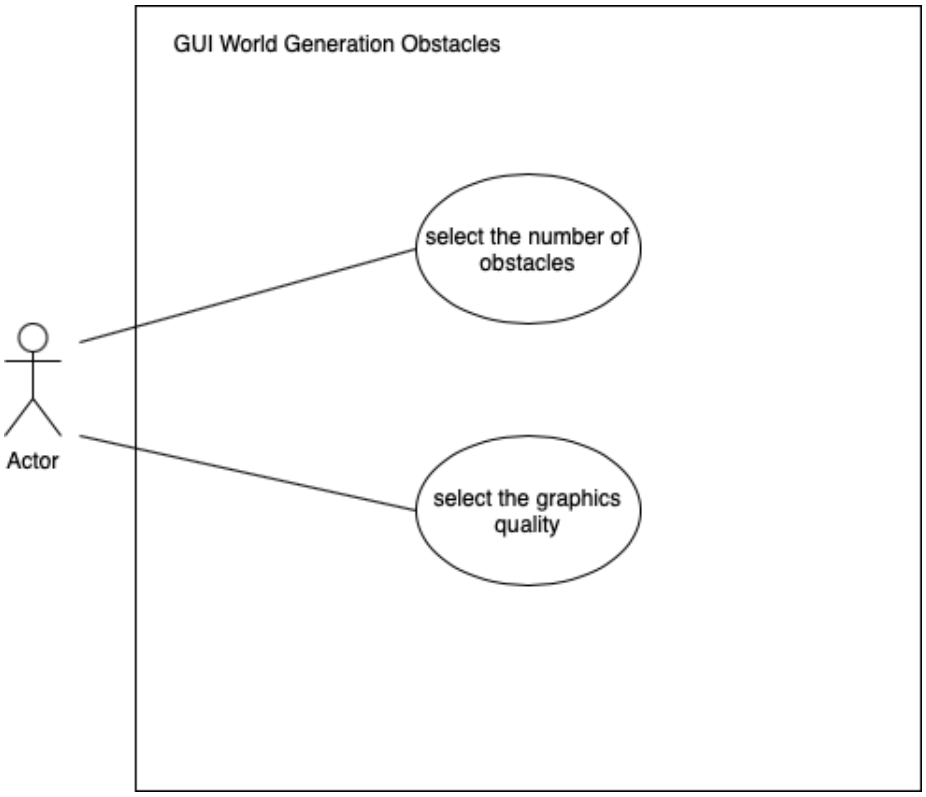


Figure 4: Use case diagram for the scenery generation features

the algorithms specified in the Important Algorithms section to the Simulation Node, which creates the next chunk of the world and sends that new location information to the UI Node. The UI Node is discussed more in the Application Analysis section, which includes State Models, Use Cases, and shows how the UI Node works using images of a GUI.

# 6 Important Algorithms

The Generation Node uses various algorithms to properly parse geographic data and generate scenery models from it. The use of these in the Generation node is indicated in figure 11, but the specific process of each is detailed here.

## 6.1 Terrain Generation Algorithm

To generate terrain, we first query the open-elevation web API. We do so by sending a POST request with a list of all the lat/lon coordinates we want to get the elevation of. This can be fairly sparse since the underlying dataset,

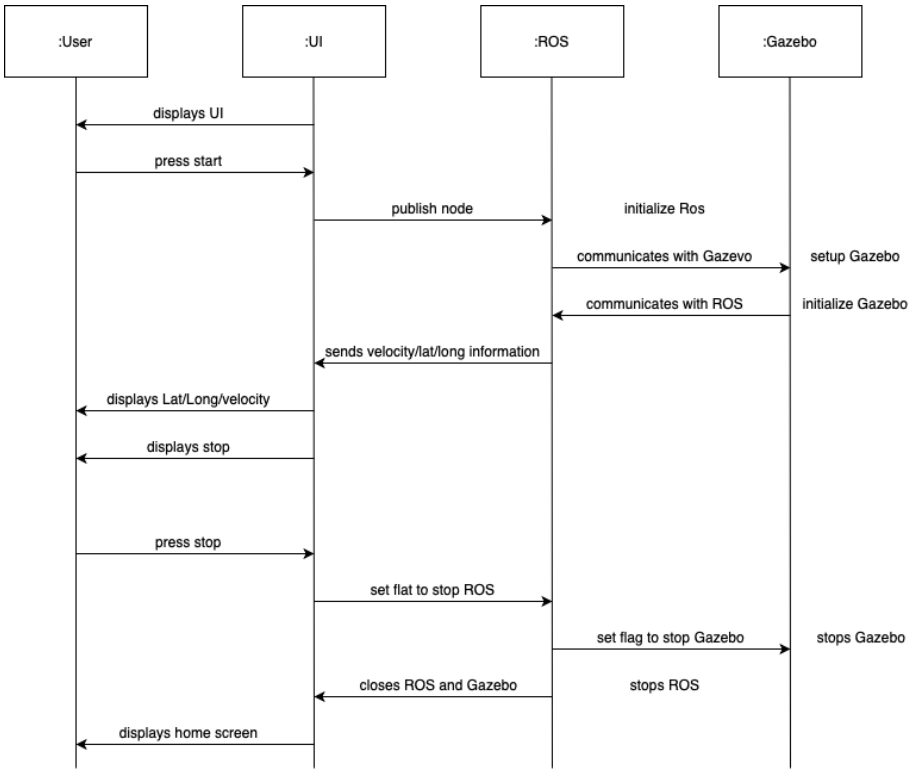


Figure 5: Sequence Diagram for User Case GUI

SRTM250m, is only accurate to 250 meters. The elevations combined with the coordinates provides us with a 3d point cloud, which can be converted to a suitable mesh of the terrain using the marching cubes algorithm. This mesh is then saved as an .STL file. A .SDF file referencing the model is also created to facilitate spawning into gazebo.

## 6.2 Road Generation Algorithm

Roads within the target area are queried using openstreetmap's Overpass API. This query returns an ordered list of points, one list for each road. By iterating linearly across the list, we create a new set of line segments from each adjacent point. A single .sdf file will be generated containing the locations of all the road segments, as well as their orientations. No model will be generated for road segments: instead a prefabricated model will be repeatedly placed between the endpoints of a line segment to fill in the space between.

