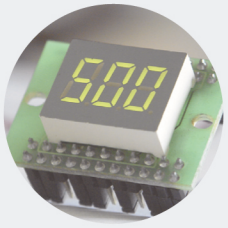
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES






Interface Homem-Máquina Multi Robótica em Unity3D

RUI RODRIGO SERRA FIGUEIRINHA
Novembro de 2020

POLITÉCNICO
DO PORTO

isep Instituto Superior de Engenharia do Porto

Multi Robotic Human Machine Interface in Unity3D

Mestrado em Engenharia Eletrotécnica e de Computadores

Rui Figueirinha
Nº 1110211

Supervisor
Alfredo Oliveira Martins

Ano Letivo: 2019-2020

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Eletrotécnica
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

Abstract

More than ever the use of autonomous vehicles to accomplish objectives deemed too dangerous or even impossible by human standards is increasing. This demand puts to the test our capabilities for managing teams of multiple robots and creating intuitive interactions with these teams is a must.

Creating means to abstract and condense the information that reaches the end user into a single kit of software would improve its manageability considerably. The development of a centralized graphical user interface is proposed to alleviate the workload of the human operator. This interface is thought out to be simple in delivering its information taking cues from video games, a well known industry in studying the theory behind the creation of user interfaces. Sensorial information is abstracted in a graphical perspective much like the attributes of a character inside a video game.

The Unity game engine was used to implement such an interface, integrating ROS with a layer of DDS to manage the communications while providing QoS settings. The DDS solves the problem of multiple ROS masters by setting up a separate network where users can connect and disconnect seamlessly from the network, without the need to restart `roscore` on each machine. Interactions between these two software is made by using `websockets` on a local network. Visual representations of the sensors onboard the autonomous vehicles transform the huge stream of data into human understandable formats for immediate response by the operator. Dynamic generation of terrain was accomplished by the use of LiDAR and side-scan sensors, if available, to map the surroundings, while Mapbox provided prefetched terrain data from OpenStreetMaps.

Resumo

Mais do que nunca, o uso de veículos autónomos para cumprir objectivos considerados demasiado perigosos ou até mesmo impossíveis segundo os padrões humanos tem vindo a aumentar. Este requerimento testa as nossas capacidades de gestão de equipas de múltiplos robôs e torna a criação de interações intuitivas com estas equipas numa necessidade.

Criar meios de abstrair e condensar a informação que chega ao utilizador final num só pacote de *software* iria melhorar a sua gestão consideravelmente. O desenvolvimento de uma interface gráfica centralizada é proposta de modo a aliviar a carga de trabalho do operador humano. Esta interface é pensada para transmitir a sua informação como um vídeo jogo, sendo que esta é uma indústria que conhecida pelo seu estudo de interfaces de utilizador. Informação sensorial é abstraída com uma perspectiva gráfica tal como os atributos de uma personagem de um vídeo jogo.

O motor de jogo *Unity* foi o utilizado para implementar tal interface integrando funcionalidades de ROS com uma camada de DDS, responsável pela gestão das comunicações, fornecendo opções de QoS. O DDS resolve o problema de múltiplos ROS *master* estabelecendo uma rede separada em que os utilizadores podem conectar-se e desconectar-se simultaneamente sem haver a necessidade de reiniciar o *roscore* em cada máquina. Interações entre os dois *software* é efetuada através de *websockets* numa rede local. Representações visuais dos sensores a bordo dos veículos autónomos transformam os enormes fluxos de dados em formatos facilmente compreensíveis por humanos para resposta imediata por parte do operador. Geração dinâmica de ambientes virtuais foi tornado possível com recurso a sensores como LiDAR e *side-scan*, caso existam, enquanto que API's como Mapbox e OpenStreetMaps forneceram dados estáticos destes ambientes.

Contents

List of Figures

Glossary

Acronym	Description
API	*Application Programming Interface*
CPU	*Central Processing Unit*
DDS	*Data Distribution Service*
DLL	*Dynamic Link Library*
DVL	*Doppler Velocity Log*
FEUP	*Faculdade de Engenharia da Universidade do Porto*
GB	*Gigabyte*
GIS	*Geographic Information System*
GPS	*Global Positioning System*
HD	*High Definition*
HID	*Human Interface Device*
HMI	*Human Machine Interface*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
IDL	*Interface Definition Language*
IMU	*Inertial Measurement Unit*
INS	*Inertial Navigation System*
ISEP	*Instituto Superior de Engenharia do Porto*
LiDAR	*Light Detection and Ranging*
LSA	*Laboratório de Sistemas Autónomos*
OMG	*Object Management Group*
OS	*Operating System*
QoS	*Quality of Service*
RAM	*Random Access Memory*
ROS	*Robot Operating System*
ROV	*Remotely Operated Vehicle*
RPM	*Rotations Per Minute*
URL	*Uniform Resource Locator*
USBL	*Ultra Short Baseline*
SBL	*Short Baseline*

Acronym	Description
SDK	*Software Development Kit*
SLAM	*Simultaneous Localization and Mapping*

Chapter 1

Introduction

The increasing complexity of the robotics world requires specialized manpower to operate. This is aggravated when we are confronted with mission scenarios that employ a team of robots, set to do different tasks, that maintain communication between themselves and the user. Allied to the integration of a plethora of different sensors, each one with their data structures and output rate further increases the complexity of the system.

Abstracting the interweave of complex systems into a palpable user friendly graphical interface that provides awareness and correct interaction with the human counterparts is the main goal of this work. These systems consist in the logging of sensors, virtual world generation and representation, mission planning, inter system communication, and providing insight of the mission at hand, all packaged into a compact software package.

The Unity game engine was used to implement all these features, given its graphical capabilities and .NET integration, backed up by a strong community with almost twenty years of development. The well known ROS (Robot Operating System) bridges the interactions between the interface and the robots, on top of a DDS (Data Distribution Service) for managing of information and providing QoS (Quality of Service).

The spectrum of mission scenarios is broad, ranging from local scale with close up detail and micromanaged manoeuvres to an overview of the robot team from far. Broad as well is the range of vehicles that get involved in these tasks. Aerial, surface and underwater vehicles constitute the different scenarios that can be simulated by this interface.

1.1 Serious games

The definition of a serious game differs within the academia and the industry. Some believe that serious games have to include an entertainment value combined with a practicality. Other researchers argue that all games have a serious purpose. This could mean that every application developed using development tools from the gaming industry can be considered a serious game. This would include simulators into the serious game category. Despite this, the most common definition of serious game is that of "games that do not have entertainment enjoyment or fun as their primary purpose". This means that serious games can the classified as video games whose primary objective is other than entertainment [1]. Given the differing opinions by the researchers it is difficult to classify this HMI (Human Machine Interface) as a serious game since it does not provide training or education to the operator even though it was developed with tools meant to be used for creating video games. It could be made to fit the wider accepted description of a serious game if in the future its capabilities were expanded to include training missions in fully simulated environments. More generally accepted examples of serious games are training simulators used by the medical, military, aeronautical, aerospace, racing industries that simulate a real world environment with high accuracy and are meant to train individuals in a controlled situation and prepare them for real world events that may occur. Games with health in mind like the Wii Sports and Wii Fit games mix fitness regimes with entertainment, while keeping track of the user's workout progress and rewarding the user for leading an active lifestyle. In [5] a Wii Fit Board (used in the Wii Fit Plus video game) was used to develop a serious game with a focus on rehabilitating stroke patients reinforcing the importance of these types of games.

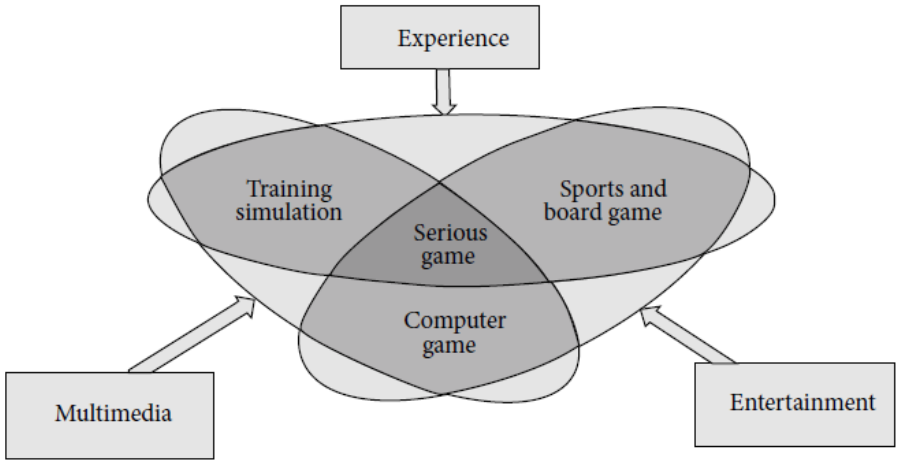


Figure 1.1: Serious game definition diagram [1]

1.2 Objectives

For creating an effective application that interfaces humans with autonomous vehicles, a set of fundamental objectives were set. The objectives are set as a guideline for creating an HMI with basic functionality. Given the flexible nature of this project, these objectives could be expanded indefinitely because the requirements are constantly changing. Real world changes, either to the vehicles, data structure changes, or to the operating scenarios would reflect on the interface.

Main objectives

- Create a centralized data and command centre
- Adapt interface to each vehicle and mission scenario
- Abstract all types of sensor data equipped by the vehicles if possible
- Implement ROS publish/subscribe features
- Integrate DDS features
- Make it compatible with existing software infrastructure
- Issue commands to the vehicles

Secondary objectives The purpose of a secondary set of objectives is to delineate bonus features that could be added to the HMI in order to improve user experience. Without accomplishing them, the resulting software should still work as expected but at a more basic level, focusing on only providing the basic functionality to the user. They can be thought as polishing touches to the final result and may not even be feasible to accomplish given the allocated time frame.

- Provide generally appealing visuals
- Allow logging of missions for later review
- User authentication
- Provide graphical options
- Integrate HID (Human Interface Device) controllers

1.3 Context

The context of this work stems from the projects carried out by LSA (Laboratório de Sistemas Autónomos). This research laboratory frequently takes part in European collaborations of autonomous missions such as bathymetry, underwater mine exploration and aerial mapping. While in the field, operating and managing these robots can be a challenging task. The creation of a centralized command centre would improve the manageability of missions providing quality of life to the operators. Currently there are graphical interfaces employed by LSA but they were developed with only one type of robot in mind and not only lack 3D rendering capabilities but are unsuitable for teams of multiple robots. The described interface addresses these issues and aims to improve and introduce new features.

1.4 Mission Scenarios

1.4.1 UNEXMIN

This project [6] employs a spherically shaped robot (UX-1, figure 1.5) to explore submerged mines, prospecting minerals. These mines are structured like a network of tunnels connecting chambers resembling a graph. Examples of these types of scenarios are Urgueiriça mine in Portugal 1.3, and Ecton mine in the United Kingdom 1.4.

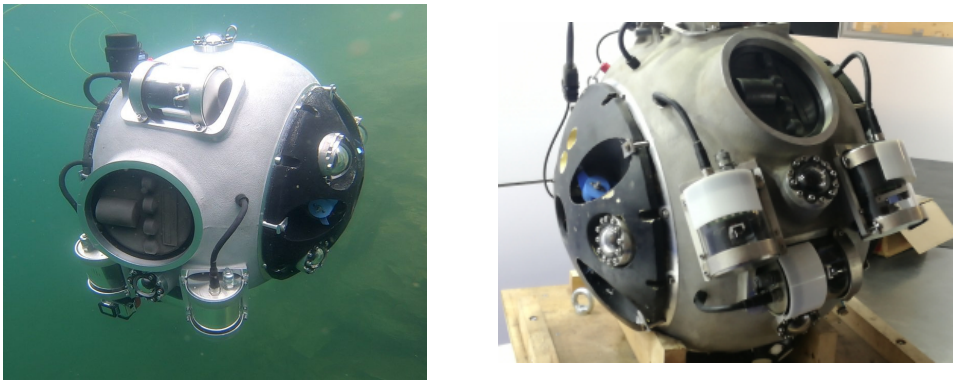


Figure 1.2: UX-1 underwater autonomous vehicle

1.4.2 Spilless

The *Spilless* project aims to contain environmental damage caused by oil spills on the ocean. It employs an aerial (STORK) and a surface robot (ROAZ II) that circumvent the spill. The former is responsible for releasing a microbial

Figure 1.3: Urgueiriça underwater mine layout, Portugal.

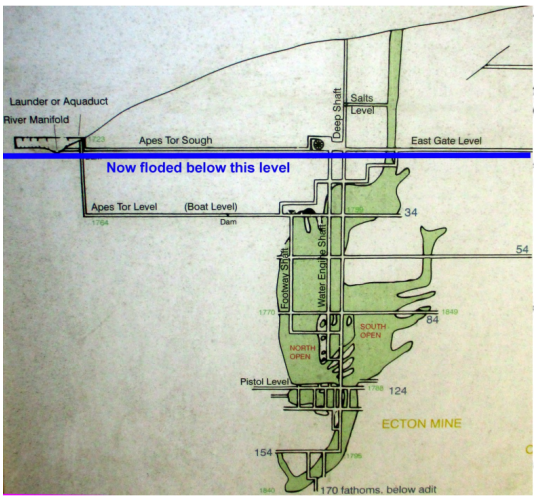


Figure 1.4: Ecton underwater mine layout, United Kingdom.

agent that breaks down the oil compounds, preventing further environmental damage.

1.4.3 TURTLE

In this scenario multiple deep sea robots (TURTLE, figure 1.6) are deployed at high depths during extended periods of time for logging or equipment transportation. This team of robots has the ability to re-position and communicate between themselves autonomously [7].

Figure 1.5: ROAZ II and STORK

Figure 1.6: TURTLE deep sea lander

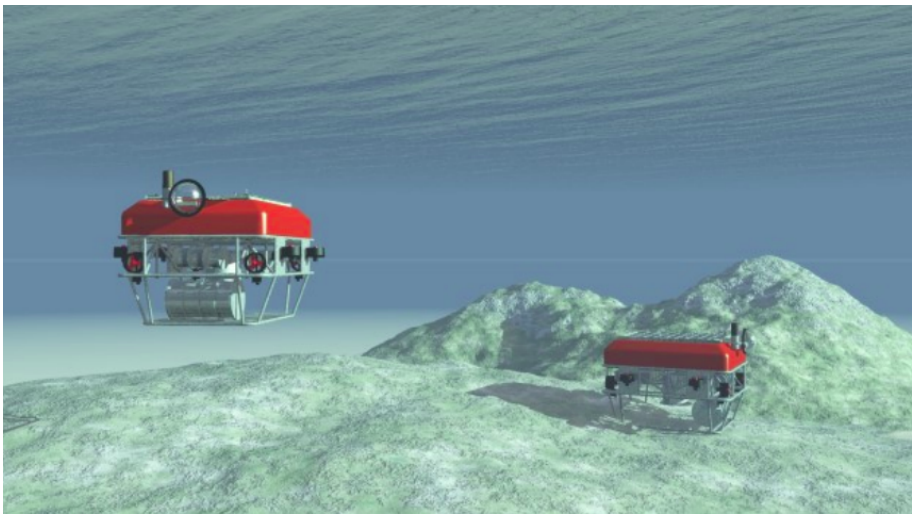


Figure 1.7: TURTLE deep sea deployment scenario

1.4.4 VAMOS

The VAMOS (Viable Alternative Mining Operating System) project [8] [9] is part of an European effort to prospect and extract mineral from abandoned submerged open pit mines. This main objective is made possible with the development of the AUV (Autonomous Underwater Vehicle) EVA (figure 1.8) for mission surveillance, combined with a positioning and navigation system. This project is meant not only for the development of robotics systems but also of its associate technologies and to determine the commercial viability of such endeavour. The robotic system is composed by an underwater mining machine, a surface launch and recovery vessel (LARV) and the aforementioned HROV(Hybrid Remotely Operated Vehicle)/AUV (EVA).

Figure 1.8: EVA underwater autonomous vehicle used in project VAMOS

Figure 1.9: Mineration structure used in the VAMOS project

Figure 1.10: VAMOS mining worksite

1.5 Structure

This thesis is divided into seven chapters. The first chapter introduces the context of this work by presenting real world scenarios and challenges when operating complex autonomous machines. The second chapter presents the studies conducted of the existing market for an application like this and shows where this work fits within others. It also evaluates the existing technologies comparing them to the used ones. The third chapter formulates the problem that this

software is fit to help solve. The fourth chapter explains various concepts and theoretical overviews of the techniques used when creating an application of this type. The fifth chapter, gives insight into the design choices associated with the creation of a user interface. The sixth chapter explains the implementations of the features that integrate the final product. Finally, in the seventh chapter conclusions of the developed work are drawn and future work is presented.

Chapter 2

State of the Art

This chapter represents existing technologies and products relevant to the project in question. The following sections represent features that inspired the development of this work, and how they shaped the final result.

2.1 Abyssal

The use of powerful 3D engines is making its way into the field of robotics and engineering as shown by the company Abyssal [2] that developed an ecosystem based on Unreal Engine 4, complete with physics simulation and photorealistic graphics operating in the oil and gas industry. They provide three products: Cloud, Simulator and Offshore. Cloud is a video repository of the underwater missions with AI (Artificial Intelligence) capabilities such as identifying underwater pipelines. It also provides a command centre for surveying worksites using GIS (Geographic Information System) technologies. Simulator is a underwater scenario simulator for training pilots in ROV (Remotely Operated Vehicle) control, assisting them in accomplishing maintenance of oil platform structures. It has a video game feel providing immersion and ease of use, along with accurate physics calculations. Lastly, Offshore has video recording capabilities and features 3D overlays for navigation assisted by augmented reality.

Its user interface is organized in order to provide real time data of the machinery in question. It provides ROV pilot training scenarios for underwater environments by establishing objectives for the pilot to complete. These comprise of simulated real world objectives like path planning and ROV control. This company's main field of actuation is supervision, planning and management of offshore oil platforms.

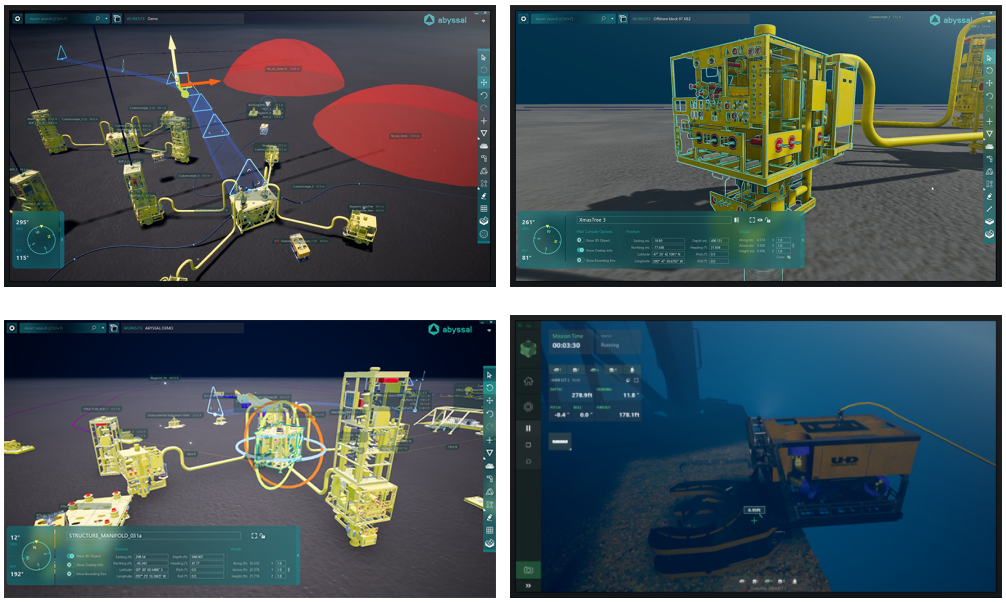


Figure 2.1: Abyssal software ecosystem [2]

2.2 Neptus (figure 2.2)

Developed in Java, the Neptus toolchain by FEUP's (Faculdade de Engenharia da Universidade do Porto) USTL (Underwater Systems and Technology Laboratory) provides planning and control for teams of unmanned vehicles, making use of OpenStreetMaps. Large part of the interface is occupied by the 2D map while the vehicles are presented in a drop-down hierarchical tree structure. Considering that nowadays the user interfaces rely on appealing visual effects, Neptus contrasts this trend by having a simple yet pragmatic interface.

2.3 OpenCPN (figure 2.3)

This open source framework is a mission planner with integrated nautical charts. It should not be considered an all-in-one solution for data logging but as an auxiliary navigation software. It is used in by marine personnel for mapping and planning. These features are meant to be integrated into the software described by this thesis, making OpenCPN a source of inspiration for the desired functionality.

2.4 Qt (figures 2.4 2.5)

This framework was used to develop the current interfaces used by LSA. It is a framework built to create user interfaces for a variety of programs with a

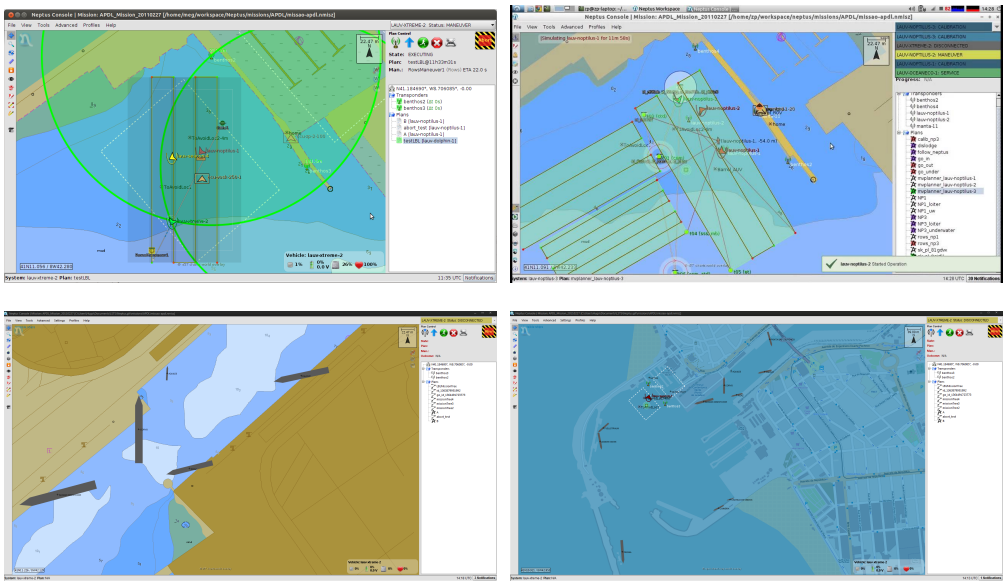


Figure 2.2: Neptus User Interface.

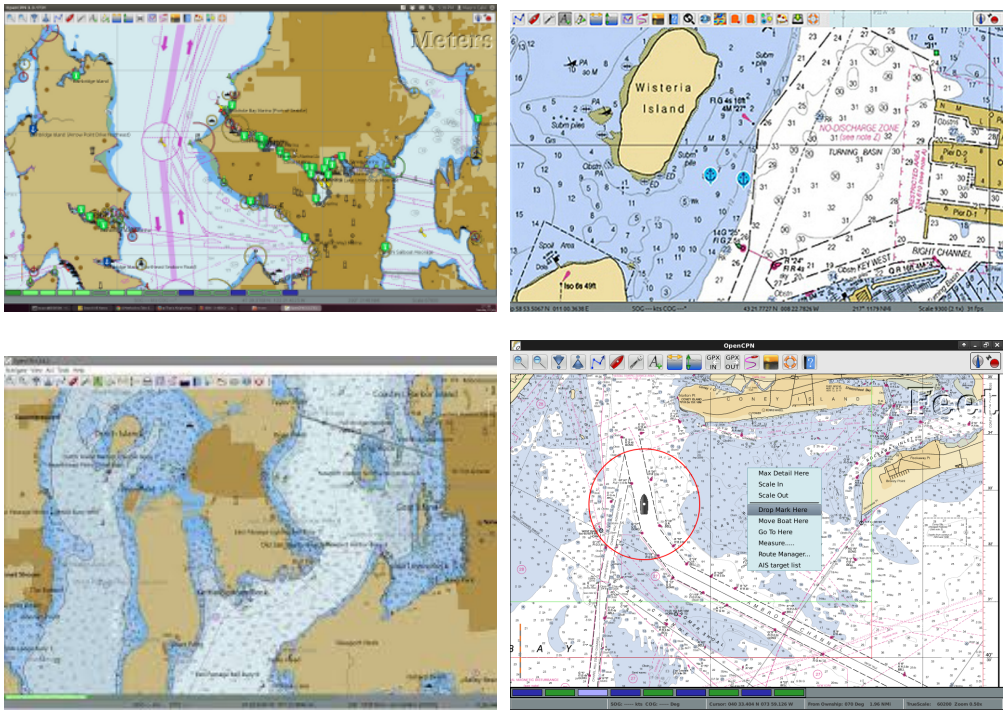
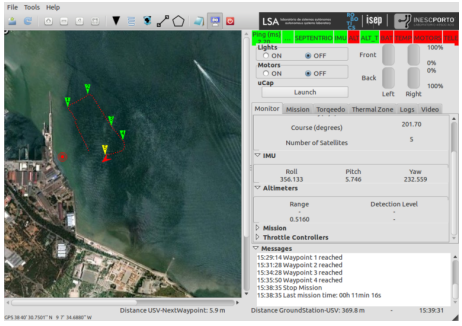
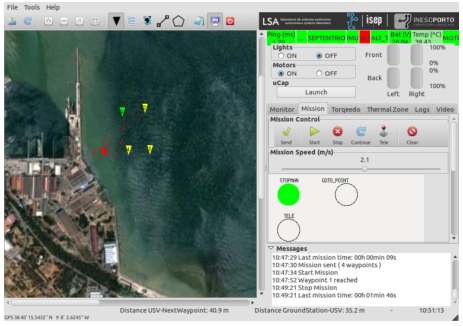


Figure 2.3: OpenCPN plotting and planning software.

