

Simulation and Visualisation Software for an Elastic Aircraft for High Altitudes based on Game Engine Technology

Liviu Stamat

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of
Science in Technology.

Helsinki, Finland 07.08.2018

Supervisor

Prof. Arto Visala,
Prof. George Nikolakopoulos

Advisor

Dipl.-Ing. Sven Wlach

Copyright © 2018 Liviu Stamat

Author Liviu Stamat

Title Simulation and Visualisation Software for an Elastic Aircraft for High Altitudes based on Game Engine Technology

Degree programme Erasmus Mundus Space Science and Technology 'Space Master'

Major Space Robotics and Automation **Code of major** ELEC3047

Supervisor Prof. Arto Visala, Prof. George Nikolakopoulos

Advisor Dipl.-Ing. Sven Wlach

Date 07.08.2018 **Number of pages** 72 **Language** English

Abstract

The aim of this thesis work was to design and develop a simulation and visualization platform based on game engine technology, that could be applied to any robotic system and would provide tools for representing the robot, visualizing the environment around it in a high level of detail and also provide means of sampling this environment in order to enable external simulation of interactions between the robot and its surroundings. The main design goal is for the platform to be able to have external physics simulations (robot and robot-environment interactions) entirely separated from the game engine environment. To this end, Unreal Engine 4 (UE4) has been chosen and the platform was implemented as a modular UE4 project, by making use of engine-specific structures. Interfacing between these modules and external ones has been achieved by designing and implementing a middleware interface for the platform, therefore enabling access to the middlewares data transfer system. Finally, this software-in-the-loop chain created between the UE4 modules and the external modules with the middleware as a transfer point has been evaluated in terms of feasibility and functionality by conducting tests on the various modules and interfaces thereof. The outcome is a powerful, flexible and ready-to-use simulation and visualization platform that can be easily adapted to any robotic system and provides the necessary means to accurately sample a customizable, high-quality environment in the vicinity of the robot.

Keywords ELAHA, Flexible, Wing, Aircraft, Unreal, Game, Engine, DLR

Preface

This thesis work is dedicated to my loving parents
Cristina and *Marius Stamat*
for their outstanding support throughout my studies.

Special thanks go to my DLR advisor
Sven Wlach
for his guidance and support throughout this work.

A special word of gratitude to my colleagues
Victor and *Nicolai*.

Helsinki, 07.08.2018

Liviu Stamat

Contents

Abstract	3
Preface	4
Contents	5
Nomenclature	7
1 Introduction	8
1.1 Motivation	8
1.2 ELAHA - ELastic Aircraft for High Altitudes	11
1.3 Task Definition	14
2 State of the Art	15
2.1 Simulation	15
2.1.1 MATLAB	15
2.1.2 V-Rep	15
2.1.3 Flight Simulators	16
2.2 Visualization	17
2.2.1 Simulink 3D Animation	17
2.2.2 LRZ Virtual Reality and Visualization Centre	17
2.3 Game engine technology	17
2.3.1 CryEngine	17
2.3.2 Amazon Lumberyard	18
2.3.3 Unreal Engine 4	18
3 Concept	21
3.1 Requirements	21
3.2 Overview	22
3.3 Modules	24
3.3.1 Robot representation	24
3.3.2 World representation	24
3.3.3 Additional components	25
3.3.4 The Middleware Interface	27
3.3.5 External Modules	27
4 Implementation	28
4.1 Engine-specific terminology	28
4.2 Setting up a UE4 workspace	31
4.3 Software Overview	32
4.4 Platform Modules	34
4.4.1 Robot representation	34
4.4.2 World representation	38
4.4.3 Additional components	39

4.4.4	The Middleware Interface	44
4.4.5	External Modules	48
5	Evaluation	50
5.1	Benchmarking	50
5.2	Frame rates	50
5.3	Scanner component	55
5.4	Collision query accuracy	62
6	Conclusions	65
7	Future Work	67
	References	70

Nomenclature

ROS	Robot Operating System
API	Application Programming Interface
GPL	General Public License
HITL	Hardware-In-The-Loop
SITL	Software-In-The-Loop
FDM	Flight Dynamics Model
GPU	Graphics Processing Unit
PID	Proportional-Integral-Derivative
UAV	Unmanned Aerial Vehicle
IDE	Integrated Development Environment

1 Introduction

1.1 Motivation

To Visualization and Simulation

Robotic systems have undergone a drastic increase in popularity over the last decade. Be it something we encounter in every-day life like a robotic lawn mower, or a much more complex system like a KUKA LWR [1], autonomous drones or Rollin' Justin [2], each one of these systems goes through a complex, multi-stage development process. Throughout most of this process, however, this system only exists in the form of a physical model – which signals the need for simulation and visualization tools required to understand the models behaviour.

Simulation is a vital element in the development of a robotic system. It allows us to monitor and understand what the system is doing in different situations versus how it is expected to behave. Modelling the system from a physics point of view and simulating – for instance – various control strategies applied to different scenarios that system might be implemented in are essential steps in order to ensure a safe, correct and successful development process. Simulation is limited, however, to the robotic system itself; trying to model and interact with the environment around it is quite a different topic and is not always feasible through a numerical simulation, which may leave important questions regarding the behaviour of the system in a real world setting unanswered up to the point where the system can be implemented and tested. This comes with a number of risks since it requires a great deal of effort, time and money, which could be lost in case the system does not behave as expected and its design needs to be adjusted. While this risk may be acceptable for a small and relatively cheap drone, more complex systems – like an aircraft or a satellite – cannot afford it. Therefore, a suitable approach is required in order to simulate the environment the system is expected to function in.

Furthermore, while modelling and simulation are must-have elements and are widely used throughout robotic research and development operations, graphical visualization is useful when aiming to fully understand the behaviour of a highly complex robotic system. This allows one to not only follow the state of the system during the simulation from a numerical point of view, but also to visually track how the system is behaving – in most cases in real-time.

Last but not least, the fact that all robotic systems are unique must be kept in mind. This points to the need of a solution capable of accommodating different robotic systems without too much effort from the user. A modular design is expected to enable this capability by providing predefined interfaces, thus allowing module replacement at any time.

To ELAHA as Underlying Project

ELAHA or the ELastic Aircraft for High Altitudes is a research and development project undertaken at the German Aerospace Agency [22]. The goal of this project is to bring forward a new, highly elastic design for a HALE aircraft. More details with regard to the project's specifics are given in chapter 1.2.

The physical simulation of ELAHA's structure and aerodynamics is the subject of another work being done concurrently [34] with this thesis at the German Aerospace Agency. This invites for a close collaboration with the developer and as such enables an easy approach to interfacing this simulation with the platform developed in the scope of this work. Furthermore, with ELAHA being a highly complex system, this simulation brings together all the required test cases - six degrees of freedom, moving elements (since the aircraft is highly flexible) and the need for a very high refresh frequency of the UE4 components, since an aircraft needs a very fast control loop. Last but not least, while ELAHA does not require interactions with the solid environment, an aircraft is being influenced by air and shifts in atmospheric patterns around it. A separate work being done in parallel to this thesis looks into externally simulating air influences on ELAHA during flight - such as thermals or wind gusts - and visualizing them within this UE4 framework [33]. As such, using ELAHA as a test case towards evaluating the platform enables an easy transition to directly using the outcomes of this work within its design process.

To Game Engine Technology

Considering the progress registered in the areas of computer graphics and GPUs, an attempt to investigate possible applications of game engine technology when it comes to simulation and visualization is only natural: this technology offers an innate solution to high-graphics visualization, since this is one of the core features of video games today, as well as the very interesting possibility of simulating an environment the robotic system can interact with - a virtual world, but derived from real world data such as a satellite map.

Another attractive feature game engines bring into play when it comes to simulating robotic systems is the capability to simulate sensors and payloads. Consider, for example, a camera payload mounted on a rover which is being developed for a mission to Mars; Having the ability to manipulate that camera while simulating the functionality of the rover itself offers the invaluable chance to test the behaviour of the payload in various hypothetical scenarios it could encounter throughout its mission. AirSim, presented in chapter 2.3.3, is a good example of such payload simulation capabilities.

Furthermore, game engines come together with a powerful physics engine. These are used in video games in order to simulate life-like interactions between characters and the game world. Since these physics engines are able to simulate not only the effects of gravity but also support collisions and other various interactions between meshes,

materials, textures and lighting within a world, they offer the basis needed to create a qualitatively accurate simulation of a robotic system. However, if the system being simulated is too complex for these physics engines - for instance a walking robot or a new aircraft design - an external simulation of the robots physics must be used and this comes with the need to investigate the possibility of interfacing such a simulation with the game engine environment by means of data transfer between the two.

Last but not least, a wide and highly-involved community as well as international endeavours such as Citizen Con [23] provide an optimal environment for continuous improvement of game engine technology and as such, the continuous evolution of a simulation and visualization software platform based on such technology.

To the Choice of Game Engine

The State of the Art presents three game engine candidates in which simulation and visualization software could be implemented. Based on the features and pros and cons presented in chapter 2.3, Unreal Engine 4 has been chosen to be the environment in which the platform will take shape, due to a number of figures of merit, including:

- Full Linux support, with modifications possible directly through source code editing,
- A full, comprehensive documentation, kept up-to-date with the latest stable release
- A wide community of users and developers, as well as multiple resources available both towards learning and enhancing/extending new projects (marketplace, forums, etc.) and
- The long-standing tradition of the developers and the franchise.

1.2 ELAHA - ELastic Aircraft for High Altitudes

The underlying project which first brought the need for a more powerful simulation and visualization software than currently available is ELAHA, represented in figure 1. This acronym stands for ELastic Aircraft for High Altitudes and is the name of a current research project aiming to develop a novel design for a stratospheric UAV, or HALE - High Altitude Long Endurance Aircraft.



Figure 1: ELAHA concept [DLR]

The motivation for such designs comes out of more than one consideration. The first and foremost advantage of HALE aircraft comes from the possibilities they bring through their scope of application. Figure 2 shows some of the most important scenarios HALE aircraft can be applied to, for example crisis management, environmental and traffic monitoring, or within communication networks. These applications are enabled by the inherent advantages of HALE aircraft operation as opposed to conventional satellites, such as the ability to change their position without being constrained by a fixed orbit and accessible maintenance options due to them being able to land. Moreover, due to their flight altitude in the stratosphere, HALE aircraft can close the resolution gap currently existing due to the flight altitude difference between conventional aircraft and satellites.

Another advantage of developing such designs is related to the need for a solution to the space debris problem. Out of around 4800 satellites currently in orbit [24], only 1738 were operational as of August 2017 [25]. This not only makes the non-communicative satellites space debris, but also invites to the creation of many more, smaller pieces of debris originating from impacts between these satellites [26]. One of the main reasons for satellites being inoperative in orbit is the lack of means towards maintenance and in-orbit servicing. There are currently a number of on-going research projects focused on countering the space debris problem, such as ESA's e.Deorbit mission [27]. While not becoming a solution to the Space Debris problem itself, HALE (High Altitude Long Endurance) aircraft bring a partial solution to the problem by reducing the need for satellites when it comes to various Earth-centred applications.

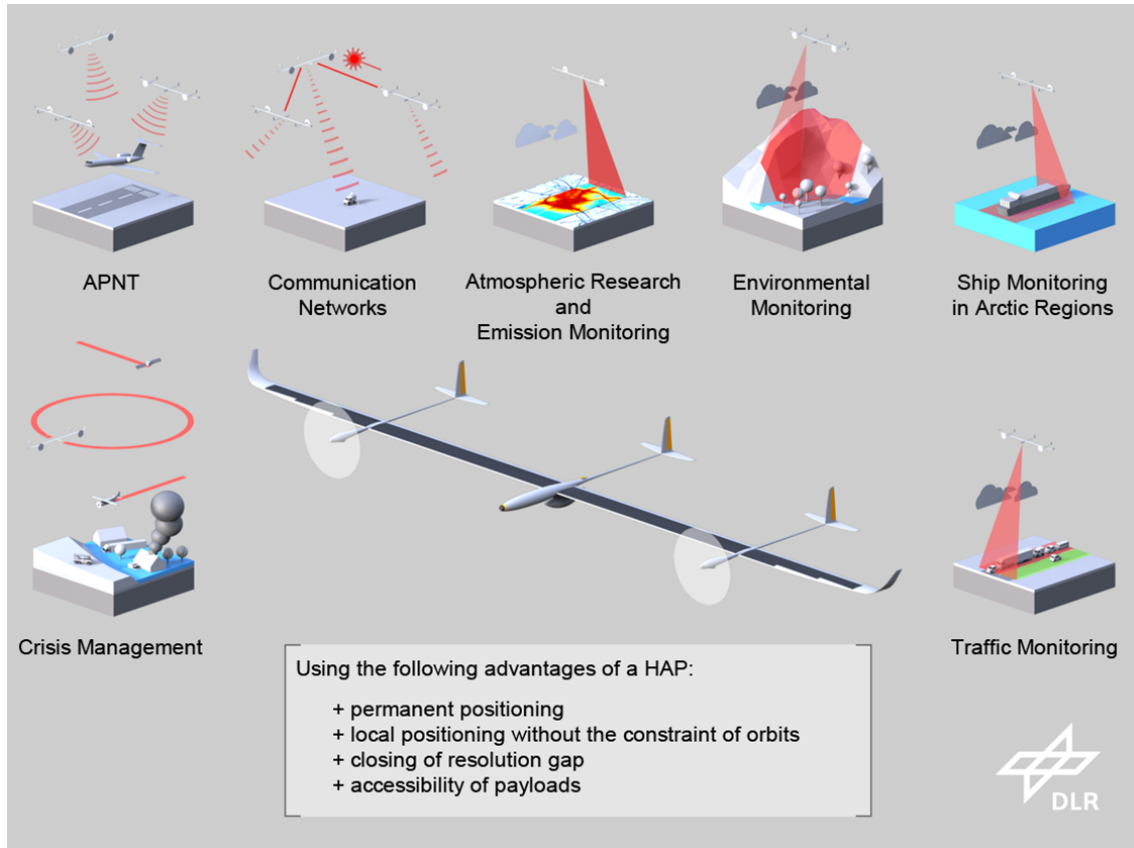


Figure 2: Potential HALE applications [DLR]

However, while advantages to this novel aircraft design include their flexible operation - due to them not being dependent on a fixed orbit - quasi-continuous flight enabled by solar arrays and the possibility to land for maintenance, there are certain disadvantages that they have to overcome in order for them to become an operational reality. A suitable example thereof is NASA's Helios [28] - shown in figure 3a - whose high battery-to-structure weight ratio enabled it to survive a long night period, but also led to a lower measure of structural resistance. Wind influences on HELIOS therefore caused it to undergo a mid-air break-up which led to the aircraft ultimately failing, as shown in figure 3b.

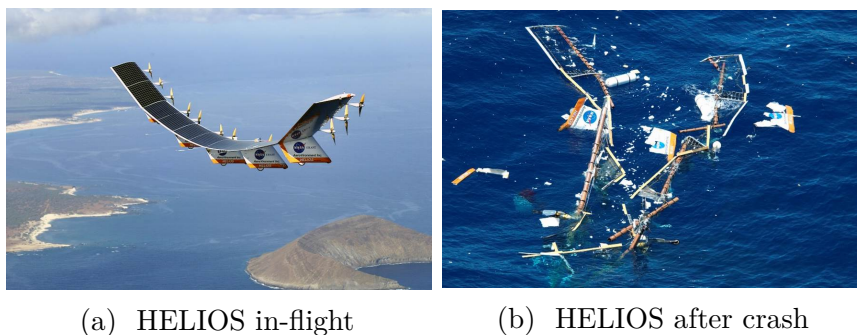


Figure 3: HELIOS [NASA]

ELAHA [29] - or the Elastic Aircraft for High Altitudes - seeks to provide a solution to this inherent structural weakness issues that stiffer HALE aircraft tend to have by employing a highly elastic design. Therefore, the main wing component of this approach should be capable of a 90 degree bend from wing tip to wing tip. This would enable the aircraft to comply with, for example, sudden vertical wind gusts by - put in simple terms - bending around them and sliding out of the way, as shown in figure 4. Another great advantage of a highly-flexible wing design is the capability of adjusting the wing angle in order for the solar panels to catch sunlight at a better angle, thus increasing their output over time.



Figure 4: ELAHA gust compliance [DLR]

Another advantage of ELAHA is its modular design which enables different aircraft dimensions and mass, based on what the payload requires, as shown in figure 5. This also allows for multiple configurations and in-flight separation of various segments.

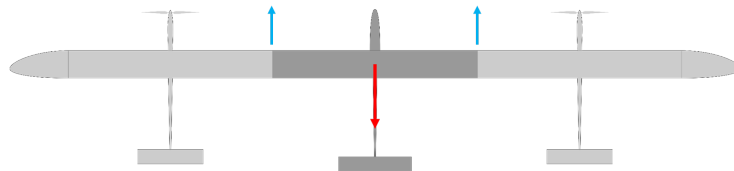


Figure 5: ELAHA segmented design [DLR]

1.3 Task Definition

Following all the reasons discussed in chapter 1.1, as well as linking them to ELAHA, this work focuses on developing a new approach to simulation and visualization that should provide a workable, long-term solution to all of the issues mentioned above. In this regard, a novel system concept based on game engine technology is designed in the form of a Unreal Engine 4 based simulation and visualization platform. This software should therefore be able to provide:

- a fast and reliable solution for real-time visualization enabled by the GPU,
- support for representing different robotic systems, ranging from drones and robot arms to aircraft and satellites,
- an accurate, high-quality representation of the environment around a robotic system along with means of sampling this environment – much like the robot would in a real-world scenario by means of various sensors and
- a means of data transfer to and from the game engine through interfacing with a third-party middleware.

The main design consideration in the scope of this work is to have all physics simulated externally, independent of the UE4 environment. This means the robot physics and the robot-environment interactions will be handled by external modules sending and receiving data to and from the UE4 environment, while completely isolating the UE4 internal physics engine.

2 State of the Art

This chapter presents an overview of simulation and visualization capabilities currently available for different applications, including usage in robotic systems research and development. Therefore, the chapter is split into three parts, covering:

1. Software used towards modelling and simulating robotic systems,
2. Available options for visualization of such simulations, and finally
3. An overview of currently available game engine technologies applicable in the scope of this work.

2.1 Simulation

2.1.1 MATLAB

MATLAB, along with Simulink and its other toolboxes is one of the most widely-used software throughout the research and development process of robotic systems. It provides support for mathematical modelling, model-based simulation and system behaviour monitoring through numerical analysis, among with many other useful functionalities [3].

However, while its simulation capabilities are both powerful and critical to the process, it lacks straight-forward means of simulating and interacting with any form of environment around the robot, as well as a powerful graphical visualization tool, aspect which is presented more thoroughly in chapter 2.2.1.

2.1.2 V-Rep

V-Rep [4] is a powerful simulation tool developed by Coppelia Robotics. It offers support for many different simulation approaches, controller programming using multiple languages and APIs to various environments such as MATLAB/Octave, ROS and others. Another useful feature of V-Rep is its visualization platform. This provides a content browser with multiple types of robots, means of interacting with the environment and scene configuration and manipulation, along with sensors usable towards probing this environment from the robots perspective.

There are, however, also drawbacks to V-Rep. One of them is the low frame rate the visualizer can achieve, thus making the rendering slow and details of the behaviour hard to track. Another downside is the non-open-source character of the software, which hinders in-house modifications when it comes to functionality and types of robotic systems, while creating an external dependency for the development team.

2.1.3 Flight Simulators

Flight simulators are a workable way of visualizing dynamics simulations for various aircraft models. This subsection will discuss the pros and cons for two of these simulators: FlightGear and X-Plane 11.

FlightGear

FlightGear [5] is an open-source flight simulator - available freely under the GPL license - which can be used for both HITL and SITL simulations. It employs three different FDMs based on look-up table approaches to dynamics simulation, but can also be configured to work together with external simulations. This means that FlightGear can be used for both pilot training and in research and development environments with both crewed and unmanned aircraft, making it all the more interesting in addition to the open-source characteristic of the software. Furthermore, it features its own aircraft modelling system which allows users to implement their own machine of choice, even with support for visualizing and user-based interaction with on-board instruments.

While FlightGear is a great way of incorporating new aircraft into a simulation and monitor their behaviour both visually and numerically, one of its main setbacks towards becoming a complete simulation and visualization platform is its scope limitation as a flight simulator. The main idea that FlightGear is being developed on is to be a better and more flexible flight simulator than its commercial competitors, all while remaining open-source [6]. This means that it is only usable in terms of aircraft simulations, with no support for other kinds of robotic systems. Moreover, while the FDMs provide simulation capability when it comes to aircraft models exclusively, FlightGear also lacks support for meaningful interaction with the environment, as well as when it comes to sampling the environment around the aircraft being simulated.

Last but not least, while medium-level graphics make FlightGear approachable for systems with a lower specification rating, they also create a limitation when seeking high-fidelity virtual representations of the environment.

X-Plane 11

X-Plane 11 [7] is a proprietary flight simulator developed and maintained by Lamina Research. In contrast to FlightGear's look-up tables approach, X-Plane makes use of the Blade Element Theory [9], a process that involves discretization of the aircraft into small elements, computing forces acting on each of them and finally converting these forces into accelerations which then provide the velocity and position of each element through integration. Being supplied in a bundle with two other applications - Plane-Maker and Airfoil-Maker - X-Plane 11 makes it easy to implement the physical properties of a new aircraft model [8].