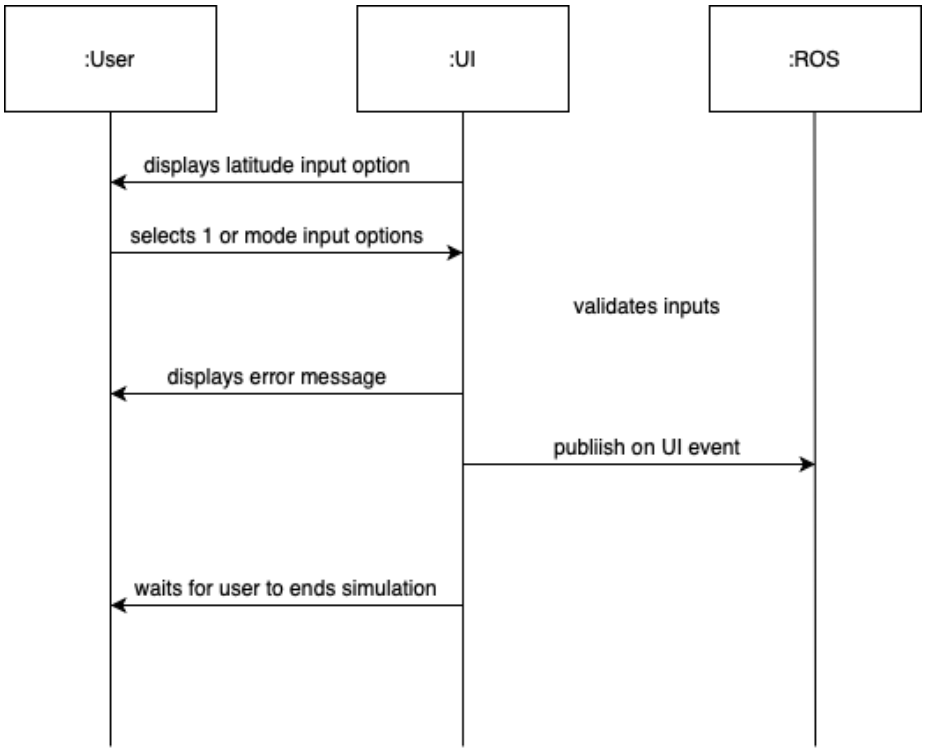


Figure 6: Sequence Diagram for the user location uses case

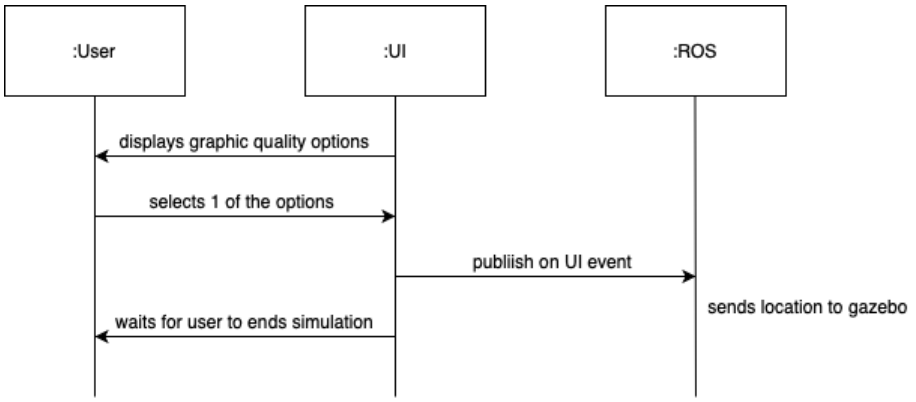


Figure 7: Sequence Diagram for the the quality option uses case

## 6.3 Building Generation Algorithm

To generate buildings, we will first query openstreetmap's Overpass API. An example query can be found in section 9.1. This query returns an unordered set

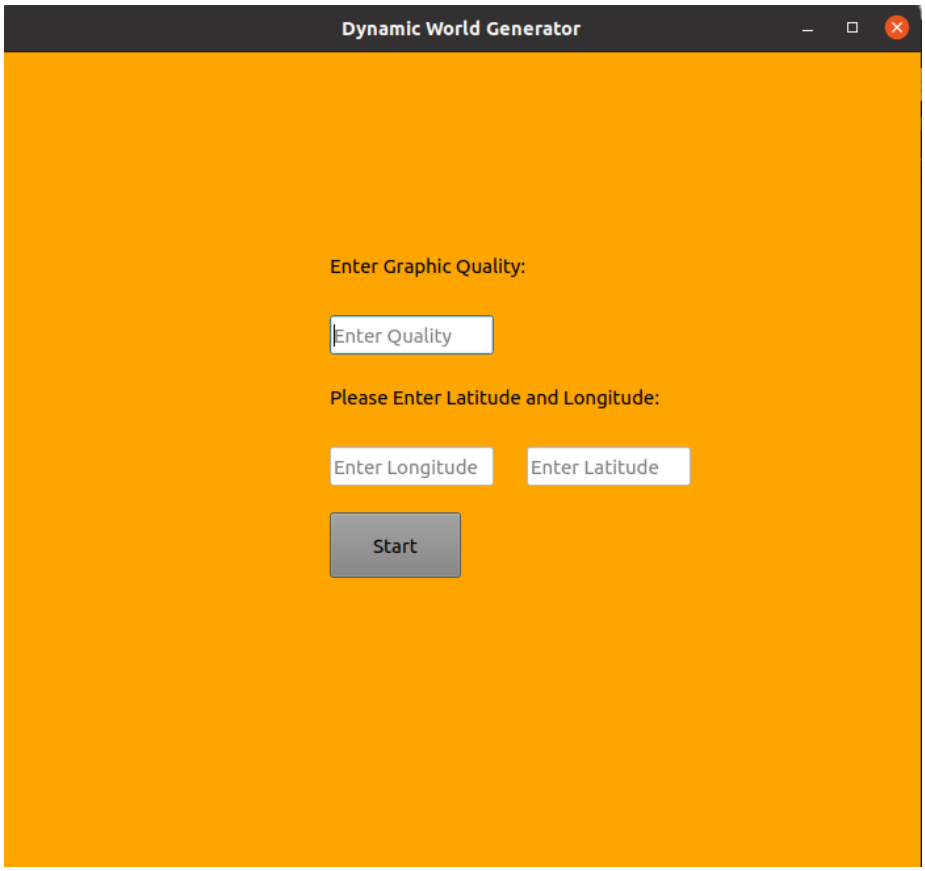


Figure 8: UI prototype .

of buildings data objects. Each building may contain an ordered list of points representing its footprint, and may also include a tag representing its height. If no footprint is provided, we will use a placeholder model chosen from a small subset based on the description tag of the building. Otherwise, we will generate a mesh for the building using a polygon extrusion algorithm. This is a simple algorithm, basically the footprint is copied upwards along the z-axis and faces are created between the points. The extrusion distance will default to 10 feet if no height data is available for the building, this should be sufficient for many driving car simulations since hopefully cars won't drive into the second floor of a building. The mesh is saved as a .STL file, and a .SDF file is generated to reference all the building models (or their placeholder). Figure 14 shows an example of building footprints collected from openstreetmap, and 15 shows an example of these same footprints extruded into 3d.

