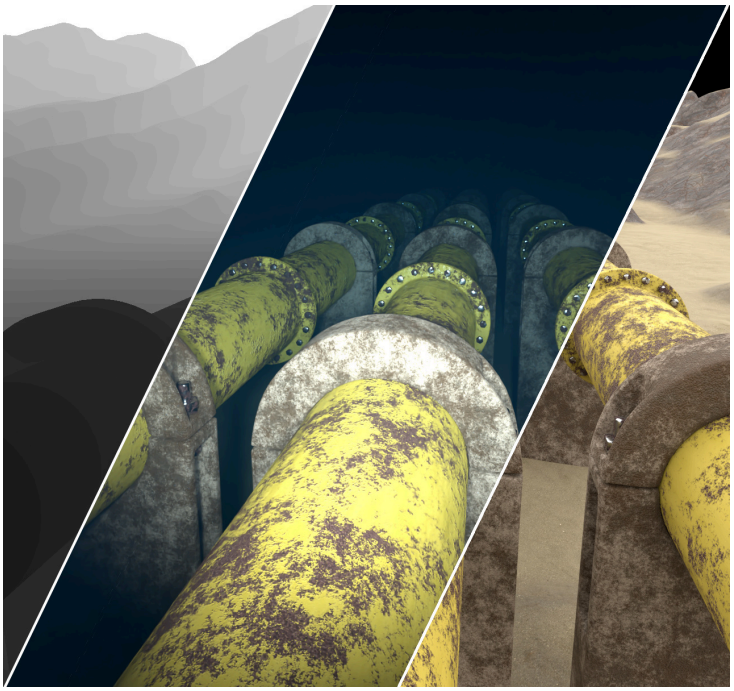
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Peder Georg Olofsson Zwilgmeyer

Creating a Synthetic Underwater Dataset for Egomotion Estimation and 3D Reconstruction

Master's thesis in Cybernetics and Robotics
Supervisor: Assoc. Prof. Annette Stahl
Co-supervisor: PhD. Student Mauhing Yip, Prof. Rudolf Mester
May 2021



NTNU
Norwegian University of
Science and Technology

Peder Georg Olofsson Zwilgmeyer

Creating a Synthetic Underwater Dataset for Egomotion Estimation and 3D Reconstruction

Master's thesis in Cybernetics and Robotics
Supervisor: Assoc. Prof. Annette Stahl
Co-supervisor: PhD. Student Mauhing Yip, Prof. Rudolf Mester
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Department of Engineering Cybernetics

TTK4900 - Master Thesis

Creating a Synthetic Underwater Dataset for Egomotion Estimation and 3D Reconstruction

Author:
Peder Georg Olofsson
Zwilgmeyer

Supervisor
Assoc. Prof. Annette Stahl

Co-supervisors
PhD. Student Mauhing Yip
Prof. Rudolf Mester

May, 2021

Preface

This thesis is the end result of a year-long cooperation with the AROS Vision Group. I wish to thank this group as a whole for the close knit cooperation throughout the entire year; my supervisor, Annette Stahl, and my co-supervisors Rudolf Mester and Mauhing Yip. I have to extend an additional thank you to Mauhing Yip for going out of his way to help with various topics, shedding light on mistakes I have overlooked and how to properly solve it. Finally, I want to thank my colleague Andreas Teigen for excellent cooperation in solving the problem of using the available hardware for rendering images.

Lastly, I am very thankful that Annette has agreed to make the AILARON server available, and NTNU's IDUN compute cluster for providing access to their GPUs for us to use when rendering images for the final benchmark sequence.

Peder Zwilgmeyer
NTNU, Trondheim
31.05.2021

Abstract

This thesis has the overall goal of creating an underwater dataset, complete with sensor data and ground truth, in cooperation with NTNUs Autonomous Robots for Ocean Sustainability (AROS) project and the branch of the AROS Vision Group, a subgroup whose main task is egomotion estimation, 3D reconstruction and environmental awareness. At the time of writing, there are no synthetic underwater datasets with physically correct images and complete ground truth data, and as such we hope to be delivering a new benchmark suite to the underwater community.

In order to create this dataset, a framework for trajectory creation and sensor measurement generation is created. To generate a trajectory, we use a physical model of the vehicle together with a control system which follows a predetermined path in a virtual underwater environment. Simultaneously the states generated from the control system are used to generate synthetic acceleration, angular velocity and depth measurements.

To be able to create a proper underwater dataset, a proper 3D environment is needed. Taking inspiration from real-world photogrammetry scans, we are able to submerge real landscapes and give them textures to suit an underwater environment. In addition, the scene it populated with various geometry to create a scene with features ranging from a simple sand floor to vertical rocks and complex geometry with significant amounts of occlusion.

The end result is a framework which is ready to generate longer image sequences, where the entire range of features in the scene is explored. Two papers are planned from this thesis, one detailing the dataset and the included data, and one which explains the implementation of the simulation framework.

Sammendrag

Denne oppgaven har som mål å lage et dataset fra et virituelt undervannsmiljø som inneholder bilder, sensordata og korrekt refereansedata. Dette gjøres i samarbeid med NTNUs "Autonomous Robots for Ocean Sustainability" (AROS) gruppe, og deres undergruppe AROS Vision Group som fokuserer på bruke et eller flere kamera ombord i roboten til 3D rekonstruksjon og til å spore hvor roboten er og har vært. Når denne oppgaven ble skrevet finnes det ingen undervanns dataset laget med syntetiske bilder som både gir tilgang til fysisk korrekte bilder, sensor data og komplett referanse data. Derfor håper vi at dette vil være et nyttig verktøy for undervannsmiljøet som helhet.