Much like FlightGear, X-Plane 11 remains a flight simulator. This means that other robotic systems are out of its scope and as such are not supported. While providing great graphics and interfaces for controller inputs, it does not support state inputs (velocity, position) from external sources and as such cannot be used together with a fully-external physical model simulation. Moreover, the non-open-source character creates an external dependency for the robotic system development team.

2.2 Visualization

2.2.1 Simulink 3D Animation

Simulink 3D Animation [10] is a collection of applications which provides functionality for linking MATLAB code and Simulink models to 3D objects in a virtual reality. X3D [11] and VRML97 [12] modelling languages are supported for 3D object representations and the world can be animated by means of manipulating object properties during real-time simulation. Furthermore, this collection also comes with included editors and viewers and provides means of sensing events within the virtual world such as collision with the environment and feeding these events back to the simulation code. Last but not least, virtual world interaction by means of devices such as force-feedback joysticks or similar hardware is also supported.

While this is a great visualization tool when used in conjunction with MATLAB and/or Simulink, outside projects such as C++-based models cannot be visualized through it. Furthermore, the graphics quality capabilities of the Simulink 3D Animation collection are relatively low and the achievable frame rates are not outstanding.

2.2.2 LRZ Virtual Reality and Visualization Centre

The Leibniz Supercomputing Centre [13] in Munich, Germany is one of the foremost European computing centres, providing cutting-edge services for scientific, research and academic purposes. Among their facilities, a Powerwall enables post-processed, top-quality results and data visualization for their users. Such functionality is not vital to gathering simulation data but is of utmost importance towards enhancing the understanding of simulation results. One such example is [14], which was put together by means of these visualization capabilities.

The natural drawbacks of this approach are its proprietary nature along with the need for specialized software to visualize data collected from each different simulation, which make it unsuitable for daily use in robotic research and development operations.

2.3 Game engine technology

2.3.1 CryEngine

CryEngine [15] is a real-time game development platform developed and maintained by Crytek. The first iteration of the engine - CryEngine 1 - has been released in

May 2002 and used in the development of the first Far Cry video game title. Today, with the release of CryEngine 5.4 in September 2017, the features offered by Crytek through their game engine have increased substantially, supporting multiple platforms such as Windows PC, XBox and others [16]. Furthermore, CryEngine V follows a new concept of "pay-what-you-want" licensing model through which users are granted access to the engine's source code. In April 2015, Crytek licensed CryEngine to Amazon, fact which later on allowed them to release their own game engine version under the name Amazon Lumberyard.

While the release of CryEngine V bring new features and functionality into play, this new engine iteration is very different to CryEngine 3. This comes with two main drawbacks: a community which is still getting the hang of the new features and development process using CryEngine V, along with an incomplete documentation, as most of it is still based on CryEngine 3 although many features have been deprecated through the release of the new engine iteration. Moreover, CryEngine is developed based on a combination between C++, Lua and C#, which means one requires knowledge of all there languages in order to successfully use certain features of the engine.

2.3.2 Amazon Lumberyard

Amazon Lumberyard [17] is a free Triple-A game engine developed and maintained by Amazon. Lumberyard is currently under Beta status with version Beta 1.14 released in May 2018. The most notable title currently being developed in this game engine is Star Citizen. The engine's source code is freely available to the public via its GitHUB repository, however under proprietary license terms.

Amazon Lumberyard is by all intents and purposes a young game engine. While it brings along a number of interesting features such as integration with Amazon Web Services [18] and GameLift [19] which can greatly complement the development of a new video game, the community is still growing and maturing and its documentation is far from complete. Moreover, the engine is based on a C++-Lua combination - which means knowledge of Lua is required in certain aspects of the development process using Lumberyard.

2.3.3 Unreal Engine 4

Unreal Engine [20] is a game engine developed by Epic Games and first released in May 1998. The title was first showcased in the 1998 first-person shooter Unreal and holds a number of awards such as the Guinness World Records award for "the most successful video game engine". Since then, multiple iterations of the engine have brought forth the second-to-latest stable release, Unreal Engine 4.19, in March 2018.

Unreal Engine 4 is freely accessible to the public, including full access to its source code under a proprietary-type license. A full and comprehensive documentation provides the tools required to take the first steps within its development environment

and the fact that the engine is developed entirely based on C++ offers it unparalleled portability. Last but not least, Unreal Engine 4 is the only game engine to date that provides full Linux support, fact which is a key advantage when the topic turns towards simulating and visualizing robotic systems under a SITL approach.

AirSim

AirSim is an open-source Unreal Engine 4 project developed and maintained by Microsoft, with the purpose of providing a basis for machine learning research in the field of aerial robotics [21]. Recently, this simulator has been complemented with extra functionality in the form of a simulated car, also with the aim of enabling computer vision based machine learning research.

AirSim not only provides a great API which enables easy implementation within a new project of choice, it also comes with the option of controlling a simulated quadcopter by means of an external joystick in the form of a HITL simulation.

3 Concept

This chapter first presents the requirements of the simulation and visualization platform to be developed then follows through by describing the concept of the software, based on the mentioned requirements.

3.1 Requirements

Before the concept of the platform is described, the underlying requirements must be mentioned. To this end, the four top-level requirements stated in chapter 1.3 will be re-iterated below, along with the lower-level requirements derived from them. Therefore, the software shall provide:

1. a fast and reliable solution for real-time visualization enabled by the GPU
 - The software should be able to achieve a runtime step frequency of 100Hz,
 - If this frequency cannot be upheld, the software shall still function as expected at the maximum achievable step frequency, without affecting the simulation;
2. support for visualizing different robotic systems, ranging from drones and robot arms to aircraft and satellites
 - the software must provide an easy way for the user to implement a new type of robotic system,
 - due to the various types of robot systems that might come in play, the software should be developed under a modular and extendible approach,
 - changing the type of robotic system in use should also trigger a cascading change of the functionality of the software and its modules;
3. an accurate, high-quality representation of the environment around a robotic system along with means of probing this environment – much like the robot would in a real-world scenario by means of various sensors
 - The environment the robotic system finds itself in should be easily customizable,
 - Environmental sampling shall provide a meaningful and accurate representation of the environment,
 - Ideally, the software should be capable of employing more than one environmental sensor simultaneously;
4. an efficient means of data transfer to and from the game engine through interfacing with a third-party middleware
 - The middleware interface shall be capable of communicating all required types of information to and from the game engine,
 - The middleware interface shall be able to satisfy the data rates required by the other modules.

3.2 Overview

The requirements described in chapter 3.1 provide a general baseline for the software, which defines the concept and later on a basis for the choices made with regard to the implementation of the platform. From a top-level perspective, an overview of the components and the interfaces thereof are shown in figure 6. It shows both what this thesis work is handling in terms of modules of the SITL, as well as the interactions between all the required modules. The dashed border shows the entire SITL in its expected final configuration. The modules marked by a dotted border are not handled within the scope of this work, however discussing their interfaces within the SITL does, since they must be defined in order for the UE4 environment to understand the information coming in from external modules. The full-line borders mark elements that must be implemented as part of this thesis work. While the external modules are independent, the Visualizer and the Simulator are both implemented within the UE4 environment. This brings along the need for a common UE4 middleware interface, which connects the external modules with the ones running in the UE4 environment - and allows for easier later connectivity of new modules.

Another critical aspect of the platform design is the data flow between all the different modules. This defines the required data interfaces and the expected functionality of the UE4 middleware interface module. The preliminary data flow is presented in the form of arrows in-between modules in figure 6. This shows the most important information that needs to be conveyed between external modules and the UE4 environment, as well as from one internal module to another. The full lines show data that may need *translating* - the possibility of having different coordinate systems and/or units in the various external modules means that data may need to be adapted between modules in order to ensure uniformity. The dashed line marks the internal data transfer between the Robot and World representations. Since both these modules are within the UE4 environment, there is no need for a middleware link between them.

Figure 6 only shows the top-level perspectives of this SITL chain and visualization concept. This is a complex software with multiple inter-dependencies between its modules. To break down the required functionality, each module shall be discussed separately below. As this concept is being implemented, ELAHA, described in chapter 1.2 will be used as an evaluation case. The features of this particular model offer an optimal test case for all parts of the platform, since it requires complete mesh manipulation, fast scanners, a relatively high data rate and a complex model for the internal physics simulation.

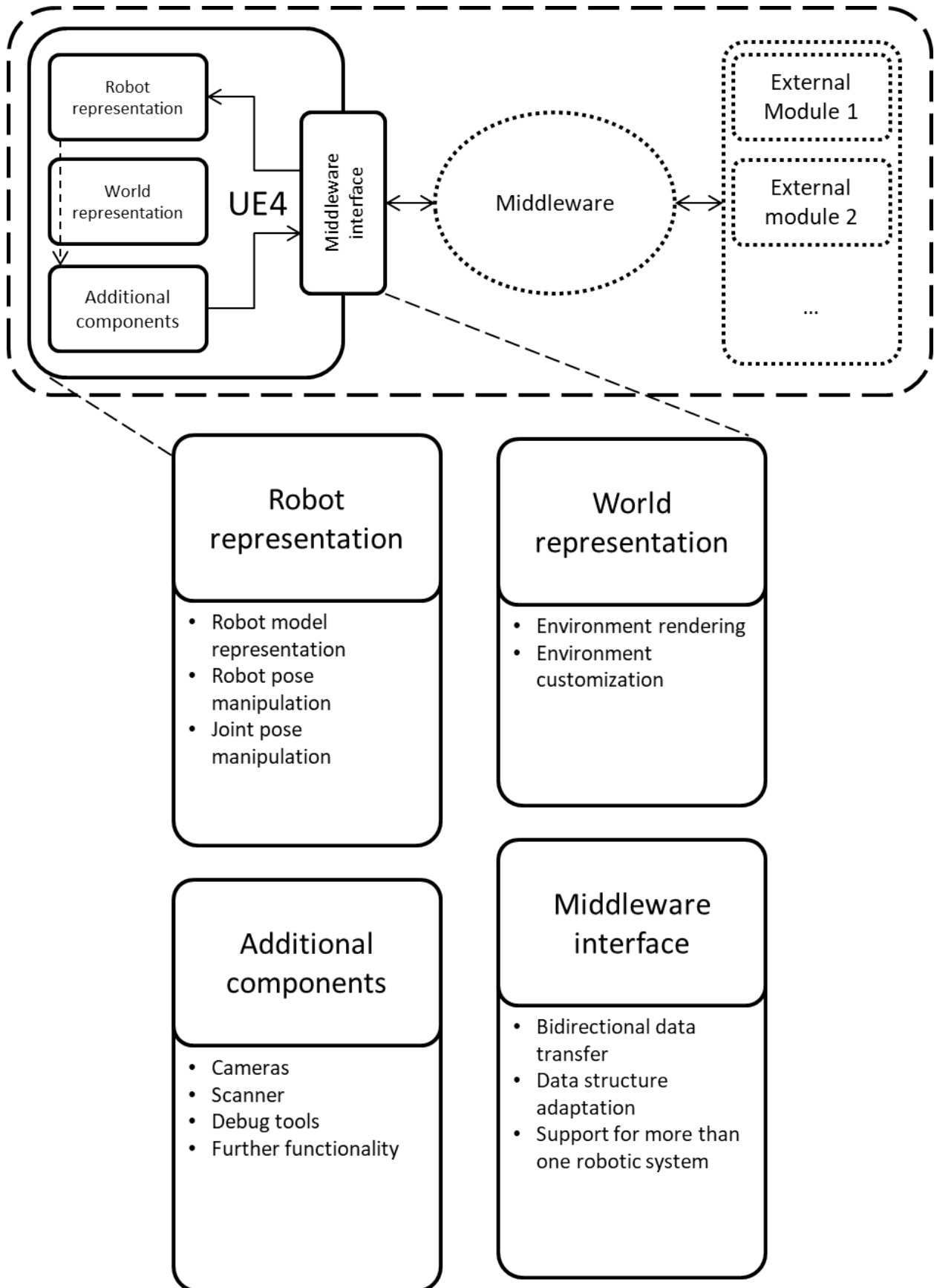


Figure 6: Top-level concept overview

3.3 Modules

3.3.1 Robot representation

The robot representation module is the most straight-forward part of the platform, since it is only meant to offer means for the graphical representation of the robotic systems within the UE4 game engine, as well as support for manipulating position and orientation of the representation along with its appearance. This will most likely be achieved through a custom-designed mesh imported in the engine environment. To this end, UE4 offers a number of different types of mesh objects, which will have to be researched in order to find the one most suitable to fit the needs of the robotic system. The general approach to designing this module is showcased in figure 7.

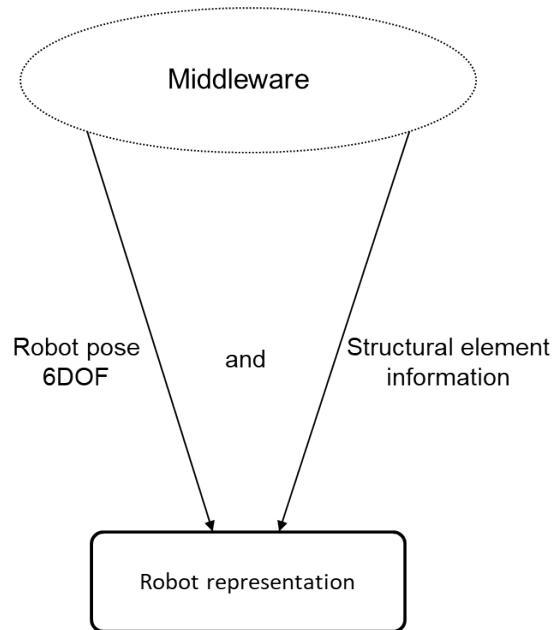


Figure 7: Robot representation concept

3.3.2 World representation

The world representation is one of the modules that UE4 brings novelty elements to. This part of the platform is tasked with providing a high-fidelity representation of the environment the robotic system finds itself in. The world simulation brings the environment into play. This is a representation of the world within the game engine, and is one of the aforementioned elements of novelty, since UE4 as a game engine is very capable of rendering and representing meshes, materials and textures in a very high level of detail. Of course, the relatively trivial drawback here is that the higher the fidelity of this rendering rises, so does the cost of the system specifications required to run the software at a satisfying frame rate. The general approach to designing this module is showcased in figure 8.

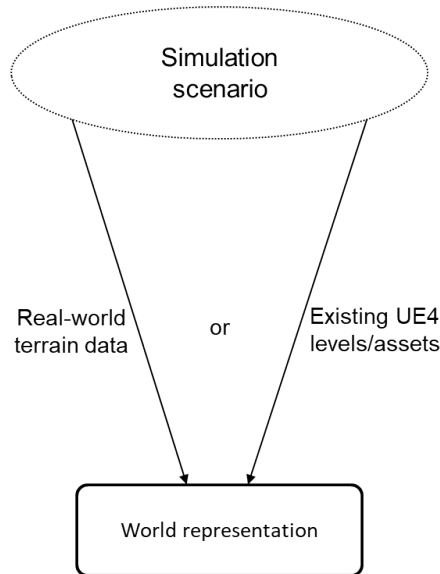


Figure 8: World representation concept

3.3.3 Additional components

Additional components are meant to be other elements of the platform - optional or not - that would provide support towards bridging the gap between the other two modules. A good example thereof are cameras which allow the user to observe the environment but could also be used toward payload simulations of a camera-carrying robotic system. An essential additional component and most important part of this work is the scanner component which is meant to sample the world around the robot and provide useful information with regard to this environment. A general concept of this scanner component is showcased in figure 9.

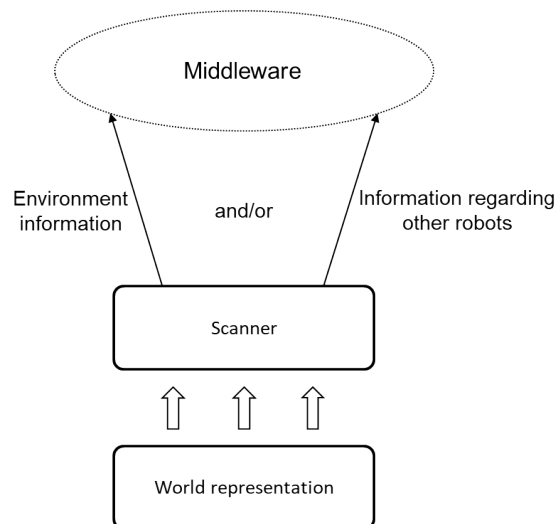


Figure 9: Scanner sampling concept

Scanner component

As mentioned above, the scanner component is the main aspect of this work, due to it being the main source of information regarding the world around the robotic system being simulated. Since the goal of this design is to externally simulate all physics, this scanner must provide valid information which an external physics module could then use to simulate various interactions between the robot and the environment around it, such as collisions. To this end, a generic concept for the scanner component is showcased in figure 10. This concept is based on an ELAHA-type application, namely landing. In order to accurately represent a landing collision between an aircraft's wheels and the ground, information with regard to the ground track of said aircraft is required. This scanner component concept aims to use UE4 functionality such as ray-tracing for a first version, in order to sample the ground beneath ELAHA with the purpose of providing an accurate representation thereof in the form of a 2.5D map. The corresponding implementation is presented in chapter 4.4.3.

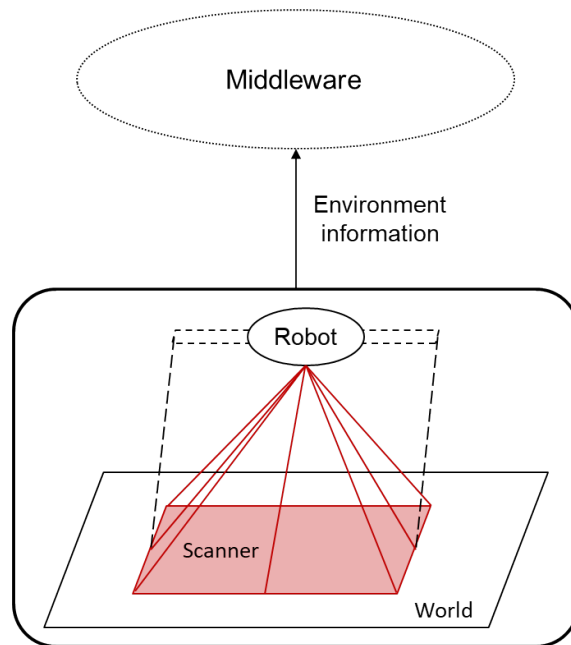


Figure 10: Scanner sampling concept

However, since the simulation and visualization platform discussed in the scope of this work is aimed at being used with any kind of robotic system, this ground probing concept is not sufficient. Therefore, once this version of the scanner component is completed, its functionality must be extended in order to be able to probe the environment all around a robotic system, thus enabling robot-environment interaction in all directions. Chapter 4.4.3 covers the extension of this concept into a new, more complete version of the scanner component, one that uses a similar approach to provide information with regard to the environment all around the robot representation. Last but not least, the scanner component should be optimized by adding customizable parameters such as scanning resolution.

3.3.4 The Middleware Interface

The modular design and functionality of the SITL chain would not be possible without a bridging module to ensure connectivity between all its different elements. In this regard, the middleware must be capable of satisfying the requirements of all the other modules present, both in terms of types of information sent or received, as well as achievable data rates. To this end, this work will attempt to bridge the gap between UE4 and an already existing middleware through an interface programmed and linked into the project code. Keeping extendibility in mind, this interface should be implemented as a module that can later be attached to any UE4 project.

3.3.5 External Modules

UE4 comes with a powerful physics engine, since game engines in general need a great deal of specialized computations in order to display realistic physics. While a video game does not need a physical simulation that is 100% accurate, evaluating the functionality of a robotic system does, since this is a measure of how accurate the simulation is in terms of expectations created for the robot when applied to a real-life scenario. In order to avoid using UE4's physics engine, one would need two critical components: one for simulating the physics of the robot itself – basic dynamics, controllers, etc. – and one for simulating the various interactions between the robotic system and the environment – collisions, atmospheric influences, etc.

As far as the robot physics simulation goes, this is going to be the source of information the robot module uses to manipulate the graphical representation of the robotic system within the game engine. In the scope of this work, the functionality of this external module will be emulated by means of a MATLAB/Simulink model. This model would later on be replaced by - for instance - the structural and dynamics simulation of ELAHA.

Simulating and observing accurate interactions between the modelled robot system and the environment are two of the main reasons for using a game engine approach. This should be achieved by external modules designed to handle these interactions. Of course, different robotic systems and applications require simulating various robot models and as such different interactions with the environment (or between multiple robots), which means various external modules will be needed. This is one of the main reasons for designing a modular platform concept, since it will allow for different modules to be connected to the UE4 platform through the middleware communication interface.

4 Implementation

This chapter first provides details of the specifics of the game engine with regard to the implementation of the simulation and visualization software. The second part covers the detailed structure and functionality of the platform and the decisions taken with regard to its implementation, along with their consequences.

4.1 Engine-specific terminology

Before diving into the structure and details of the platform, some background knowledge of Unreal Engine 4 is required. Gathering this knowledge and getting accustomed to the programming interface and specifics of the UE4 systems was the first part of this thesis. This sub-chapter presents a collection of information regarding UE4 modules of interest in the scope of this work, from engine-specific coding and data structures to game systems, levels and assets.

