University of Bremen

TRIPLESIM - UNDERWATER EXPLORATION OF AN ICY MOON IN A VIRTUAL ENVIRONMENT

REPORT ON A MASTER PROJECT

from

Christian Hoffmann
Darius Schlese
Niklas Stockfisch
Nusrat Jahan Tamanna
Maurice Zerreßen

10. April 2024

Contents

List of Figures

Glossary

AUV An AUV, or Autonomous Underwater Vehicle, is a self-propelled robotic vehicle designed to operate underwater without direct human control. It is equipped with sensors, navigation systems, and often a payload for various tasks such as oceanographic research, underwater mapping, or surveillance. 1, 5–16, 22–25, 28, 34, 35, 37–43, 47, 55, 57–62, 67, 72, 73, 82

Blender Blender is a open source 3D computer graphics software tool. This can be used to create 3D models, for example.. 28, 29, 36

CSS CSS, or Cascading Style Sheets, is a style sheet language used to define the presentation and layout of HTML documents. It allows developers to control the appearance of web pages by specifying styles such as colors, fonts, margins, and positioning. CSS works by selecting HTML elements and applying styling rules to them, enabling the creation of visually appealing and responsive web designs. 88

GIMP "GNU Image Manipulation Program, commonly known by its acronym GIMP, is a free and open-source raster graphics editor used for image manipulation (retouching) and image editing, free-form drawing, transcoding between different image file formats, and more specialized tasks." [41]. 28

Go Go, also known as Golang, is a statically typed, compiled language developed by Google. It emphasizes simplicity, efficiency, and concurrency, making it well-suited for building scalable and reliable software systems. 86

GrapesJS GrapesJS is an open-source web builder framework that allows developers and designers to create responsive web pages visually. It provides a drag-and-drop interface for building web pages without requiring coding skills, making it accessible to a wide range of users. 86

HTML HTML, or Hypertext Markup Language, is the standard markup language used for creating web pages and web applications. It provides the structure and content of web pages through a series of tags and elements. HTML elements are used to define the different components of a webpage, such as headings, paragraphs, links, and images. 86, 88, 89

Hugo Hugo is an open-source framework for static site generation. 86

IMU An Inertial Measurement Unit (IMU) is a device that typically combines multiple sensors, such as accelerometers, gyroscopes, and magnetometers, to measure and report on an object's orientation, velocity, and gravitational forces. It is commonly used in various applications such as navigation systems in aircraft, drones, smartphones for gesture recognition, and virtual reality systems for motion tracking. IMUs provide crucial data for maintaining stability, determining position, and enabling accurate movement tracking in dynamic environments.. 21, 63, 64, 67

JavaScript JavaScript is a high-level, interpreted programming language primarily used for web development. It allows developers to create dynamic and interactive content within web browsers. JavaScript is versatile, supporting both object-oriented and functional programming paradigms, and is essential for creating modern web applications. 86, 88

Markdown Markdown is a lightweight markup language commonly used for formatting plain text documents. It allows users to easily create formatted text using simple syntax, such as using asterisks for emphasis or pound signs for headings. Markdown is widely used in various contexts, including writing documentation, creating web content, and collaborating on projects. 86, 88

Niagara Niagara is the current particle system of Unreal Engine. 16, 18–21, 29–32, 34, 36, 37, 39, 40, 43–45, 47, 48

Quixel Bridge "Quixel Bridge plugin for Unreal Engine gives you full featured access to the Megascans library within the Level Editor. You can browse collections, search for specific assets, and add assets to your Unreal Engine projects. "[11]. 28, 49

RVT "A Runtime Virtual Texture (RVT) creates its texel data on demand using the GPU at runtime, and works similarly to traditional texture mapping. The RVT caches shading data over large areas, making them a good fit for Landscape shading that uses decal-like materials and splines that are well suited to conform to the terrain." [12]. 52, 53

ToF The Time of Flight is the time between sending a ping and receiving a ping. 5, 8, 9, 11

TRIPLE The TRIPLE (Technologies for Rapid Ice Penetration and subglacial Lake Exploration) project seeks solutions to the problem of how to conduct scientific studies of subglacial lakes on other planets. 88

UI User Interface. 21, 62, 63, 65, 70–72

UMG Unreal UMG (Unreal Motion Graphics) is a robust visual UI authoring tool within the Unreal Engine game development environment. It allows developers to design and create user interfaces using a node-based system and a variety of widgets, such as buttons, text boxes, and images. 63–65, 68

Unreal Engine Unreal Engine is a 3D computer graphics engine that is primarily used for game development and simulation purposes. The current version is Unreal Engine 5. 1, 5, 7, 9, 11, 12, 15, 21, 27–29, 31, 36, 37, 45, 49, 57–61, 71, 72, 82–85

USBL USBL (Ultra-Short Baseline) is a type of underwater acoustic positioning system used to track the location of submerged objects or vehicles in real-time with high precision. It operates by measuring the time it takes for acoustic signals to travel between a transceiver on the surface and a transponder attached to the underwater object. 5–7, 11–14, 61, 62, 67, IV

VaMEx The VaMEx project, short for Valles Marineris Explorer, searches for a swarm-based exploration approach for future mars missions. For this goal, several autonomous drone units work together in the same network, consisting of rovers, crawlers and unmanned aerial vehicles. 1, 11, 14, 15, 21, 58, 88

VaMEx-VTB The VaMEx-VTB is a subproject of the VaMEx project with the goal to firstly provide a test framework for swarm-based exploration systems and secondly contain tools for evaluation. To achieve this, a communication framework between Unreal Engine 5 and ROS has been implemented. 57, 58

1. Introduction

The German Aerospace Center (DLR) is planning an expedition to one of Saturn's moons, the icy moon Encelauds, for which the Triple project was founded. Triple stands for Technologies for Rapid Ice Penetration and subglacial Lake Exploration. A large part of the project is the development of a melting probe that can melt through layers of ice to reach a potential lake. The melting probe will deploy an AUV, which will take a series of samples that will then be analyzed. The AUV is also being developed as part of the triple project. The AUV should travel around the ocean/sea, take samples and analyze the environment, the AUV should return to the melting probe/base station after a certain time and deliver the information. The goal of this project is to simulate such a mission. Simulations are often used to test various aspects of such a mission, e.g. the amount of energy required, the type of sensors or the general behavior in certain scenarios.

To develop such a simulation, we are building on the work of the VaMEx project and we use the Unreal Engine. The VaMEx project, short for Valles Marineris Explorer, searches for a swarm-based exploration approach for future Mars missions. For this goal, several autonomous drone units work together in the same network, consisting of rovers, crawlers and unmanned aerial vehicles. The VaMEx-VTB is a subproject of the University of Bremen with the goal to firstly provide a test framework for swarm-based exploration systems and secondly contain tools for evaluation. To achieve this, a communication framework between Unreal Engine 5 and ROS has been implemented, which will also be used in this project.

We simulated different sensors of the AUV and other aspects of the environment such as an ocean current or black smokers. We also simulated part of the environment of Enceladus based on existing information, but since Enceladus is unknown, we can only make assumptions and do it similarly to the existing examples on Earth, e.g. Antarctica. To simulate the different sensors, we used existing research and real sensors as a basis.

In this report, we will first explain the different sensors and their implementation, discuss the results and the problems we encountered during the development. Then we will explain and discuss the simulation environment. Finally, we will explain the ROS system, give an overview, explain some components and discuss our changes and problems encountered during development. We will also explain how to set up the simulation and how to control the user interface of the simulation. We will also introduce our website for the simulation.

1.1 Project goals

We want to create a simulation that simulates various aspects of such a mission. Our goal is to create largely physically correct sensors, an environment that could be similar to that of Enceladus, and an overall good-looking simulation to attract people for the project.

1.2 Organization

In this section, we will explain our internal organization and project management. We discuss our milestones and how we have managed the project overall.

1.2.1 Milestones

We created three Milestones for the project.

The first was a vertical slice, having some sensors, a basic level, other features such as the input system, multiple camera views and the ROS2 system integrated.

The second milestone included all sensor implementations and visualizations and a complete terrain and environment and features such as the ocean current map and battery simulation.

The third and last milestone is the finished project, which includes more polishing/adjustments of the already implemented features, the complete demo, website and report.

1.2.2 Meetings

We had group meetings with the supervisors once a week. Furthermore, we had at least one meeting without the supervisors every week, but also additional meetings depending on the current tasks and how much we had to discuss.

1.2.3 Protocols

For each meeting a protocol was created to help document things that were discussed and decisions that were made. The protocol writer changed on a weekly basis in a round-robin.

2. Overview of TripleSim

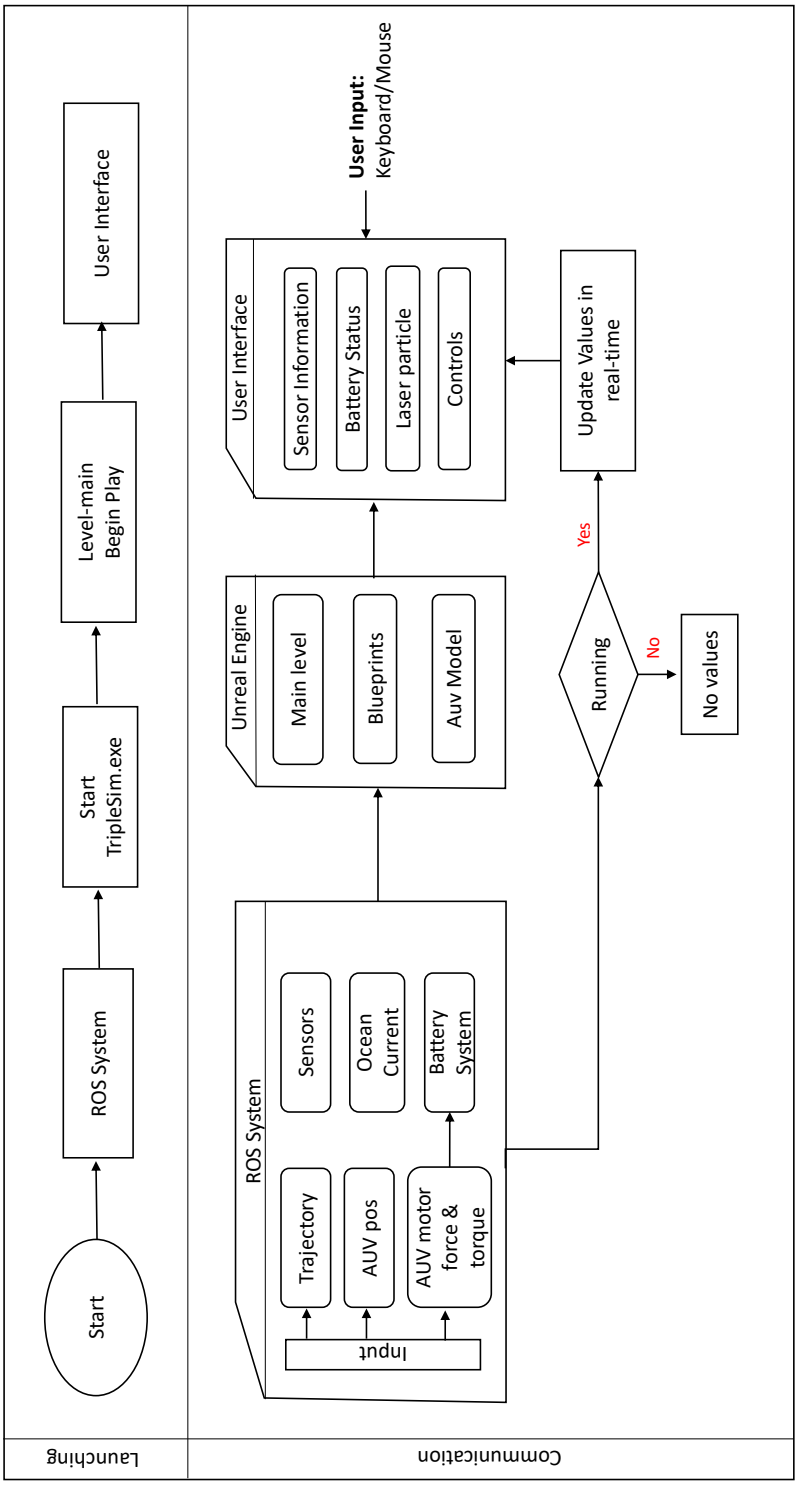

Figure 2.1: Overview of TripleSim

This document outlines the basic concept of running the simulation. Its purpose is to provide clear guidance for initiating the project and observing the simulation and interaction through the UI. This will facilitate smooth usage for new users in the future.

All installation and execution instructions are provided in the README files located within each subfolder corresponding to different parts of the project:

• To install 'triplesim' dependencies, refer to the triplesim README.

• To install Marum ros ws dependencies, refer to the Marum ros ws README.

We have successfully installed all the required dependencies for both Windows 10 and 11 during our project. Once all dependencies are correctly installed, the steps are as follows:

1. Start the Docker.

2. From the command shell in Ubuntu, launch the ("run uv sim.bash") file located in the (.."/triplesim/Marum ros ws) folder. This will initiate the ROS system.

3. After starting the ROS system, launch the tripleSim project.

4. Our main level is named "Demo". Launching this level will initiate the simulation.

5. A separate window will open with controls provided. Sensor information and battery status will be updated there and can be checked by clicking on each tab.

6. It's essential to start the ROS system before initiating the simulation. Otherwise, there will be no simulation or information available in the game mode.

For more detailed information, refer to the subsequent chapters in this report.

3. Sensors

In the following sections we will explain different types of sensors that we use for our simulation, we will explain the details of the implementation and discuss the results.

3.1 USBL

The AUV consits of several sensors that are used to automatically control the AUV, one of these Sensors is the Ultra Short Baseline (USBL) system. The USBL is a method for acoustic underwater positioning. In our case, the USBL consists of two transceivers, one is mounted on the AUV and the other one on the base station. One of these transceivers would therefore send a ping into the ocean and the other would wait until it receives this ping (see Figure 3.1). Based on a phase shift, which can be calculated in the hydrophones of the transceiver, and the time difference of the impact, the transceiver that detected the ping can calculate a direction from the ping. The transceiver would then take the calculated direction and send another ping in this calculated direction. This ping would be received by the original transceiver that sent the first ping. Using the direction calculated from the phase shift and the time of impact of each hydrophone, the transceiver could calculate the direction, and using the time of flight (ToF), which is basically the time elapsed between the sending of the first ping and the arrival of the second ping, the transceiver can calculate a distance, via the speed of sound in water and the time of flight.

$$PrePoint = cLoT + Direction \cdot Distance \tag{3.1}$$

$$Distance = Speed * ToF \tag{3.2}$$

Eq. 3.1 & 3.2: PrePoint is the calculated position that results from the current location of the transceiver (cLoT), the direction of the beam and the calculated distance. The distance is calculated using the speed of sound in the water and the time of flight (ToF)

In the following, we will discuss the challenges that need to be overcome, the various implementation approaches and finally the results of the USBL.

3.1.1 Challenges

One of the main challenges for the USBL was the simulation of sound waves in the Unreal Engine, there was already some research on this topic [28, 7]. These papers each presented a way to simulate the sound waves in Unreal Engine, they use multiple rays and form them into one bundled beam. The rays of this beam would update there position every frame. Since a sound wave loses power over time as it travels through a medium that depends on the density of this medium. In addition, the wave would lose some of its power and be reflected when hitting surfaces or objects. To simulate these aspects of the sound wave, we did it similarly to the paper "Development of a Simulation Environment for Evaluation of a Forward Looking Sonar System for Small AUVs" from Morency et al.[28]. The paper from Morency et al. [28] also introduced several parameters for the environment, which we modified to fit our potential environment.

Another challenge would be to simulate the transceiver of the USBL, the paper "Deep-Sea Model-Aided Navigation Accuracy for Autonomous Underwater Vehicles Using Online Calibrated Dynamic Models" from Oertel et al. [32] presented a method how to simulate such a USBL. Five hydrophones per transceiver are used in our simulation, and they are similarly placed as described in the paper from Oertel et al. [32]. Each hydrophone consists of a bounding box to detect the incoming rays. Each transceiver consists of five hydrophones and is capable of emitting a bundled beam of rays in a specific direction. In our simulation there are two transceivers, one for the AUV and one for the base station.

The final challenge is to balance the accuracy of the USBL with the performance of the simulation, for example the number of rays per bundled beam, the size of the hydrophone bounding boxes, the ping rate of the USBL et cetera.

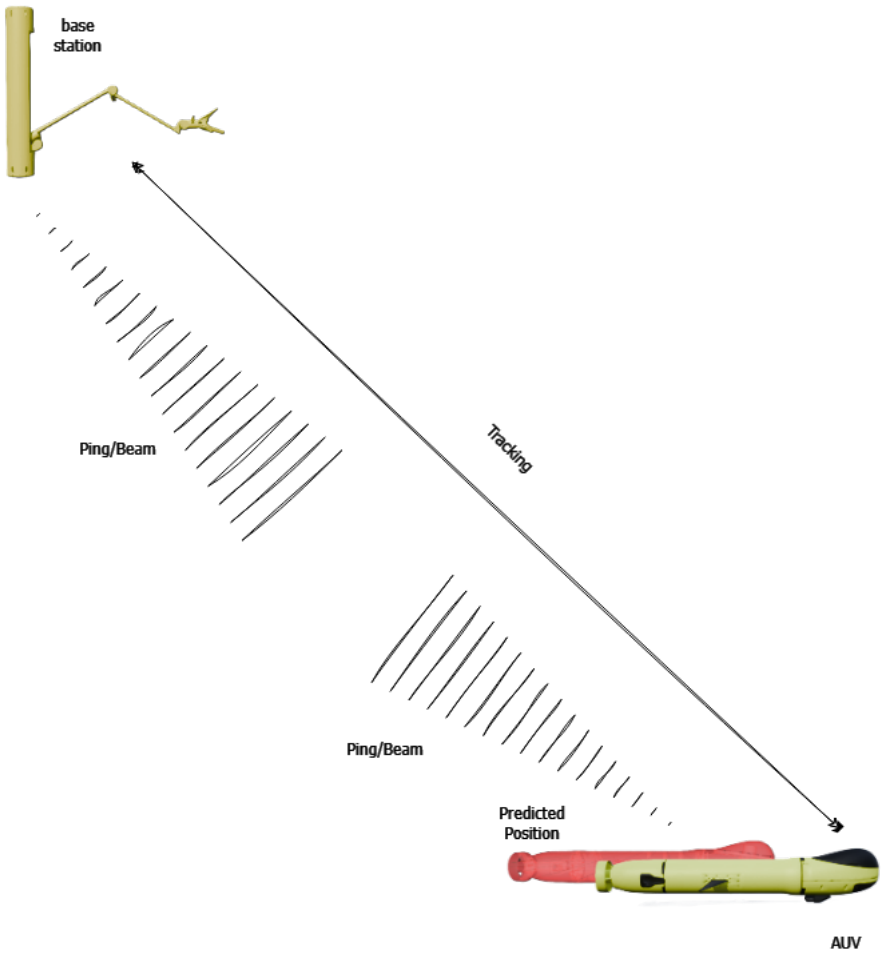

Figure 3.1: Sketch of the communication of the USBL

3.1.2 Implementation

In the following section, we will first explain the current implementation and then discuss other methods we have tried and explain why we do not use them.

3.1.2.1 Current Implementation of the transceiver

An Unreal Engine actor class named "EchoBeam" was created for the transceiver part of the USBL, and another Unreal Engine actor class named "EchoRay" was created for the rays. In our simulation, the base station served as the primary transmitter, which means that the AUV is always waiting for rays. The AUV "EchoBeam" only transmitted a beam upon receiving rays, although this behavior could be easily modified with the "isAUV" boolean parameter. This Boolean parameter would deactivate the initial transmission function and make the "EchoBeam" wait until it has received a few rays, per default this boolean is deactivated at the base station and activated at the AUV.

The "EchoBeam" consists of a center point and five bounding boxes. The bounding boxes are placed similarly as described in the paper "Deep-Sea Model-Aided Navigation Accuracy for Autonomous Underwater Vehicles Using Online Calibrated Dynamic Model" by David Oertel [32], as they represent the hydrophones. Each "EchoBeam" also calculate the source level of the sound wave based on the values of a similiar USBL [14].The bounding boxes are scaled at each frame based on the euclidean distance between the base station and the AUV to reduce the number of rays and increase the overall performance of the simulation.

The base station's "EchoBeam" also has a timer that sends a beam/ping every ten seconds, but this timer can be blocked if another sensor that uses this type of beam is currently running, for example the echo sounder (see 3.2), this is due to some performance issues we had on our local computer. When the "EchoBeam" is allowed to send a beam, the beam is constructed, for this the "EchoBeam" spawns multiple rays, with the spawn points based on polar coordinates around the center point. The orientation of the rays is also based on the type of "EchoBeam". The direction of the "EchoBeam" of the base station is always upwards (0,0,1) and can be changed by rotating the "EchoBeam", and the orientation of the AUV is based on the calculated orientation from the incoming rays. If the "EchoBeam" of the base station has already predicted a direction, it automatically turns in this direction and sends the next beam in this direction, but if it does not receive a new direction in a certain time, it sends the next beam upwards again (0,0,1). The "EchoBeam" of the base station is rotated by 180 degrees so that the original upward direction (0,0,1) points downwards. The opening angle of the beam can be set via a lower and upper boundary for the polar and azimuth angle. By default, the "EchoBeam" of the base station has a lower polar boundary of 100 degrees and an upper boundary of -100 degrees, the AUV has a lower polar boundary of -60 degrees and an upper boundary of 60 degrees. The azimuth for both would be 360 degrees in the upper boundary and 0 in the lower boundary. The spherical coordinates were then calculated based on the polar coordinates. A vector was created based on these spherical coordinates, and this vector was rotated according to the rotation matrix of the current rotation of the "EchoBeam". The resulting vector served as the direction for the spawned ray. The "EchoBeam's" also set some parameters for the rays, e.g. the source level, which is called "TransmittedPower" in our simulation or a reference of the EchoBeam that generated these rays, this is used for collision detection in the rays.

In a real USBL, the hydrophones would typically wait until they were hit by the sound wave. In our simulation, however, the opposite approach was taken, for reasons that will be discussed later in section 3.1.4. In our simulation, the rays will do the collision detection and set some parameters in an array for the "EchoBeam". The "EchoBeam" receives

the time at which the rays collide with a bounding box, the impact point on the bounding box, the name of the ray and the current endpoint of the ray, these values will be inserted into a array, the structure of the array is the following *Array<Tuple<String, Vector, float, Vector>,...>*. The rays will create a new tuple if they impact with the "EchoBeam" and will insert it into this array as an entry. The "EchoBeam" waits until it has some values in this array. After the first entry, the "EchoBeam" waits 1 second to receive further possible rays.

After this second the "EchoBeam" starts calculating the predicted position, for this the array is sorted by the ray names, so the array structure would look something like this *[Ray1[Ray1BB1Tupel, Ray1BB2Tupel, Ray1BB3Tupel, Ray1BB4Tupel,..], Ray2[...], ...]*.

This was performed to simulate a kind of phase shift, as afterward, a loop was made over this array to sort each element (where the elements represented arrays of the rays, such as *[Ray1BB1Tupel, Ray1BB2Tupel,....]*) based on the impact time. Subsequently, a direction was calculated for each ray by extracting the maximum and minimum impact times and computing an average impact point. This involved normalizing each impact point based on the impact time and calculating the average impact point of a ray. Then, the direction between the average impact point and the last impact point of each ray was calculated (See equation 3.3 and 3.4). Once each ray had a direction, an average direction was calculated. The ToF was determined as the smallest minimum impact time among all rays. Following the acquisition of direction and ToF, a position was calculated. Additionally, zero mean Gaussian white noise was added to a random axis of this position, and an error correction was made for the size of bounding boxes by simply adding the X value of the bounding box extent to the depth. The AUV's "EchoBeam" then sent another beam in this direction, and the base station created a spawn point at this position, using the AUV model with a glowing red color as the spawn point. This spawning point does not decay, so it can be used to indicate different positions or paths per cycle.

$$averageImpPntPerRay = \sum_{i=0}^{n} (ImpPnt(x_i) \cdot (1 - \frac{time(x_i) - minTime}{maxTime - minTime})) \quad (3.3)$$

$$rayDirection = averageImpPntPerRay - GetNormal(ImpPnt(maxTimeRay(x))) \quad (3.4)$$

Eq. 3.3 is the calculation of the average impact point per ray, where each normalized impact point, on the bounding boxes, of a ray is summed, its bassically the normalized sum. In Equation 3.4, the direction is calculated using the calculated average impact point and the impact point on the bounding box with the maximal impact time.
In Eq. 3.3 *x* is the array per rays (for example *x = [Ray1BB1Tupel, Ray1BB2Tupel,....]*), *n* is the size of this array, minTime and maxTime are the maximum and minimum impact time values of x. maxTimeEntry(x) is the entry which has the maximum impact time

The decision was made to use a timer for each ray instead of a timer at the "EchoBeam" because several problems have occurred with the other method, which is based on the accuracy of time. The biggest problem was that the timer would start counting when the beam was constructed, but then the timer was too early and started counting before the first rays could move and setting the timer after the construction of the beam we had the problem, that the timer was later than the timer of the rays and the real ToF, and

therefor inaccurate. This could possibly be fixed by creating a synchronized timer that is synchronized with the rays, but for simplicity we decided to use the timer in the rays because we use the timer anyway. Also the timer error was only in the millisecond range at the "EchoBeam's", but the error of the timer of the rays was just smaller and therefor more accurate.

3.1.2.2 Current Implementation of the Rays

For the current implementation of the rays we used the paper "Development of a Simulation Enviroment for Evaluation of a Forward Looking Sonar System for Small AUV's" from Morency et al. [28], this paper presented some formulas for the simulation of sonar underwater. We use the formulas for the Sound Velocity, Attenuation, Volumen Backscatter, Bottom Backscatter and the formula to calculate the Reflected Energy from a Object based on the Target Strenght and the Reflected energy from the sea bottom via the Bottom backscatter.

As values (see Table 3.1) for these formulas we use typical values from the arctis and some other typical values for sound waves and transducers:

Table 3.1: The default ray parameter

Parameter Type	Value
Temperatur	-1.8 degree
Salinity	34.6 psu
Frequency	180 kHz
Transducers vertical length	1 cm
Transducers horizontal length	1 cm
Enceladus water depth	10 km

The attenuation and volume backscatter are calculated in each frame, along with the current endpoint of the ray, which is essential for later collision detection. Additionally, a timer is incremented each frame, which is used for calculating the ToF of the rays. Considering that the remaining power of the beam is influenced by attenuation and volume backscatter, the current power is calculated for each frame (see equation 3.3) and checked to see whether this power is greater than 0. If this is not the case, the ray is destroyed; if this is the case, a collision check is carried out.

$$CurrentPower = OriginalPower - (Attenuation + VolumenBackscatter) \quad (3.5)$$

Eq. 3.5: The original power is the initial power of the ray, which is calculated by the "EchoBeam's", so the current power is the original power minus the attenuation and backscatter of the volume.

For the collision detection, the "LineTraceSingleByChannel" function from the Unreal Engine is used. This function receives a start and an end point as inputs and a collision channel with which it should check whether the line would collide with something between the start and end point. The origin position of the ray served as the start point, and the current end point is calculated per frame. "ECC_Visibility" was selected as the

collision channel, which includes all visible objects. Furthermore, the invisible bounding boxes were configured to be included in this collision detection by setting their property for this channel to true. It is possible to create your own collision channel, but for the sake of simplicity the channel "ECC_Visibility" was chosen.

When the line trace collided with an object, the type of object is checked, whether it is the environment or another "EchoBeam".

When the ray hits the seabed or the ice shelf, the ray would be reflected. For simplicity, it is assumed that each surface acts like a perfect mirror. The reflected energy is calculated according to the formula for calculating reflected energy from the seabed via bottom backscattering by Morency et al. [28], with "rock" selected as the bottom type, this formula is used for both the seabed and the ice shelf. After calculating the reflected energy and reflected direction, a new ray was created with its origin point set as the impact point. In addition, a Boolean parameter is set to indicate whether the ray is reflected. This is used for the echo sounder (see chapter 3.2), however, it could be considered that this could possibly be deleted, as the incoming direction of the rays was checked anyway in order to filter them out.

When the line trace function hit an "EchoBeam", the type of the "EchoBeam" was checked, whether it is the the base station, the AUV, or the echo sounder (see chapter 3.2). After that, the direction of the rays which impacted is checked. This is because hydrophones work as directional microphones in the real world, and to simulate this aspect, the direction of the rays is checked, which can filter out some of the rays based on their direction. For example, the AUV would be directed upwards so that rays coming from below can be ignored. It is checked which bounding box was hit, also a tupel is created which is used as an entry in the array that is used to calculate the position in the "EchoBeam". For the next frame the line trace function will ignore the bounding box, which was previously hit by the ray, such that the array contains only one hit per ray per bounding box. The impact point was set as the hitpoint in the array, and the impact time was determined by multiplying the rays timer by the time of the line trace function, which is an indicator at which position the line trace function has hit the object, 0 would be the start and 1 the endpoint. In addition, a live timer is enabled for the rays, which is implemented to increase the performance of the simulation, after the live timer expires, the ray is destroyed.

When neither the environment or another "EchoBeam" is hit but something else got hit, the ray is reflected, and the function to calculate reflected energy from an object based on target strength by Morency et al. [28] is used, as the target strength the value from a sedimentary rock, which was found in China [39], is used. This value is chosen because we assumed that we would probably come across some rock structures, like the black smoker or something else.

For debugging purposes, a function named "UpdateRayLine" was also created, which could display all rays in the world. However, this function is not activated by default.

3.1.3 Results

With the current implementation, it is possible to generate about 500 rays per beam, which is due to the fact that "LineTraceSingleByChannel" uses the CPU instead of the GPU. The accuracy depends on the angle between the AUV and the base station and the distance. When the AUV is close to the base station, the position of the predicted point and the

AUV only differs by a few centimeters, but the largest errors occur when the AUV is close to the edges of the beam and far away from the base station, fewer rays hit the bounding boxes of the AUV and the size of the bounding boxes depends on the Euclidean distance, so some rays hit the boundaries of the bounding boxes, which may be far from the original AUV position, but this could easily be fixed with more computing power and possibly collision detection via the GPU instead of the CPU, so more rays could be used and the bounding boxes could be smaller. Also an error correction for the Z and Y axis could be implemented, similar to the correction applied to the X axis (which is simply calculating $Distance + (BoxExtent.X \cdot 2)$), so this would be another approach to fix this error.