For å lage dette datasettet trenger må et rammeverk for generering av kjøretøyets bane i miljøet og sensor målinger lages. For å generere en bane gjennom det virituelle miljøet brukes en fysisk modell av kjøretøyet sammen med et styresystem som bruker en grov bane gjennom miljøet som referanse. Dataen fra styresystemet brukes så til å generere syntetiske målinger for akselerasjon, vinkel hastighet og kjøretøyets dybde i havet.

Dersom bildene skal være realistiske er det påkrevd med et 3D miljø som gjenspeiler det som kan forventes å finne under vann. Vi benytter 3D modeller som er rekonstruert fra bilder, hvor teksturene blir tilpasset det som kan forventes for et undervannsmiljø. I tillegg til dette legges ulike 3D modeller til undervannsmiljøet, slik at det inneholder områder med alt fra kun en enkel sandbunn til vertikale steinformasjoner, til områder med kompleks geometri.

Resultatet av denne oppgaven er et rammeverk som er klar til å generere lengre sekvenser, hvor alle de ulike områdene i undervannsmiljøet er utforsket. Fra denne oppgaven er det planlagt å skrive to artikler, en som tar for seg datasettet og inkludert data, og en som beskriver simuleringsrammeverket vi har brukt for å generere denne dataen.

Table of Contents

1 Introduction

1.1 Motivation

There are a number of datasets available today for development of computer vision algorithms such as Visual Odometry (VO), Structure from Motion (SfM), 3D reconstruction and Visual Simultaneous Localization and Mapping (VSLAM). The majority of the datasets consists of recorded above-water RGB images, with varying degrees of sensor- and ground truth information available.

The datasets KITTI[1] and LiU[2] are made using recorded information from cars augmented with an array of sensors. They deliver both images and sensor measurements from an inertial measurement unit, hereafter abbreviated IMU, that measures the vehicles acceleration and angular velocity, velocity sensors and a GPS to mention the main sensors.

Aerial datasets have been created, one of which is the EuRoC[3] dataset. In addition to providing images and IMU data, this dataset provides ground truth position and pointcloud measurements with $\approx 1mm$ accuracy from a fixed Leica MS50 Multistation.

However, for underwater applications GPS signals are attenuated and cannot be used, and the use of multistations is impossible. In the underwater dataset AQUALOC[4], Colmap[5] is used to create a very accurate offline 3D reconstruction of the area surveyed by the camera. Depth-, IMU and magnetometer measurements are included as well. While Colmap can create very good results, the ground truth is still based on the camera input, which quality is heavily dependent on the visual conditions present. Sonar is an underwater alternative which is independent of visual conditions, and was used in combination with a camera when creating a dataset of an underwater cave complex[6].

As previously mentioned, one of the larger issues of underwater benchmarks is the lack of proper absolute ground truth data for positioning. Another key point is that current weather conditions for sallow waters, algae growth or marine snow[1], can significantly impact the visual quality of the acquired images. As the methods from acquiring ground truth from [4, 6] are both dependent on camera input, changes in the visual conditions can negatively impact the quality of the generated ground truth data.

By generating the data for the sequences using simulations and computer generated images, it is possible to achieve realistic imagery and have access to completely accurate information about the vehicle's pose and simulated sensor measurements for every timestep. As the 3D geometry of the scene is known, this can be used to generate correct pointclouds and depth maps in all environments that can be simulated. The COnGRATS[7] dataset is one synthetic traffic dataset including such information coupled with realistic path-traced imagery.

Underwater simulations for generating benchmarking sequences is explored to a much lesser extent. UWSim[8] is one commonly used simulator which provides a solid simulation framework from simulated sensors to network interfaces. However, this framework does not create physically correct path-traced imagery. Another approach commonly used is to use image mosaics from the sea floor, and generate images with a desired level of turbidity[2][9]. This approach does require prior knowledge of a scene in the form of an image mosaic, which should be as close to the real colors of the seafloor as possible. However, by using a flat plane the use is severely restricted. It has to be viewed from the top down, and shadows for a nearby light source will not cast shadows from the apparent 3D geometry present in the image.

None of the methods above provides access to an underwater simulation environment, complete with both simple and complex 3D models with physically correct textures, the possibility to place cameras and lights where needed, and most importantly, physically correct path-traced images. By creating a simulation framework with these criteria in mind, it is possible to test extensively test computer vision algorithms with access to complete ground truth data for result comparison.

[1] Marine snow inorganic matter which sinks to the seafloor looking similar to snow

[2] Turbidity: Large amounts of suspended particles in the water causes it to be cloudy or hazy.

1.2 Objective and Scope

This thesis aims to create a framework for underwater image simulation, together with a proper benchmark sequence for underwater computer vision applications. Using Blender[10], an open source 3D software suite capable of 3D modelling, generating physically correct images and physics simulations, as our simulation tool allows us to create physically correct path-traced image, together with other types of image data as ground truth information. At the time this thesis is delivered, this information includes pixel-perfect depth images, surface world-normal vectors for each pixel, direct light and shadow masks for each pixel, indirect volumetric scattering and uniformly lit RGB images. In addition to this, sensor measurements from an IMU and depth sensor are generated. These are delivered as noise-free data, such that they can later be augmented with noise to fit a wide range of sensor-models depending on the users application.