To blueprints and C++ editing

Blueprints are one way of adding game system logic to a UE4 project. They can only be accessed and edited through the corresponding UE4 editor functionality. Essentially, blueprinting is block-based programming. However, while blueprinting within the editor can be useful for creating world-level game system logic such as AI behaviour algorithms, blueprints do not always offer the desired functionality. UE4 has a database of different blocks, each with their own properties and functionality. While new blueprint blocks can be programmed in C++, accessing and editing C++ code for game systems directly is vital to having complete control and flexibility within the environment. This, however, cannot be done through the editor and must be done by using an external IDE.

To levels and worlds

The first aspect one comes in contact with when opening a new UE4 project is the scene, which happens to be utterly empty. This is because a new project is completely clean and the assets must be added by the developer. There are plenty of basic assets to choose from and experiment with and a great deal more available in the UE4 Marketplace. However what is of interest here is what this scene means. In UE4, this is a level. One game can have multiple levels and each of them can be visually and functionally different. For instance, let's say we want to simulate the functionality of an autonomous UAV in different scenarios. Each of these can be implemented within the same game in the form of levels, with the option to change between them through either a user menu or hard-coded game logic.

The world is an interface one uses to access information about a level when it comes to developing game logic. In a general way, this is the virtual representation of the level in the understanding of the game engine. It contains references and information with regard to all the objects in that level and provides the developer

with support functions used in spawning actors, generating debug elements and much more. In other words, the level and the world are two sides of the same coin, the former being a visual perspective and the latter providing access to the underlying information.

To primitives, actors and pawns

There are many types of objects used when developing a UE4 project. The most common ones are primitives. As per their name, these are basic shapes that come with incorporated support towards physics simulations, be that a falling object or an obstacle in the path of an actor, producing a collision upon overlapping positions. These can also be used as static objects in the level with no physics or colliders attached, thus becoming interesting as evaluation cases for environment probing coupled with external physics simulations.

Actors are a vital part to a UE4 project, even more so in this case since the objective of this work is developing simulation and visualization software. Actors are generic engine structures that can be manipulated and interacted with. They create the backbone of this implementation, since they are controllable and can have their own logic hard-coded within their system. This is particularly interesting later on when discussing meshes and bones, as well as interactions between multiple entities. Pawns are a type of actor designed to receive input signals. This allows them to be controlled directly by means of a keyboard and/or a mouse, thus making them ideal candidates towards carrying logic for the first robotic system representation within the engine environment.

To meshes and bones

Meshes are critical to developing not only the visualization aspect but also the simulation part of the software. While primitives have their own meshes (i.e. a sphere primitive has a spherical mesh), custom meshes are needed when it comes to robotic systems, namely one mesh per robot type simulated. These can be created using 3D modelling software such as Blender, AUTODESK Inventor or others and UE4 provides support for importing them into the game environment as assets. The most important aspects of these meshes are its triangle count and bone definitions. Triangulation transforms the mesh into a collection of triangles. The triangle count of a mesh directly impacts the performance of the software, since a higher triangle count requires more graphical computation from the GPU.

Bone definition is the process of marking the mobile components of the mesh. For instance, the bones of a Kuka Robot Arm would be the elements between its joints, thus allowing each element to move independently with respect to the rest of the mesh. This simulates the arms joints within the engine environment and allows them to be manipulated individually. Since ELAHA's design features flexible wings, these will also be discretized into bone elements in order to simulate and visualize their high mobility. The number of bone elements the wings will be split into will

be decided with respect to the UAV's physical simulation. Of course, the higher the bone element count, the smoother the wings movement will look like within the engine environment.

To ray-tracing and ray-scanning

Ray-tracing is one of the most widely-used functionalities of game engines. This fires a virtual ray from one point to another in the world, detecting one or more collisions along its path. While this is a very useful functionality when it comes to detecting collisions on trajectories of small sphere-shaped objects, it can also be implemented to simulate a LIDAR-type sensor. However, in the case of scanning the environment around - for instance - ELAHA, one ray will most likely not suffice. An interesting approach involving ray-tracing, however, was presented at Citizen Con [23] which might be applicable in this scenario as well. This approach - also known as ray-scanning - has been presented at Citizen Con as a means of probing the surface where a humanoid character is about to step, in order to allow very accurate, procedurally-generated movement animations. While this approach could be used when simulating a walking robot such as Toro [30], a variation of the same idea might be applied in a more general environment-probing case as well.

To the UE4 Property System

Unreal Engine 4 comes with a so-called property system, which allows one to design properties for every object programmed into a UE4 project. These properties can be variables of any data type that have a default value hard-coded in the objects code. The user can then choose whether to make this property visible to the editor. This shows the property in the UE4 editor sidebar as a changeable value. The great advantage of using this system is runtime experimentation - the properties always keep their hard-coded values upon game start, but these values can then be changed during game-play towards runtime testing. Any object variable - from a simple boolean to a Skeletal Mesh can be set to be a visible property as shown in figure 11. The *EditAnywhere* flag is one of the flags that publishes the property to the UE4 editor.

```
UPROPERTY(EditAnywhere, Category = "Booleans")
    bool isCameraControlEnabled = false;
```

Figure 11: UE4 Property coding

To data structures

Unreal Engine 4 features its own collection of data structures, from engine-specific ones such as actors, actor components and cameras to simple mathematical constructs like three-element vectors. These data structures will naturally be used towards realizing the in-engine functionality, however external structures will be used to realize the middleware connectivity. Therefore, interfaces between internal and external data structures are necessary.

4.2 Setting up a UE4 workspace

This sub-chapter presents the necessary first steps which must be done in order to set up an UE4 environment suitable to the software discussed within this work. The first main step is cloning the UE4 source code from its git repository into a local directory. Before building the engine, however, a couple of minor changes must be done to the UBT source code, namely to the hard-coded compiler flags. To this end, one must browse to the the engine source code and look for the *LinuxToolChain.cs* file. Once found, two lines of code need to be changed. The first of the two is found on **line 479**, where the two compiler flags *-Wall* and *-Werror* must be commented out. The second change must be done on **line 596** where the *-Wshadow* compiler flag must also be commented out. These two changes are required in order for a UE4 project to build with the Links and Nodes headers. Once this is done, the engine must be built as per the standard instruction provided by Epic Games. To ease this process, one can open, configure and build the UE4 source code using Qt Creator, as presented in the UE4 documentation [31]. Once the engine is built and a project is created, there is one last change that needs to be done within the new projects build system, namely in the *ProjectName.Target.cs* file found in the Source directory in the project path. In this file, the line **bForceEnableExceptions = true;** must be added to the body of the class. This change is not required while using the editor with the project, however it is needed when wanting to launch or cook the project as a standalone application.

The next vital steps to perform before starting an UE4 project is access to the source code and an IDE. This set-up process is OS-specific and should be handled with care. Under Windows, access to the source code of a project is gained through the use of Microsoft Visual Studio. Creating a C++ project through the UE4 Project Creator also creates the needed file structure and gives one the ability of starting Visual Studio from within the UE4 editor. Alternatively, there is a *.sln* file inside of the project directory through which one can access and edit project source code. The C++ code can be compiled through Visual Studio's build button or the compile tab found on the top toolbar in the UE4 editor.

When it comes to Linux, this process becomes a little trickier, however once configured, code editing and building becomes straight-forward. In order to first gain access to the game engine, one needs to clone the UE4 repository and build the engine. This can take anywhere between 30 minutes and over an hour depending on the system specifications. Once this is done, a new C++ project needs to be created through UE4's Project Creator and the easiest way to access the code is through QT Creator. Configuring this IDE to work with UE4's build system is a one-time process per project but is fairly simple. Information on this matter can be found in the UE4 documentation [31]. One last important thing to note is the correct way of building game code. Using the QT Creator build button attempts to re-compile the engine and has no effect on the game code. The only way one should attempt to build game code is through the compile tab found on the top toolbar in the UE4 editor.

Finally, the middleware library - Links and Nodes - must be prepared for UE4. This process involves both editing the UE4 project *C#* files and making changes to the Links and Nodes build properties, such as the toolchain being used for compiling the library object. This process is argued and explained thoroughly in chapter 4.4.4.

4.3 Software Overview

This sub-chapter provides a top-level perspective of the implementation structure. As stated at a concept level 3.2, the simulation and visualization platform developed in the scope of this work takes the form of a Software-in-the-Loop simulation. An overview of the modules within this loop is shown in figure 12. This shows an updated concept diagram, which now presents where every modules fits in, along with its implemented functionality.

Following the diagram legend used in before, each type of line has a meaning. The long-dashed border envelops the entire SITL environment within which the UE4 modules should be used. The dotted borders represent both the external module space and each of the three external modules interfaced with and used throughout this work. Furthermore, the dotted line bordering the Links and Nodes module signifies that the middleware used within the scope of this work is an already existing module, while the interface between it and UE4 has been implemented as part of this work.

Last but not least, the full line borders envelop both the entire UE4 environment, as well as the individual UE4 modules implemented as part of the platform developed throughout this work. The LN Wrapper module is the aforementioned middleware interface written for a UE4 project and as such overlaps with the border of the environment. The three internal modules - the Pawn, the Custom Level and the Cameras and Scanner components - represent each part of the required simulation and visualization platform functionality - the Robot representation, the World representation and the Additional Components modules respectively.

In order for a clear explanation of the implementation, each module presented and its interfaces will be discussed separately below.

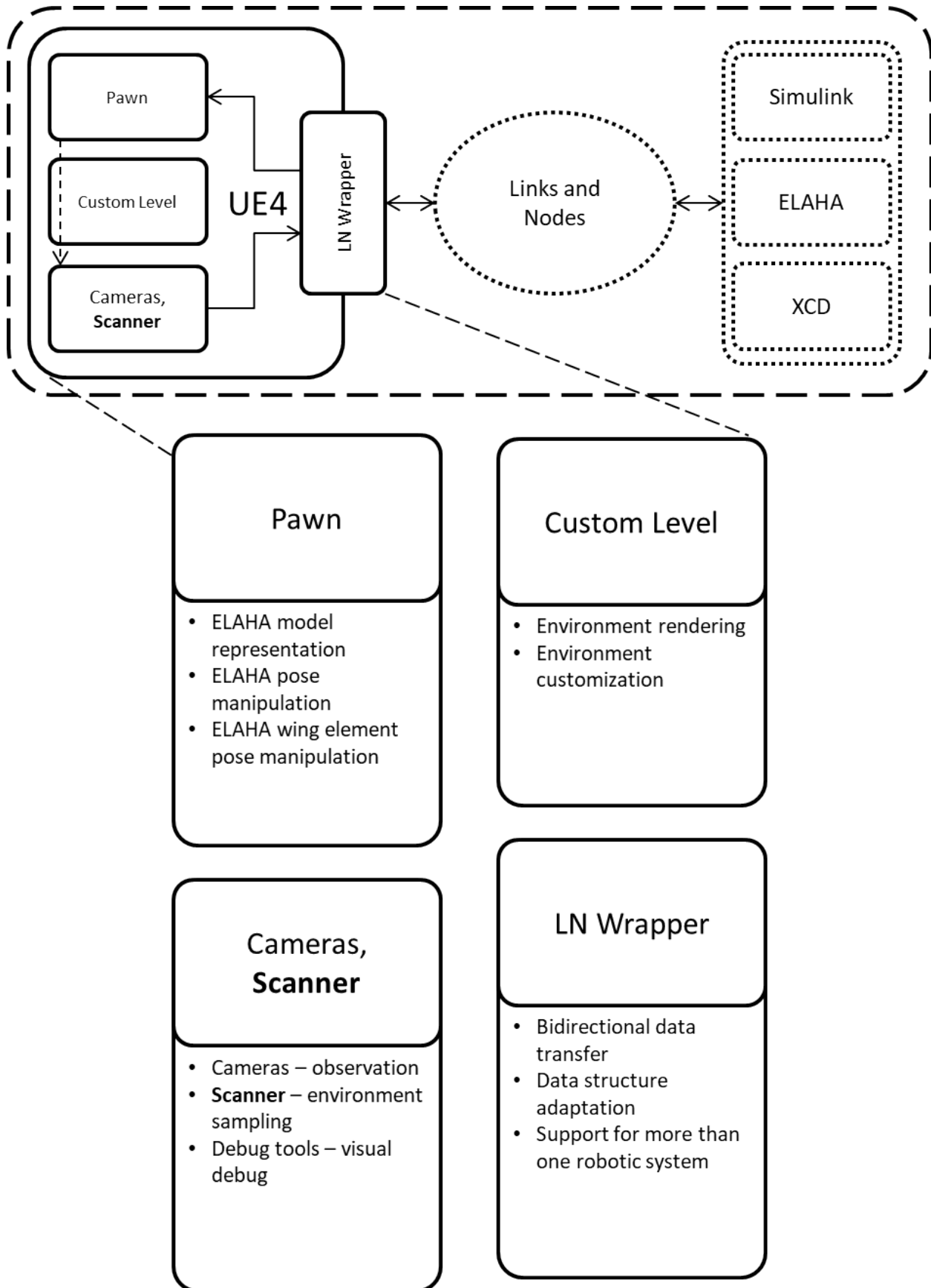


Figure 12: Top-level implementation overview

4.4 Platform Modules

Following the software structure discussed in 3.3, the Robot and World modules have been implemented. In the scope of this work, implementing the Middleware module refers to realizing the interface between UE4 and an already existing middleware. Furthermore, the external modules are based on other works and will be discussed as such, with this thesis showcasing the interfaces required for the inclusion of such modules in order to achieve the expected SITL functionality.

4.4.1 Robot representation

As stated in the concept phase, the Robot representation is the first of three main modules implemented within the UE4 environment. This module provides functionality purely for representing and manipulating a robotic system within the game engine. To this end, a choice has been made to implement the Robot module as a Pawn.

The pre-defined functionality of the Pawn - a Begin-Play function (ran once, when the module is started) along with a Tick function (ran every frame) - offer the optimal environment for the Robot module as well as a handy way of monitoring its refresh frequency - since the Robot updates every Tick, which is in turn called every frame. Keeping simplicity and ease of development in mind, this custom Pawn is entirely C++-based, thus avoiding confusion between hard-coded settings and editor-editable ones. To this end, the Property system that UE4 provides allows for publishing hard-coded settings to the editor so that the user can experiment with changing these properties during runtime without having to worry about keeping track of or reverting changes.

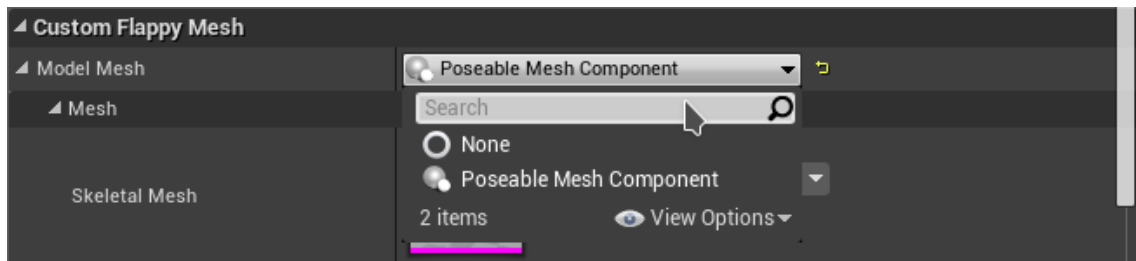
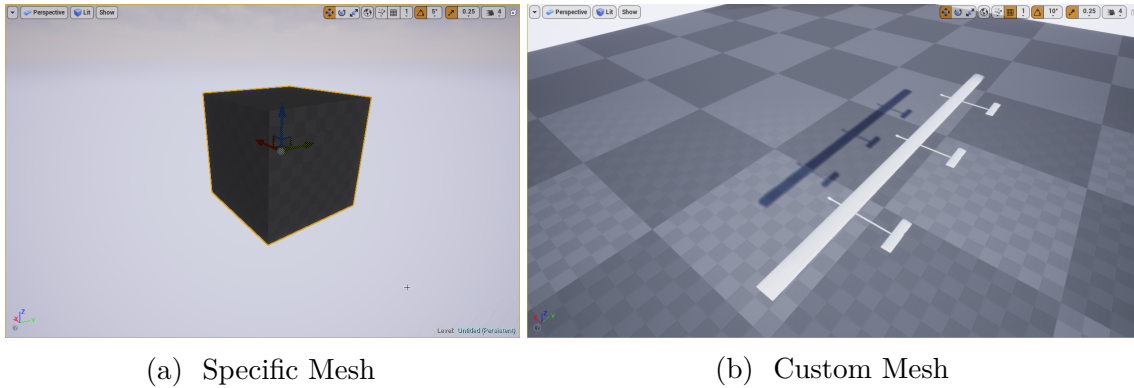


Figure 13: *ModelMesh* property in editor view

Once the custom Pawn class has been created, an engine-specific mesh or a custom one can be assigned to it. Figure 13 shows a *ModelMesh* property where a new mesh can be set for the Pawn in question. Figures 14a and 14b exemplify both cases respectively, where the basic mesh is a simple cube and the custom mesh is a Blender-designed one based on ELAHA.



(a) Specific Mesh

(b) Custom Mesh

Figure 14: Meshes in UE4 editor view

While the custom mesh represents the robotic system in question - here, ELAHA - The next required step is ensuring functionality with regard to manipulating its six DOF. To this end, four sub-steps are taken. First, a keyboard input approach is investigated. This entails two keys per degree of freedom. An advantage to this approach is the very accessible Input System that UE4 provides through a Pawn: the keyboard input must be assigned to a unique name within the editor, which will then be linked through to a function in the Pawn code using a set of flags marking the keys moment-of-effect (i.e. Pressed, Released, etc.) and this functions must then be defined. Figures 15, 16a and 16b show these three steps respectively.

```
PlayerInputComponent->BindAction("ToggleCamera", IE_Pressed, this, &AFlappyWing::OnToggleCamera);
PlayerInputComponent->BindAction("ToggleCameraControl", IE_Pressed, this, &AFlappyWing::OnToggleCameraControl);
```

Figure 15: Key link



(a) Key editor

(b) Key code

Figure 16: Keyboard input set-up

While this approach is fast and easy to implement, it may prove to be somewhat counter-intuitive in the scope of controlling an aircraft. As a solution, a joystick-based input approach is implemented. This provides an intuitive way for controlling the

aircraft towards simple testing - without the need for external physics modules - while also bypassing the need for in-editor settings, since the keyboard input is no longer needed. The accompanying downside to this approach is the need to implement a C++ wrapper for the joystick functionality which is also dependant on the system architecture - which means different drivers must be used under Windows and Linux respectively. In the scope of this work a third-party joystick driver [32] has been used. While UE4 polls for keyboard inputs every frame on its own, the joystick input must be polled manually through the Tick function.

While user input is essential throughout the development phase and to a complete set of robotic testing systems, ELAHA is meant to be autonomous. The next two approaches to 6DOF control are based on external input - the former based on MATLAB/Simulink and the latter on ELAHA's own structural and aerodynamics simulation. This is made possible through either direct shared memory access, or a middleware. Chapters 4.4.4 and 4.4.5 discuss the implementation of a middleware interface within the UE4 environment and the software modules that provide external input to the Robot module respectively.

With preliminary input components now set up for the main mesh, focus turns towards joint manipulation. In the context of UE4, joint are named bones. There are thus, three main types of meshes: static, skeletal and poseable. Static meshes are those used and illustrated above. Skeletal meshes are those that also contain bone information. In ELAHA's case, a skeletal mesh doesn't contain defined joints, however the wing itself can be discretized in wing elements, which can then be considered bones. The trade-off here stands as follows: more bones mean more changes in a rendering per frame, but also provide a smoother transition at bone link points. In ELAHA's current 3D iteration [33] (the mesh shown above), the wing itself has been split into 22 bones and the elevators also define one individual bone each. Since this custom mesh is imported via a .fbx file exported from Blender, UE4 receives the Blender-designed bone hierarchy. Thus UE4's interpretation of the ELAHA skeletal mesh is represented in figure 17.

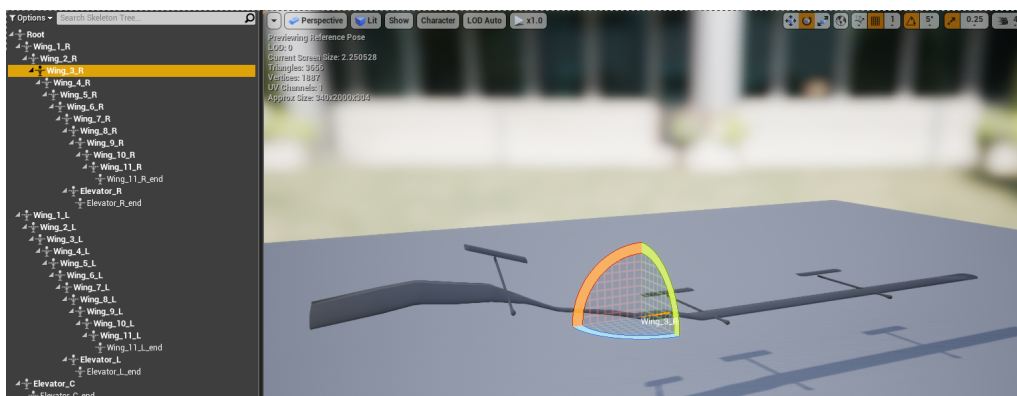


Figure 17: UE4 overview of ELAHA bones

The bone names showcased above must now be assigned to indexes. UE4 can access information on them by using either names or indexes, however indexing them makes procedural manipulation possible, thus enabling easy bone access through a pre-defined name-to-index structure. To this end and in order to avoid confusion, the root bone - which in this case overlaps with the root of the mesh - is indexed at 0, the right wing elements are indexed one through ten, the left wing bones eleven through twenty and the elevators are left as the last three. Since the coordinate frames for bones on the left wing are rotated from those on the right wing, this indexing structure makes most sense in the context of procedurally manipulating bone elements throughout a simulation.