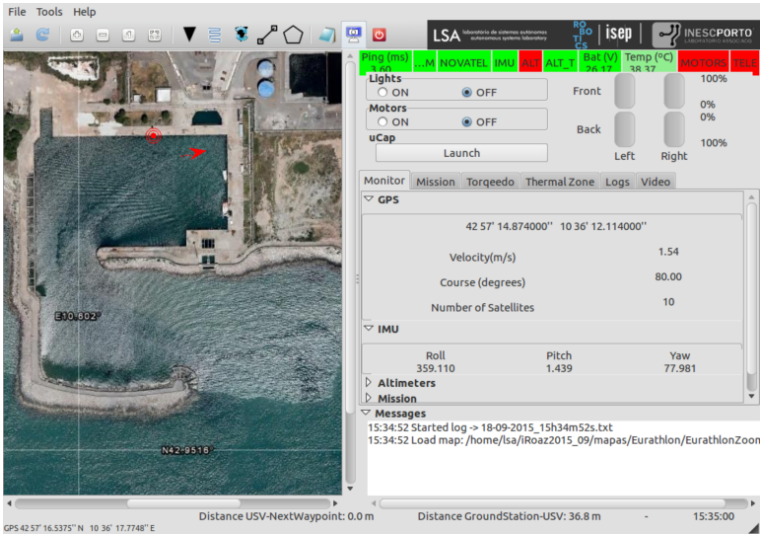
simple, well documented API (Application Programming Interface). These programs were developed with the aerial robots and ROAZ II autonomous boat in mind and provide full sensorial management, mission logging and planning. Given the nonexistent 3D graphical capabilities of this framework, it is unsuitable for a more advanced program like the one described by this thesis. However, it provides insight into the desired capabilities of the work in question. A focus on the map can be noted in the ROAZ II user interface since this robot generally has a support role in the mission it integrates, which makes its location in the world a very important status to be acknowledged. In contrast, if we take a look at the aerial robots user interface we notice a split focus between its location in the world and its cameras. Since these robots generally take a reconnaissance role in the missions what is "seen" by these vehicles is very important.



(a)

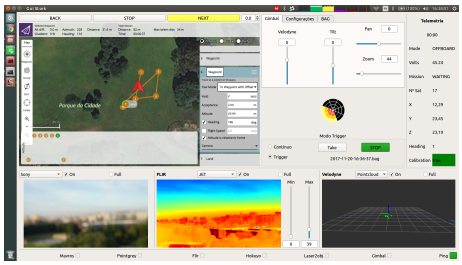


(b)

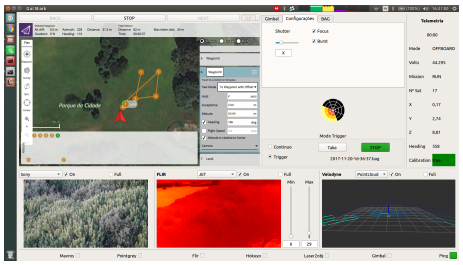


(c)

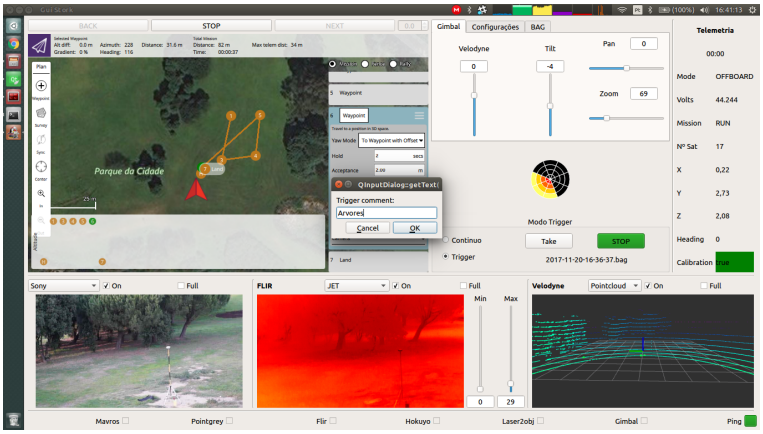
Figure 2.4: ROAZ II user interface



(a)



(b)



(c)

Figure 2.5: Aerial vehicles user interface

2.5 Autonomous control of unmanned ship with Unity3D

Yang et al [10] integrates Unity3D in an artificial intelligence scenario, making use of the Unity3D Machine Learning Agents toolkit, provided by Unity Technologies [11]. The scene environment (figure 2.6) is constructed with the acquirement of a height map where each pixel contains the height information of the area (figure 2.7). This map is laid on a terrain `gameObject` in Unity that takes into account the height map pixel values and terraforms the terrain accordingly. Further embellishment is included in the scene, such as waterways, flora and miscellaneous meshes like bridges and boats modelled in 3ds Max. The developed work acts as a simulator for wind speed, wind direction whose forces exert on the ship.

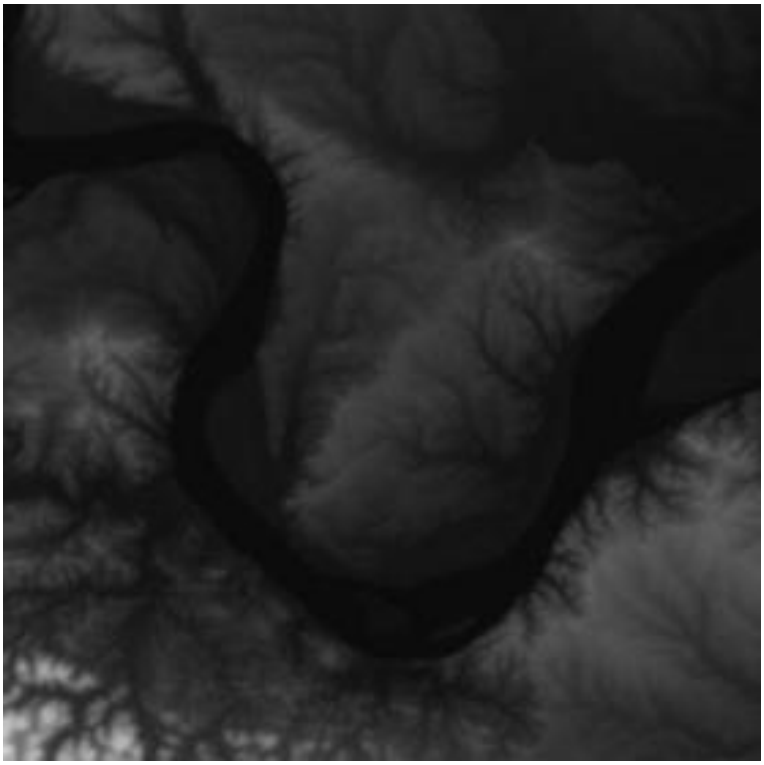


Figure 2.6: Heightmap used to generate the terrain.

Figure 2.7: Terraformed terrain with aditional assets.

2.6 Open Source Simulator for Unmanned Vehicles with ROS and Unity3D

The Institute of Science and Technology developed an open source simulator for UUV (Unmanned Underwater Vehicles) using ROS and Unity3D designated by URSim (Unity ROS Simulator) [12]. This program was developed to study and test complex systems while maintaining costs and environmental disruption to a minimum. URSim is capable of simulating feedback control systems, underwater vision, mission planning and calculating collision kinematics. UUV's (Unmanned Underwater Vehicles) cameras are simulated using Unity scene cameras and its frames are published as compressed image topics. Sensors like IMU and pressure sensors are modelled using C# scripts attached to the game objects. To these components is added Gaussian noise to better simulate real world data. An example scene is represented by figure 2.8

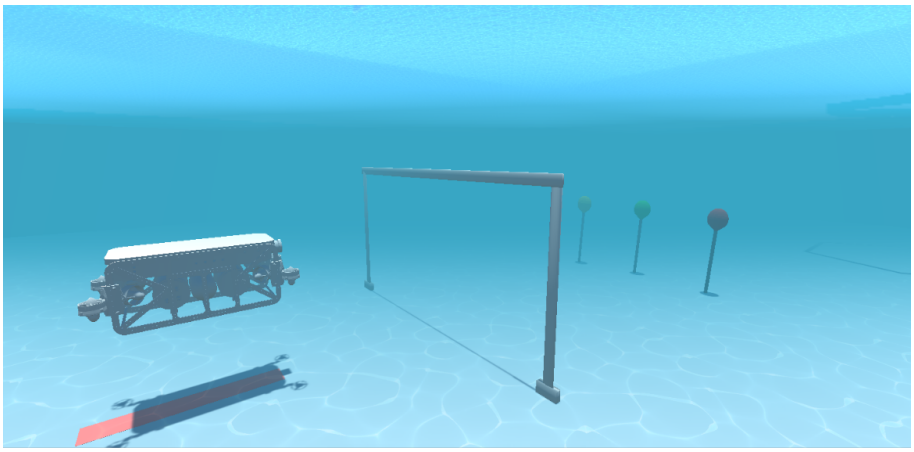


Figure 2.8: URSim example scene

2.7 VAMOS virtual reality human machine interface

Underwater mining European project VAMOS relied on a virtual reality human machine interface for representing real world mapping and issuing commands to the vehicles. The main feature of this HMI is the real time modelling of the underwater mine, using SDF (Signed Distance Function) voxel maps for building a coloured representation, taking in data from various perception sensors like multibeam, 3D imaging sonar and structured light scanners and fusing them into the final result. The interface is constructed to provide mission awareness to the user, showing information about the position and orientation of the vehicles and other support assets in the world. This is important for successfully and safely operate this machinery from a remote environment [13].

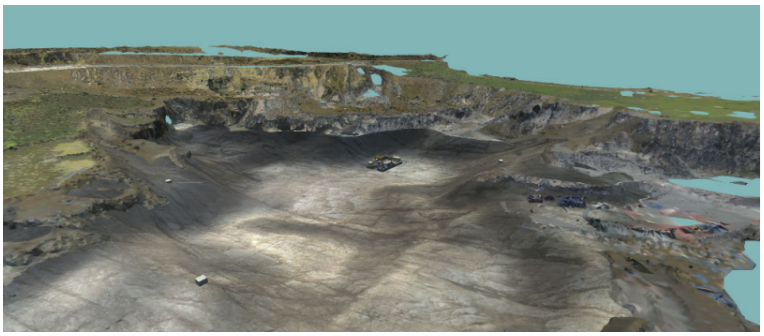


Figure 2.9: 3D mapping of Lee Moor underwater mine

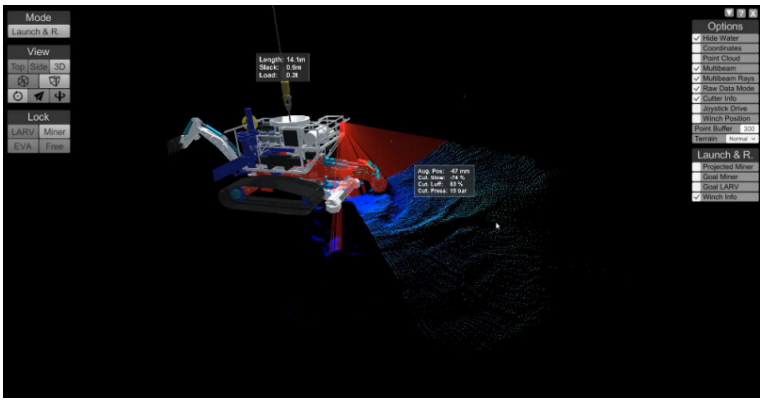


Figure 2.10: Information overlay

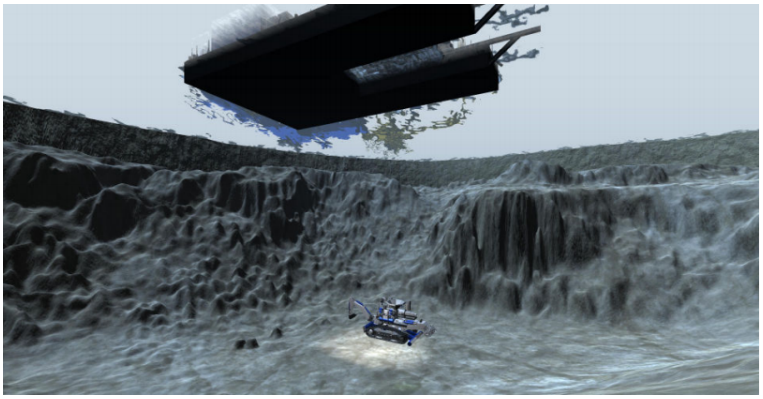


Figure 2.11: 3D environment setup

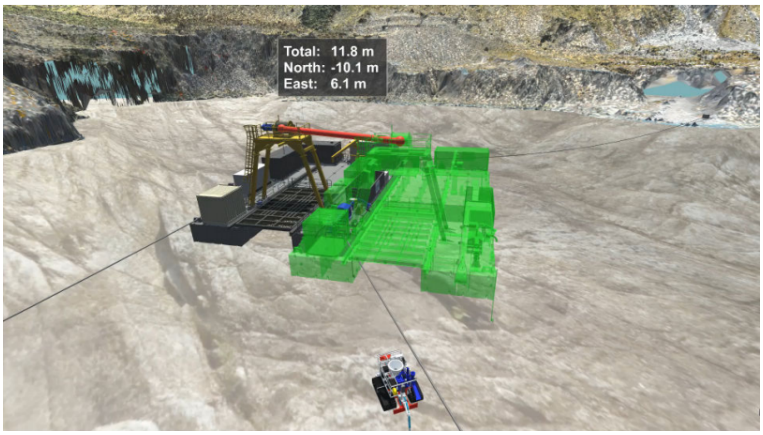


Figure 2.12: Change of asset position highlighting

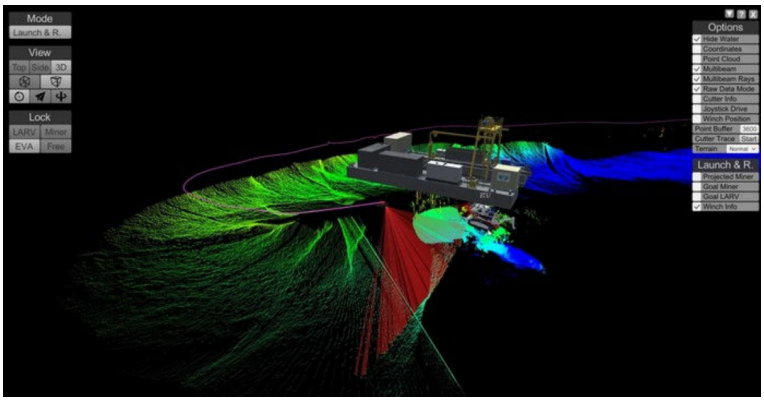


Figure 2.13: Real time multibeam 3D scanning

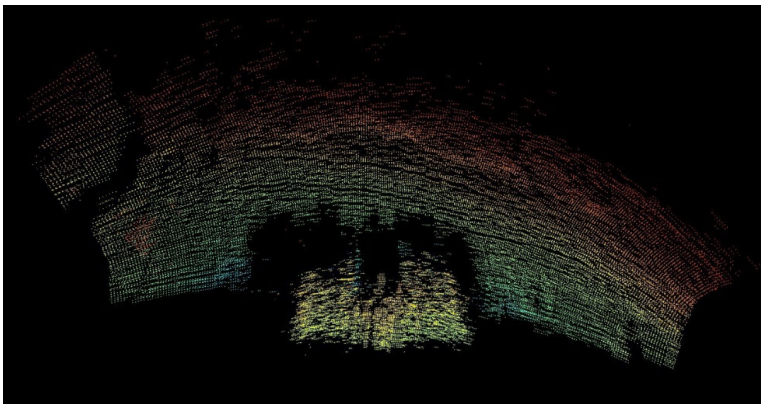


Figure 2.14: Multibeam sonar modeling. Representation done in RViz

2.8 Visual representations on a ROS system

Two popular way of representing data in a ROS system is the use of RViz [14] (ROS Visualization) and `rqt` [15]. These two programs are packaged in every ROS distribution to aid developers in visualizing the data structures and data flow in a ROS system. Like every other ROS software they are both open-source and frequently maintained, meaning that customizations to the source code can be done in order to expand their functionality.

2.8.1 RViz

RViz makes use of the Ogre 3D Graphics Engine for rendering the onscreen data. It is used for rendering basic ROS data-types like point clouds, laser scans,

marker arrays, image streams, etc. Figures 2.14, 2.15[1] and 2.16[2] represent examples of this software in use.

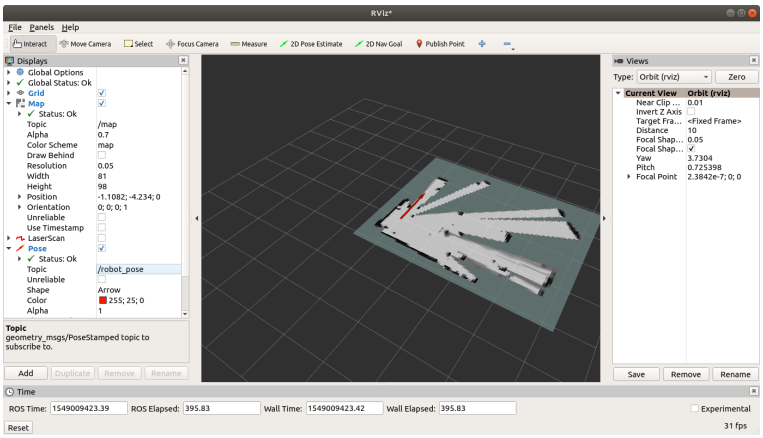


Figure 2.15: RViz representation of a mapping operation.

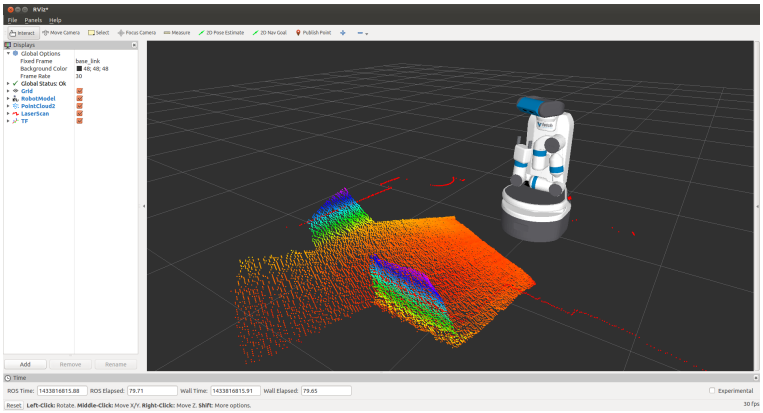


Figure 2.16: Point cloud real time rendering from a LiDAR sensor.

2.8.2 `rqt`

Rqt are a set of ROS tools for plotting data and showing information flow between topics. It employs various GUI tools using a plugin format that can be added to a Qt window. It is generally used for data analysis but one can also develop its own application for expanding the software functionality. Figures 2.17[3] and 2.18[4] show `rqt` usage scenarios.

[1] https://ardupilot.org/dev/_images/ros-rviz.png
[2] https://docs.fetchrobotics.com/_images/rviz.png
[3] http://wiki.ros.org/rqt?action=AttachFiledo=gettarget=ros_gui.png
[4] https://roboticsbackend.com/wp-content/uploads/2019/07/rqt_graph_turtlesim_2_turtles.png

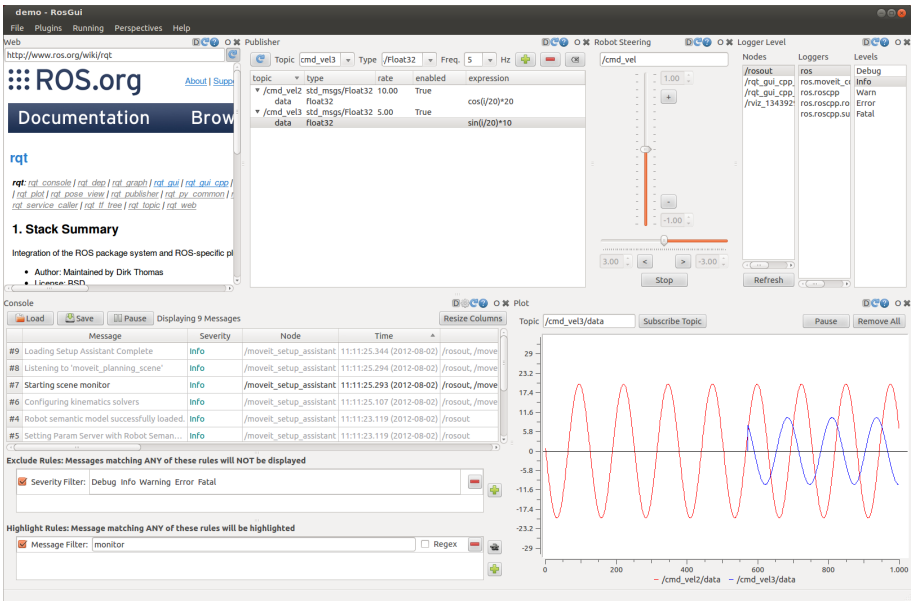


Figure 2.17: `rqt` interface showing multiple plugins in a single window.

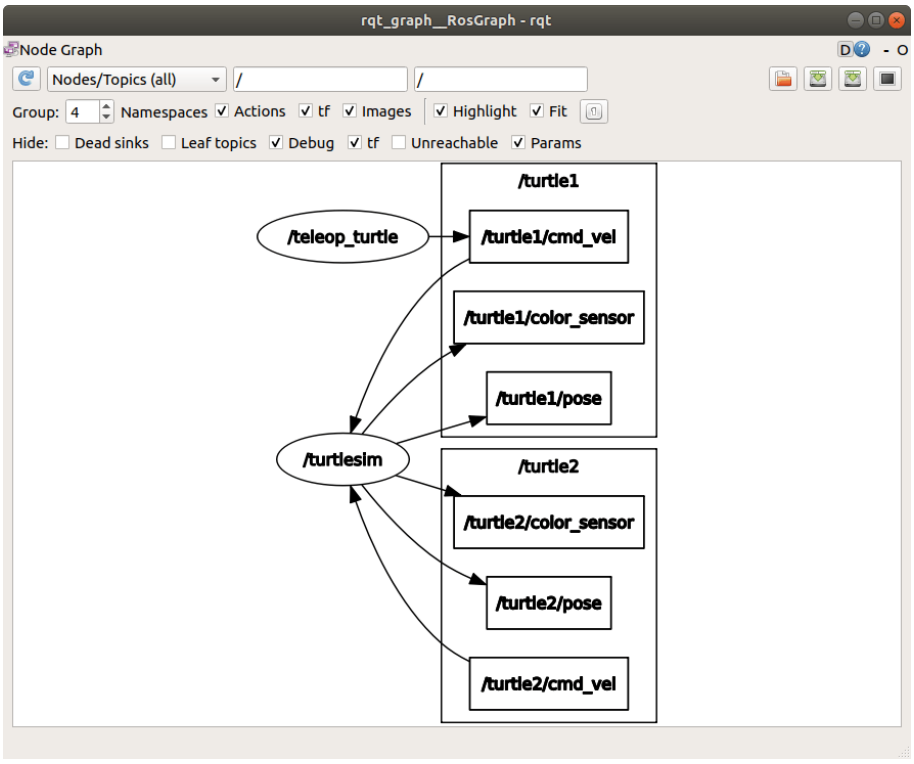


Figure 2.18: `rqt_graph` is a plugin for showing the transmission of topics between ROS nodes.

2.9 Game Engines

This section covers the most popular game engines on the market nowadays, explaining its capabilities and reason to choose them for constructing the interface. Given the vast amount of game engines available on market it is impossible to describe and analyse them all so only the best candidates for this work are mentioned.

2.9.1 Unity

Developed by Unity Technologies, this engine provides powerful 3D capabilities, rigid body calculations, an asset store for ease of development and strong community as backup. It comes with four payment plans, Personal, Plus, Pro and Enterprise, with more advanced features like engine source code access, special development toolkits and source control solutions reserved for the upper tiers. Another thing to note is that the free tier is available only if the revenue of software made with Unity is less than one hundred thousand American dollars a year. C# is used as a scripting language, where each script shapes the behaviour of the game objects associated with it. Games like *Subnautica*, *Hollow Knight*, *Hearthstone*, *Kerbal Space Program* and *Cuphead* are some examples of games developed with Unity. Recent developments of Unity's HDRP (High Definition Render Pipeline) allowed the development of applications with greater graphical fidelity, rivaling its main competitor Unreal Engine 4. It remains to this day as a very popular game engine in the indie game development community and a lot of its popularity stems from its ease of use, asset store and multi platform compiling [16].

2.9.2 Unreal Engine

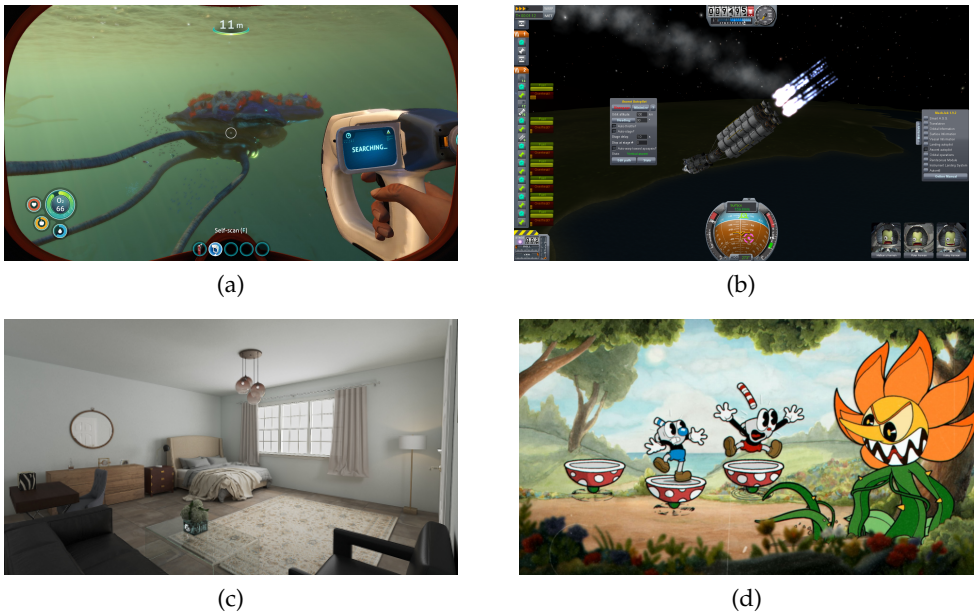
Developed by Epic Games, it is considered the main rival of Unity. It also provides outstanding 3D capabilities and physics calculations although with a steeper learning curve than Unity. As of the moment its source code is open and available on GitHub. However, Epic Games employs a royalty system for every copy of software sold that makes use of Unreal. Behaviour of the in-game objects (called Actors) is accomplished with C++ classes or a visual scripting language called Blueprints. It originated some of the most popular video games like *Fortnite*, *Player Unknown's Battlegrounds* and *Kingdom Hearts III*, among many others. Not only it is very popular among the video game community but its accurate physics and rendering capabilities draws the attention of engineering and architecture companies [17].

2.9.3 Cryengine

Created by the german company Crytek, this engine provided much of the advanced rendering capabilities also present in Unreal and Unity. It makes use of C++, C# and Lua programming languages. It presents many similarities with its counterparts providing physics simulations and advanced graphical capabilities. Some games developed with this engine are the *Crysis* series, *Kingdom Come: Deliverance* and the *Prey* series [18].