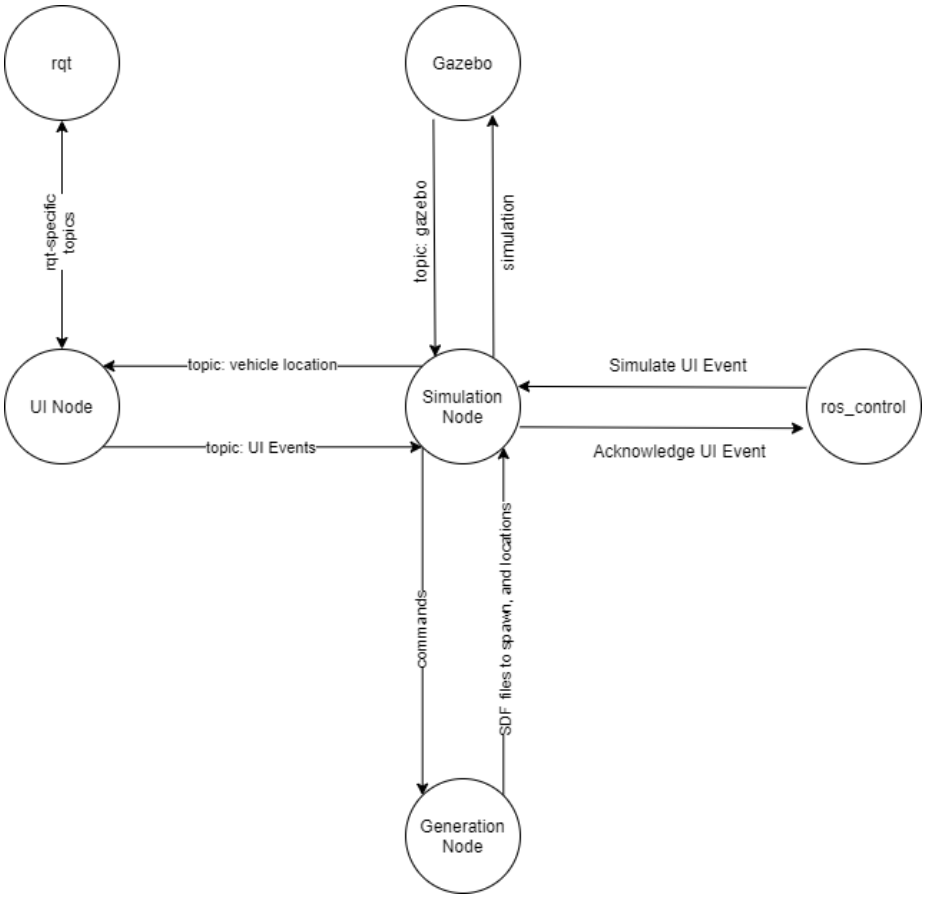


Figure 9: ROS nodes and topics

## 6.4 Scenery Generation Algorithm

Scenery generation will take the road generation algorithm's line segments and the building generation algorithm's footprints as input. Then, For each line segment, we will randomly choose to place objects either next to or on to the road. We will check if objects would be inside a building footprint with a simple crossing number algorithm. A .sdf file is generated for all these objects, with their locations and their models. Models are chosen from a small set of prefabricated models.

## 6.5 Route-Finding Algorithm

To determine the path to take between two coordinates, we first find the closest road segment endpoint to navigate to. This can be looked up efficiently if we store the points into an data structure like a quadtree first. Next, we will

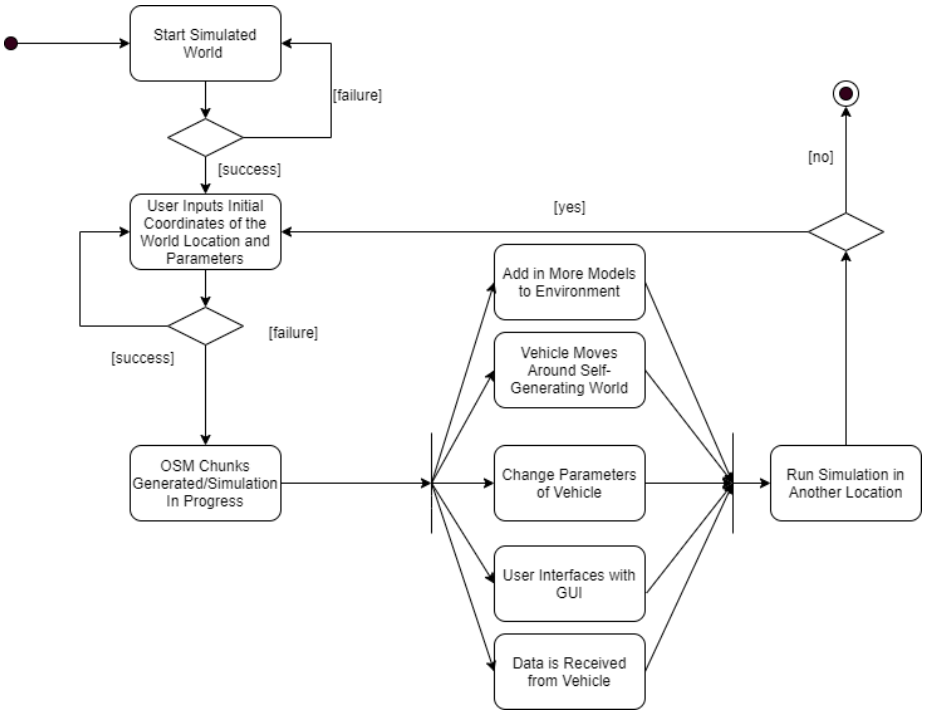


Figure 10: Top-level activity diagram of program execution

create an undirected graph using the points as nodes and the line segments as edges. To choose the path, We will perform A* search using the "as the crow flies" distance of a line segment's endpoint from our desired destination as our heuristic. The points in the resulting path will be saved, so they can be used by the Simulation node as a series of waypoints for the vehicle.