One thing that must be kept in mind is that the bone hierarchy shown in figure 17 also hints on how manipulating the pose of one bone propagates through to the others - this follows a parent-child relationship, which means moving or rotating a bone, applies the same relative transformation to all the bones along that child branch.

This bone hierarchy also enables easier manipulation of the individual poses, since each child bone moves when a parent is moved. Therefore, the only requirement here is to set bone poses within the Visualizer by always iterating through them from root to wing tip. Since the elevators are individual, they can simply follow this approach with no consequences. Therefore, once one understands the C++-based bone accessing, the coding logic becomes relatively straight-forward: retrieve bone transformation matrix, change the rotation part (since there is no need for bone translation) and reapply this new transform to the bone in question. Within UE4, this logic may be implemented inside of a function loop as shown in the caption below.

```
FTransform bonetr;
if ((index > 0) && (index < 11))
{
    bonetr = this->ModelMesh
                ->GetBoneTransformByName(BoneNames[index],
                EBoneSpaces::ComponentSpace);
    bonetr.SetRotation(bonetr.GetRotation() *
                FQuat::MakeFromEuler(FVector(0.f, 0.f, 0.f)));
    this->ModelMesh->SetBoneTransformByName(BoneNames[index],
                bonetr,
                EBoneSpaces::ComponentSpace);
}
```

The captioned code had no visual effect on the bones since it simply rotates their orientations by 0deg on every axis, but the 0 values may be replaced with - for instance - joystick input values or angles/angular rates coming from a simulation. Since the Visualizer must be kept up-to-date with the externally simulated model, this procedure should be ran every frame (every iteration of the Tick function) to ensure smooth transitions between steps and states.

4.4.2 World representation

Since the World is the greatest element of novelty that UE4 brings into play in the context of robotic simulation and visualization, there are multiple aspects that need to be investigated. The first step is avoiding the process of creating a new map from ground up by finding a suitable environment in the UE4 Marketplace. Since the Marketplace is not available under Linux, this environment needs to be downloaded on a Windows machine and later on imported into the Content folder of the current project under Linux. In order for this environment to fit the needs of this work, it must come bundled with an example level. To this end, Epic Games's Infinity Blade content is used since the available test level fully fits the needs of this project - it has multiple elements of complexity such as water, different elevations, steps and foliage. However, this level has been designed for a video game, which means it has a large number of elements not needed in the current scope, such as bounding volumes, various lighting effects and others. Deleting those will help improve performance of the software. This process already provides a high-quality world suitable for experimentation.

The next preliminary step with regard to the World is saving and building the map in order to avoid repeating this process. Saving it ensures availability of the current design of the map - lower object count, higher performance - for future use in this or other projects. Building the map refers to allowing the engine to compile the maps shaders and lighting. This prepares the map for active play and thus saves time in the testing phase, since the building process can take some time, depending on object count and material and texture complexity. The initial built version of the map is shown in figure 18 and 19.

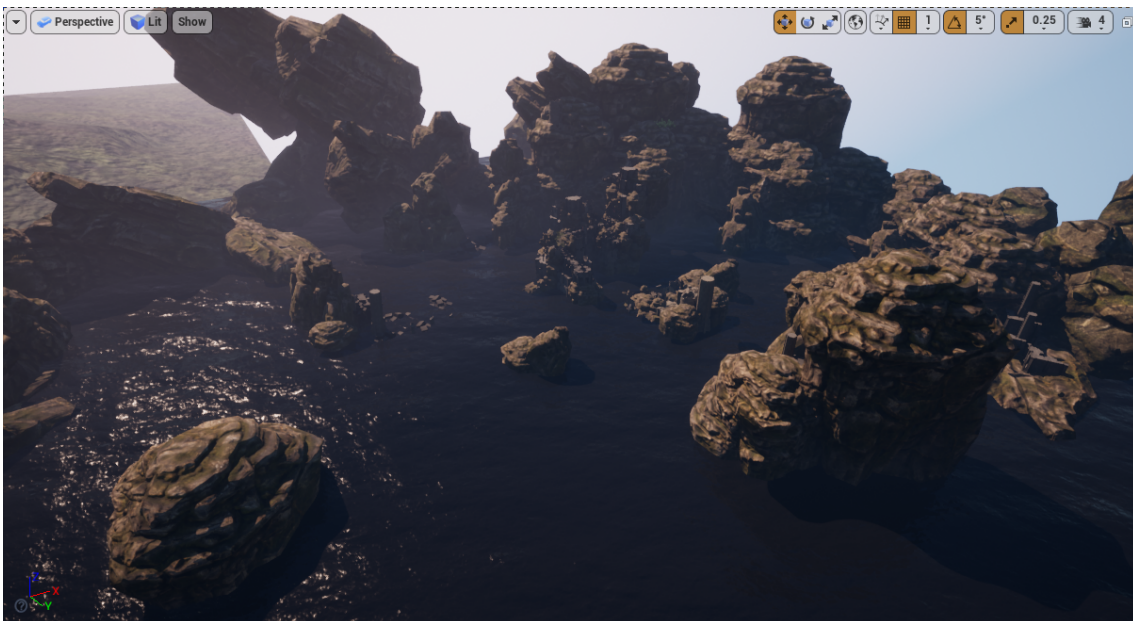


Figure 18: Built world



Figure 19: Near-far views in the custom world

One last important aspect in preparing the world is making sure the meshes within it are visible to ray-tracing. This must be done due to the chosen scanner concept, since rays - as mentioned before - are being cast in a certain channel. Therefore, all actors in the world must be assigned to the same channel as the one used for ray-tracing. In the scope of this work, the used channel will be *WorldStatic*. Last but not least, the meshes themselves need to be made visible. UE4 materials have an option to enable/disable collisions at material level for each individual mesh. This saves memory requirements during gameplay, but also makes those specific meshes invisible to rays. Therefore, all meshes used in the custom world need to be checked to ensure that collisions at material level are enabled. This behaviour is showcased in figure 20 by means of an example where the right-hand statue is invisible to rays in figure 20a but made visible by means of material-level collisions in figure 20b.



(a) Collisions disabled

(b) Collisions enabled

Figure 20: Mesh preparation for scanning concept

4.4.3 Additional components

Cameras

The first additional components that were implemented in the scope of this work are two cameras. UE4 enables accessible camera functionality through its own Camera Actor Component structures. These are fully customizable by means of hard-coded settings - position, orientation, distance from parent component - and can also be attached to another structure called a Spring Arm Component, which acts as a virtual pole that ties the camera to the Pawn and can be configured to induce a

forced movement lag to the camera, making the camera follow the Pawn slower than with the Pawns rotation. Last but not least, cameras are not only a vital video interface for the user, but can also be later implemented as payload or sensor simulation for various robot applications such as environment observations or collision avoidance. Towards enabling this future usage, joystick-based rotation control has been implemented for the trailing camera.



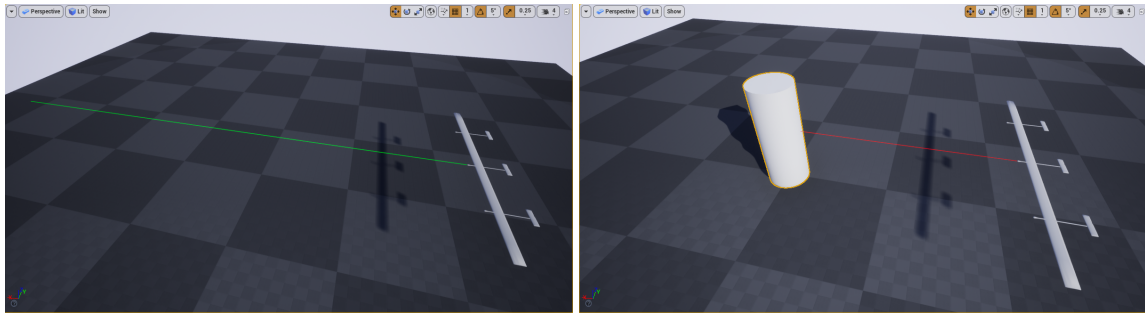
Figure 21: Current cameras

Figures 21a and 21b show the two cameras currently implemented in the scope of this work. The cameras can be switched between during runtime by means of a custom key-binding.

The Scanner component

The Scanner component is the most important aspect of this work, since it provides the means required for sampling the environment around a robotic system and prepare this information for use in external modules. This component therefore creates a bridge between the robot simulation and the environment in which the robot is located. Following the concept described in chapter 3.3.3, the Scanner component employs a ray-tracing-based design in order to sample the environment around the robotic system it is attached to. Four iterations of this Scanner component have been implemented, each building upon the functionality of the previous, as described below.

The initial decision was to cast one ray in the plane's forward direction, in order to investigate its functionality. UE4 has its own functions for ray-tracing, which require a *channel*. To this end, a decision has been made to have all scanner iterations within this work cast rays in UE4's *WorldStatic* channel, based on the fact that all current world objects, except for ELAHA, are of stationary nature - rocks, cliffs, etc. The behaviours of the initial ray without a hit and the same ray with a hit are shown in figures 22a and 22b respectively.



(a) Initial ray cast

(b) Initial ray hit

Figure 22: Initial ray-trace

Having achieved this, the next iteration of the Scanner would be a scan of the ground directly below ELAHA. This approach follows the concept described in chapter 3.3.3 and is inspired from a Citizen Con presentation [23] discussing ray-scanning being used towards achieving accurate procedural walking animation generation. Therefore a much larger number of rays is being used and a distance of one meter has been chosen between any two scan points. Moreover, the maximum scanned distance in one direction has been chosen to be 25 meters, since ELAHA should be capable of a maximum forward velocity of around $10m/s$. Therefore, in this iteration of the Scanner, all rays are cast every frame, from the aircraft altitude vertically downwards, at 1 meter intervals and up to 25 meters in front, to the back and to each side of the plane. This provides a 1-meter-resolution 2.5D map of the terrain under ELAHA within 25 meters in each direction, which could potentially be used in order to simulate aircraft-to-runway collision when landing. The scanner functionality is shown in figure 23.

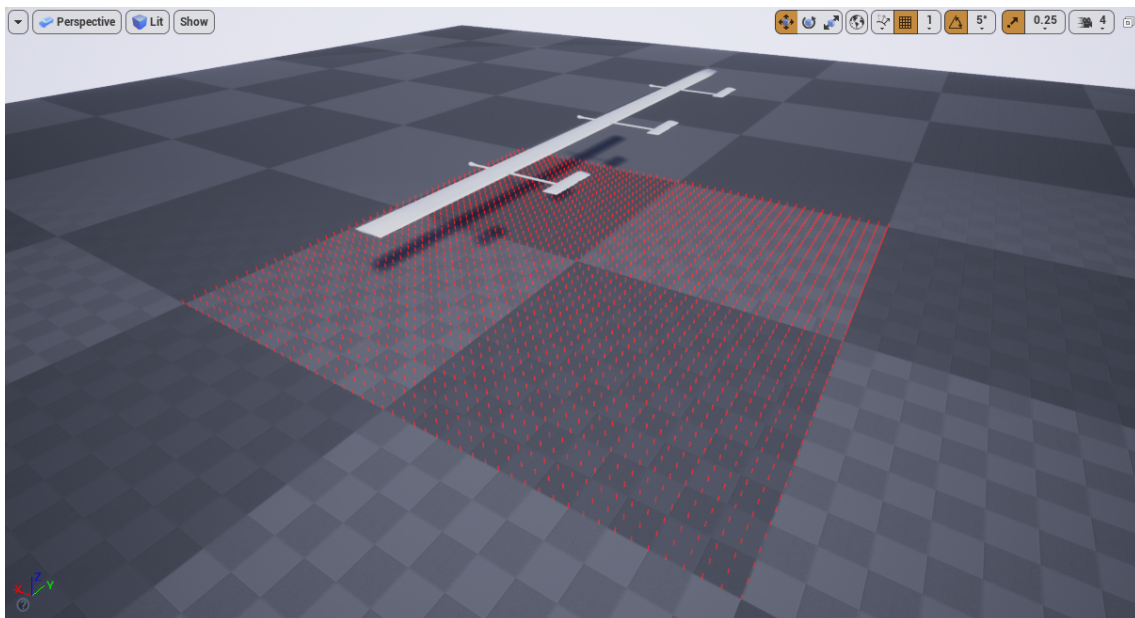


Figure 23: Ground scanner

The third iteration of the Scanner seeks to extend the ground mapping functionality to all directions around the aircraft. To this end, the ground scanner approach is modified in order to cast rays from all three planes intersecting in the origin of ELAHA's mesh - or more generally, in the position of the robots representation - in both positive and negative direction of each plane. Having a 1 meter gap between rays and a covered distance of 25 meters in every direction computes to 51 by 51 rays per plane, for 6 planes, which gives $51 \cdot 51 \cdot 6 = 15606$ total scanned points per frame. In this approach, the Scanner can be considered to take the form of a box around ELAHA, with its centre in the position of the aircraft. The Scanner's functionality in this iteration is presented in figure 24.

The Scanner's fourth and final iteration in the scope of this work is following the same box shape and the same point resolution as the third iteration. This way, the output of the Scanner is separated in six inter-locked 2.5D voxel maps. This gives a full scan of ELAHA's environment. This fourth Scanner iteration has added functionality for saving all scanned voxels in *world coordinates*. This is important, because it ensures the Scanner provides meaningful data to an External Physics simulation - in the form of the voxel map - which then only additionally requires ELAHA's pose in order to check whether the aircraft is in collision with a point in any of the point clouds. In order to further enable this potential external collision check, this scanner iteration also adds an *occupancy* parameter to each voxel which can take different values:

- -1 or Unvisited when the grid element does not contain a valid voxel,
- 0 or Free when the ray hits no obstacle within the 25 meter box,
- 1 or Occupied when the ray finds an occupied voxel.

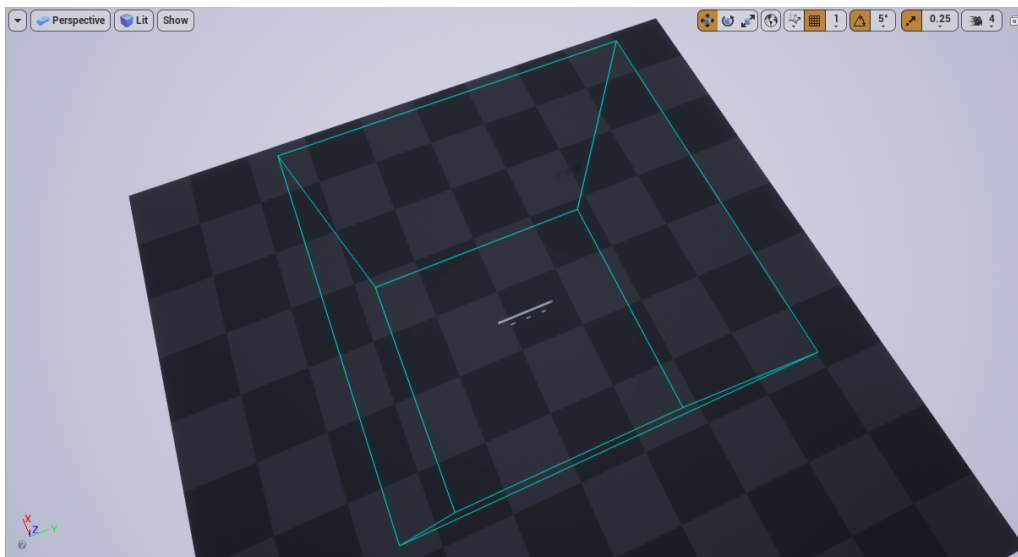


Figure 24: Box scanner

Another feature that has been added to the fourth iteration of the Scanner is refresh logic. While the third iteration re-scans the entire area every frame, two variations of the fourth iteration have been developed in an attempt to split the load between frames. This is made possible by the very high frame rate UE4 achieves during runtime, since a lower update frequency might cause some maps of the Scanner to lag behind, leading to failure. The two variations split the scanning process in 6 steps (one map per frame) and 12 steps (half a map per frame) respectively. Since the maps are vital to ensure a correct output from the Scanner, the 6-stage variation will be used for the remainder of this work, thus avoiding map break-up through unforeseen frame drops or other events. This Scanner version scans each face of the scanning box per frame and as such will hence-forth be referred to as the Face Scanner.

Figure 25 shows the scanner component with activated debug lines applied to the custom level. According to the logic above, green indicates a free voxel, red - an occupied one. The functionality of the scanner component is evaluated and discussed in chapter 5, more particularly in chapter 5.3.

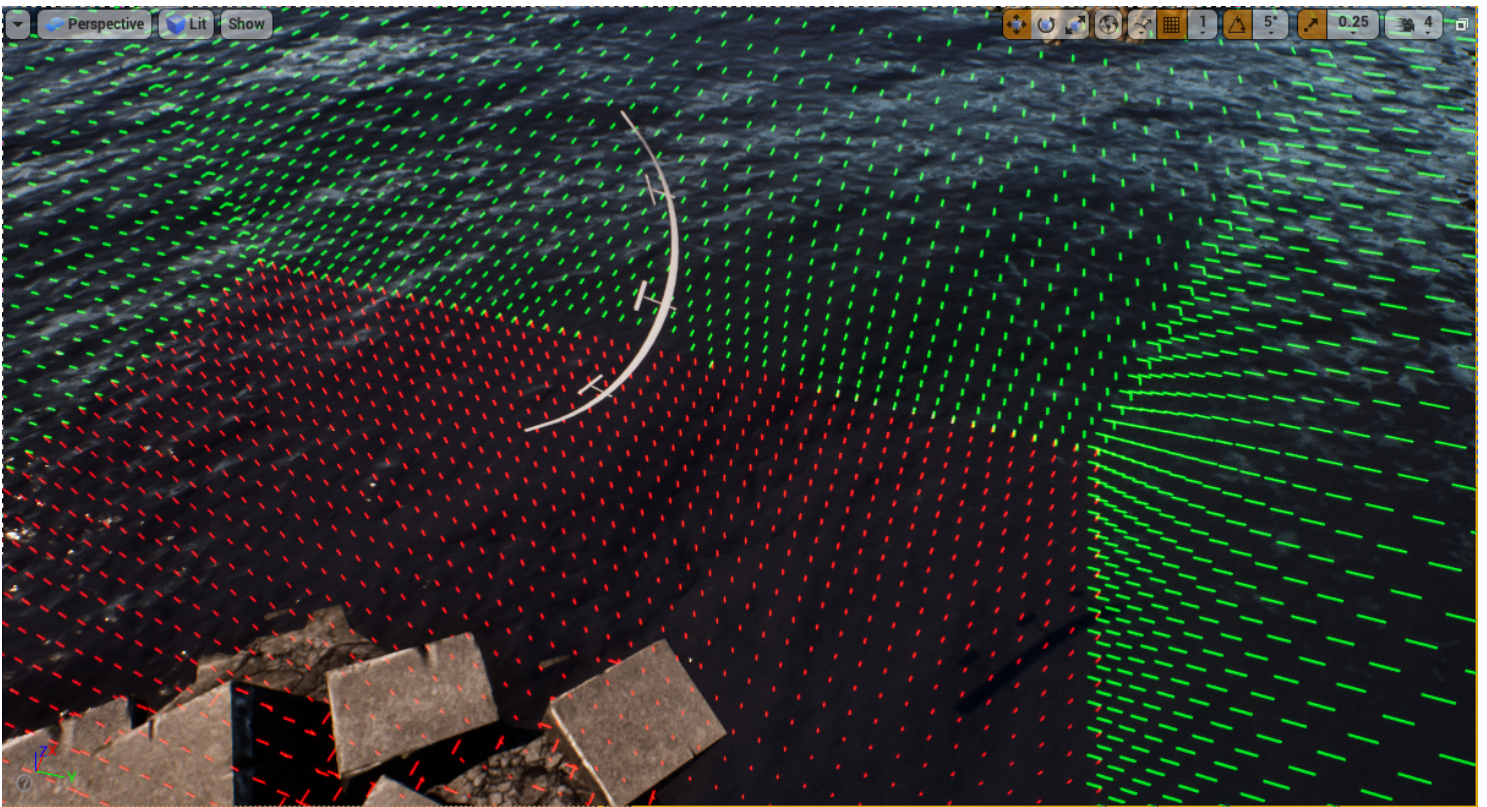


Figure 25: Face Scanner applied to the custom world

4.4.4 The Middleware Interface

It must be mentioned that this module does not entail the implementation of a middleware, but rather using an already existing one and creating an interface between it and the UE4 environment. Therefore, the middleware used in the scope of this work - Links and Nodes - is a closed-source software widely used within the DLR.

Links and Nodes

Links and Nodes, in short LN, is a software developed in-house at the DLR. It enables fast and secure data transfer between different applications on the same computer - through shared memory - or on different machines on the same network (computers, robots, etc.) - by making use of UDP protocols. Links and Nodes is therefore the optimal tool to use in order to achieve the modular design of the platform described in this work, allowing for the SITL chain to be split up on different machines, thus lowering the load on any one machine to a minimum, therefore improving performance. Figure 26 presents the Graphical User Interface of the manager.

