Robotics Simulation on Unreal Engine

TEINTURIER Gabin

May - September 2025

RMA.be Brussels
The only Belgian military university
ROBOTICS & AUTONOMOUS SYSTEMS

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Special Thanks

I would like to thank M. Émile Le Flécher and M. Emmanouil Maroulis, robotics researchers at the Robotics and Autonomous Systems Lab at Brussels' Royal Military Academy for their support and assistance through this internship.

Introduction

Unreal Engine (UE) is an open-source 3D game engine developed by Epic Games. Released for free to download with its version 4.0 on April 2014, it then established itself as a pillar on the game making scene, sharing with Unity most of the market shares. Its strengths, including C++ based code, portability on most platforms, multithreading, adaptive level of detail, raytracing and procedural generation made it a leader in the field of realism and heavy environment generation as can be seen in world famous titles such as Fortnite, Valorant, Harry Potter : Hogwarts Legacy or more recently Black Myth Wukong. This capacity to handle heavy environments with thousands of objects led new industries outside of gaming to use it as a tool. Amongst those, we can quote the film industry with the Star Wars series The Mandalorian using Unreal footage as background instead of classic green screens, and immersive VR experiences to train people in various very realistic situations. For instance, the Royal Military Academy (RMA) developed a VR environment to train firefighter personnel for the Navy. This diversification in usage can be seen immediately after opening the Unreal Editor with templates projects sorted into 5 categories: game, film/video, architecture, automotive product design and simulation.

The last one of those categories is the one I have been focusing on during this internship. Indeed, robotics computer simulations have played an important part on the development of new solutions in the past years as they are cheaper, faster and safer than tests on real world systems whilst keeping a physical behavior close enough to the real world to make sure that the control used on the simulation can be ported onto real systems. The most well-known software for robotics simulation is Gazebo, praised for its precise physics engine DART (Dynamic Animation and Robotics Toolkit) and seamless integration with the ROS2 framework, an industry standard in the robotics field. What makes Unreal Engine stand out as a simulation software is its capacity to

produce ultra-high-quality environments which is great to train robots with cameras that are meant to be used in a real-world setup. On the other hand, Unreal's physics engine is optimized for performance, so it is not as precise as DART, but this issue is moderated in the case of simple robots such as four-wheeled UGVs (Unmanned Ground Vehicle).

The objective of this internship was to familiarize with the Unreal Engine Editor and its possibilities, recreate the robot Summit XL from Robotnik [1] inside the editor, link it to ROS2 to be able to launch missions, generate several realistic environments and finally use the ROS2 framework Nav2 to launch a GPS waypoint following mission.

Figure 1 – Photo of the Summit XL rover

I. Working with Unreal Engine

Never having used the Unreal Engine beforehand, I had to learn everything about it. This goes from learning how the unreal projects are structured to learning how to use the various tools available.

1. Structure of Unreal projects

The first thing to know is the structure of an Unreal project. It is as follows:

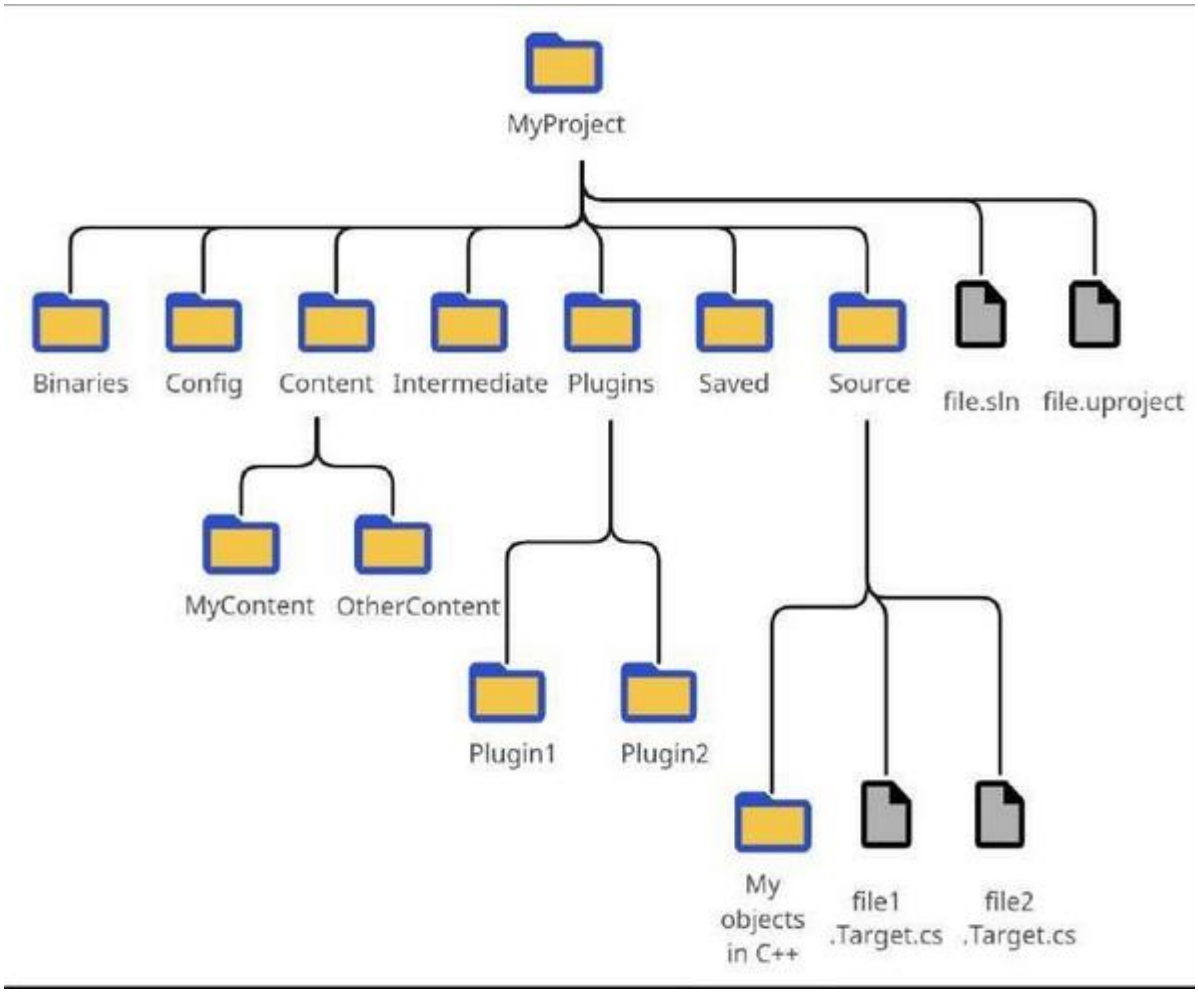

Figure 2 – Structure of an Unreal project

The folders are separated into two categories, the ones containing the data and the ones created at compilation. Here is how it is organized:

- Config: this folder contains the configuration parameters for the editor. It should remain untouched as the parameters can be modified directly from within the editor.
- Content: this folder contains all the assets used inside the project. These go from static meshes (3D models) to animations, including materials, textures, custom user blueprint classes and in general everything user imported.
- Plugins: this folder contains packages of additional content made by other users of the Engine that enable new functionalities inside the editor. Plugins are an important part of the Unreal economy as many ones are coded and sold by people exterior to Epic Games on the official Unreal marketplace.
- Source: this folder contains all the user custom C++ code used inside the project.