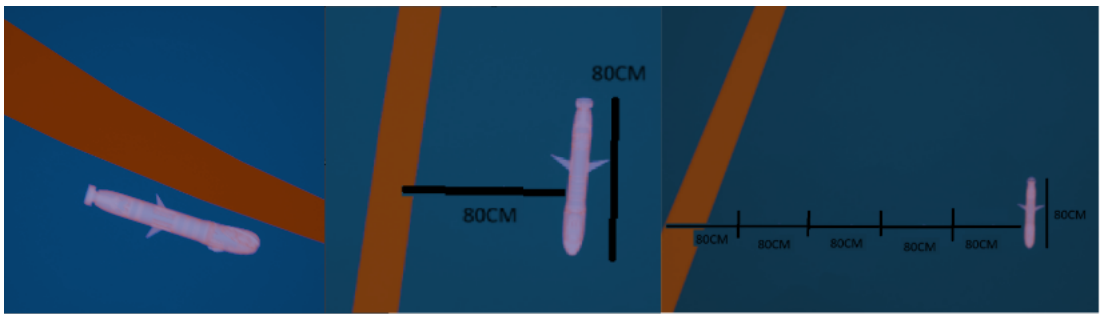

Figure 3.2: The results obtained by the USBL, the left image represents one of the best results, where the error is only a few centimeters, the middle image represents one of the average errors, where the error is less than 1 meter, and the right image represents one of the worst errors, where the error is about 4.5 meters. In our simulation, the AUV has a length of 80 centimeters and the red line is the real travel path of the AUV. (for more information about the path visualization see 6.4.1

The predicted point is then published via ROS2, for this the ROS parts of VaMEx [6] are used, the TopicName is simply "USBL" and the frequency is set so that it roughly corresponds to the ping rate, the position then published would be the position of the AUV.

3.1.4 Other approaches

In this section we will discuss other approaches and explain why we do not use them.

3.1.4.1 Collision detection at the "EchoBeam's"

Our first approach was to implement it the "right" way and not the opposite way as we currently do. The idea was to check if a ray enters the bounding boxes, for this the timer of the "EchoBeam" would be used for the ToF. One problem was that the Unreal Engine is not able to recognize our rays, the Unreal Engine can only recognize if the rays has a mesh or a bounding box, One solution would be to add a small bounding box at the end of the rays. This approach had a big problem, because the bounding box is only at the tip, and the speed of sound is quite high, so the end point of the ray would also make large steps, so it could happen that the bounding box at the end point skips the bounding boxes of the "EchoBeam", This would result in the ray passing the bounding box but not being detected because the bounding box at the tip of the ray does not enter the bounding box of the "EchoBeam". One solution to this problem would be to extend the bounding box to cover the entire ray. That was done, but for some reason the performance

degraded significantly, the decision was made to do the collision detection in the rays. The performance probably got so bad because the bounding boxes were constantly rotated, scaled and shifted per frame.

3.1.4.2 Sphere trace and box trace instead of line trace

Another idea was to use sphere and box trace instead of line trace. The difference between these different types of traces is that the Unreal Engine uses a sphere or a box for collision detection instead of a single line. The idea was to close the empty space between the rays with a box or a sphere. This also worked, but the problem was that the Unreal Engine calculates the impact point differently, so the impact point was always wrong in our tests. Due to deadlines it was decided to use line trace again and scale the bounding boxes of the "EchoBeam", but more research could be done on this topic and it could lead to great results.

3.1.4.3 Raypool

Another approach which was implemented is the raypool. The idea behind the raypool is to improve the runtime of the USBL by having a pool of rays that can be used instead of spawning new actors. Theoretically, it should be less time consuming to move an actor instead of spawning one, so the idea was to implement a pool of rays that can be reused. The raypool was implemented and contained around 5000 rays, these rays can be reserved and released by other actors, so instead of spawning a ray in the "EchoBeam", a ray would be reserved and then moved to the right position and rotated correctly and instead of destroying the ray, the ray would simply be released. If there were no free rays the raypool could also generate new rays if needed. The raypool works, but there was no recognizable difference between the spawning and destroying approach. Since the pool could spawn about 5000 rays at the start of the simulation, the simulation had to load a lot in the beginning, which it didn't do when the rays where destroyed and spawned, so we decided not to use the raypool.

3.2 Echo sounder

The echo sounder is another sensor that is usually integrated into an AUV. It can be used for automatic navigation or for mapping the seabed (bathymetry), which we do not do in our simulation. An echo sounder consists of a transceiver that sends a sonar ping to the seabed. These sound waves are reflected from the seabed and, ideally, received back by the transceiver. Based on the time of flight (ToF) and the known speed of sound, the height of the AUV to the seabed can be calculated.

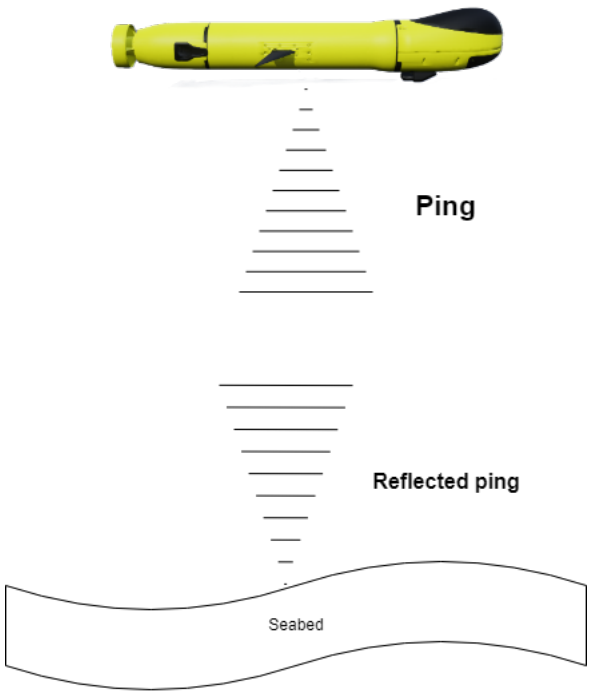

Figure 3.3: Sketch of the function of an echo sounder: The AUV sends a ping that is reflected from the seabed and the distance can be calculated based on the time of flight.

3.2.1 Implementation

The whole concept is similar to the USBL (see 3.1) as both sensors use sound waves, the same functions/actors are used to simulate this sensor. The "EchoBeam" actor is also used again, as it was used in the USBL (see 3.1) before, a variable called "isEchosounder" is created and set to true in the "EchoBeam", this boolean lets the "EchoBeam" calculate the depth instead of a position, as it happens in the USBL (see 3.1), and it also makes the beam of the rays slightly smaller and reduces the number of rays per beam, also the internal timer used to send the beams at an interval is also increased to every 15 seconds instead of every 10 seconds like the USBL does (see 3.1). The echo sounder is also blocked if the USBL is currently sending a ping. In addition, a boolean called "IsDepth" is set to true as a variable for the rays which is used to check if a ray is coming from the echo sounder, this is from a previous version and could probably be deleted anyway because the USBL checks the direction anyway, the only change that would needed to be made would be that the echo sounder would also check the direction, as it happens in the USBL and probably also check whether the echo sounder is active.

The "EchoRay" is also the same as in the USBL (see 3.1) and the only change that is made there is that the rays check if the hit actor is an echo sounder or not, for this the boolean "isEchosounder" is checked of the hit actor and it is checked if the current ray is sent from the seabed, both checks could be changed to a similar check as in the USBL (see 3.1) where the direction of the ray is checked, but both methods have the same result and are used to simulate the directional aspect of a directional hydrophone. Since the different impact points were not relevant, the rays do not check whether they hit one of the bounding boxes. Instead, the impact time, the impact position, the name of the ray and the current endpoint of the ray were simply inserted into the array which is also used in USBL for calculating the position (see 3.1).

$$Distance = ((minTime(X) \cdot speed) + (BoxExtent.X) + noise) \tag{3.6}$$

Eq. 3.6 Calculates the distance from the seabed to the AUV, minTime(X) returns the minimum impact time in the array X, speed is the speed of sound in the water in cm, BoxExtent.X is used for error correction to add the difference between the bounding box border and the center of the "EchoBeam"

The "EchoBeam" for the echo sounder also waits until it has some entries in its array, as is the case in the USBL (see 3.1). When the array has some entries, the array is sorted. Also the smallest impact time is extracted, based on this impac time the depth is calculated, in addition some noise is added to this depth. The calculated depth is published via ROS2 for which the VaMEx parts are used, the TopicName is simply "Echosounder_Distance_To_AUV", which essentially stands for the height of the AUV to the seabed.

3.2.2 Results

The echo sounder returns good results (see figure 3.4) because we only care about the depth, we don't have errors like with the USBL (see 3.1). At the moment only the minimum impact time of the rays is used, but this could be extended to take into account the other impact times and the impact points of the rays on the bounding boxes, so it would be possible to create a sensor for bathymetry, but for our purpose this sensor works very well.

Figure 3.4: The results of the echo sounder, the green dot is the predicted position if the current AUV position is used as the origin position and a downward direction (0,0,-1), and the depth (in cm) of the echo sounder (The green dot is only used for debugging and is not displayed in the simulation). The white circle is the ripple effect that we use to visualize the "EchoBeam" (see chapter 6.4.2 for more information), and the green and red lines are the path visualization (see chapter 3-Trajectory for more information).

3.3 Pressure/depth sensor

The pressure sensor is also a common sensor inAUVs and is used to determine the depth of the AUV to the ice shelf. To do this, the pressure sensor measures the pressure and the pressure increases with depth, the deeper I go, the higher the pressure, 1 bar corresponds to 10 meters. For the sake of simplicity, the Z coordinate from the AUV in our world is used, because the ice shelf is approximately at the zero point, so the Z coordinate would represent the current depth. A Unreal Engine actor called "PressureSensor" is implemented, which is attached to the AUV, in each frame the "PressureSensor" gets the current position of the AUV and extracts the Z coordinate, in addition some noise is added to this coordinate and publish this depth in meters via ROS2 with the TopicnName "PressureSensorDepthInMeter", the VaMEx ros parts are used again as the publisher.

3.4 Laser

The laser is a sensor that can be used to measure the density of particles. It is attached to the AUV and extends straight upwards. The features of the laser include visualization, collision and backscattering of the laser when the laser collides with the surface of other objects and the bins in the laser, which can be used to measure the density in different areas of the laser. A heatmap can be generated from the bins and their densities. The development process and all features are described in more detail in the following

subsections.

3.4.1 Development process

The laser is a sensor which does not yet exist in the real AUV, but will be added in the future. Accordingly, the development of the laser for the simulation was a very iterative process in which the laser was continuously improved. Initially, the laser was presented as something purely visual that could be used to point at objects in the level and where particles are reflecting the light from the laser. This finally evolved into a laser with which you can measure the density of particles in different areas of the laser and the particles reflecting the light from the laser.

The first version of the laser is a purely visual laser, which was created using a Niagara particle system. This consists of an emitter that uses a ribbon renderer to display the laser. Another emitter is used to create the particles that should be visible within the laser. The particles have a spawn location with the same dimensions as the emitter with the ribbon renderer so that the particles are only visible within the laser and create the illusion that the particles are visible because of the laser. The number of particles is increased based on the depth of the AUV.

3.4.2 Implementation

The final blueprint of the laser can be found in the project under the following path: Content/TripleSim/Laser/BP_Laser. The exact implementation and details can be looked up in BP_Laser. The code and blueprints are commented.
The final laser was implemented in such a way that it has a maximum length and a current length. If the laser does not collide with any object in the level, a spread value is added to the current length in each frame until the current distance matches the maximum distance. This simulates the spread of the laser. If the laser collides with an object, the spread value is only added to the current distance until it matches the distance between the object collided with and the origin of the laser. This was implemented with a line trace by channel.

The collision of the laser with other objects and the backscattering of the laser from the hit surface of an object were solved with a line trace by channel and a Niagara particle system. The line trace by channel provides the position at which the laser hit the object. The position can be used to calculate the current length of the laser, which is the distance between the origin of the laser and the previously determined position at which the collision occurred. On the other hand, a Niagara particle system can be activated at the position, which simulates the backscattering of the laser from the surface of the object that was hit. The particle system uses real light sources, so that the particles from the backscattering also generate real dynamic light, which illuminates the environment a little.
Figure 3.5 shows the laser colliding with an object and the previously mentioned backscattering.

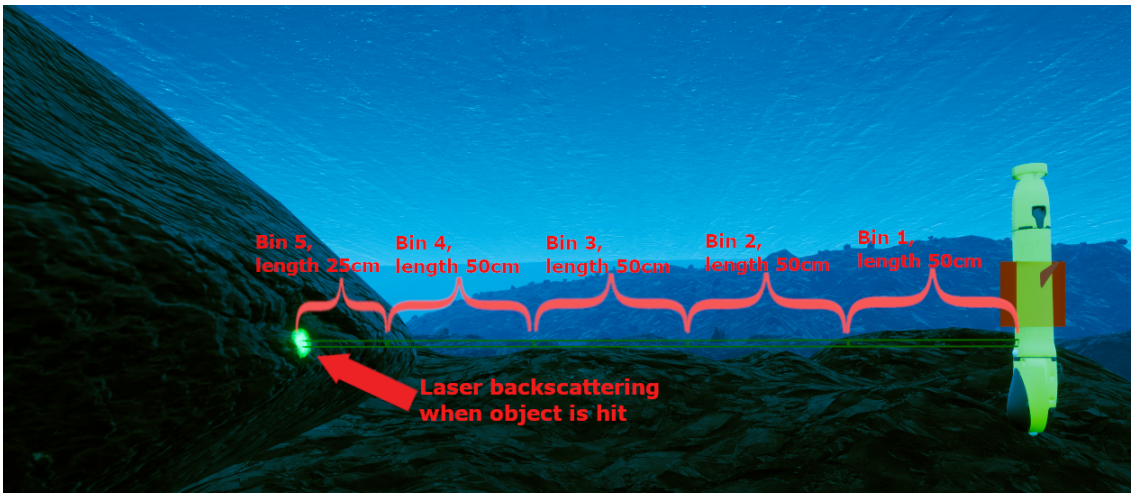

Figure 3.5: Laser colliding with an object

3.4.3 Laser bins

A laser bin always has a fixed size, including length, width and height. A bin can be used to check how many particles or how high the density is inside a bin. Using the bins and the densities, a heatmap can be created, which is explained in more detail in chapter 6.5.3.1.
In the first implementation of the bins, the values were stored in several arrays, e.g. something like the position and rotation of each bin. Although this version worked, it was not as performant, as a new position had to be calculated for each bin in each frame and the required resources also correlate with a higher number of bins. I have therefore decided to implement a new one that runs more efficiently and makes it easier to visualize the bins. You can find more information about the visualization of the laser and the bins in chapter 6.4.3.

In the final implementation, I use an actor that contains several collision boxes that represent the bins. Thus, the number of bins and the number of collision boxes are the same. This means that only the position of each bin has to be calculated once at the beginning and a certain offset value between the collision boxes. The dimensions of the collision boxes, the offset value and the number of boxes are calculated using the laser properties. It is important at this point that the entire functionality of the collision boxes is disabled beforehand so that no unnecessary resources are used. We only use the advantage that the collision boxes have a scale box extend, a position and that we can visualize the collision boxes. The actor with the collision boxes is attached to the laser, which means that the position and rotation of each bin no longer have to be calculated individually, as this is automatically done by the attachment. This avoids the unnecessary calculations that had to be done in every frame in the old implementation. Although the bins have a fixed position and size within the laser, these values must be adjusted for some bins if the laser collides with another object.
To explain this simply, I will use example data at this point. Let's assume that the laser has a maximum length of 500cm, a width of 2cm and a height of 2 cm. We also assume that we have previously defined 10 bins. This means that each bin has a length of 50cm (500cm÷10bins), 2cm width and 2cm height. Let's assume that the laser hits an object, this will shorten the actual length of the laser, as the distance between the origin of the

laser and the hit spot is only 225cm. In this case, only 4.5 bins (225cm÷50cm) should be visible. This means that bin 1,2,3,4 are completely visible with a length of 50cm, height of 2cm and width of 2cm. Bin 5 is only 50% visible, i.e. it has a length of 25cm, width of 2cm and height of 2cm. Therefore, the size of this bin must be adjusted and in this context also the offset value, so that the bin still fits perfectly to the bin before it and no gap is created between the bins. All bins after bin 5 (i.e. 6,7,8,9,10) have a size of 0. A visualization of the scenario with the example data can be seen in Figure 3.5.
As soon as the laser no longer collides with any objects and has reached its maximum length of 500cm again, all 10 bins get their full size again. To save resources, the bins are not recalculated in every frame. They are only recalculated if the current length of the laser from the last frame and the current length of the laser from the current frame are different.

The sizes and positions of the collision boxes are used for the particle detection, since it is possible to pass these values to the particles so they can check whether they are in the bin. Further information on this is in chapter 3.4.5.

3.4.4 Change the properties of the laser

The most important variables and their current default values are listed below. These variables can be changed to modify the properties of the laser.

- MaxDistance: Defines the maximum length that the laser can have. The current default value is 500cm
- LaserWidth: Defines the width of the laser. The current default value is 1cm. It is important to note that this is a scale box extend value. This means that the width goes from one point, 1cm in the negative y-direction and 1cm in the positive y-direction, so that the laser ends up with a total width of 2cm.
- LaserHeight: Defines the height of the laser. The current default value is 1cm. This is also a scale box extend value, so that the laser ends up with a total height of 2cm.
- NumBins: Defines how many bins the laser has. The current default value is 40.

3.4.5 Particle detection

Various approaches were tried for particle detection, such as using a collision box or line traces to detect whether a particle is in the laser. The problem with these approaches was that the particles would need real meshes for each particle, because normal Niagara sprites do not work with the overlapping event of collision boxes and cannot be hit by line traces. With real meshes, only a few particles could be used, because meshes are much more costly in terms of resources. I have therefore chosen a different method. The basic idea of this method is that we give each particle the dimensions and rotation of the laser and each particle itself checks whether it is currently inside the laser.

3.4.5.1 Particle detection implementation

The BP_Particle_Parent actor implements the basic functionality for all particle systems that should interact with the laser. This means that other blueprints with particle systems can inherit from this actor. This actor can be found under the path Content/TripleSim/-Particle/BP_Particle_Parent. Exact implementation details can be looked up there, since

everything is commented.
Each Niagara particle system that should interact with the laser uses a self-written Niagara scratchpad, which is called Overlapping Scratchpad. This scratchpad is used to check if a particle is in the laser. The exact design of the Niagara particle system varies from particle system to particle system. In order to explain the basic implementation better, we will assume a fictional particle system. The real particle systems will be explained in more detail later in their respective chapters 4.4 and 4.5. A simplified overview of the particle detection process is shown in figure 3.6.

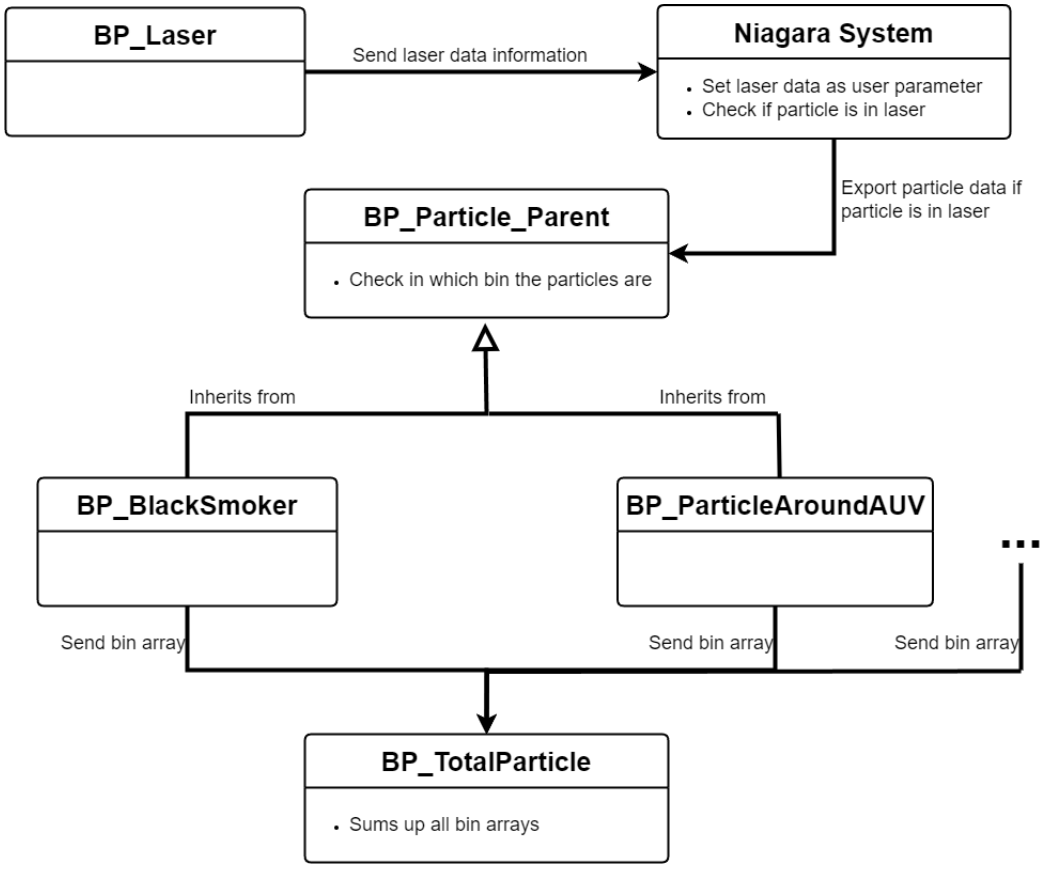

Figure 3.6: Simplified overview of the particle detection

Niagara side
Since the communication between a Niagara system and a blueprint is not very intuitive, I can recommend a video [16] to understand how this works.
In Niagara, it is possible to create user parameters that can have different data types. The values of these user parameters can be changed via blueprints, even during runtime. This is used to give the particle system a vector with the dimensions of the laser, i.e. a scale box extend in x,y,z, a vector with the origin of the laser and the 3 rotation vectors of the laser. These user parameters are updated by a blueprint in each frame.
Niagara offers the option of creating scratchpad modules. These are self-written Niagara modules. They allow you to use something similar to blueprints, with the difference that these scratchpads are not as powerful, e.g. no loops can be used. The scratchpads can read various values from the particle system. For example, they can also read the data of the previously mentioned user parameters.
Furthermore, Niagara particle systems have particle attributes. In contrast to the user

parameters, where the values apply to the entire particle system, particle attributes are individual for each particle. For example, there is an attribute that contains the position of a particle. You can also create your own particle attributes. A self-created boolean particle attribute with the name ExportData is used for the particle system.
The scratchpad uses the user parameter data of the laser and the particle attribute, which provides us the position of the particle, to check whether the particle is within the laser. If the particle is inside the laser, the self-created attribute ExportData is set to True. ExportData is used at the end in the Export Particle Data to Blueprint module as the export condition. This means that not all data from all particles has to be exported to the blueprint in every frame. Instead, only the data of the particles that are actually in the laser in the current frame are exported.
Exporting particle data is quite limited. Only 2 vectors and 1 float per particle can be exported. The 2 vectors are named Position and Velocity, but can contain any vector. However, they are listed in the blueprint with these names. The float has the name Size, but can contain any float value. In our case, we export the world position, the velocity and the sprite size of the particle.

Blueprint side
In BP_Particle_Parent, the callback handler of the Niagara system must be set so that the Niagara system knows which blueprint it must communicate with. The Event Receive Particle Data Interface is also implemented. This is called when particle data is exported from the Niagara system. The particle data is only exported if the particle is located somewhere in the laser. Therefore, Event Receive Particle Data checks in which bin exactly the particle is located. The check is first done in the blueprint and not directly in the scratchpad, because no loops can be used in scratchpads. To avoid having to check whether the particle is in the bin for each individual bin, the distance between the origin of the laser and the world position of the particle is calculated first. Based on this, the bin in which the particle is located can be estimated. The system then checks whether the particle is actually located in the previously estimated bin. If this is not the case, the two neighbour bins are checked.
The total density of a particle system is stored in the corresponding blueprint in the integer variable NumParticleInsideLaser. The density of each individual bin from a particle system is stored in the array of integers with the name NumParticlesInsideBin. The index of the array represents the respective bin. This means NumParticlesInsideBin[0] contains the density of bin 1, NumParticlesInsideBin[1] contains the density of bin 2 and so on.
BP_Particle_Parent also contains the integer variable SmokerDensityMultitply, the boolean isBlackSmoker and the boolean CorrectTooHighVelocity. SmokerDensityMultitply determines the value by which a particle is multiplied or how high the density of a particle is. Let's assume that SmokerDensityMultiply is set to 3 and there are 3 particles in the laser, then the laser has a total density of 9 (3·3). By default, the value is 1 and is only changed by black smoker blueprints. If new particle systems are developed in the future that should interact with the laser, the value of SmokerDensityMultiply can be adjusted. CorrectTooHighVelocity and isBlackSmoker are only used in the blueprint with a black smoker particle system. This is therefore explained in more detail in the black smoker chapter 4.4.

3.4.5.2 Total density of all particle systems

Since not only one particle system is used that interacts with the laser, such as the black smoker or the particles around the AUV, the densities from the bins of all particle systems must be added together. This is done in a new actor with the name BP_TotalParticle. This actor can be found under the following path: Content/TripleSim/Particle/BP_TotalParticle. Since the parent actor BP_Particle_Parent was created and all particle systems that interact with the laser inherit from this class, we can simply search for and reference all actors of type BP_Particle_Parent in the level in BP_TotalParticle. This allows us to sum up the array with the densities of the bins of each actor in each frame. The same is done for total particles of all actors. The result is the total density of each bin from all particle systems that interact with the laser. Figure 3.6 contains a simple overview of the communication and process between the Niagara system, BP_Particle_Parent and BP_TotalParticle.
The array of bins and the integer with the total density in BP_TotalParticle is used in the laser UI (chapter 6.17) to visually display the densities of the bins and the total density.

3.4.6 Challenges

There were several challenges for the laser. As I couldn't find anything similar to laser simulation underwater or particle detection with a laser in Unreal Engine, I had to think a lot about which solutions are available and which would be best for our project. This means that I had to do a lot of experimenting and testing.
Since we have several sensors and many expensive particle systems running at the same time, another challenge was to implement the laser and particle detection in such a way that it requires as less computational effort as possible. This includes, for example, exporting only the particle data of the particles that are actually in this frame in the laser or that an estimation is made beforehand based on the distance from the origin of the laser and the particle position, so that it is not necessary to check for each bin whether the particle is located in this bin.

3.5 IMU

An IMU sensor, short for inertial measurement unit, is usually used to determine the position of an object in space. Various parameters can be determined for this purpose, usually these are acceleration, angular velocity and orientation. A combination of different measuring devices such as accelerometers, gyroscopes and magnetometers is used for this. The IMU usually measures values along the x, y and z axes and has installed three of the corresponding components for this purpose. Usually, an IMU sensor measures orientation, angular velocity and acceleration along these three axes; it is also referred to as a 9 DoF IMU ([36]).

Since an implementation for an IMU sensor already exists in the VaMEx project, which is the basis for this project, it was used largely unchanged. However, the functionality has been slightly adjusted. Some components are not needed in this configuration, so they have been removed. The following changes have been made:

- The simulation time stamp and the according utility class have been removed
- Commented out code has been removed
- The calculated sensor values are now passed to the IMUVisualizer (see 6.5.2)

The position data of the AUV's unreal object is used to calculate the sensor values. These are stored in the *FTransform* class. This class contains a variety of data that is sufficient to calculate orientation, acceleration and angular velocity. The calculation is done in a tick function; in addition to the *FTransform*, the delta time since the last tick is also required as a parameter. This is done in the following function:

```
void UIMUSensor::PhysicsSubstep(float DeltaTime, float SimTime,
   FTransform transform)
```

Since the values are to be passed to the ROS system, a corresponding message type is used. There is a predefined message type *sensor_msgs/msg/Imu Message* for this purpose. This has the following form:

```
std_msgs/Header header

geometry_msgs/Quaternion orientation
#float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
#float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
#float64[9] linear_acceleration_covariance # Row major x, y z
```

These values can then be used for visualization (see 6.5.2).

4. Environment

In the following we will discuss various features that we have implemented for the environment. We will explain various sound & visualization effects, the black smoker, organic particles around the AUV and we will discuss our terrain.

4.1 Light & Fog

During the course of the project, it was agreed that two different lighting modes should be available for the scene: a realistic and a demo mode. Since it is assumed that the lakes to be examined are located under thick layers of ice (see 4.7), the entire scene would be without light, as no light would penetrate through such layers of ice. This view should be recreated in the realistic lighting setup. However, since the demo is also an important part of the project, a second mode is added in which ambient light is present to a certain extent so that at least terrain features can be recognized. Nevertheless, the view should also be limited in this setup so that the user cannot see through the entire scene.

To illuminate the scene, all objects used had to be adapted to the surroundings, as the ice cover created a closed space. In most other Unreal projects, however, you have a direct light source from above, usually sunlight. However, since the direct radiation is blocked by the ice cover, various adjustments were made. Nevertheless, the scene used all the objects that are typically used in Unreal to light a scene. These include *DirectionalLight*, *SkyAtmosphere*, *SkyLight*, *SkySphere* and *VolumetricCloud*.