# 7 Class Design

At the top level, the functionality of our project is divided into three Nodes: The Generation node, the Simulation node, and the UI node. These nodes communicate with each other and external ROS nodes via topics, as seen in figure 9.

The class diagram in figure 16 portrays the publisher/subscriber relationships between the nodes, and specifies the data format of the topics that they use to communicate. In this diagram, we assume the following features are handled by ROS:

- Publisher and Subscriber Classes Specified by ROS
- ROS notifies subscribers, publishers notify ROS

- SimulationNode interfaces with Gazebo node
- SimulationNode interfaces with ros_control node

Exceptions are indirectly handled by the simulation node through the use of timeouts. The main causes of exception are network and file i/o exceptions when the simulation node requests chunk generation from the Generation node. The generation node's web API calls may fail due to network dropout, or it may fail to write data to the disk due to permissions or capacity issues. In these cases, nothing will be sent back to the Simulation node until the coordinate is resent or a new one is received. The Simulation node will detect that an error has occurred if nothing is sent back within the timeout period.

# 8 Testing Strategy

## 8.1 B Requirement Tests

- Requirement: The user shall have control over the following variables: starting coordinate or preset location from a list, number of chunks to load around car, car movement speed.
- Test: Run tests on the created GUI to make sure that it is giving the user correct options, by seeing the data being received from the GUI when the user enters a location/speed or presses a button. Visually using Gazebo to see what happens is also a good test, because it will show the location in the world and the vehicle with its speed put into account. A generated world file and working URDF of a vehicle in the world must be made first, before proceeding with the UI.
- Requirement: The program shall generate the environment with a minimum real-world dimension of 1000x1000m at a time.
- Test: Run the world file generated and measure the length and width of the world. If it's 1000x1000m, then the first step succeeded. Include the vehicle URDF file in this chunk. Move around the vehicle until the next chunk of the world is loaded in. If this chunk measures 1000x1000m, then the process succeeded. One would need the empty world file and SDF models to visualize this in a more efficient way, as well as the URDF vehicle file to run this test.
- Requirement: The program shall base its environments on real-world data.
- Test: Use OpenStreetMap to generate meshes of a given area and if the meshes do not come through the transition to an SDF file, then the process wasn't successful. A conversion from the OSM data to an SDF file must be made, before testing this case. A world file containing this SDF data must be made as well.

- Requirement: The generated world shall include world-accurate terrain heights.
- Test: Using Gazebo and OSM, one can take the OSM data and compare it to the visuals in Gazebo, using the Gazebo height map. OSM data must be generated in Gazebo, before proceeding with this test.
- Requirement: The generated world shall include roads with world-accurate positions, connections, and lengths.
- Test: Using OSM data and Gazebo, road SDF files can be generated and compared to real-life locations. If these real-life locations compare well with the generated Gazebo models, then this test is successful. OSM data needs to be converted to SDF files that Gazebo can use to generate the models before this test can be done.
- Requirement: The generated world shall include buildings with world-accurate footprints and heights.
- Test: The generated Gazebo simulation will be compared with real-life locations and if they compare well, then this test was a success. OSM data must be transitioned into SDF models for the buildings using different algorithms in order to test this case.
- Requirement: The program shall fulfill all other requirements in city block, neighborhood, and freeway environments.
- Test: Separate world files can be made in order to test the vehicle in different areas. These world files can be smaller in order to run tests in them. If the vehicle returns correct data, the GUI returns correct inputs from the user and everything looks correct in the Gazebo simulations, then this test is successful. World files with models, including the URDF file for the vehicle and the GUI must be made in order to test this case.
- Requirement: The program shall generate a world around the reference point such that the vehicle's sensors won't detect the unloaded world.
- Test: Generate a world file and see if the vehicle is the center/reference point of this world. Also verify that the sensors are not protruding outside the generated world, by giving the sensors a designated radius. If the car is centered and the sensors are not collecting empty chunk data, then this test is a success. A world file, with a model of the URDF vehicle file must be made, and a node to collect the sensor data of the world must be made before this test can be done.
- Requirement: The program shall start the simulation from a reference coordinate.

- Test: A world file will start the simulation in a designated coordinate area. If this starting position is not the world file start location specified then this process did not work. A world file for the start position must be created, with models for visuals, before this test can be done.
- Requirement: During the simulation the vehicle shall stay on the roads limits.
- Test: Simulate the vehicle in Gazebo with the road models included. If the car steers into the road edge and does not go past it, then this test is successful. Limits with the vehicle URDF file and the generated world file must be taken into account, before proceeding with this test.
- Requirement: The speed of the vehicle shall be constant.
- Test: Once the world and vehicle are generated, and the constant input velocity in the GUI is set, view the topic that the vehicle velocity is subscribed to and verify that this velocity is relatively constant. If it is, then this test is successful. Nodes must be created, with the world file, and URDF file, and a node must be subscribed to the URDF file before running this test.
- Requirement: The simulation shall be generated on Gazebo.
- Test: Once the files all are created and everything looks correct, Gazebo will run the simulation to show that it works correctly. If Gazebo has problems generating any files or models are missing in Gazebo, then this test has failed. All other B requirements and tests should be done before proceeding with this test.

## 8.2 A Requirement Tests

- Requirement: The car shall navigate between two GPS coordinates, following the roads where possible.
- Test: If the user inputs another set of GPS coordinates and the vehicle moves along the roads to those coordinates, and does not get stuck on any of roads to that location, then this test was successful. An updated GUI and URDF file that will move the vehicle accurately along roads to a second set of GPS coordinates must be created in order to verify this test.
- Requirement: Dynamic speed will change based on the environment.
- Test: Run the vehicle URDF file in the simulation and if the velocity slows down when the car is going to collide with anything, or if the vehicle speeds up when there's nothing in the way, then this test was successful. The URDF file must be updated in order to change velocity based on the collision settings with other objects and update the nodes receiving the URDF data to reflect this as well, before running this test.