The remaining folders (Binaries, sometimes DerivedDataCache, Intermediate, and Saved) are the ones generated at compilation. It is good to know that, when an issue occurs with the editor, sometimes a solution is to delete these folders and let Unreal recompile the project. The Saved folder contains the in-editor screenshots, so it is recommended to make sure that you don't want to keep them before deleting it.

It is also interesting to add the name of the downloaded plugins inside the Source/ProjectName/ProjectName.Build.cs file to be able to use their defined classes and functions inside your own code. This is done by adding your plugin names in this line of code (here RapyutaSimulationPlugins, rclUE and CesiumRuntime are examples of plugins I used that will be discussed later):

```
PublicDependencyModuleNames.AddRange(new string[] { "Core",
"CoreUObject", "Engine", "InputCore", "Foliage", "CesiumRuntime",
"RapyutaSimulationPlugins", "rclUE", "RHI", "RenderCore", "HTTP" });
```

2. Functionalities native to Unreal

Unreal Engine comes filled with a lot of built in features. In this paragraph, I will only discuss the ones useful for the internship.

- Manual Environment making with Landscape and Foliage editors:
  Unreal has several modes including Landscaping, which allows to create a floor and sculpt it by adding mountains, holes and noise to make it more interesting, and Foliage, which allows to easily add static meshes onto the landscape by painting it. Below is an example of what can be done with these tools.

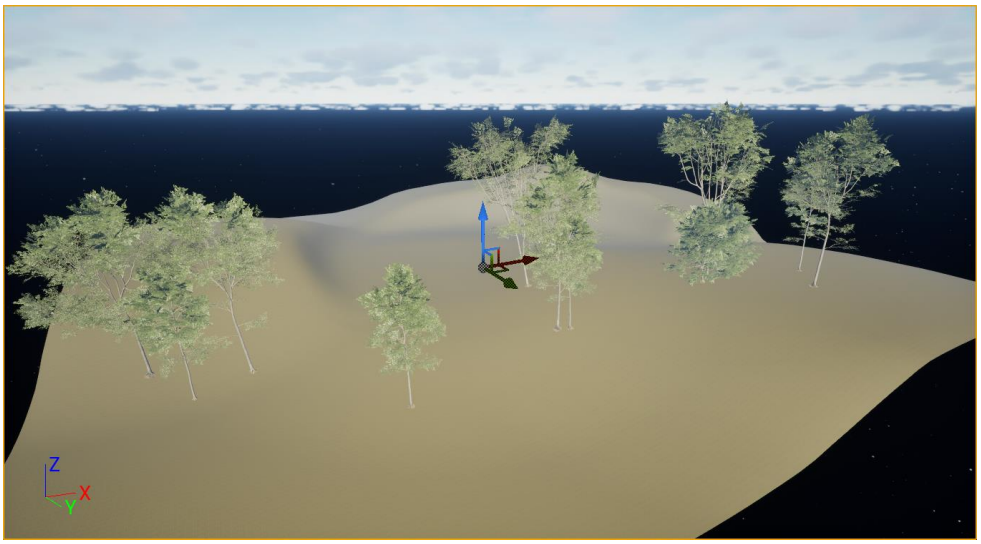

Figure 3 – Simple environment with terrain sculpted with landscape mode and trees added with foliage mode

- Ready to use assets: some assets are defaulted on the projects templates like Third person controllable characters, simple shapes 3D models, … To get more specific assets, the official Unreal marketplace is accessible from within the editor and is packed with free assets and features. An important one is Quixel assets. Those are very detailed real-life materials and objects that were scanned, transformed into Unreal assets and provided for free to the users.

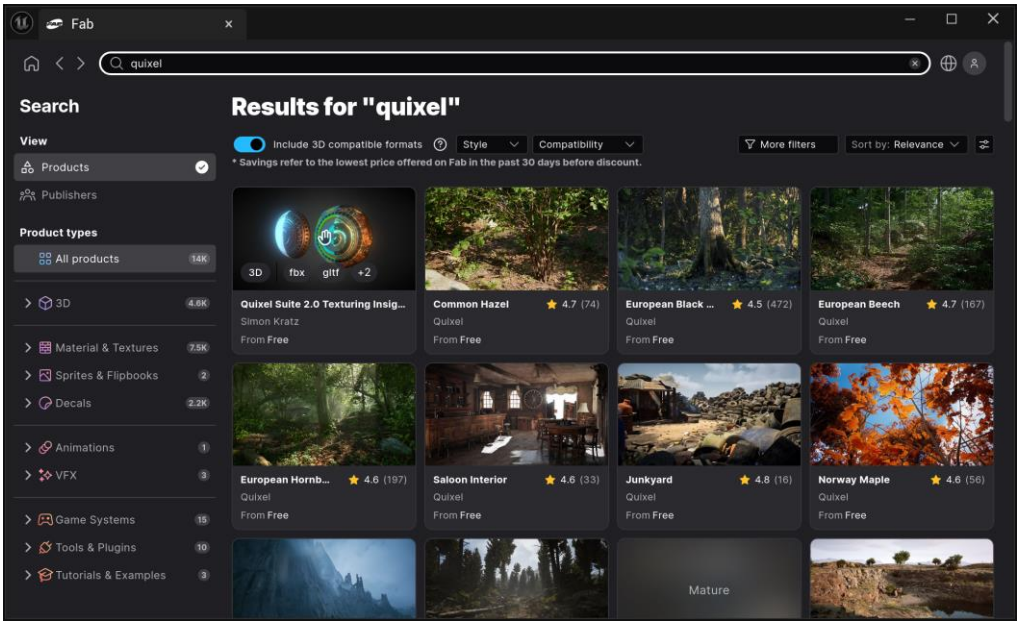

Figure 4 – Some Quixel assets seen in the marketplace inside the Unreal Editor

- Blueprints: in Unreal Engine, a Blueprint is a node-based visual scripting system that allows gameplay mechanics, interactions, and behaviors for games without writing traditional code. They are used through Blueprint Classes that are just like C++ classes but enable a more visual way of working, better understanding and modification.