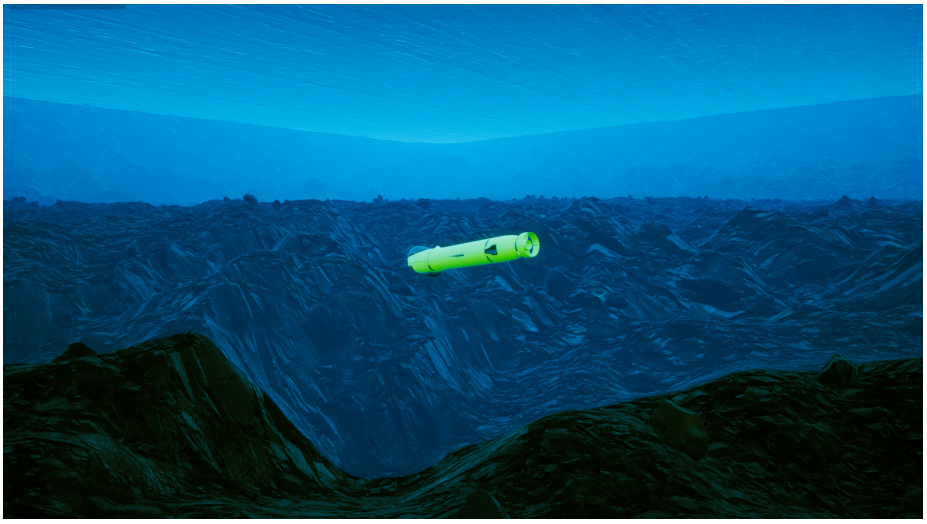

Figure 4.1: Scene without fog

In combination with the post process volume (see 4.6.2), a realistic view of an underwater scene is created. Since the direct light source is above the ice layer, it must be set relatively high in order to still generate global illumination. However, you can already see from the image above that there are some problems with this setup. This makes it possible to look through the entire scene, which seems very unrealistic for a 2.5km^2 scene. Also, there is no decreasing visibility with increasing depth, which would normally be expected. Ultimately, this is problematic for the light source on the AUV in that no light cone is

visible. The lighting can only be recognized in this way if the corresponding material on the AUV is activated (see 6.3.2).

To simulate these effects, an exceptional height fog was added. This component can be used to overlay a fog over the entire scene. The fog has a starting point from which it becomes denser as it gets deeper. This already simulates relatively well the effect that there should be less visibility as depth increases. Various parameters can also be used to define the extent to which the view should be restricted. In order to solve the problem of the visible light cone, it is also necessary to activate the volumetric fog. This is the best solution to the problem that we were able to find during the course of the project, but it also brings with it some other problems (4.1.1). The result looks like this:

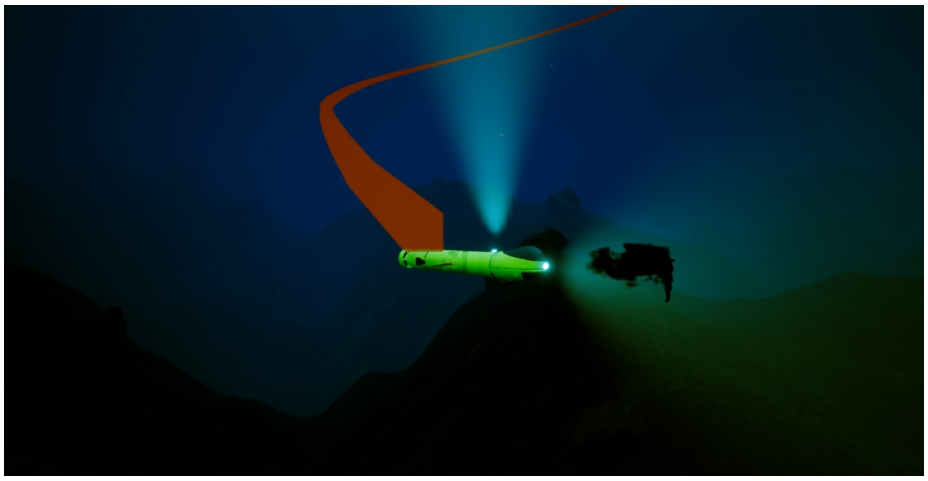

Figure 4.2: Scene with light fog

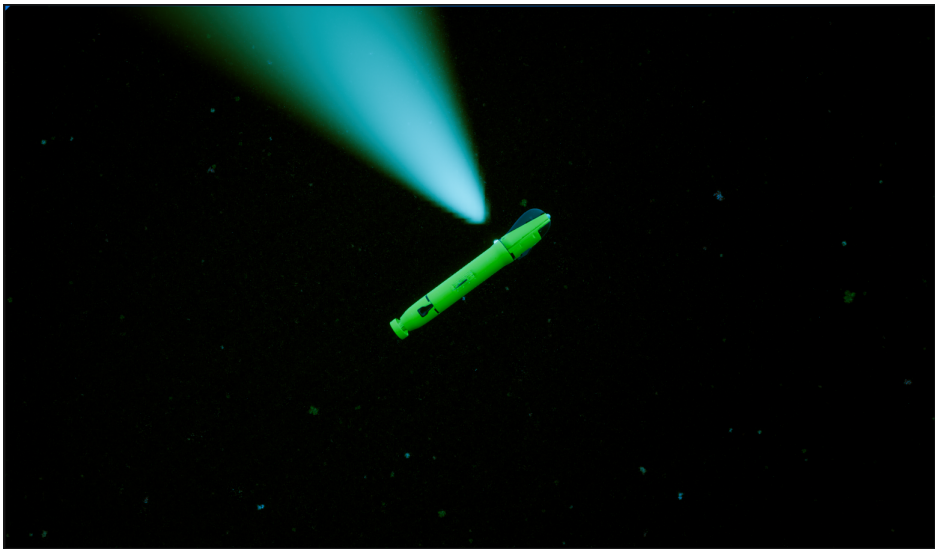

Figure 4.3: Scene with dense fog

4.1.1 Problems of the volumetric Fog

Due to some limitations with the volumetric fog, there is a ghost trail of light (see figure 4.4) when the light moves too fast. To fix this, the temporal reprojection was set to zero

with the command "r.VolumetricFog.TemporalReprojection 0", which gets executed every time the simulations starts, for that the AUV blueprint execute this command at the begin play function. This is a known limitation of the Unreal engine, which offers no solution other than disabling the temporal reprojection.

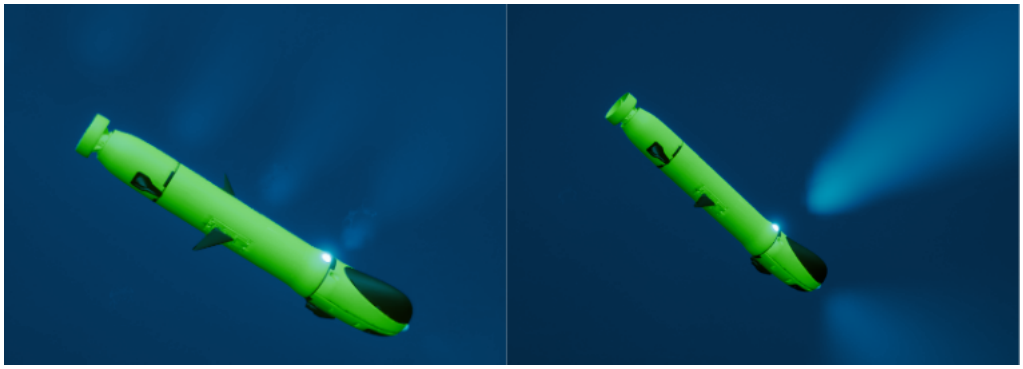

Figure 4.4: At the left image the temporal reprojection is activated and at the right image the temporal reprojection is deactivated

4.1.2 Light at the AUV

A spotlight is added to the AUV, which is directed upwards and is intended to illuminate a camera that is also directed upwards. Two modes for the light were implemented, a manual mode and an automatic mode. The user can switch between these two modes via the control panel in the user interface. In manual mode the user can turn the light on or off via the UI and in automatic mode the user can change the off and on time (in milliseconds) of the light so that the light flickers a little, this should simulate the exposure time for the camera on the AUV, in a real scenario the light would also turn on or off a certain time. Also a spotlight on the top of the AUV was installed, which is unrealistic, but we included it anyway for visualization purposes, this light is only activated in the demo environment mode (see section 4.1)

4.2 Ice Shapes

Saturn's large moon Enceladus is one of the most enigmatic worlds in our solar system. Enceladus harbors a subsurface ocean of liquid water. Given our limited knowledge of Enceladus, we can liken its environment to Antarctica on Earth. I have prepared the ice shapes in different variations and as well as iceberg shapes. According to our weekly meeting with our supervisors, these got modified and finalized. I have prepared the model using blender.

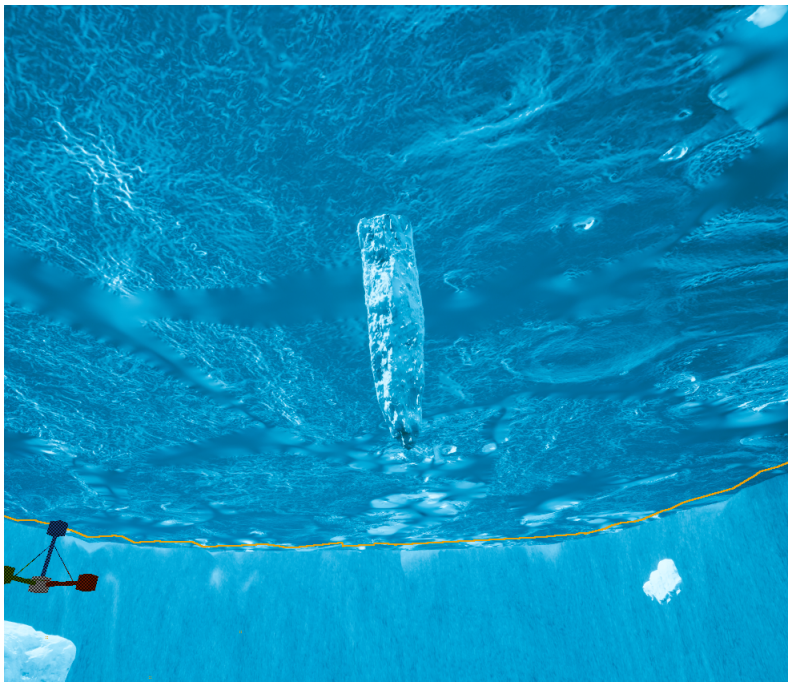

Figure 4.5: ice berg 1

Figure 4.6: ice berg 2

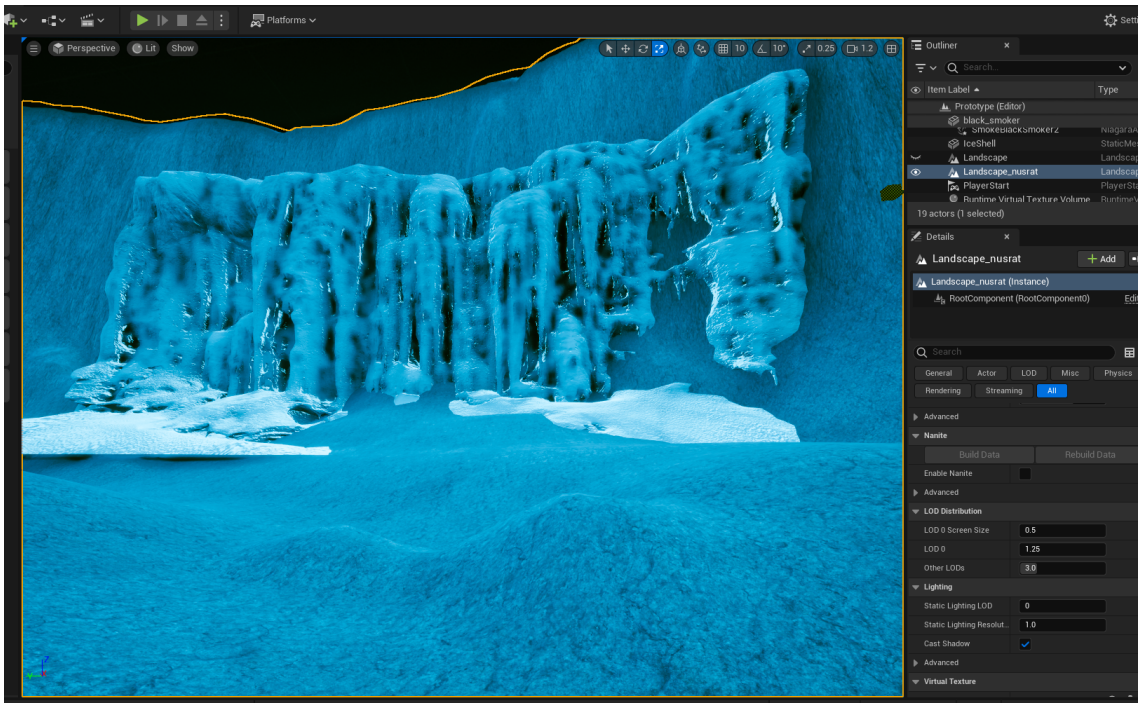

Figure 4.7: ice wall

Figure 4.8: ice surface

Finally, in our main scene, it is not so visible because of volumetric fog. We had to use these to make the environment real and in real scenarios there will be no light and that's why in our project scene as well, it is not much visible. To optimize and avoid performance issues, only few shapes(4.8) are added.

4.3 SFX

To enhance the immersion in the simulation, some sound effects are added to our environment. Some of the sound effects may not be exactly suitable for a real-life scenario, but for non-experts in this field they might fit. MetaSounds from the Unreal Engine is used, MetaSounds is used instead of the normal sound system from Unreal Engine because it is easier, and therefore sounds better, to customize the sound files. For the whole demo level, an underwater ambience sound clip [34] is added and a snapping shrimps sound

[23] is added, these sound effects can be heard throughout the level and always have the same volume, these sounds are not changed via MetaSounds.

Also a sound effect for the black smoker (4.4) is added, this sound is a recorded sound clip from a real black smoker [9]. The pitch is not adjusted and the black smoker sound is always on. An audio attenuation with the Unreal Engine audio attenuation module is created. This attenuation ensures that the volume of the sound file is loudest in the center of the attenuations center, which is placed directly on the black smoker, and that it gets quieter the further you move away from the center.

A sonar ping sound [33] is also added, this sound will be triggered when the USBL (3.1) sends a ping. To do this, the AUV checks whether its USBL part sends a ping, and if so, it plays the sound effect. The sound effect also has sound attenuation and is attached to the AUV so that it can only be heard in the vicinity of the AUV.

The last sound effect is the engine sounds, for the sound clip the Marum provided us an audio clip of the IMGAM AUV's engine [25]. The pitch of this file is changed via MetaSounds, the pitch changes depending on the current speed of the AUV. To do this, the current speed is calculated. The current speed is then set as the pitch parameter. The speed is mapped (clamped) to a specific range and bounded by maximum and minimum values to prevent the audio file from producing any noisy sounds.

The user can activate or deactivate all sounds via the user interface; there is a checkbox under the controls for this purpose; all sounds are deactivated by default.

4.4 Black smoker

Black smokers are hydrothermal vents on the seabed. During the process, cold seawater enters the ground where it is heated by geothermal activity, for example. During this process, minerals and metals are dissolved from the ground. As soon as a certain heat is reached, the hot water with the minerals and metals is released from the ground back into the cold seawater. Black smokers form at the exit points, with the structures consisting of metal sulfides. The "black smoke" that comes out of these structures consists of hot water with very fine particles of minerals and metal sulfides [26].

4.4.1 Black smoker model

There were two 3D models to choose from for the exit point of the black smoker. One was provided by Marum [24] and the other one is from Sketchfab [21].
The model from Marum was not a complete 3D model, it was rather a reconstruction of the surface of a black smoker, from one perspective. In addition, the texturing was broken in some places, so that some parts had no texture. I reworked the texture of the model with Blender so that the whole model ended up with a realistic material. I solved the other problem by cutting the model into several parts. I arranged these different parts in such a way that the result was a complete 3D model that looks correct from every perspective and not just from one particular perspective.
The Sketchfab model also had problems that had to be fixed. Firstly, the model had no texture or material and secondly, the UV mapping of the model was broken. I found a suitable texture in Unreal Engine via the Quixel Bridge [35]. I created a material using this texture and created my own normal map with GIMP to display the depths of the material. Due to the broken UV mapping, the texture was displayed very strangely in

some places, so that lines were sometimes very stretched and looked completely wrong. I was able to fix the UV mapping using Blender so that the material is displayed correctly without any graphical errors.
In the end, we decided to use the 3D model from Sketchfab because it looks and fits better into our environment.

4.4.2 Black smoker plume

This subchapter describes the plume of the black smoker. It explains why I chose which technology, the development process and the most important properties of the plume.

4.4.2.1 Niagara vs. Niagara Fluids

For the visualization of the black smoker's plume I used the Niagara particle system from Unreal Engine. Here I had to decide whether to use the normal Niagara, which mainly works with 2D sprites that are always rotated to the camera, or the beta plugin Niagara Fluids. Arguments for using normal Niagara are that it doesn't need so many resources and that there are many tutorials on the internet. Arguments against normal Niagara are that it is difficult or impossible to simulate smoke realistically, such as collision with the smoke or illumination of the smoke.
The arguments for Niagara Fluids are that things like collision, illumination of the smoke or the movement behavior of the smoke can be simulated more realistically. One argument against Niagara Fluids is that it is still a relatively new beta plugin and is therefore not as mature or stable. Accordingly, there is less learning material on the internet. Furthermore, Niagara Fluid particle systems are more expensive in terms of the resources.
In the end, I chose Niagara Fluids because this project requires a simulation that is as realistic as possible.

4.4.2.2 Development process

I created many plumes during the project, improved them and always adapted them according to the wishes of the project managers. A small progression of the plume can be seen in figure 4.9.

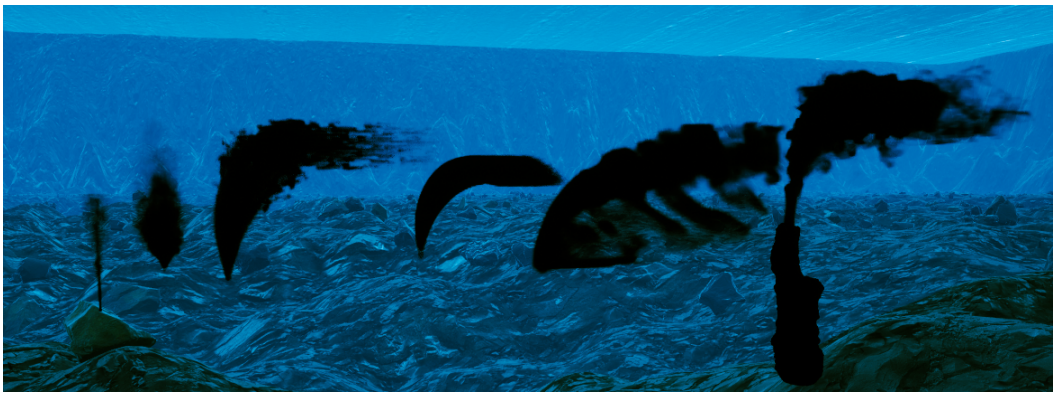

Figure 4.9: Black smoker plume progression

Since it is not possible to get the exact position of each smoke particle in Niagara Fluids, I used a combination of Niagara Fluids to visualize the plume and normal Niagara to detect the particles. I gave the particles from the normal Niagara the same shape or path as the smoke from Niagara Fluids. This was done using scratchpad modules. The particles from

the normal Niagara are transparent so that they are not visible for the user and we only take advantage of the fact that it is possible to detect them with the laser. In the following, I am talking about real particles if they are the particles from the normal Niagara that can be detected by the laser. Chapter 3.4.5 on particle detection explains how the particles are implemented so that they can be detected by the laser. Accordingly, these would also be the real particles.

Two plumes have been chosen for the shortlist. The first plume was implemented by creating a Niagara particle system with an emitter. This emitter generates the real particles and also implements the Niagara side for particle detection, which was discussed in chapter 3.6. The emitter spawns the particles in a sphere at the exit point of the black smoker model. The particles receive a cone velocity and by using a self-written scratchpad, the behavior is defined that the particles bend over time in the ocean current direction and have approximately the shape of a real black smoker plume. In addition, another Niagara Fluids emitter was created to generate the smoke. This emitter uses every real particle and emits smoke from them. Properties such as density, lifetime and velocity of the Niagara Fluids emitter are constant. This ensures a 1-to-1 correspondence between real particles which can be detected by the laser and the smoke. Figure 4.10 attempts to visualize this 1-to-1 correspondence between real particles and smoke. In the figure, the real particles are white to show the correspondence. These particles are normally transparent.
Although this plume has the advantage that it has a 1-to-1 correspondence, the disadvantage is that it does not look as good as the other version. By ensuring the correspondence, the smoke cannot be diluted over time and spread slowly on the horizontal plane. That's why our group chose the second version as the final plume.

Figure 4.10: Black smoker plume 1-to-1 correspondence between real particles and smoke

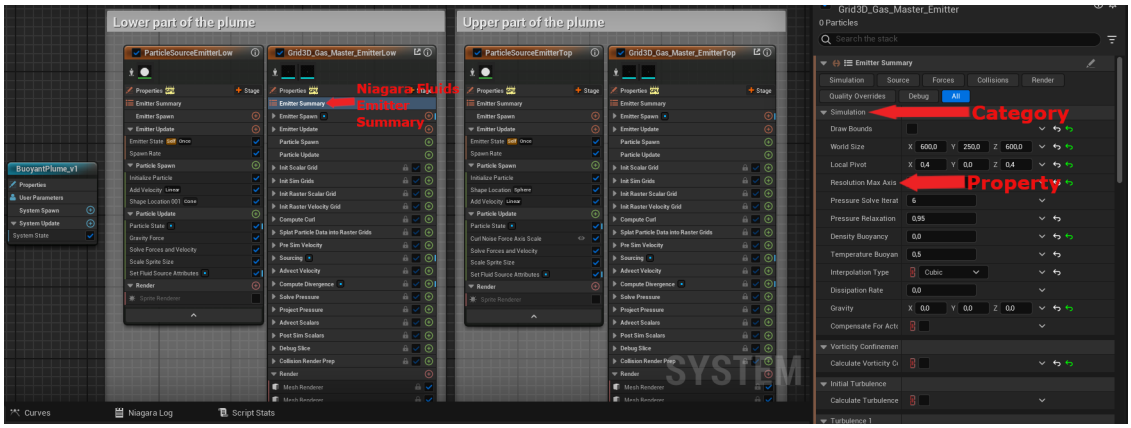

Figure 4.11: Overview of the Niagara particle system of the black smoker

4.4.2.3 Final Plume

The particle system for the final plume is located in Unreal Engine at the following path: Content/TripleSim/VFX/Niagara/HydrothermalVents/BuoyantPlume_v1
The final plume consists of two parts. The first part is responsible for the lower part of the plume, i.e. the vertical smoke, and the other part is responsible for the upper part, i.e. the horizontal spread of the smoke. The figure 4.11 shows an overview of the two parts and their emitters in Niagara. One part consists of a normal Niagara emitter and a Niagara Fluids emitter. For a better understanding, I will list the names of the emitters, explain their functionality and most important modules.
The lower part of the plume consists of a normal Niagara emitter with the name ParticleSourceEmitterLow and a Niagara Fluids emitter with the name Grid3D_Gas_Master_EmitterLow. ParticleSourceEmitterLow spawns particles in a cone, which have a velocity in the z-direction. The particles have a very short lifetime and are transparent so that they cannot be seen. In addition, the Set Fluid Source Attributes module is used so that the smoke from Grid3D_Gas_Master_EmitterLow can emit from the particles of ParticleSourceEmitterLow.
Grid3D_Gas_Master_EmitterLow uses a large number of standard modules that are required to generate the smoke. The easiest way to change the adjustable properties is the Emitter Summary module. All adjustable properties of the modules are listed there and separated into categories. In the following, I explain what I consider to be the most important categories and adjustable properties for the visualization of the plume. Figure 4.11 shows what is meant by category and property and where the Emitter Summary module can be found.
The World Size property is located in the Simulation category, where the size of the simulation area can be defined. It is important to note that the required resources increase with a bigger World Size. The Resolution Max Axis and Pressure Solve Iterations properties are responsible for the visual quality of the simulation. Higher values result in a more detailed and accurate smoke. However, this also increases the required resources.
The Wind category contains the Calculate Wind, Wind Direction and Wind Magnitude properties. This category is used to implement the bending of the plume in the ocean current direction. Suitable values must be used for the properties so that the bending looks as realistic as possible. I used a sphere as a collider so that the plume does not break out directly at the side, as with a real plume of a black smoker. I explain this in more detail in chapter 4.4.3 on the final blueprint for the black smoker.

Particle Source specifies that the particles from ParticleSourceEmitterLow are used as emitters for the smoke. The most important properties are Density Scale, Density Radius Scale, Velocity Scale and Velocity Radius Scale. These properties must be adjusted with suitable values so that there is a high density and velocity at the beginning, which decrease and spread over time.
In the Attributes category, the Dissipation Rate Density, Substraction Rate Density, Substraction Amount Density and Attribute Resolution Multiplier properties are the most important. These determine how quickly and how much of the smoke disappears over time.
The color of the plume can be specified in the Render Color category. In the Collide Against category, the Use Mesh Collisions property is the most important so that the smoke can interact or collide with other objects. It is essential that all actors in the level that should interact with the smoke have the "Collider" tag. This allows the Niagara particle system to know which meshes it needs to interact with.

The upper part of the plume consists of the normal Niagara emitter ParticleSourceEmitterTop and the Niagara Fluids emitter Grid3D_Gas_Master_EmitterTop. ParticleSourceEmitterTop is identical to ParticleSourceEmitterLow, except that the values of the properties are different and the Curl Noise Force Axis Scale module is also used. This module ensures that the particles have a certain amount of noise and that the smoke spreads slowly on the horizontal plane and dilutes. Emitter Grid3D_Gas_Master_EmitterTop is also identical to Grid3D_Gas_Master_EmitterLow, only the values of the properties have been adjusted accordingly.

The particle detection with the real particles is implemented in another particle system. It can be accessed under the following path: Content/TripleSim/VFX/Niagara/Particles/-ParticleInsideSmoker_v2
This particle system attempts to approximate the shape of the particle system with the smoke. Therefore, there is no direct 1-to-1 correspondence between smoke and real particles. The particle system with the real particles uses the Curl Noise Force module. This provides a certain amount of noise to ensure that not all particles behave exactly the same in terms of speed. As a result, each particle moves slightly differently. The Drag module is also used to slow down the particles over time. 10,000 particles are spawned per second, with each particle having a lifespan of 6 seconds. This means that there are a maximum of 60,000 real particles in the entire plume. The particles from the black smoker are multiplied by a higher density multiplier (SmokerDensityMultiply) during particle detection with the laser. More on this in the subchapter 4.4.4, which deals with the particle detection of the real particles from the black smoker.
The particle system with the real particles is implemented as described in chapter 3.4.5. The figure 4.12 shows the final plume on the left side, in which the real particles have been made visible. On the right side is the complete black smoker as it can be seen in the level.

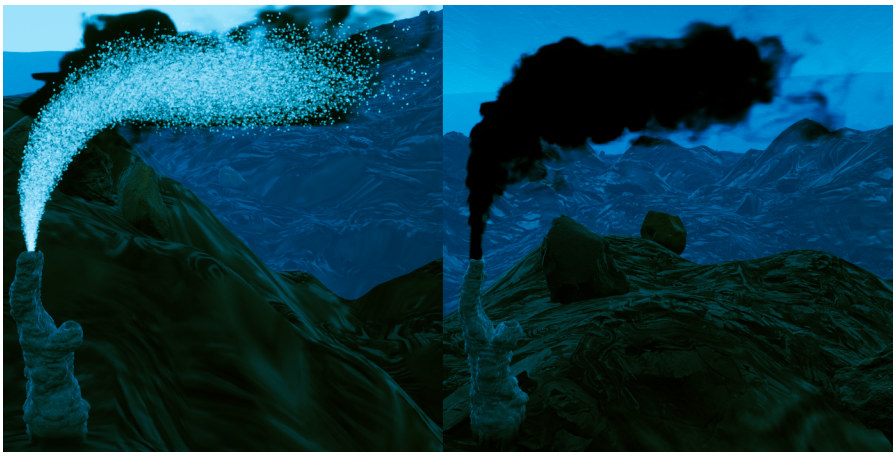

Figure 4.12: Black smoker where the real particles are visible vs. final black smoker

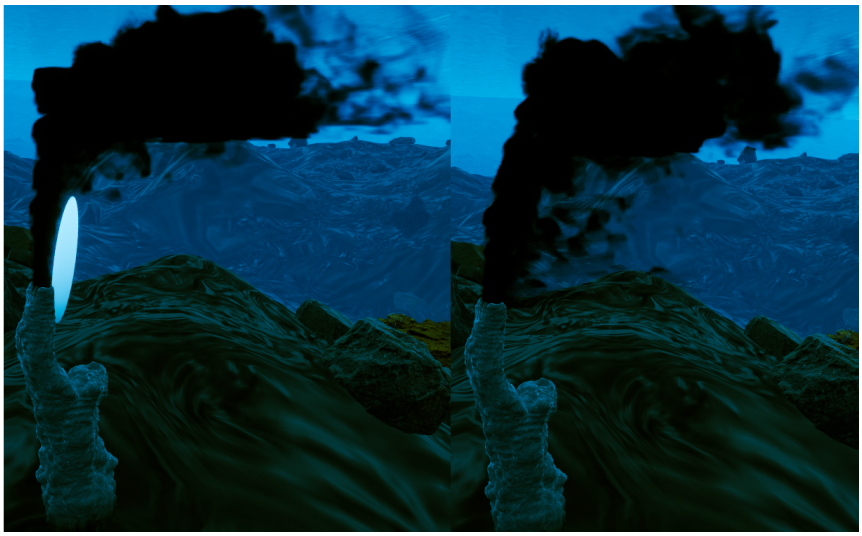

Figure 4.13: Black smoker with visible sphere vs. black smoker without sphere

4.4.3 Black smoker blueprint

All components required for the black smoker have been combined in a blueprint. This can be dragged into any level and can be found under the following path: Content/TripleSim/HydrothermalVents/BP_BlackSmoker_v1
The black smoker blueprint inherits from BP_Particle_Parent so that it has the basic functionalities for particle detection (see chapter 3.4.5). The black smoker blueprint consists of the 3D model, the particle system with the smoke or visual plume, a sphere that serves as a collider, the particle system with the real particles that can be detected by the laser and two collision boxes. The sphere is used to prevent the smoke from directly breaking out at the side. The sphere is removed from the rendering so that it is not visible to the user. Figure 4.13 shows the black smoker on the left, where the sphere has not been removed from the rendering to show how it blocks the smoke. On the right side is the black smoker without the sphere (where the smoke breaks out directly at the side).
The two collision boxes are used as a replacement for the ocean current modifiers (see chapter 6.7.1) at the exit point of the black smoker. This is done so that the particles

around the AUV can interact correctly with the lower part of the plume. These collision boxes are used exclusively by the particles around the AUV. This is why this part is closely related to chapter 4.5.1.5 and is explained in more detail there.
The plume of the black smoker consists of 60,000 real particles. Due to performance reasons, I decided to not use more real particles. Instead, I multiply the density of each real particle from the black smoker that was detected by the laser by the value in the SmokerDensityMultiply variable. This value is set to 10 for the black smoker blueprint. Let's assume that the laser could detect all real particles of the plume at the same time. Then the total density would be 600,000 (60,000·10). The value in SmokerDensityMultiply can be changed as required and, for example, reset to the default value of 1.
However, the laser is not so large that it can cover the entire plume at once. If you go over the opening of the plume with the current size of the laser, around 400 particles are measured. This is also shown in Figure 4.15, where you can also see how the plume is illuminated by the AUV.