2.9.4 Godot

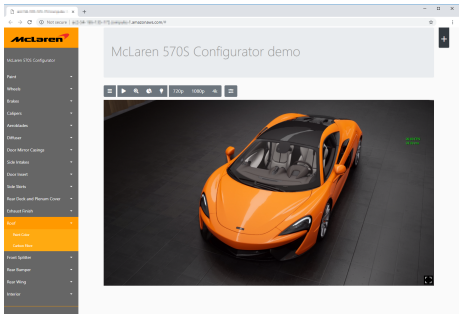
Being totally free of charge makes this engine great for learning and among the indie community. Being open-source also makes it easier to expand its capabilities by altering its source code making it expandable. Despite these advantages, limited support for general programming languages like C++ and C#, and instead relying on a proprietary scripting language make this game engine a poor choice for the accomplishing the desired goals. Moreover, Godot is not as mature as older game engines like Unity or Unreal Engine since it has been in development for a shorter amount of time which reflects on the amount of available features and limited community to back it up [19]. Because of this, Godot is more intended for education or indie development at the moment.



(a) (b)

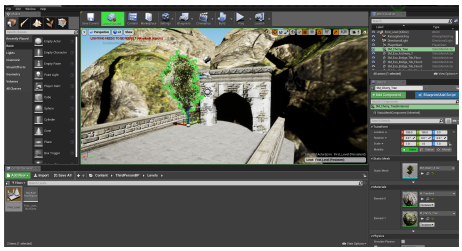
(c) (d)

Figure 2.19: Unity showcase. (a) Subnautica. (b) Kerbal Space Program. (c) Photorealistic environment in Unity. (d) Cuphead.



(a)

(b)

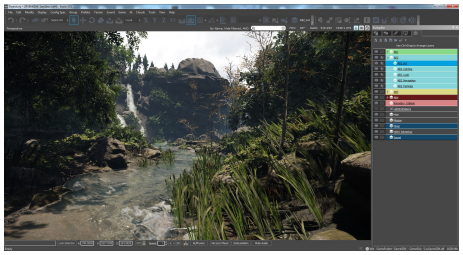


(c)

(d)

Figure 2.20: Unreal Engine 4 showcase. (a) Mclaren car configurator. (b) Photo-realistic environment. (c) Unreal Engine Editor. (d) Fortnite.

(a)

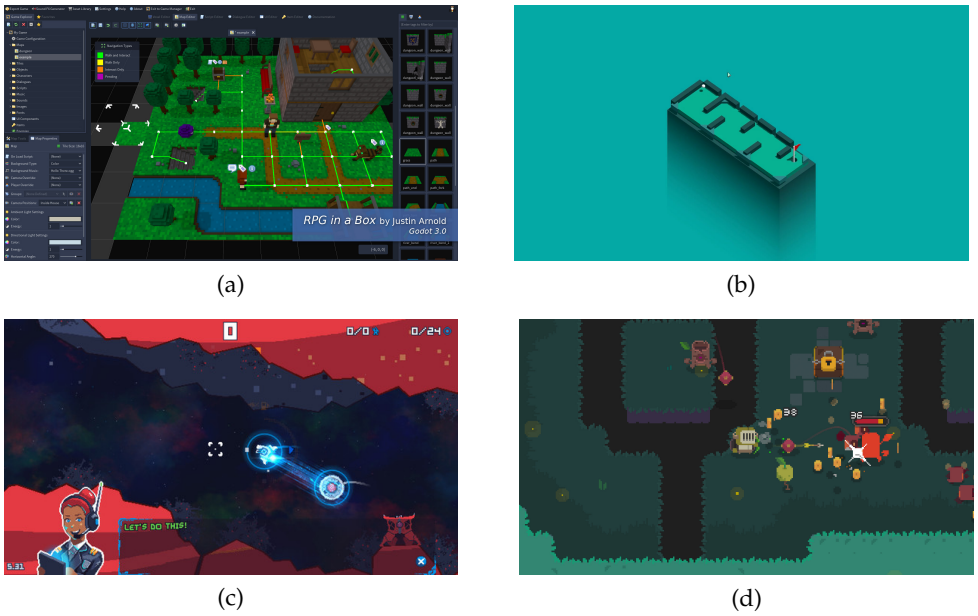


(b)

(c)

(d)

Figure 2.21: Cryengine showcase.(a) Crysis 3. (b) Cryengine Editor. (c) Prey. (d) Ryze: Son of Rome.



(a) (b)

(c) (d)

Figure 2.22: Godot showcase. (a) Godot Editor. (b) Hyperputt. (c) Gravity Ace. (d) Helms of Fury.

Chapter 3

Problem formulation

Abstracting the information that a single robot generates is done by converting the large amount of calculations that its sensors provide and translating it into a context that can be easily interpreted by a human. Considering that humans tend to interpret visual representations much more rapidly than numbers, the data from these sensors needs to be abstracted into visual elements.

LSA's autonomous vehicles can be summarized as complex computer systems made up of a plethora of different sensors connected by a managed communication layer. The challenge of integrating a new system into this mainly manifests itself as a communication problem. Being ROS the communication layer used by the vehicles means that any external system that does not understand this format is left out of the loop. Solving this requires the translation between two communication layers.

The complexity and insurmountable amount of the data flowing in each one of these vehicles provides the operator with a significant challenge in interpreting and acting upon this data. For this, an HMI needs to be developed, providing the tools for overcoming any control and surveillance challenges, and giving the user the awareness needed for safely operating these machines.

Cues on how to develop such an HMI can be taken from industries that have shown the experience in visually representing data like the video games industry, even though their data may not be real, it can be made to tailor real world data. Mixing these two worlds and creating an experience that is visually appealing to the user but that also has real world value associated with it, is an enticing approach to the problem.

A software like this could easily escalate in complexity given the amount of features and abstractions that are required. In an environment with multiple

robots in scene, giving the ability for the user to keep track of every single one of them is not only an computer engineering challenge but also a graphic design one. A single computer screen has limited space, which requires clever use and reuse if we want to display the current operation status.

At the moment, graphical interfaces used by LSA are fragmented into multiple applications. The proposition is to create a single multi purpose centralized graphical interface for all vehicles and adaptable to any mission.

Designing a general purpose application to integrate a network of robots while conforming with their existing communication infrastructures and specific set of commands is a challenge in itself.

Robot autonomy is a never ending subject tackled by researchers from around the world [20]. How can we make a machine that is self sufficient, makes its own decisions safely and still accomplishes the objectives proposed to it without mistakes? It is a never ending problem that robotics engineers are constantly trying to solve, by making these machines more self aware and developing new techniques to give the robots the necessary tools for accomplishing their missions without human intervention. What is a sci-fi concept may someday become the norm especially considering the modern advances of machine learning and AI (Artificial Intelligence). Nowadays robots are still not as intelligent as we would like to which means that humans still play a significant role in the course of autonomous missions. Robots still need a fair amount of human action in order to perform as expected which in return means that we are constantly trying to improve on human-robot interactions. Studying how SLAM (Simultaneous Localization and Mapping) techniques can be implemented play a significant role in giving the robot the necessary tools for it to build its world and be made self aware in an unknown environment. By fusing multiple perception systems the robot can build its own virtual representation of the real world and navigate it in a safe and efficient manner.

The environments at which these vehicles are subjected are generally hostile towards humans. Teleoperation of robots is essential to supplement their mainly autonomous operations in these environments. The problem with remotely operating a machine is the sensory deprivation that the user is subjected to. Humans act upon what their five senses percept, meaning that a good HMI design should be able to translate what the robot senses to an human understandable perception. The use of joysticks, audiovisual feedback and at a more advanced level, augmented reality helps bridge the gap between human understanding and artificial sensor systems that let the user immerse itself into what the vehicle is experiencing. Teleoperation also introduces new challenges like the need of low latency connections to prevent lag upon issuing commands. Moreover, the user needs to receive an acknowledge from the vehicle each time a

new command is issued in order to establish a trustworthy connection between the operator and the machine in question. This feedback is key in informing the user that its commands are being received successfully.

Several types of user interface can be built depending on the target audience that uses it. As robotics becomes more accessible to the general public, interfaces with an high level of abstraction need to be developed so that a non trained operator can successfully control a complex robotics system. For an example of this we can examine the market of semi-autonomous aerial vehicles. The popular aerial drone company DJI helped popularize the the drone market by providing accessibility to these hard to control vehicles. By automating many of the systems of a drone, consumers without any technical experience of background in building or controlling aerial vehicles are now able to do it, simply by using an interface tailored for easily controlling these types of vehicles. Giving the user various "safety nets" to prevent destructive behaviour, builds the user's confidence and encourage the use of their products.

Chapter 4

Theoretical concepts

This chapter focuses on explaining the various techniques and mathematical models used throughout the development of this software. The mathematical techniques that are used to perform 3D space transformations or on how a camera works. Coordinate systems conversions between local and world space coordinate systems is also approached. Insight on how the Unity game engine functions and lastly, the communication infrastructure is explained with all the serialization that the data suffers through from the emitting to the receiving end.

4.1 Concepts of a 3D environment

This section is dedicated to presenting the concepts behind three dimensional computer graphics and relating them to real world concepts. Creating convincing virtual worlds is reliant on a wide variety of mathematical algorithms and real world concepts. Even though most of the explained calculations are hidden by a layer of abstraction in Unity, it is important to understand how this game engine and consequently how three dimensional rendering functions.

4.1.1 Coordinate system

Unity's coordinate system follows the left-hand rule which differs from the right-hand rule coordinate system generally used in an engineering scenario. This stems from the fact that when rendering 3D environments, the x and y axis represent the 2D screen coordinates and the z axis the depth.

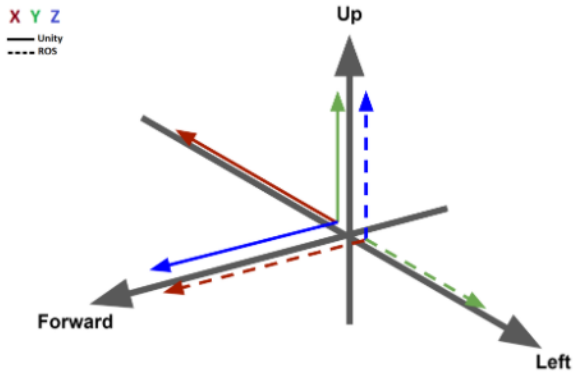


Figure 4.1: Comparing Unity's coordinate system with ROS' [3]

When introducing data from ROS into the application a transformation between the two coordinate systems is needed.

	Unity	ROS	Transformation from ROS to Unity
x	right	forward	$\text{Unity}_x = -ROS_y$
y	up	left	$\text{Unity}_y = ROS_z$
z	forward	up	$\text{Unity}_z = ROS_x$

Table 4.1: Conversion table for ROS to Unity coordinate systems

4.1.2 World coordinates to local coordinates

Converting GPS (Global Positioning System) coordinates from a Earth scale to a local one is done in two steps [21]:

- Convert from geodetic system to ECEF (Earth Centered Earth Fixed)
- Convert from ECEF to a local coordinate system like NED (North East Down)

Geodetic to ECEF

In the first step we need to convert the geodetic coordinates to an intermediate coordinate system like ECEF. Considering P_g a GPS coordinate:

$$P_g = \begin{bmatrix} \lambda \\ \varphi \\ h \end{bmatrix} \tag{4.1}$$

λ: Longitude φ: Latitude h: Altitude

$$N_E = \frac{R_T}{\sqrt{1 - e^2 sin^2 \varphi}} \tag{4.2}$$

$$P_e = \begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} = \begin{bmatrix} (N_E + h) cos\varphi cos\lambda \\ (N_E + h) cos\varphi sin\lambda \\ [N_E(1 - e^2) + h] sin\varphi \end{bmatrix} \tag{4.3}$$

x_e: ECEF x coordinate y_e: ECEF y coordinate z_e: ECEF z coordinate

N_E: Prime vertical radius e: First eccentricity R_T: Earth's radius

ECEF to NED

After obtaining the ECEF coordinates we can convert them to a local reference frame like NED. $P_{e,ref}$ represents the origin of the NED frame in ECEF coordinates and $R_{n/e}$ is the rotation matrix from the ECEF frame to the NED frame.

$$R_{NED/ECEF} = \begin{bmatrix} -sin\varphi_{ref} cos\lambda_{ref} & -sin\varphi_{ref} sin\lambda_{ref} & cos\varphi_{ref} \\ -sin\lambda_{ref} & cos\lambda_{ref} & 0 \\ -cos\varphi_{ref} cos\lambda_{ref} & -cos\varphi_{ref} sin\lambda_{ref} & -sin\varphi_{ref} \end{bmatrix} \tag{4.4}$$

λ_{ref} e φ_{ref} represent the geodetic latitude and longitude relative to $P_{e,ref}$.

$$P_n = R_{n/e}(P_e - P_{e,ref}) \tag{4.5}$$

$$P_n = \begin{bmatrix} north \\ east \\ down \end{bmatrix} \tag{4.6}$$

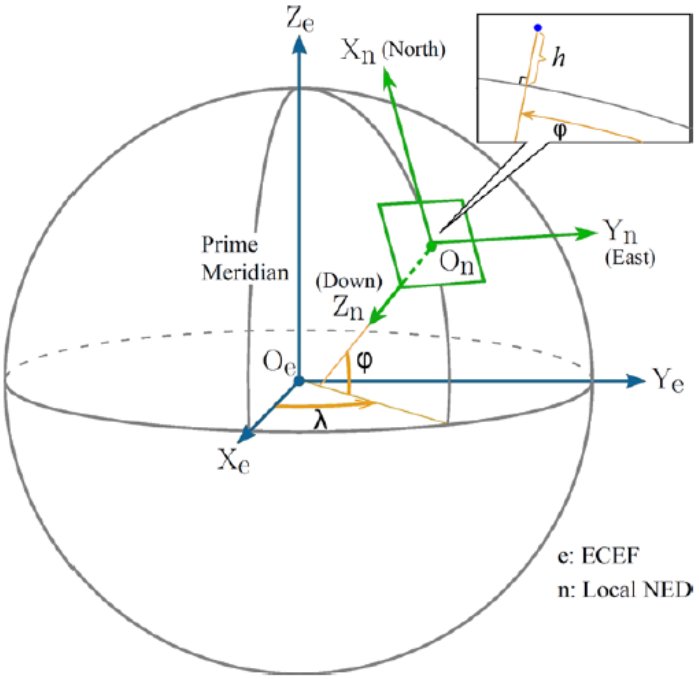


Figure 4.2: Geodetic, ECEF and local NED coordinate systems.

P_n is the column vector representing the three coordinates in the NED local coordinate system.

The inverse can be done to convert from the local NED coordinate system to ECEF and then to geodetic using an iterative algorithm like [22].

$$P_e = R_{n/e}^T P_n + P_{e,ref} \tag{4.7}$$

NED to ENU

NED can be converted to ENU (East North Up) which is the same referential axis that ROS uses. From here we can convert ENU/ROS to Unity referential using table 4.1. The following rotation matrix, calculated from three successive rotations in each axis (ZYX), transforms a NED coordinate system into a ENU coordinate system:

$$R_{NED/ENU} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi & -sin\phi \\ 0 & sin\phi & cos\phi \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{4.8}$$

$$\theta = -\frac{\pi}{2} \; \phi = \pi:$$

Geodetic Datum

The conversion from geodetic coordinates to a local frame is not perfect. The error associated with this conversion is because the Earth is not a perfect ellipsoid which means that the selected model for representing its dimensions is only an approximation of its shape. A generally used datum is the WGS84 (World Geodetic System 1984) model. This features a good approximation of the Earth's shape on a global scale. However, it may not be the best datum for carrying out the conversion, depending on the worksite's location. When performing the calculations mentioned above, it is important to select the datum that would result in the least amount of projection error. This can be done by picking the datum that better approximates the Earth's shape to the worksite's location on the world.

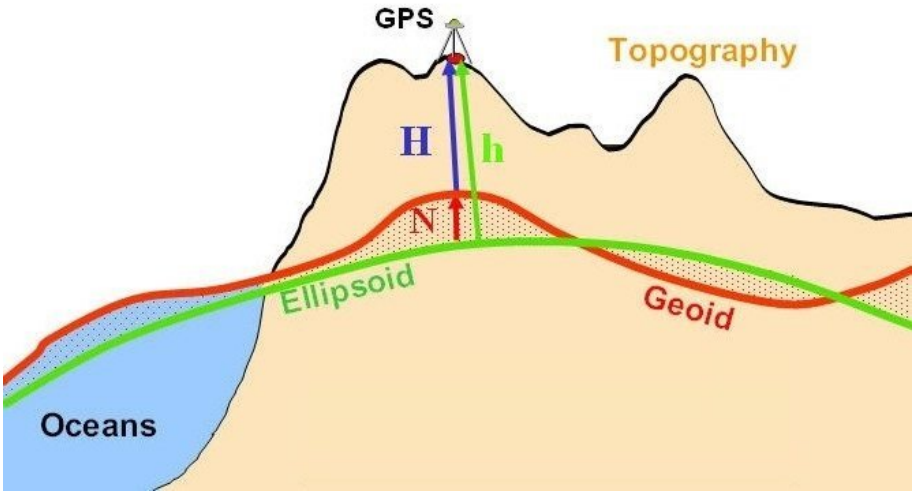


Figure 4.3: Ellipsoid approximation to the Earth's topology.

4.2 Three dimensional space transformations

An object's position and attitude in a three dimensional environment is constrained to six degrees of freedom.

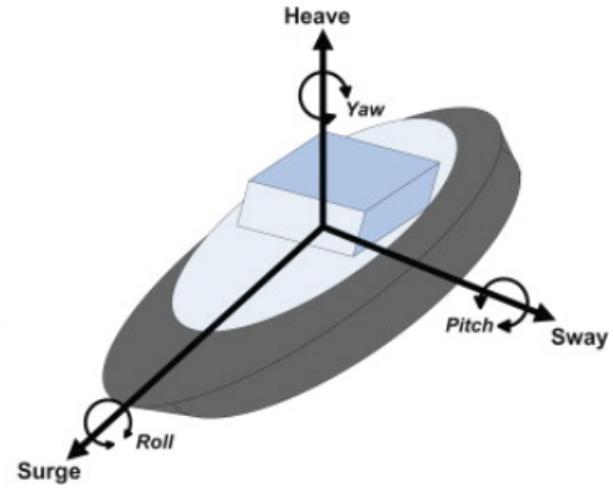


Figure 4.4: 3D object six degrees of freedom.

4.2.1 Translation

In a 3D space an object translation is done by multiplying the object's coordinates matrix with a translation matrix.

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.9}$$

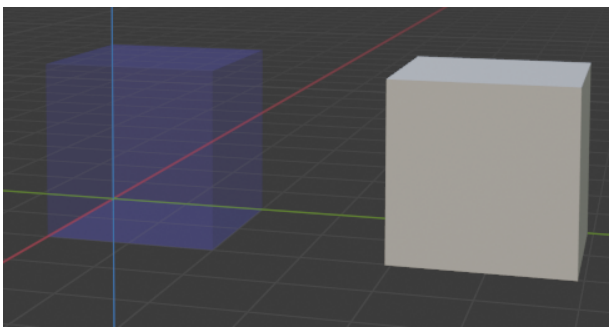


Figure 4.5: 3D translation.

4.2.2 Rotation

In a 3D space we can rotate in each axis by multiplying the object's coordinates with the corresponding rotation matrix.

Rotation matrix for rotating around the x axis:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.10}$$

Rotation matrix for rotating around the y axis:

$$R_y = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

Rotation matrix for rotating around the z axis:

$$R_z = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.12}$$

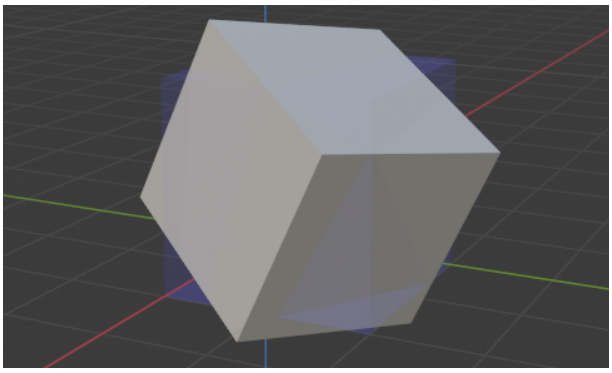


Figure 4.6: 3D rotation.

4.2.3 Scaling

Scaling an object in 3D space is done by multiplying the scaling matrix with the object's coordinates.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.13}$$

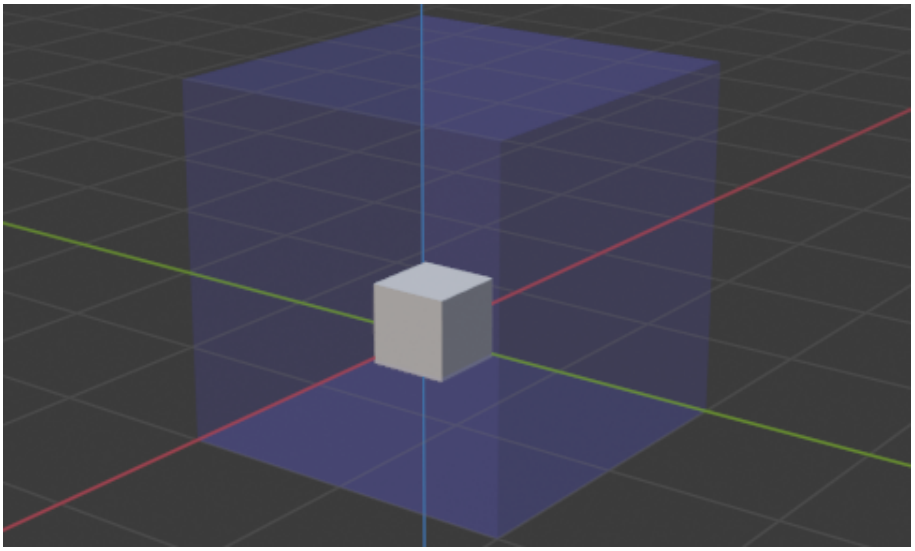


Figure 4.7: 3D scaling.

4.2.4 Shearing

Shearing is a transformation done for changing the object's shape along an axis.

It is done by multiplying the object's coordinates with a shearing matrix:

Shearing in the x axis:

$$Sh_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ s_y & 1 & 0 & 0 \\ s_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.14}$$

Shearing in the y axis:

$$Sh_y = \begin{bmatrix} 1 & s_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & s_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.15}$$

Shearing in the z axis:

$$Sh_z = \begin{bmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.16}$$

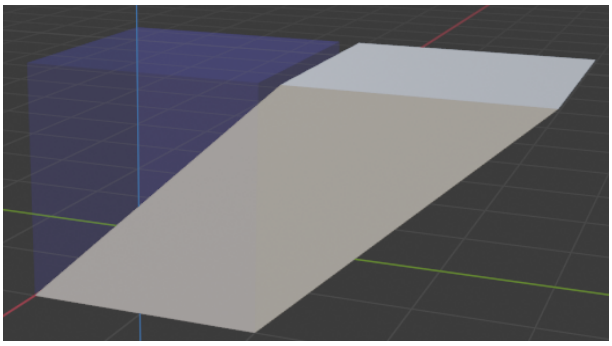


Figure 4.8: 3D shearing.

4.3 Camera

Understanding how a camera works in the real world is essential for emulating their properties in a virtual environment. Notions on how a camera functions allows the design of realistic virtual world interactions and more importantly model what and how the user sees a recreated virtual environment. With these notions, complex scene compositions can be developed and improve the user experience when interacting with the elements that compose the 3D software.

Pinhole model

A camera can be represented by the simplest camera model, the pinhole camera model [23]. By placing a barrier with a small aperture between the 3D scene and the photographic sensor, only a small amount of light rays from passes through the barrier, projecting an image onto the sensor. The size of the aperture is important in obtaining a focused image because if too many light rays are projected onto the sensor, the result is a blurry picture and if the aperture is too small, only a few light rays are projected which results in a focused but dark picture.

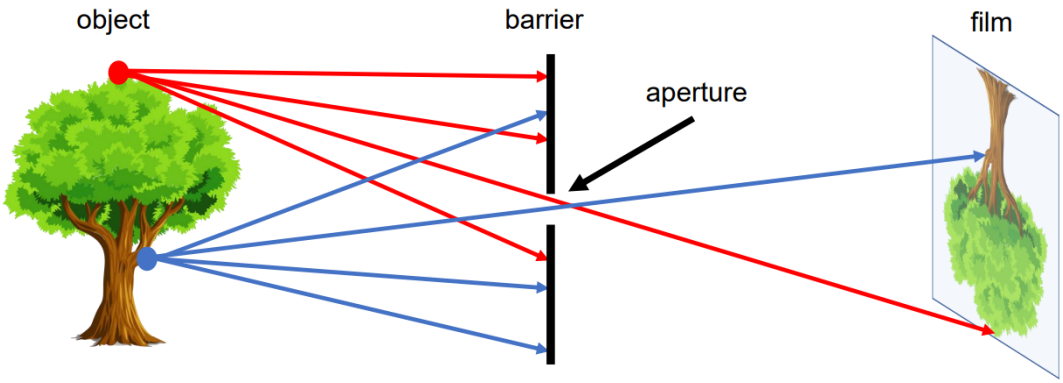


Figure 4.9: Camera pinhole model.

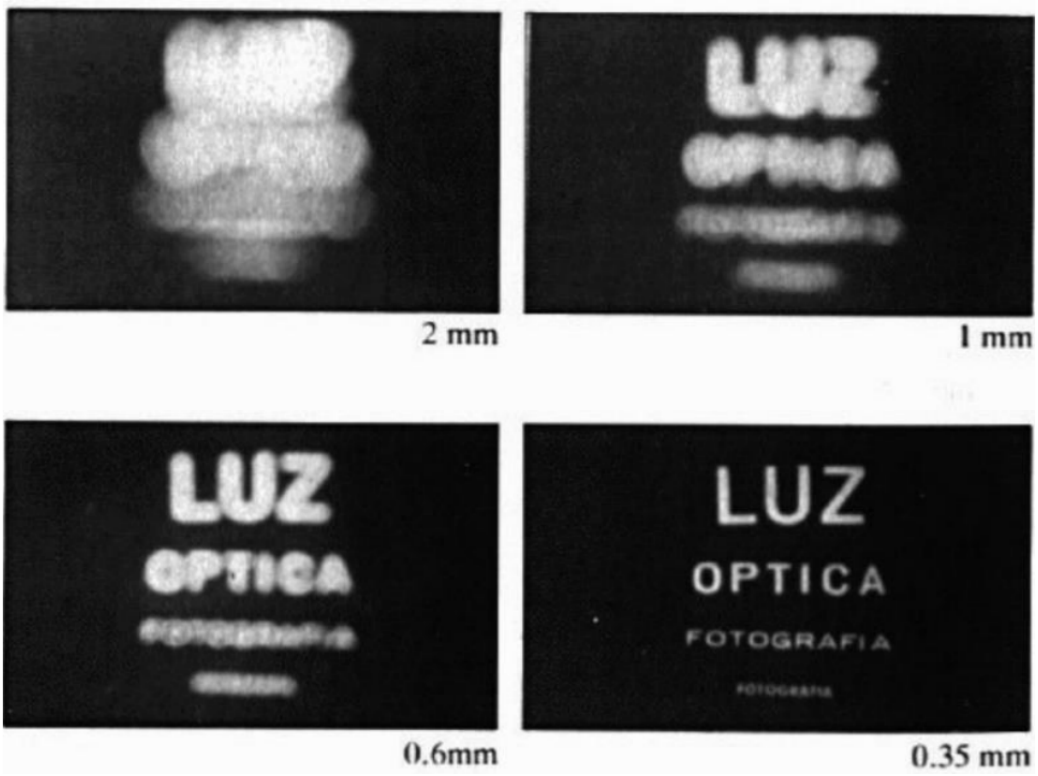


Figure 4.10: Different aperture sizes result in different image acquisition. A small aperture results in a crisp image but a large one results in a bright but blurry picture.