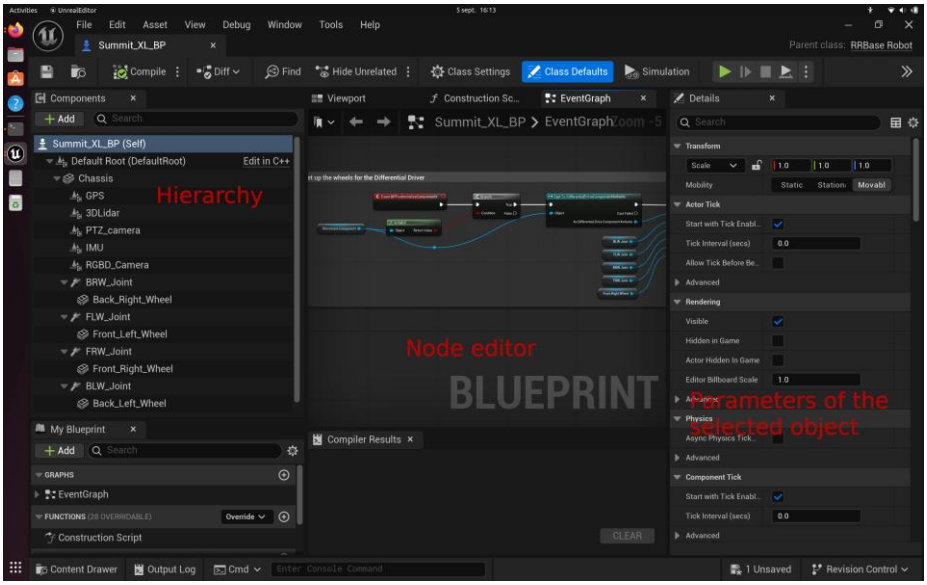

Figure 5 – Interface of the blueprint editor

II. Recreating the robot inside Unreal

1. Physical structure

This step was straightforward, indeed, the structure for a robot is very similar to the one in CoppeliaSim, a robotics simulation software used during the classes at ENSTA. The robot was built as a blueprint class; this makes it easier to modify and understand later. The robot inherits the RRBaseRobot class (see the top right corner of Figure 5) defined in the plugin RapyutaSimulationPlugins. This is required to make it work with ROS2. This part is discussed in the next chapter.

The Hierarchy of the robot can be seen on the left side of figure 5. The main component is the chassis (static mesh) to which are attached several sensors (GPS, IMU, Lidar and two cameras) as well as the joints defining the behavior of the wheels in relation to the chassis. The static meshes of the wheels are then attached to their respective joints. The relative positions of the wheels and sensors regarding the chassis were obtained within the URDF (Unified Robot Description Format) file of the Summit XL found on the GitHub page summit_xl_common [1].

URDF is an XML-based file format used to describe a robot's physical structure, kinematics, and dynamics. It defines a robot as a hierarchy of links (rigid bodies) connected by joints. It is useful because a URDF file can be automatically imported inside Gazebo to create a robot and link it to ROS2. This is not quite as easy in Unreal since there are not a lot of plugins available to extract URDF files and they are linked to ROS1, which is not supported anymore. Moreover, the XML-based nature of URDF files makes them easy to read and understand by a human so it is fast to recreate the robot manually.

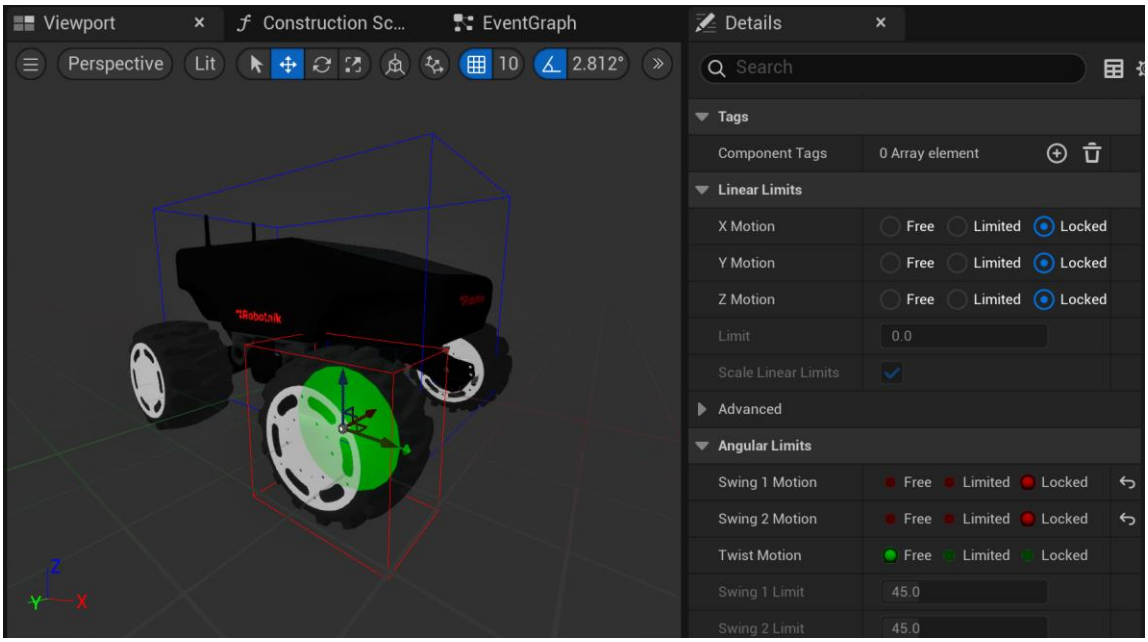

Figure 6 – View of one of the joints inside the blueprint editor

The last step in defining the structure of the robot is to configure the different joints (sensor configurations are described in the ROS2 chapter). The configuration of the joints is done by mentioning what are the two meshes that will be linked, by disabling the collision between those two meshes to avoid weird glitchy behavior and finally by defining the degrees of freedom of the joints. On the right side of figure 6, we can see that the wheels have only one degree of freedom in rotation along the twist axis. The motorization of the wheels is handled by a custom C++ code that takes the inputs published on the /cmd_vel ROS2 topic.

2. Custom collision shapes

So far, we have a robot with sensors and static meshes but no collision. Indeed, the process of importing a static mesh in Unreal is quite constrained. There are two ways of having a collision mesh associated to a static mesh. The first one is to import the mesh without a collision and adding it directly inside the Unreal Editor. This method is easy but limited since Unreal only provides 3 types of basic collision shapes plus methods to automatically generate collisions but those are not adapted to robotics because the results are often heterogenous. Yet it is possible to use the 3 available basic shapes (cube, sphere and capsule) and scale them accordingly to fit around the model, this

solution is still not the best mostly because of the lack of a cylinder shape for wheel collisions. Therefore, I chose to go with the second option which is creating custom collisions in Blender, a free and open-source 3D creation suite used to create animated films, visual effects, art, 3D-printed models, and more. Below is the result of the custom collisions shapes for the chassis:

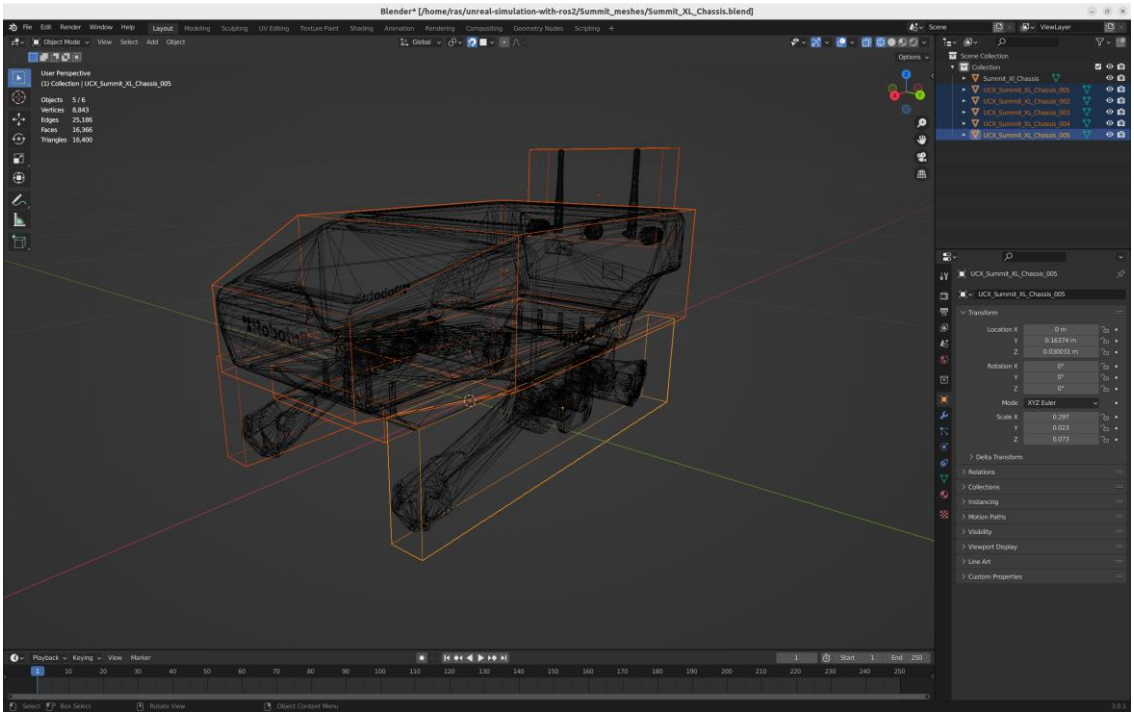

Figure 7 – Custom collision shapes for the chassis

There are several things to consider when making collisions that way. Firstly, and most importantly, Unreal doesn't accept nonconvex collision shapes. A shape is convex if, given any two points within the shape, the shape contains the line between them. To make complex collisions, it is necessary to have multiple, individually convex, collision shapes. This can be seen in Figure 7, there are 5 individual convex collision shapes, they are kept simple to reduce the amount of computation during the simulation. Finally, the collision shapes should be renamed UCX_MeshName_XXX with XXX being a number from 001 to 999 (No 2 shapes should have the same number).

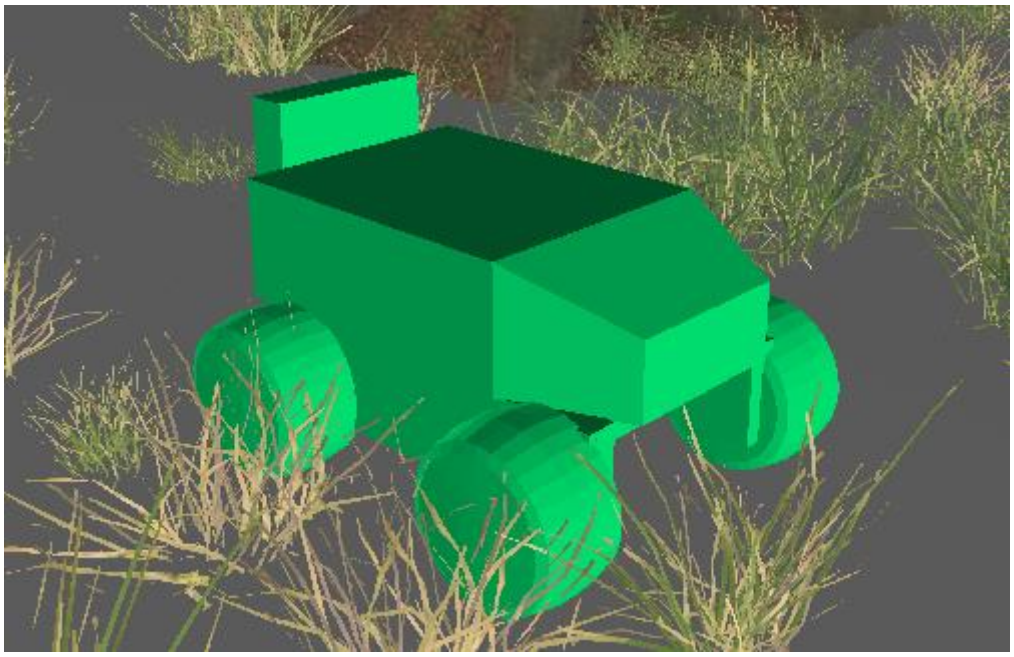

Figure 8 – Result of the collisions in the same view as the cover page of this report

With the collisions and the structure all set up, the robot is ready to go onto the next step, publishing and receiving ROS2 topics.

III. Linking the robot to ROS2

Now that we have a blueprint class of our robot the idea is to link it to ROS2. For that, we are using RapyutaSimulationPlugins [2], a plugin written especially for ROS2 integration in Unreal Engine 5. This plugin handles publishers and subscribers extremely easily. Indeed, all the sensors are already coded in the plugin, and if they are attached to an actor of type RRBaseRobot they will start publishing their respective measurements at the start of the simulation. Another advantage of Rapyuta is that it prefixes all the topics with the name of the robot which makes it easier for multi-robot simulation.

Therefore, the main operations required were to make sure the data published by the sensors was coherent and there was also a need to make a custom Movement Component (the one that gets the /cmd_vel values and moves the wheels accordingly) since the only one already in the plugin was for 2-wheeled robots and ours has 4.

1. Testing the several sensors

I wrote very simple python scripts to test the lidar and cameras. I didn't write code to test both the imu and the GPS because I didn't find an added value of plotting the measures of these sensors in python rather than getting them with a simple "ros2 topic echo" command. Below are the results of the tests for the lidar and camera sensors, where the measures are gathered through ROS2:

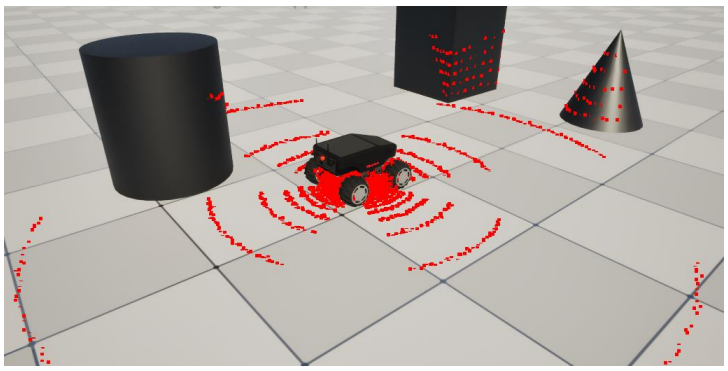

Figure 9 – In editor view of the test scene (the red points are the lidar measures)