4.4.4 Particle detection for real particles from the black smoker

The real particles from the black smoker have a high velocity at the exit point (partially up to 320). The velocity describes how much distance (in cm) the particle has traveled in one second. If the particles are too fast, the laser may not be able to detect some of them. For example, let's assume that a particle travels 320cm in one second and our system is running at 30 FPS. Then a particle travels 10.67cm (320÷30) per frame. As the laser has a width of only 2 cm, the distance traveled per frame is greater than the width of the laser. As a result, the laser may not detect this particle. It is true that particles that are too fast could not be detected in real life too, but this makes no sense with the real particles from the black smoker. I only use 60,000 particles anyway for performance reasons and multiply the density by the SmokerDensityMultiply variable to suggest that there are more particles. That's why I decided to fix this error for the real particles from the black smoker. For the other particle systems, which are not related to the black smoker, this error is not fixed, as it is a natural phenomenon.
There were two possible solutions to fix the error. In the first solution, a mesh was created that has the same size and position as the laser. For particle detection, the Collision module was used in the Niagara system instead of the self-written scratchpad (covered in chapter 3.4.5). The Collision module offers different methods to check whether a particle has collided. The most precise method is ray tracing for the particles. Ray tracing allows you to specify which meshes the particle should collide with. The problem is that ray tracing only works correctly with CPU particle systems. However, CPU particle systems can only display a few thousand particles. GPU particle systems have recently added a ray tracing function, but this is declared by Unreal as "experimental" and "is in heavy development". In addition, GPU ray tracing is only possible with DirectX12. Therefore, the GPU depth buffer was used as collision method. This method uses the depth buffer in the level to calculate the collision. The image of the camera is displayed with black to white color tones to calculate how close an object is to the camera. Although the depth buffer method is a very cheap way of calculating collisions for the particles, it is not very accurate. This means that the collision can be different depending on the perspective of the camera. In addition, the ratio of pixel density and particle size plays an important role. If, for example, the camera is not pointed at the particle system, no collision is detected.
Another problem is that the depth buffer method does not allow you to specify that the

particles should only collide with certain meshes. I have developed a workaround for this. The custom depth buffer can be used for the GPU depth buffer method. To ensure that the particles only collide with the mesh in the laser, the mesh is removed from the main depth buffer and added to the custom depth buffer as the only object. This makes it possible for the particles to collide only with this single mesh, even with the depth buffer method. The export condition for the particle data is the particle attribute, which indicates whether a particle has collided.

Since this solution to fix the error is not very accurate, I decided to use the second solution. This solution uses the self-written scratchpad Overlapping Scratchpad as described in chapter 3.4.5. However, this scratchpad is modified a little. For each particle, the speed of the particle is checked. If it is so fast that the particle could be missed by the laser, the laser is increased for this one particle. To do this, a quaternion is made from the direction vectors of the laser. The velocity vector of the particle is multiplied by this quaternion. I call this vector newVector. Next, it is calculated for x,y,z whether the laser must be increased in one of these directions so that the laser can detect the particle. This is done with the following formula for the x-direction:
$|x\ value\ of\ newVelocity| \div Framerate - Laser\ length - Sprite\ size$
If the result is > 0, the laser must be increased by this value in the x-direction. If the result is <= 0, the laser does not need to be increased in the x-direction. This is done identically for the y-direction (width of the laser) and z-direction (height of the laser). Let's assume that the dimensions of the laser are x=200cm, y=2cm, z=2cm; framerate=30; particle sprite size=2; newVelocity=200. Then the calculation for increasing the laser in the y-direction would be: $|200| \div 30 - 2 - 2 = 2.67$
This means that the laser must be increased by 2.67cm in the y-direction for this one particle. This calculates the laser dimensions for each particle individually, based on how fast the particle is. If the particle would not cause an error, then the laser dimension is not changed. However, it is important to note that this is only done for the real particles of the black smoker.

4.4.5 Collision and illumination of the plume

The plume of the black smoker has collision. However, this is not automatically active for all meshes. For an object to be able to collide with the black smoker's plume, the object must have the Collider tag. This lets the plume know which objects it should interact with. Figure 4.14 shows a sphere colliding with or blocking the smoke. The collider tag can be set in the details area of the respective object. The exact location is also shown in figure 4.14.
The plume of the black smoker can be illuminated by all light sources as standard. This also includes the spotlight that is attached to the AUV. Figure 4.15 shows the light from the AUV illuminating the plume of the black smoker.

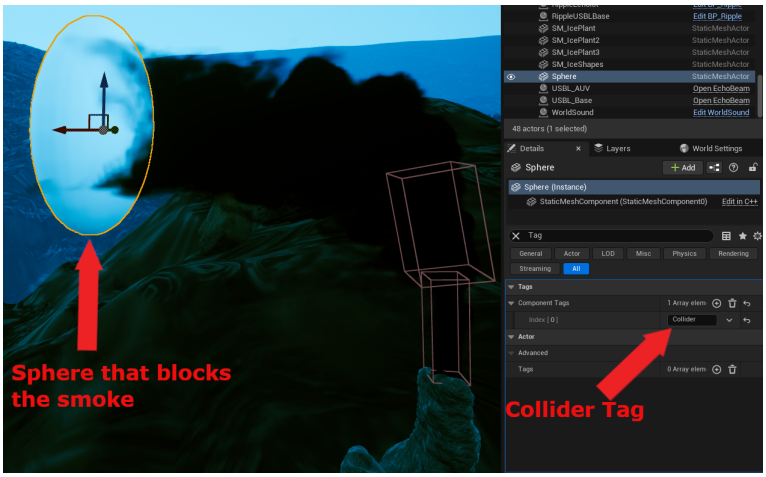

Figure 4.14: Sphere that blocks the plume and Collider tag

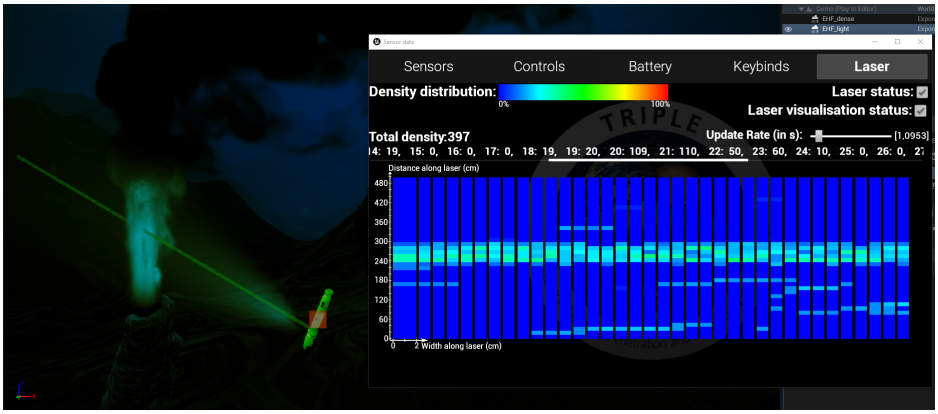

Figure 4.15: Density inside plume and illumination of the plume

4.4.6 Challenges

As there are unfortunately no digital media students in our project, the biggest challenge for me was to take on the tasks of a digital media student. This includes creating normal maps, textures, materials, sprites, working with Blender, particle systems i.e. the visual part of it and 3D modeling. Before the project, I had minimal to no experience in these areas.

Another challenge was that Niagara Fluids simulations only work with GPU particle systems. Compared to CPU systems, GPU systems can only take place in a limited area. This means that I was much more limited, because I had to define the area in which the simulation would take place beforehand. Here it was important to find a good balance between performance and the size of the area.

The next challenge was that a lot of particles were needed for the real particles in the plume of the black smoker. Here I used the workaround that there is a maximum of 60,000 real particles, but these are multiplied by a higher SmokerDensityMultiply.

Since Niagara Fluids is still a beta plugin, it is not always stable. Especially while creating the plume, my Unreal Engine crashed very often. During these crashes, even the things that were actually saved by the auto save function were not saved. Therefore, when working with Niagara Fluids, you should save manually as often as possible. As soon as

the creation of a Niagara Fluids particle system is finished, there are no more problems with crashes. The particle system or the corresponding blueprint can be dragged into any level and will not crash. In addition, Unreal Engine crashed a lot when I tried to start our level. I was the only one in the group who had this problem. After changing my RHI from Vulkan to DirectX12 in the project settings, I had no more crashes.

4.4.7 Remarks

The black smoker chapter is closely related to chapter 4.5, which deals with the particles around the AUV. For example, the number of particles around the AUV is increased based on how close the AUV is to the black smoker. The collision boxes in the black smoker blueprint also have an important role there. These replace the ocean current modifiers of the black smoker for the particles around the AUV, so this is explained in chapter 4.5.1.5. The chapter on future works of the particles around the AUV is also related to the future works of the black smoker. To avoid having to describe things twice, I will only describe it once in chapter 4.5.5.

4.5 Particles around the AUV

Enceladus is still very unexplored, so we do not know what kind of particles and how many particles there are. For the simulation, we assume that there are mainly organic particles in the water. Since the area in which our simulation takes place is very large, not enough particles can be placed everywhere. That's why I decided to place them in a limited area around the AUV.
The basic idea is that the particle system with the organic particles that are supposed to be around the AUV is attached to the AUV. Since the cameras are only in the area close to the AUV, the user does not even notice that the particles are only in a certain area. For each particle, the ocean current vector is added to the velocity vector of the particle. This causes the particles around the AUV to move in the ocean current direction (see chapter 6.7.1). Additionally, the particles have a noise. This ensures that each particle has a small random standard velocity so that it looks as if the particles are floating in the water. Otherwise, the particles would be completely static in the water if there were no ocean currents and this would look very unrealistic.
I have created several sprites for the organic particles myself. When the particle system spawns, each particle is assigned a random sprite. Furthermore, the particles around the AUV interact with the ocean current modifier replacements from the black smoker. If a particle enters one of the two ocean current modifier replacements, it is pushed upwards.

4.5.1 Implementation

The exact implementation can be looked up in the particle system Content/TripleSim/VFX/-Niagara/Particles/ParticleAroundAUV and the corresponding blueprint Content/TripleSim/Particle/BP_ParticleAroundAUV. Everything is commented. The basic implementation of the particles around the AUV is as described in chapter 3.4.5 so that these particles can be detected by the laser.

4.5.1.1 Box containing the particles

To define the area in which the particles should be around the AUV, a user parameter of type vector was created. This specifies the length, width and height of the box in which

the particles should be located. This user parameter is used to spawn all particles within this box. It is also used in the scratchpad Check if Outside Scratchpad. This checks whether the particle is still inside the box. If the particle is no longer inside the box, it will be teleported to a random position inside the box. Let's assume that the dimensions of the box are x=1000, y=1000, z=1000 and the AUV is exactly in the middle of this box. Then there are particles up to 500cm above, below, in front, behind, to the left and to the right of the AUV.

4.5.1.2 Movement of the particles

Since the particle system is attached to the AUV, the problem occurs that the particles also moves with the AUV. To solve this problem, I calculate in the AUV blueprint how many cm the AUV has moved in each direction. I always calculate this for one frame, i.e. the distance in x, y and z direction that was covered between the last frame and the current frame. In this context, it is also determined whether the AUV has moved in the respective positive or negative direction. Based on this, the value is determined that each particle must have added to its current position so that it does not move with the AUV. This value is sent from the AUV blueprint to the particle system and is added to each particle in the scratchpad Subtract AUVDistance Traveled Scratchpad. This allows the AUV to move forward while the particles remain at the same position. The rotation of the particle system is reset to 0 in each frame in the AUV blueprint so that the particle system does not automatically rotate when the AUV rotates.
To avoid the particles from being completely static, the Curls Noise Force module is used to give each particle a small random velocity. This makes the particles less static if there are no ocean currents.
The vector from the ocean currents is added to each particle via the self-created ParticleVelocity user parameter. BP_ParticleAroundAUV gets the vector of the ocean currents and then sets the ocean current vector in the user parameter ParticleVelocity. ParticleVelocity is then used as input for the Add Velocity and Linear Force modules in the particle system. As a result, each particle ends up with its small standard velocity due to the Curl Noise Force and the ocean current vector is also added to this velocity.

4.5.1.3 Particle sprites

The material that the particles use as a sprite can be found under the following path: Content/TripleSim/Assets/Materials/Particles/M_OrganicParticle.
In order to have different shapes and colors for the organic particles, I work with two switch nodes in the material. The first switch determines the color of the particle and the second switch determines the shape of the particle. Mixed textures and simple colors are used for the color of the material. The colors have darker green, brown and gray tones. I generated the textures for the color using Filter Forge [17, 18]. The second switch is responsible for the shape and the alpha value of the particle. I found suitable textures for this from various sources [1, 20, 40] on the internet. Both switches have 10 inputs, so there are 10 different organic particle sprites at the end.
To assign a random shape and color for a sprite, a random number between 0 and 9 must be generated. This allows a random color and a random shape to be selected in the material via the switches. The Dynamic Material Parameters module must be used in the particle system so that a random number can be generated and passed to the material. This is therefore responsible for the communication between the particle system and the material. Figure 4.16 shows a few of the different sprites for the organic particles around the AUV.

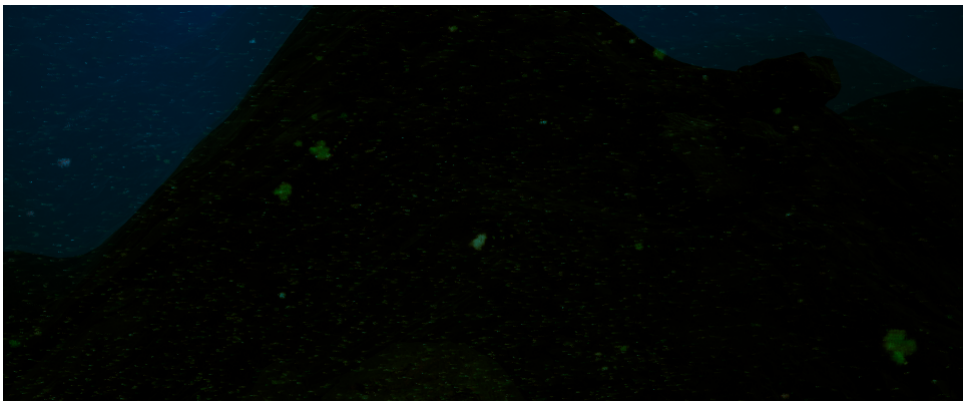

Figure 4.16: Sprites of the organic particles

4.5.1.4 Illumination of the particles

In the particle system, the Overlapping Scratchpad is used as described in chapter 3.4.5. This was extended for the particles around the AUV so that the particles reflect the light when the laser hits them. The standard Overlapping Scratchpad checks in each frame whether the particle is inside the laser. This has been extended so that the particles light up in the color of the laser if they are inside the laser. Underwater lasers often use a green color with a wavelength between 510nm and 530nm [38]. That is why our laser uses a green color with a wavelength of 520nm. The emissive color in the particle material makes the particles glow. To change the emissive color during runtime, I use the color and the alpha value of the particle color from the particle system. The color for the glow has the same color as the laser to suggest that the particles reflect the light from the laser. In the particle system, the alpha value of the particle color is set to 0 in the Overlapping Scratchpad if the particle is not in the laser. If it is in the laser, the alpha value is increased again. In the material, the particle color and the sprite size of a particle can be accessed from the Niagara system. There, the sprite size of the particle is multiplied by the StrengthEmissiveColor parameter. This value determines how strongly the particles reflect the light. The strength of the reflection is therefore also related to the size of the particle. StrengthEmissiveColor can be changed as required if the particles should reflect more or less in the future. The calculated value is multiplied by the alpha value of the particle color and the result is transferred as the emissive color. As a result, every particle that is hit by the laser reflects the light of the laser.
The particles around the AUV are also visible through other light sources, such as directional light or skylight. The AUV also has a normal light (spotlight), which also makes the particles visible. In the realistic mode of the simulation, there are no light sources other than the light from the AUV. This ensures that only the particles that are in the cone of the AUV light are visible, all other particles that are not in the cone are not visible. This also makes it possible to see the light cone of the light on the AUV. In the other mode of the simulation is a directional light and a skylight. This mode is intended to allow the user to see more of the surroundings by providing more lighting and a better view. This makes it easier to see the black smokers, for example. In this mode, all particles are also illuminated by the directional light and the sky light. This means that all particles around the AUV are always visible. Figure 4.17 shows on the left side an image where only the light from the AUV is activated and the directional light and skylight are deactivated. It is easy to see that only the particles that are illuminated by the cone are visible. Furthermore, the laser is activated and there are a few particles in the laser that reflect the laser. On the right

side is an image in which the directional light and skylight are activated. There you can clearly see how all the particles are visible because they are illuminated by the directional light and skylight. The laser is also activated there and particles reflect the light of it.

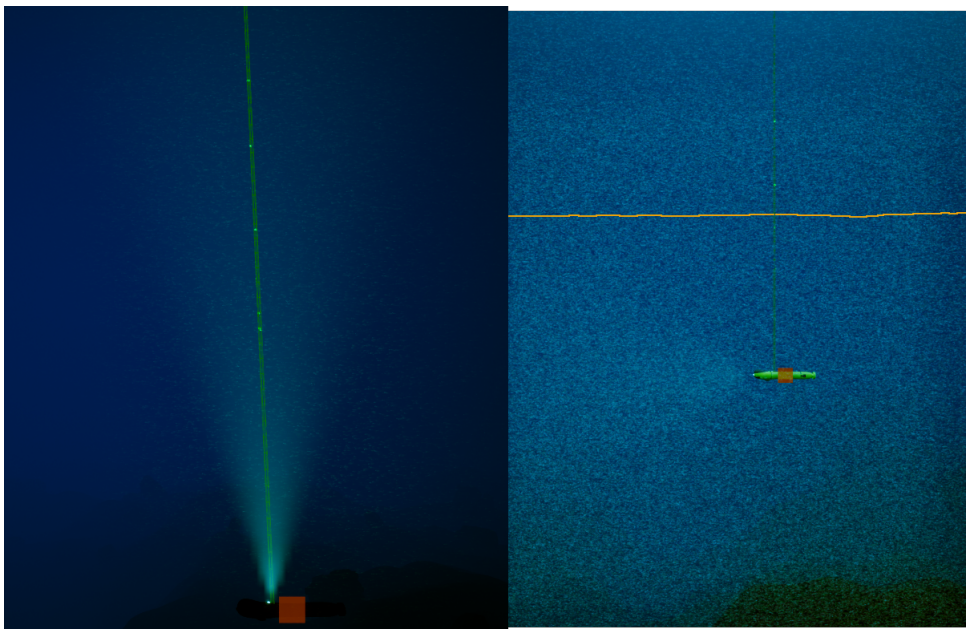

Figure 4.17: Particles around the AUV illuminated only by the AUV light vs. illuminated by directional light and skylight

4.5.1.5 Ocean current modifier replacement

This part is related to chapter 4.4. It would be an advantage if this was read beforehand.
Black smokers use current modifiers (see chapter 6.7.1) at the exit point of the smoke, as the smoke first flows upwards and then over time in the normal ocean current direction. An array with all ocean current modifiers would have to be sent to the particle system, where a loop would have to be used to check whether and in which ocean current modifier a particle is located. The vector of the ocean current modifier could then be added to the velocity of the particle. The problem is that no loops can be used in the Niagara scratchpads. Accordingly, it is not possible to iterate through the ocean current modifier array. Theoretically, it would be possible to use loops with a workaround by using custom HLSL (High Level Shading Language) code and looping there. However, Niagara and the custom HLSL node explicitly state that no loops or complex if nesting should be used, as otherwise the particle system will become very slow and the performance will be extremely poor. I have therefore decided on a workaround for the ocean current modifier, which I call ocean current modifier replacement.
Each black smoker has exactly two collision boxes to replace the ocean current modifiers. These collision boxes have been placed in the black smoker so that they cover the lower part of the plume where the normal ocean current modifiers would be. The dimensions of the two collision boxes are sent to the Niagara system via user parameters. In the Niagara system, two scratchpads are used to check whether one of the particles from the particles around the AUV is in one of the two collision boxes (ocean current modifier replacement). If a particle is in a collision box, it is pushed towards the z-direction until it is no longer in the collision boxes.
To save performance, I check the distance between the AUV and the nearest black smoker.

If the distance is so small that a particle could be in the ocean current modifier replacement, the check is performed to see if there is a particle in one of the ocean current modifier replacements. To ensure that the correct black smoker is always defined as the closest black smoker, the BP_ParticleAroundAUV checks every 30 seconds whether there is a new closest black smoker. Based on this, the two ocean current modifier replacements are then set in the user parameters of the particle system.

The real ocean current modifiers have way more cone segments or octrees (see chapter 6.7.1), while these ocean current modifier replacements only have two collision boxes. That is why it is not quite as accurate as the normal ocean current modifiers. These are much finer or smaller. However, the ocean current modifier replacements are better than nothing. As a result, the particles around the AUV interact with the exit point of the plume. Figure 4.18 shows the two ocean current modifier replacements of the black smoker. These are shown in a red color not visible in the final product.

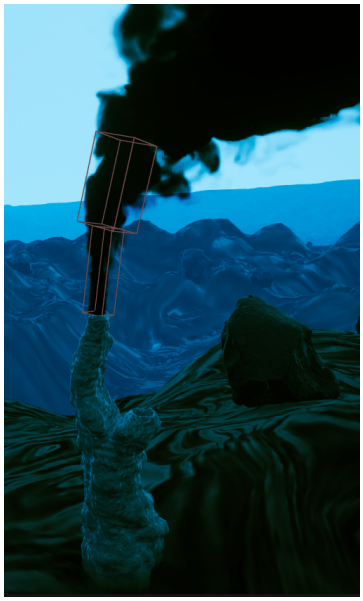

Figure 4.18: Ocean current modifier replacement

4.5.2 Change the properties of the particle system

The most important properties, their current values and where they can be changed are described below.
In the Spawn Burst Instantaneous module, Spawn Count can be used to specify how many particles should be around the AUV. The current value is 600,000. Please note that more particles also increase the resources required.
In the Initialize Particle module, the minimum size of a particle can be defined with Uniform Sprite Size Min and the maximum size with Uniform Sprite Size Max. A random value between these two values is taken for the particle size. The current values are 0.1 and 0.7, which means that all particles have a random size between 0.1cm and 0.7cm.
In the ParticlesAroundAUV particle system, the noise that each particle should have can be defined in the Curl Noise Force module. The Noise Strength variable can be used to specify the strength of the noise and Noise Frequency can be used to specify how often the noise is recalculated. Noise Strength is currently set to 0.5 and Noise Frequency to

50. The noise can also be completely removed by deleting or deactivating the module.
The SpawnBoxDimensions user parameter defines the size of the box around the AUV in which the particles are located. The current value is 1000 for x, y and z. This means that the box is 1000cm high, 1000cm wide and 1000cm long. It is important to note that this box should be at least twice as large in each direction as the laser is long. This means that the laser should be completely covered by the particles so that the simulation for the laser detection is still correct. The laser is currently 500cm long. This means that no matter in which direction the AUV or the laser is rotated, the laser is still completely covered by this box. Let's assume, for example, that this box only goes 500cm in each direction. If the laser is now pointing straight upwards, 250cm of it would not be covered by the particles. However, it should be noted that the density changes when the box size is changed, as the same number of particles are still distributed over a larger or smaller area.
The strength of the reflection of the laser light when a particle is hit by the laser can be defined in the particle material (M_OrganicParticle). The strength is defined via the StrengthEmissiveColor parameter. The current value is 10,000. The textures for the sprites can of course also be exchanged in the material.

4.5.3 Increase particles around AUV based on black smoker

Towards the end of the project, the idea was introduced that the particles around the AUV should increase as the AUV moves closer to a black smoker. For this purpose, a second particle system is used to simulate the particles from the black smoker that are distributed in the water over time. It can be found under the following path: Content/TripleSim/VFX/Niagara/Particles/ParticlesFromSmoker. The corresponding blueprint is under the path: Content/TripleSim/Particle/BP_ParticleAroundAUVFromSmoker.
The particle system and blueprint have an identical construction to the organic particles around the AUV and are also attached to the AUV. However, these particles are not visible. There are a total of 300,000 particles around the AUV, which should represent the particles coming from the black smoker. The SmokerDensityMultiply value of these particles is changed based on how close the AUV is to a black smoker. If the AUV is very far away from the black smoker, SmokerDensityMultiply is 0. If the AUV is right next to the black smoker, the value is 20. Depending on how close the AUV is, the value can be between 0 and 20. Only the SmokerDensityMultiply value is increased and not the number of particles, as this could otherwise lead to performance problems.
In addition, I encountered a problem. My first idea was to increase the number of particles and not just the SmokerDensityMultiply value. To implement this, I would need the number of particles. This value would have to be read from the Engine Provided Attribute NumParticles. This should contain the current number of particles in the system. I came across a bug that has not yet been fixed by Unreal. It seems that it is not possible to read out this value in a GPU simulation. It always returns 0. Then I tried a CPU simulation and it always gave me the correct value. However, since I am working with several hundred thousand particles, the GPU simulation must be used.
Since the idea only came up at the end of the project, the quickest and simplest solution was used. There are better and more realistic methods that are not based on how close the AUV is to the black smoker, but on the relative position between the black smoker and the AUV. For example, the particles only increase if the AUV is on the side of the smoker where the ocean currents flow. Chapter 4.5.5 describes a better implementation idea that can be implemented in the future.

4.5.4 Challenges

As up to 900,000 real particles are sometimes active at the same time, the biggest challenge was dealing with so many particles at the same time. It was important to implement things in such a way that they are as efficient as possible. Another challenge was that loops cannot be used in Niagara scratchpads. This meant I was much more limited and had to be creative to replace the ocean current modifiers with something else. This is how I came up with the idea of ocean current modifier replacements, which were discussed in chapter 4.5.1.5.

4.5.5 Future works

In the future, the increase of the particles can be improved if the AUV is closer to the black smoker. I have two possible solutions that I unfortunately didn't have time to implement in the end.
The first idea would be to use another particle system that starts from the plume of the black smoker. The particle system must of course be implemented as described in chapter 3.4.5 so that the particles can be detected by the laser. The particles are given a small noise and the vector of the ocean currents is added. This ensures that the particles flow from the black smoker in the ocean current direction. The particles can then be given a different lifetime using a random range float. For example, you could define that 50% randomly live between 2 and 10 seconds, 20% between 10 and 30 seconds, 10% between 30 and 70 seconds, and so on. These are only example numbers, the correct ones must be determined by testing. You can also create much finer or more transitions to make it more realistic.
The second idea is to shoot several rays from the plume of the black smoker. The rays are shot in the direction of the ocean currents. When a ray hits the AUV, the distance between the AUV and the black smoker is determined. Based on this, the number of particles around the AUV is increased. This means that the particles around the AUV are only increased if the AUV is in the path of the ocean currents. It is not so realistic if the ocean currents would flow in the opposite direction of the AUV and the number of particles around the AUV would still increase.

Another idea that was mentioned by the project managers at the end is that there should be more particles at the ice shelf. This could be implemented, for example, by creating a particle system that can be detected by the laser. This particle system is placed at the top of the ice shelf and the ocean current is added to the velocity of the particles. In this case, the particles should not exist forever, but only for a certain period of time. However, a few particles should die over time so that you don't end up with an infinite number of particles. Because the particles die and respawn, they are only located near the ice shelf. To avoid having to cover the entire ice shelf with these particles, you can set a smaller area for the particle system. In a blueprint, only the x and y coordinates of the AUV could then be read out and passed to the particle system. This means that the particle system is always above the AUV at the ice shelf and does not consume so many resources. Here you could also define that some particles only live between 5 and 10 seconds, some between 10 and 20, and so on. This means that you have more particles at the ice shelf, which decrease more and more towards the bottom. Here, of course, it is important to find suitable values.

4.6 VFX

This chapter covers other particle systems and visual effects that were created during the project.

4.6.1 1. Particle System

The project should include a simulation of particles in water. For the first milestone, a prototype of such a particle system should therefore be implemented. The visualization is first prioritized and some basic functions for the distribution of particles in the scene are implemented.

The Niagara plugin included in Unreal is used to simulate these particles. It is a powerful visual effects system that allows to create complex, dynamic particle effects within Unreal. It offers a node-based interface for designing particle behavior and appearance. Niagara also supports advanced features such as GPU simulation, data-driven behaviors, and customizable modules. Every particle system starts by emitting particles from a given emitter. These emitters spawn particles in separate ways, giving them an individual starting behaviour. After this initial force application, this will in most cases be then modified by applying other forces regularily, for example on a tick function. An example for such a force is drag, which will continuously slow down the particle.

Figure 4.19: Niagara setup

This setup includes three different types of particles. These differ in the size and shape of the particles as well as in their behavior. The particles each receive different forces in the update. In addition, noise is added to each particle so that no two particles behave completely the same. Since new particles are constantly being spawned, they have a limited lifespan and disappear after this time. This simulation uses CPU as a simulation target. Even if better performance could be achieved with a GPU, it is not relevant in

this case as not enough particles are spawned to cause serious performance problems. CPU also offers the advantage over GPU that complete collision detection with all other objects, meshes, etc. is possible. This is not possible when using the GPU.