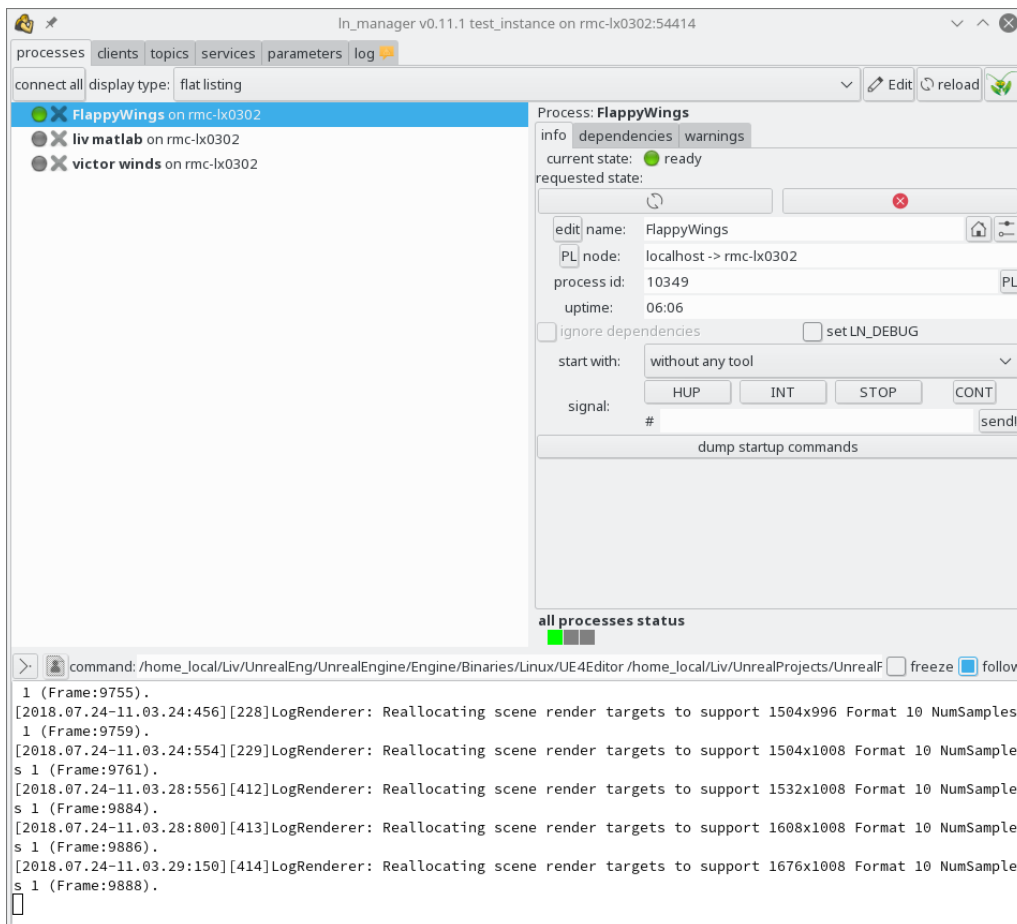
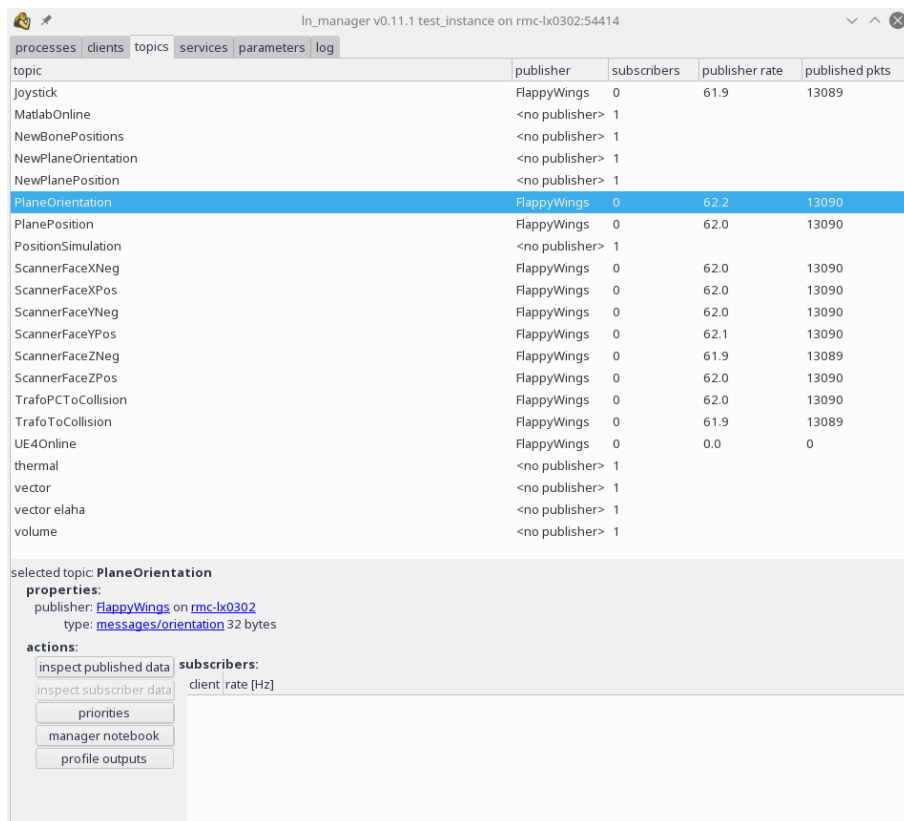


Figure 26: Links and Nodes manager GUI

Links and Nodes is built upon a manager-client schematic. This means that multiple clients - defined by unique names - are bound together by the same manager and directly exchange information through pockets of data, which are called topics. The manager itself only connects the clients together but does not deal with the data exchanges, thus increasing efficiency of the system. Each topic has a unique name by which it is identified, as shown in figure 27. Each client can either publish a topic - which means the client updates the data in the topic - or subscribe to one - which means it can only read data from the topic without altering it. Multiple clients can subscribe to the same topic, as long as the topic name is unique, but only one client can be its publisher, fact which ensures data safety. While the publish function write data when called, the read function has two modes of operation: blocking and non-blocking. If used in a blocking configuration, calling the read function pauses the software that called it until new data is available on the topic. While this allows for synchronization between modules in certain scenarios, it can also damage the flow of the SITL chain if one of the modules publishing on a certain topic slows down too much. The non-blocking way of using the read function attempts to read a topic and read the latest available data on the topic, regardless of whether or not it already read that data set. This means the read function always retrieves the latest available data from the topic. This means the SITL modules will not be slowed down by a slower module, but in some cases, it could also mean reusing old data which are not meant to be reused.



topic	publisher	subscribers	publisher rate	published pkts
Joystick	FlappyWings	0	61.9	13089
MatlabOnline	<no publisher>	1		
NewBonePositions	<no publisher>	1		
NewPlaneOrientation	<no publisher>	1		
NewPlanePosition	<no publisher>	1		
PlaneOrientation	FlappyWings	0	62.2	13090
PlanePosition	FlappyWings	0	62.0	13090
PositionSimulation	<no publisher>	1		
ScannerFaceXNeg	FlappyWings	0	62.0	13090
ScannerFaceXPos	FlappyWings	0	62.0	13090
ScannerFaceYNeg	FlappyWings	0	62.0	13090
ScannerFaceYPos	FlappyWings	0	62.1	13090
ScannerFaceZNeg	FlappyWings	0	61.9	13089
ScannerFaceZPos	FlappyWings	0	62.0	13090
TrafoPCToCollision	FlappyWings	0	62.0	13090
TrafoToCollision	FlappyWings	0	61.9	13089
UE4Online	FlappyWings	0	0.0	0
thermal	<no publisher>	1		
vector	<no publisher>	1		
vector elaha	<no publisher>	1		
volume	<no publisher>	1		

selected topic: **PlaneOrientation**

properties:
publisher: [FlappyWings](#) on [rmc-ix0302](#)
type: [messages/orientation](#) 32 bytes

actions:

subscribers:
client rate [Hz]

Figure 27: Links and Nodes manager topics

LN as a library

As stated at concept level, part of this work focuses on implementing an interface between LN and the UE4 environment. The first step to this is bringing the third-party LN library object into the UE4 project build environment. UE4 uses its own custom build tool for compiling game system code, named Unreal Build Tool (UBT). UBT is built from source code together with the Engine and employs a complex build structure based on *C#* and *CMake* in order to compile project source code. Bringing the library into the projects build path is therefore not trivial and requires some degree of understanding UBT and the build hierarchy of a UE4 project.

The Unreal Engine build system - Unreal Build Tool - is based on *C++* and *C#*. The compiler flags are hard-coded into the Unreal Build Tool, its source being written in *C#*. This source code also sets a parameter that controls the required *C++* toolchain for project compiling under Linux. In the case of Unreal Engine 4.19, the Linux build is only made to compile on *clang*. Therefore, in order to link the LN library, it must also be compiled with *clang*, due to the presence of a *C++*-specific symbol in the LN library object. To this end, the Makefile contained within the library folder of the LN source code must be edited. Since the operating system used within this work is openSuse 42.3, the existing Makefile entry for this architecture must be extended for *clang*. Once done, one must compile the LN library for this new architecture-toolchain pair, using this edited makefile. The outcome should be a static library object.

Finally, the UE4 project build system uses *C#* source files in order to set the build parameters used by *clang*, such as flags and source files. The LN library must be added - headers and library object - in the *ProjectName.Build.cs* file found in the source directory of a UE4 project. The two commands required in this class file are:

- `PublicIncludePaths.Add("Path/To/LN/Header/Files");`
- `PublicAdditionalLibraries.Add("Path/To/LN/Library/Object/library.a");`

The Interface

The LN Wrapper class has three main parts - the Initialization-Destruction (I-D) chain, a receiver part and a transmitter part. The I-D chain is the part of the class called at start-up and end-of-program. At start-up, it creates a new client with the specified name, requests a connection to the manager on the given address and port, initializes the message data containers and publishes and/or subscribes to all the topics required by the application. It is important to note that trying to connect to a manager without an active manager instance results in a time-out after a number of tries. Once a connection to a manager is established, subscribing to topics that haven't been published yet is possible, since the manager will create a new topic with the given name, assigning the subscriber to it but waiting for a publisher. At end-of-program, the I-D chain unpublishes and/or unsubscribes all topics and then terminates the client. This is important since a manager only allows clients with a

unique name and any attempt to connect a client with an already existing client name results in an error code being thrown and the client application crashing. The same methodology applies to topic names.

<pre>void LnnWrapper::InitializeLN() { In_cl = new In::client("FlappyWings_Unreal", "rnc-1x0302:54414"); NewBonesPosePort = In_cl->subscribe("NewBonePositions", "messages/bones"); IsMatlabOnPort = In_cl->subscribe("MatlabOnline", "messages/connection"); NewPlanePositionPort = In_cl->subscribe("NewPlanePosition", "messages/position"); NewPlaneOrientationPort = In_cl->subscribe("NewPlaneOrientation", "messages/orientation"); }</pre>	<pre>In_cl->unsubscribe(NewBonesPosePort); In_cl->unsubscribe(IsMatlabOnPort); In_cl->unsubscribe(NewPlanePositionPort); In_cl->unsubscribe(NewPlaneOrientationPort);</pre>
(a) Links and Nodes ID I	(b) Links and Nodes ID D

Figure 28: Links and Nodes I-D chain sample code

The other two parts of the LN Wrapper class are implemented in the form of an API and as such need to be called from the main UE4 modules. The receiver function has a straight forward functionality - upon call, it reads all the topics the client is subscribed to and updates all the corresponding message structures with any new data. An important aspect of this function are the blocking and non-blocking modes of LNs read functions. In the scope of this work, all the read functions that were implemented are set to be non-blocking, in order to avoid the publishing rate of one of the clients/applications influencing another. The transmitter part provides functionality for uploading data to the published topics. Since new data comes from different sources and might be produced at different rates, the transmitter is implemented following a one-function-per-topic approach. This enables the UE4 modules to upload data on each published topic individually, without having to depend on one another. Furthermore, as mentioned, before, data coming from UE4 may need processing before being uploaded to a topic, which means every publisher function provides support for data conversion to satisfy this requirement.

<pre>void LnnWrapper::Update() { NewBonesPosePort->read(&NewBonesPose); NewPlanePositionPort->read(&NewPlanePosition); NewPlaneOrientationPort->read(&NewPlaneOrientation); } </pre>	<pre>void LnnWrapper::SendPPLN(double x, double y, double z) { PlanePosition.x = x; PlanePosition.y = y; PlanePosition.z = z; PlanePositionPort->write(&PlanePosition); } </pre>
(a) Links and Nodes Update	(b) Links and Nodes Send

Figure 29: Links and Nodes Update-Send functionality sample code

Bandwidth requirements

When using LN with all clients on the same computer, shared memory enables fast and worry-free data transmission. When a UDP connection is required, bandwidth becomes important, since all data has to be transmitted fast and complete every frame. In the current iteration of the platform, all data types being transmitted through the middleware are either integer, float or double. The most important topics are the robots position and orientation (seven doubles, since orientation is represented as a quaternion), bone orientations (22 bones, two doubles per bone, representing pitch and roll of the corresponding wing element), the Scanner data (all

six faces, 2601 points each, one integer and three floats per point) and the incoming information from the collision query module (one force and one torque, six doubles total). Therefore, 1 shows the elements enumerated in terms of individual minimum data volume requirements and the total minimum required bandwidth calculation is given in 2.

$$\begin{aligned}
 \text{Pose} : P &= 7 \times 8B = 56B \\
 \text{Bones} : B &= 22 \times 2 \times 8B = 352B \\
 \text{Scan} : S &= 6 \times 2601 \times 4B + 6 \times 2601 \times 3 \times 4B = 249696B \\
 \text{Collision} : C &= 6 \times 8B = 48B
 \end{aligned} \tag{1}$$

$$\begin{aligned}
 P + B + S + C &= 250152B \\
 &= 2001216\text{bits}/\text{update} \\
 &\approx 244.3\text{KB}/\text{update}
 \end{aligned} \tag{2}$$

4.4.5 External Modules

The modules discussed in this sub-chapter are - as the name states - external software used in conjunction with the platform developed in the scope of this work. They have been chosen to be used as test cases for the SITL chain and are standalone applications/libraries on their own.

ELAHA Dynamics Simulation

A concurrent work currently being conducted at the DLR is the ELAHA flight dynamics model simulation [34]. This work focuses on modelling and simulating the ELAHA aero-elastic behaviour, in other words the interaction between structural and aerodynamic forces and the resulting motion and deformation.

XCD Collision Query

XCD is a collision detection library designed for haptic applications [35] and is being developed in-house at the DLR. Toward evaluating the reliability of the scanner component implemented in this work, XCD is interfaced with in order to attempt to externally simulate collisions between the UE4 Pawn and the environment around it, by means of data coming from the Scanner Component.

XCD determines collisions by comparing the positions of points in a point cloud with respect to a voxel map. In the scope of this work, two voxel maps are created - one of a simple cube and one based on the ELAHA mesh by means of its Blender file - and the point cloud is built on the output of the Scanner Component, namely six different point clouds are created, representing the six 2.5D scanner maps in world coordinates. In order to do this, XCD needs the coordinates of each point within a point cloud with respect to the voxel map pose. In the current case, since all the points are in

world coordinates, it means that XCD requires the inverse transformation from the Pawn coordinate frame to the world frame.

Once this information has been passed on to XCD, a brief explanation of the algorithms functionality follows:

- The algorithm iterates through all the points of the point cloud and checks whether any of them are within the bounds of the layered voxel map, extended by a pre-defined safety distance;
- For each colliding point, the corresponding layer of the voxel map is computed;
- Based on the points penetration through this layer and the position of the colliding points with respect to the voxel map origin, forces and torques acting on the voxel map are computed;
- These forces are then summed up and the resulting force and torque acting on the pawn are then published on their respective topics.

```
Penetration = 0.0200001  
Number of points colliding = 25  
Timestamp [s], t_comp [ms], F [N?], T [Nm?] = 27.657, 0.134033, [ -0, -0, 0.500003 ], [ -8.3819e-09, 5.58794e-09, -0 ]
```

Figure 30: XCD output

5 Evaluation

5.1 Benchmarking

The system handled in the scope of this work is highly resource reliant, both in terms of GPU and CPU power. As such, the performance of the platform must be evaluated against the system specifications of the computer it runs on. The machine used for the development of this simulation and visualization software based on Unreal Engine 4 has the following specifications:

- CPU Intel Xeon E5-1630, 8 cores @ 3.70GHz
- GPU NVidia Quadro P600, 2GB dedicated video memory
- RAM 15.6GB
- OS openSUSE Leap 42.3 64-bit

This software is implemented using Unreal Engine version 4.19, released in March, 2018. Complete forward and/or backward compatibility is not assured, thus adaptations may need to be applied when attempting to run the platform on a different engine version.

5.2 Frame rates

The first part of the evaluation refers to the overall performance of the system. Robotic systems are usually being simulated at a high step frequency, which means this platform must also be capable of running at a high refresh rate. While it is not imperative that the UE4 software runs at the same step frequency as the robotic system - fact which simply cannot be achieved when the robot is being simulated at, for instance, a 1KHz step frequency, a smooth transition of the robot representation must be assured between steps. Furthermore, the scanner component is strictly dependant on the platforms frame rate and as such frames must be short enough in order for a full scan of the environment to be conducted in the shortest time possible.

Empty world

The following tests have been conducted three times each, on different samples of 500 frames each. The frame durations in milliseconds - represented by the blue line - have been plotted against the frame numbers. Figure 31 shows frame rates recorded for the pawn in an empty level without the scanner functionality. This means only the robot representation module is active in this setup. Figure 32 is similar, but with the Scanner module activated. This adds the functionality of an additional component to the software loop, thus increasing load per frame. Last but not least, figure 33 adds visual debug to the setup, thus further increasing load per frame as well as adding a visual rendering factor which, when considering a game engine, should be the most resource demanding type of operation.