- Requirement: The quality of generating the world around the car will improve.
- Test: Update the world file with more improved models (i.e., if the model isn't a .dae file, try to make it a .dae file), and compare visuals of the world models before and after they change file types (if possible). If the new world generated has better quality of models than the original world, then this process succeeded. The complete, simulated world must be made before proceeding with this test.
- Requirement: The program will Generate obstacles along/on the road, such as people, cars, streetlights, etc.
- Test: The completed world file can be added onto with models that can move or interact with the environment, and if these models are included in the world when it is generated, then this test was successful. The completed world must be made before proceeding with this test.
- Requirement: The program shall generate a scenic environment. (water, trees, building aesthetics, cacti, etc.)
- Test: The completed world file can be added onto with more ascetic models, and if these ascetics are included in the world when it is generated, then this test was successful. The completed world must be made before proceeding with this test.
- Requirement: The generated world shall have common traffic control such as stop lights, crosswalks, and stop signs.
- Test: This test will be proven correct if the updated URDF vehicle file takes into account common traffic control with relative precision, when these models are put into the world. A completed world file that includes these new models and an updated URDF file that takes these models into account with updated nodes that correctly perceive the URDF data must be made before proceeding with this test.
- Requirement: The user shall control the car's movement with a keyboard.
- Test: Keybinding different keys to the nodes that control the movement of the vehicle can be used to move the car in the simulation. If all these keys are successful in moving the car around, then this test is successful. An updated URDF and updated vehicle control nodes that take these keybind events into account must be made before proceeding with this test.

# 9 Integration with Platform

## 9.1 data acquisition platform

To collect the data used to generate the world, we interface with several web API's. To collect elevation data, we use open-elevation.org's API at endpoint

/api/v1/lookup. This API takes a json object containing query points of the following form:

{"locations":[{"latitude":...,"longitude": ...},...}

and returns a new json object of form

{"results":[{"latitude":...,"longitude": ...,"elevation":...},...}

To collect data on roads and buildings, we use openstreetmap.org's Overpass API. This is an alternative API to OSM's generic API, and is optimized for reading from the database. This API is hosted on multiple different servers due to the open source nature of the project. Queries are made using Overpass Query Language (OQL). For example, we can fetch all building footprints in a certain area using the following query:

way["building"](30,20,31,21); out geom; relation["building"](30,20,31,21);out; way(r)[!"building:part"]; out geom;

## 9.2 Gazebo integration

Gazebo has a Gazebo node that can connect to ROS for communicating purposes. ROS commands can be used to launch world files in a Gazebo simulation and add different models into the running simulation. The data presented from OSM is converted to an STL file, via the Model Generation Algorithm, which gets put into an SDF file, which gets put into a world file, and this world file can be launched in Gazebo. Other SDF files with different models can be put into the running simulation via the spawn model command, which asks for the model needed, and location to place said model, as well as the orientation of the model. A URDF is the file that will work with ROS and Gazebo. This file is similar to an STL model, except that it has extra parameters that can be used in ROS to take data from Gazebo and Publish said data, as well as have parameters implemented into it using a GUI for control of the vehicle. The vehicle used in the simulation will be a URDF file for the reason of collecting sensor and collision data from it. A chunk will be generated and taken away based on the location of the car relative to the world edge. This can be done by simulating different coordinates of the chunks and add them in when the vehicle limit is reached, partially deleting a chunk behind the vehicle.

## 9.3 rqt integration

rqt is a software framework of ROS (Qt-based) that implements various GUI tools in the form of plugins. The creating of plugins for rqt can be done either on Python or C++. There are 3 metapackages which provide different functionality.

| Requirement | Rory | Sahachel | Michael |
|---|---|---|---|
| B1 | 0 | 1 | 0 |
| B2 | 0 | 0 | 1 |
| B3 | 1 | 0 | 0 |
| B4 | 1 | 1 | 1 |
| B5 | 1 | 0 | 1 |
| B6 | 1 | 1 | 1 |
| B7 | 1 | 1 | 1 |
| B8 | 0 | 1 | 1 |
| B9 | 0 | 1 | 0 |
| B10 | 1 | 1 | 1 |
| B11 | 0 | 1 | 0 |
| B12 | 1 | 1 | 1 |
| A1 | 1 | 1 | 1 |
| A2 | 1 | 1 | 1 |
| A3 | 1 | 1 | 1 |
| A4 | 0 | 1 | 0 |
| A5 | 1 | 0 | 1 |
| A6 | 1 | 1 | 1 |
| A7 | 1 | 1 | 1 |

Table 1: Table of Implementation of Requirements to Group Members

The firs one is rqt which as mentioned before, it provides ROS GUI functionality. The second option is rqt_common_plugins which main functionality is to provide ROS back-end graphical tools. The last plugin is rqt_robot_plugins which is used for robots ruing their run-time. For the gui, rqt is the best option since the UI display to the user very few options and the goal of it is to gather and display current simulation data. A python file, which includes the necessary plugins to create the desired gui will be created. After creating the gui, the UInode will use the python file to generate the gui. Then the node will gather the information it needs and create subscriber/ publisher to share the data with ROS and gazebo.

# 10 Task allocation and breakdown

Table 1 shows the distribution of all requirements between team members. If a 1 is under the name, then that person has a part in that requirement and if a 0 is under the name, the person does not have a part in the requirement. Multiple people can be part of the same requirement and all team members are a part of every test case for each requirement.

## 11 Timeline for completion

## 12 Global/shared tasks and experience

When designing the project, we each researched a different area related to each of the three nodes. Since this has given us each some specific experience, we've divided the tasks along the same lines. Rory will implement the Generation node since he researched how to collect and parse OpenStreetMap data, and how to programmatically create 3d models based off of these. Sahachel will implement the UI node since he researched how to create GUI's with simulink and rqt. Michael will implement the simulation node since he researched how to programmatically spawn and move objects in Gazebo using ROS.

Testing will be a shared task, so that the person who tests the code is not the same person who wrote it. Additionally, we will all share in the task of test case creation and scenery model content acquisition.