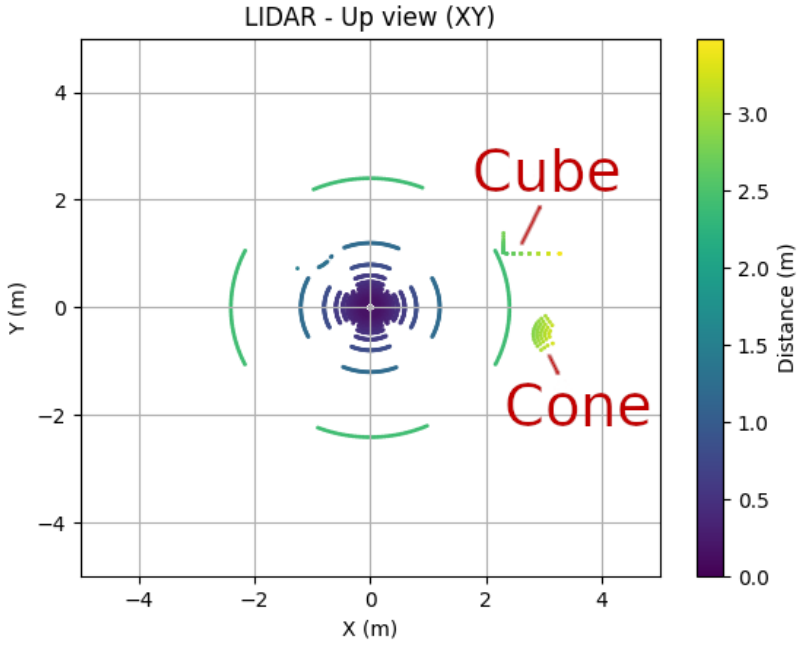

Figure 10 – View of the lidar points recovered from ROS2

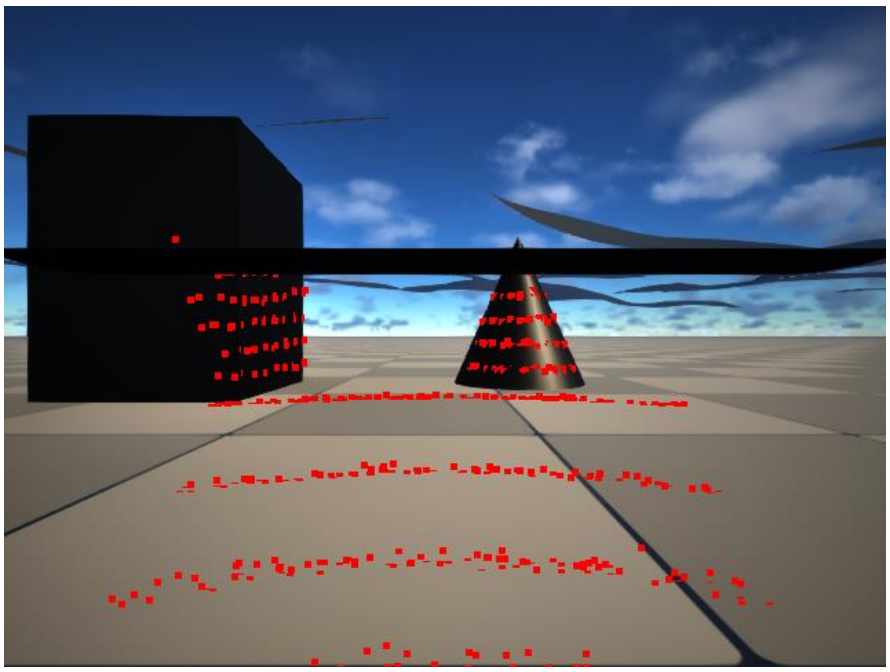

Figure 11 – View of one of the cameras recovered from ROS2

2. Custom Movement Component

The custom movement component adapted to a 4-wheeled robot was basically an extension of the provided code for 2-wheeled robots, but 2 main features were added:

- Where the wheel radius was previously left for the user to define, I changed it so that if the user doesn't mention any value, the radius is calculated from the bounding box of the wheel collision mesh.
- In the same idea, previously it was left for the user to give the distance between left and right wheels, which I changed to be automatically calculated from the positions of the joints.

These two new features enable easier implementation as the user has less things to define before being able to simulate missions.

IV. Creating realistic environments

Everything is all set up; therefore, our goal is to use the very thing Unreal is good at, terrain and environment generation. In this chapter, we will see how to create two types of environments, a city and a forest. To achieve this, we will use a plugin called Cesium for the first one and a native Unreal feature for the latter.

1. Realistic city environments: Cesium for Unreal plugin

Cesium [3] is an open platform for software applications providing 3D data all over the world. It is available on several platforms including Unreal Engine. Its concept is that it puts the entire world in Unreal. To use and understand it, I followed the tutorials available on the official website [4]. Basically, upon entering a newly created level, you should add a CesiumSunSky to have realistic sunlight and a DynamicPawn to be able to move freely around in the simulation. Those two things can be added from the Cesium panel seen on the left in Figure 12. Once done, the level is still empty. That is because no Cesium Ion assets have been added yet. Cesium Ion assets can belong to three different types: Terrain, Imagery or 3D Tiles. Terrain represents the floor with varying heights; Imagery manages what texture is applied to the terrain; 3D Tiles are data often representing buildings in various cities across the world. Those assets are added from the Cesium Ion Assets panel on the bottom in Figure 12.

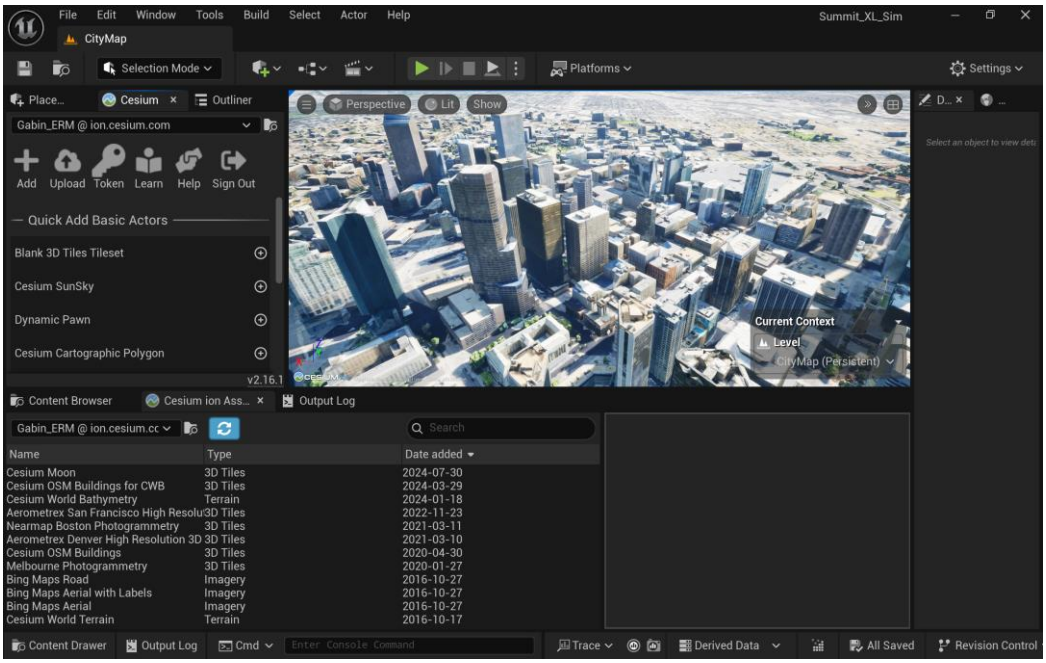

Figure 12 – Cesium usage and visual of Denver 3D tiles asset

A Cesium level should always have Cesium World Terrain, which is the terrain data for the entire world except oceans. Some parts of ocean ground can be added with Cesium World Bathymetry asset, but it is limited [5]. The basic imagery to add upon Cesium World Terrain is Bing Maps Aerial. This means that to each point in the world terrain is associated the color corresponding to its position in bing maps. Basically, the visuals gathered from bing maps are projected onto the terrain. Finally, 3D Tiles can be added and removed freely. As most of them only represent one city you can have several of them in a single level. An exception is Cesium OSM Buildings which generates buildings all around the world based on OpenStreetMap data. With this asset, the generated buildings are not textured