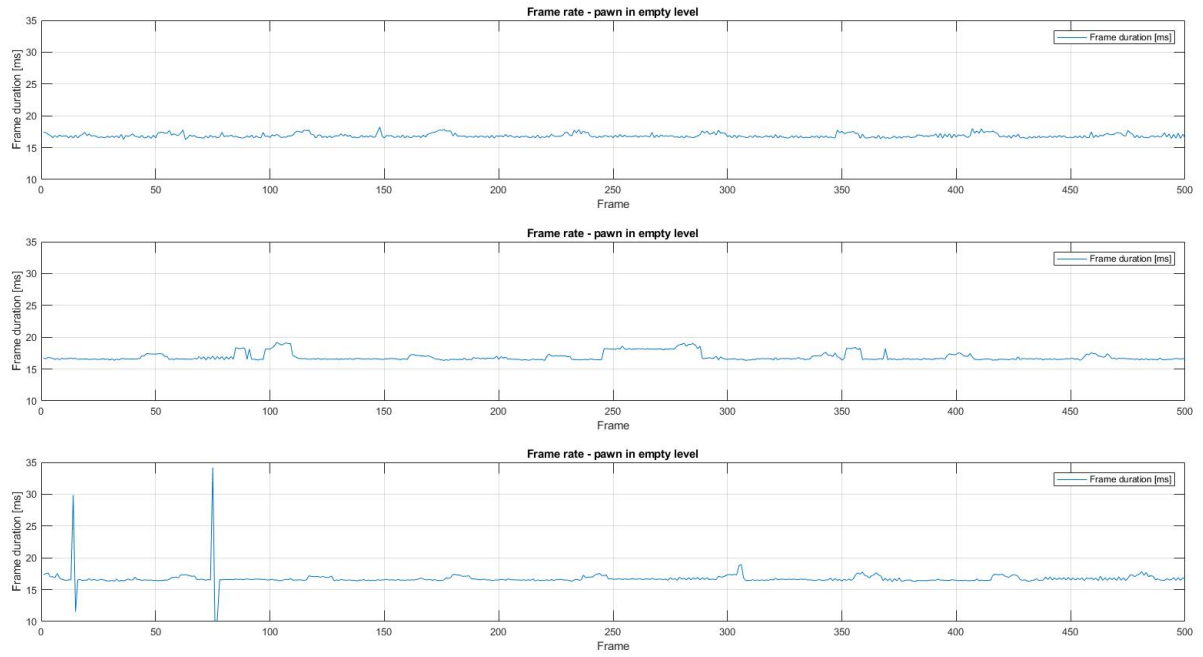


Figure 31: Frame rates for Pawn in empty level

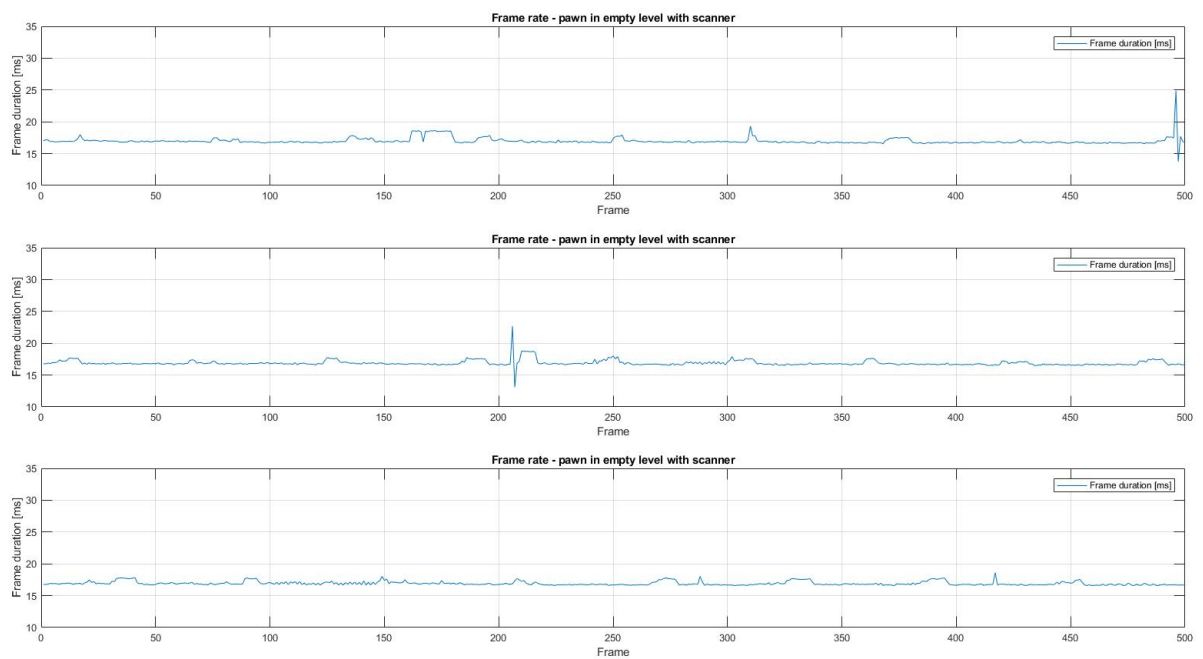


Figure 32: Frame rates for Pawn in empty level with scanner

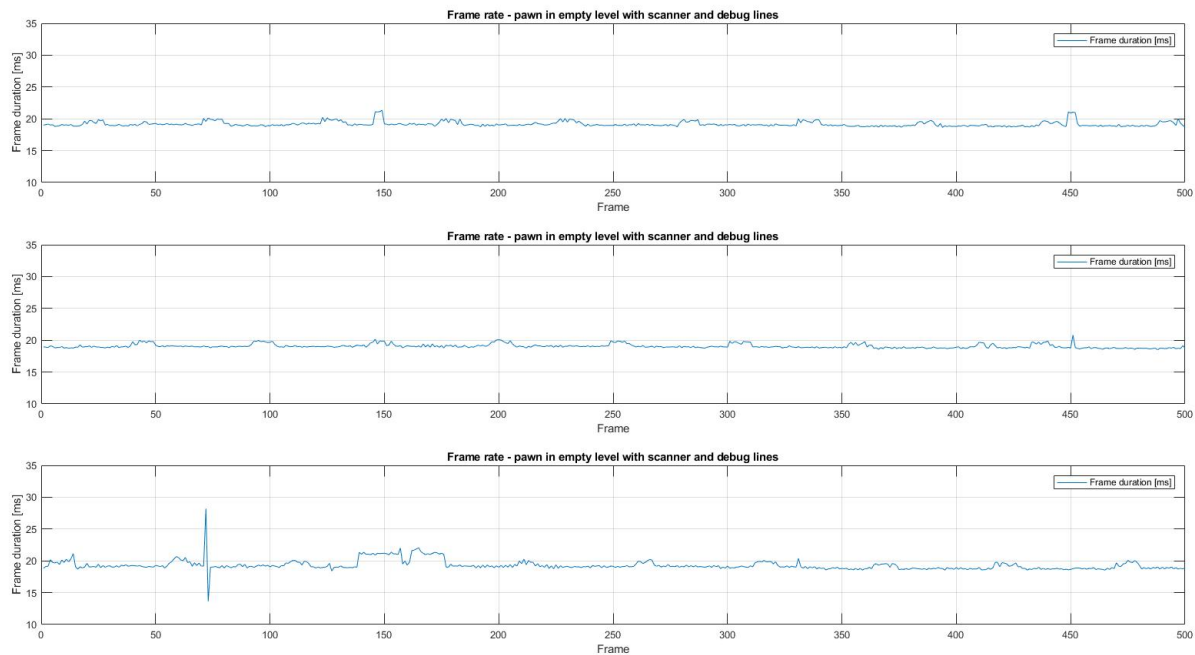


Figure 33: Frame rates for Pawn in empty level with scanner and debug

Custom world

The following tests have been conducted in the same way as the previous, three times each, on different samples of 500 frames each. The frame durations in milliseconds - represented by the blue line - have been plotted against the frame numbers. Figure 34 shows frame rates recorded for the pawn in the custom Infinity Blade 4.4.2 level without the scanner functionality. This means both the robot representation and the world representation modules are active in this setup. Figure 35 is similar, but with the scanner module activated. This adds the functionality of an additional component to the software loop, thus increasing load per frame. Last but not least, figure 36 adds visual debug to the setup, thus further increasing load per frame as well as adding a visual rendering factor which, when considering a game engine, should be the most resource demanding type of operation. The main difference expected in the outcomes of these tests is a higher overall frame duration due to the GPU resource requirements of the world representation module.

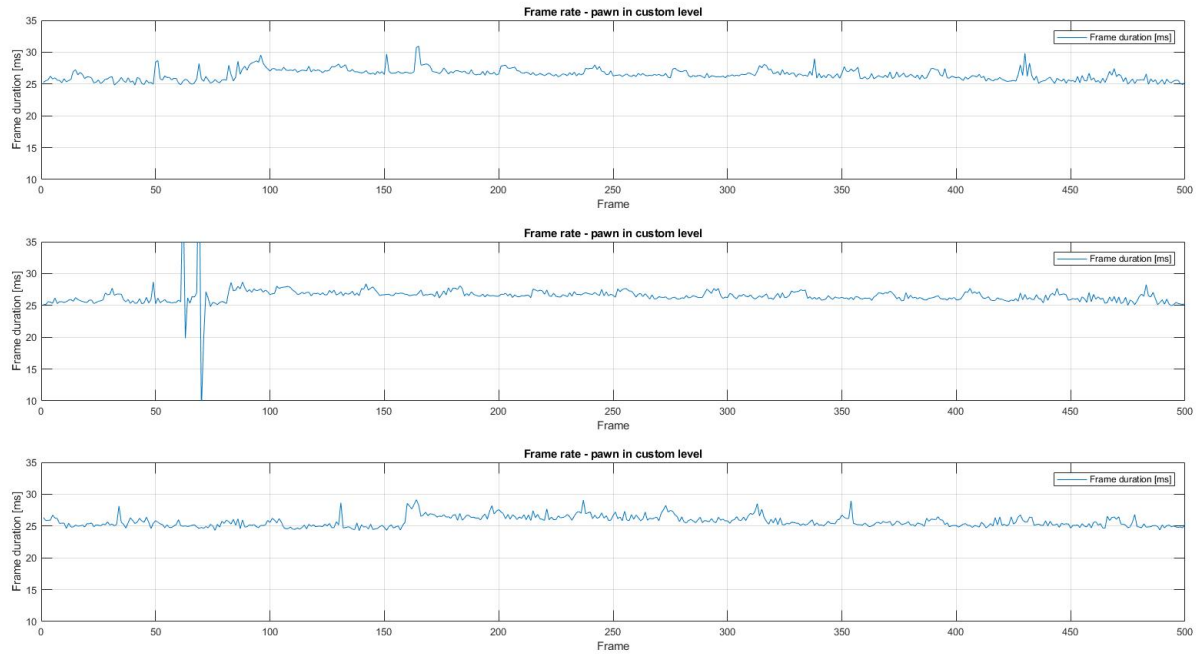


Figure 34: Frame rates for Pawn in custom level

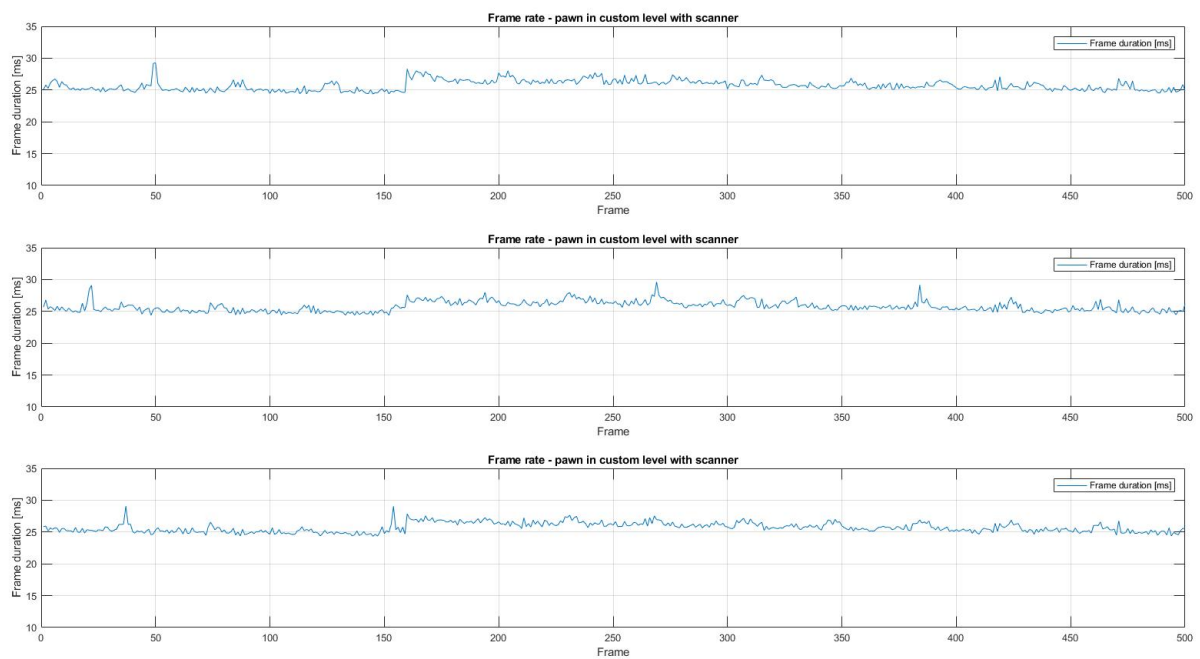


Figure 35: Frame rates for Pawn in custom level with scanner

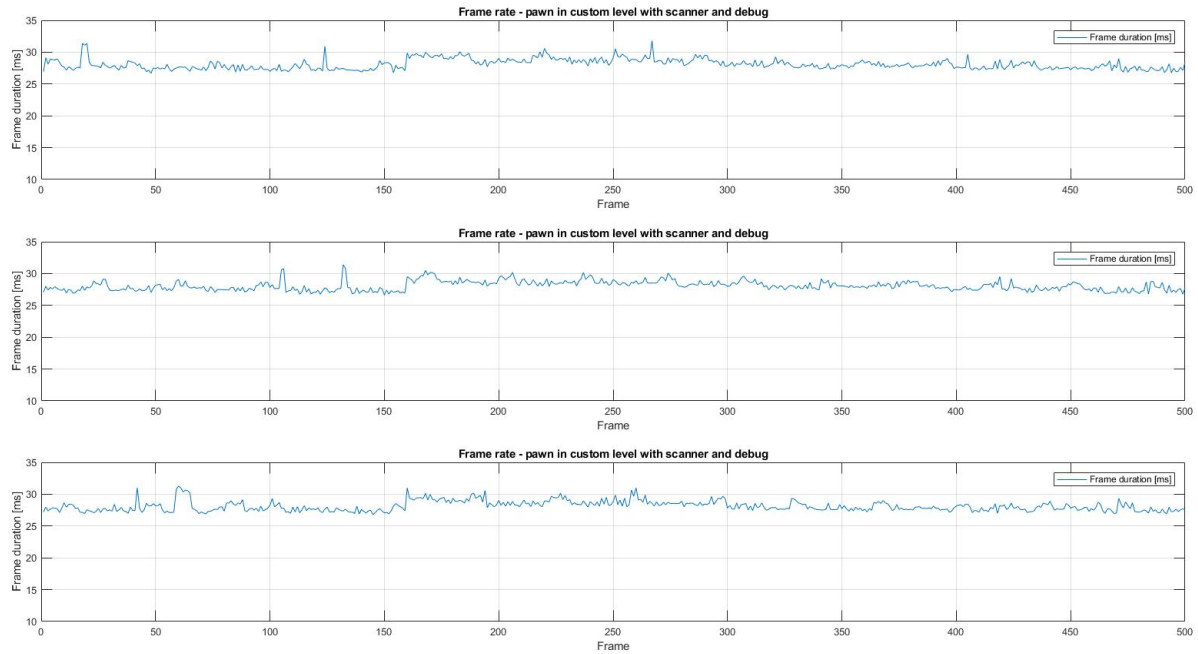


Figure 36: Frame rates for Pawn in custom level with scanner and debug

First and foremost, it is clearly visible that all iterations of the same test showcase the same average frame rate over three iterations. This speaks to the reliability of the system over extended use. The spikes visible for example in the third plot of figure 31 or the second plot of figure 34 are present due to the soft real-time characteristic of the platform, running on a non-real-time operating system. Secondly, when considering the two main test categories independently (empty and custom world), it is visible that adding the scanner functionality does not influence the frame duration in any way, however adding visual debug functionality does. This emphasizes the speed of the scanning approach but is also expected when adding visual elements to the frame load, since the main bottleneck for the performance of this platform is the GPU. Thirdly, when considering the transition between the empty and custom world test categories, a significant frame duration increase is visible. This is expected behaviour due to the same GPU resource bottleneck but could potentially still be optimized upon through follow-up work by means of different game engine systems such as terrain streaming.

5.3 Scanner component

This part of the evaluation seeks to assess the scanner's functionality in terms of accuracy, reliability and data validity. Since this is the main information interface between the simulation and visualization platform and external modules tasked with simulating robot-world interactions, data provided by the scanner need to be both reliable and easily understandable by the external module, requirements kept in mind during the proposed implementation 4.4.3. Therefore, the first part of this evaluation showcases a visual understanding of the scanner's functionality, while the second part looks into how an external application understands scanner data.

Visual evaluation

Figures 37 and 38 present the pawn in the empty world with the second iteration of the scanner and the fourth iteration of the scanner active respectively. A cone primitive has been placed under the aircraft, namely in its negative Z direction. The test here aims at visually evaluating the scanned points. The green and red lines are drawn from a scanner point inwards (towards the Pawn), indicating a point that was hit within the maximum ray length (red) or a point that was scanned at the end of the maximum ray length without encountering a collision (green). The analogy is red for an obstacle point, green for a free point. Figure 37 showcases the expected scanner functionality clearly, where the rays being cast towards the surface of the cone find points correctly and then stop upon collision, with the other rays hitting nothing but instead travelling the maximum given distance. Thus a 2.5D map of the ground track of the pawn is created. Figure 38 shows the same functionality, this time however with the full Face Scanner. The interesting difference visible here is that the rays being cast from the X-Y plane of the pawn towards either direction do not scan the cone primitive, because all those rays have their starting positions within the cone primitive, and therefore the object is invisible to them. This is because the cone primitive has a mesh which is only being rendered from the outside, since it is not meant to contain other objects inside. This is a procedure used in order to save resources on render, but is something that should be taken into account when applying the scanner component to a world.

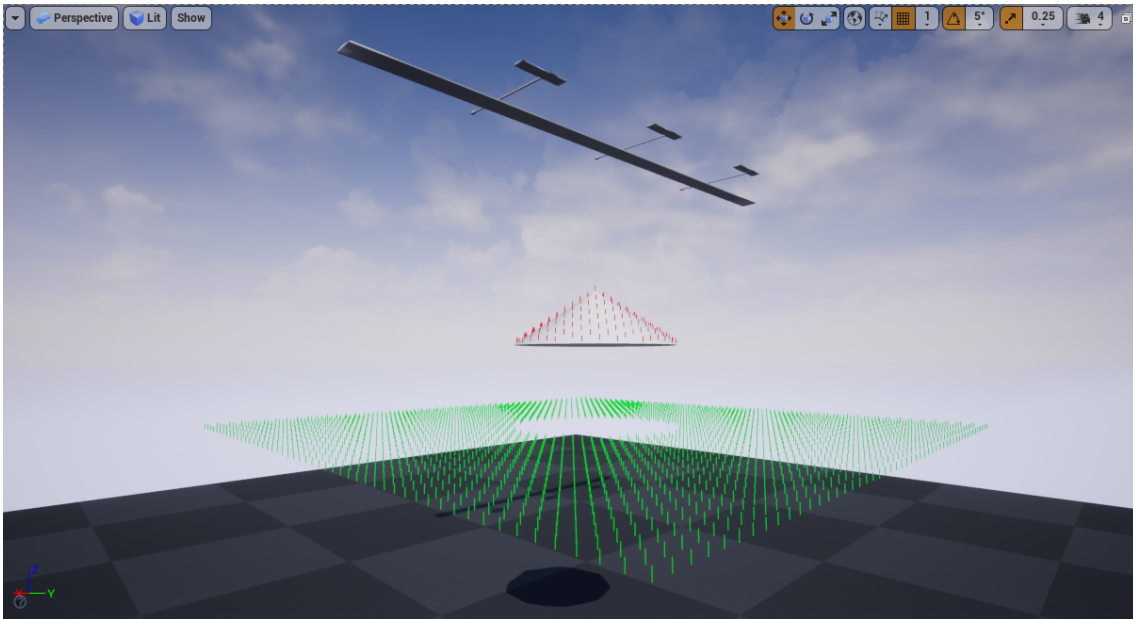


Figure 37: Ground scanner scanning a cone primitive

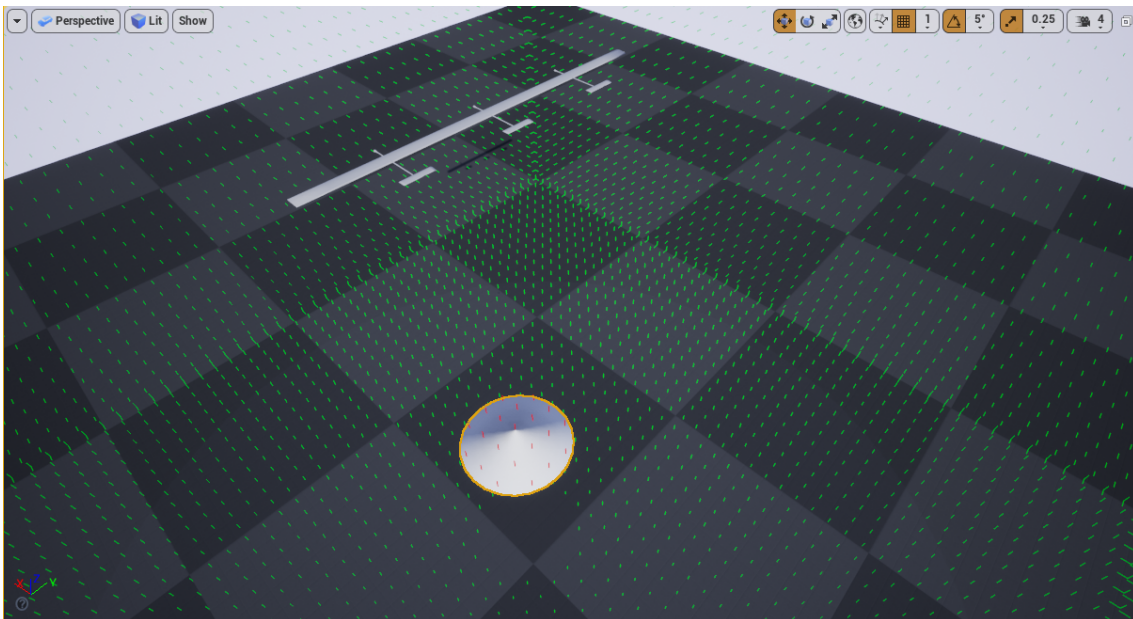


Figure 38: Face Scanner scanning a cone primitive

Figures 39 aim to visually evaluate the scanner functionality when applied within the custom world. As can be seen in figure 39b, the scanner component deals with the custom level actors in the same way as with the cone primitive in the empty level. The rays that hit meshes stop at the obstacle point, while the others reach their maximum length and return a free point. The same drawback presented in the empty level can be seen in 39a when the rays starting from within the mesh of the boulder mesh do not correctly scan the object, since the mesh is not being

rendered from an inside perspective. This is not a major disadvantage for the scanner functionality in the scope of this work, however, since one of the assumptions made when choosing this scanner concept is that the pawn doesn't start the level from inside of another world object.

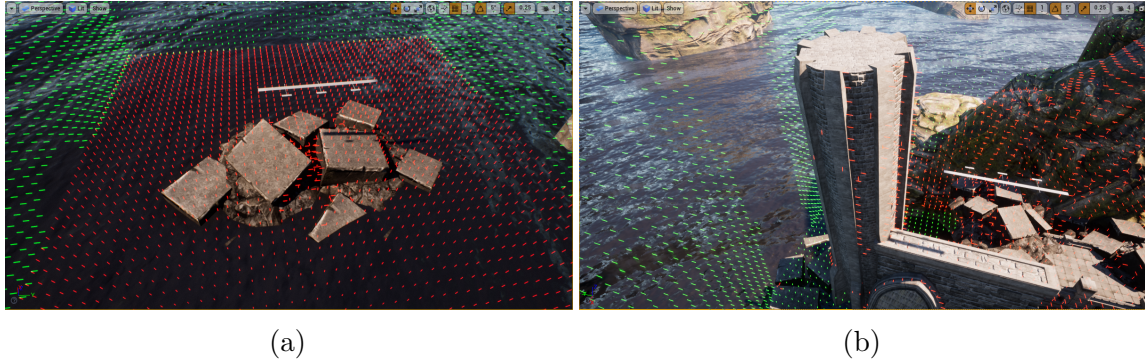


Figure 39: Face Scanner applied to the custom world

External data interpretation

This subsection of the scanner component evaluation seeks to assess how an external module would understand the information output coming from the scanner. Since all its six faces follow the same concept, only one face will be tested here, namely the negative Z face - the one being scanned downwards from the pawn. To this end, raw data (X, Y, Z coordinates) coming from the scanner are being converted into a MATLAB variable and visualized by means of a surface plot. The expected output would be a 2.5D surface map looking like the area scanned by the scanner, viewed from roughly the same perspective. Figure pairs 40, 41, 42 and 43 are being created following this approach, where the top figure shows the scanned area and the bottom one shows the surface plot created with the corresponding data. It must be mentioned that the one meter resolution of the scanner component was used in these test cases.