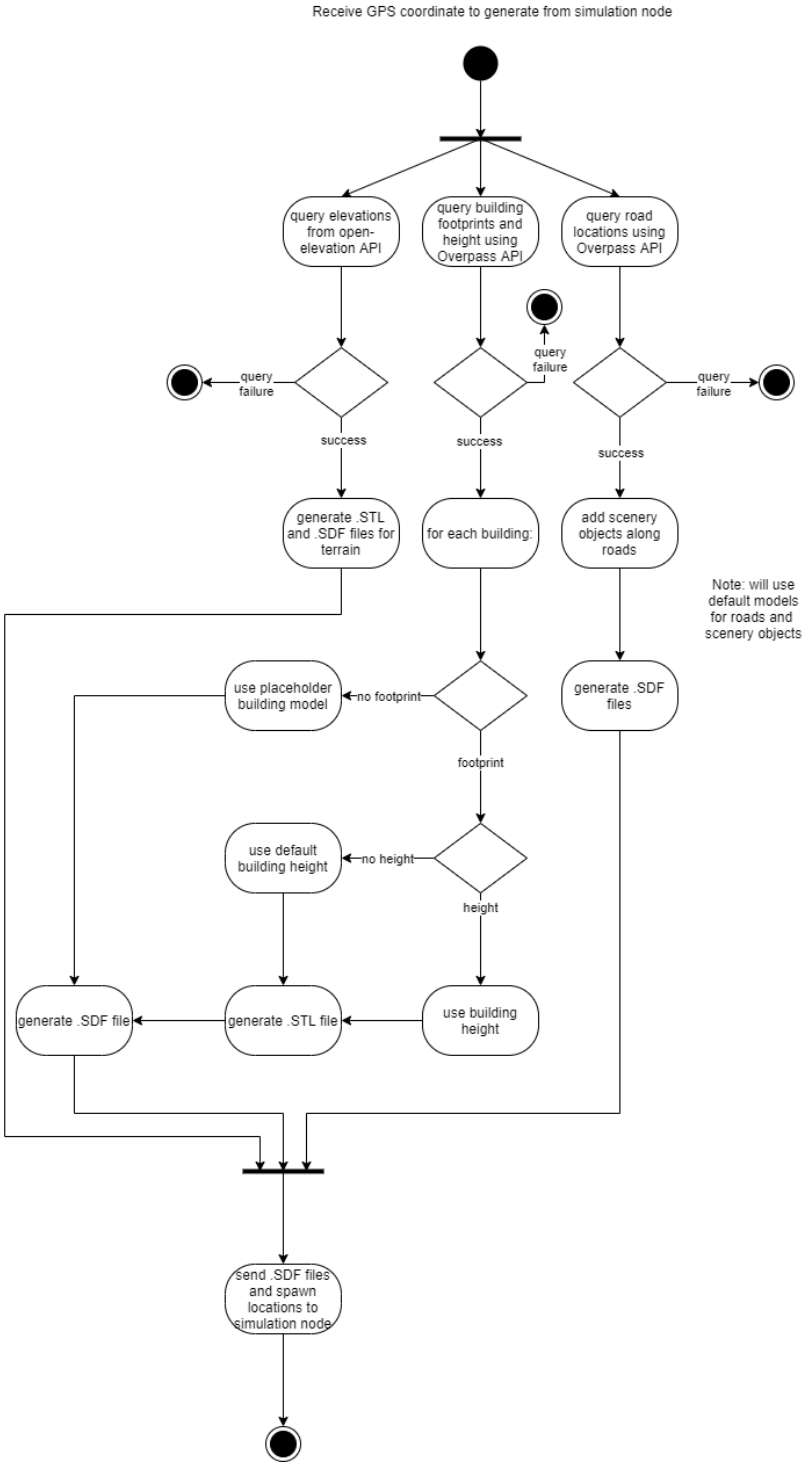


Figure 11: Activity diagram outlining execution of the Generation node after a new coordinate is received