When learning about this plugin, I quickly stumbled upon an issue with the resolution of the 3D tiles. Indeed, most of them are generated from aerial and satellite data which makes it very convincing from far away but very blurry close by. The issue is that we want a detailed environment on a small scale for our robot. Fortunately, amongst the many 3D tiles already available in Cesium, two of them are High Resolution 3D models with Street Level Enhanced 3D. Provided by a company named Aerometrex, these models are very detailed since they use helicopter-captured data to reconstruct the buildings.

In the final map I created, named CityMap, I used these two 3D tiles and added several options:

- By pressing 0 and 1 on the keyboard, the simulation scene changes from Denver to San Francisco using a native fly to function.
- By pressing the AZERTY keys on the keyboard, the view and controls change following: A = flying pawn controlled with the arrow keys, Z = third person character controlled with the arrow keys, E = first person character controlled with the arrow keys, R = view of the robot from the back, T = view of the robot in first person, Y = top view of the robot

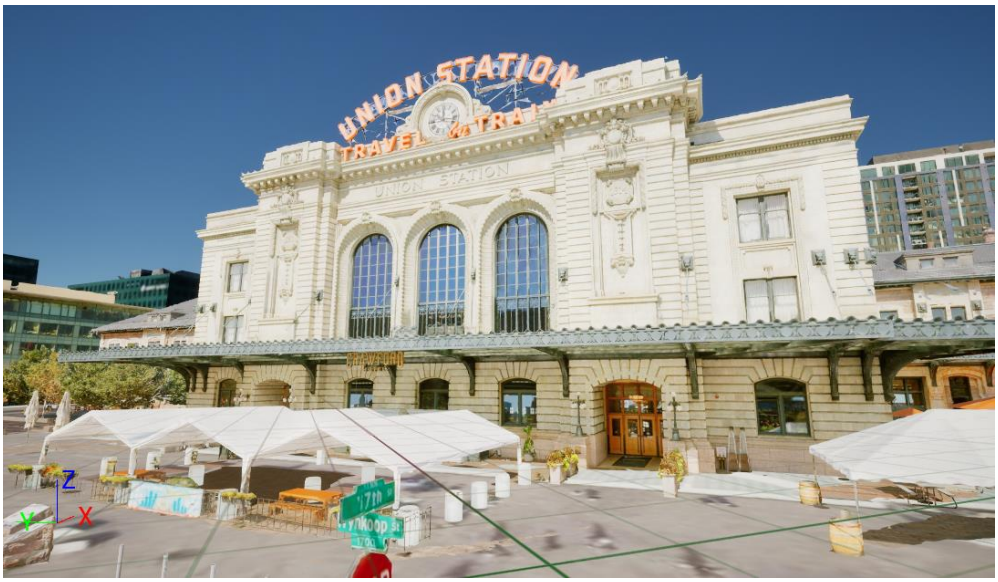

Figure 13 – High resolution 3D tile of Denver

Figure 14 – The robot placed inside its highly detailed city environment

2. New sensor to use with Cesium, the GPS

Since RapyutaSimulationPlugins didn't include natively GPS measures it didn't have a GPS sensor either. However, Cesium uses GPS measures to locate the assets placed on the globe and to make sure that they are moved when the user changes his location. This introduction of GPS data means it is now possible to create a GPS sensor.

I created a custom GPS sensor inherited from the RRROS2BaseSensorComponent class available in Rapyuta. The advantage of this class is that it automatically introduces noise in the measures, and this noise can be modified easily and directly in blueprint. The main operation to do was to get the position of the robot relative to the CesiumGeoReference object and publish it as a NavSatFix message while introducing the noise in the measures. The CesiumGeoReference is an object which puts a reference frame in the world and makes the robot work in a local plan around it. It is mandatory to make the GPS Component work.

3. Realistic forest environments: Procedural generation

The first idea that came to my mind to generate realistic forest environments was to use the landscape and foliage tools already present in the Unreal Editor (depicted in chapter I.2.). While this solution works well, it is fastidious to sculpt the landscape and hand place every asset on the landscape. Moreover, this solution is not modular at all, to generate a new environment you'll need to restart from zero.

To generate realistic natural environments, there is a need for randomness. Indeed, nature is almost never straight and even. To achieve a realistic forest environment, I chose to focus on a tool provided by Unreal Engine: the procedural generation graph (PCG Graph). This tool generates randomly a certain number of points on a landscape and there are many options available on those points to achieve different results.

An issue with PCG Graphs is that they work mostly on landscapes (there are ways to make them work on other types of objects but it's way harder) and unfortunately Cesium World Terrain is a custom object type which is not inherited from the Landscape type. This makes it very hard to use PCG Graphs on Cesium terrain. I decided to go with a second map dedicated to the forest environment. This choice brings with it one major advantage and one major disadvantage.

On the good side, since Cesium terrain texture is only satellite imagery projected, it would not have been great for forest floor since, where there are forests, the floor is textured with the top of the trees. On the contrary, by using landscapes, we can use a Quixel forest ground texture which is highly detailed since it was scanned from the real world.

The issue on the other hand remains in the fact that because the new map doesn't use Cesium no more, the GPS sensor doesn't work in the forest map. This could be avoided by either managing to make Cesium work with Landscapes objects or by creating a new custom GPS publisher that turns xyz position into GPS coordinates, which was not done during the internship.

To make the terrain for the new map, rather than sculpting the terrain by hand, I used a website [6] where, after choosing a point on a map, it outputs a black and white png heightmap of the terrain in a square zone around the selected point. Because black and white png heightmaps can be directly converted into landscapes into the unreal editor, this solution is great to retrieve realistic terrain while also being able to use PCG.