We can lay out the different components that compose this model, constructing a more formal representation of the pinhole model.

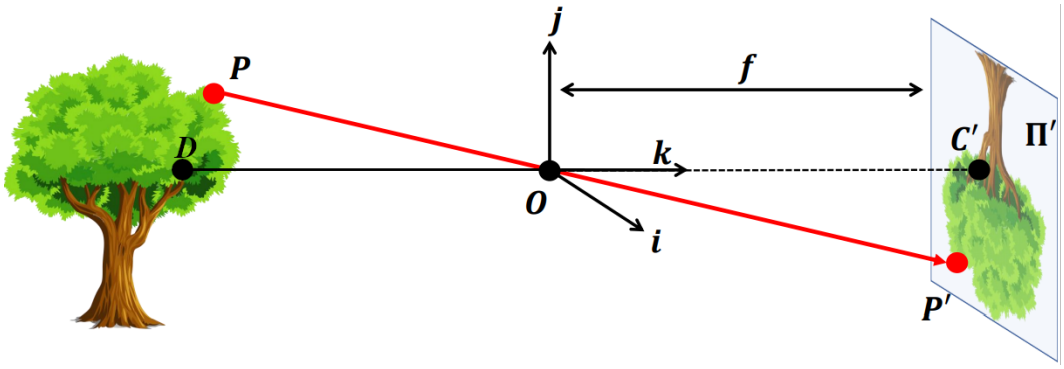


Figure 4.11: Camera pinhole model.

Using the representation in figure 4.12 we can decompose the different components of this model as O for the aperture or center of camera, the distance

between this point and the image plane is the focal length f.

If we consider $P = \begin{bmatrix} x & y \end{bmatrix}^T$ as a random point of an object visible to the camera, the light rays emitted by this point pass through the aperture, projecting onto the plane $\prod'$ resulting in the projected point $P' = \begin{bmatrix} x' & y' \end{bmatrix}^T$. The pinhole itself can be projected onto the plane as point C'. Defining $\begin{bmatrix} i & j & k \end{bmatrix}$ as the camera coordinate system, centered at the origin O, k is the line defined by C' and O referred to as the optical axis. Using this system we can calculate the coordinate at every point of the projected image using the law of similar triangles since the triangle formed by points $P'C'O$ is similar to the triangle formed by points PDO:

$$P' = \begin{bmatrix} x' & y' \end{bmatrix}^T = \begin{bmatrix} f\frac{x}{z} & f\frac{y}{z} \end{bmatrix} \tag{4.17}$$

Lens projection

Mitigating the dilemma between the crispness of an image and its brightness is done by the use of lenses. If we replace the pinhole with a converging lens, light from multiple origins can be converged on a single point called the focus point. This however is only true if the object is at a certain distance interval called the depth of field in which we can refer to the object as in focus of the camera. Objects outside of this interval will appear blurred in the resulting picture.

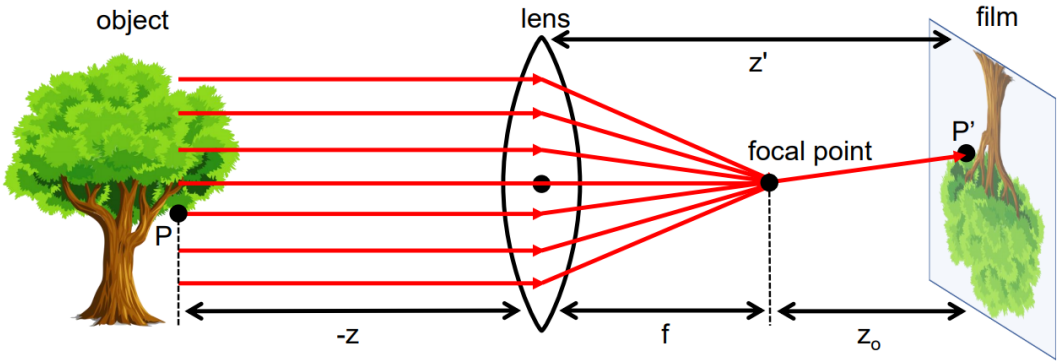


Figure 4.12: Camera setup with a converging lens. Parallel rays of light converge into a single point called the focus point.

Much like the pinhole model we can calculate every point of the projected image as such:

$$P' = \begin{bmatrix} x' & y' \end{bmatrix}^T = \begin{bmatrix} z'\frac{x}{z} & z'\frac{y}{z} \end{bmatrix} \tag{4.18}$$

A result of using a camera setup with a converging lens is image distortion. Lens distortion are mitigated by proper camera calibration.

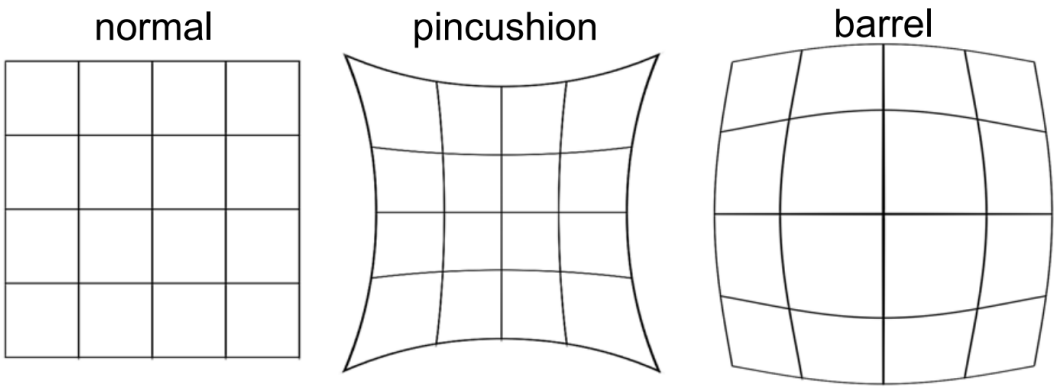


Figure 4.13: Types of distortion caused by lenses.

4.3.1 Cameras in 3D environments

In 3D environments like the ones created in an OpenGL context, the concept of a camera does not exist. It is emulated by manipulating the rendered picture to make it seem like it is being captured by a camera. For example, to give the user the illusion of movement inside of a scene, the entire scene is moved in the opposite direction. A virtual camera also does not suffer from real camera effects like lens flare or depth of field, being these properties added to the image as a post processing effect.

4.3.2 Camera projection

A camera projection matrix is used for mapping the 3D points of the rendered environment onto the 2D screen. The two projection types that are most used are the perspective and the orthographic projection.

Perspective This projection simulates how the human eye sees, making objects that are farther from view seem smaller giving a sense of distance. A good example of this is the existence of vanishing points in images with this projection.

Orthographic This projection is mainly used in CAD software because it does not distort object dimensions. It is used when dimensions and angles need stay unchanged. Objects that are further away retain their shape and size.

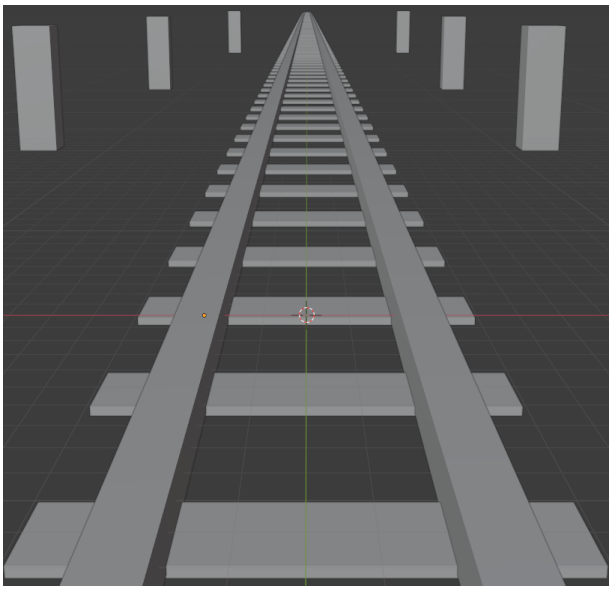


Figure 4.14: Blender scene projected in perspective view.

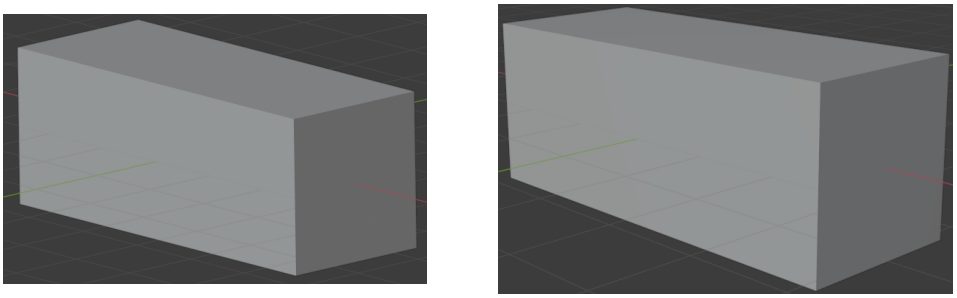


Figure 4.15: Left image shows an orthographic projection of a cube and the right image shows a perspective projection of the same cube.

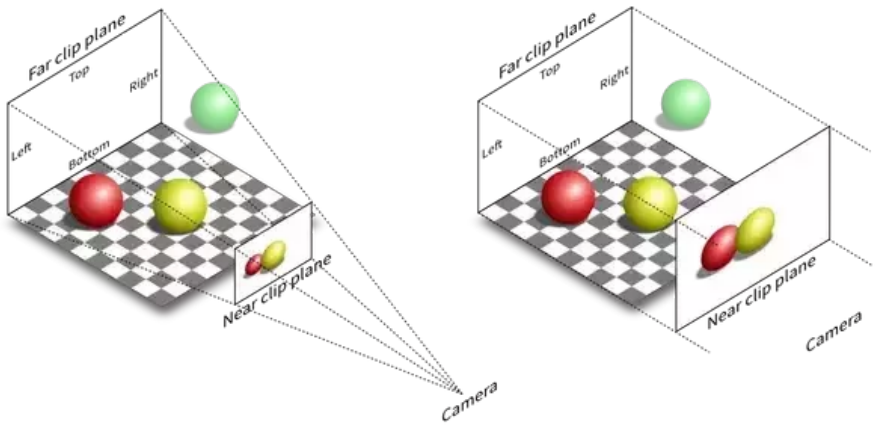


Figure 4.16: Perspective and Orthographic camera projections respectively.

4.4 Unity

A game engine is an IDE (Integrated Development Environment) that provides a set of tools relevant mainly for the creation of video games. Collision system, audio, graphics, artificial intelligence, networking, animation, scripting and physics engine are some of the tools that come bundled in most game engines. This set of features is not only useful for the creation of video games but can be also be used for the development of simulators and product demonstrations as shown by engineering and architectural companies.

The Unity Editor is a development environment where the game is designed, programmed, debugged and compiled into the final product. Every application starts with an empty scene with only a camera and a light added to it. It can be thought as a movie set where a camera captures the action that a light illuminates. A scene can be compared to the various levels that make up a game and every scene is independent of each other and is made up of different components and behaviours that model the flow of the game.

`Game Objects` [] are the fundamental objects in Unity. They can represent assets, terrain, cameras, lights, etc, being their functionality modelled by the components that are added to it. If for example we take an empty `GameObject` and add to it a camera component, we end up with a camera object inside the scene, that can be placed anywhere in the world and capture the action. `GameObjects` can be made inactive during application runtime when they are not needed or even destroyed to remove them from the scene. Changing their status can be used to save system resources or to toggle their visibility and interactions with other objects in scene. Scripts can also be added to these objects making it possible to program behaviour.

The hierarchy panel on the left side of the editor shows the game objects that the scene is composed of. It is arranged in a tree like structure composed of parent and child objects. If we imagine a folder like structure, the parent object represents the main directory and the child objects are the sub-directories inside. The advantage of this layout is not only organizational but also practical since the child objects follow the transformations (translation, rotation, scaling) of the parent object. This facilitates the manipulation of multiple assets in the scene.

Unity's scripting backend is based on the Mono framework [24]. This is an open source implementation of Microsoft's `.NET` framework. Using Mono is what enabled Unity to compile cross platform code instead of only being available in Windows or MacOS. Because of this, it is possible to use `dlls` in a Linux environment. Mono also offers the possibility of loading managed and native plugins. Managed plugins are libraries written in C# that can be loaded directly by Unity for expanded functionality. Native plugins offer even more expandability since they can be written in C/C++ or Objective-C allowing for even

more libraries to be loaded by Unity. Native plugins can take the form of Low Level Rendering plugins [25], giving the developer direct access to low level graphics API's like `OpenGL`, `DirectX` or `Vulkan` for manipulating visual effects directly.

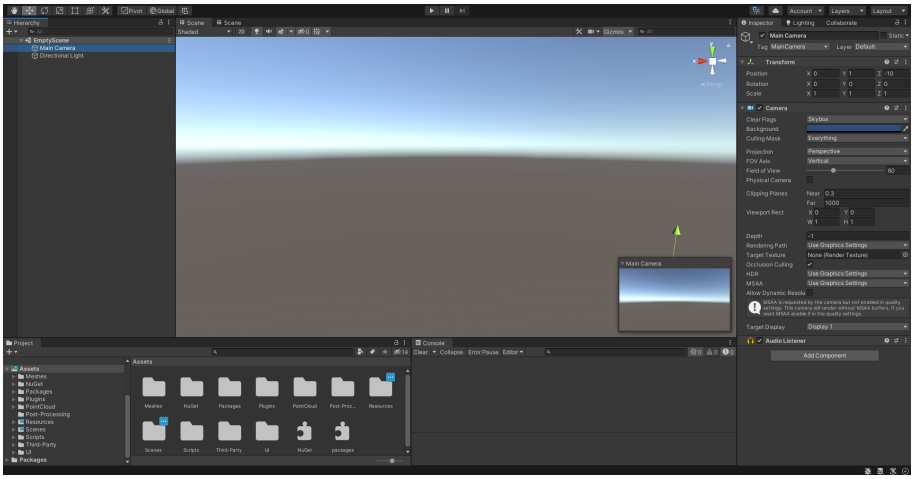


Figure 4.17: Empty scene in Unity Editor (center). Left panel shows the object hierarchy. Right panel shows the inspector panel. In the bottom the project file structure is shown, along with the debug console.

To render a scene a main camera is needed. This camera is responsible for capturing what the user sees on screen and can be configured to mimic real world cameras and simulate properties such as field of view, near and far frustum, sensor size, focal length and lens shift. These properties can be manipulated to customize the rendering of a scene and to introduce different rendering techniques or to simulate real world cameras. Multiple cameras can be created for rendering different parts of the scene and introduce various perspectives of the scene.

Unity's user interface is a toolkit for building an user interface inside of the game. It is composed of various UI elements that bridge user interactions with the application. With these elements the user surveys, controls and updates its knowledge about the action on screen. These elements take the form of buttons, toggles, dropdown menus, sliders, input fields, textures and text. User input is captured by the program and used for a variety of different ends like displaying new or auxiliary information, configuring the environment, adding new assets to the scene. These components provide feedback to user input and allow bidirectional flow of information from the interface to the user. UI elements are drawn on screen in order of their hierarchy which means that an element that is first in the hierarchy is drawn to the screen first than one that is further down

the hierarchy. With this in mind, altering and element's position on the hierarchy can be used for overlaying information on screen. An UI can have different display modes [26]:

Screen space overlay
A canvas rendered in this mode is overlaid on top of the scene. If enables it is always visible on all of the cameras present in the scene.

Screen space camera This mode is similar to the previous but a canvas with this mode enabled is rendered exclusively on the camera it is assigned to. The distance from the camera can be set for creating different effects. Also, if the camera is rendering in perspective mode the canvas will also be rendered in perspective and will follow the field of view of the camera as well.

World space A world space canvas acts as a 3D object inside the game scene. This is used for building immersive user interfaces that integrate the 3D space of the game and provide a modern feel to the interface. An obvious integration of this option is in VR (Virtual Reality) scenarios where the head of the user acts a pointing device. This is a prime example in favour for building an UI in world space.

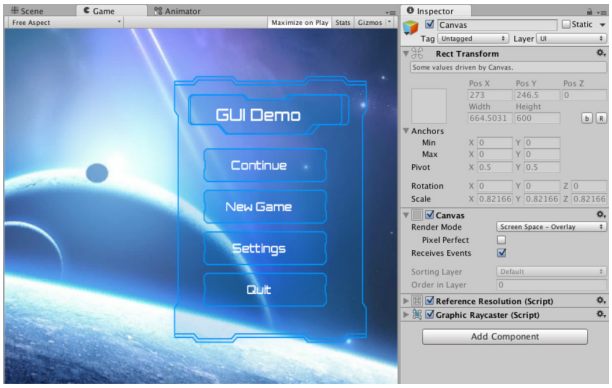


Figure 4.18: Overlay UI

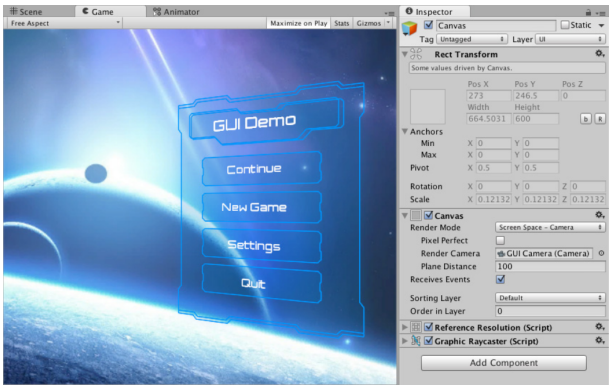


Figure 4.19: Screen space camera UI

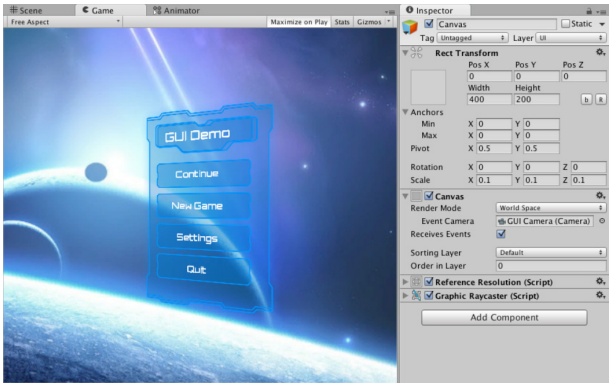


Figure 4.20: World space UI

Every Unity script derives from the `MonoBehaviour` class. Deriving from this class is mandatory in order to add it to a `GameObject` and model its behaviour. It offers many life cycle methods for controlling the flow of execution of the object during application runtime. Figure 4.21 shows the execution timing of the methods in this class. Table 4.2 shows some of uses for the most used methods in the `MonoBehaviour` class.

Method	Description
Awake	Used for initializations. Runs before the Start method
Start	Used for initializations
Update	Runs every frame
FixedUpdate	Used for physics calculations. Runs at approximately 50 times a second
LateUpdate	Called every frame after the Update method
OnEnable	Invoked when the `GameObject` is enabled
OnDisable	Invoked when the `GameObject` is disabled
OnCollisionEnter	Called when an attached collider/rigidbody component detects collision
OnCollisionExit	Called when a collision is no longer detected
OnDestroy	Called when the `GameObject` is destroyed
OnPostRender	Used for post processing effects. Runs after the camera finishes rendering a frame
OnApplicationExit	Used for cleanup. Runs when the main application is exited

Table 4.2: MonoBehaviour methods and descriptions.

4.5 ROS

ROS (Robot Operating System) is not an operating system as its name implies, since it does not manage processes or schedules tasks but a communication middleware built on the publish/subscribe paradigm. ROS is based on a peer-to-peer topology that relies on a `master` for allowing processes to find each other at runtime. Processes in this framework are called `nodes` and they are built in languages like `C++` or `Python`, making ROS language neutral. The nodes are the blocks that make up the full system. They can take the form of the robot's constituent systems like perception, estimation, data filtering, logging, energy management and networking. They communicate with each other via an IDL that defines the `topics` that flow through the communication infrastructure. These messages are interpreted by a code generator that build them in a ROS comprehensible language which are then serialized and deserialized when sent or received. Although the publish/subscribe paradigm implemented by ROS is a flexible and simple to use system it is not appropriate for synchronous transmission [27]. For this, `services` are defined by a pair of messages, one for request and another for reply. A ROS node offers a `service` under a name and a client calls this `service` and waits for the reply as if it were a remote procedure call [28].

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#

Header header        # Header timestamp should be acquisition time of image
# Header frame_id should be optical frame of camera
# origin of frame should be optical center of camera
# +x should point to the right in the image
# +y should point down in the image
# +z should point into to plane of the image
# If the frame_id here and the frame_id of the CameraInfo
# message associated with the image conflict
# the behavior is undefined

uint32 height         # image height, that is, number of rows
uint32 width          # image width, that is, number of columns
```

```
# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.sourceforge.net and send an email proposing a new encoding.

string encoding       # Encoding of pixels -- channel meaning, ordering, size
# taken from the list of strings in include/sensor_msgs/image_encodings.h

uint8 is_bigendian    # is this data bigendian?
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

Listing 4.1: ROS sensor_msgs/Image topic structure

For inserting a new message topic into the system we only need to specify it in a `msg` using strongly typed language. This file is stored inside a ROS packages and get translated into a ROS message when the package is compiled. This flexibility means that new data structures can be specified easily, regardless of their application, and integrate the communication infrastructure seamlessly. LSA did this, creating proprietary message structures for communications between their custom built robots.

4.5.1 Building packages

For building ROS packages, `catkin` build tools setup a template package whose dependencies are defined in an XML (Extensible Markup Language) file. This tools makes use of `CMake` for compiling the package and simplifies the process of integrating it into the ROS ecossystem.

4.5.2 Debugging

ROS minimizes the complexity of debugging because of its modular design, that isolates packages which makes it easier to track down bugs. With this, it is possible to run the system module by module and isolate problems. This is particularly useful on complex systems with a lot of different packages. For monitoring data transfer and topics, the installation comes bundled with a variety of tools with this purpose in mind.

4.5.3 Data logging

Data logging of systems is done by using `rosbags`, files stored in disk that contain recorded topics. This `bag` can be replayed at a later date to analyse the data or recreate the scenario of execution.

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#

Header header        # Header timestamp should be acquisition time of image
# Header frame_id should be optical frame of camera
# origin of frame should be optical center of camera
# +x should point to the right in the image
```

```
# +y should point down in the image
# +z should point into to plane of the image
# If the frame_id here and the frame_id of the CameraInfo
# message associated with the image conflict
# the behavior is undefined

uint32 height         # image height, that is, number of rows
uint32 width          # image width, that is, number of columns
string encoding       # Encoding of pixels -- channel meaning, ordering, size

uint8 is_bigendian    # is this data bigendian?
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

Listing 4.2: ROS sensor_msgs/Image topic example

4.6 Data Distribution Service

A Data Distribution Service (DDS) [29] is implemented into the system for providing QoS settings and seamless connections and reconnections to the system. It solves the multi master problem introduced by having multiple ROS masters (one for each robot) communicating between themselves. This problem makes it impossible to restart one master without disturbing the others. All of the topics from each vehicle is published into the GDS (Global Data Space) and made available to the clients connected to this shared virtual space. The DDS acts as a middleware between the vehicles and the UI. In the real world DDS is employed in medical, military, banking, robotics, aeronautical, just to name a few, proving its widespread usefulness.

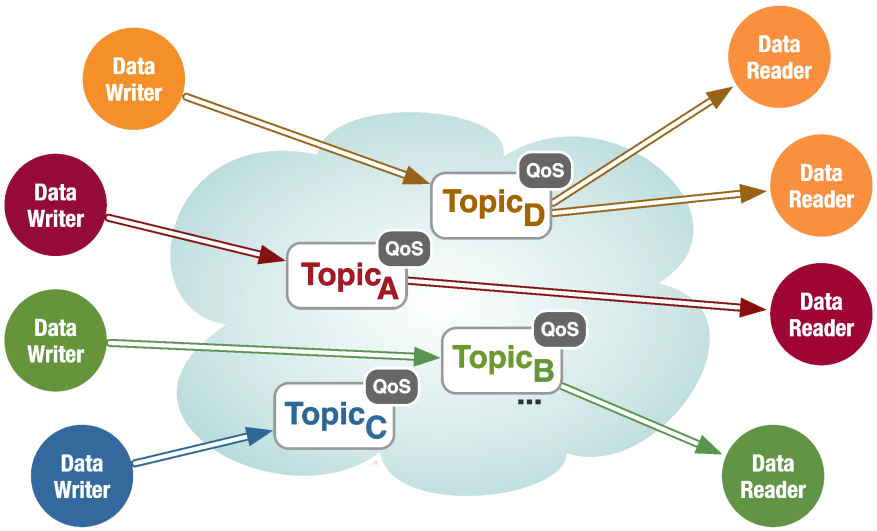


Figure 4.22: The Global Data Space

4.6.1 DDS messaging structure

The topics transmitted by the DDS follow the OMG (Object Management Group) IDL messaging standard that defines how the DDS data is structured. The syntax is "C" like and the topics are defined with `structs`. This is stored in an `.idl` file that is used as a blueprint. This is then parsed by a code generator provided by the DDS vendor that translates the files into a code library. In the context of the DDS, a topic is composed by three elements:

- type
- unique name
- QoS policies

The type and the unique name are defined in the IDL and QoS policies are defined when the topic is associated with a `Data Writer`. Keyless topics have only one instance and can be though as singletons while keyed topics have one instance per key-value

```
struct BatteryState
{
        Header header;
        float voltage; //float32
        float current;
        float charge;
        float capacity;
        float design_capacity;
        float percentage;
        octet power_supply_status;      //octet replaces the int8
        octet power_supply_health;
        octet power_supply_technology;
        boolean present; //bool
        sequence <float> cell_voltage;  //sequence acts like a C++ STL vector
        string location;
        string serial_number;
};
//Enables the generation of the DDS read and write functions of BatteryState
#pragma keylist BatteryState
```

Listing 4.3: ROS topic BatteryState represented in OMG IDL. Note the `#pragma keylist` directive that specifies the BatteryState key

4.6.2 Data writer

Writing data to the DDS is as simple as instantiating a `data writer` object. Several QoS settings can be set for the published topic.

4.6.3 Data reader

For reading data from the GDS, a `Data reader` class is instantiated. This class specifies the type of

Polling This is the simplest way of extracting data from the GDS. Polling simply checks if there is data to obtain in a non blocking call. Since this instruction is non blocking and returns even when there is no data it is not the most efficient way to read data.