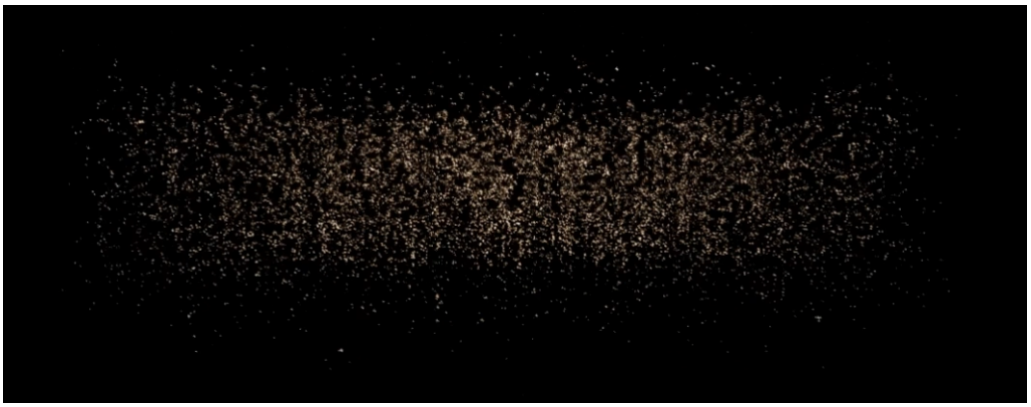

Figure 4.20: Isolated view of the finished particle effect from a side perspective

The particles have randomly chosen sizes ranging from a few millimeters to about a centimeter. In this way they are just visible in the viewport of the simulation and break the otherwise monotonous impression of the scene. This means they have exactly the effect that was intended to be achieved. In the next step, this particle system is then expanded to include additional interactions, for example an interaction with ocean currents and black smokers is added (see 4.5).

4.6.2 Underwater shader

For our project, I made an underwater shader, which should give the user the feeling that he is really underwater. I created a screen warping effect for this. This was created using two distortion texture samples where the UVs are shifted over time. The coordinates of these are then used to sample the scene texture. Furthermore, a vignette mask with a lens distortion was created. The further it goes to the edge of the vignette mask, the more the image is blurred. As a result, the center of the image is completely sharp, while it becomes blurrier towards the outside.
The shader also has a custom fog. This is created by lerping between the scene depth and the desired fog color. It is possible to set where the fog starts and how strong the fog increases until it reaches its maximum strength. The problem with the custom fog is that the smoke from the black smoker was displayed very strange or was not displayed at all in some cases. If there was no object behind the smoke, this phenomenon occurred. It seems that the scene depth node in the material does not work correctly with Niagara Fluids. Maybe this bug will be fixed by Unreal in the future. Afterward, we simply decided to use the exponential height fog of the Unreal Engine.
The underwater shader is not used in the final demo level, as it displays the image rather from a first-person perspective of a human, similar to how it is done in video games. However, since our simulation is also about precisely seeing and measuring things, the shader is not used. The shader can still be found under the following path: Content/TripleSim/Assets/Materials/UnderwaterShader/UnderwaterPostProcess. The shader must be added to a post process volume in the level. Figure 4.21 shows what the image looks like with and without the shader.

Figure 4.21: Underwater shader

4.6.3 Schlieren effect

At the beginning of the project it was discussed that there could be a schlieren effect underwater. Therefore, I created several materials that simulate the schlieren effect. These can be found under the following path: Content/TripleSim/Assets/Materials/Schlieren.
The basic idea of all materials is to use a texture and manipulate or change its UV mapping over time. This is then used as input for the Normal of the material. In addition, a refraction is used to refract the image a little. The schlieren effect is not used in our final demo level as there was no specific use case for it. One schlieren effect is shown in figure 4.22.

Figure 4.22: Schlieren effect

4.6.4 Bioluminescence

Bioluminescence is a kind of natural light source that can occur in water. These are produced, for example, by bacteria such as Vibrio fischeri. These produce light through mechanical stimulation [10, 27]. This phenomenon can be seen in a video [5]. For our simulation, I have created two Niagara particle systems that simulate bioluminescence. These can be found under the following path: Content/TripleSim/VFX/Niagara/Bioluminescence.
The first particle system uses normal Niagara, accordingly it uses 2D sprites which are rotated to the camera. The Spawn Per Unit module is used in the emitter. This ensures that the particles are generated based on how fast the particle system is moving in the level. This means that if the particle system is not moving, no particles are generated. I created my own material for the particles, which consists of a bluish emissive color. This causes the particles to generate light in the environment.
The idea is to attach the particle system to the AUV and place trigger boxes at various points in the level. The trigger boxes should ensure that the particle system is activated at certain areas and then deactivated again, so that it seems as if bioluminescence has been generated by the mechanical stimulation of the AUV. The advantage of this particle system is that it uses 2D sprites and therefore requires significantly less resources.
The second particle system uses smoke from Niagara Fluids. This has been adapted so that it also emits a bluish light and also uses the Spawn Per Unit module. This particle system looks better than the first one, as a much smoother and finer transition can be created. The disadvantage is that it requires significantly more resources.
The particle systems are both finished, but there was not enough time in the end to integrate this effect into the final demo level. To do this, the trigger boxes would have to be placed in the level and a blueprint with one of the particle systems would have to be created. In this blueprint, an event would have to be implemented that activates or deactivates the particle system when the AUV enters a trigger box. Figure 4.23 shows the particle system with normal Niagara on the left and the particle system with Niagara Fluids on the right.

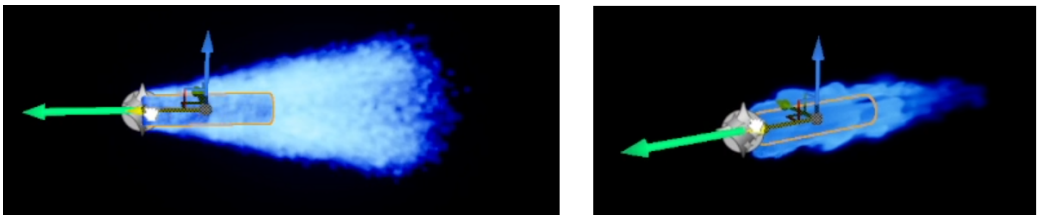

Figure 4.23: Bioluminescence

4.6.5 Ice particles

For our simulation, I created ice particles that slowly fall from the ice shelf until they melt in the water after a certain time and disappear completely. The ice particles also reflect the light from the laser. The blueprint can be found under the following path: Content/TripleSim/Particle/BP_IceParticle.
To ensure that the ice particles do not look all the same, I generated six different sprites using Filter Forge [31]. The Dynamic Material Parameter module is used to give a particle a random sprite. The particles have a random size between 1cm and 5cm and the lifetime of a particle is based on its size. This was implemented using a scratchpad. The particles become smaller over time to suggest that they melt slowly. The Curl Noise Force module is used to ensure that not every particle behaves in exactly the same way. This adds an individual small noise to the velocity of each particle. The interaction with the laser was implemented as described in chapter 3.4.5.
The ice particles are not used in the final demo level as they look cool but are probably not realistic. Figure 4.24 shows a part of the ice particles at the ice shelf.

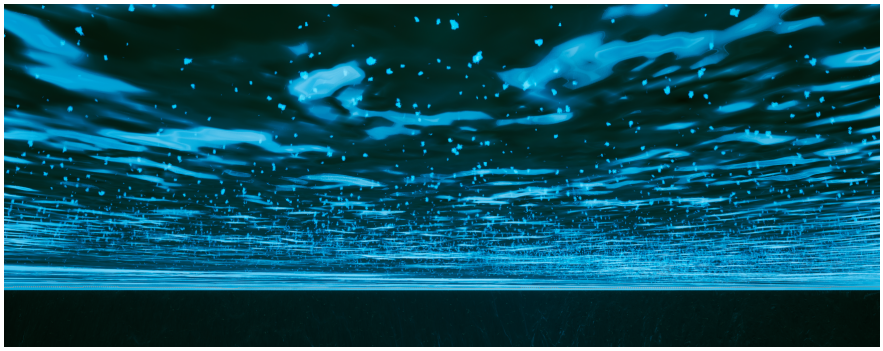

Figure 4.24: Ice particles

4.6.6 Bubbles

Enceladus is not very explored yet, so we don't know much about the environment. Since there is the phenomenon of methane bubbles [13] on the seabed, I decided to simulate them using a Niagara Fluids particle system. This makes the level look more lively and less empty. The final blueprint with the particle system can be found under the following path: Content/TripleSim/Bubbles/BP_Bubbles.
The ParticleSurfacing emitter serves as a template for the bubble particle system. This already generates a kind of liquid. The emitter was adjusted so that bubbles are generated which initially go upwards. In addition, a self-created user parameter is used to add the ocean current vector to the velocity of the bubbles so that the bubbles flow in the direction

of the ocean currents. The material instance MI_WaterSDF can be adjusted in the Water Renderer. For example, the color, specular and opacity of the bubbles can be defined there.
The previously mentioned blueprint consists of 3 bubble particle systems and a 3D model. The 3D model is a rocky ground with cracks from which the bubbles come out. I got the 3D model for the cracked ground from the Quixel Bridge. The blueprint can be placed as often as required in a level. Figure 4.25 shows a picture of the bubbles.

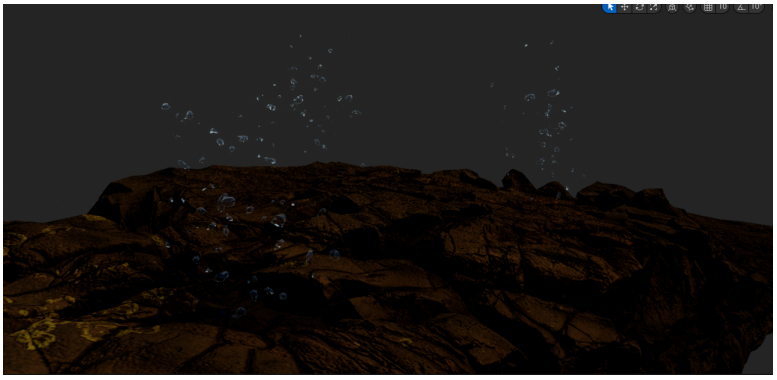

Figure 4.25: Bubbles

4.7 Terrain

This chapter is about the development of our terrain. It deals with the terrain of our first prototype and the subsequent versions as well as the final version.

4.7.1 1. Prototype

I created the terrain for our first prototype level so that all the sensors could be tested directly in a level. I did the sculpting in the Unreal Engine and tried to create an uneven ground. Furthermore, I gave the ground a rocky texture, which is from the Quixel Bridge [35]. The ice shelf is a simple plane, which also got an ice texture from the Quixel Bridge. In addition, I adjusted the color of the image with a post process volume so that the environment looks like it is underwater. Figure 4.26 shows the first prototype level.

Figure 4.26: First prototype

4.7.2 Later Versions

This first prototype, which was mainly used to prototype other objects such as the first sensors, was to be replaced by a more realistic scene as the project progressed. As a result of the first milestone (see 1.2.1), the dimensions that the scene should have became clear. A size of 2.5km^2 and a depth of between 500 and 1000 metres was agreed upon. In addition to the size, the environmental details should also be adjusted.

It is not known what exactly it looks like at the bottom of a larger lake on an icy moon. The only scientific investigations are based on the flyby of probes, such as the Cassini probe, which was able to collect data by flying past the icy moon Enceladus ([19]). This allows various assumptions to be made about the expected terrain. Although it is possible that life could exist on these planets, it is to be expected that these are simple life forms ([30]). More complex life forms such as plants are not to be expected. There is a possibility that hydrothermal vents are present at the bottom; these would be an interesting opportunity to investigate where life could arise ([22]). For Enceladus, measurements from Cassini also suggest that there is an ocean about 10 kilometers deep beneath a 30 to 40 kilometer thick layer of ice ([2]). The following image gives an idea of what it might look like at the bottom of such an ocean:

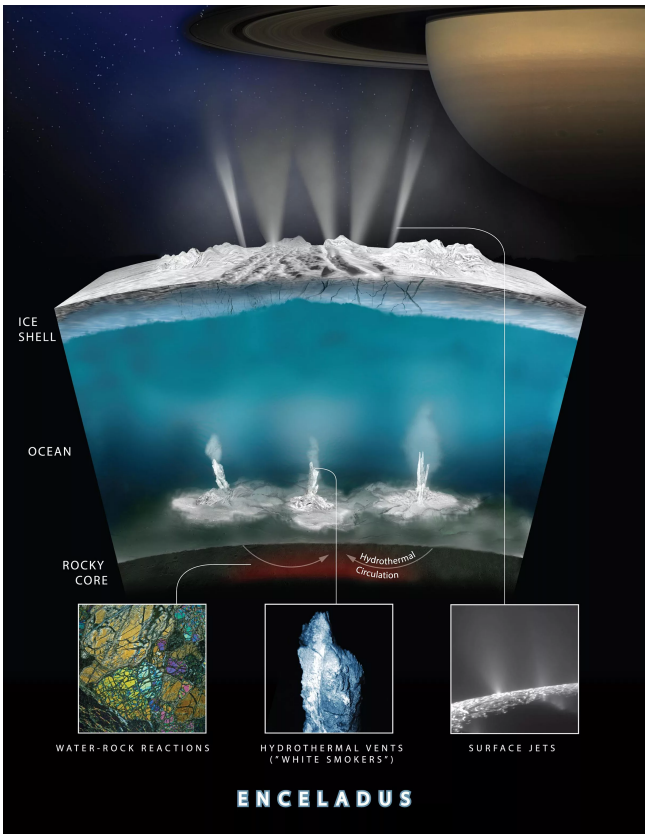

Figure 4.27: Enceladus sea floor ([29])

This results in some changes that should be made to the scene:

- General shape of a lake
- Limited on the top by an ice cover
- The ground should not be flat, but varied
- Stones on the seabed

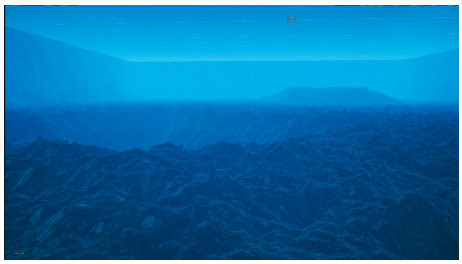

Figure 4.28: Seabed

Figure 4.29: Foliage

For this project it is assumed that the scene is constrained in all directions by rising ground. An alternative would have been open edges. However, since the shape of a lake seems just as plausible, we decided on this. The scene is limited to the top by a sheet of ice. Even though this is expected to be much thicker, a thickness of around 50 meters is assumed in this project because no simulation is planned on the ice layer. The process of ice melting should also not be included, so an accurate simulation makes no sense in

this case. Even if no plants or similar are to be expected on the lake bottom, at least some objects such as stones and rocks should be added to make the bottom appear more varied.

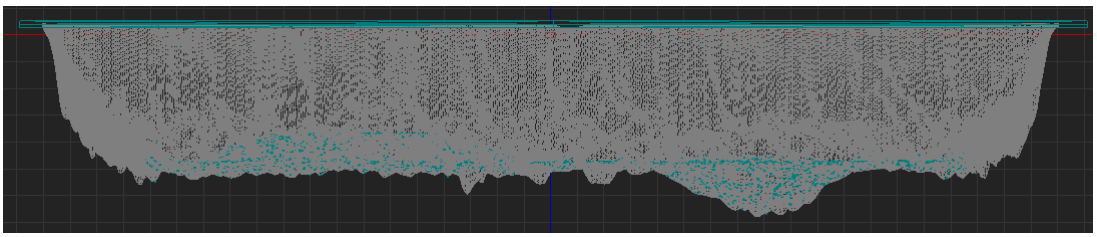

Figure 4.30: Scene sideview 1

As can be seen in figure 4.30, the edges of the scene are relatively steep. This is due to the fact that in Unreal's landscaping tool the maximum depth is limited to 400 meters, which made it difficult to reach the defined depth and achieve a realistic gradient. Overall, however, this represents a problem because the agreed depth of 500 to 1000 meters cannot be achieved in this way. Therefore, the scene was changed accordingly as the project progressed.

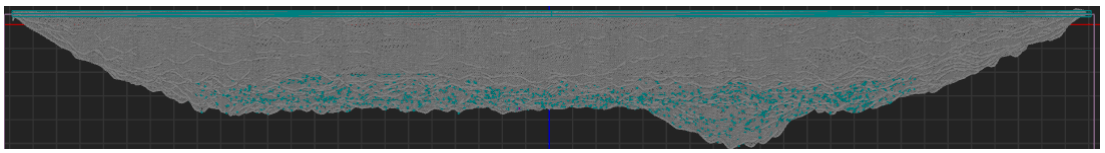

Figure 4.31: Scene sideview 2

In this figure you can see that the edges are less steep. Overall, a shape was achieved that more closely corresponds to the natural bottom of a lake. The depth of the scene has also been adjusted, which is not clear from the illustration. In Unreal it is possible to adjust the scaling of individual axes of the landscape. In this case we decided to double the scale of the z axis. This achieves a maximum depth of up to 700 meters, which is within the defined range.

4.8 Blending with Runtime Virtual Texturing

In order to place objects realistically on our landscape, I use Runtime Virtual Texturing [12] to blend the material from the landscape with the material from the 3D model on the landscape. This effect makes the objects really look as if they are part of the environment and not just placed on top of the landscape. This also means that there is no hard switch or edges between the landscape and 3D model.

Two RVTs of the landscape were used for the implementation. The first RVT contains the World Height and the other the Base Color, Normal, Roughness and Specular. It is important to note that the two RVTs must be set up again if the landscape is changed. The RVTs are located in the folder: Content/TripleSim/Assets/VirtualTextures.
The blending was implemented in a material function. This can be found under the following path: Content/TripleSim/Assets/MaterialFunctions/MF_VT. The material function receives material attributes as input. These are the attributes that come from the material of the 3D model. The material attributes are blended with the RVT with the Base Color,

Normal, Roughness and Specular of the landscape. To ensure that the system knows exactly where to blend, the RVT with the World Height of the landscape is used. This is subtracted from the Absolute World Position and added to the Object Bounds of the 3D model.
I have created some parameters so that you can customize the blending a little. These include the VirtualBlendHeight parameter, which specifies how high the blending should go for the 3D model, and VirtualFallOff, which specifies how smooth the transition should be.
The material function must be integrated into the parent material for the blending to work. To do this, the material function is simply dragged into the parent material, which creates the material function as a node. The old output of the parent material is dragged into the material function node and the result of this is dragged into the output. An example of this can be seen in the black smoker material. As a result, each material instance of the material has the blending with Runtime Virtual Texturing.
The VirtualBlendHeight and VirtualFalloff parameters mentioned above can then be adjusted in the material instance. There is also the UseVT parameter. This can be used to turn the blending with RVT on or off. All parameters can also be adjusted during runtime.

The blending with RVT is currently used for the black smoker and the bubble blueprints, but can easily be added for new objects in the future by using the material function. Figure 4.32 shows in the top image where you can find and adjust the parameters in the material instance. At the bottom left, the black smoker is displayed without blending and at the bottom right with blending.

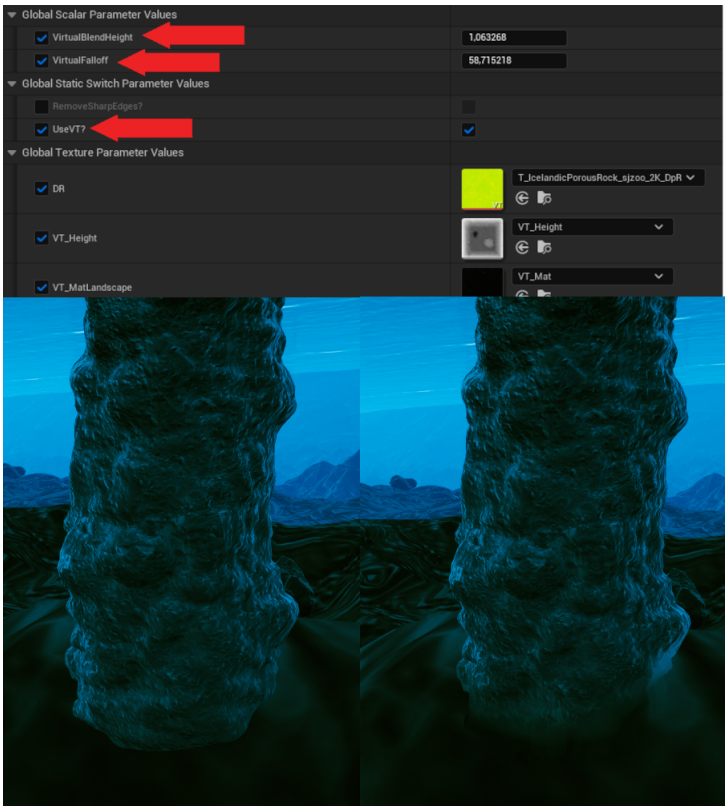

Figure 4.32: Blending with Runtime Virtual Textures

5. ROS System

To simulate vehicle movement, navigation and path planning a ROS2 based system developed by MARUM is used.

We used it to control the AUV visualized in the Unreal project and have it react to controls and other environmental forces.

5.1 Components

The ROS2 system consists of different nodes that each having a clear defined purpose.

In general it takes external disturbances, optional manual input and multiple control signals as input and outputs the vehicle state as odometry/pose and the trajectory.

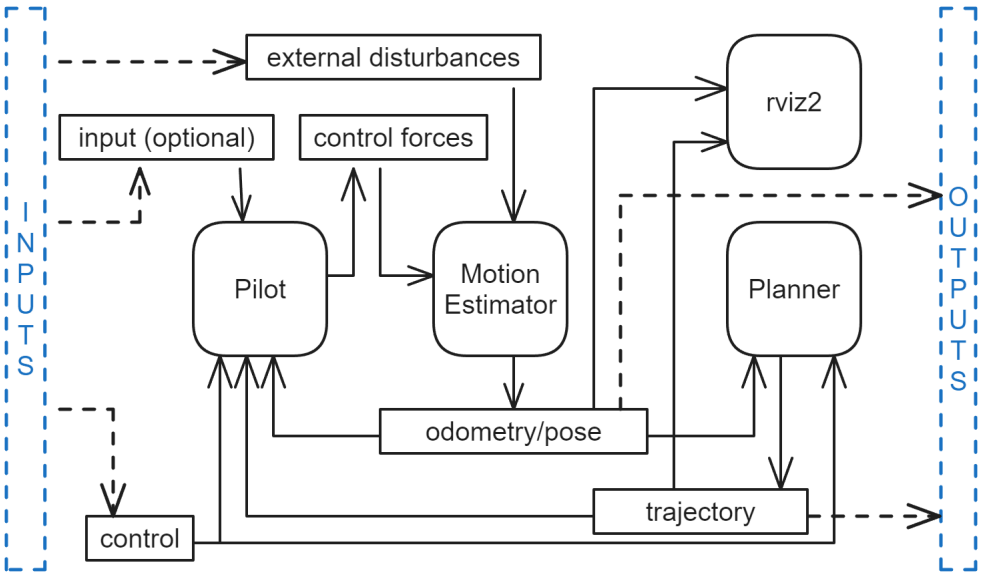

Figure 5.1: ROS2 system overview

The Motion Estimator node takes the body forces from the pilot node and external disturbances and outputs the estimated position, rotation and velocity based on the calculation of the dynamics model.

The Planner node uses a predefined algorithm or analytical function with a given set of configuration parameters to generate and output a trajectory of points for the navigation of the UV. The generation advances the trajectory based on time that elapsed and optionally the current AUV position.

The Pilot node uses either the trajectory of the planner node or manual input to generate the forces to steer the vehicle.

The rviz2 node displays the position and rotation of the UV from the motion estimator and the generated trajectory of the planner node in a 3D viewer.

5.2 Changes

During the project multiple changes were made to the ROS2 system to fix problems that came up or fulfill new requirements.

The first change that was implemented is in regard to the ocean current simulation to make it work with the already existing calculations. There was already a 2D disturbance velocity implemented, which can used for the ocean current, this was then extended to 3D. Furthermore the old implementation only used a constant parameter loaded at the start of the simulation as the velocity, to make it dynamically changeable an additional subscriber was added to update the ocean current velocity.

The second modification that was implemented tries to fix the problem that was observed, when increasing the height amplitude to generate a trajectory that dives deeper. As the trajectory advances faster than the underwater vehicle can travel.

This leads to situations where, when running the simulation for longer than a few minutes, a clear distance between the current underwater vehicle position and the newest trajectory point can be observed.

To fix this problem a distance-based trajectory generation was implemented. There the trajectory points are only advanced forwards when the underwater vehicle is under a given distance from the newest trajectory point.

In one of the following meetings this was discussed and remarked that in some use-cases such an approach can fail to navigate the underwater vehicle correctly. One such case is when external disturbances prevent the vehicle from reaching the current trajectory point. And as in our simulation the ocean current produced by the black smokers are very fast such a case could happen.

This was solved adding an extra timer that triggers the generation of a new trajectory point when a specified amount of time elapsed without causing the normal trajectory generation.

Later during the project Christian Meurer provided an improved version. This added more realistic DOFs restrictions for the vehicle. Coupled with that for DOFs that are not actively controlled a line of sight like generation is used. Additionally it now includes the current AUV position into the trajectory generation, thereby fixing the problems mentioned before. Furthermore other performance improvements and bug fixes were made.

These changes replaced the custom trajectory implementation that was created before.

Furthermore for the battery system simulation the functionality to enable and disable the motor of the vehicle and override trajectory generation with a path to the starting position was implemented by adding additional subscribers.

5.3 Problems

During the project one problem arose for that no solution was found, that is that when executing the ROS2 System directly in Windows the simulation would result in erroneously vehicle movement This is visible as the vehicle just spins chaotically without following the trajectory.

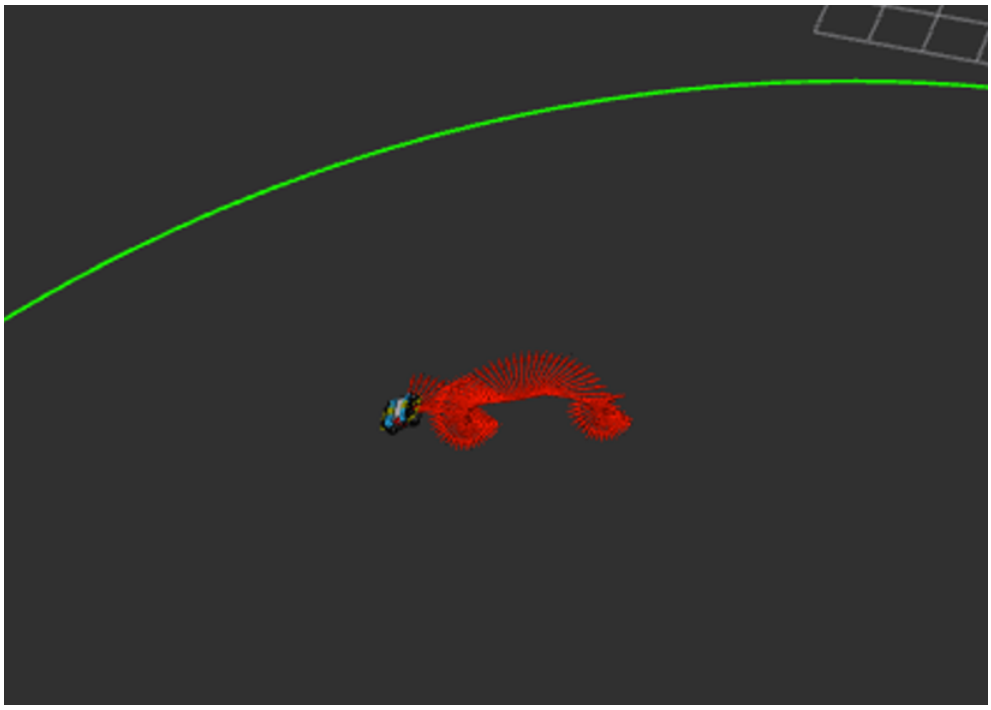

Figure 5.2: rviz2 showing the AUV spinning, red arrows showing last few orientations

As most developers using ROS2 are running it using Linux no solution to the problem was found. As such we used Windows Subsystem for Linux to run Ubuntu 22.04 on the Windows Host system.

Another variant of executing the ROS2 system would have been to use a Docker container, this was tried but due to networking problems between the Windows host system and the application running in the container not working this could not be used.

The network communication problems that happened can be related to the default transport layer implementation FastDDS that ROS2 uses as its default.

One of the reasons for the problem seemed to be that the discovery of nodes is not working correctly between the Windows host and the docker container. This is partly the case because ROS2 uses shared memory for communication as it has the smallest performance overhead. To change this it is nessesary to create a configuration file for FastDDS.

Multiple different FastDDS and Docker container configurations were tried but none were working.

The only way that ROS2 networking had no problems using Docker containers, was for the case of container to container communication. This resulted in the search of a way to containerize the Unreal Engine project, more regarding that in chapter 6.8.

6. Unreal Project

The following chapters contain detailed information about the Unreal Engine project itself, the AUV, visualizations and simulations.

6.1 VaMEx-VTB

The Triplesim project is essentially based on the VaMEx-VTB project. At the beginning of the project, a clone of this project should be used, which contains all parts that are relevant for the upcoming project. Since the VaMEx-VTB project is simulated in a completely different environment and the robots used also have some differences, many of the files included are no longer needed. The following figure shows which elements were retained and outlines the data flow between the individual elements and the external ROS system:

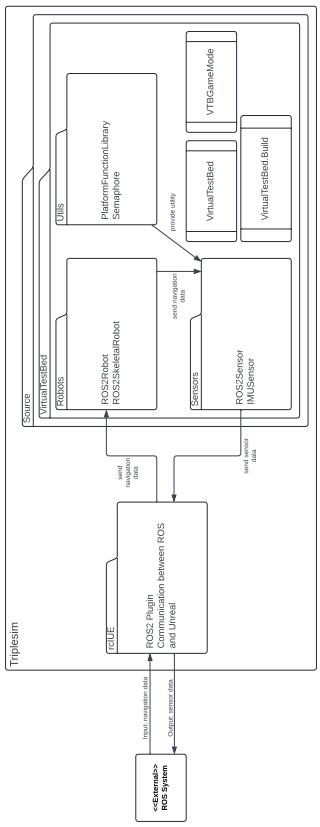

Figure 6.1: Remaining components

For clarity, some elements of packages have not been added. For example, in addition

to *VirtualTestBed.Build*, a few other build files were retained, which are, however, not relevant to the basic structure. In addition to the *rcIUE* plugin (see 6.1.2), the *Robots*, *Sensors* and *Utils* packages were essentially retained. These contain many of the basic functionalities needed to add a robot and enable ROS communication. The *IMUSensor* already contains an implementation of a sensor, which should also be implemented in this project. Other files, such as all assets, have been removed.

6.1.1 Version

The VaMEx-VTB project provided works with Unreal Engine 5.2. Since version 5.3 was already available at the start of our project, attempts were made to update the project to this version. Apart from performance improvements, this version does not contain any new features that are relevant to the project. Therefore, this point was assigned relatively little importance. Since various problems arose when updating the version, the decision was made to keep version 5.2.