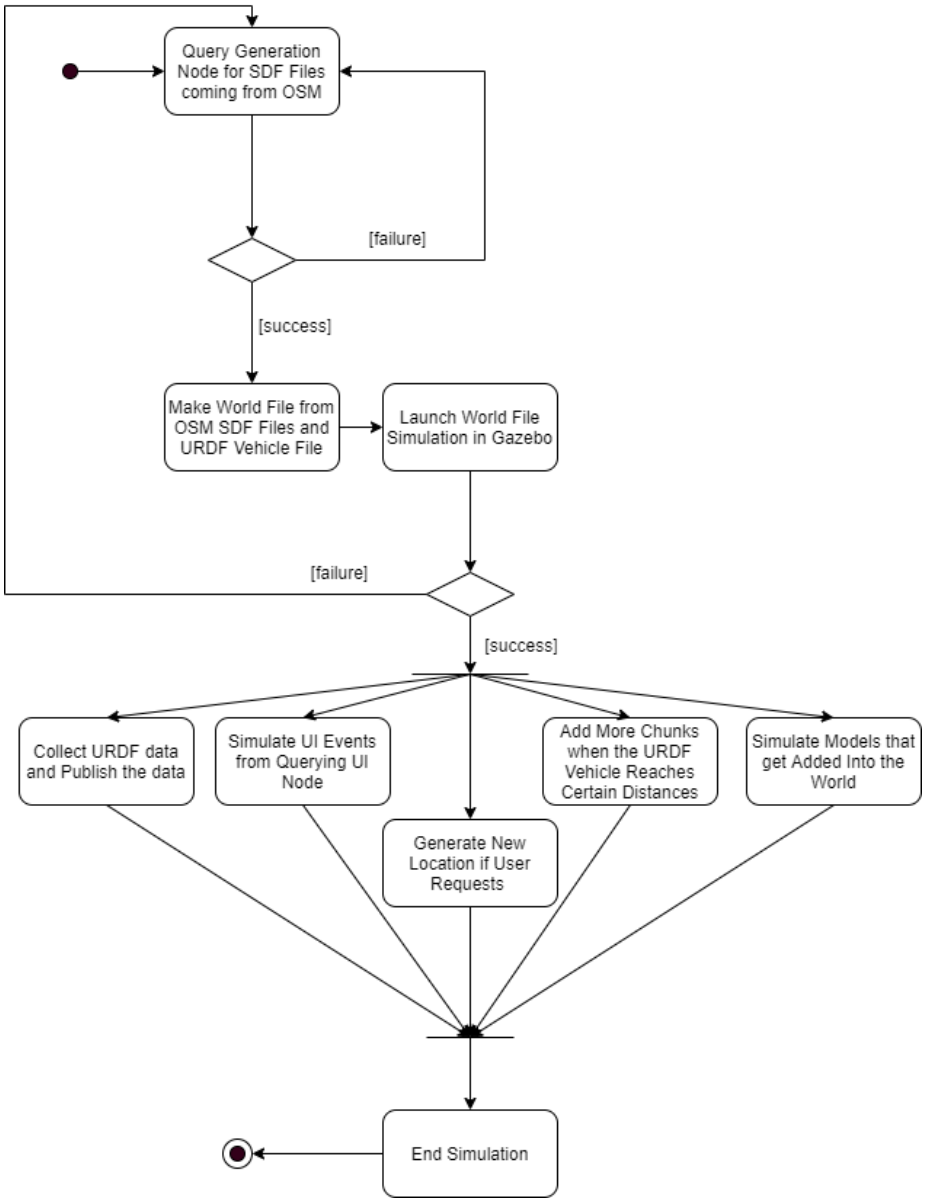


Figure 12: activity diagram of the simulation node's desired behaviour

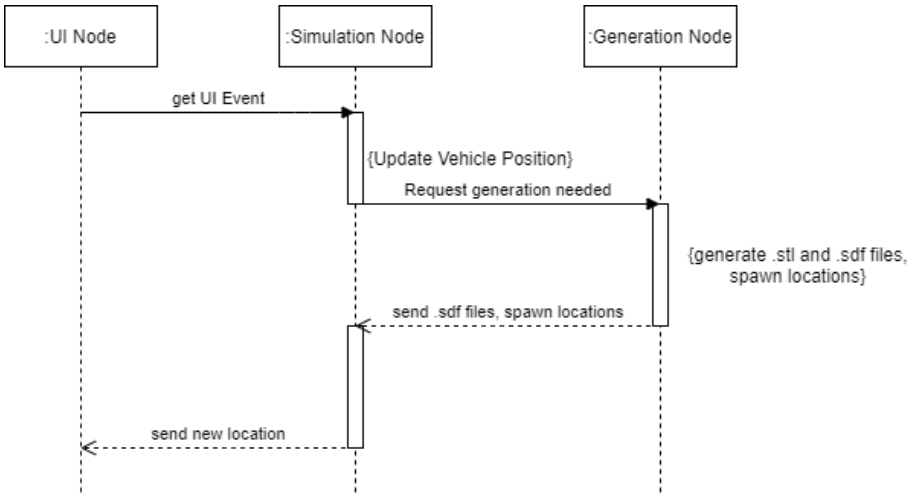


Figure 13: Sequence diagram of the interactions between the three ROS nodes

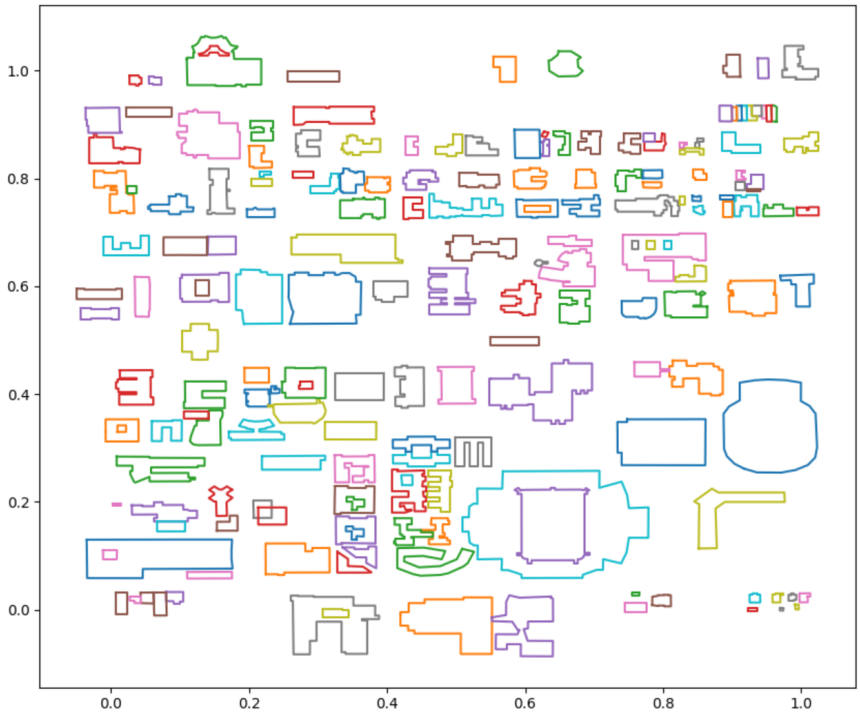


Figure 14: example building footprints generated from openstreetmap data, portraying the U of A campus

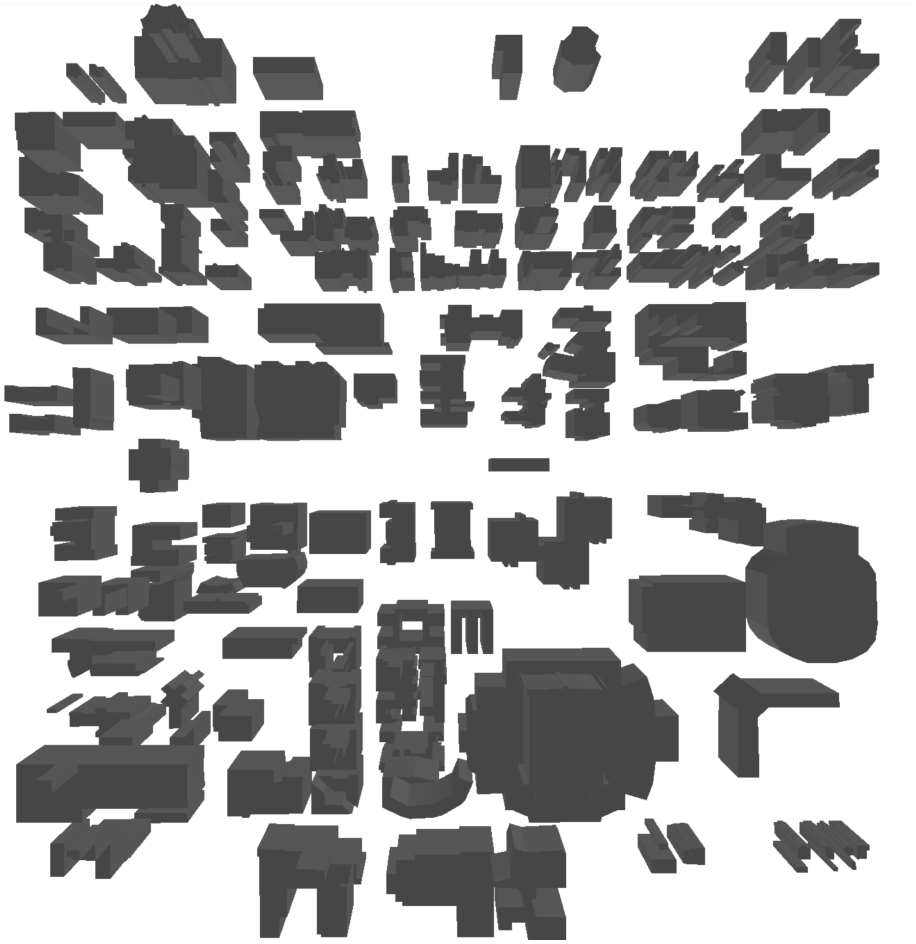

Figure 15: 3d models extruded from the footprints of buildings on the u of a campus