These sequences require physically correct motion, and the vehicle itself is given a mathematical model with defined inputs, which is controlled by a control system using a series of waypoints around the underwater scene as references. Due to the wide scope of the task, the vehicle model is made to be mathematically simple, yet complex enough to provide physically correct motion.

1.3 Contributions

The main contribution of this thesis is the creation of a underwater sequence benchmark containing images, IMU and depth data, as well as a series of corresponding ground truth data which pose a significant challenge to acquire from sequences recorded in a real-world environment.

While each sub-system of the thesis isn't necessarily a novel idea, the resulting framework provides data and simulation possibilities that has previously been unavailable.

Initially this framework was intended for internal use in the AROS Project for development of underwater computer vision algorithms. Due to the limited availability of underwater datasets with proper ground truth, we now aim to publish this dataset, where we stand to be one of the first to publish a synthetic underwater dataset with physically correct path traced images. After some time, the simulation framework will also be made available for the underwater community at large to use and adapt to their needs.

1.4 Outline

The motivation and objective of the thesis has already been explained in this section. Next is the vehicle control system, abbreviated VCS, in section 2. This section provides motivation for the need of a VCS, the system model, controller design and how waypoints are used to generate a path through the environment.

After the VCS is defined, the IMU is derived in section 3. This section contains an overview of the coordinate frames and transformations used and discusses the potential for state interpolation for high-frequency IMU measurements before the derivation of the mathematical model for the IMU sensor itself.

Section 4 provides an overview and explanation of the data generated by the simulation framework. This includes sensor data from the IMU and depth sensors, image data, ground truth data, image naming convention and the final data structure.

Section 5 outlines the requirements and setups for the simulation itself. This includes how the simulation environment was created, which assets were used and motivates these choices. The section then provides information on the camera setup to accurately simulate a camera with matching intrinsic parameters to that of NTNU's EELY500 Eelume[11] robot. Finally, the image rendering parameters and an automated render process is discussed.

Finally, section 6 discusses the potential for future improvements of the simulation framework. The framework in its current state is in no way finished, and has a lot of potential for increased

functionality and more complex vehicle models. Plans for paper publishing and dataset availability wraps up this section.

1.5 Notation

Table 1.1: Commonly used symbols

Symbol	Description
c	Rotational drag coefficient
C_d	Drag coefficient
ζ	Water dynamic viscosity $[Pa\ s]$
F	Force $[N]$
F_d	Drag force $[N]$
I	Moment of inertia $[kg\ m^2]$
m	Mass $[kg]$
Φ	Radiant flux $[W]$
ρ	Density $[kg/m^3]$
r	Radius $[m]$
$s,\ \boldsymbol{s}$	State, displacement $[m]$
θ	Angle $[rad]$
τ	Torque (VCS), Current timestep (IMU)
t	Time $[s]$
Δt	Time-step $[s^{-1}]$
$\boldsymbol{a}$	Acceleration, sensor measurement
d	Depth, sensor measurement
$\omega,\ \boldsymbol{\omega}$	Angular velocity, scalar and vector, sensor measurement
A	Cross-sectional area
$\boldsymbol{A}$	Linear system matrix
$\boldsymbol{B}$	Linear system input matrix
$\boldsymbol{C}$	Linear system output matrix
$\boldsymbol{K}$	LQR state feedback matrix
$\boldsymbol{\eta}$	State vector
$\boldsymbol{\eta}_r$	State reference vector
$\boldsymbol{P}$	LQR feed-forward matrix
$\bar{\boldsymbol{Q}}$	LQR state weighting matrix
$\bar{\boldsymbol{R}}$	LQR input weighting matrix
$\boldsymbol{R}$	Rotation matrix
$\mathbb{R}^n$	Real coordinate space of dimension n
$\mathbb{SE}(n)$	Special Euclidean group of dimension n
$\mathbb{SO}(n)$	Special Orthogonal group of dimension n
$u, \boldsymbol{u}$	System input, scalar and vector
$\boldsymbol{v}_0,\ \boldsymbol{v}_1$	Input dynamics states
$\boldsymbol{V}$	Input dynamics system matrix
$\boldsymbol{y}$	Linear system output
λ	Eigenvalue of a linear system (VCS), wavelength (Simulation)
$x,\ y,\ z$	Coordinate frame axes

Table 1.2: Commonly used abbreviations

Abbreviation	Description
GPS	Global Positioning System
CCW	Counter-clockwise
CPU	Central Processing Unit
CW	Clockwise
GPU	Graphics Processing Unit
IMU	Inertial Measurement Unit
LQR	Linear Quadratic Regulator
RGB	Red, Geen, Blue
sRGB	Standard RGB
VCS	Vehicle Control System
VO	Visual Odometry
VSLAM	Visual Simultaneous Localization And Mapping

Positive rotation direction: The direction for a positive rotation around a given axis follows the right hand rule. Point the thumb in the positive direction of the axis of rotation, then the fingers will point in the direction of a positive rotation. If the axis of rotation comes out of a 2D plane, then the positive rotation direction is counter-clockwise (CCW) following this convention.

Coordinate frame orientation: All coordinate frames follow the right-hand rule. Unless specified, an arbitrary coordinate system has a positive z-direction in the "up" direction, whereas the xy-plane spans a flat horizontal plane.