6.1.2 ROS2 plugin

To integrate the Unreal Engine project with ROS2 the plugin rclUE is used. It is a fork developed by the CGVR to improve the original plugin developed by Rapyuta Robotics.

There are multiple abstractions of the plugin functionality implemented by the VaMEx project. For example there is the ROS2Robot actor which can have multiple ROS2Sensors, which are actor components, each representing the simulation and messaging of single sensor. The ROS2Robot then manages the initialization and updating of each sensor.

6.1.2.1 Usage

We used ROS2Sensors for the data publishing, they are components of the AUV blueprint, which is the main actor in which everything comes together. Subscribers regarding data of the AUV are also part of the AUV blueprint. Other subscribers have been implemented separately using extra ROS2Node actors.

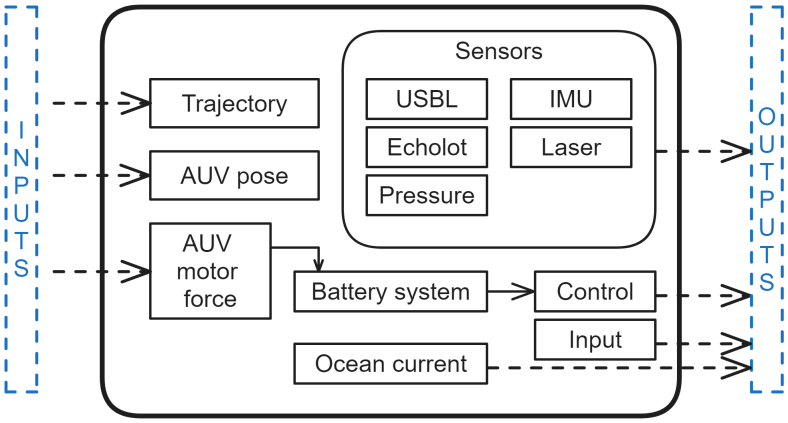

Figure 6.2: Data send and received using ROS2

6.1.2.2 Problems

There were multiple problems produced by the plugin during the project.

One problem, was that sometimes when using Live-Coding of the Unreal Engine the

plugin library files (.dll) could not be copied to the output folder because the file is already being used by Unreal Engine. Which also seemed a weird as no changes to the plugin were made and those files during the complete project never changed. The exception that was thrown was then just catched and ignored.

Another problem happens also when using Live-Coding, the engine crashes if a level is currently running because of some problems produced by the plugin.

Furthermore the plugin is not correctly configured such that when creating a Windows release/shipping build of the project the library files of the plugin were not accessible to the project, resulting in errors on startup.

Moreover the functionality of the ROS2Publisher class is not working correctly. It is not possible to create and use a publisher without using the inbuilt timer functionality. Manually calling the method to send a ROS2 message is not working. This is the reason why in many cases instead of just a ROS2Node with a publisher instead a ROS2Sensor was used to send data, as it seemed like the only way that worked 100% of the time.

6.2 Input

The input is implemented using the enhanced input system as it makes adding new inputs and different hardware easy and fast. This works by defining a input action for each thing that needs input and then combining them into a mapping context where then Keyboard buttons are assigned.

The input system in the beginning was primarily used to toggle different features on and off and for the manual input.

Additionally as the manual control is very hard, a depth stabilization was implemented. It is enabled by setting a target depth. Which internally overrides the up-axis movement of the normal manual input that is send to the vehicle dynamics system.

Later in the project the focus shifted more into using the trajectory algorithm to control the vehicle and using the check boxes implemented in the user interface to toggle features. Which lead to a more sparse usage in the final project.

6.3 AUV

The AUV is discussed in more detail in the following subchapters. This includes the 3D model and the light from the AUV.

6.3.1 3D model

For the AUV there were provided two different models over the span of the project.

The first version was a very simple capsule model. The second version was a more realistic model with fins, the turbine and more detailed geometry.

Additionally a model for the section of the melting probe that stores the AUV and deploys it with an arm was provided. But as no group member had the needed experience, no unload animation was created.

All the models were given as stl files and then using Blender converted to fbx. In the

case of the second AUV version it was separated into files for each of the different colors used in the model. Furthermore it was not possible to shade smooth the model, as the geometry was not very clean around screw holes or other finer details and every face was also duplicated as one inner and one outer face.

6.3.2 Lighting

The AUV should have a light source, which is primarily intended to ensure sufficient lighting during the camera's lighting time. However, since the demo is an important part of the project, we decided to add two different light sources. The headlights are located at the front of the AUV. There is another light on the AUV right next to the camera.

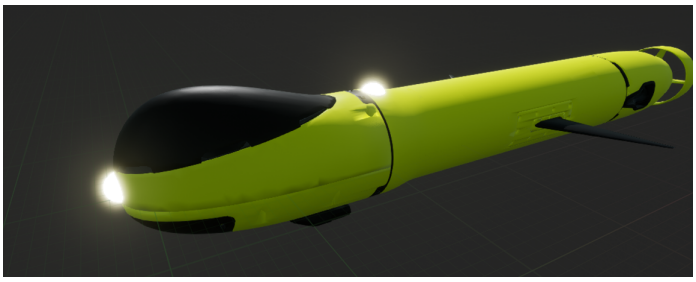

Figure 6.3: AUV emissive material

For this purpose, there are two components attached to the AUV. As you can see in the picture above, these have an emissive material that creates the impression of a lamp being switched on. Both can be turned on and off. Since the front light is only intended for demo purposes, the control is only carried out manually, whereas the light on the top can be operated both manually and via an interval switch (see 4.1.2).

In addition to the material, both have a spotlight light source that emits light in a cone-shaped area. This light only becomes visible when it collides with a surface. In order to still achieve the effect of a visible light cone, an exceptional height fog was used in this project (see 4.1)

6.4 Visualization

This chapter describes the visualization of our sensors and the path of the AUV.

6.4.1 Path & Trajectory

The trajectory generated by the planner node and the actual path that the AUV moved along are visualized as line strips. They are currently limited in the amount of points stored to render the last two minutes. As more points lead to a bigger negative performance impact.

They are implemented using Unreal Engine's render pipeline using a custom component that inherits from the primitive component and then uses a custom scene proxy to render the points. The component provides multiple ways to update the rendered line strip that then triggers a re-render with the new data. Additionally the width it is rendered with can be changed and the rendering can also be toggled on and off.

One of the problems that occurred was that the line strips were only rendered in the level when the actor on which the custom component was attached to was visible. The only solution that was found is to extend the bounds of the line strips over the complete level. There probably exists a better and more correct solution but for that more in depth knowledge of Unreal Engine's render pipeline is needed.

Another problem for that no solution was found is that the rendering order of multiple line strips is random which leads to a flickering effect where every few frames the line strip that is rendered above the other changes. This probably relates to some depth settings that is not set correctly.

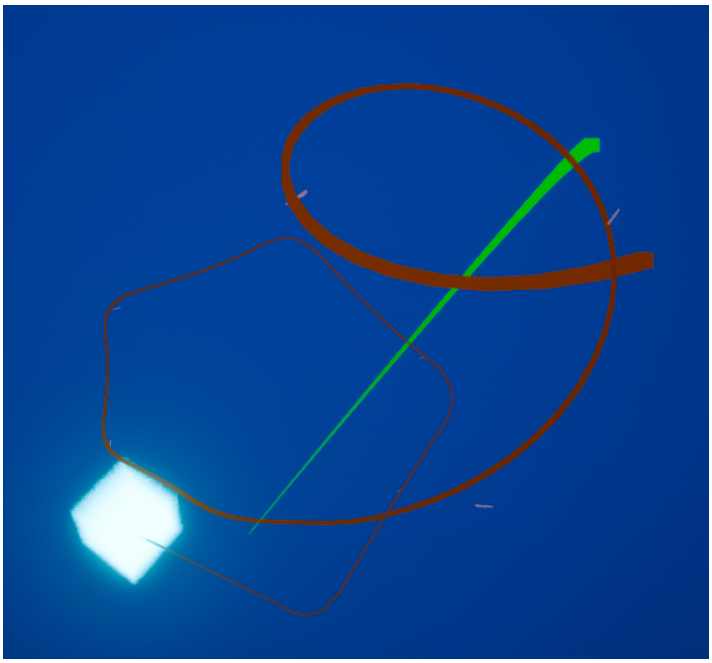

Figure 6.4: Planned trajectory (green) and actual AUV path (red), glowing cube are lit particles around AUV (see 4.5)

6.4.2 Ripple Effect

To visualize the pings from the USBL (see 3.1) and the echo sounder (see 3.2), a ripple effect was created (see figure 6.5). The idea is to spawn particles via the Niagara particle system, for which a special particle shape and color was created. A white circle is created that performs some refraction. This circle is used as particle which should be spawned by Niagara (for more information, see here [8]). It is implemented that Niagara spawns the particles in a straight line pointing downwards, also it is implemented that the particles grow over their lifetime, they also have a fixed lifetime of a few seconds, when that lifetime is done the particles are destroyed, the particles are also destroyed when they reach the AUV or the seabed. We have decided that it is sufficient to send the particles only from the base station to the AUV, and from the AUV to the seabed.

The particles are much slower than the real rays, because the real rays are very fast, about 1480 meters per second, and therefore we would not see much of the effect, even if 2 particles overlap it looks weird, that's also the reason why the particles are send from one direction. These particles can be spawned via a Unreal Engine actor called "BP_Ripple",

there are two of these actors, one on the base station and one on the AUV. The one on the base station will send the particles towards the AUV when the USBL is active. The other actor, which is on the AUV, will send the particles directly downwards according to the echo sounder. If the USBL or the echo sounder is not active, the actors do not send any particles.

Figure 6.5: The ripple effect that gets spawned from the Basestation

6.4.3 Laser and bin visualization

Collision boxes were used to implement the laser and the bins. One collision box represents one bin. Furthermore, collision boxes already have an integrated functionality to visualize them. The visualization can be seen in figure 6.6. The bins or collision boxes are visualized by an outline. In the details area of the collision box, you can use Shape Color to specify the color of the outline and Line Thickness to specify how thick the outline should be. The laser or bin visualization can be turned on and off at the top right of the Laser UI (chapter 6.17) with a mouse click. The laser visualization is turned on by default. If the entire laser is deactivated, the laser visualization is also not displayed.

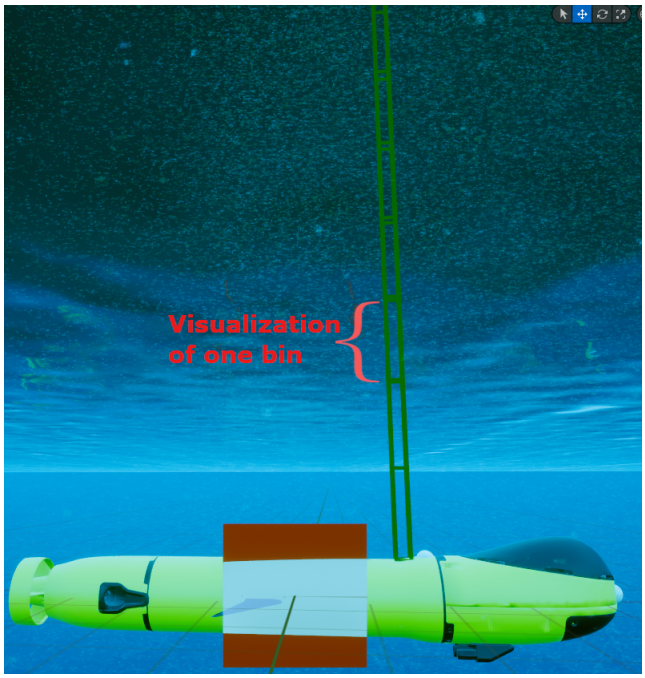

Figure 6.6: Laser bin visualization

6.5 User Interface

This chapter explains the user interface in more detail. We will look at the general layout and the individual tabs in the UI that display various data.

6.5.1 General Layout

The application has a user interface that both displays data and accepts input during the simulation. Some sensor data, such as IMU measurements or a laser heatmap, can be viewed here (see 6.5.2 and 6.5.3). There is also the option, for example, to change the vehicle's light settings. Static information, such as keyboard assignments, is also displayed.

The user interface was developed in several steps. At the beginning of development it was only planned to use the user interface to display IMU measurements. For this purpose, the first version of the UI was designed as an overlay directly in Unreal. The initial two pages could be switched through one after the other with one button. Unreal's built-in UI functionalities in the form of the UMG library were used for this. UMG provides a variety of predefined elements, which can be arranged accordingly in a graphical interface. The designer offers options for drag and drop editing as well as coding. There is also the option to develop the UI for specific screen sizes. Another useful feature are anchor points, which enable the alignment of UI elements based on their relative position on the screen.

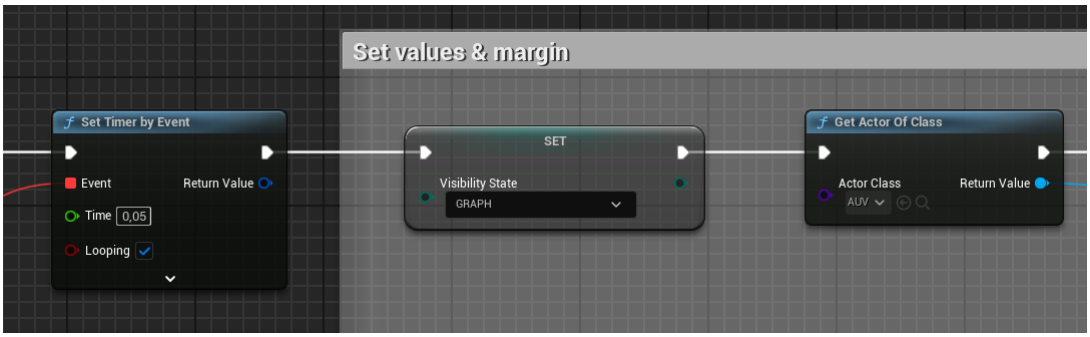

Figure 6.7: UMG Blueprint example

Communication with Unreal via UMG is very simple: communication can be carried out directly via blueprints. UMG contains its own graph editor, which provides the same script functionalities as blueprints. All UMG elements can be declared as variables so that they are accessible in these scripts. This feature is very helpful, for example, for dynamic display of numerical values.

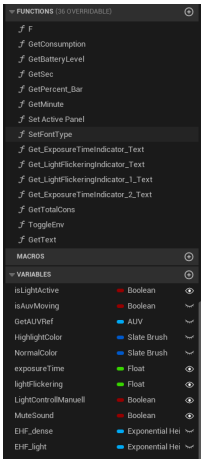

Figure 6.8: UMG Variables example

An external reference is necessary for communication with objects used in the scene. The UMG environment is closed in itself; references must be set separately. For this project we did this largely via public variables. This enables communication in a bidirectional direction: on the one hand, it is possible to set a reference to it outside the UMG blueprint, and on the other hand, the UMG blueprint can also be given a reference to the external object directly.

Version 1

As described above, the actual purpose for this first version was to display IMU measurements. In this version there are three different views:

1. Numerical
2. Graph
3. Hidden

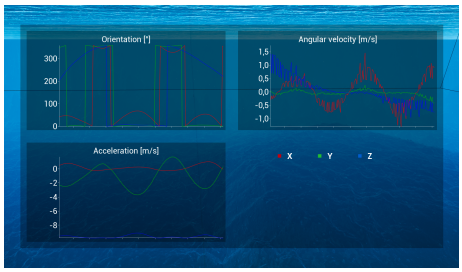

Figure 6.9: User Interface Version 1: Graph view

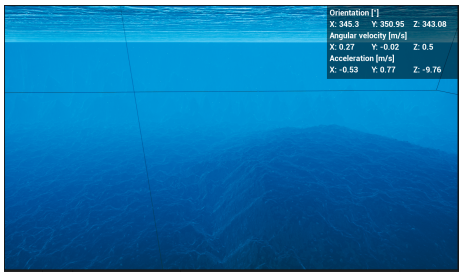

Figure 6.10: User Interface Version 1: Numeric view

Version 2

As the project progressed, however, it became clear that other elements in the UI would also be helpful. More elements have been added in this next version. These include, among other things, keybinds and a battery status. The general layout initially remained the same, the elements are still displayed one after the other as an overlay. In this version, additional pages have been added for the corresponding elements.

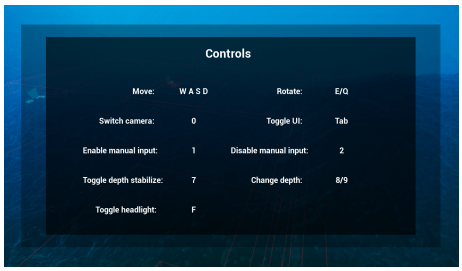

Figure 6.11: User Interface Version 2: Keybinds

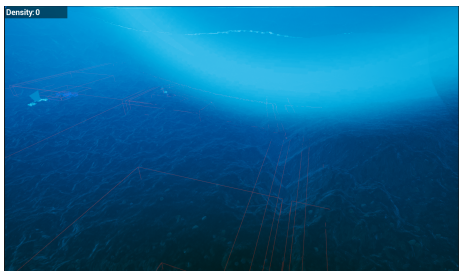

Figure 6.12: User Interface Version 2: Density

Final Version

In the 2nd milestone, the wish was expressed that the UI should be available as a separate window. This is problematic because Unreal is designed as an application that only occupies one window. To solve this problem, the inheritance hierarchy of UMG is relevant. UMG is a child of Slate, which can be viewed as the predecessor of UMG. Slate is also a library for creating user interfaces, but unlike UMG, it is not directly tied to Unreal. In addition, Slate only offers the option of creating UI elements exclusively in code. Slate also provides window management functionality, and since each UMG element inherits from Slate, these can be used to move the user interface to its own window by wrapping the existing UMG content inside Slate objects. The key elements of this process are outlined in the following images:

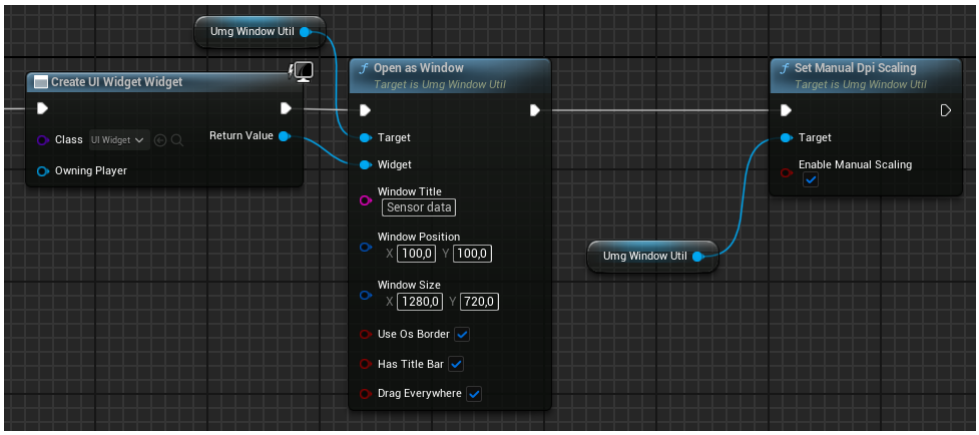

Figure 6.13: UMG Blueprint: Open UI in a separate window

```
void AUmgWindowUtil::OpenAsWindow(UUserWidget* Widget, FString
    WindowTitle, FVector2D WindowPosition, FVector2D WindowSize, bool
    bUseOsBorder, bool bHasTitleBar, bool bDragEverywhere)
{
  UmgWidget = Widget;
  WindowOriginalSize = WindowSize;

  ThisWindow = SNew(SWindow)
    .Title(FText::FromString(WindowTitle))
    .ScreenPosition(WindowPosition)
    .ClientSize(WindowSize)
    .UseOSWindowBorder(bUseOsBorder)
    .bDragAnywhere(bDragEverywhere)
    .CreateTitleBar(bHasTitleBar);

  FSlateApplication::Get().AddWindow(ThisWindow.ToSharedRef());
  ThisWindow.Get()->BringToFront(true);
  WindowClosedDelegate.BindUObject(this, &AUmgWindowUtil::OnWindowClose
      );
  ThisWindow->SetOnWindowClosed(WindowClosedDelegate);
  TSharedRef<SWidget> SlateWidget = Widget->TakeWidget();
  ThisWindow->SetContent(SlateWidget);
}
```

Since the application now runs in its own window and no longer as an overlay, some profound changes are necessary:

- Input options need to be changed
- Interface must make all contained elements clearly recognizable at first glance
- The interface must be adapted to changing screen sizes
- When exiting the application, the window must also be closed

The input options must be adapted to the application now running in a different window. Changing the window using a button no longer makes sense in this view, as the focus has to be on the Unreal Engine and not on the user interface. Since all displayed content is now displayed in a menu bar, it also makes sense to be able to switch between them freely instead of rotating them in a fixed order. A new window is also freely movable for the user. For example, it is possible that he either moves the window to another monitor

or resizes the window to be able to view both elements on one monitor. Therefore, the window has been modified to adapt to changing screen sizes.

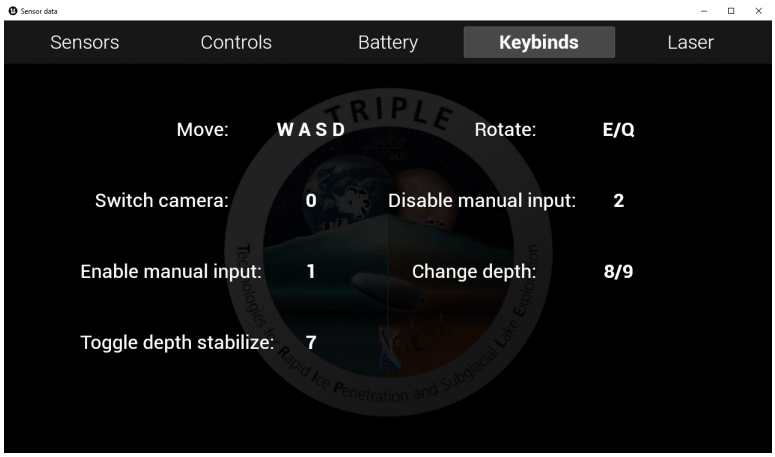

Figure 6.14: Final user interface layout

This window contains all the content that was previously there. Every element that could previously be switched through is now available as an independent window. However, since the window for the data from the USBL and depth sensor did not contain enough content, these were combined together with the IMU measurements in a tab. By clicking on the respective element in the menu bar, the individual pages can be viewed.

However, this change also resulted in the problem that the system contained various keybinds that triggered features. For example, it was possible to control the AUV's light via a keybind. This new window does not guarantee that the focus is permanently on Unreal Engine, so there is a possibility that the user experience will be limited as the user often has to switch between both windows. For this purpose, many keybinds that were not needed for direct navigation or scene viewing were moved to the user interface. The "Controls" section was added where various features can be controlled.

6.5.2 IMU UI

To visualize the IMU sensor, we had to decide between several possible implementations. On the one hand, it would have been possible to visualize this data directly in the scene. A possible implementation could have been to display a static coordinate system at the origin of the AUV. This would have had to be supplemented with additional vectors, for example for acceleration and angular velocity. After a few tests, however, it quickly became clear that this type of visualization could not be implemented in the scene in such a way that sensor data could be meaningfully read. The decision was therefore made not to display the visualization directly in the scene, but in a separate user interface.

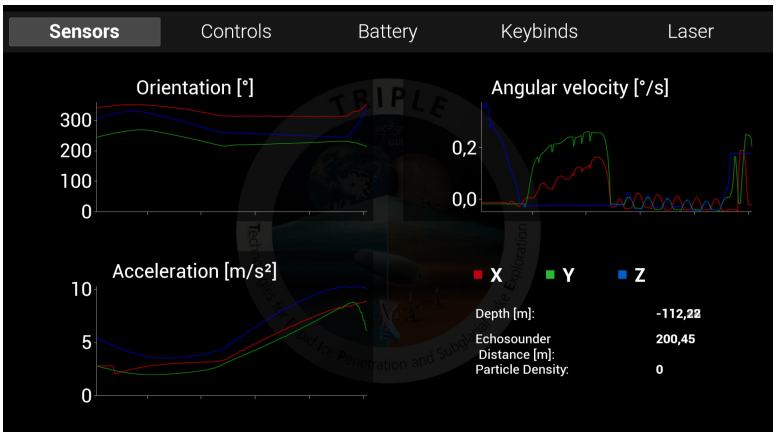

Figure 6.15: IMU UI

For this purpose, the sensor data is visualized using graphs. This representation offers the advantage that, on the one hand, it is always clear for each value what was measured at the respective time, but on the other hand, the previous values from a few seconds ago can also be viewed. The *Kantan Charts* plugin was used to create the graphs ([3]). There are a few data types included in this plugin that represent different graph types. Each type already has its own UMG widget, which can be inserted into the user interface. A few more steps are necessary to initialize a data series for a graph. When you start the application, all graphs are first created, given the corresponding ID and added to the UMG widget. Data points can later be added to the respective data series using this ID.

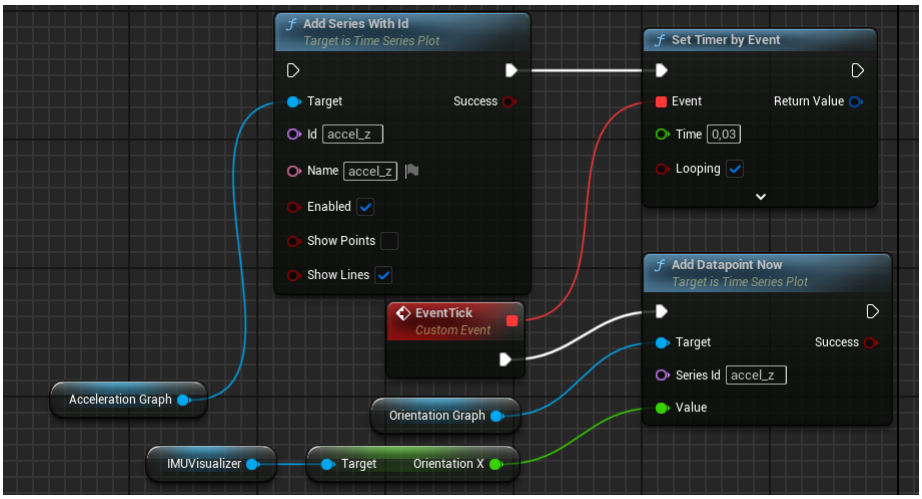

Figure 6.16: Creation and update of a data series with Kantan Charts

In earlier versions of the application there was initially another representation of the sensor data, which was presented purely numerically (see 6.5.1). This representation had the advantage that it was more compact than the graph representation and could be used parallel to the actual scene without restrictions. However, since the values change with every frame update, it is hardly possible to read exact values here. This was also a reason for discarding this representation as the project progressed.

6.5.2.1 Problems

One problem was large values that were significantly above or below the range of values otherwise shown. There are some situations where such large values can occur. These include, for example, moving the Unreal Engine application, changing the editor layout at runtime or while starting the application. In these cases it can be observed that the simulation is paused and only continues when the respective process is completed. Since the respective positions of the scene objects of the current and last frame as well as the delta time are used to calculate the measured values, this should actually not be a problem. It was not possible to conclusively clarify what exactly the cause of this error was. However, it is reasonable to assume that this is due to the interaction with the ROS system. For example, while the window is being moved, the Unreal application is paused. However, since the ROS system works independently, when the simulation is continued, a position is output that is calculated independently of the paused simulation and therefore no longer matches the delta time that is used for the calculation.

This is problematic because the y-axis of the graph takes on very high values. Unfortunately, the plugin used does not allow for the scale to be automatically adjusted again. This means that normal values can no longer be recognized. After some tests it became clear that in normal operation the fluctuations in the values are in a very small range, so that a new value never differs from its predecessor by more than about three times. This makes it possible to filter all values that exceed this threshold.

For this purpose, some changes have been made to the visualization. The sensor data is visualized in the *IMUVisualizer* class. In order to filter out these values, it is first necessary to save all values that are currently displayed in the graph. Unfortunately, there is no way to get these values in the Kantan Charts plugin, so a custom implementation has to be made. However, since not all values of each frame are interesting for each graph, but only the largest and smallest, this somewhat limits the values that need to be saved.

```
/* Store the highest and lowest values that are added to the graphs in
   this tick. Also store the average of the highest and lowest values
   in each graph to later filter out extreme spike values */
float valueToAdd = max(max(angVel_x, angVel_y), angVel_z);
angVelMax.push_back(make_tuple(timer, valueToAdd));
float weight = angVelMax.size();
angVelMaxAvg = angVelMaxAvg == 0 ? valueToAdd : (angVelMaxAvg * ((
   weight-1)/weight)) + (valueToAdd * (1/weight));
```

By simultaneously recording the average when inserting new values, you avoid having to constantly re-iterate the list of available values. The calculated average can then be used to check whether new values are above or below the threshold. The average of all values is used here instead of the maximum or minimum. When using maximum and minimum, multiple values that are just below the threshold may be added within a short period of time. This causes this threshold for new values to be added to constantly increase or decrease, which can ultimately cause the filter to stop functioning properly until these values are out of view. This is prevented by using the average.

In addition, a short delay must be added at the beginning of the application. This is because, like moving a window, very large values are output. Since there is no further data at the beginning that would make it possible to filter this value, this was solved by inserting a short delay at the beginning.