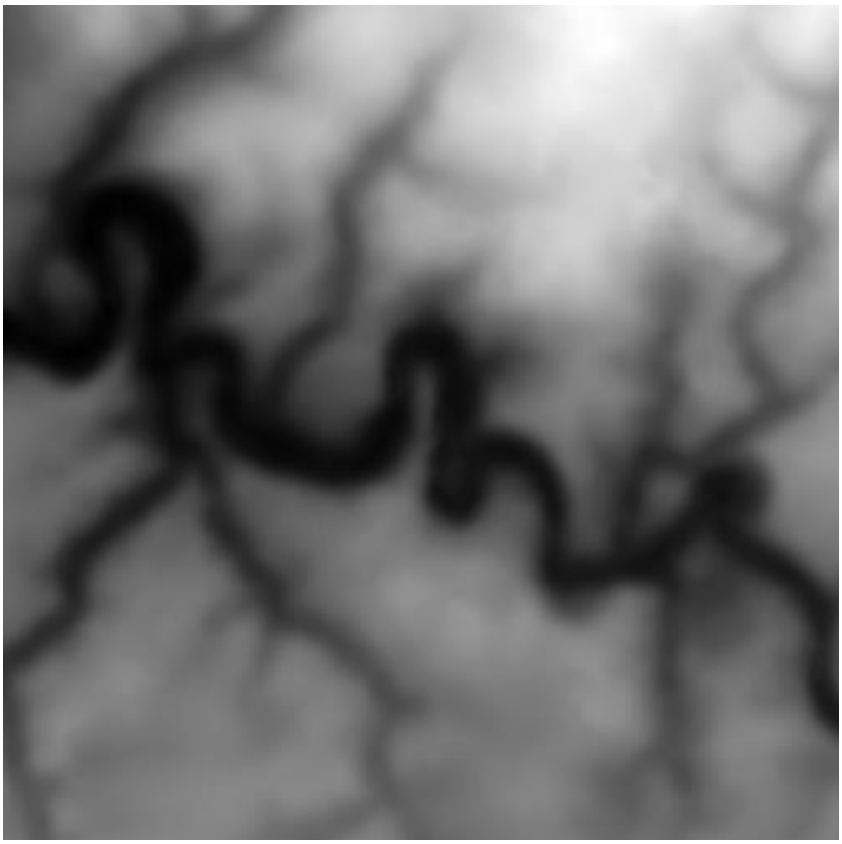

Figure 15 – Example of what a black and white png heightmap can look like. Black zones are lower and white zones are higher. Here, the black zone represents a river

Once the terrain is imported and the PCG Graph object is placed, the forest appears. After that step is done it is easy to change the distribution of the trees across the landscape. Indeed, the PCG node Surface Sampler responsible for the random generation of the trees comes with a seed parameter which is an integer that determines the placement of the random points (it is not true randomness since it comes from a seed). Thus, it is possible to change the forest by only tweaking the seed value as can be seen on Figure 16.

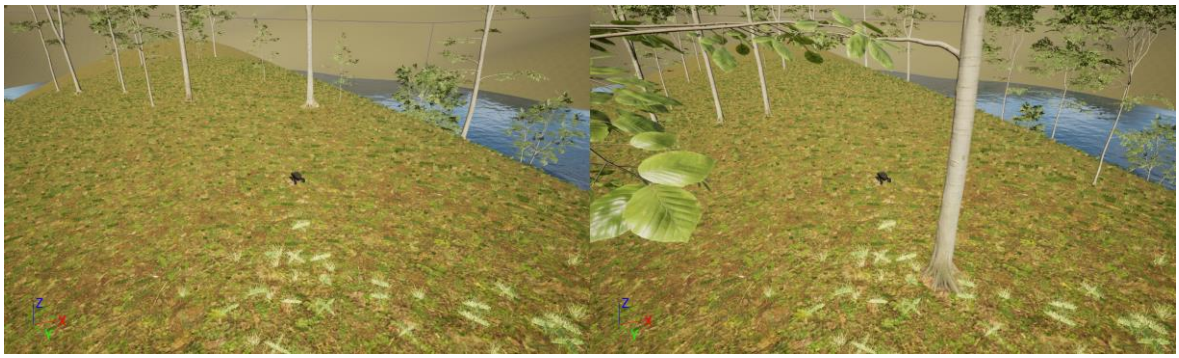

Figure 16 – Difference in tree placement when changing the seed value

You may notice two details worth mentioning in the figure above. Firstly, there are not that many trees in a forest. This is because the computational resources needed to generate a lot of objects quickly become important, even more when the static meshes are spawned with collisions activated. In our case, since the trees must have collisions enabled to interact with the robot and its lidar, it takes a lot of time to spawn multiple trees, so I decided to spawn less of them. A workaround could be to spawn the trees without collision and spawn basic cylinder meshes with collisions at the same places. This was not done because of missing time but it could be a great solution.

The other detail worth mentioning is the water on the top right corner. This water was placed in the scene to replicate the river (the landscape is the one generated from Figure 15) but it is here for visual purposes only. It does not have water behavior neither does it have collision.

4. Adding realism to the forest: the splines

To add an extra bit of realism to the forest already generated, it's good to add a path with no trees in the forest. To do so, we must use objects called splines. Those represent paths made up of points linked to one another. In the PCG Graph, there is a node called difference that allows you to spawn meshes on a terrain while avoiding a defined area. When this area is defined to be the area around the spline, it enables you to generate trees everywhere inside the PCG Graphs area except around the spline. This effect is automatic so the forest changes when the spline is changed. This is illustrated in Figure 17 below.

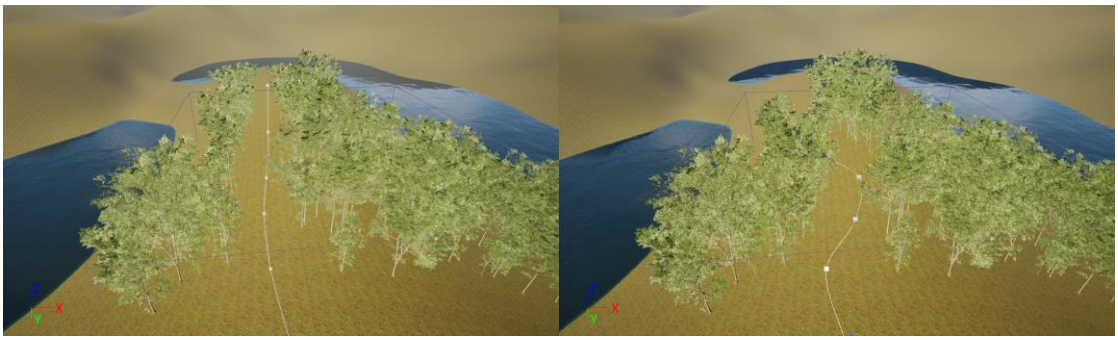

Figure 17 – Effect of changing the spline, the forest automatically regenerates to avoid having trees too close to the spline

V. Launching a GPS waypoint following mission using Nav2

Finally, everything is set up from the robot structure to its communication with ROS2 and its environment. The last thing remaining is to put it into practice by launching a GPS Waypoint following mission using Nav2, a robotics navigation framework based on ROS2.

1. Publishing the robot transform frames (tfs)

Nav2 is a Ros2 navigation framework (perception, planning, control, localization, visualisation, ...) for surface robots. This framework enables mobile robots to navigate through complex environments to complete user-defined application tasks.

Working with Nav2 enables simplicity in mission planning since all the nodes for control, localization, ... are already given and can be called directly. The main thing to do to use it is then to make sure that all the topics in the simulation are linked to Nav2 the right way, but also to make sure that all the transform frames are well defined and published.