The showcased scenarios all show an expected scanner behaviour. Figure 40 only shows a flat surface corresponding to the scanned water surface. Figure 41 shows an accurate map of the rocks present in the upper-right quadrant of the scanned area, even at a one meter resolution between sampled points. The test case presented in figure 42 has been chosen in order to investigate the functionality of the scanner when it comes to relatively thin obstacles. The tall stone wall section in the very far upper-right of the scanned area showcases precisely that and proves that the one meter resolution is still enough to provide a rough map of such an obstacle. Last but not least, figure 43 has been chosen to evaluate whether the scanner functionality remains valid when it comes to a more complex mesh. The outcome is positive, with the surface plot showing both the rough shape of the ruined tower, as well as the small rocky top visible in the middle-left side of the map and the elevation layers of the big boulder to the right. The way the ruined tower is being scanned already

hints to a successful application of this Scanner component for simulating a drone flying through closed, tight spaces.

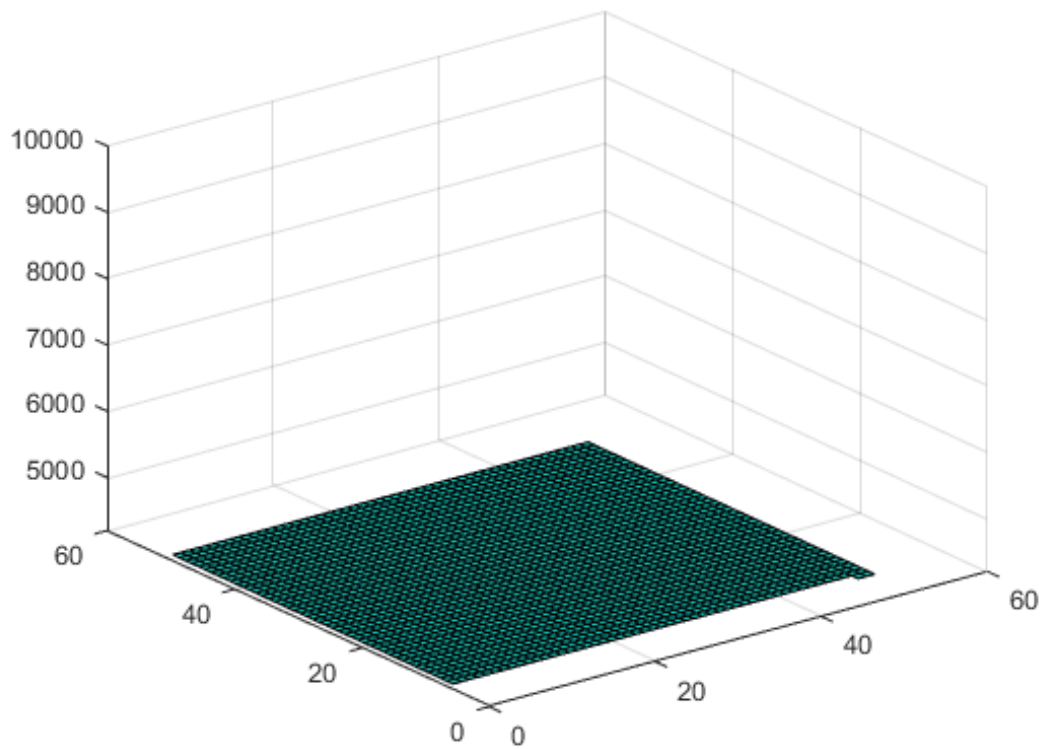
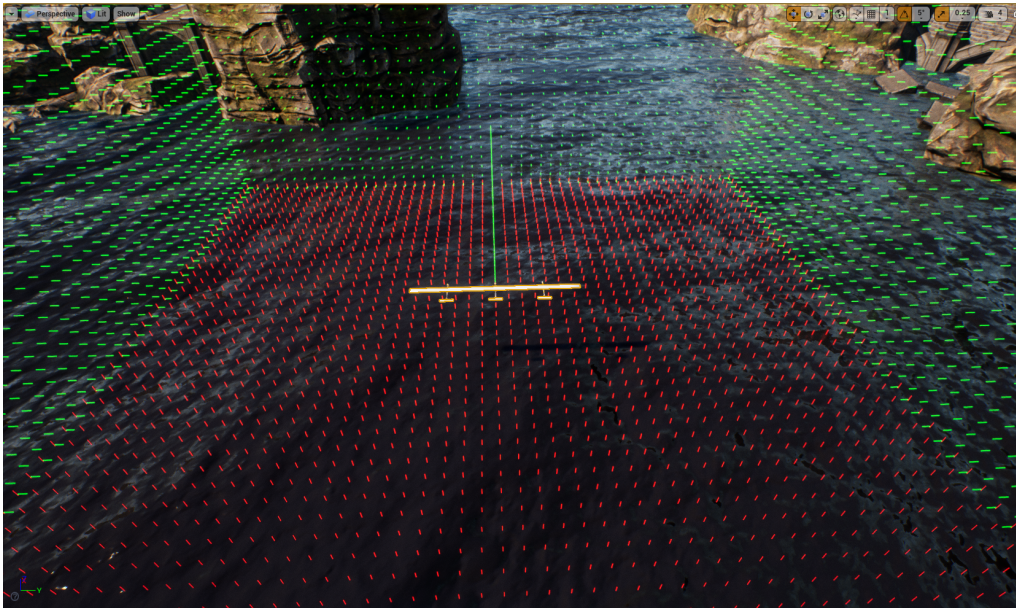


Figure 40: Surface plot of water scan data

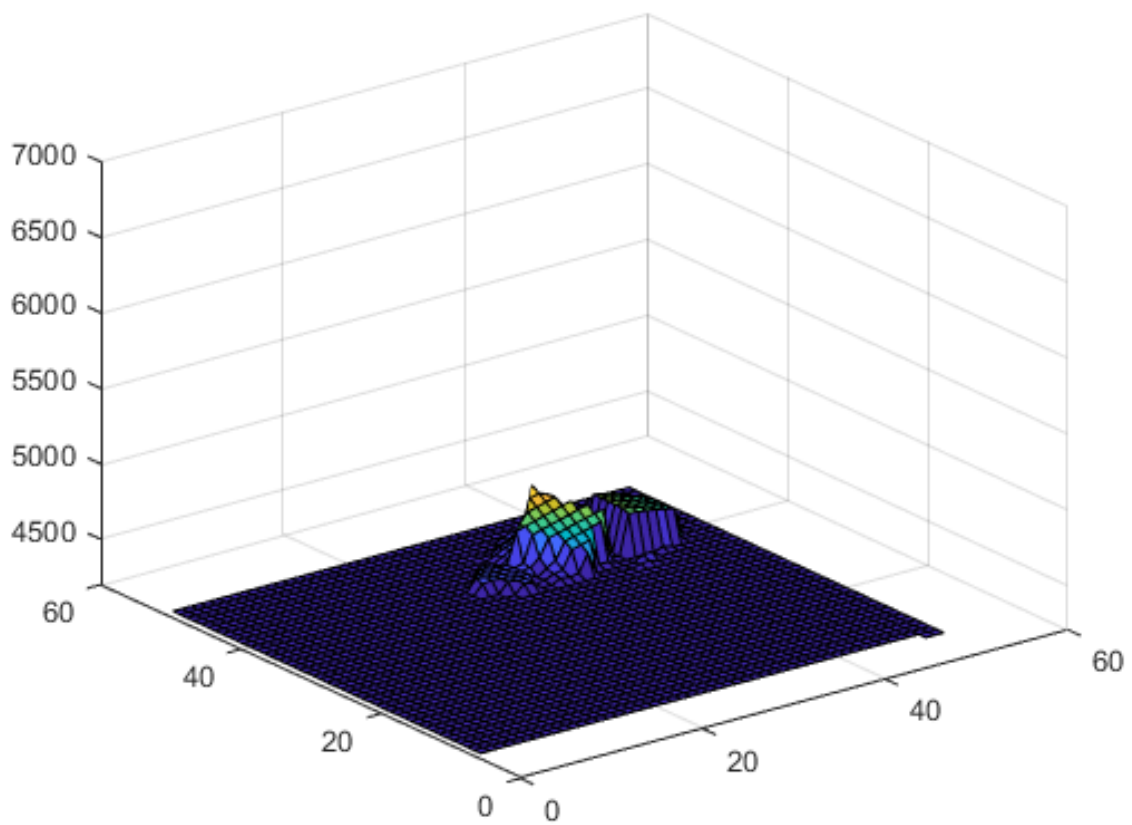
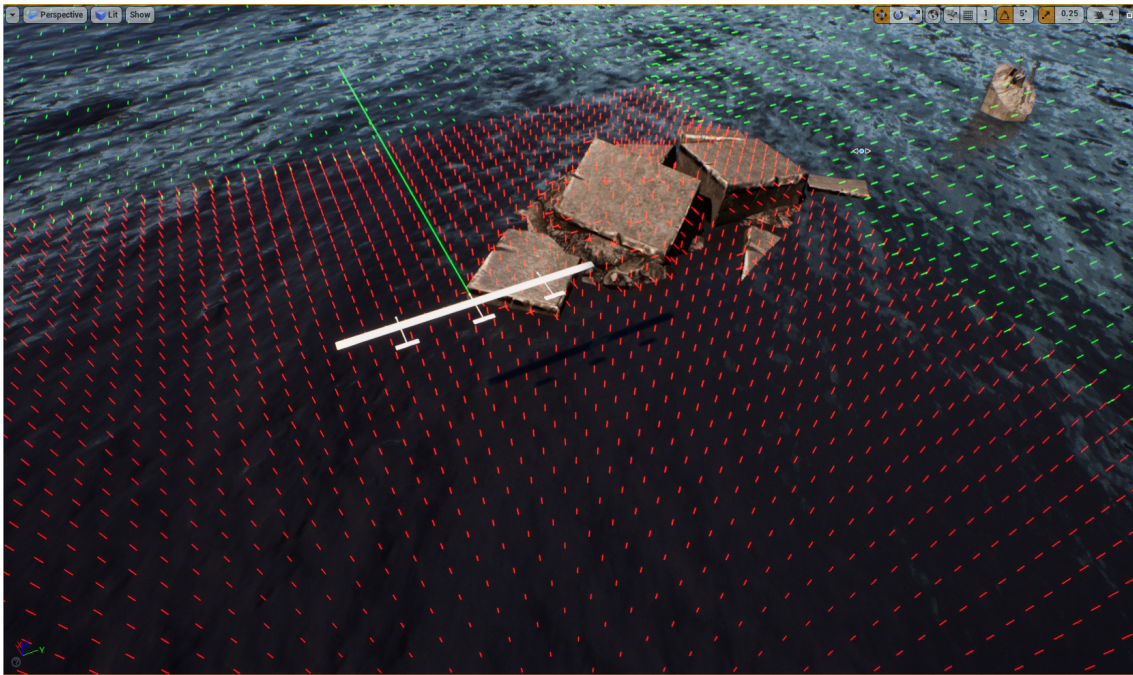


Figure 41: Surface plot of small rock scan data

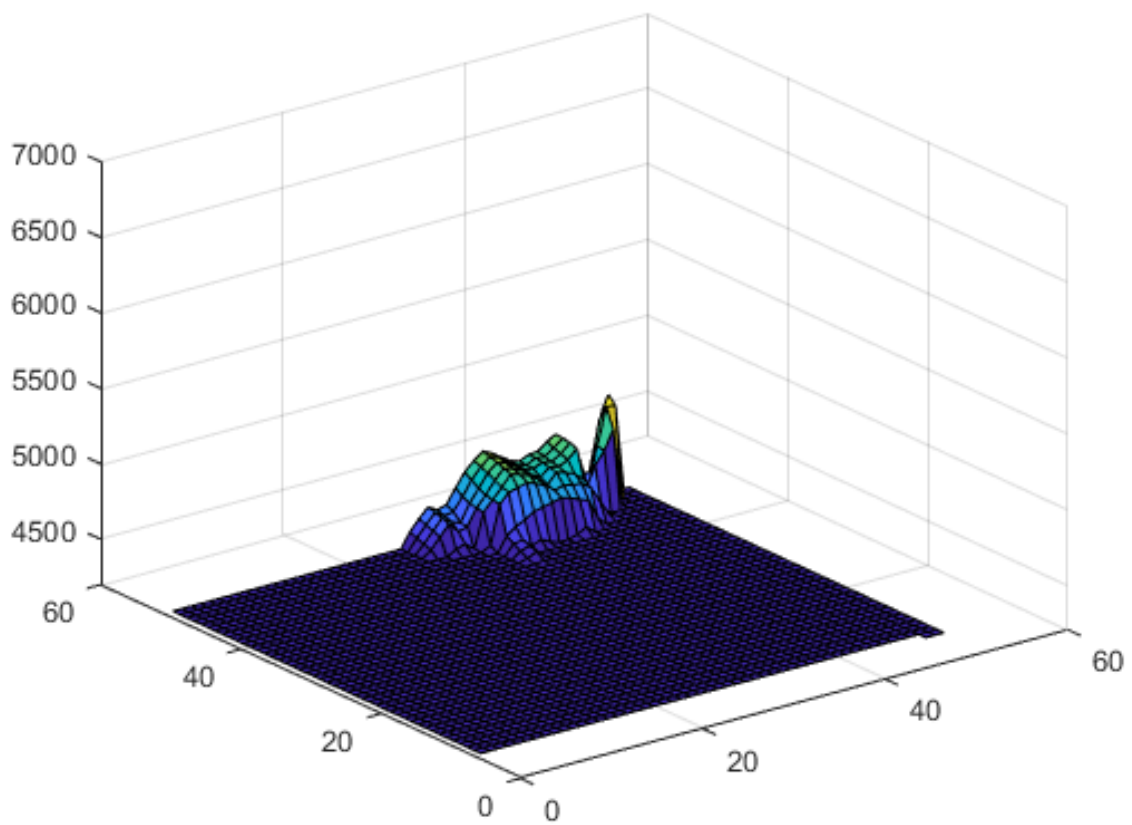
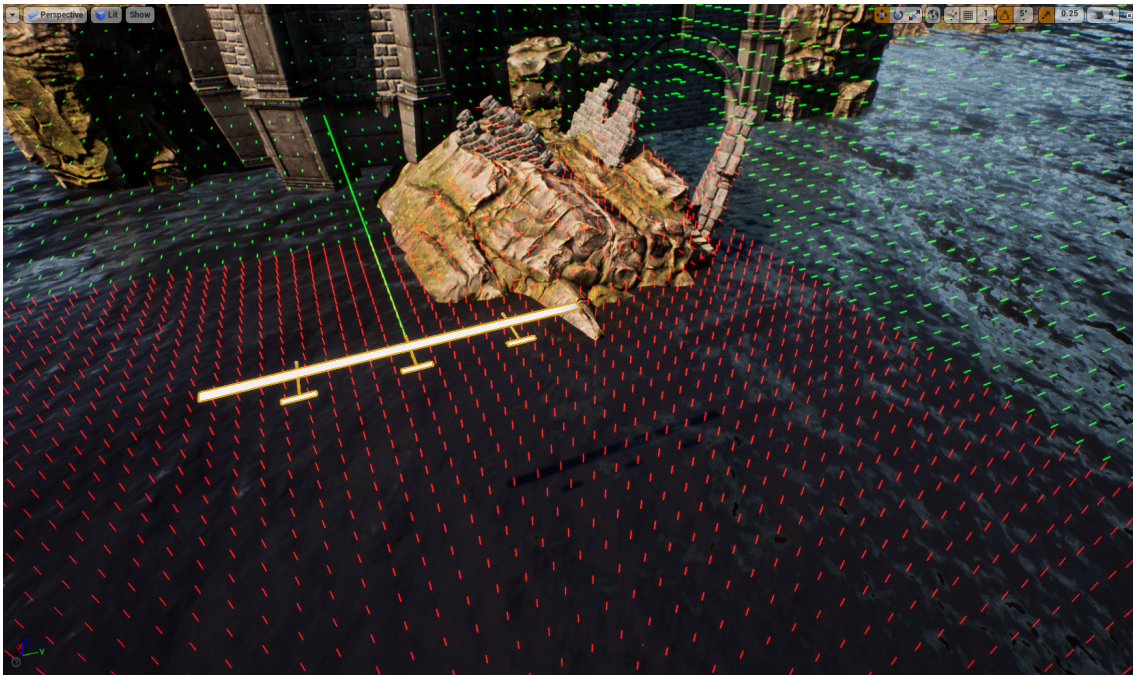


Figure 42: Surface plot of big rock scan data

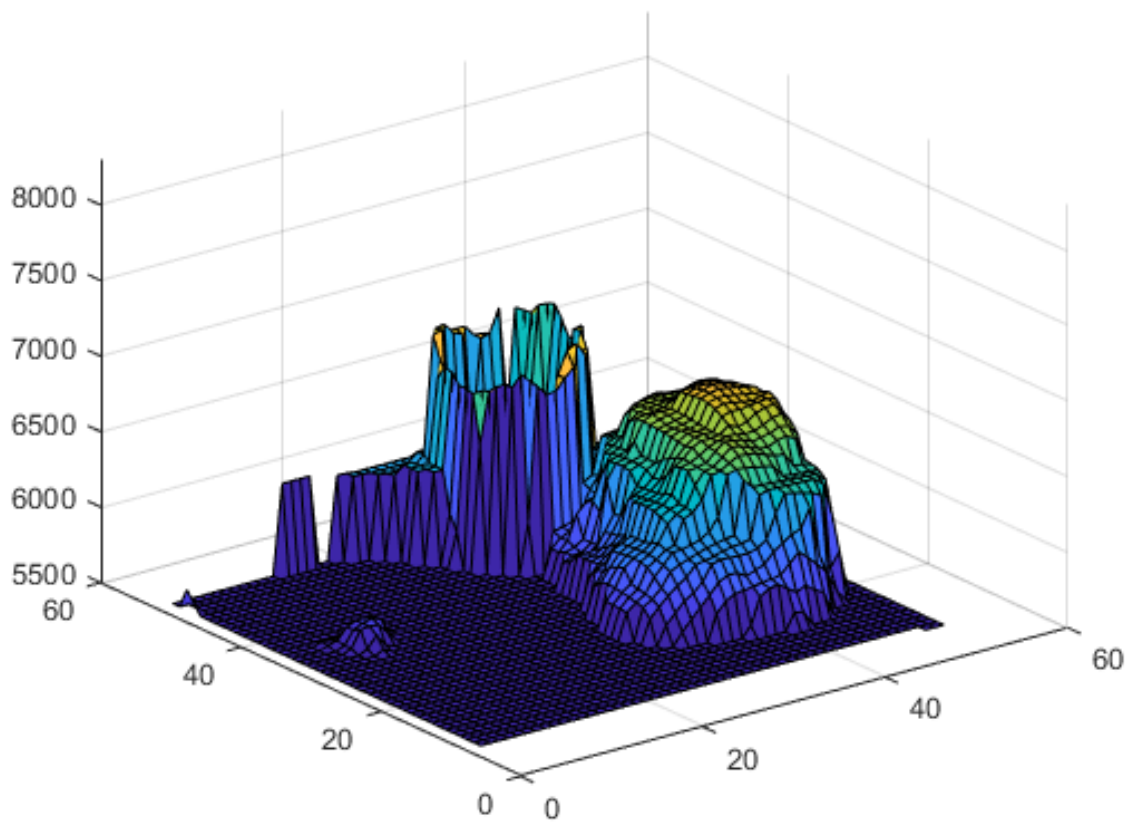
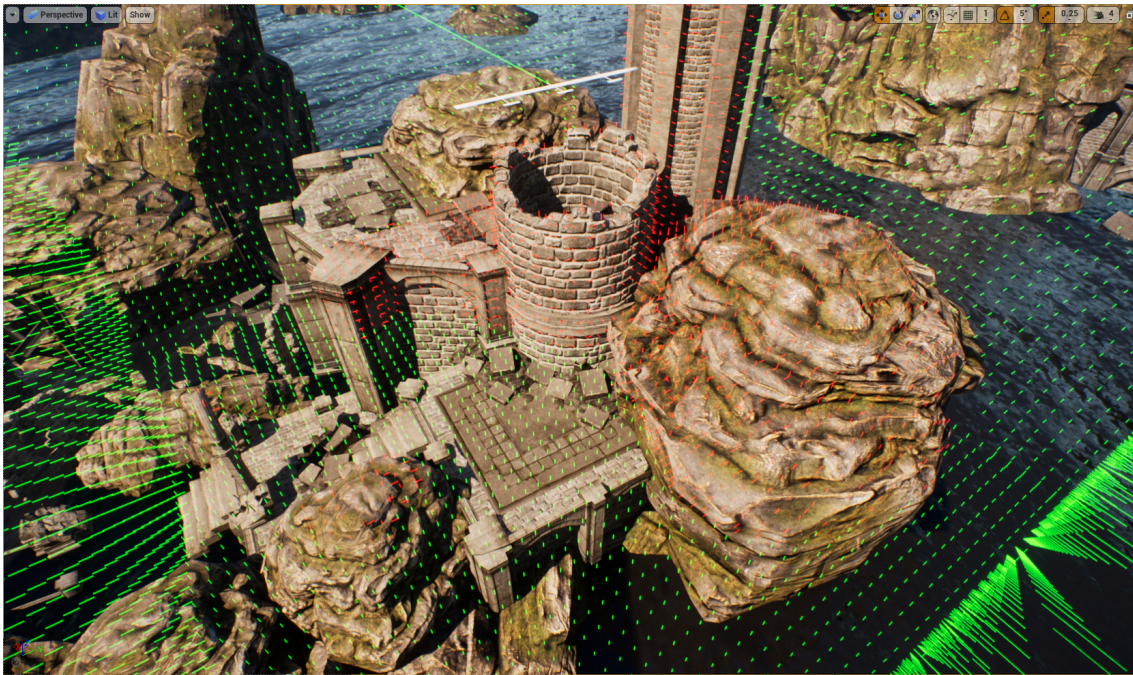


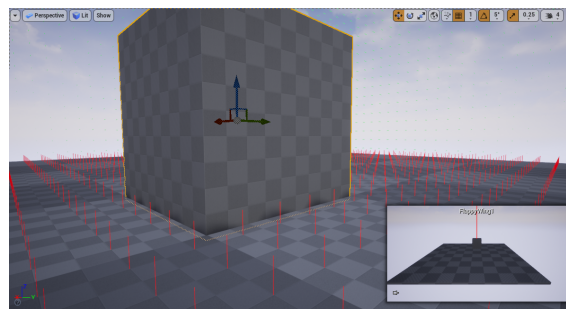
Figure 43: Surface plot of ruined tower scan data

5.4 Collision query accuracy

The main aim of this chapter is to further investigate whether the scanner data is reliable enough in order to be used for externally computed collisions, as an example of application. To this end, the XCD module is used to read the scanner data as six point clouds (the six 2.5D maps) and use a voxel map based on a simple cube mesh in order to detect collisions between said cube and the sampled environment around it. Two simple set-ups are used as test cases, one with the cube falling on a flat surface and one where the same cube falls onto the edge of an obstacle.