Listeners `Listeners` are event that notify asynchronously the registered handler. These handler are notified of the data when it arrives. This method is advantageous over polling since it frees the CPU.

Waitsets `Waitsets` are a mechanism for waiting on conditions. The reader can be made to wait for the data to arrive. This has the advantage of making the data available as soon as it enters the GDS.

4.6.4 Quality of service

The main advantage of implementing a DDS into a communication system are its highly customizable quality of service settings. They specify a wide range of customizations for detailing the communication service for the task at hand. This can be adapted for reliable data transmission in case of critical systems or performance for fast and dynamic systems. Options for configuring data logging also exist for data keeping purposes. The QoS settings can be split into four categories: data availability, data delivery, data timeliness, and resources.

4.6.4.1 Data availability (table 4.3)

These policies manage the availability of topics from a GDS participant point of view. They manage the dynamic environments with publishers and subscribers continuously leaving and entering the GDS.

4.6.4.2 Data delivery (table 4.4)

These policies manage the reliability and availability of data, being responsible for delivering the desired information to the correct destinations at the right time

4.6.4.3 Data timeliness (table 4.5)

These policies control the temporal properties of the data. They manage the bandwidth requirements and the urgency for data to reach its destination.

4.6.4.4 Resources (table 4.6)

The resources policies configure the network and computing resources of the service for optimal data transmission.

Policy	Description
DURABILITY	Persistence of data in the GDS
LIFESPAN	Interval of time which a data sample is valid
HISTORY	Number of data samples stored for readers or writers

Table 4.3: Data availability QoS policies

Policy	Description
PRESENTATION	Controls how data changes are presented to subscribers
RELIABILITY	Level of reliability of data diffusion
PARTITION	Controls association between DDS partitions. Segregation of traffic by different partitions
DESTINATION ORDER	Orders changes made by publishers on one topic Ordering by source timestamps or destination timestamps
OWNERSHIP	Controls which writer owns the write access to a specific topic when there are multiple publishers. This value can be shared so multiple publishers can concurrently update a topic

Table 4.4: Data delivery QoS policies

Policy	Description
DEADLINE	Defines deadlines for data arrival
LATENCY BUDGET	Latency limit for information distribution
TRANSPORT PRIORITY	Priority of data delivery

Table 4.5: Data timeliness QoS policies

Policy	Description
TIME BASED FILTER	Maximum rate of data transmission
RESOURCE LIMITS	Maximum storage available for topic instances and historical samples

Table 4.6: Resources QoS policies

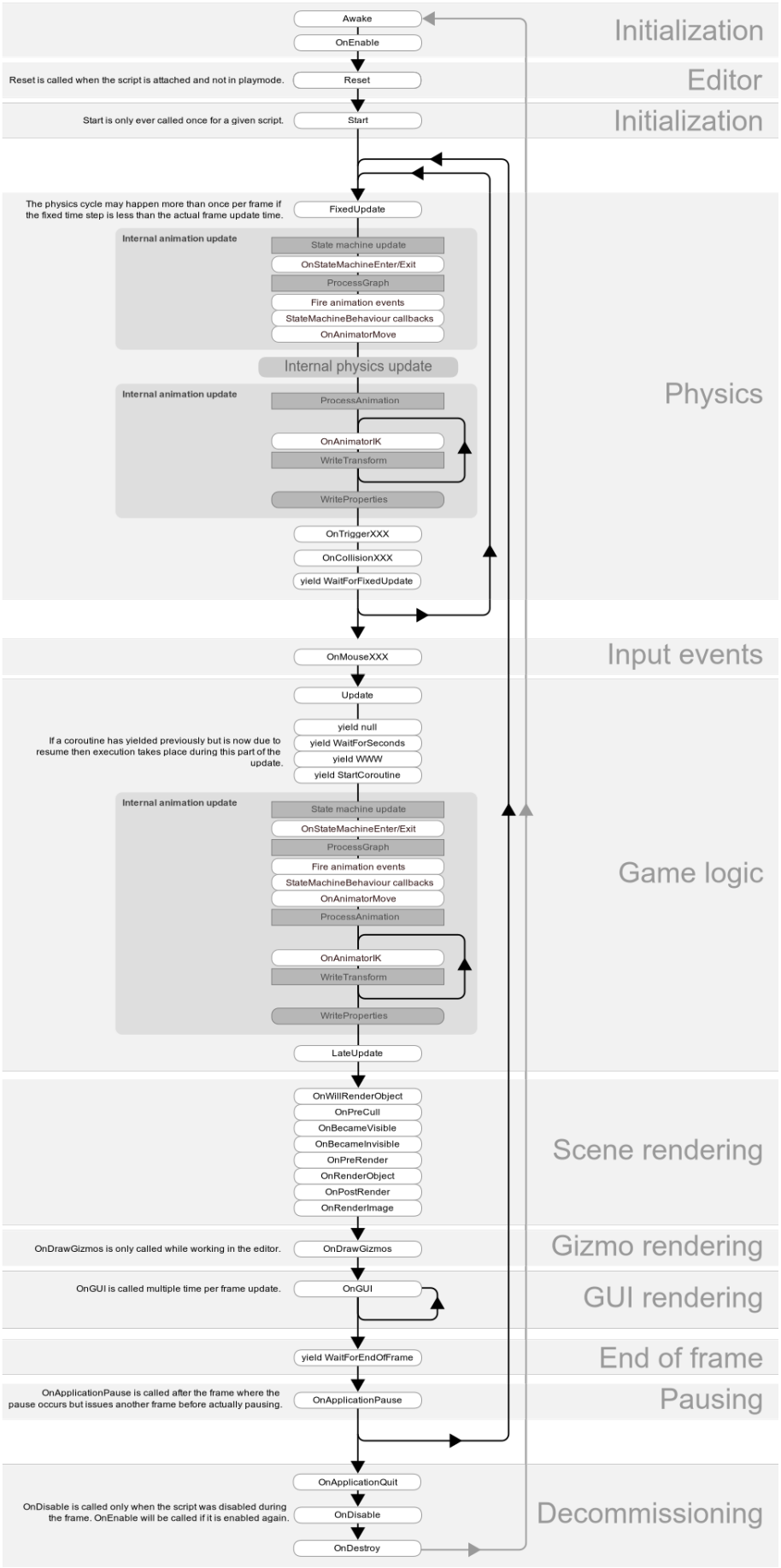


Figure 4.21: Unity order of execution methods [4]

Chapter 5

Design approach

Developing an interface for teleoperating robots raises many design and implementation challenges. Many features and design routes that are taken for granted by the user have a lot of thought process behind them being many times based on studies of human psychology and on how a human interacts with an artificial environment. The developed interface is no different from this and implements many of the design choices present in existing software. Text font, colors, animations are all components that make up a reactive HMI, providing hints to the user on how to use the product without explicitly telling on how to do so.

5.1 Color

The color of an interface component is a subtle way of conveying meaning. It is generally thought that warm colors like red, orange or yellow draw the user's attention [30] and are associated with negative actions (exit, delete, critical, emergency, warning) and because of this it is generally used for components that represent any of these actions. Buttons that delete a component, close a window, exit the program are colored red when the mouse hovers. This subconsciously informs the user that is about to perform any of these perceived negative actions. Considering the importance of the meaning behind this colour, it should be used only in situations when the user's attention needs to be drawn.

When building neutral parts of the interface, dull colors like shades of grey or pastel colors should be used because they are easier on the eyes and do not convey any special meaning.

The limited palette of colours used combine visual style with functionality while fulfilling the desired aesthetics. An elegant and streamlined user interface

is more likely to receive a positive response by the user resulting in a bigger inclination to use it. If the style of the interface does not suit the user, even if it contains a full range of state of the art features, in the end they may not be used as effectively or at all.

Figure 5.1: The use of warm colors for drawing user attention. The first image shows an hover effect of a close button and the second shows the currently selected object in the scene.

Figure 5.2: Colors represent the connections status to the sensors.

Transparency of the UI elements was implemented at first because of its benefits in providing a less claustrophobic user interface. Being able to see through elements can be at times a good feature to have, expanding the scene space and uncluttering it as well. However, it was noticed that depending on the scene brightness, the interface visibility suffered. Because of this, in the end, the use of transparent elements was discarded in favour of opaque elements, improving visibility.

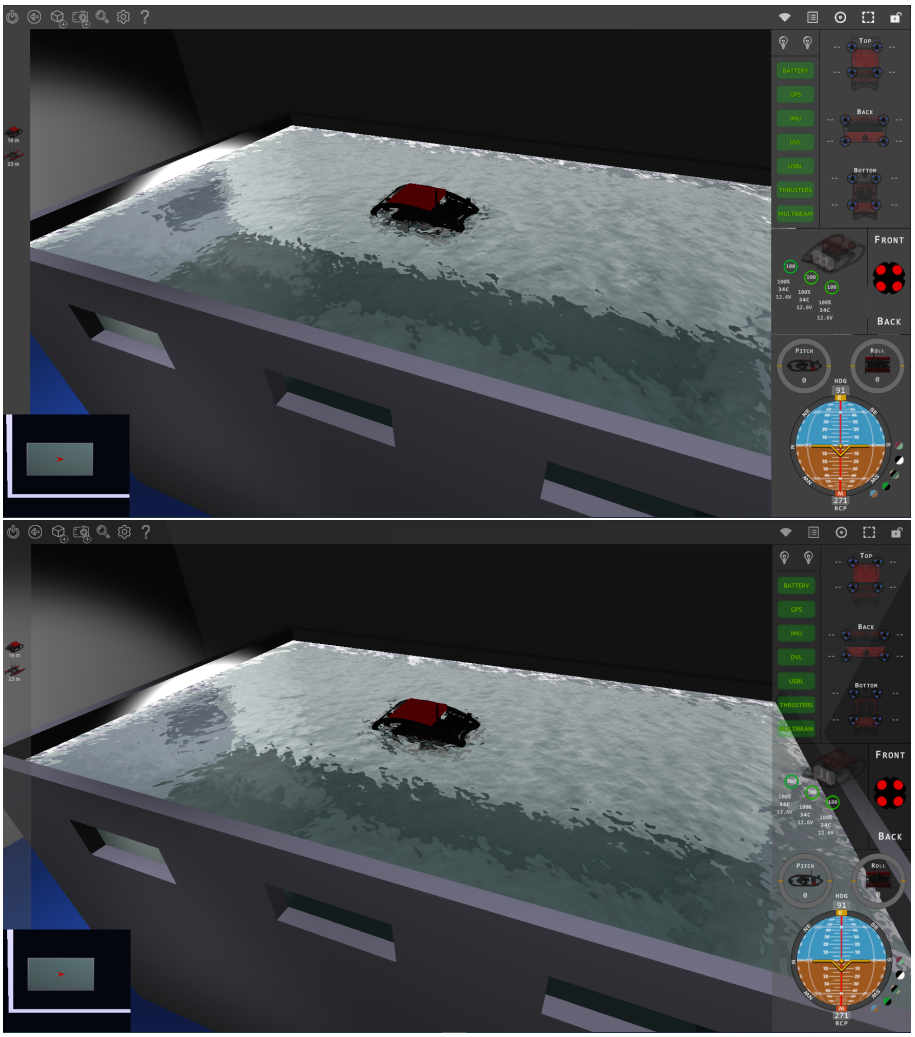


Figure 5.3: The use of transparency on UI elements gives a trade-off between the visibility of the scene view and the interface elements. Depending on what is rendered on screen, the visibility of UI elements is affected.

5.2 Text font

Despite its perceived lack of importance, the type of font presented to the user plays an important part of the user experience. `Consolas` was used mainly in the status panels for each robot and the logging window. This font is monospaced which means that all characters that compose it have a fixed width. This is fundamental in designing an UI because only the number of characters alter the physical size of the string independently of the characters that compose it. The use of expressions fully written in capital letters was also avoided because it is perceived as an aggressive way of transmitting information.

5.3 Feedback

When interacting with an interface, the user expects for its inputs to be acknowledged by the machine. If the user presses a button, loads a scene, interacts with the machine in any way, some kind of indication of that action should be given.

5.3.1 Loading

The act of loading information is done when the user requests something that needs time to be processed or loaded into the environment. A popular way of showing this is by displaying a loading screen or a loading animation that asks the user to wait until the end of the operation. This not only shows to the user that the application has not stopped responding but it is also processing the input.

Figure 5.4: Loading screen used for loading between scenes.

5.3.2 Visual cues

Whenever we interact with an onscreen element we expect an input acknowledge by the computer. An animation, color change are generally used to inform the user that the request was received and is now being processed. Throughout the software, cursor animations are used to provide feedback. For example in figure 5.1 the cursor is changed to a pointing hand to show the the button is clickable. Other cursors can be used to inform of other actions like text input, prohibited actions, drag, grab, etc. Tooltips are also a good way of dis-

playing additional information. These show up when the user hovers a button or an element presenting a small line of text.

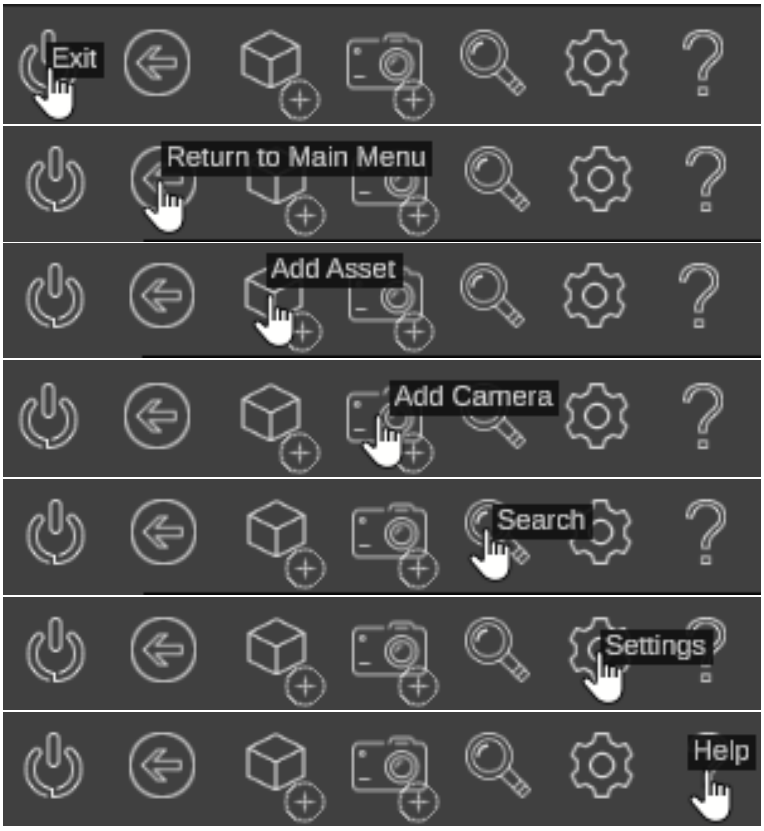


Figure 5.5: Tooltips show additional information when hovering an element.

5.3.3 Use of screen space

Given that a computer screen has a limited amount of available space, the information displayed may need to be segmented into sub menus or hidden whenever not relevant. This not only saves space but also prevents overwhelming the user with too much onscreen information. Comparing to real world interfaces we can examine a commercial airplane's controls. These interfaces are designed in a way to expose the commands without hiding them behind an extra step.

Providing this kind of interface is not feasible because there is limited screen space, therefore recycling space was needed. For example only one asset's information is shown on screen at a time, selecting another asset, changes the UI accordingly.

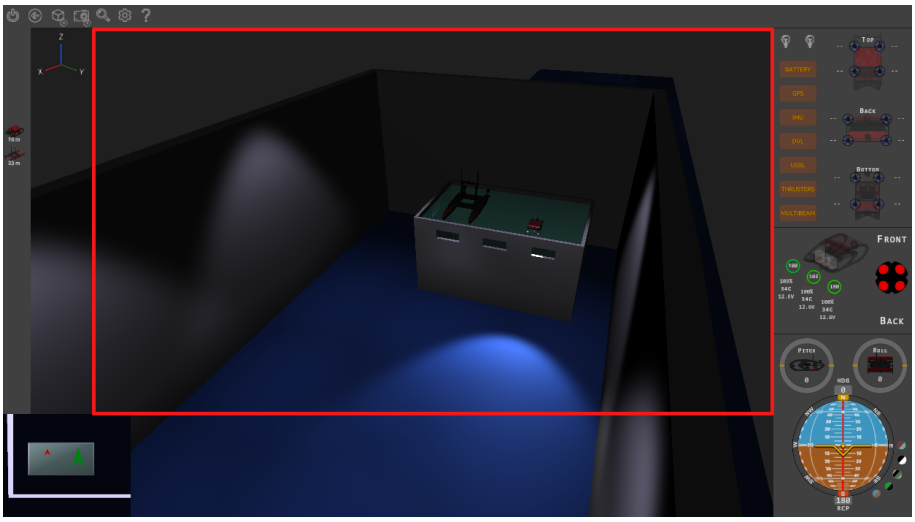


Figure 5.6: A safe space area was delineated. This area is rarely intersected by an UI element.

5.4 UI mockups

Two mockups were created to show how the UI would look as final result. By creating these prototypes it is possible to evaluate how different components on the interface fit together and how much space should they occupy.

Figure 5.7: World map scene mockup.

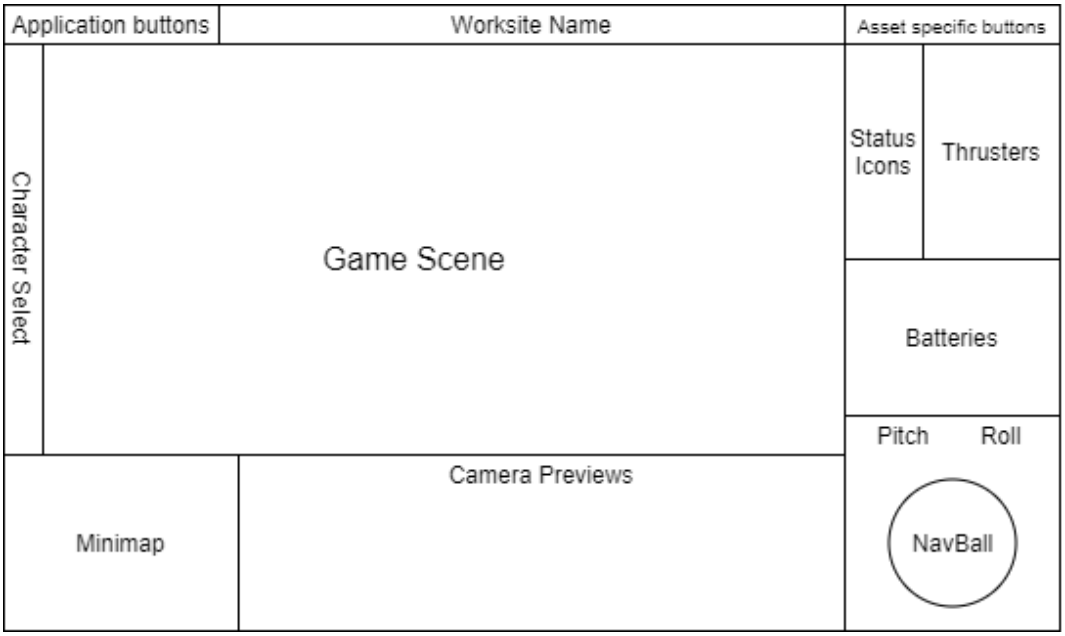


Figure 5.8: Worksite scene mockup.

Chapter 6

HMI development

This chapter represents the implementation of the various technologies and methodologies that compose the system as a whole, the challenges faced and how they were overcome. It mainly focuses on the correct interpretation of data, positioning of the vehicle in a virtual world, methods for information delivery to the user and how they were implemented. It was made an effort to prevent altering or adding new software to the vehicles. This is because increasing the CPU load will change their functioning dynamics. As a result four ROS packages are are needed, `rosbridge`, `mjpeg_server`, `async_web_server_cpp` and `ros_h264_streamer`. The last two are dependencies of `mjpeg_server`. The software was built for Linux since it is the OS (Operating System) widely used in LSA's computers.

6.1 Architecture

The entire system can be decomposed into abstract "black boxes" that integrate the application.

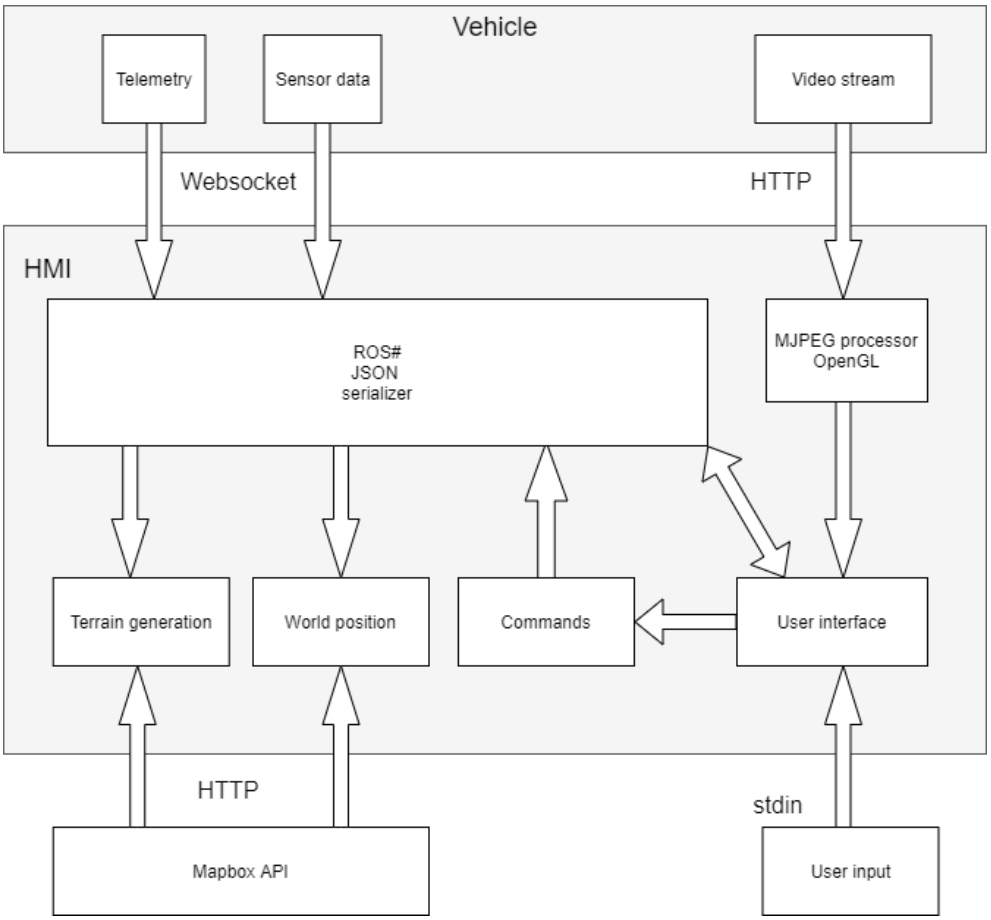


Figure 6.1: System architecture diagram.

6.2 Communications

For getting data to and from Unity several packages and technologies were implemented. ROS formatted data structures need to be sent to Unity for updating vehicle and mission status, setting up virtual worlds, and communicating with the user. The flow of data between the vehicles and Unity is done in three steps. The first step is inserting the ROS topic into the GDS (Global Data Space). A ROS node is responsible for that, subscribing to a specific topic, mapping it into a DDS topic and inserting it into the GDS. This is done in the vehicle onboard computer.

The use of a GDS decouples the publisher of the subscriber. It acts as a shared space independent of the two entities. Using this architecture, these two entities do not have to know the existence of each other being their main concerns only to publish or subscribe a message to this space. Since the GDS uses a standard-

ized message structure the only requirement that a new software has is implementing this standard for communicating with the GDS.

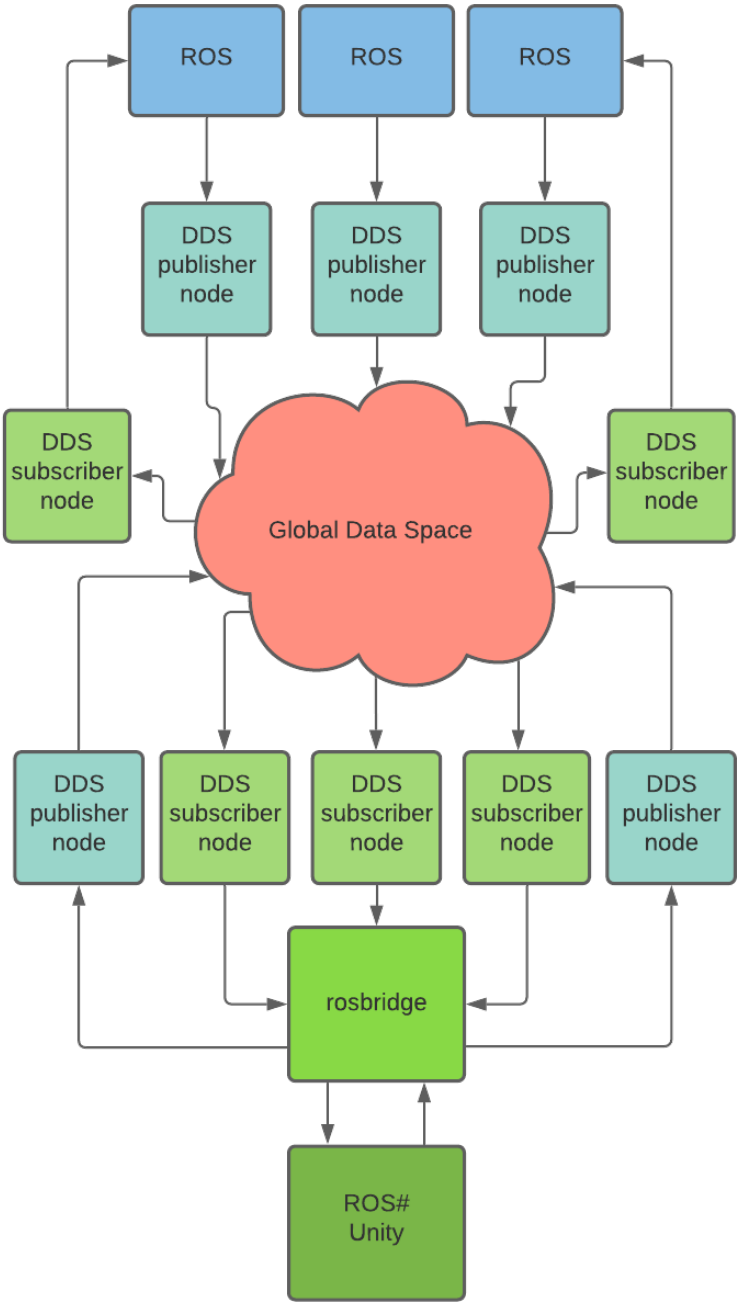


Figure 6.2: Communication flow between the ROS topic space and Unity, through the DDS middleware. Note that the only data that is not accessed through rosbridge is the video stream. For this `mjpeg_server` was used.