6.5.3 Laser UI

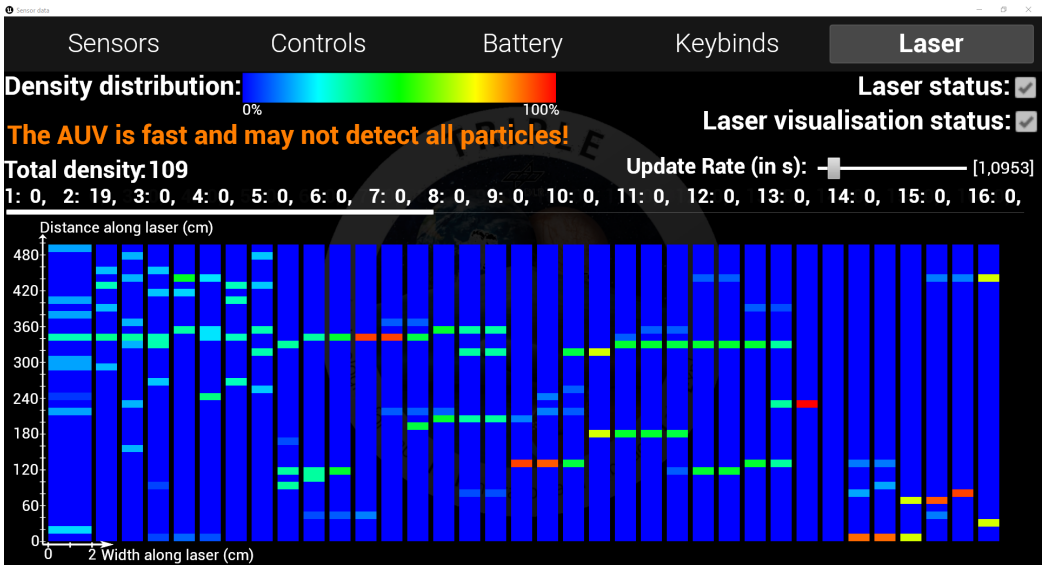

Figure 6.17: Laser UI

The laser UI can be found under the following path: Content/TripleSim/UserInterface/UI-_Laser. Exact implementation details can be looked up there, as everything is commented. Figure 6.17 shows the UI of the laser. The laser UI can be accessed via the "Laser" tab. At the top right, the laser and the laser visualization can be turned on and off with a mouse click. The total density is also displayed in the UI. This indicates how high the density was in the entire laser at the last measured time. Below the total density, a scrollable list is displayed which shows how high the density was in each individual bin at the last measured time. The structure is as follows: *Bin number: Density in the bin*. 1: 0 would therefore mean that the first bin has a density of 0. The values for the total density and the bins are taken from the arrays of the blueprint BP_TotalParticle
In addition, the laser UI displays a message in an orange color. This message is only displayed if the AUV is traveling too fast and it could happen that particles cannot be detected by the laser. The topic of the too fast particles or AUV has already been discussed in chapter 4.4.4 which deals with particle detection for the particles of the black smoker.

6.5.3.1 Heatmap

The laser UI has a heatmap that is generated based on the density of the bins. This is a 5-color-heatmap [42], which ranges from blue to red. Blue means that there are no particles in the bin and red means that all particles are in this bin. Since green is exactly in the middle of the color gradient from blue to red, this would mean that 50% of the particles are in this bin. Let's assume that we have a total density of 12. 6 particles are in bin 1, 3 particles in bin 4 and 3 particles in bin 10. Bin 1 would then be green because 50% of the particles are in this bin. Bins 4 and 10 would have the color turquoise because they each contain 25% of the particles. All other bins would have the color blue because there are no particles in these bins. At the top left of the UI is the density distribution. This is a kind of legend that translates the colors into percentages so that you know which color stands for which number. The exact number is always different, depending on how many particles are currently in the laser. If the last measured value in the laser is 25, then the

100% is represented by the number 25. The first heatmap also has a coordinate system that shows how big the laser and the bins along the laser are and how wide the laser and the bins are.
Several heatmaps are displayed in the UI. This is a history of the heatmaps over time. The first heatmap is slightly thicker as it shows the last measured densities. The density measurement for the heatmap takes place at a certain interval and is called the update rate in the UI. This specifies how often the density is measured per second. The update rate is a slider and can be changed by the user. By default, the update rate is 0.03. As the system runs at 30 FPS, 0.03 means that the densities are measured and the heatmap is updated every frame. The update rate can be increased up to 10 seconds. As soon as the density is measured again, the heatmaps shift one position to the right. This creates the heatmap over time. Let's assume the default value of 0.03. The new density is then measured. The heatmap at position 1 receives the newly measured values, heatmap 2 receives the values from heatmap 1 from the last frame, heatmap 3 receives the values from heatmap 2 from the last frame, and so on.

6.5.3.2 Implementation

Creating a heatmap directly in the Unreal Engine UI is normally not possible. However, with a lot of testing, I have found a workaround to make this possible.
Every heatmap is actually a ListView. This ListView consists of border widgets, which are normally used to give UI elements a border. I use these borders to visualize the bins in the heatmap, because it is possible to assign different colors to the borders during runtime. The size of the borders is calculated based on the number of bins and the size of the laser. The color is calculated during runtime based on the densities of the bins. This data comes from the BP_TotalParticle and has already been discussed in chapter 3.4.5.2. The color for a bin is calculated using several lerps between the 5 main colors. It is lerped between the RGB colors (0,0,255) & (0,255,255), (0,255,255) & (0,255,0), (0,255,0) & (255,255,0) and (255,255,0) & (255,0,0). The density in the bin is used to decide between which two colors to lerp. In addition, the alpha value for the lerp is calculated so that the respective color proportion between the two colors can be determined. This ensures that the correct color is always generated for the respective bin in the heatmap.
The coordinate system for the first heatmap is an image that I created myself. However, the numbers on the coordinate system are text widgets. These numbers are automatically set correctly based on the size and number of bins. For example, you can change the size of the laser and the numbers in the coordinate system will adjust correctly.

6.5.3.3 Laser data ROS

Since it is not possible to create a full 3D heatmap with rotations in the Unreal Engine UI, I send all the data required to create such a heatmap to ROS. The data is sent in the BP_ExportLaserData. This actor must be placed in the level so that the data can be sent to ROS. The values from ROS can then be read out and used to generate a real 3D heatmap, e.g. in Python.
The array TotalParticleBins from BP_TotalParticle is sent, which contains the densities of all bins. The corresponding topic is called LaserBinsDensity. In addition, the origin of the laser is sent as a vector and has the topic name LaserOriginPosition. Finally, the rotation of the laser is sent as a vector. The topic name is LaserOriginRotation. The time is sent as standard in ROS anyway. All other values required to generate the 3D heatmap are fixed values or can be derived from the ROS data sent. This includes the dimensions of the laser. These are fixed and therefore do not need to be constantly sent to ROS. The

laser currently has a length of 500cm, a width of 2cm and a height of 2cm. The number of bins is also fixed and is currently 40. The size of a bin can be calculated using these two values. A bin has the same width and height as the laser, i.e. 2cm each. The length of a bin can be calculated by the *length of the laser ÷ number of bins*, i.e. 12.5cm (500÷40). The rotation of the bins can be derived from the origin of the laser and the rotation of the laser. The exact position of the bins can also be calculated. Since we know that the first bin starts at the origin of the laser and has a length of 12.5cm, the second bin then comes directly after the first bin, i.e. starts at 12.5cm and goes up to 25cm, and so on.

6.5.3.4 Future works

In the future, an attempt could be made to generate the 3D heatmap with rotations in Python and then send an image of the heatmap back to Unreal Engine during runtime. This would allow the image to be displayed in the Laser UI. As the idea of the heatmap only came up at the end of the project, I was unable to implement this due to a lack of time.

6.6 Camera views

There are various camera perspectives in our system. These are explained in the following subchapters.

6.6.1 Basic views

Early in the project the need for multiple camera views became apparent, without it either the environment or the vehicle could not be observed in a good way.

As such the first version included a freely controllable spectator camera with which it is possible to move through the complete level. And a fixed camera on the vehicle with which a more realistic view of what the AUV sees is presented. It was possible to switch between them using a keyboard input.

6.6.2 Orbit camera

As the project progressed, it became apparent that more camera perspectives were needed. Due to the limited view within the scene, it is difficult to maneuver without a fixed point on which the camera focuses and not lose the spatial impression. Although the camera on the AUV delivers realistic images from the vehicle's perspective, the fixed positioning limits the field of view and does not always necessarily capture all interesting objects. For this reason a third view was added.

The orbit camera is focused on the AUV and allows circular movements around it. The camera can be activated via a keyboard assignment, which can also be used to exit it again. You can navigate around the vehicle by moving the mouse in the x or y direction. There are also other parameters that define the behavior of the camera, such as camera speed. However, these are not directly accessible during runtime.

In the first version it was possible to move around the AUV without any further restrictions. This presented some problems. In this way it was possible to lose spatial orientation, especially if the AUV itself was not exactly in space. In addition, an error could occur on the y-axis when exceeding 360°, which caused the camera position to change suddenly, resulting in a very unpleasant flicker.

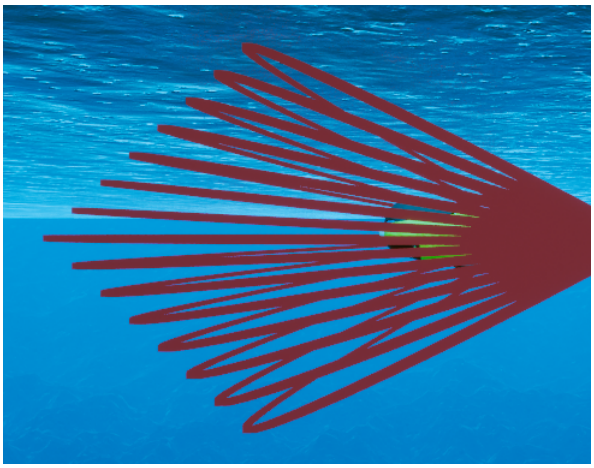

Figure 6.18: Orbit camera rotation bounds around y-axis

In the second version, the camera was adapted accordingly. The range of motion on the x axis is still unrestricted, but on the y axis it can only be maneuvered within a range of -35° to 35°. This makes it much easier to track the actual spatial position of the AUV. This also solves the problem of flickering, as the camera never reaches the required degrees. Finally, a zoom function was added. This makes it possible to observe the AUV from a distance of between one and ten meters. Zoom can be adjusted by operating the mouse wheel.

6.7 Simulation

This chapter discusses the ocean current simulation and the simulation of the battery system in more detail.

6.7.1 Ocean current simulation

The ocean currents are simulated to add external disturbances to the AUV to make the simulation more realistic. Currently the only source of ocean current velocities is a constant velocity for the complete level combined with local changes around black smokers.

These changes are defined by creating OceanCurrentModifier actors which influence the local velocities around them by defining a cone from multiple cone segments to model the shape and a velocity that has a normal distribution based on the distance from the center to the edges.

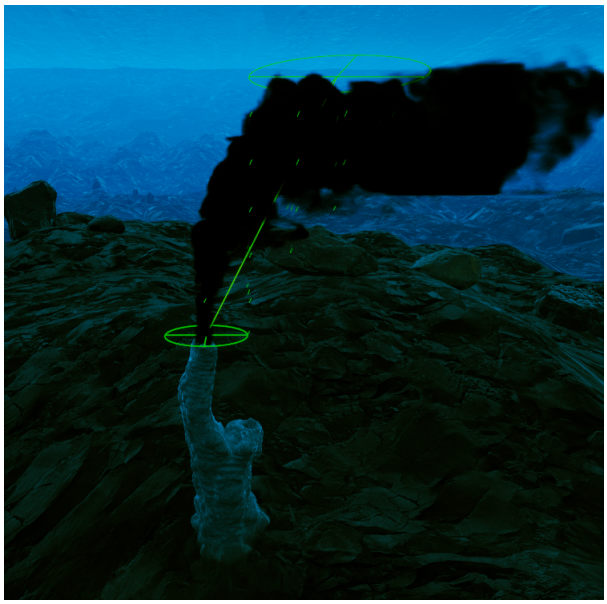

Figure 6.19: OceanCurrentModifier that uses a single cone segment

For debugging purposes there are multiple options to directly view the velocities as arrows in the level. Additionally there is the OceanCurrentViewer that just shows the velocities in a grid of variable size and step size between sample points. These arrows have the length proportional to the length of the velocity. Additionally the arrows are blue if the velocity has the default value and green otherwise.

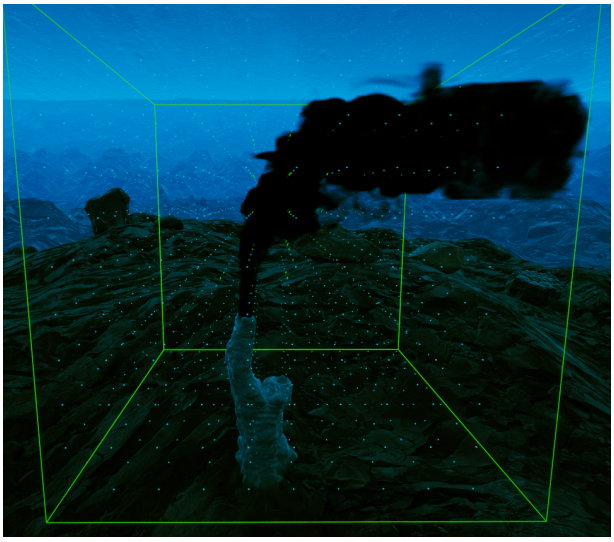

Figure 6.20: OceanCurrentViewer showing velocities with arrows

The computation of the ocean current map happens on a per level basis using the parameters set in the OceanCurrent actor and all OceanCurrentModifier actors existing in the level. After which it is saved to disk to just load the next time the level is started. The values are calculated by averaging all OceanCurrentModifier velocities, that are near enough the evaluation position to have an influence, with the default velocity.

With the approach of pre-computing the values a one-time generation time in the order of seconds or minutes does not matter, as all subsequent starts can load the map in a few

milliseconds. Additionally more complex algorithms of computational fluid dynamics could be used to get more realistic results, for this project they were not realistically implementable in the given time frame.

After finishing the pre-computation the underlying data structure is saved directly from memory as a binary file. This makes loading fast, as it does not have any overhead in contrast to HDF5 or other formats.

6.7.1.1 Implementations

The first implementation used a self-implemented generic 3D grid to store the values. But it was clear that it is not possible to store the values in a fine-enough resolution to be useful as that would consume too much ressources. The level has a size of 5000x5000x750 meters with the black smokers having an opening about the size of 0.2 meters a resolution of 0.1 meters would be appropriate. This would result in $50000 \cdot 50000 \cdot 7500 = 1.875 \cdot 10^{13}$ values. When using 3D floats to save the values it results in $12Bytes \cdot 1.875 \cdot 10^{13} \approx 204.636TiB$ which is definitely to much to have in system memory or save to disk for normal computers.

Consequently a data structure that does not need so much system memory is needed, the octree is a good solution as it partitions space into a tree of octants and areas that store the same value can be combined into one area.

The nodes of the tree store whether it is a leaf or inner node as a boolean, indices to the child nodes as integers and a value over which the octree is generated. For which the storage size can only determined at time of usage as the implementation is generic to be usable with other kinds of maps/data. This created the problem of not knowing the data type while constructing the octree, which lead to the need of having functions as parameters to control the construction process. These functions are a sampling function to generate the value given a 3D world position and a splitting function to determine whether two values are different enough from each other and the octants should be split into smaller octants. The values are then inserted into the octree which makes the construction process independent of the amount of points that the octree should store.

But a further optimization was to convert the octree to a sparse octree. Which makes use of the value variable in inner nodes to store values for child nodes that would have the same value. This changes how the octree is expanded, now only one child node needs to be created for a new value instead of the eight when using a standard octree. This drastically reduces the amount of nodes in the higher level inner nodes of the octree where a complete black smoker fits into one octant needing only one instead of eight nodes.

Additionally as to smooth out transitions between octants the AUV uses trilinear interpolation, where the eight points are at the corners of a oriented bounding box of the AUV.

6.7.1.2 Results

The following images were made with one black smoker in the level and a cell size of 25cm this results in a build time of around 100 milliseconds with a node count of 4000 and a file size of 250 KiB. A higher cell size was chosen for the images as the boxes are rendered using debug boxes which do not have a good performance.

For a level with three black smokers and a cell size of 10cm the map builds in 10.71 seconds and has around 150k nodes and has a file size of 9.08 MiB.

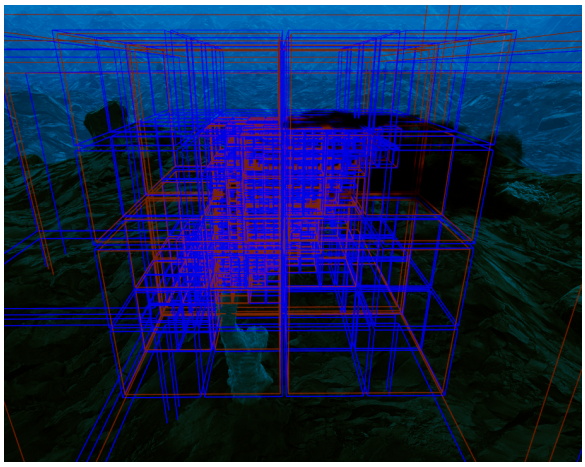

Figure 6.21: Octree close to a black smoker, box colors: blue=leaf, red=inner node

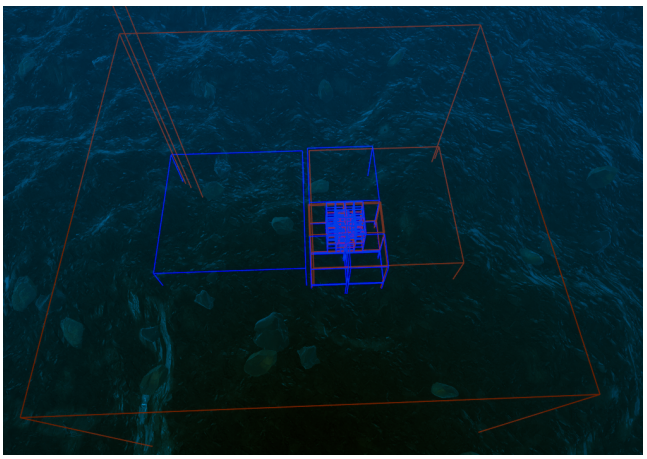

Figure 6.22: Octree with a little distance from black smoker, box colors: blue=leaf, red=inner node

6.7.1.3 Future work

In the future an improvement to the current saving/loading and octree generation system could be to move those implementations into its own library, such that external application can generate octree files. Furthermore currently the parameters are stored in the file names, it would be good to directly store those information in the file or create separate meta files.

Another big improvement could be made to the way ocean current velocities are calculated. Maybe some open-source computational fluid dynamics solver or self-written implementations could be used.

A aspect that is currently completely neglected, is the how ocean currents change with time and through tidal influences. To save such data efficiently the data structure temporal index tree could be used. It builds a tree where nodes store a time period and the data that has small temporal variance during this period, which in our case would be a sparse octree.

6.7.2 Battery system simulation

Our simulation also simulates the battery system of the AUV. This consists of the part in Unreal and the ROS side. In the following this will be explained in more detail.

6.7.2.1 Battery system

One of the features of our project is battery simulation, which was included in our second milestone. The purpose of this feature is to provide an estimate of the average energy consumption for our operations. I obtained the values from our supervisors at MARUM and prepared the dynamic battery simulation model accordingly.

Parameters	Quantity	Wh
Propulsion Power	1	23(average)
Communication Power	1	20
Depth Sensor	2	2
IMU Sensor	1	1
Echolot	1	3
USBL	1	4
Light	1	100
Laser	1	2
RGB Camera	1	5
Total energy Consumption (W)		=160

Table 6.1: Spreadsheet

The power-hungry components of the nano AUV are the propulsion system, sensors, communication devices. Thus, we have to consider the minimum amount it needs to operate and came back to the base station. So we are considering 20% safety Margin.

Total Energy Consumption = 160 Wh

Desired Operation Time = 1 hr

Battery Capacity = 200 Wh

6.7.2.2 Calculation of Propulsion System power

Propulsion System:

Autonomous Underwater Vehicle (AUV) typically utilizes a small-scale propulsion system due to its miniature size. These propulsion systems are designed to be efficient, lightweight, and capable of maneuvering in confined spaces.

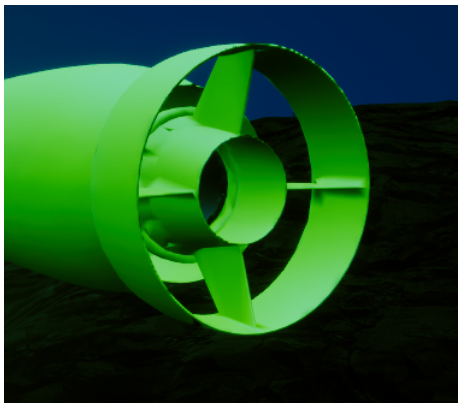

Figure 6.23: Thruster in the AUV model

Nano AUVs often use miniature thrusters, typically electrically powered, to generate thrust for propulsion. These thrusters can be designed to operate efficiently in underwater conditions while being compact enough to fit within the small form factor of the vehicle. Propeller thrusters are common propulsion systems for nano AUVs, making autonomous underwater robotics an affordable possibility for researchers, scientists, and businesses.

According to the technical details available on the Blue robotics website[4], we considered for 10 V

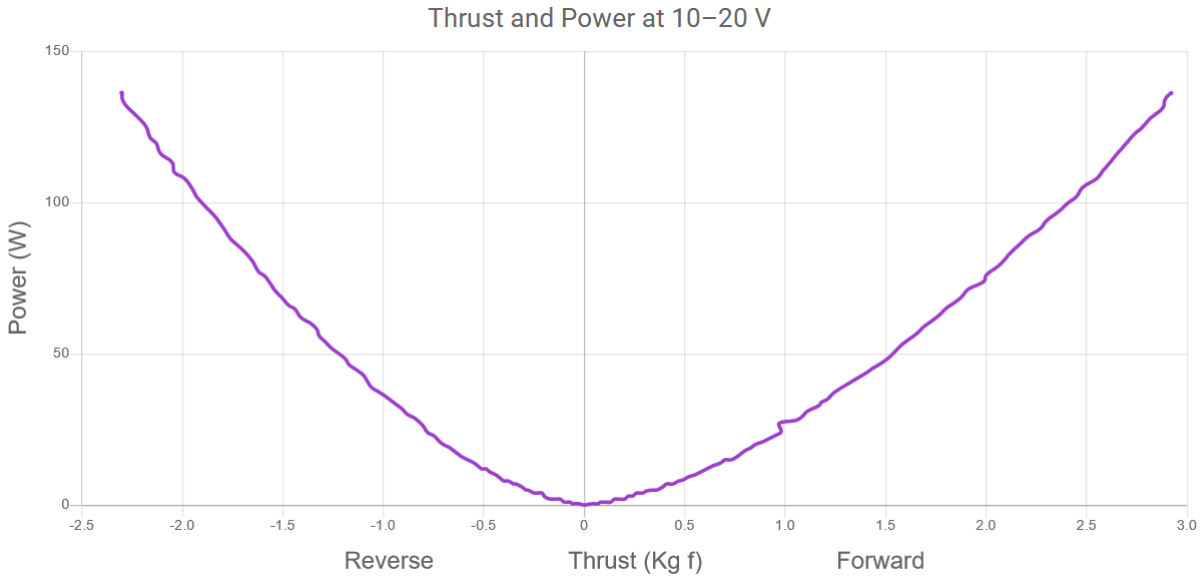

Figure 6.24: Power Vs Thrust curve

This figure shows the power against the thrust and the unit of thrust is in kilogram force, as we don not know about gravitational force on Enceladus. We kept the value as it is.

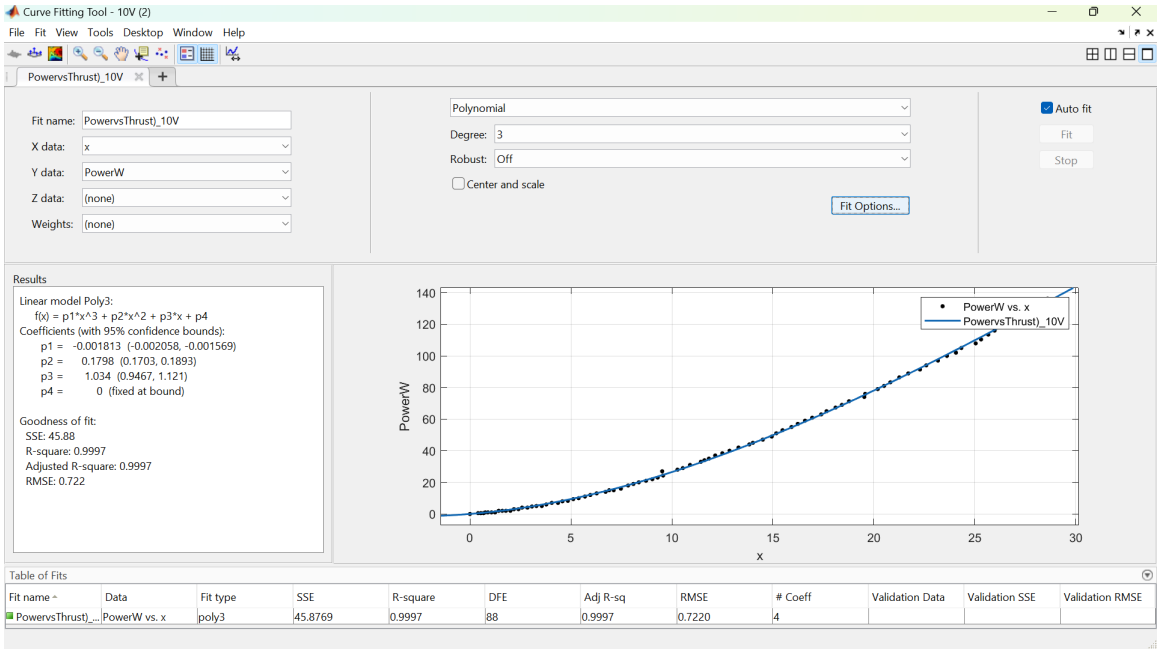

Figure 6.25: Formula for power calculation

$$f(x) = p_1x^3 + p_2x^2 + p_3x + p_4 \tag{6.1}$$

I have used Matlab curve fitting tool and extracted the equation to calculate propulsion power for the battery system.

To calculate the propulsion power with the help of achieved formula, I have calculated the force and toque that is being received from the ROS system.

Force = $x_1 + x_2 + x_3$

Torque = $y_1 + y_2 + y_3$

Propulsion power = $\sqrt{x_1^2 + x_2^2 + x_3^2 + y_1^2 + y_2^2 + y_3^2}$

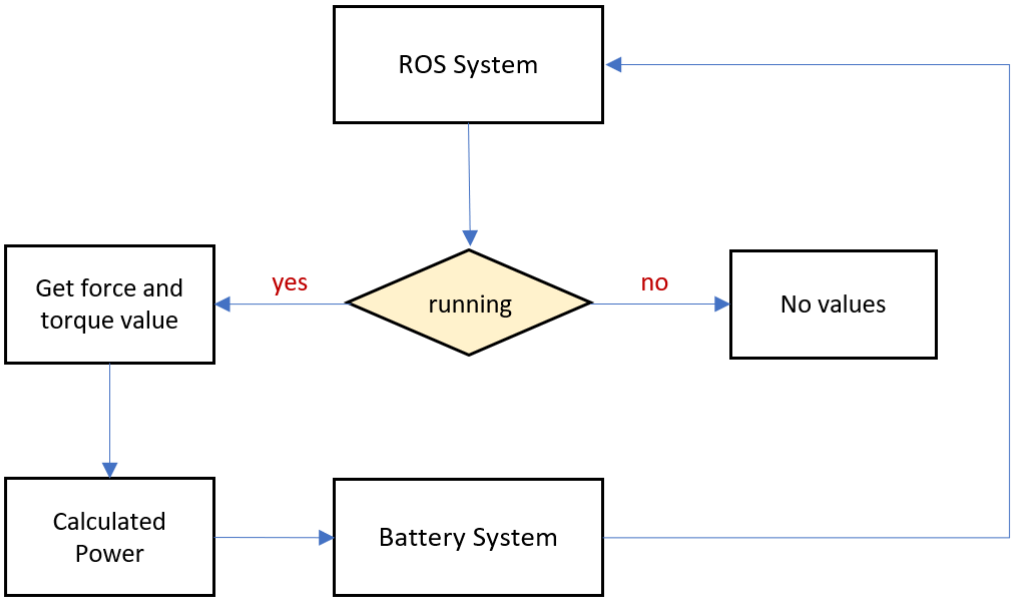

Figure 6.26: Propulsion Power Calculation

The ROS System provides the values. Without ROS system the battery system will not be executed, as it is proving the propulsion power.

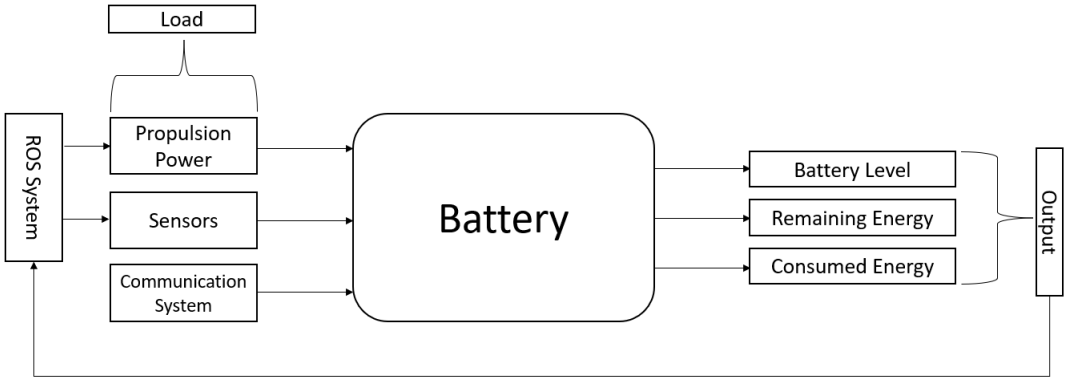

Figure 6.27: Battery Sketch

This fig shows the load on the battery and the final output showing the remaining level and energy.

6.7.2.3 Implementation

Display Battery Status

The primary goal is to monitor battery status to assess its performance during operations, both with and without additional loads such as lights. It doesn't need to be always on during the operation, and energy can be saved. For communication, the communication board must be always active. For calculation, we are considering the sensors are active by default as they consume very less power and we are not considering duty cycle. We

considered a safety margin so that when the battery is less than 20%, the AUV can return to the base station. I have implemented logic in Blueprints to monitor the battery level continuously. When the battery level falls below 20%, it triggers an alert displaying a warning message.

In the main level, with a key binding, it is not required to display the battery status all the time. According to discussions in our weekly meetings, I assigned a key binding to display the battery status when the main level is running. So, when the key is pressed, it will show the battery status. For the second milestone, I prepared a power consumption UI widget to display battery features in the main scene. Later on, after modifications and based on discussions, we agreed to display everything together in one UI. I have continuously iterated on the UI design based on feedback from discussions and meetings.