Coordinate frame points: A vector $\boldsymbol{v}$ expressed in a coordinate system $\boldsymbol{a}$ is written as $\boldsymbol{v}^a$.

Coordinate frame rotations: Unless specified, all rotations are extrinsic. This means, the frame of which the rotations are performed is fixed, and does not change with each rotation.

A rotation from frame $\boldsymbol{b}$ to $\boldsymbol{a}$ expressed in frame $\boldsymbol{a}$ is written as $\boldsymbol{R}_{ab}^a$. The superscript denotes the rotation frame of reference, whereas the subscript first denotes the end frame, then the start frame.

Coordinate frame transformations: The same notation as expressed for coordinate frame rotations applies to coordinate frame transformations.

Forces, torque, inputs and states: Forces, torque, inputs or a givens state that is dependent on a local axis has the corresponding local axis marked as the last subscript. For instance, the drag force of an object is denoted F_d. To further specify that it is the drag force along the x-axis, the axis subscript is appended: $F_{d,x}$

2 Vehicle control system

The vehicle control system creates a model of the virtual vehicle in the simulation, which is controlled using linear control theory to obtain smooth and continuous motion.

2.1 Motivation

Blender[10] contains tools for animation where objects can move between poses set at different keyframes, using Bezier-curves[12] where the default interpolation uses with linear acceleration. These curves can be manually adjusted to create a desired motion, however this is a very time consuming animation job which doesn't guarantee a physically correct vehicle trajectory. Please note that Blender is operating with frames, hence frames is used analogous with time in this subsection. Let P_1, P_2 and P_3 denote three poses for the vehicle in three consecutive keyframes. The acceleration $a_{12}(t)$ between P_1 and P_2 will be linear, with an initial magnitude determined by the distance between the keyframes and the number of frames between these keyframes. When P_2 is reached, the linear acceleration is set to $a_{23}(t)$. Due to differences in the number of frames between P_2 and P_2 compared to P_1 and P_2, and the possibility of a different distance. For instance, say that P_2 is located at $t = \tau$. In this transition, the identity $a_{12}(\tau) = a_{23}(\tau)$ does generally not hold with the exception of a very few corner cases.

2.2 Translation and rotation definitions

In the vehicle control system, hereafter abbreviated *VCS*, the translation and rotation states are defined as follows. Translation is measured in meters $[m]$ along their respective axes. Rotations are defined as total counter clockwise (CCW) rotation around their respective axes in radians $[rad]$. Rotation matrices are not used as they can be constructed from the Euler angles, and as well as storing the total CCW rotation eliminates the previous angle-looping issue discussed in the IMU measurement generation. For this to work, the rotations in the reference vector exported from Blender is given in total CCW Euler rotation.

Gimbal lock is another important phenomenon which can cause issues. In this implementation, each rotation axis is fixed in the world frame. Hence the axes are always orthogonal, avoiding the issue of certain rotations causing a gimbal lock.

2.3 System model

The system model models a virtual underwater sphere as the vehicle used for moving throughout the simulation environment. In order to generate physically correct motion, it is given reasonable size, mass and input limitations. In order to be relatively easily able to linearize the model, the translational and rotational input is assumed to be decoupled. In this implementation the inputs are virtual torques and forces acting on each of the vehicle's axes. To contribute to the realistic motion, translational and rotational drag forces are included.

2.3.1 Partial 2D model

The version is modeled in the 2D plane to simplify the system matrices during the initial modelling stage. As a first version, it is desired that the model should be not too complex to derive while still being sufficiently complex to showcase the desired motion characteristics. Hence a 2D version is used, showcased in fig. 2.1.

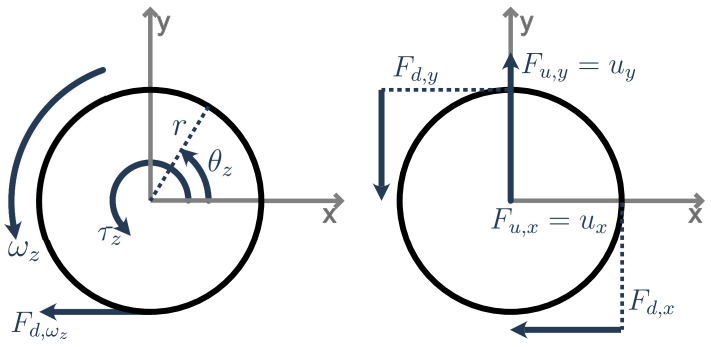


Figure 2.1: A 2D version of the vehicle. Here the XY-plane is visible, with the positive z-axis pointing out of the paper. $F_{u,x} = u_x$ and $F_{u,y} = u_y$ denotes the input forces along the x- and y-axes respectively. τ_z is the torque, and ω_z is the angular velocity around the z-axis. F_{d,ω_z} denotes rotational frictional drag forces, and $F_{d,x}$ and $F_{d,y}$ denotes the translational drag forces.

First the translational equations of motions are considered. By summing the translational forces along the x-axis, the acceleration $\ddot{s_x}$ is found:

$$\begin{aligned} \sum F &= F_{u,x} - F_{d,x} \\ m\ddot{s}_x &= u_x - \frac{1}{2}\rho C_d A \dot{s}_x^2 \\ m\ddot{s}_x &= u_x - d\dot{s}_x^2, \qquad d = \frac{1}{2}\rho C_d A \\ \ddot{s}_x &= \frac{1}{m}u_x - \frac{d}{m}\dot{s}_x^2 \end{aligned} \tag{2.1}$$

Where $F_{d,x}$ is the translational drag along the x-axis and $F_{u,x} = u_x$ is the input force along the axis. For a small $\dot{s}_x$, it is possible to linearize the equations of motion around $\dot{s}_x = 0$.