Even though the DDS possesses various desirable features, its integration is not mandatory. The system was made to function even if no DDS layer is present. This shifted the communication system from a centralized data structure to a peer-to-peer one, where each robot is assigned a unique IP address inside a LAN (Local Area Network). This address is used by `rosbridge` for sending data to Unity via HTTP.

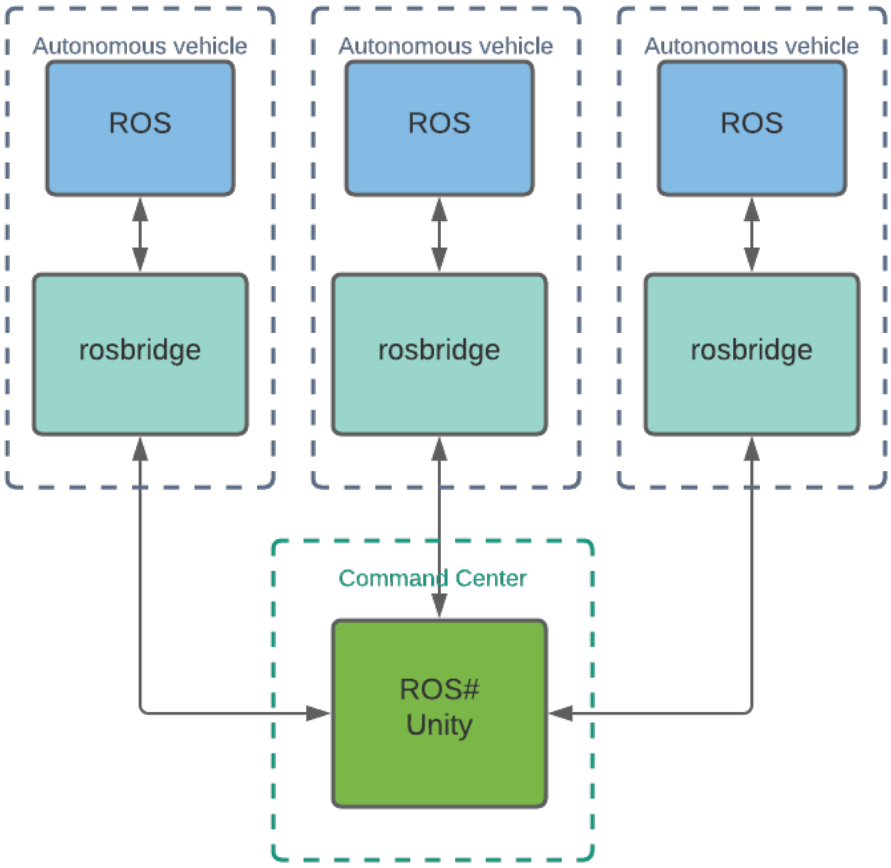


Figure 6.3: Communication flow between the ROS topic space and Unity, without using the DDS middleware

6.2.1 Data Distribution Service

A ROS package was created that embedded the DDS into a ROS node. This way the publisher/subscriber could be configured before launching the node with a `roslaunch` file. This node translates topics from the ROS space to the DDS space and vice-versa. This way, DDS encapsulated topics are exposed in the ROS space ready to be accessed by the Unity application.

6.2.2 RosBridge

`Rosbridge` ROS package is used to tunnel the available topics into `websockets`. This `node` subscribes to the ROS topics, converts them into JSON (JavaScript Object Notation) format and serves then over a `websocket` connection. This ability enables non-ROS application to access the ROS topics.

6.2.3 ROS#

The JSON serialized data that comes through the `websocket` connections created by `rosbridge` is de-serialized in this final step. Siemens made available a tool for communicating with ROS from .NET applications. This library was imported into the Unity code structure via `dlls`.

Interacting with the ROS workspace by connecting to the `websockets` provided by `RosBridge`. This library de-serializes the JSONs tunnelled through the `websockets` and instantiates C# data structures containing the data.

6.2.4 Cross platform compatibility

This communication structure allows the exchange of data between machines running different OSes. If for example the Unity interface were to run in a Windows machine, we could access the data because it is served through `websockets` (telemetry, perception, commands) and `HTTP` (video). Running ROS on the command center machine is not mandatory for accessing this information. To run with DDS we would only need a DDS subscriber and a local server distributing the DDS topics.

6.3 UI Layout

The user interface is split into various parts that are responsible for showing different kinds of information or providing options to the user. On the right side

The layout of the UI was split into five sections. The center of the screen was obviously reserved for displaying the main scene. The borders of the screen each play a different role in conveying information or providing access to different features. Top bar holds the buttons for application user options like quitting the application toggling the options menu. The left side provides thumbnails for each of one of the vehicles in scene. Bottom side is home of the minimap and camera previews. Lastly, right side shows information about the selected vehicle.

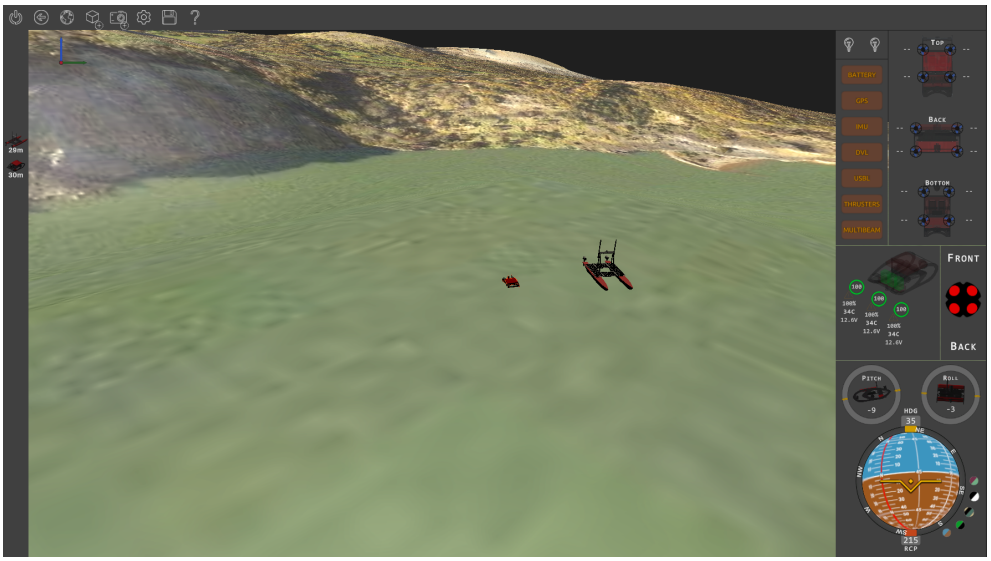


Figure 6.4: UI implementation for EVA.

6.3.1 Right section

Positioned to the right side of the screen, this area is reserved for displaying vehicle specific information like sensor data, commands and control. Autonomous vehicles are equipped with a wide variety of sensors. It is very important the their data is interpreted correctly since they represent real world data.

Attitude panel

Arguably, one of the most important sensor systems is the INS (Inertial Navigation System). Present in every single robot, its correct representation is key in understanding the current attitude and positioning of the vehicle in the world, especially in an underwater scenario, where physical limitations inhibit GPS data forcing the reliance in dead reckoning. The raw INS data is fused with other sensors generally by using an Extended Kalman Filter by the vehicle's onboard computer.

This calculation is done outside of Unity which simplifies the interpretation of information. In this case, Unity is only responsible for representing correctly the already processed data. This position is then referenced to the origin point of the world map inside Unity. This is done by referencing the first position reported by the robot to the centre of the virtual world map.

Heavily influenced by the space exploration video game *Kerbal Space Program*, the navigation ball provides information about the three axis of rotation (roll, pitch, yaw) in a compact and intuitive way. It follows the attitude of the

selected vehicle. Its border acts as a compass pointing to the current heading and reciprocal heading. Additionally the navball texture can be changed in order to provide accessibility options to the user. Also in this panel two images for representing the pitch and roll of the vehicle were added.

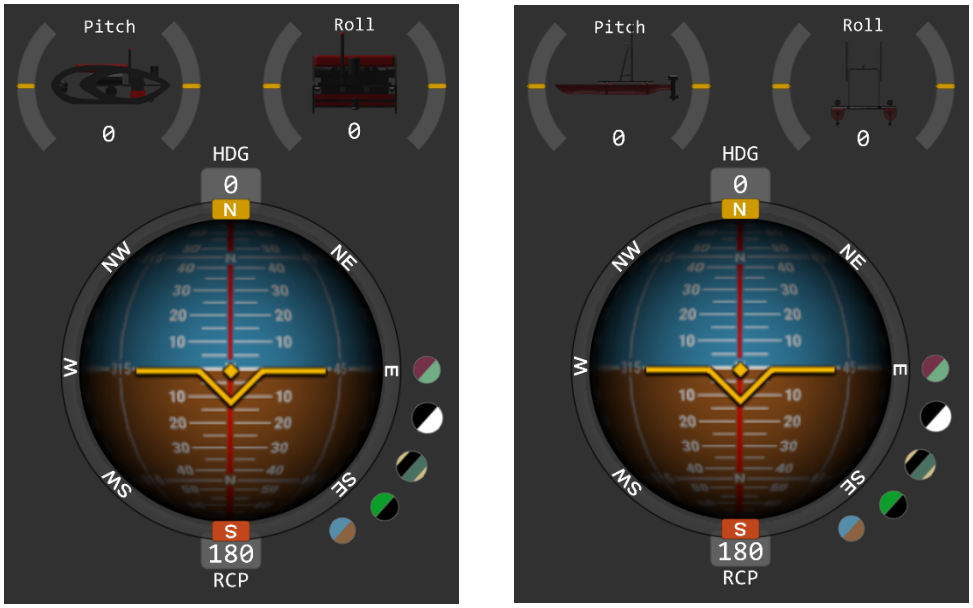


Figure 6.5: Attitude Panel

In order to aid in identifying the current attitude of the vehicle, an axis gizmo and a grid plane were implemented.

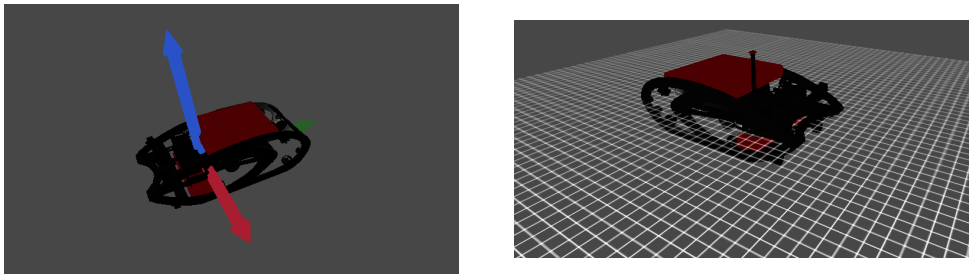


Figure 6.6: Axis gizmo and grid plane

Batteries

Remaining battery display is very important in the planning of an autonomous or semi-autonomous mission. A circular shape filled by the amount of energy percentage left along with different stages of battery levels highlighted by color conveys a direct and instant message to the user. A green hue is generally associated with good feedback so this color is used for full charge. In contrast, the

color red is associated with bad, urgent, emergency, so this color is used for the critical levels of battery. Between these two poles a color based on a gradient (*figure* 6.7) is set for each value of battery.

Figure 6.7: Battery circle color gradient

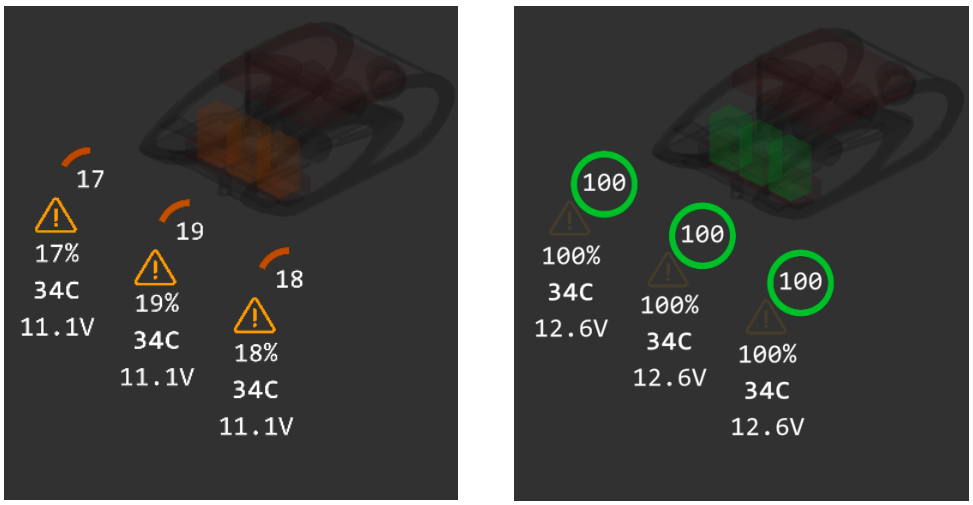


Figure 6.8: EVA's battery panel

The SoC of each battery is represented independently and a critical level is set to alert the user for low battery levels accompanied with a low battery indicator. Along with the graphic display of battery, the temperature, voltage and state of charge are also displayed by text.

DVL panel

Present in underwater vehicles like EVA, the doppler velocity log calculates the linear velocity of the robot in three dimensions by taking advantage of the Doppler effect perceived on its acoustic beams. Along with the velocity of the vehicle its distance to the seabed can also be obtained. The representation of this sensor depicts an image of the DVL with the distance to the seabed represented by a green to red colour on each of its beams, being red close distance and green safe or far distance to terrain. Much like the batteries representation, the DVL beams are colored depending on the proximity to terrain.

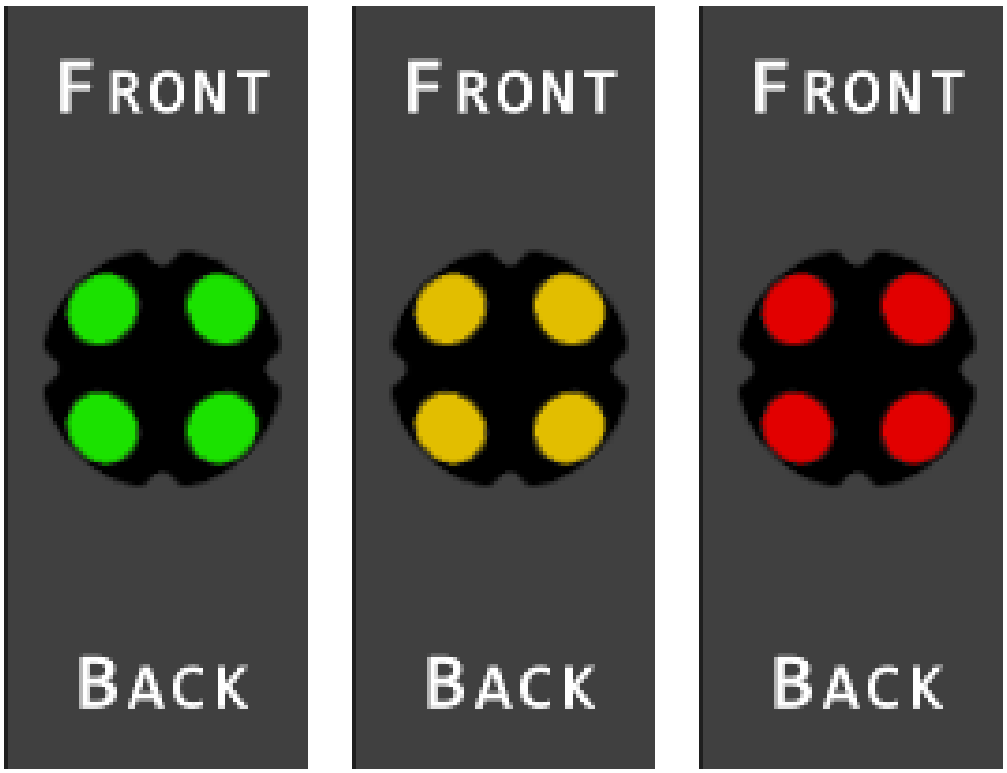


Figure 6.9: DVL panel.

Thrusters

The thrusters are represented from a top, bottom and back perspective of the vehicle when applicable. Propeller sprites are overlaid over the different vehicle planes and rotated based on the real speed of the propellers. RPM (Revolutions Per Minute) data is also presented in text form.

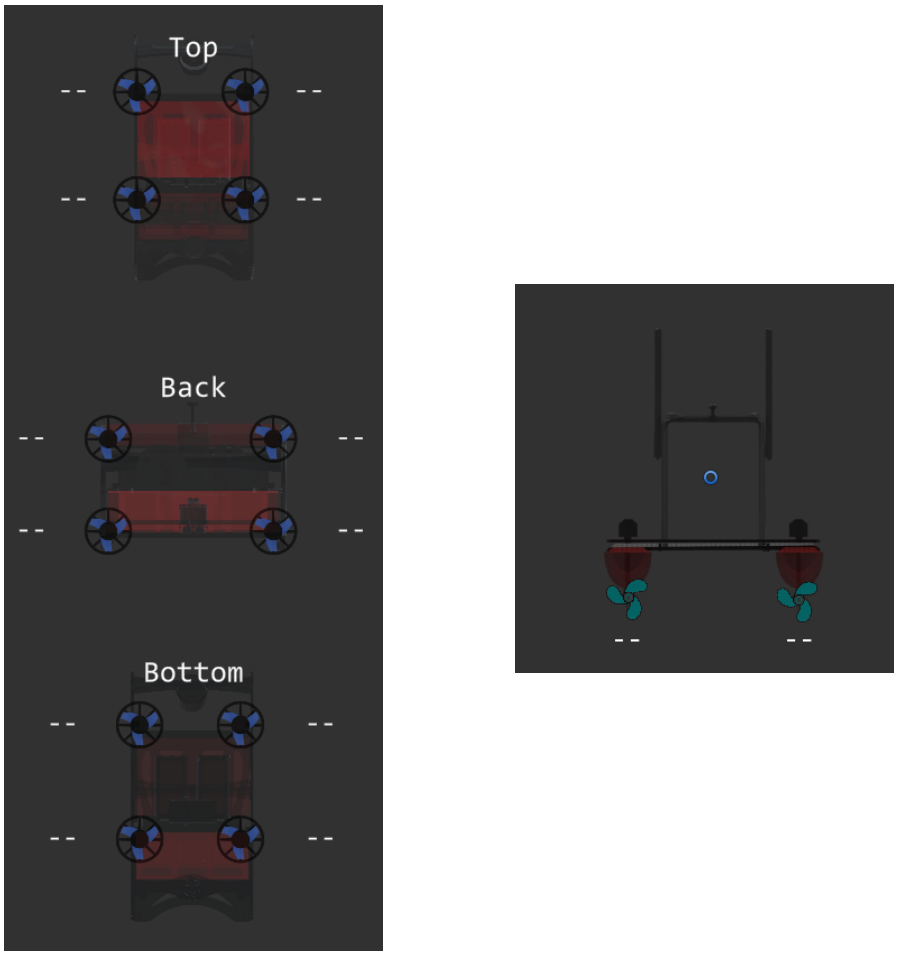


Figure 6.10: Thrusters panel for EVA (left) and ROAZII (right). The RPM is shown for each thruster.

6.3.2 Bottom section

The bottom section of the UI houses the minimap and the camera video streams and are added by the user whenever they are needed.

Camera previews

It is an important feature for the user to be able to see what the vehicle "sees". The implementation of camera streams is done, in simple terms, by extracting the frames captured by the camera sensor and drawing them to a Unity texture. Initially, `ROS#` was used to transfer the frames to Unity via websockets. This posed big bandwidth problems. With a few calculations we can infer the amount of bandwidth that only one video stream would require. For example, given a camera with an HD (High Definition) resolution of 1280x720 at 30 frames per second, encoded in bgr8 (8 bit per color channel), requires an array of unsigned

8bit integers with a size of $1280 * 720 * 3 = 2764800$ bytes or about 2.8 Mbytes, since each pixel requires 3 bytes to be represented. To be able to transfer all of this data as fast as it is produced, we would need a bandwidth of $2.8 * 30 = 84$ MB/s. This is calculated while disregarding the data occupied by the rest of the ROS `Image` `struct` like the message header, encoding, width and height and the endianess of the data. This problem would escalate to unpractical levels as more cameras would be added. ROS also provides a `CompressedImage` type that compresses the video stream to `png` or `jpeg`. This reduces the bandwidth requirements considerably and this way camera streams could be transmitted to Unity effectively.

6.3.2.1 Publishing video streams

Even though `rosbridge` could be used to subscribe directly to the `Image` topics this is less than ideal because bandwidth limitations are encountered and they are not available as a through an URL (Uniform Resource Locator) so flexibility is lost. The best solution encountered was to use a ROS package called `mjpeg_server`. This package subscribes to `Image` topics, compresses them into `mjpeg` and makes them exposes them through an `HTTP` (Hypertext Transfer Protocol) URL. By using this package the topics are compressed and are made accessible. A drawback of using this method is the bypassing of the DDS when transmitting video streams. A possible workflow that would retain the DDS layer would be to subscribe to `Compressed Image` topics, convert them into `Image` topics and serve them locally with `mjpeg_server`.

6.3.2.2 Parsing video streams

Now that the video stream is available through an URL a way to process the bytes that it returns is needed. A `jpeg` image is an array of bytes in which the first (`0xFF 0xD8`) and last (`0xFF 0xD9`) two bytes are known as indicated by the `JPEG` standard [31]. This can be visualized in an hexadecimal editor program like in figure 6.12. In this case the end bytes are not needed because the `HTTP` request only returns the `jpeg` array so the EOF (End-of-File) is the last byte of the array. Given this only the start of the array is inspected. Upon detecting the two start bytes it is known that the rest of the array corresponds to the compressed `jpeg` image frame. The result is then saved in memory for decompressing and later assignment to a texture.

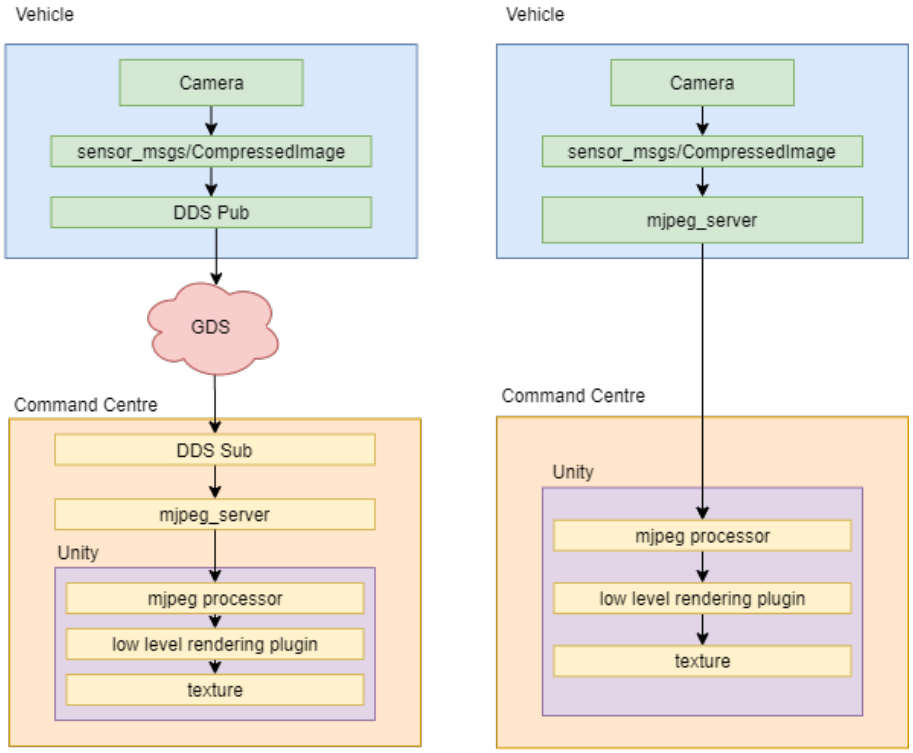


Figure 6.11: Video stream workflow with (left) and without DDS (right). It assumes that the cameras inside the vehicle publish `CompressedImage` topics encoded in `jpeg`.

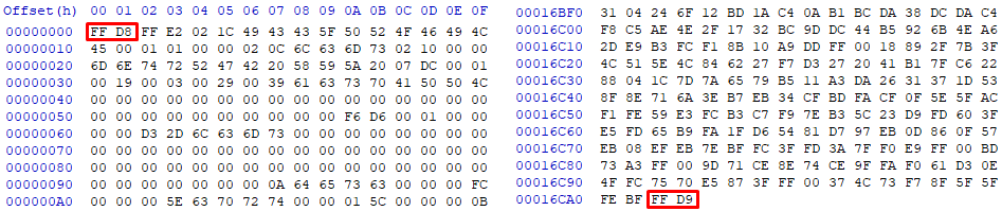


Figure 6.12: JPEG hexadecimal view showing the start and end bytes

The aforementioned data array was parsed fully by software at first. Upon receiving the image topic, this array was converted into a `Texture2D` [32] and then into a `RenderTexture` [33] before being displayed in the UI. The process of loading the `jpeg` image into a texture can be done by using Unity's `Texture2d .LoadImage` class method. The main issue with using this function is that it is an instruction that decompresses the `jpeg` and assigns it into the texture while blocking the rendering thread. This causes performance slowdowns each time a new frame is received. To solve this issue, a low level rendering plugin [25] for decompressing the `jpeg` (`libturbojpeg` [34]) and OpenGL for assigning the

image to the texture was developed. The idea behind this was to make use of Unity's multi-threaded rendering pipeline and free the main rendering thread. The way Unity deals with this interaction is via a `dll` (Dynamic-link library). Upon receiving the message, a pointer to the Unity Texture2D and a pointer to the data array were passed onto the rendering plugin so that OpenGL can write the data to a texture and return it to Unity. Figure 6.13 shows the widget the displays the image stream. this widget can switch between topics by using a dropdown menu and can launch an external player with the video stream.

Figure 6.13: Camera preview widget.