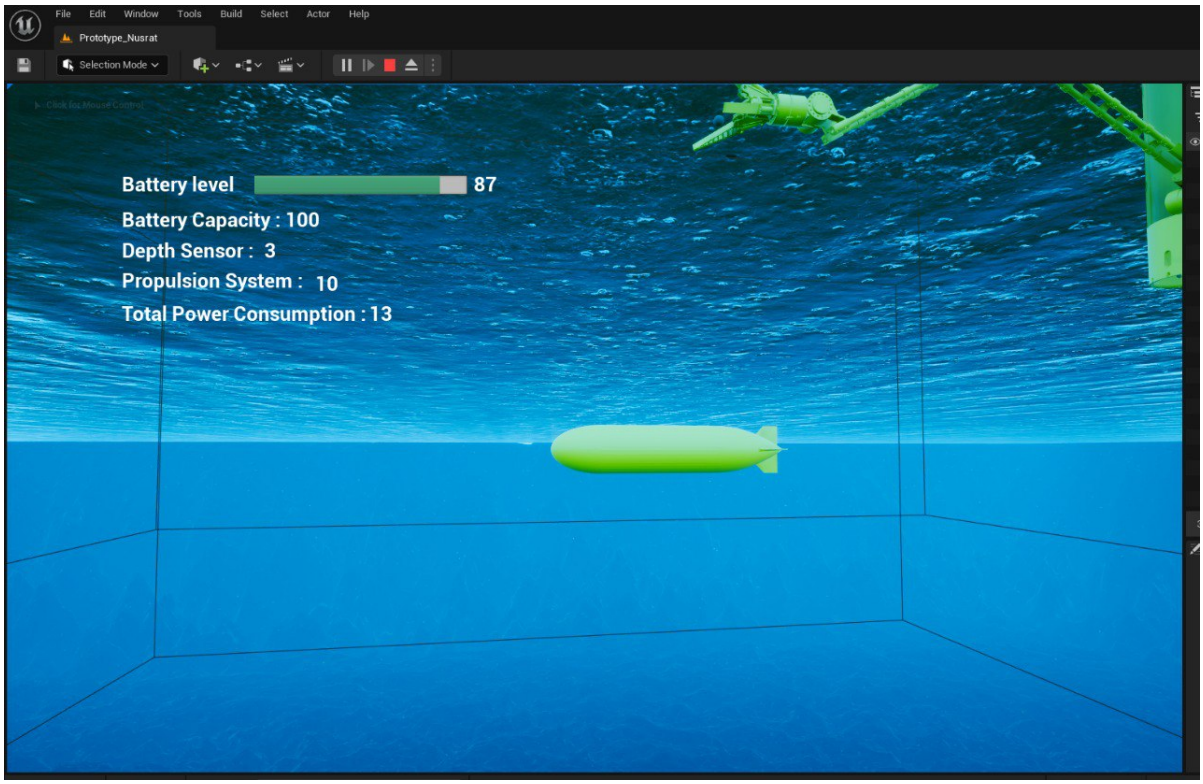

Figure 6.28: initial version

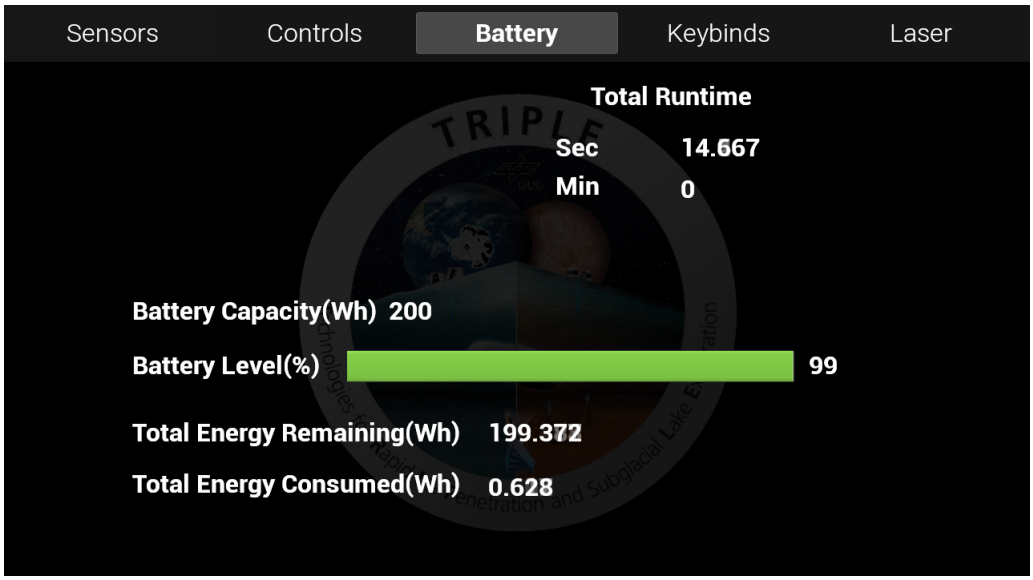

Figure 6.29: Final Version

6.7.2.4 Challenges

UI widget should display real-time data on energy usage and we are displaying everything together in one UI, UI elements render quickly and efficiently, even in complex scenes but Complex UI layouts with numerous widgets can impact performances. So, I had to think alternately for implementing custom event handling and data binding mechanisms for keeping UI content synchronized with the game world.

6.7.2.5 ROS2 integration

To make it possible to enable and disable the engine of the AUV in case of complete battery depletion or have the AUV cancel the currently planned trajectory and return to the melting probe additional components were added to the AUV actor. These are used to send the control message to the ROS2 system. To make usage of this functionality easier a wrapper around the components was created to combine all into one actor.

Additionally to compute the power usage the motor output of the AUV was needed. Therefore an extra actor was created to subscribe to the control forces output by the pilot node.

This functionality is completely implemented and working, but not used in the Unreal Engine battery system simulation.

6.8 Containerization & Execution

As already mentioned in chapter 5.3 there was the need to containerize our Unreal Engine project, as ROS2 communication between containers was the only working alternative to running the unreal project directly on the Windows host and ROS2 application in Ubuntu 22.04 using WSL.

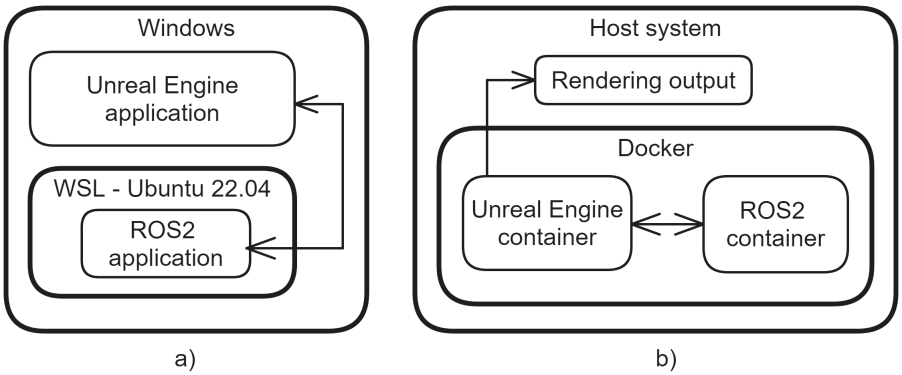

Figure 6.30: Different ways to execute the project stack. a) Running UE application directly on windows and ROS2 using Ubuntu 22.04 on WSL b) Running everything using Docker

Another aspect of why a containerized approach would be better, is that to run the Unreal project in Windows a lot of installation and setup is needed because of ROS2. So people that want to run the software in the future need to have all the right software installed otherwise it would not work.

There a containerized approach to deployment has the advantage that it would be possible to create custom Docker images and use the GitLab package registry to store them. Then running the complete application stack would only require a *docker compose up* to start all containers, all needed images could be build using GitLabs CI/CD pipelines and then directly stored in the package registry of the repository.

To complete this goal the first step would be to have the finished release build binaries for the Unreal Engine project. As the containerization of the ROS2 application is already finished.

Epic Games provides Docker images for building and running of Unreal Engine projects. The problem with them was that the base image they are based on is Ubuntu 18.04 which is not supported by the ROS2 version Humble that we are using.

This lead to the question of do we even need automated builds at this point in time? The answer to that is no, as the runtime images are also based on Ubuntu 18.04 running the project with this approach would also not work.

This lead shifting the goal to just manually building the project for Linux with Unreal Engine's cross-compilation. And trying to just run the application in a container.

To accomplish this, more in depth research was made into what does our Unreal Engine application need to run on Linux.

The requirements are the following:

- Full Linux distributions
- GPU acceleration
- ROS2 humble (Ubuntu 22.04)

Full Linux distribution come with a lot of libraries and application pre-installed that are needed for all kinds of purposes and also needed by Unreal Engine applications. Therefore more minimal distributions such as Alpine would not work out of the box and need a lot of additional work.

Additionally the container needs to support GPU acceleration for Unreal Engine to be able to use the GPU to do rendering. Currently our project uses Unreal Engine version 5.2, for which the render pipeline of Linux builds only supports Vulkan. Older versions such as 4.25 also supported OpenGL, but that is no longer the case.

Consequently a container is needed that uses Ubuntu 22.04, the NVIDIA container toolkit for GPU acceleration, supports Vulkan and provides a way to display the application windows from inside of the container on the host system.

According to these requirements already existing images were found. The images by Adam Rehn on Dockerhub provide pre-configured images for all kinds of environments and configuration how Unreal Engine projects can be run.

After running our project with the image *adamrehn/ue4-runtime:22.04-vulkan-x11* which uses X11 to display the windows on the host system. The error that no compatible Vulkan device was found appeared.

This was weird as the images were functioning for many other users. After trying multiple different images without success, I researched how to debug the Vulkan API. One such application is *vulkaninfo* it provides detailed information about all Vulkan devices and their capabilities. There only *llvmpipe* was listed as a device, which is basically just using the CPU as a Vulkan device.

Now the question was why does Vulkan not list the installed GPU although it is listed using NVIDIA's system management interface (nvidia-smi)?

As it turns out this is caused by the missing support for Vulkan drivers when running Docker on a Windows host system through WSL. Also it was not possible to switch to Linux as the operation system, because all our project members are using Windows 10 or 11 and not all may be experienced/able to work using Linux.

Therefore other Vulkan drivers that work in containers with a Windows host and WSL was needed. The only one that seemed promising was Mesa, which is a library of open-source implementations of multiple rendering pipeline specifications. It contains the driver called Dozen which is a translation layer from the Vulkan API to the DirectX12 API. The development to include the driver in Mesa begun in january of 2022. This allows running Vulkan applications through DirectX12 in WSL on a Windows host.

It is important to note that Dozen is currently experimental and is not included in the packages of Mesa in the package registries, because of this it was needed to build Mesa from source.

After installation *vulkaninfo* now shows the correct GPU hardware and the Vulkan test application *vulkancube* also starts correctly and works using the GPU.

Sadly when running the Unreal Engine project it crashes on startup. This appears to be caused by the Dozen driver being incomplete and not supporting all of Vulkan's functionalities.

No other way to run Vulkan applications in a container on a Windows host using WSL was found. As a consequence the containerization of the Unreal Engine project is not possible, which combined with the networking issues regarding ROS2 between container and host system means, that the application stack has to be run without the use of containers.

This leads to the way of running the Unreal Engine project on Windows which needs a complete ROS2 installation. Furthermore to avoid the erroneous movement of the AUV the ROS2 application has to be run on Ubuntu 22.04 using WSL.

7. Website

Another goal of this project was to provide a website that shows our progress and contains general project information. Since this website is only required to display information, we have decided to not integrate any further back-end implementation. Instead, the focus is on the clear representation of the existing data. During the development process there were essentially two iterations of the website, which are described below.

7.1 Version 1

The first iteration relies on using a framework to create static web pages. This framework is Hugo, which says it is one of the best-known and most popular frameworks for creating static websites ([15]). Hugo does not rely on JavaScript, but rather on Go as a scripting language. A major advantage of using this technology is that creating a simple website is possible with very little effort. In most cases it is sufficient to follow the few installation steps, clone a template and integrate it into a project. Most templates are structured in such a way that they create new content as sub pages and simply add links to them on the main page, like in a blog. New content can then be added in just a few steps. Unlike other frameworks, Markdown files are used directly; HTML code is not usually written directly, even though this is possible. Since Markdown files require relatively little effort and experience, it is very quick to add new content this way. Most of the general site layout and formatting is already handled by the chosen template. Since our website seemed to fit exactly into this framework, we initially agreed to use it.

Another alternative that has been considered is GrapesJS. Unlike Hugo, GrapesJS, as the name suggests, uses JavaScript as scripting language. GrapesJS is also less of a standalone framework like Hugo, but rather intended for integration into other content management systems, even though it can be used alone. A big advantage of GrapesJS is the presence of a drag & drop editor for creating the interface. In this way, in many cases HTML code is abstracted and worked directly with visual feedback, even if direct coding is possible at any time. Like Hugo, predefined templates are also available here, but the selection is significantly smaller.

We ultimately decided against using GrapesJS because the expected effort was considered to be higher than Hugo. Hugo already contains all the layout elements that our site should contain, whereas in GrapesJS these would probably have required additional work.

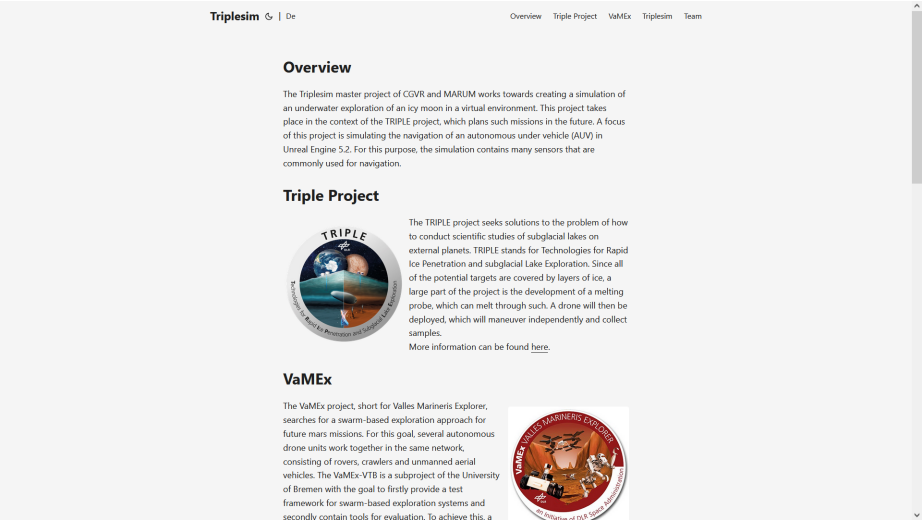

Figure 7.1: Website Version 1 (1)

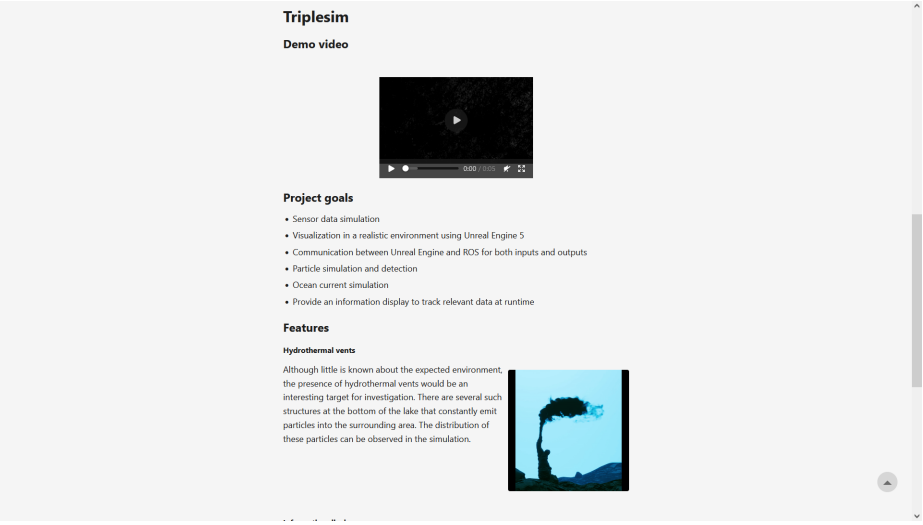

Figure 7.2: Website Version 1 (2)

The layout shown in the figure is largely determined by the used template which is called "PaperMod". A menu bar is already provided, there is also a dark mode and internationalization. Likewise, any content will be centered and only fill the middle 50% of the screen. Furthermore, many features are already included that increase user comfort. These include, among other things, smooth scrolling, scroll to top and pleasantly readable formatting.

The content that should be displayed on the website can be divided into the following subcategories:

1. Overview/Introduction
2. Project Information of related project
3. Project information about our project including demo video
4. Project features

5. Team

In the introductory part, the project will first be roughly outlined, and the project framework will be explained. Since the TRIPLE and VaMEx projects are relevant to the project context, they will be described directly afterwards. The project goals are briefly outlined below, and the demo video is shown, followed by some selected project features. We limited ourselves to a selection of four features, all of which contain both a text description and an image or GIF. The team section at the end is intended to inform the reader about which people participated in the project.

Even though this first version meets all the criteria that we initially declared as the goal for the website, some errors became apparent during creation. Markdown files are too inflexible for the purposes intended here. Despite the possibility to quickly create a page that contains mostly text, formatting images and text presents some problems. For example, it is not possible to display an image next to text in Markdown; HTML or CSS is required for this. In addition, it became apparent during implementation that structuring into multiple sub pages did not make sense given the amount of content. However, since the template provides for such structuring, this is also problematic, for example when creating the menu bar entries. Even though the page in this version is easy to read and functional, it still appears very simple and does not have a title page. The way the content is structured, this is a problem because when you first read the page, too little information about the Triplesim project is presented before the other projects are presented. For these reasons, it became obvious that this framework does not really fit the intended use, as many features are not or hardly used.

7.2 Version 2

No framework was used for the second iteration, this implementation relies solely on HTML, CSS and JavaScript. Among other things, this version should fix the following problems:

- Provide a title page for the project
- Better content formatting
- Less simple representation
- Include animations

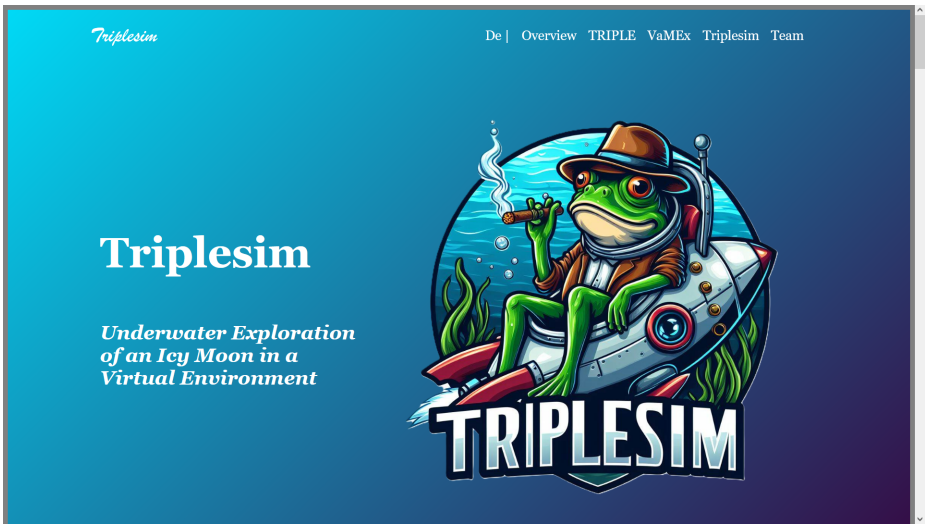

Figure 7.3: Website Version 2 (1)

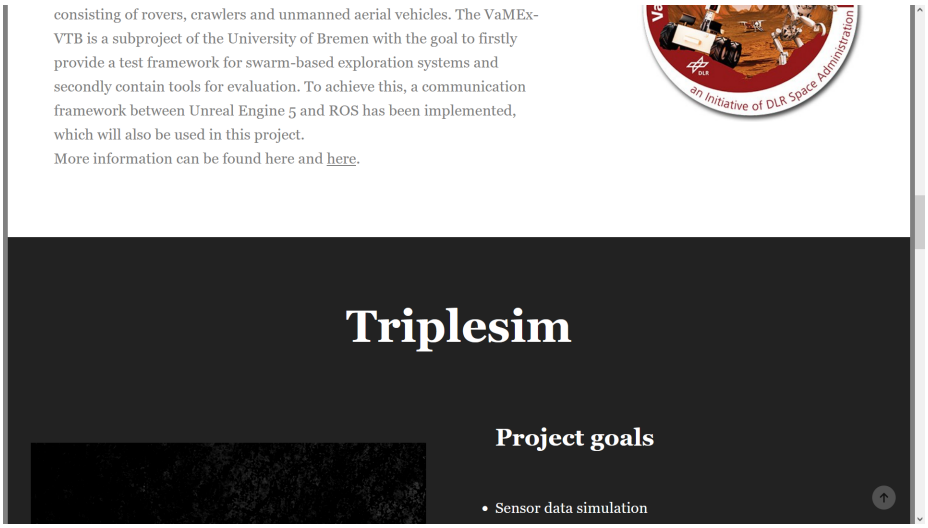

Figure 7.4: Website Version 2 (2)

This version includes a title page which, together with the overview, adequately introduces the project before the other projects are introduced. The site makes better use of the available space which also allows font and size customization. The website floats on a grey background, which is only visible in an eight-pixel wide gap at the side. This provides a more dynamic feel when scrolling through the page, since this background stays the same, while the content seems to be structured in individual boxes or cards. Animations were used to underline this effect. Most content now fades in when the reader scrolls to it. For these animations, an open-source library called "Animation on Scroll" ([37]) was used. This commonly used library contains many easily accessible animations that can be included in any HTML page.

7.3 Improvements

The site works on all monitors in landscape format up to a very low resolution of around 800x600. On lower resolutions, formatting errors may occur. There currently is no re-

sponsive design included, so the website is not properly displayed on mobile devices or devices in portrait format. Due to time constraints, this can no longer be integrated into the project.

8. Conclusion

All in all, we are very pleased with the progress we made in this project. In its current state, the project simulates a realistic scene of an underwater ocean of an icy moon. Many sensors are included and fully simulated, their values are output both in a user interface that is accessible at runtime and via ROS. Therefore we would describe the current status as a success.

However, there is definitely some room for improvements. In many sections of this report there are already possible improvements mentioned. Due to a lack of people in the project, we couldn't realize many features, that very initially planned. Also the visualization could be improved. In the current state there are no animations included, which could be a possible improvement in the future.

Bibliography

[1] Ahkâm. Dust png transparent background. https://www.freeiconspng.com/img/35071, 2014. last accessed 24. March 2024.

[2] J. Amos. Saturn's enceladus moon hides 'great lake' of water. https://www.bbc.com/news/science-environment-26872184, 2014. last accessed: 2024-03-24.

[3] C. Angus and I. O. Tarifa. Kantan charts. https://github.com/kamrann/KantanCharts, 2016. last accessed 31.03.2024.

[4] BlueRobotics. T200 thruster. https://bluerobotics.com/store/thrusters/t100-t200-thrusters/t200-thruster-r2-rp/. last accessed 10.04.2024.

[5] CBS TEXAS. Plankton light the sea in wales: 'it really is something magical'. https://www.youtube.com/watch?v=eWIhn16RsHU, 2018. last accessed 25. March 2024.

[6] CGVR. VaMEx-VTB - U Bremen. https://cgvr.cs.uni-bremen.de/research/vamex-vtb/. last accessed: 2024-03-22.

[7] W.-S. Choi, D. R. Olson, D. Davis, M. Zhang, A. Racson, B. Bingham, M. McCarrin, C. Vogt, and J. Herman. Physics-Based Modelling and Simulation of Multibeam Echosounder Perception for Autonomous Underwater Manipulation. Frontiers in Robotics and AI, 8, Sept. 2021.

[8] CodeLikeMe. Unreal engine 5 - sound ripple effect. https://www.youtube.com/watch?v=0Hk7Ykcq4i8, 2022. last accessed 25.03.2024.

[9] T. J. Crone, W. S. D. Wilcock, A. H. Barclay, and J. D. Parsons. The Sound Generated by Mid-Ocean Ridge Black Smoker Hydrothermal Vents. PLOS ONE, 1(1):e133, Dec. 2006.

[10] J. Engebrecht, K. Nealson, and M. Silverman. Bacterial bioluminescence: isolation and genetic analysis of functions from vibrio fischeri. Cell, 32(3):773–781, 1983.

[11] Epic Games. Quixel bridge plugin for unreal engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/quixel-bridge-plugin-for-unreal-engine?application_version=5.0. last accessed 27. March 2024.

[12] Epic Games. Runtime virtual texturing - an overview of runtime virtual textures in unreal engine. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/VirtualTexturing/Runtime/. last accessed 26. March 2024.

[13] EVNautilus. Spectacular methane hydrate bubble plumes | nautilus live. https://www.youtube.com/watch?v=LajZxIQmZzs, 2016. last accessed 26. March 2024.

[14] EvoLogics. S2C R 48/78 USBL | EvoLogics. https://evologics.com/product/s2c-r-48-78-usbl-25. last accessed: 2024-03-23.

[15] S. Francia. Hugo. https://gohugo.io/. last accessed: 2024-03-23.

[16] GameDev Outpost. Ue4 - niagara export particle data to blueprint - collision. https://www.youtube.com/watch?v=aA_8NLzbUTA, 2021. last accessed 20. March 2024.

[17] V. Golovin. Caviar. https://www.filterforge.com/filters/209.html, 2010. last accessed 24. March 2024.

[18] V. Golovin. Organics. https://www.filterforge.com/filters/218.html, 2010. last accessed 24. March 2024.

[19] J. H. W. Jr., W. S. Lewis, B. A. Magee, J. I. Lunine, and W. B. McKinnon. Liquid water on enceladus from observations of ammonia and 40ar in the plume. In Nature, Volume 460 Nr. 7254, pages 487–490, july 2009.

[20] Kenney.nl. Particle pack (80+ sprites). https://opengameart.org/content/particle-pack-80-sprites, 2018. last accessed 24. March 2024.

[21] Klogg23. Black smoker (hydrothermal vent). https://sketchfab.com/3d-models/black-smoker-hydrothermal-vent-7210f54bb30943ea80347472ce00b64f, 2020. last accessed 21. March 2024.

[22] J. P. Laboratory. Saturn moon's ocean may harbor hydrothermal activity. https://web.archive.org/web/20151204083326/http://solarsystem.nasa.gov/news/display.cfm?News_ID=48922NASA, 2015. last accessed: 2024-03-24.

[23] H. Leila. Oh Snap! What Tiny Shrimp Can Tell Us About Habitat Health. https://sanctuaries.noaa.gov/news/dec21/oh-snap.html, Dec. 2021. last accessed: 2024-03-25.

[24] Marum. Marum - dem meer auf den grund gehen! https://www.marum.de. last accessed 21. March 2024.

[25] Marum. Successful sea trials on HEINCKE-expedition. https://www.marum.de/en/The-Ocean-Floor/Successful-sea-trials-on-HEINCKE-expedition.html, Mar. 2022. last accessed: 2024-03-24.

[26] MarumTV. Nachgefragt - folge 1: Wie entstehen schwarze raucher? https://www.youtube.com/watch?v=rhrI-JpET3c, 2019. last accessed 20. March 2024.

[27] T. Miyashiro and E. G. Ruby. Shedding light on bioluminescence regulation in vibrio fischeri. Molecular microbiology, 84(5):795–806, 2012.

[28] C. Morency, D. J. Stilwell, and S. Hess. Development of a Simulation Environment for Evaluation of a Forward Looking Sonar System for Small AUVs. In OCEANS 2019 MTS/IEEE SEATTLE, pages 1–9, Oct. 2019. arXiv:2210.06535 [cs, eess].

[29] B. Murray. Ocean floor of enceladus. https://www.planetary.org/space-images/ocean-floor-of-enceladus. last accessed: 2024-03-25.

[30] NASA. Nasa cassini data reveals building block for life in enceladus' ocean. https://www.nasa.gov/missions/cassini/nasa-cassini-data-reveals-building-block-for-life-in-enceladus-ocean/. last accessed: 2024-03-24.

[31] C. Nisbet. Ice chunk sprite 2.0. https://www.filterforge.com/filters/12607.html, 2014. last accessed 26. March 2024.

[32] D. Oertel. Deep-Sea Model-Aided Navigation Accuracy for Autonomous Underwater Vehicles Using Online Calibrated Dynamic Models. 2018. doi: 10.5445/IR/1000081117.

[33] Pixabay. Sonar ping. https://pixabay.com/de/sound-effects/sonar-ping-95840/. last accessed: 2024-03-25.

[34] Pixabay. Underwater Ambience. https://pixabay.com/de/sound-effects/underwater-ambiencewav-14428/. last accessed: 2024-03-25.

[35] Quixel. Quixel bridge. https://quixel.com/bridge. last accessed 21. March 2024.

[36] G. Robits. Imu and robotics: All you need to know. https://www.generationrobots.com/blog/en/imu-and-robotics-all-you-need-to-know/, 2020. last accessed 31.03.2024.

[37] M. Sajnóg. Aos. https://github.com/michalsnik/aos. last accessed: 2024-03-23.

[38] Subsea Imaging. Particle pack (80+ sprites). https://de.subcimaging.com/case-studies/faq-underwater-lasers-for-subsea-imaging, 2016. last accessed 24. March 2024.

[39] A. Suzuki, S. Hakura, T. Hamura, M. Hattori, R. Hayama, T. Ikeda, H. Kusuno, H. Kuwahara, Y. Muto, K. Nagaki, R. Niimi, Y. Ogata, T. Okamoto, T. Sasamori, C. Sekigawa, T. Yoshihara, S. Hasegawa, K. Kurosawa, T. Kadono, A. M. Nakamura, S. Sugita, and M. Arakawa. Laboratory experiments on crater scaling-law for sedimentary rocks in the strength regime. Journal of Geophysical Research: Planets, 117(E8):2012JE004064, Aug. 2012. ,doi:10.1029/2012JE004064, issn:0148-0227.

[40] Upklyak. Brown dusty cloud. https://www.freepik.com/free-vector/brown-dusty-cloud_6538010.htm#from_view=detail_alsolike. last accessed 24. March 2024.

[41] Wikipedia. Gimp. https://en.wikipedia.org/wiki/GIMP#:~:text=GNU%20Image%20Manipulation%20Program%2C%20commonly,formats%2C%20and%20more%20specialized%20tasks. last accessed 27. March 2024.

[42] Wikipedia. Code - heatmaps and color gradients. https://www.andrewnoske.com/wiki/Code_-_heatmaps_and_color_gradients, 2019. last accessed 23. March 2024.