This linearization requires that the second-degree friction term $F_{d,x}$ can be expressed linearly in $\dot{s}_x$. Therefore, the friction model is modified to be linearly dependent on the velocity $\dot{s}_x$, instead of the quadratic dependency presented in eq. (2.1). If the system were to be linearized around the Origin before this simplification, the corresponding index for the drag forces in the system matrix $\boldsymbol{A}_{s,x}$ would be zero. As the simulated craft is a relatively slow moving exploring underwater vehicle, the error resulting from this simplification will stay small. This simplification of eq. (2.1) results in the following equation:

$$\ddot{s}_x = \frac{1}{m}u_x - \frac{d}{m}\dot{s}_x \tag{2.2}$$

Given the system structure:

$$\dot{\boldsymbol{\eta}}_{s,x} = \mathbf{A}_{s,x}\boldsymbol{\eta}_{s,x} + \boldsymbol{B}_{s,x}u_{s,x}, \qquad \boldsymbol{\eta}_{s,x} = [s_x, \dot{s}_x]^\top, \quad \dot{\boldsymbol{\eta}}_{s,x} = [\dot{s}_x, \ddot{s}_x]^\top$$

the 2-by-2 matrix $\mathbf{A}_x$ is found by linearizing as follows:

$$\begin{aligned} \dot{\boldsymbol{\eta}}_x &= \begin{bmatrix} \frac{\delta}{\delta s_x}\dot{s}_x & \frac{\delta}{\delta \dot{s}_x}\dot{s}_x \\ \frac{\delta}{\delta s_x}(-\frac{d}{m}\dot{s}_x) & \frac{\delta}{\delta \dot{s}_x}(-\frac{d}{m}\dot{s}_x) \end{bmatrix} \begin{bmatrix} s_x \\ \dot{s}_x \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u_x \\ &= \begin{bmatrix} 0 & 1 \\ 0 & -\frac{d}{m} \end{bmatrix} \begin{bmatrix} s_x \\ \dot{s}_x \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u_x \\ &= \boldsymbol{A}_{s,x}\boldsymbol{\eta}_{s,x} + \boldsymbol{B}_{s,x}u_x \end{aligned} \tag{2.3}$$

Now that a model for the translation is established, it is time to create a model of the rotation. Let's examine the rotation around the z-axis in fig. 2.1 with the input torque $\tau_{u,z}$. Let I denote the moment of inertia of the sphere which is modelled as a solid sphere with mass m and radius r. Then the inertia is given as

$$I = \frac{2}{5}mr^2.$$

When rotating, there is a small amount of rotational friction. Let c denote the drag coefficient for rotational friction for a sphere. From seqtion 4.3.1: motion of a single hard sphere in [13], we have that the rotational drag coefficient c is then given by

$$c = 6\pi\zeta r \tag{2.4}$$

where ζ is the dynamic viscosity of water. For a temperature of 5 degrees Celsius, from table 7 in [14], we have that

$$\zeta = 1519.3\mu Pa\ s = 1.5193 \cdot 10^{-3} Pa\ s.$$

Applying Newtons second law for rotation around the z-axis and applying that $\tau_{z,d} = F_{d,\omega_z} r = (cU)r$, where U is the surface velocity $U = (\omega_z r)$, from [13], we have:

$$\begin{aligned}
\tau_z &= \tau_{u,z} - \tau_{z,d} \\
&= \frac{d}{dt}(I\omega_z) - F_{d,\omega_z} r \\
&= \frac{d}{dt}(I\omega_z) - cr(r\omega_z) \\
&= I\ddot{\theta}_z - cr^2\dot{\theta}_z \\
&\Updownarrow \\
\ddot{\theta}_z &= -\frac{cr}{I}\dot{\theta}_z + \frac{1}{I}\tau_{u,z}
\end{aligned} \tag{2.5}$$

The resulting linear system of equations is as follows:

$$\begin{aligned}
\dot{\boldsymbol{\eta}}_{\theta,z} &= \begin{bmatrix} \frac{\delta}{\delta\theta_z}\dot{\theta}_z & \frac{\delta}{\delta\dot{\theta}_z}\dot{\theta}_z \\ \frac{\delta}{\delta\theta_z}(-\frac{cr}{I}\dot{\theta}_z) & \frac{\delta}{\delta\dot{\theta}_z}(-\frac{cr}{I}\dot{\theta}_z) \end{bmatrix} \begin{bmatrix} \theta_z \\ \dot{\theta}_z \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{I} \end{bmatrix} \tau_{u,z} \\
&= \begin{bmatrix} 0 & 1 \\ 0 & -\frac{cr}{I} \end{bmatrix} \begin{bmatrix} \theta_z \\ \dot{\theta}_z \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{I} \end{bmatrix} \tau_{u,z} \\
&= \boldsymbol{A}_{\theta,z}\boldsymbol{\eta}_{\theta,z} + \boldsymbol{B}_{\theta,z}\tau_{u,z}
\end{aligned} \tag{2.6}$$