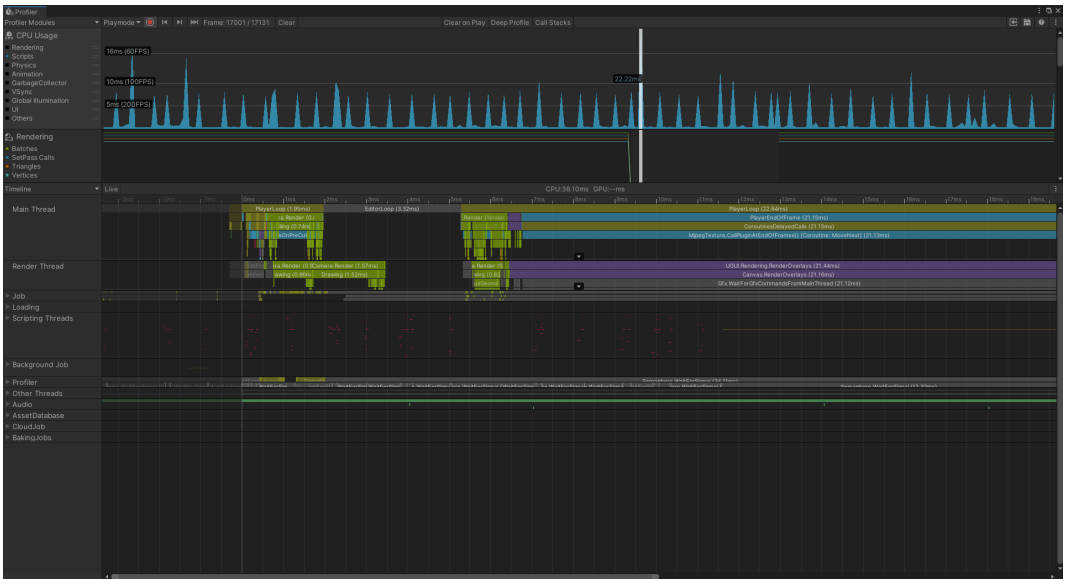


Figure 6.14: Unity profiler showing the CPU usage when using the `LoadImage` Unity method.

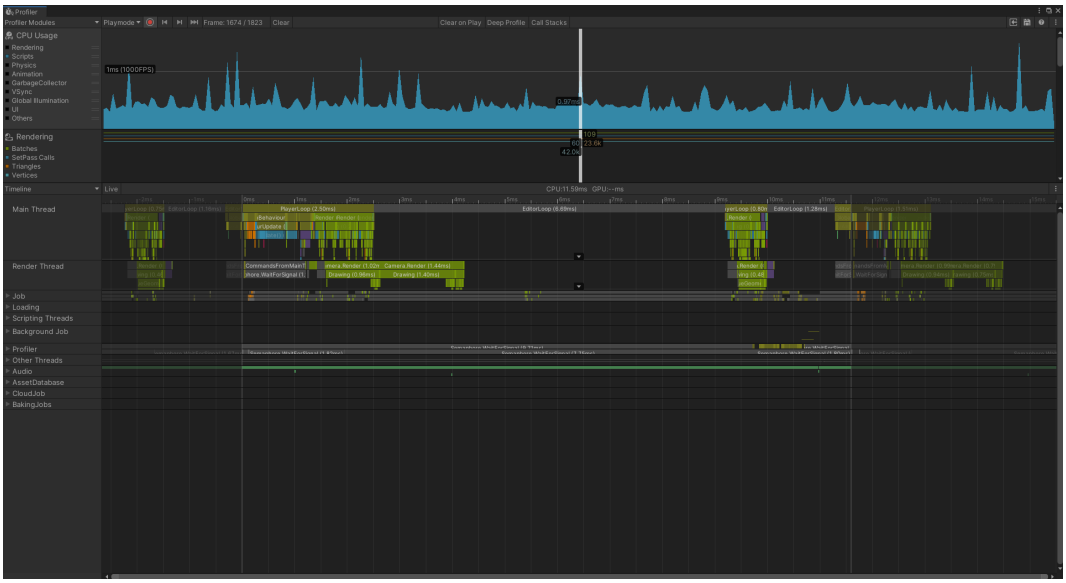


Figure 6.15: Unity profiler showing the CPU usage when using the rendering plugin. Note the reduced CPU usage spikes.

6.3.2.3 External video player

The limited amount of available space in the UI meant that multiple camera view could potentially occupy much space. Giving the user the option of opening the video streams outside of Unity frees a lot of screen real estate. This is done by opening an external player like `mpv`, `vlc`, `ffplay` or a web browser and loading the URL for the stream. This has the advantage of giving the user the ability to position the window in an external monitor in case of a multi screen setup. The external player can be selected by the user in the options menu.

Figure 6.16: VLC player used for displaying a ROS image topic video stream

6.4 Minimap

The minimap is a section of the UI that shows a 2D top view of the mission scenario. It displays icons that represent the vehicles and other points of interest like waypoints and objectives. Its design is heavily inspired by the video game industry as it is present in a great variety of games. For implementing this ele-

ment, a second world map is obtained from Mapbox but without elevation data and with a satellite layer like the main world map. A camera in the Z axis pointing down at the map plane is used to render the image. Mapbox API provides the flexibility for adding markers to this map by transforming a position of the virtual map into a real world geodetic position. Both vehicle minimap markers and waypoint markers are added this way. GPS (Global Positioning System) geodetic information is layered onto the minimap.

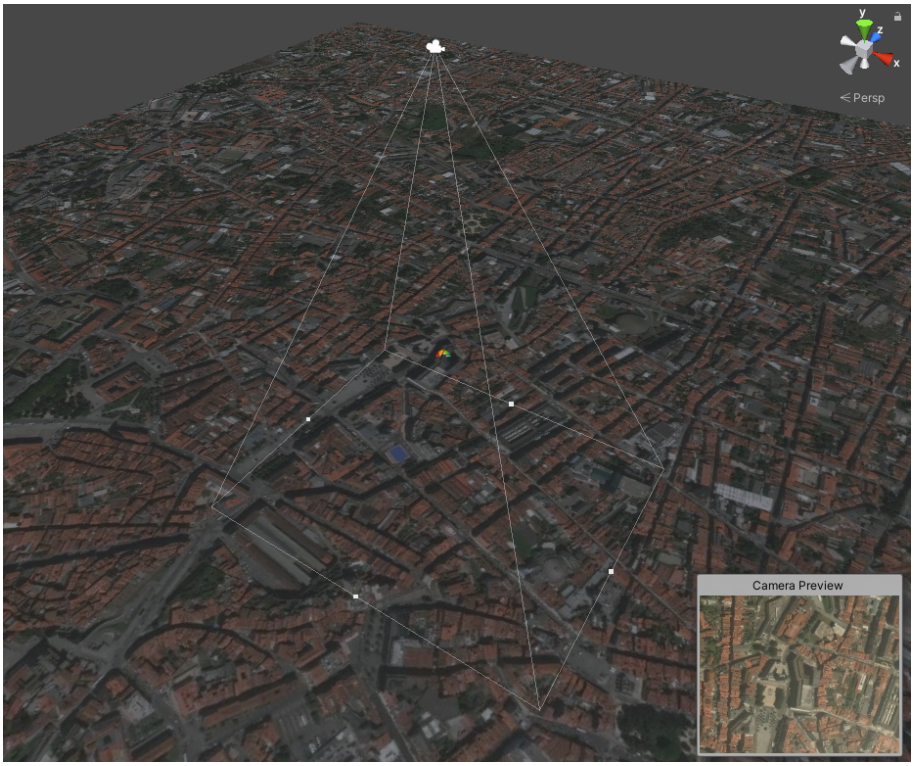


Figure 6.17: Camera rendering the minimap view

Figure 6.18: LSA's water tank scene with minimap (bottom left) showing the two assets. Green icon represents ROAZII and the red EVA.

6.4.1 Left section

In the left section is positioned the "character select". This widget tracks the vehicles in the scene and provides quick access to them. An image of each one is displayed in this panel along with the distance in meters of the main camera to them. This helps the user locate itself in the scene and provides a way to select a vehicle with one click and focus it by double clicking the image. The distance represents the Euclidean distance between the two objects in space.

6.4.2 Top section

The top section of the screen was reserved for auxiliary buttons. These buttons give the user general options like adding an asset to scene or adding a camera stream or exiting the program. Also in this bar, space was reserved for the logger output. This component displays instant information to inform the user of actions that occur during the execution of the program, with a timestamp. Some of these logs can be to inform that an asset was successfully added to the scene or that a vehicle has reached its destination. This log can then be saved to disk for later analysis.

6.5 Meshes

This section describes the various meshes and methods that compose a scene. They can take the form of the robots in action, terrain, instantiated point clouds and world space user interface.

6.5.1 Vehicles

Meshes that represent the vehicles in scene are constructed initially using CAD (Computer Aided Design) programs. Since they are part of engineering projects its measurements are precise and its mesh is complex. This complexity is reflected in the mesh, modelling every single screw, bolt and intricate details. However, since this added complexity translates into vertex count we end up with unneeded detail in the mesh. This is resolved by decimating the mesh using a 3D editor like Blender.

Decimation procedure

Firstly, the mesh processing program MeshLab is used to convert between mesh types from `STEP` to `stl`, a format recognized by Blender. After this, Blender is used to inspect the wireframe of the mesh and analyse where excess vertices are and delete them. With this treatment the meshes visual quality can remain the same since we are eliminating detail that is not seen. This way performance can be improved by not rendering excess vertices.

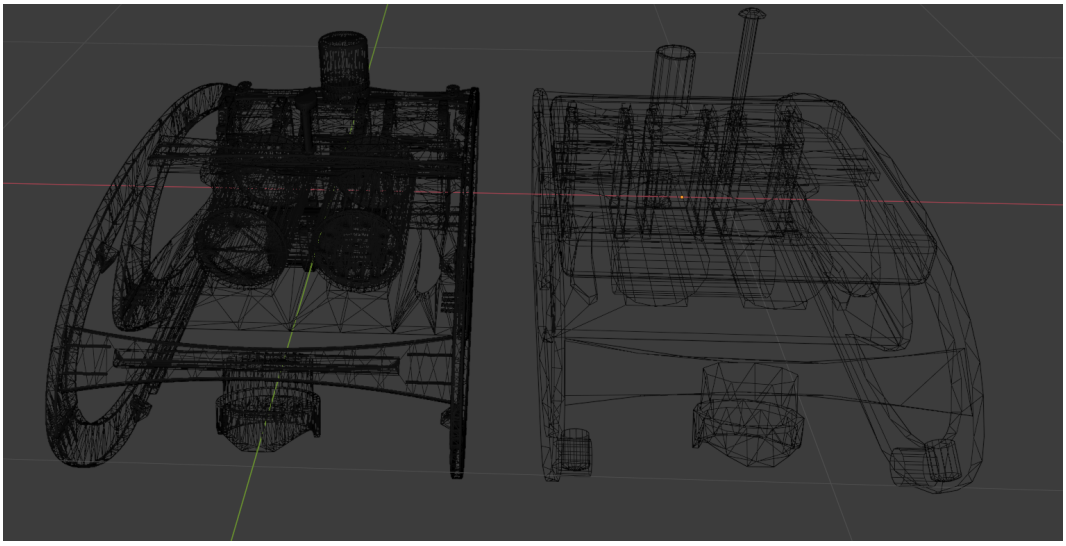


Figure 6.19: EVA decimation procedure before and after. The resulting mesh saw a 98.7% decrease in the number of vertices, while not compromising visual fidelity.

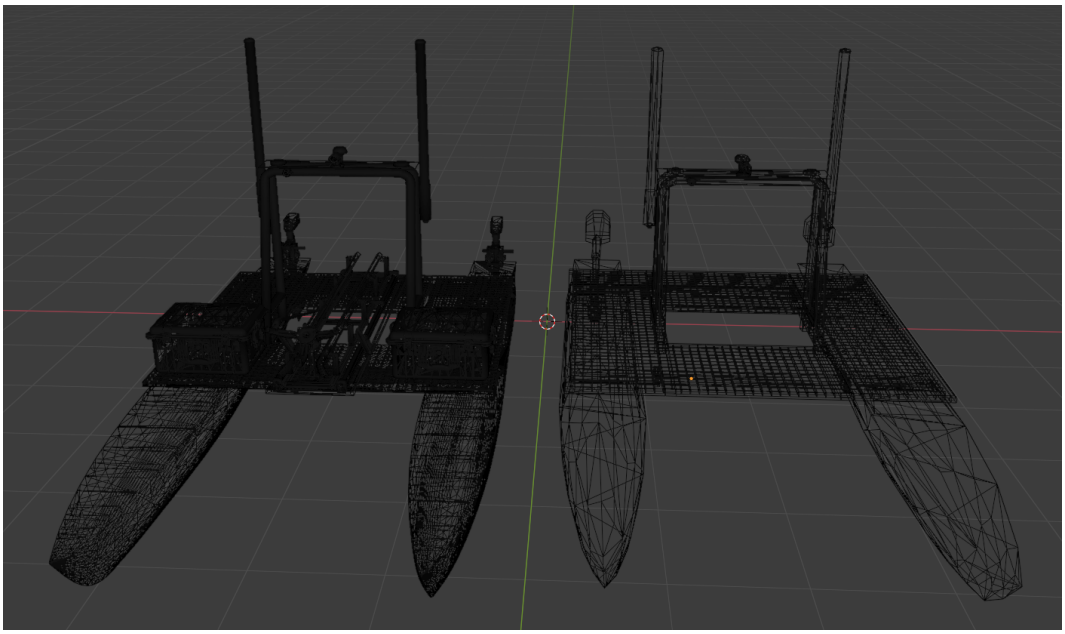


Figure 6.20: ROAZII decimation procedure before and after. The resulting mesh saw a 99.1% decrease in the number of vertices.

6.6 Terrain

It is comprised as terrain every mesh or feature that represents the world where the vehicles actuate. This section describes the type of terrains and their integration in a scene.

6.6.1 Instantiated Terrain

The main feature provided by the Mapbox API is its capabilities for instantiating terrain based on real world satellite imagery. For this, a center geographic position is provided to the API, along with the range of the map in kilometers. With this information, the API is capable of instantiating a square section of a map with the specified range centered on the provided geographic location. Elevation data is also embedded in this map.

Figure 6.21: Terrain mesh instantiated using the Mapbox API. Silvermines, Ireland.

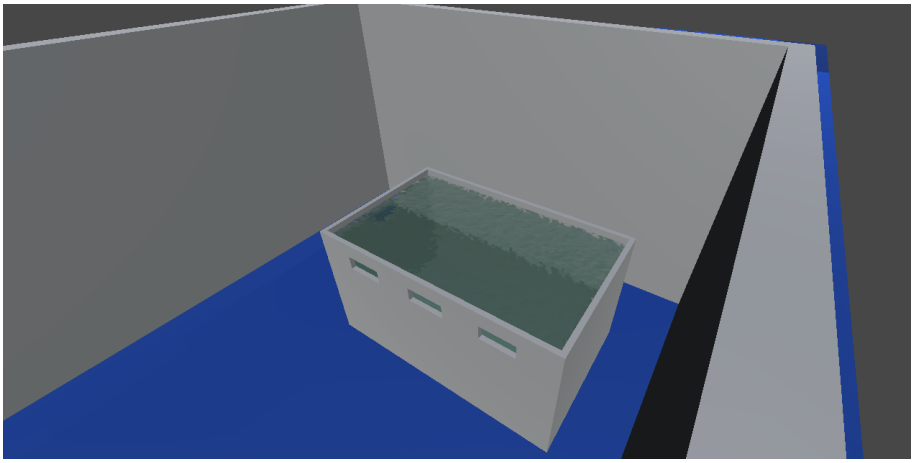


Figure 6.22: Static mesh representing the LSA water tank used for the test scene.

6.6.2 Procedurally generated terrain

Sensors like a LiDAR or a multibeam sonar are capable of sweeping an area and provide a 3D overview of the scanned area. The generated point clouds are

interpreted in Unity as meshes displaying the topology of the terrain. Figure 6.23 shows a point cloud of a LiDAR sweep made using an aerial vehicle. The point cloud information is stored in an xyz file containing each point position and color. Reading this type of file requires multithreading to prevent freezing the application since they have a significant size. By reading the file on a worker thread, the application can continue its execution. Meshes in Unity have a maximum vertex count of 32 bit which translates to roughly 4 billion points. In the corner case of existing a point cloud that exceeds this amount of vertices, the meshes would need to be split into multiple objects.

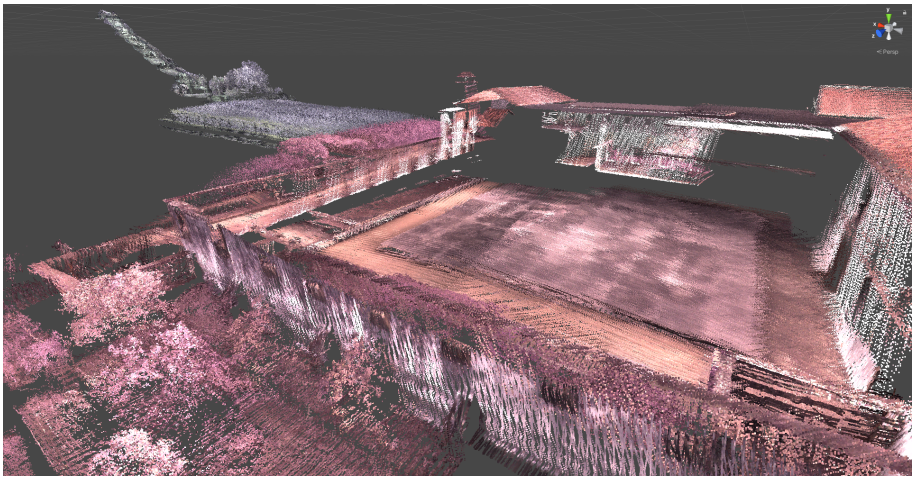


Figure 6.23: RGB point cloud of Tibães monastery instantiated inside Unity.

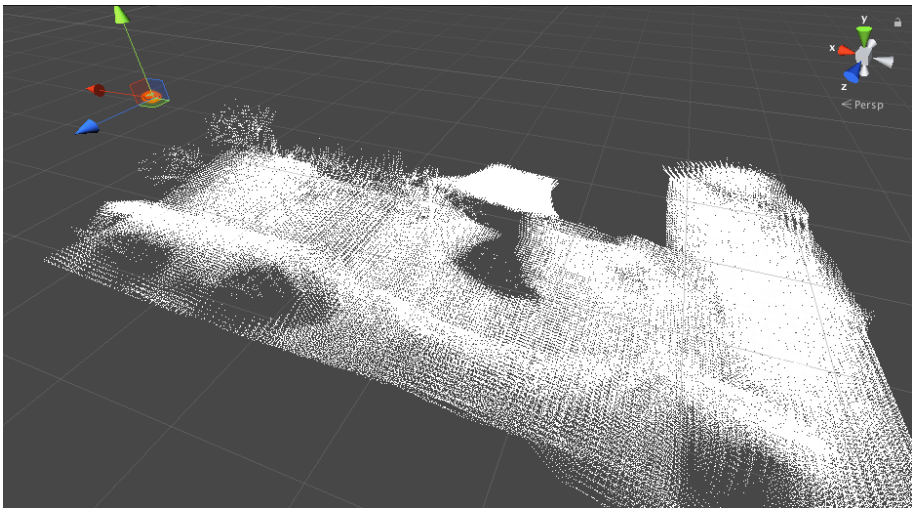


Figure 6.24: Point cloud generated by a vehicle.

Instead of reading a file, point cloud data can be instantiated by subscribing to a point cloud topic from a vehicle asset. Each message from this topic holds point cloud data that is interpreted by Unity and instantiated in the scene, relative to the vehicle that published it.

6.7 Object selection

While browsing the scene the simple act of selecting an object in view is natural to the user. The implementation of this feature is done by emitting a raycast when the left mouse button is clicked in the scene. This raycast interacts with colliders that wrap the selectable objects. If there is an intersection between these two elements, an action is triggered, notifying the Unity event system that a collision of this type has occurred. A trigger like this returns the object with which the raycast collided. With this, an outline shader is set to the material of the object, giving the user feedback of the selection process.

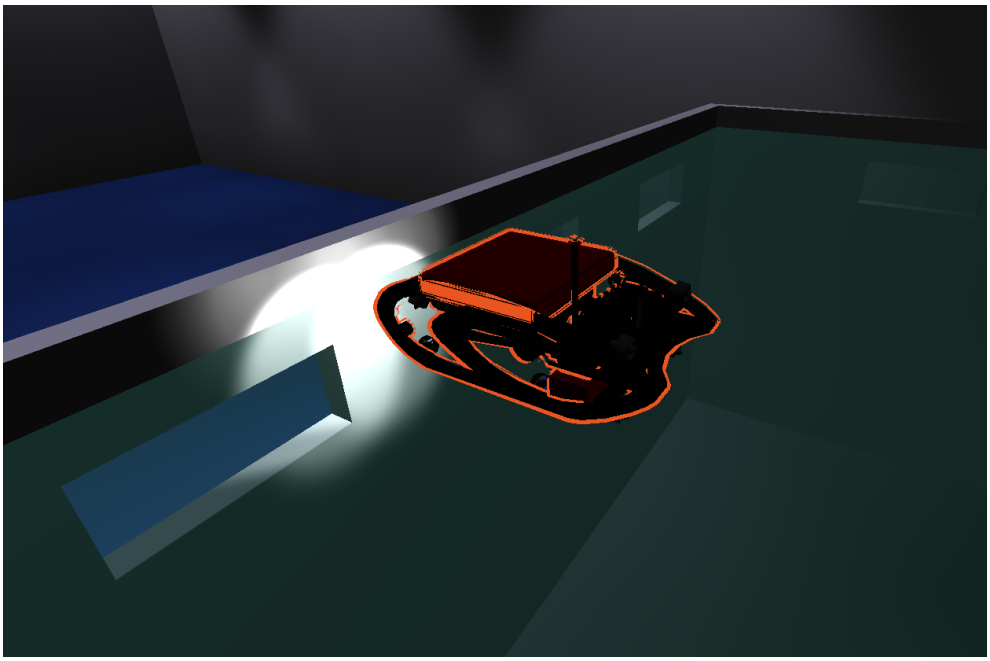


Figure 6.25: Current selection object outline.

6.8 Camera modes

6.8.1 Free camera

A free camera, like the name implies, is not constrained to any axis or object in the scene, being only locked in roll. Movement around the scene is done using

the WASD keys for forward, left, backwards, right, respectively and the QE key for moving up and down. Pitch and yaw control is done by holding the right mouse button and moving the mouse in the desired direction. Unity's input system is used for polling input data on each game frame. In can it detects any keypress behaviour can be programmed to act upon this event and in this case move that camera as desired.

6.8.2 Orbit camera

The orbit camera is used to constrain the camera movement to a vehicle or object in the scene. The view will be focused on that object and with right mouse click it can be orbited around this object. To achieve this we can calculate the direction to where the camera needs to point to by subtracting the camera forward pointing vector with the target's position vector. This difference results in a new vector that represents the direction to where the camera has to look at. By multiplying a quaternion with this vector, we can apply a rotation to it and therefore resulting in an orbit effect around the targeted position.

6.9 World map

A world map view was created to give the user an overview of the available worksites from around the world or create a new one and start setting up the mission environment. The map is instantiated using Mapbox's API, giving the user the possibility of panning and zooming the view for closer inspection of the map area. The existing worksites are loaded from JSON files with the worksite latitude and longitude, location description, assets it contains and an unique GUID (Global Unique Identifier). The user can also change the map tiling to render the map in street view or satellite view. Unfortunately this feature is only available with an internet connection which may not be available on the field. In this case, the user should load or create a new empty worksite.

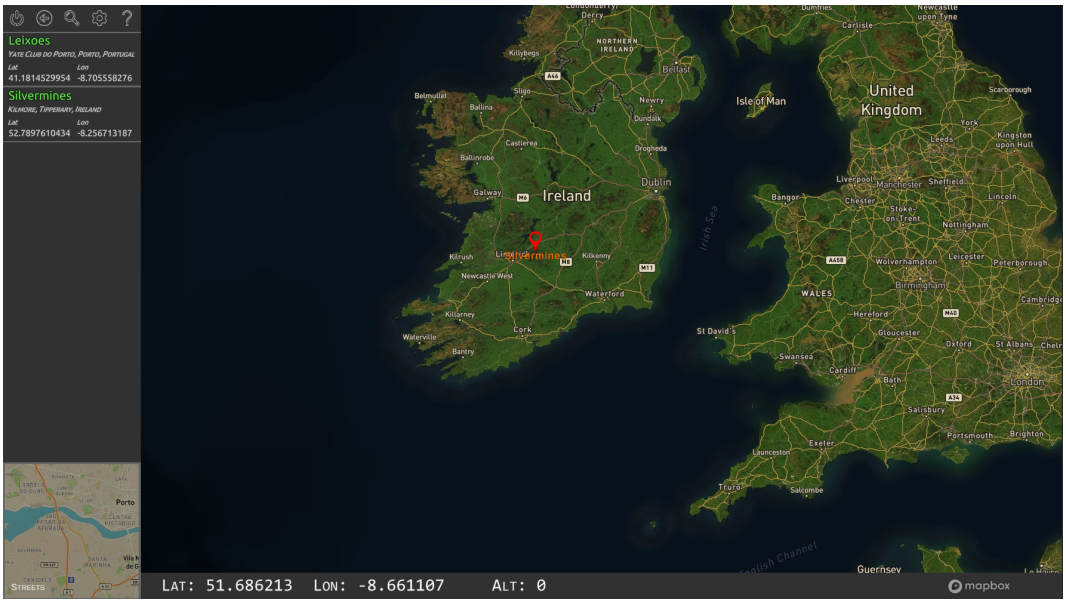


Figure 6.26: World map scene.

6.10 Scenarios

Robotics missions take place in a variety of difference scenarios. A focus on underwater environments was made because of the importance that they play in LSA's mission repertoire. However, the existence of a general, empty scenario means that an environment for any kind of mission could be created dynamically, through the use of mapping sensors. For example on a surface-aerial mission, the map generated by Mapbox, like the one in figure 6.21 is enough to conduct these types of missions.

6.10.1 Underwater

While underwater vehicles all apply to this type of scenario, there are distinctions between the types of underwater missions. An example of this is EVA, Turtle I and II and UX1. While all of them are underwater vehicles, they are designed to actuate in different scenarios. EVA is mainly used for inspection and bathymetry of spacious bodies of water, or acting as a support role in a mission. Turtle I and II are deep sea landers, designed for staying in deep sea environment for prolonged amount of time for data gathering. UX1 is designed for operating in underwater mines with tight corridors.

A general underwater scenario was set. It is composed of a large empty 3D space meant to be dynamically filled with assets. In this environment, vehicles like EVA can be setup for bathymetry and assemble the environment. Variants

of this scenario were set to satisfy the requirements imposed by the missions of other underwater vehicles

Underwater mines

In this scenario, spherical AUV UX1 maps the interior of an underwater mine characterized by tight corridors connecting spacious galleries. A way to present the peculiar layout of these locations needed to be developed. In this scenario three minimaps are included for each view of the scene (top, side, back). Each minimap has a simplified presentation of the layout of the mine from each view.

To represent the layout a relationship between the spaces much like a factor graph where the nodes represent the spacious galleries and the edges the tight corridors and intersections was established. A node from the AUV informs whether the space where it is right now is a corridor or a gallery. With this and the displacement of the vehicle we can infer how long the corridor is. For representing this length, three sizes of corridors were established, small, medium and large. The final result disregards real world scale focusing instead in the spatial relation between the map components.