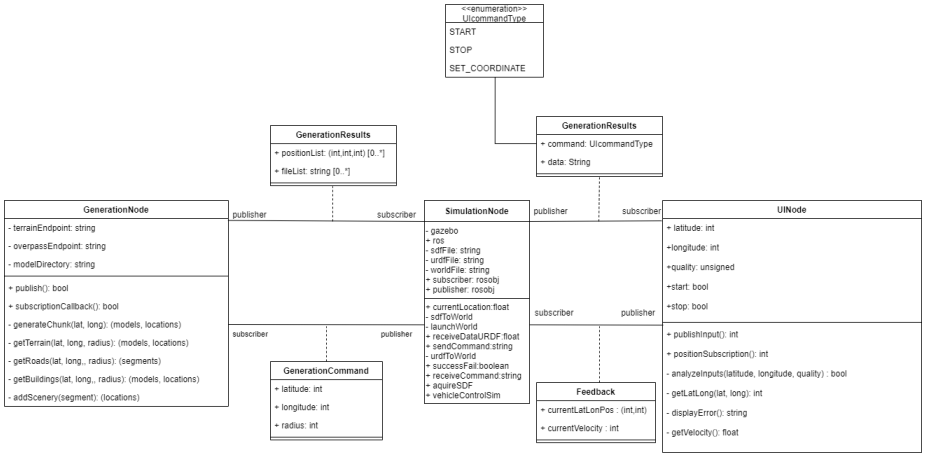


Figure 16: Class Diagram for project

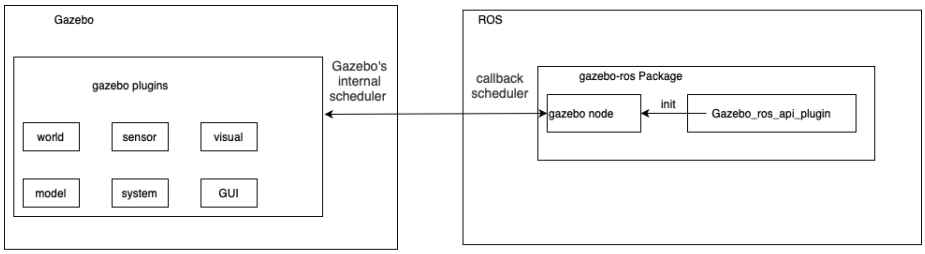


Figure 17: Gazebo-ROS communication using gazebo-ros package

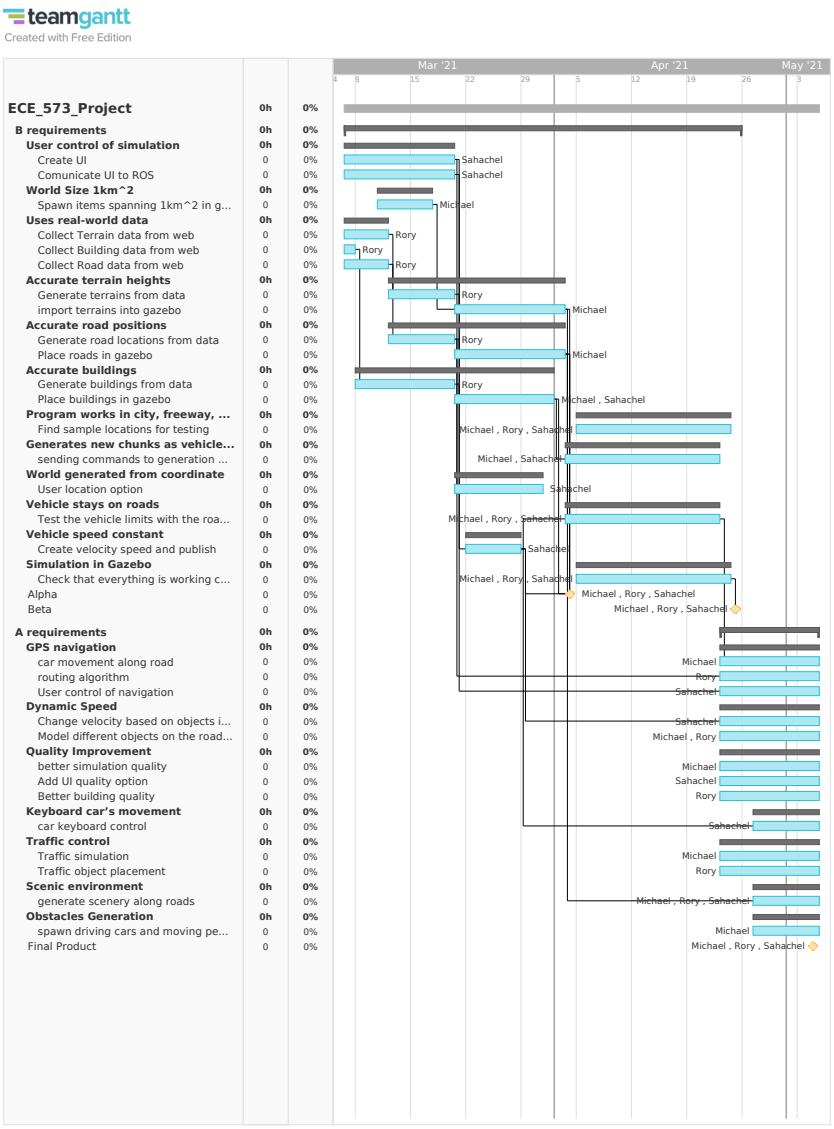


Figure 18: gantt chart of our project