As the translation and rotation are decoupled in this model, creating the overall model is done by combining the subsystems for the translation along the $x-$ and $y-$axes and the rotation around the $z-$axis. Let

$$\dot{\boldsymbol{\eta}}_{2D} = \begin{bmatrix} s_x, \dot{s}_x, s_y, \dot{s}_y, \theta_z, \dot{\theta}_z \end{bmatrix}^\top$$

denote the state vector for the 2D model in the XY-plane. Note that $\boldsymbol{A}_{s,x} = \boldsymbol{A}_{s,y} = \boldsymbol{A}_s$. This is also true for $\boldsymbol{A}_{\theta,z} = \boldsymbol{A}_\theta$ Then the full 2D model is created:

$$\begin{aligned}
\dot{\boldsymbol{\eta}}_{2D} = \begin{bmatrix} \dot{s}_x \\ \ddot{s}_x \\ \dot{s}_y \\ \ddot{s}_y \\ \dot{\theta}_z \\ \ddot{\theta}_z \end{bmatrix} &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -\frac{d}{m} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{d}{m} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -\frac{c}{I} \end{bmatrix} \boldsymbol{\eta}_{2D} + \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{1}{m} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{I} \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ \tau_{u,z} \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{A}_s & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{A}_s & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{A}_\theta \end{bmatrix} \boldsymbol{\eta}_{2D} + \begin{bmatrix} \boldsymbol{B}_{s,x} \\ \boldsymbol{B}_{s,y} \\ \boldsymbol{B}_{\theta,z} \end{bmatrix} \boldsymbol{u}_{2D}
\end{aligned} \tag{2.7}$$

2.3.2 Full 3D model

The 3D model is an expansion of the 2D model, where the only difference is more degrees of freedom where each extra subsystem is decoupled from the others. The complete model is shown in fig. 2.2.

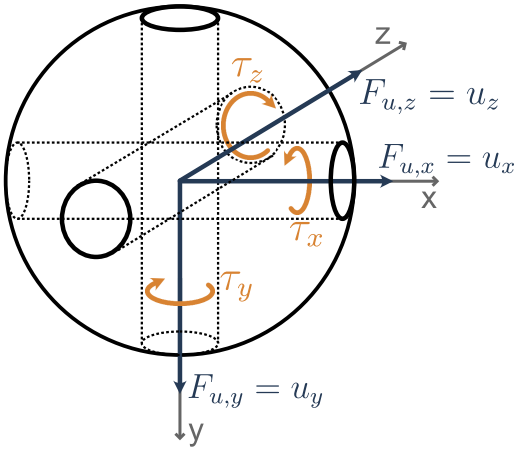


Figure 2.2: A 3D version of the vehicle, where $\tau_x/\tau_y/\tau_z$ and $u_x/u_y/u_z$ are the torques and input forces around each of the $x-, y-$ and z-axes.

The mathematical derivations are similar to that of the 2D model with only different subscripts for the new axes. Hence the full system will be directly shown as all relevant derivations are found in the 2D case. First, the state vector $\boldsymbol{\eta}$ is defined as:

$$\boldsymbol{\eta} = \left[s_x, \dot{s_x}, s_y, \dot{s_y}, s_z, \dot{s_z}, \theta_x, \dot{\theta_x}, \theta_y, \dot{\theta_y}, \theta_z, \dot{\theta_z}\right]^\top$$

Then the full model can be described with the following system:

$$\begin{aligned}
\dot{\boldsymbol{\eta}} = \begin{bmatrix} \dot{s}_x \\ \ddot{s}_x \\ \dot{s}_y \\ \ddot{s}_y \\ \dot{s}_z \\ \ddot{s}_z \\ \dot{\theta}_x \\ \ddot{\theta}_x \\ \dot{\theta}_y \\ \ddot{\theta}_y \\ \dot{\theta}_z \\ \ddot{\theta}_z \end{bmatrix} &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{d}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{d}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{d}{m} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{c}{I} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{c}{I} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{c}{I} \end{bmatrix} \begin{bmatrix} s_x \\ \dot{s}_x \\ s_y \\ \dot{s}_y \\ s_z \\ \dot{s}_z \\ \theta_x \\ \dot{\theta}_x \\ \theta_y \\ \dot{\theta}_y \\ \theta_z \\ \dot{\theta}_z \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{m} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{I} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{I} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{I} \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \\ \tau_{u,x} \\ \tau_{u,y} \\ \tau_{u,z} \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{A}_s & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{A}_s & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{A}_s & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{A}_\theta & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{A}_\theta & \boldsymbol{0}_{2,2} \\ \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{0}_{2,2} & \boldsymbol{A}_\theta \end{bmatrix} \boldsymbol{\eta} + \begin{bmatrix} \boldsymbol{B}_{s,x} \\ \boldsymbol{B}_{s,y} \\ \boldsymbol{B}_{s,z} \\ \boldsymbol{B}_{\theta,x} \\ \boldsymbol{B}_{\theta,y} \\ \boldsymbol{B}_{\theta,z} \end{bmatrix} \boldsymbol{u} \\
&= \boldsymbol{A}\boldsymbol{\eta} + \boldsymbol{B}\boldsymbol{u}
\end{aligned} \tag{2.8}$$

2.4 Controller design

The purpose of a controller is to control the input $\boldsymbol{u}$ such that the states $\boldsymbol{\eta}$ approaches desired references stored in a reference vector $\boldsymbol{\eta}_r$. The controller used is an infinite horizon Linear Quadratic Regulator, hereafter LQR for short. This controller will be introduced in depth in section 2.4.3.