Falling on flat ground

Figure 45 showcases the behaviour of a cube falling on a flat surface through a series of runtime snapshots. As can be seen in figures 45a through 45e, the XCD module starts to sense collisions once the cube and the points overlap, however, since XCD computes buoyancy-like forces, some time passes before the cube starts rising. This behaviour would be almost accurate if the cube were to fall in water and float and this is expected due to the buoyancy force character of the computed XCD forces. Furthermore, after a certain time, the numerical instabilities affecting the torque acting on the cube take hold. The total torque should be zero seeing as the points are symmetrically distributed over the colliding cube face, but instead it has very small values that over time add up in a torque that starts rotating the cube. This behaviour is showcased in figures 45f and 45g. This however, is not a problem of the scanner, but an incomplete interfacing of the XCD module in terms of force and torque calculations. The module still reads the points when the voxel map meets the point cloud, namely when the cube and the flat surface overlap within the UE4 environment. This is supported by figures 44a and 44b which show that the number of points the box overlaps and the penetration distance shown in the UE4 snapshot are correctly detected within the XCD module output.



(a)

```
Penetration = 0.0200001
Number of points colliding = 25
Timestamp [s], t_comp [ms], F [N?], T [Nm?] = 27.657, 0.134033, [ -0, -0, 0.500003 ], [ -8.3819e-09, 5.58794e-09, -0 ]
```

(b)

Figure 44: Scanner points detected by XCD

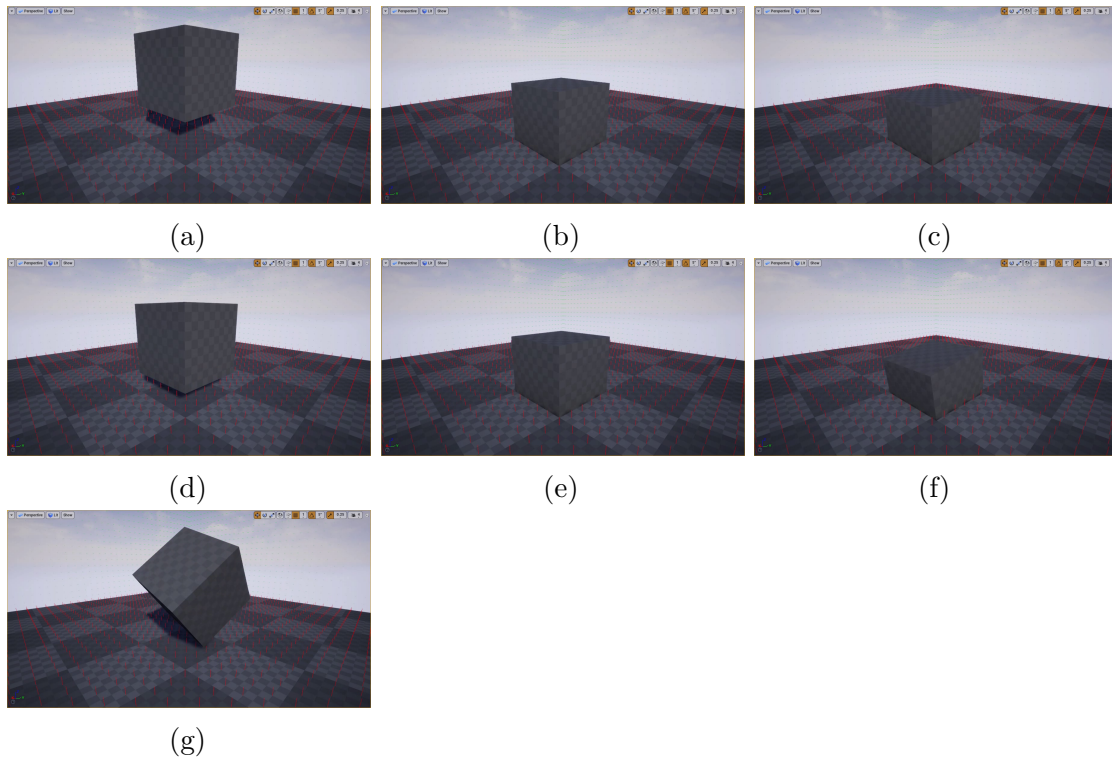


Figure 45: Runtime snapshot of a box falling onto a flat surface

Falling on a stair-like obstacle

Figure 45 showcases the behaviour of a cube falling on the edge of a surface through a series of runtime snapshots. While the contact is still very elastic due to the buoyancy-like forces and torques, figures 46a through 46f show an expected behaviour of the box beginning to rotate and slightly moving away from the stair upon contact. Figures 47a and 47b show the evolution of the force and torque acting on the cube throughout its motion. This also shows expected behaviour, where forces act on the Y and Z axes of the cube, while the torque rotates the cube on its X axis, in the direction of the gap. Figure 48 provides a numerical representation of how the orientation of the box evolves during its motion.

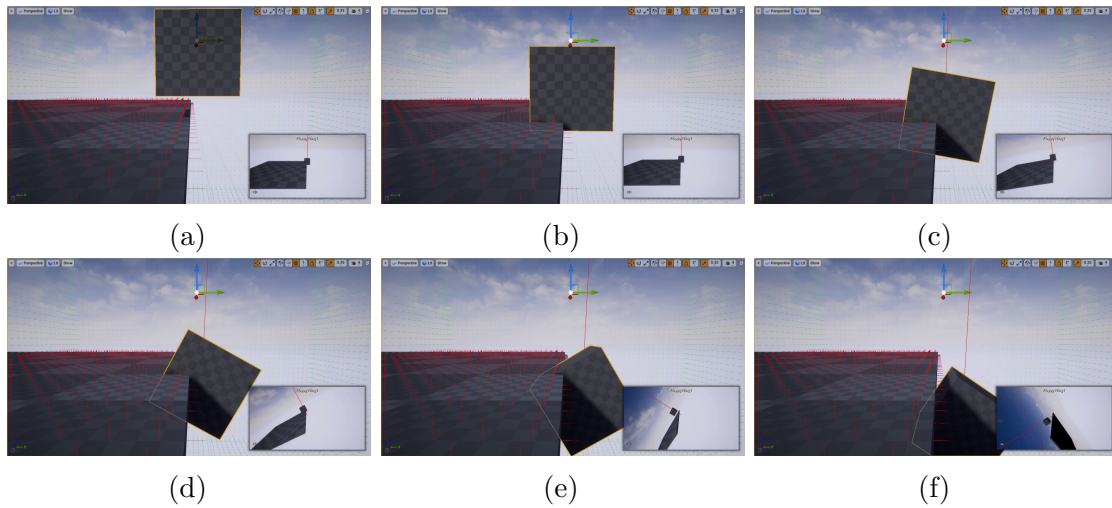
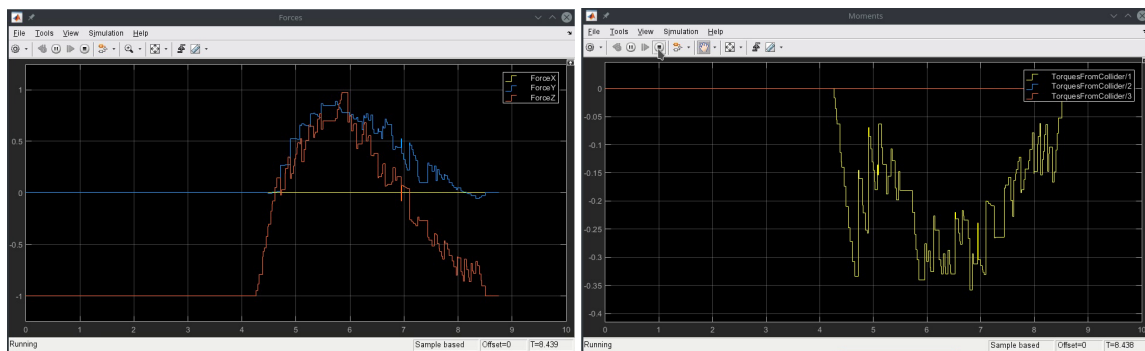


Figure 46: Runtime snapshots of a box falling on an obstacle edge



(a) Stair force profile plot

(b) Stair torque profile plot

Figure 47: Box forces-torques profile plots upon contact with obstacle edge

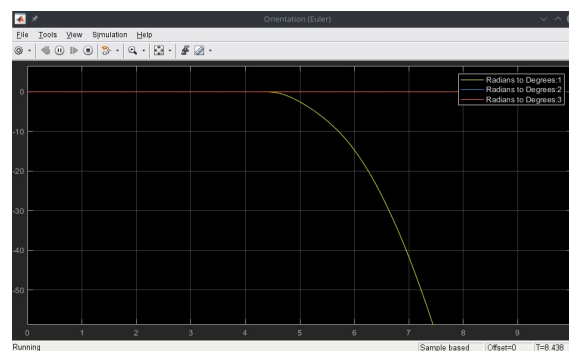


Figure 48: Box orientation profile plot upon contact with obstacle edge

6 Conclusions

Following the completed implementation chapter 4 and evaluation chapter 5 of the proposed simulation and visualization platform, a number of conclusions can be drawn with regard to the feasibility and validity of using game engine within the area of robotics research and development.

First and foremost, choosing to implement the proposed platform within a game engine environment has clear advantages when it comes to representing and manipulating both the robot and the environment around it, due to the scope of application of a game engine.

Secondly, while Unreal Engine 4 comes with a high learning curve, the functionality it comes with more than compensates for the time spent understanding its systems and interfaces. Once the basis of this simulation and visualization platform exists, adding more modules and functionality to it becomes a clear process. Moreover, having access to the core of the platform and being able to extend it to any robotic system, environment and application scenario gives a great deal of flexibility and accessibility when it comes to simulating different robotic systems in various scenarios.

Thirdly, being able to use the game engine with external modules such as physics simulations of any kind poses an invaluable advantage to using this approach within the development process of a robotic system, due to not having to rely on the UE4 internal physics engine. This means one can use this platform with already existing simulation based on MATLAB/Simulink, C++, etc. without having to adapt the simulation software to a new physics engine or - when it comes to aircraft - to external FDMs. Furthermore, the frame rates showcased by the evaluation in chapter 5.2 enable fluid visualization and valid environment sampling, while a more powerful computer would reduce this GPU bottleneck and allow for an even higher achievable frame rate.

Last but not least, the environment sampling functionality showcased and evaluated in this work already provides complete access to scanning and interacting with the environment around a robot system. Being developed as a component, it can be redeployed in a fast and easy way to any new robotic systems implemented within the platform. Moreover, its resolution can be changed in order for the scanner component to offer more reliable data to a more sensitive robotic system, while the default one meter resolution already shows great results, as showcased by the evaluation in chapter 5.3.

All in all, undertaking simulation and visualization through a game engine proves to be a good approach to visually representing the robots behaviour in the world while also enabling a very important addition to the simulation loop in the form of taking robot-environment interactions into account. While work is still required towards externally simulating these interactions, the platform itself is already in

a ready-to-use state when it comes to visualizing the robot, receiving input from external physics modules, representing a high-quality world, as well as providing sampled environment data to an external module.

7 Future Work

While the proposed simulation and visualization platform is in a ready-to-use state, incentives for follow-up research exist. While the most important next step is to finalize and validate the externally simulated interactions between the robot and the world based on sampled environment data as shown by the evaluation in [5.4](#), there are aspects of the platform itself that could be extended upon.

First and foremost, extending the platform to enable "multi-player" support would greatly benefit robotic research in applications where the focus is on robot-robot cooperation, such as robotic swarm technology. This not only entails adapting the middleware interface to support multiple Pawns, but also implementing a new environment sampling approach, since the current Scanner version is meant to scan static environments such as the world itself, as opposed to fast-moving objects.

Secondly, another game engine functionality known as terrain streaming should be investigated. This allows for a very big world to be split into pieces and each piece only loaded when the robot enters it, therefore drastically reducing the resource requirements for world terrain rendering. This is especially effective when applying the platform to aircraft or similar fast-moving robots which cover a great distance in a short time, and thus require a very big world rendered within a level.

List of Figures

1	ELAHA concept [DLR]	11
2	Potential HALE applications [DLR]	12
3	HELIOS [NASA]	12
4	ELAHA gust compliance [DLR]	13
5	ELAHA segmented design [DLR]	13
6	Top-level concept overview	23
7	Robot representation concept	24
8	World representation concept	25
9	Scanner sampling concept	25
10	Scanner sampling concept	26
11	UE4 Property coding	30
12	Top-level implementation overview	33
13	<i>ModelMesh</i> property in editor view	34
14	Meshes in UE4 editor view	35
15	Key link	35
16	Keyboard input set-up	35
17	UE4 overview of ELAHA bones	36
18	Built world	38
19	Near-far views in the custom world	39
20	Mesh preparation for scanning concept	39
21	Current cameras	40
22	Initial ray-trace	41
23	Ground scanner	41
24	Box scanner	42
25	Face Scanner applied to the custom world	43
26	Links and Nodes manager GUI	44
27	Links and Nodes manager topics	45
28	Links and Nodes I-D chain sample code	47
29	Links and Nodes Update-Send functionality sample code	47
30	XCD output	49
31	Frame rates for Pawn in empty level	51
32	Frame rates for Pawn in empty level with scanner	51
33	Frame rates for Pawn in empty level with scanner and debug	52
34	Frame rates for Pawn in custom level	53
35	Frame rates for Pawn in custom level with scanner	53
36	Frame rates for Pawn in custom level with scanner and debug	54
37	Ground scanner scanning a cone primitive	56
38	Face Scanner scanning a cone primitive	56
39	Face Scanner applied to the custom world	57
40	Surface plot of water scan data	58
41	Surface plot of small rock scan data	59
42	Surface plot of big rock scan data	60
43	Surface plot of ruined tower scan data	61

44	Scanner points detected by XCD	62
45	Runtime snapshot of a box falling onto a flat surface	63
46	Runtime snapshots of a box falling on an obstacle edge	64
47	Box forces-torques profile plots upon contact with obstacle edge	64
48	Box orientation profile plot upon contact with obstacle edge	64

References

- [1] KUKA, https://www.kukakore.com/wp-content/uploads/2012/07/KUKA_LBR4plus_ENLISCH.pdf, Last accessed: 2018-08-07.
- [2] Deutsches Luft- und Raumfahrtzentrum, <https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-11427/#gallery/29202>, Last accessed: 2018-08-07.
- [3] MathWorks, <https://se.mathworks.com/products/simulink.html>, Last accessed: 2018-05-07.
- [4] Coppelia Robotics, <http://www.coppeliarobotics.com>, Last accessed: 2018-06-15.
- [5] FlightGear Flight Simulator, <https://home.flightgear.org>, Last accessed: 2018-06-03.
- [6] FlightGear Flight Simulator, <https://home.flightgear.org/about>, Last accessed: 2018-06-03.
- [7] XPlane, https://www.x-plane.com/desktop/meet_x-plane, Last accessed: 2018-06-03.
- [8] XPlane, <https://developer.x-plane.com/manuals/planemaker>, Last accessed: 2018-06-03.
- [9] XPlane, <https://www.x-plane.com/desktop/how-x-plane-works>, Last accessed: 2018-06-03.
- [10] MathWorks, <https://se.mathworks.com/products/3d-animation.html>, Last accessed: 2018-02-16.
- [11] Wikipedia, <https://en.wikipedia.org/wiki/X3D>, Last accessed: 2018-02-16.
- [12] Wikipedia, <https://en.wikipedia.org/wiki/VRML>, Last accessed: 2018-02-16.
- [13] Leibniz Supercomputing Center, https://www.lrz.de/services/v2c_en/, Last accessed: 2018-06-29.
- [14] Leong S. H. , C. Anthes, F. Magnon, A. Spinuso, E. Casarotti. Advance Visualisation of Seismic Wave Propagation and Speed Model. In inSiDE, Vol. 13, No. 1, p. 34-37, Spring 2015.
- [15] Crytek, <https://www.cryengine.com>, Last accessed: 2018-01-30.
- [16] Crytek, <https://www.cryengine.com/features>, Last accessed: 2018-01-30.
- [17] Amazon, <https://aws.amazon.com/lumberyard>, Last accessed: 2018-01-10.

- [18] Wikipedia, https://en.wikipedia.org/wiki/Amazon_Web_Services, Last accessed: 2018-01-10.
- [19] Amazon, <https://aws.amazon.com/gamelift/>, Last accessed: 2018-01-10.
- [20] Epic Games, <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>, Last accessed: 2018-07-19.
- [21] Shah, S. and Dey, D. and Lovett, C. and Kapoor, A. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. *Field and Service Robotics* 2017, <https://arxiv.org/abs/1705.05065>.
- [22] Deutsches Luft- und Raumfahrtzentrum, <https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-12517/#gallery/29260>, Last accessed: 2018-07-10.
- [23] Bender, S. and Cole, J. and Herzeg, I. Presentation: Enhancing Mocap Data with Procedural Systems. *CitizenCon 2017*, 2017.
- [24] United Nations Office for Outer Space Affairs. Online Index of Objects Launched into Outer Space, 2018, http://www.unoosa.org/oosa/osoindex/index.jsp?lf_id=. Last accessed: 2018-07-31.
- [25] Union of Concerned Scientists. UCS Satellite Database, 2018, <https://www.ucsusa.org/nuclear-weapons/space-weapons/satellite-database#.W2BeHbizJhE>. Last accessed: 2018-07-31.
- [26] Wikipedia, https://en.wikipedia.org/wiki/2009_satellite_collision, Last accessed: 2018-04-23.
- [27] European Space Agency, http://www.esa.int/Our_Activities/Space_Engineering_Technology/Clean_Space/e.Deorbit, Last accessed: 2018-04-23.
- [28] Noll, T. and Brown, J. and Perez-Davis, M. and Ishmael, S. and Tiffany, G. and Gaier, M. Investigation of the Helios Prototype Aircraft Mishap, 2004.
- [29] Wlach, S. and Balmer, G. and Germann, M. and Wüsthoff, T. ELAHA - Elastic Aircraft for High Altitudes. *Proceedings of the 23rd ESA Symposium on European Rocket and Balloon Programmes and Related Research (ESA PAC)*, 2017.
- [30] Deutsches Luft- und Raumfahrtzentrum, https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-11678/20400_read-47718/, Last accessed: 2018-07-11.
- [31] Epic Games, <https://docs.unrealengine.com/en-us/Platforms/Linux/BeginnerLinuxDeveloper/SettingUpQtCreator>, Last accessed: 2018-07-13.

- [32] Drew Noakes, <https://github.com/drewnoakes/joystick>, Last accessed: 2018-03-08.
- [33] Adloff, V. Master Thesis. Development of a Visualization Environment for Robotic Systems using Game Engine Technology in the Context of Highly Elastic UAVs.
- [34] Bechtel, N. Master Thesis. Implementierung einer echtzeitfähigen Simulation eines hochelastischen Fluggeräts mithilfe vereinfachter Modellierungsmethoden.
- [35] Sagardia, M and Turrillas, M. and Thomas, H. Realtime Collision Avoidance for Mechanisms with Complex Geometries. *Proceedings of the IEEE Virtual Reality 2018*, Reutlingen, Germany.