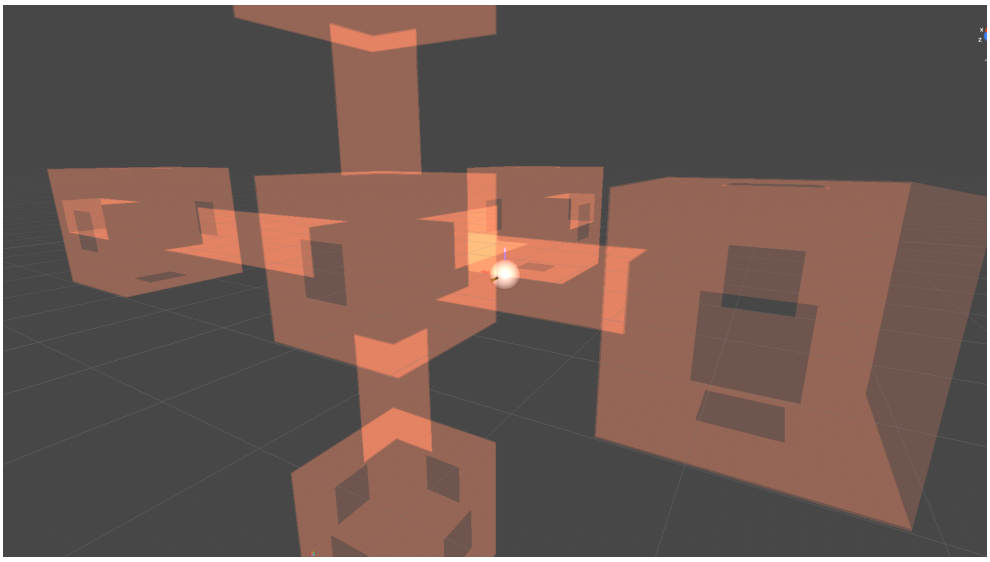


Figure 6.27: Mapping of an underwater mine. A transparent shader is applied to the world mesh for spotting the vehicle.

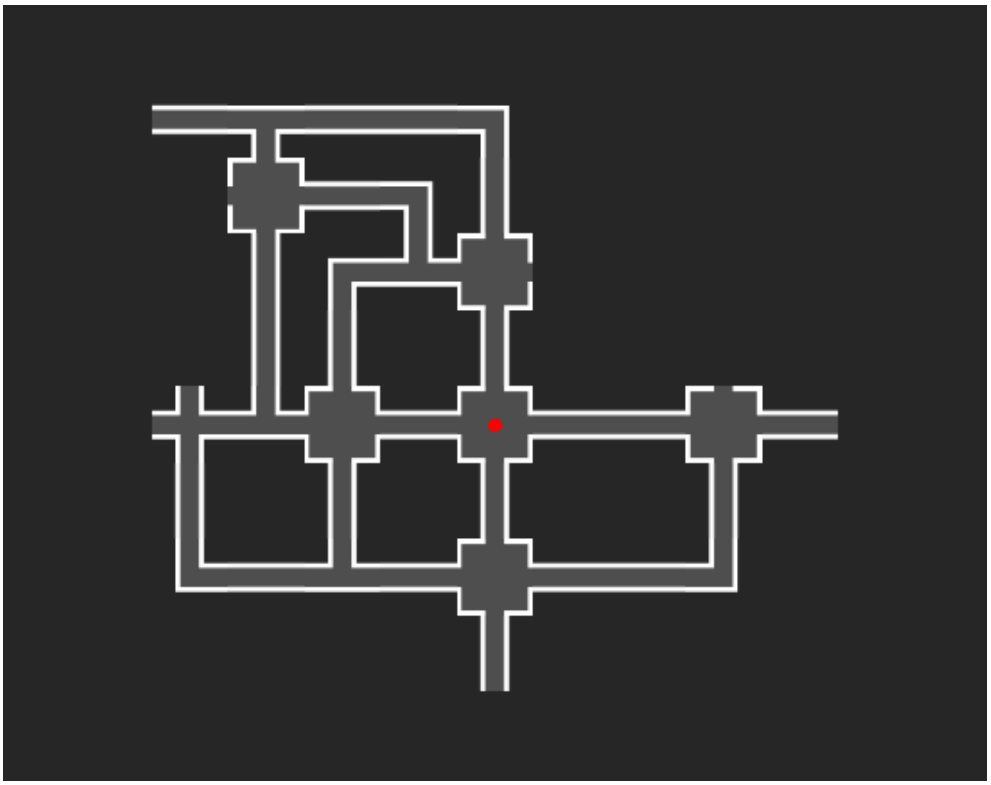


Figure 6.28: Unexmin scene minimap implementation. The red dot shows the vehicle position on the world.

This scenario's minimap is composed by a grid where each cell represents a part of the map. It is meant to show a relation between the world features. With a few image components we can make a 2D map of the underwater mine. An example of this implementation is shown in figure 6.27

Figure 6.29: Components that integrate the Unexmin minimap.

6.11 Odometry

The odometry of a vehicle can be plotted on the screen by using a `Path Renderer` component. This provides a way to visualize the path that a vehicle took during the mission.

Figure 6.30: Vehicle odometry rendering.

6.12 Commands

Issuing commands to vehicles is as simple as publishing a message to the corresponding topic. Commands can take many forms like an array of waypoints, a mode select or an emergency stop. This type of messages benefit from being under a DDS since the QoS options that it provides allow for reliable delivery to the receiving vehicle. It is easy to understand why this is important given that an operator expects the vehicle to obey the given command.

Setting waypoints can be done by using the using the world map or the dedicated panel, setting the latitude and longitude of each waypoint. Another command is issued to start the navigation.

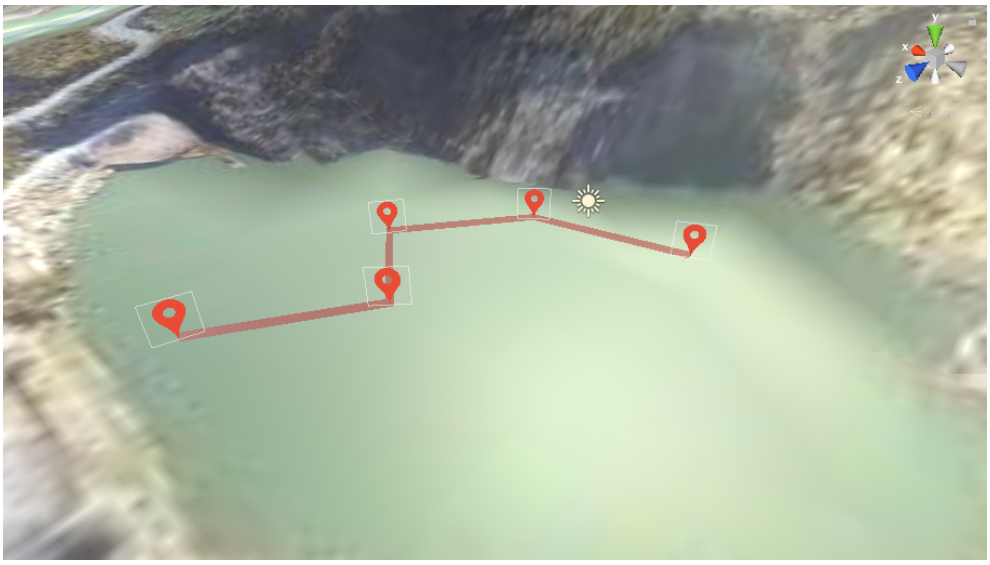


Figure 6.31: Array of waypoints overlaid on the world map.

6.13 Save files

The information regarding a specific worksite or asset is stored locally in the format of a JSON file. This can be used to save application state between sessions. Unity scripting is compatible with .NET's JSON utility classes that simplify the process of serializing and deserializing JSON files, much like how ROS# interprets ROS messages. Future integration with databases was kept in mind when developing this feature by using a standard that is used by many cloud databases.

```
    {
        "id": "fbeda49e-60d8-42b2-9e95-ebef0123882e",
        "latitude": 52.7897610434199,
        "longitude": -8.25671318760975,
        "location": "Kilmore, Tipperary, Ireland",
        "datum": "TM65",
        "name": "Silvermines",
        "assets": [EVA, ROAZII],
        "timestamp": "10/07/2020 15:57:20"
    }
```

Listing 6.1: JSON file describing a worksite

```
{
    "id": "abeva49e-65d8-42b2-3e95-ebtf0123882e",
    "latitude": 52.7897610434199,
    "longitude": -8.25671318760975,
    "posx": 10.432,
    "posy": 0.2321,
    "posz": 28.3342,
    "name": "EVA",
    "sensors": [
        {
            "camL": ["/eva/cameraL/image_raw", "sensor_msgs/Image"],
```

```
            "camR": ["/eva/cameraR/image_raw","sensor_msgs/Image"],
            "IMU": ["/eva/imu","lsa_auv_msgs/Imu"],
            "multibeam": ["/eva/multibeam", "sensor_msgs/PointCloud2"]
        }
    ],
    "waypoints":[],
    "timestamp": "09/07/2020 14:57:20"
}
```

Listing 6.2: JSON file describing an asset

6.14 Build system

Unity is known for being able to build application to various platforms. To build the project, we only need to select the platform where the application will run and specify the scenes that integrate it. Unity then compiles the scripts and packages the assets into proprietary formats for security reasons. This results in a file structure composed of libraries, assets and miscellaneous proprietary files.

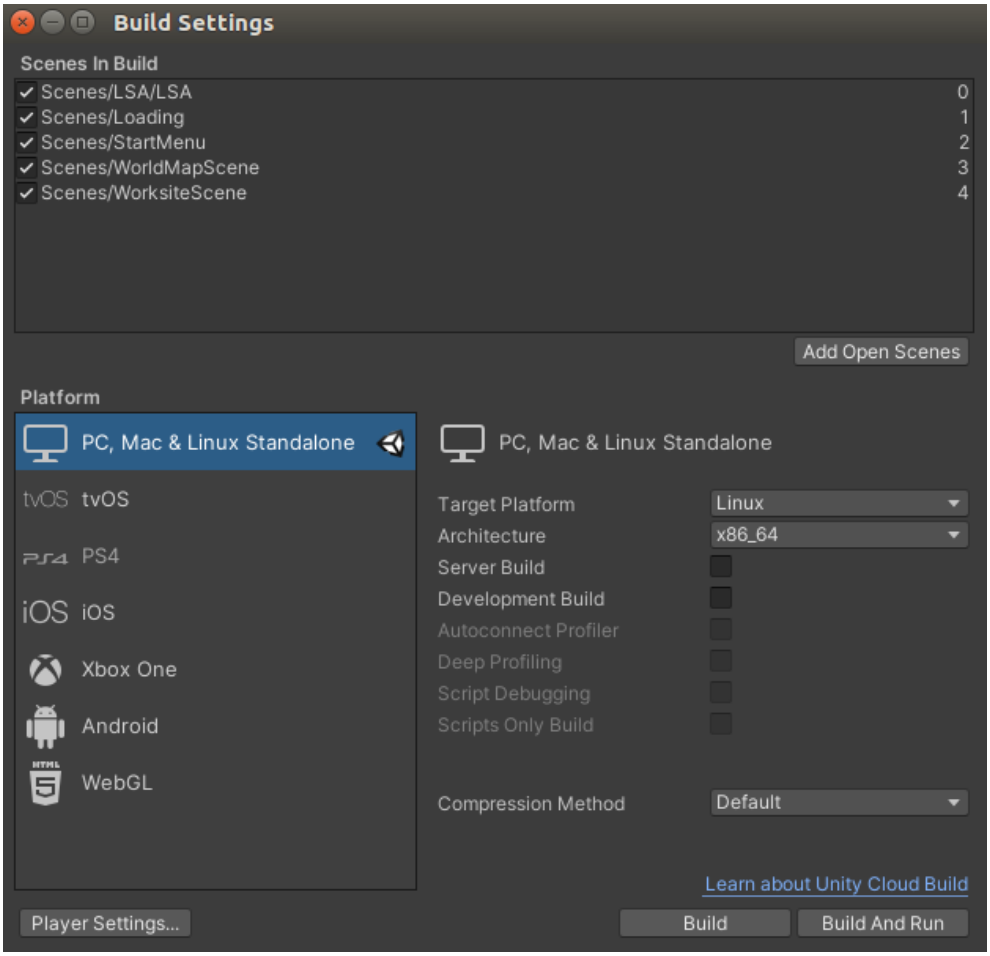


Figure 6.32: Unity's build settings panel.

6.15 Performance

The performance of the system was tested using a laptop with the following specifications:

- Intel Core i7-2670QM @ 2.2 GHz quad core CPU
- 8GB DDR3 RAM
- Intel HD Graphics 3000

Performance of the software, since it is a 3D graphical application can be measured in FPS (frames per second) which hovered at around 100 FPS using one asset in a scenario with the LSA water tank. Generally 3D applications run smoothly at 30 FPS or more so the result can be considered good. The major performance bottleneck is introduced when displaying video streams using the camera preview widget. Even though efforts were made to reduce the load of this feature by using multi threaded rendering. The lack of a dedicated graphics card in the machine translates into an 30-40 FPS drop. CPU and memory usage were not did not pose any performance issues to the execution of the program settling for around 20% CPU usage along with approximately 1 GB (Gigabyte) of memory usage. Latency in video delivery was around 1 second which is not perfect for teleoperation but still inside an acceptable range.

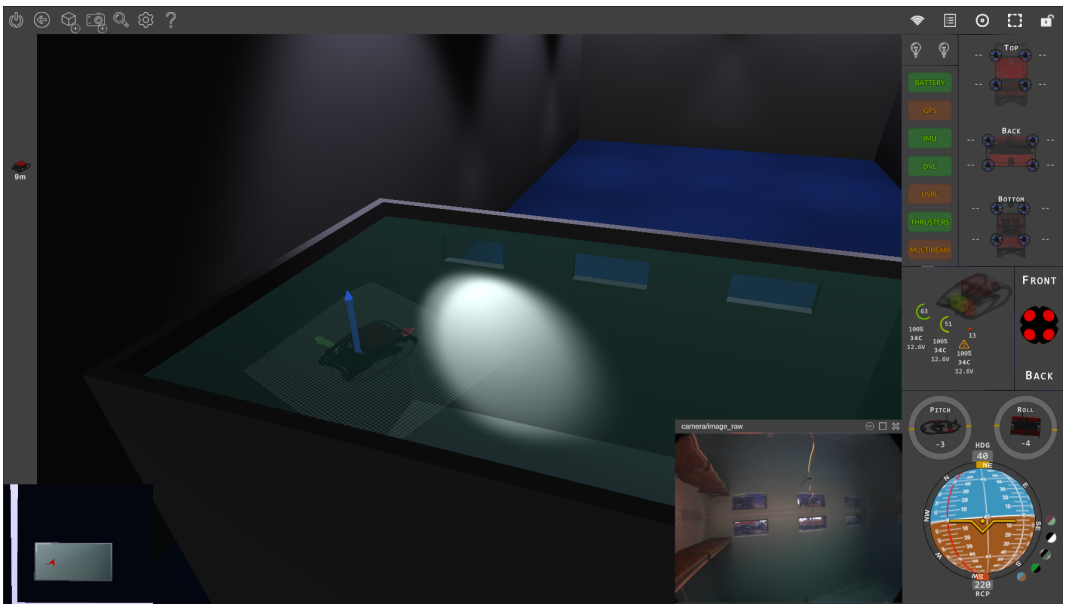


Figure 6.33: Test scene used for performance evaluation.

Chapter 7

Conclusions and future work

In this chapter it is given an overview of the finished product with an insight about the challenges faced during development. The final thoughts about the project are approached and any possible future work.

7.1 Challenges

Live video streaming was by far the biggest challenge to surpass. It is a subject tricky to tackle when a low latency transmission is needed since it is meant to be used for teleoperated vehicles. Large bandwidth requirements demand video compression adding to the complexity of the integration.

Given the real world integration of the software, knowing where the vehicles are located on a map is very important and requires correct representation. Failing to achieve this, the operator could be lead to believe that a vehicle is where it is not. Mapbox SDK simplified this process by providing maps scaled to Unity units and supplied tools to project a real world geodetic location onto a loaded map. Methods applied in geodesy were also implemented eliminating the dependency on Mapbox, allowing the program to calculate vehicle positioning using only its sensors.

Linux integration of the Unity engine is not as developed as the Windows integration. Because of this, some plugins that exist in the open-source domain were only meant to be used on Windows. This posed a problem because this software was made with a Linux integration in mind. This was made possible by compiling these plugins for the Linux platform but meant re engineering some of the code.

7.2 Conclusion

The work described in this document represents a user interface with a variety of different features to aid the human operator in the management of a robotic mission. The potential for this software to grow is endless, since it can be adapted to new requirements desired by the end user. There is always room for improvement in the type of work like this. Technologies in the fields of 3D rendering and game engine development are constantly improving and being created giving this software always space to grow. The work developed tried to integrate as many features as it could in the time slot allocated for its development while trying to fulfill as many requirements imposed by the end user. However given the nature of the work there is and there will always be a margin for improvements with endless possibilities of what can be added. The main objectives proposed were accomplished successfully resulting in an interface that can integrate desirable features when operating semi-autonomous vehicles remotely. Despite this, not all secondary objectives were accomplished, more especially the user authentication, the integration of HID controller support and the logging of missions for later review.

7.3 Future work

The nature of the developed software makes it a good candidate for future work. The implementation of new features is very dependant on the long term use of the program. The wide variety of techniques, algorithms, vehicles types and visual representations of sensors make it impossible to implement every single desired feature. Not only this but the constant improvement of development tools also contribute to the constant need to update this application. In this section are presented some features that could be implemented in order to improve the quality, functionality and flexibility of the program but were not feasible to be integrated into the final result due to time constraints or developer inexperience. The wide array of fields of study that a work of this type includes is a also a weighting factor preventing the implementation of some of these features.

7.3.1 Draggable and resizeable windows

While this feature looks good on paper, it brings a lot of complexity and uncertainty to an UI design. Boundary checking to prevent out of bounds windows is potentially buggy and complex to implement. An interface with static windows is easier to implement and could be made semi-dynamic with collapsible menus and panels.

7.3.2 Web browser

The inclusion of a web browser inside the interface would prove useful in a variety of situations. A sea chart could be implemented by using `OpenSeaMaps` and the video streams could be displayed without the need to use a low level rendering plugin. Despite these advantages, efforts in searching for a way to implement this feature were in vain, given the requirements for the software to run in Linux. At the moment of writing this document, it was not found a simple integration. Every integration found was prepared to work under Windows but not under Linux, which created a steep integration effort for porting it. Because of this, the web browser feature was scraped in favour of other implementations that achieved the same goal.

7.3.3 Amazon cloud services

The information about the work sites where missions are take place is saved locally, in the computer that runs the software. By placing this data in a cloud database, with worldwide data access, we can achieve secure data storage making the data available to multiple computers simultaneously.

7.3.4 Plugin system

Like previously mentioned this software has a lot of margin for improvements. Like many open-source software nowadays, the ability for the community to contribute for its development could result in the creation of many more features that it was first intended to have. A plugin system, giving users the endpoints for developing new features could prove to be an interesting feature to include in this software. Scripting languages like `Lua` or `Python` are nowadays used to mod video games or expand software with the added bonus that the whole software does not need to be recompiled since these languages are interpreted.

7.3.5 Visual fidelity

The graphical quality displayed in the final result is enough for conveying the necessary information to the user. Providing a more immersive environment could result in an increase of user satisfaction (HDRP High Definition Render Pipeline). To use this pipeline effectively the developer needs to have advanced knowledge of graphical design since it requires the creation of more complex assets. Despite being another complex topic to grasp, shaders would upgrade the visual fidelity and expand the way how data is presented to the user.

7.3.6 Bugs

Every software designed by a human programmer runs the risk of encountering bugs, being this application no different. Bugfixes are important tasks that a programmer needs to fulfil in order for the software to run as it was designed to. Various bugs were encountered during the development of this software but debugging is an inherently hard and time consuming task that can hamper the development of one's work. Finding and fixing all the bugs present in a software is an impracticable task and it is expectable that bugs make their way to the final result.

7.3.7 Virtual Reality

Unity's VR (Virtual Reality) tools could make this an interesting addition to this software. Adding this feature is no simple task given its complexity. Expensive hardware requirements also contribute to the lack of interest on this technology. Despite these drawbacks, a good implementation of VR technology in this software would boost the immersive experience of the user. An increase in usability would depend greatly on the quality of the implementation, running the risk of even decreasing the usability of the interface and making it cumbersome.

7.3.8 Web browser

This powerful feature would include a lot of flexibility to the software capabilities. Being able to run HTML (Hypertext Markup Language) would facilitate the addition of features reserved to websites like accessing OpenSeaMaps for free nautical charts or displaying the video streams without the need of creating a low level rendering plugin. Despite the attempts of including a web browser in the interface, this was never successfully achieved mainly because the amount of work that needed to be done for this to work under Linux. This problem was not encountered under Windows because of the wider support for CEF (Chromium Embedded Framework). `Servo` framework's integration was also investigated but without success.

7.3.9 Nautical charts

Including nautical charts in this software was a desirable requirement because of the information that it conveys to the user about the area in a maritime environment. OpenSeaMaps data is free of charge but its acquirement is not easy since it does not provide any API or SDK (Software Development Kit) for integrating into Unity. Offline data can be downloaded as `mbfiles` and processed by Mapbox Studio creating a map layer that is then included in the minimap.

This approach, however, is not free of charge because Mapbox charges for the squared kilometer of raster tiles that contain the nautical chart data.

7.3.10 Documentation

Good documentation is a major advantage in the success of a software. Including a tutorial stage or showing the user tips on how to use the software improves the quality the user experience and teaches how to correctly and efficiently use its tools. However, since the target audience for the developed software is of a technical background, this is not as important as if the target audience was the general public.

7.3.11 Operator training

The integration of input devices like joysticks or game controllers open an array of training possibilities for a vehicle operator. By creating a sandbox environment with various objectives for an operator to complete this application could be used for practicing teleoperation. It would also require accurate physics simulations which is something challenging to accomplish correctly.

References

[1] F. Laamarti, M. Eid, and A. El Saddik, "An overview of serious games," *International Journal of Computer Games Technology*, vol. 2014, 10 2014. [Quoted on p. v, 2]

[2] `https://abyssal.eu/`. Abyssal. [Quoted on p. v, 11, 12]

[3] `https://github.com/siemens/ros-sharp/wiki/Dev_ROSUnityCoordinateSystemConversion`. [Quoted on p. vi, 32]

[4] `https://docs.unity3d.com/Manual/ExecutionOrder.html`. OrderExecutionsEvent. [Quoted on p. vi, 55]

[5] R. Baranyi, R. Willinger, N. Lederer, T. Grechenig, and W. Schramm, "Chances for serious games in rehabilitation of stroke patients on the example of utilizing the wii fit balance board," in *2013 IEEE 2nd International Conference on Serious Games and Applications for Health (SeGAH)*, pp. 1–7, 2013. [Quoted on p. 2]

[6] `https://www.unexmin.eu/the-project/project-overview-2/`. UNEXMIN. [Quoted on p. 4]

[7] E. Silva, A. Martins, J. Almeida, H. Ferreira, A. Valente, M. Camilo, A. Figueiredo, and C. Pinheiro, "Turtle - a robotic autonomous deep sea lander," pp. 1–5, 09 2016. [Quoted on p. 5]

[8] `https://www.vamos-project.eu/`. VAMOS. [Quoted on p. 7]

[9] A. Martins, J. Almeida, C. Almeida, B. Matias, S. Kapusniak, and E. Silva, "Eva a hybrid rov/auv for underwater mining operations support," pp. 1–7, 05 2018. [Quoted on p. 7]

[10] J. Yang, L. Liu, Q. Zhang, and C. Liu, "Research on autonomous navigation control of unmanned ship based on unity3d," in *2019 5th International Con-*

ference on Control, Automation and Robotics (ICCAR), pp. 422–426, April 2019. [Quoted on p. 16]

[11] `https://unity3d.com/machine-learning`. UnityMachineLearning. [Quoted on p. 16]

[12] P. Katara, M. Khanna, H. Nagar, and A. Panaiyappan, "Open source simulator for unmanned underwater vehicles using ros and unity3d," in *2019 IEEE Underwater Technology (UT)*, pp. 1–7, April 2019. [Quoted on p. 17]

[13] J. Almeida, A. Martins, C. Almeida, A. Dias, B. Matias, A. Ferreira, P. Jorge, R. Martins, M. Bleier, A. Nuchter, J. Pidgeon, S. Kapusniak, and E. Silva, "Positioning. navigation and awareness of the !vamos! underwater robotic mining system," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1527–1533, 2018. [Quoted on p. 18]

[14] `http://wiki.ros.org/rviz`. RVIZ. [Quoted on p. 20]

[15] `http://wiki.ros.org/rqt`. ROS rqt. [Quoted on p. 20]

[16] `https://unity.com/`. Unity. [Quoted on p. 23]

[17] `https://www.unrealengine.com/`. Unreal. [Quoted on p. 23]

[18] `https://www.cryengine.com/`. CryEngine. [Quoted on p. 24]

[19] `https://godotengine.org/`. Godot. [Quoted on p. 24]

[20] R. Parasuraman, T. Sheridan, and C. Wickens, "A model for types and levels of human interaction with automation. ieee trans. syst. man cybern. part a syst. hum. 30(3), 286-297," *IEEE transactions on systems, man, and cybernetics. Part A, Systems and humans : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 30, pp. 286–97, 06 2000. [Quoted on p. 28]

[21] G. Cai, B. M. Chen, and T. H. Lee, *Unmanned Rotorcraft Systems*. Springer Publishing Company, Incorporated, 1st ed., 2011. [Quoted on p. 32]

[22] J. Zhu, "Conversion of earth-centered earth-fixed coordinates to geodetic coordinates," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 30, no. 3, pp. 957–961, 1994. [Quoted on p. 34]

[23] K. Hata and S. Savarese, "Cs231a course notes 1: Camera models." [Quoted on p. 40]

[24] `https://www.mono-project.com/docs/about-mono/`. [Quoted on p. 45]

[25] `https://docs.unity3d.com/Manual/NativePluginInterface.html`. lowlevelrenderingPlugin. [Quoted on p. 46, 76]

[26] `https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html`. [Quoted on p. 47]

[27] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," vol. 3, 01 2009. [Quoted on p. 49]

[28] `http://wiki.ros.org/Services`. [Quoted on p. 49]

[29] AdLink, "The dds tutorial," pp. 1 – 6, 06 2009. [Quoted on p. 51]

[30] S. Deodhar, P. Agrawal, and A. Helekar, "Effective use of colors in hmi design," *International Journal of Engineering Research and Applications*, vol. 4, no. 2, pp. 384–387, 2014. [Quoted on p. 57]

[31] E. Hamilton, "Jpeg file interchange format," September 1992. `https://www.w3.org/Graphics/JPEG/itu-t81.pdf`. [Quoted on p. 75]

[32] `https://docs.unity3d.com/ScriptReference/Texture2D.html`. texture2dUnity. [Quoted on p. 76]

[33] `https://docs.unity3d.com/Manual/class-RenderTexture.html`. rendertextureUnity. [Quoted on p. 76]

[34] `https://libjpeg-turbo.org/`. TurboJPEG. [Quoted on p. 76]