2.4.1 Controllability

Before designing the controller, the system needs to be verified as controllable. That means, the states can be controlled through the input. From observation it is clear that the system is controllable, as the input directly influences the double derivative of *every* state. From a mathematical point of view, controllability requires that the rank of the controllability matrix $\mathcal{C}$ is equal to the number of states in the system. For a system to be controllable, it must satisfy the condition presented in eq. (2.9), where $\mathcal{C}$ is a 12×72 matrix.

$$\mathcal{C} = \begin{bmatrix} \boldsymbol{B} & \boldsymbol{AB} & \boldsymbol{A}^2\boldsymbol{B} & \dots & \boldsymbol{A}^{11}\boldsymbol{B} \end{bmatrix}, \qquad rank(\mathcal{C}) = 12 \tag{2.9}$$

Due to the size of the aforementioned controllability matrix, the rank is determined by using Python and the NumPy library. Using a *for-loop* the matrix $\mathcal{C}$ is generated, where the rank is found by using the built-in NumPy function *numpy.linalg.matrix_rank*($\mathcal{C}$). This returns $rank(\mathcal{C}) = 12$, which means that the system in controllable and a controller can confidently be designed.

2.4.2 Reference states and output matrix

The reference state vector is:

$$\boldsymbol{\eta}_r = \begin{bmatrix} s_{x,ref}, s_{y,ref}, s_{z,ref}, \theta_{x,ref}, \theta_{y,ref}, \theta_{z,ref} \end{bmatrix}^\top \tag{2.10}$$

The controller is an infinite horizon LQR, hence $\lim_{t\to\infty} \boldsymbol{y} \Rightarrow \boldsymbol{\eta}_r$, where $\boldsymbol{y}$ is the measured states. This way, it is possible to determine the output matrix $\boldsymbol{C}$ such that the equation below holds:

$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{\eta} = \boldsymbol{\eta}_r \quad \Rightarrow \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} s_x \\ \dot{s}_x \\ s_y \\ \dot{s}_y \\ s_z \\ \dot{s}_z \\ \theta_x \\ \dot{\theta}_x \\ \theta_y \\ \dot{\theta}_y \\ \theta_z \\ \dot{\theta}_z \end{bmatrix} = \begin{bmatrix} s_{x,ref} \\ s_{y,ref} \\ s_{z,ref} \\ \theta_{x,ref} \\ \theta_{y,ref} \\ \theta_{z,ref} \end{bmatrix} \tag{2.11}$$

2.4.3 Linear quadratic regulator, LQR

Given the 6×6 feed-forward matrix $\boldsymbol{P}$ and the 6×12 state feedback matrix $\boldsymbol{K}$, the controller determines the input $\boldsymbol{u}$. The controller is written as

$$\boldsymbol{u} = \boldsymbol{P}\boldsymbol{\eta}_r - \boldsymbol{K}\boldsymbol{\eta}. \tag{2.12}$$

In order to find $\boldsymbol{P}$ and $\boldsymbol{K}$, it is assumed that the system enters an equilibrium state as $t \to \infty$. Substituting eq. (2.12) into eq. (2.8) yields:

$$\begin{aligned} \dot{\boldsymbol{\eta}} = \mathbf{0} &= \boldsymbol{A\eta} + \boldsymbol{Bu} \\ &= \boldsymbol{A\eta} + \boldsymbol{B}(\boldsymbol{P\eta}_r - \boldsymbol{K\eta}) \\ &= (\boldsymbol{A} - \boldsymbol{BK})\boldsymbol{\eta} + \boldsymbol{BP\eta}_r \end{aligned} \tag{2.13}$$

As an infinite horizon LQR controller is used, let $\lim_{t\to\infty} \boldsymbol{\eta} \Rightarrow \boldsymbol{\eta}_\infty$ and rewrite eq. (2.13):

$$\begin{aligned} \mathbf{0} &= (\boldsymbol{A} - \boldsymbol{BK})\boldsymbol{\eta}_\infty + \boldsymbol{BP\eta}_r \\ (\boldsymbol{BK} - \boldsymbol{A})\boldsymbol{\eta}_\infty &= \boldsymbol{BP\eta}_r \\ \boldsymbol{\eta}_\infty &= (\boldsymbol{BK} - \boldsymbol{A})^{-1}\boldsymbol{BP\eta}_r \end{aligned} \tag{2.14}$$

The output of the system in eq. (2.8) is given by:

$$\boldsymbol{y} = \boldsymbol{C\eta}$$

Substituting with $\lim_{t\to\infty} \boldsymbol{\eta} \Rightarrow \boldsymbol{\eta}_\infty$ and $\lim_{t\to\infty} \boldsymbol{y} \Rightarrow \boldsymbol{\eta}_r$, and inserting eq. (2.13) yields:

$$\begin{aligned} \boldsymbol{\eta}_r &= \boldsymbol{C\eta}_\infty \\ \boldsymbol{\eta}_r &= \boldsymbol{C}(\boldsymbol{BK} - \boldsymbol{A})^{-1}\boldsymbol{BP\eta}_r \\ \boldsymbol{I} &= \boldsymbol{C}(\boldsymbol{BK} - \boldsymbol{A})^{-1}\boldsymbol{BP} \\ \boldsymbol{C}(\boldsymbol{BK} - \boldsymbol{A})^{-1}\boldsymbol{BP} &= \boldsymbol{I} \\ \boldsymbol{P} &= (\boldsymbol{C}(\boldsymbol{BK} - \boldsymbol{A})^{-1}\boldsymbol{BP})^{-1} \end{aligned} \tag{2.15}$$

From this it is apparent that the state feedback matrix $\boldsymbol{K}$ is needed in order to find the reference feed-forward matrix $\boldsymbol{P}$. Finding $\boldsymbol{K}$ requires two additional matrices, the state weight matrix $\boldsymbol{Q}$ and input weight matrix $\boldsymbol{R}$. Further use of these two matrices during tuning will be discussed in section 2.4.5.