Indeed, a transform frame (tf) is a coordinate frame that defines a specific spatial reference point or orientation in 3D space, and the transform describes how to convert

positions, orientations, and vectors between those frames. The tfs of a robot are mandatory to use Nav2 and the Rapyuta Plugin only publishes one of them which is the odom -> base_link frame (the frame that depicts the transform between the center of mass of the robot and its estimated position in the world frame). There is a need to publish all the static transform frames and the map -> odom frame.

The static transform frames refer to the tfs that will never move during the mission. Typically, those refer to the tfs between base_link (center of mass of the robot) and all the sensors_link. Since most of the time the sensors are fixed to the body of the robot and will never move, these transforms are fixed. To publish all the static tfs, I used a code given by my tutor that does exactly this once some .yaml files are filled with the right values. The values for the tfs were obtained from reading the urdf file of the robot.

Finally, Nav2 handles the publication of the map -> odom frame when working with gps waypoint following. As the terrain can be very large for this kind of application, Nav2 can't just have one map representing all the surroundings of the robots. Fortunately, the behavior of producing several maps and changing the placement of the map frame is already coded into Nav2 and nothing is to be done for that.

Finally, after launching the sensors static publisher and the Nav2 nodes, we get a full tf tree representing our robot inside its environment.

2. Nav2 GPS Waypoint Follower

To achieve the gps waypoint following mission, we planned to use the given nav2 gps waypoint follower package and adapt it to our use-case which is navigationg within the Unreal Engine. The issue is that the gps waypoint follower provided by nav2 [7] only works with ros2 Iron or newer, which wasn't my case since I used Ros2 Humble during the internship. To avoid having to manually change the code to make it work with Humble, I searched for an open-source version someone would have already modified, which luckily was the case [8]. I then tested it, and it worked perfectly. This code was made to have a gps waypoint following with the turtlebot inside gazebo. The final step was to have the robotnik Summit XL instead of the turtlebot, and Unreal Engine instead of gazebo.

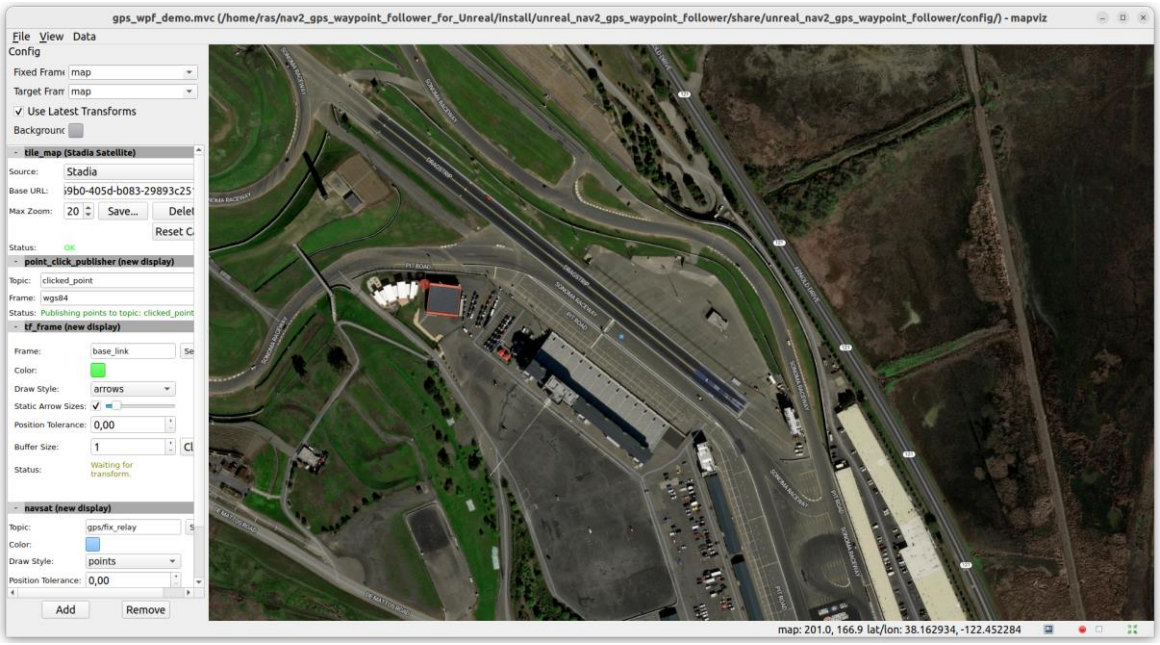

Figure 18 – view of the tutorial gps waypoint follower with gazebo and the turtlebot. Here, the small blue dot in the center is the gps position of the robot in mapviz

I couldn't manage to get this final part done before the end of the internship due to an issue with namespaces. Indeed, by default the Rapyuta Plugin namespaces all the tfs, topics and nodes related to a robot, and I tried digging into the source code of the plugin and found it very hard to remove this behavior. I then decided to try to make Nav2 work with my namespaced topics, nodes, and tfs but had a very hard time doing so. I found out that Nav2 nodes have a parameter called namespaced but that it sometimes wasn't used even when I gave a value to it. I ended the internship stuck on that issue. The furthest I went with the gps waypoint follower is that I got Nav2 to recieve the gps data from my robot, processing it through the navsat node, and having mapviz plotting the small bleu dot representing the position on Earth where the robot was detected.

Conclusion

As a conclusion, I would say I am glad for the opportunity to work on making an Unreal Engine simulation, learning the software, using it for a robotics simulation purpose, making realistic environments and launching a mission close to what is done in the field of robotics. Even though I would have liked to get all the way through with the gps waypoint follower, I am still grateful for learning a lot of things regarding simulation, software and even game design. I really enjoyed this internship at RMA and wanted to thank again my tutors and people at RMA for their reception and for making it possible.

Annexes

All the code and assets are available on the following GitLab repo (only those with an ENSTA account can access it): https://gitlab.ensta-bretagne.fr/teintuga/unreal-simulation-with-ros2

Bibliography

[1] GitHub of the Summit XL robot, used to get the URDF file and the different meshes https://github.com/RobotnikAutomation/summit_xl_common

[2] GitHub of RapyutaSimulationPlugins and rclUE, used solely to get the plugins. Rapyuta will not work without rclUE https://github.com/rapyuta-robotics/RapyutaSimulationPlugins / https://github.com/rapyuta-robotics/rclUE

[3] GitHub of Cesium, used solely to get the plugin https://github.com/CesiumGS/cesium-unreal

[4] Official tutorial page to learn how to use Cesium https://cesium.com/learn/unreal/unreal-quickstart/

[5] Information on Cesium World Bathymetry data https://cesium.com/platform/cesium-ion/content/cesium-world-bathymetry/

[6] Website where it is possible to get black and white png heightmaps for Unreal given a localization in the world https://manticorp.github.io/unrealheightmap/#latitude/27.9944014110462/longitude/86.9252014160156/zoom/13/outputzoom/13/width/505/height/505

[7] Tutorial Nav2 GPS Waypoint follower https://docs.nav2.org/tutorials/docs/navigation2_with_gps.html

[8] Tutorial Nav2 GPS Waypoint follower adapted with Ros2 Humble https://github.com/Gutierrez-Cornejo-Emanuel/nav2_gps_waypoint_follower_demo/tree/main