$\boldsymbol{K}$ is found through the LQR-solver in the *Python Control Systems Library*[15], using the function *control.matlab.lqr*$(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{Q}, \boldsymbol{R})$.

2.4.4 Input dynamics

It is important to address one dynamic not currently modelled: input ramp-up. Allowing the input torques and forces $\boldsymbol{u}$ to be directly controlled will result in discontinuous jumps in the input when a new reference is set, which directly translates to discontinuous jumps in acceleration. To avoid this, input dynamics are added. A physical equivalent to this dynamic is similar to that of a propeller. When a propeller rotates, it exerts a force on the medium around it, and the medium itself exerts a force back onto the propeller as stated by Newtons third law. This force is the current input. However, the propeller cannot go from 0 rotations per minute (RPM) to 800 RPM instantaneously. A real-world control system would control the *voltage* to the electric propeller, and the propeller would spin up to the RPM corresponding to the voltage level set. This is a huge simplification, as these dynamics encompass are more complex, but it illustrates the effect necessary to *emulate* in simulation data. The vehicles' acceleration should be smooth for the entirety of a simulated sequence.

Let $\boldsymbol{u}$ denote the forces and torques applied to the vehicle, and $\vee$ be the input to the input ramp-up system. Then an expression for $\boldsymbol{u}$ is:

$$\dot{\boldsymbol{u}} = \boldsymbol{V}\boldsymbol{u} + \boldsymbol{v}$$

where $\boldsymbol{V}$ is a diagonal matrix consisting entirely of the negative eigenvalues of the input ramp-up system. These eigenvalues determine poles and by extension the speed of the input ramp-up system. The poles should be chosen to be faster than that of the original system. The poles in question are found by solving $|\boldsymbol{I}\lambda_v - \boldsymbol{V}|$ and $|\boldsymbol{I}\lambda_{sys} - \boldsymbol{A}|$ for λ_v and λ_{sys} respectively.

Using *numpy.linalg.eig*($\boldsymbol{A}$), the poles for the vehicle dynamics are found to be the following (rounded to 5 decimals):

$$\boldsymbol{\lambda}_{sys} = \begin{bmatrix} 0, & -0.07854, & 0, & -0.07854, & 0, & -0.07854, & 0, & -0.00358, & 0, & -0.00358, & 0, & -0.00358 \end{bmatrix}^\top$$

The input ramp-up dynamics needs to be significantly faster than the system dynamics, hence the poles are chosen to lie further away in the left half plane. Introducing an input delay will be affecting the controller as well. Therefore the poles are chosen such that the input dynamics gives the desired effect on the acceleration, yet they are fast enough to not significantly interfere with the control system. The corresponding eigenvalues to $\boldsymbol{u} = [u_x, u_y, u_z, \tau_{u,x}, \tau_{u,y}, \tau_{u,z}]$ are chosen to be

$$\boldsymbol{\lambda}_v = \begin{bmatrix} \lambda_{u_x}, & \lambda_{u_y}, & \lambda_{u_z}, & \lambda_{\tau_{u,x}}, & \lambda_{\tau_{u,y}}, & \lambda_{\tau_{u,z}} \end{bmatrix}^\top \begin{bmatrix} -8, & -8, & -10, & -7, & -7, & -7 \end{bmatrix}^\top$$

through testing and evaluating the response time for a step-response. The desired response time is in the interval of $50ms - 120ms$.

Saturation limits for the inputs are placed on the input $\boldsymbol{v} \in [\boldsymbol{v}_{min}, \boldsymbol{v}_{max}]$ to ensure that when the saturation limit is reached, $\boldsymbol{u}$ transitions smoothly to the saturation limit instead of a sharp cutoff.

This First-order input dynamics did not produce the desired results, as the acceleration still has an abrupt jump when a new reference is given. Therefore another identical input dynamic system is added to add a second order response to the acceleration input. This adds the characteristic S-curve response to the acceleration response.

Let $\boldsymbol{v}_0$ denote the output from the control system, $\boldsymbol{v}_1$ the internal state of the second order system and let $\boldsymbol{u}$ denote the output from the second order acceleration dynamics. Then the system can be written as in eq. (2.16) with input limits placed on $\boldsymbol{v}_1$.

$$\begin{aligned} \dot{\boldsymbol{v}_1} &= \boldsymbol{V}\boldsymbol{v}_1 + \boldsymbol{v}_0 \\ \dot{\boldsymbol{u}} &= \boldsymbol{V}\boldsymbol{u} + \boldsymbol{v}_1 \end{aligned} \tag{2.16}$$

2.4.5 LQR tuning

Tuning of the LQR controller is done by weighting the matrices $\bar{\boldsymbol{Q}}$ and $\bar{\boldsymbol{R}}$. The diagonal entries of $\bar{\boldsymbol{Q}}$ correspond to the state in $\boldsymbol{\eta}$ at the given index. Likewise for the diagonal entries in $\bar{\boldsymbol{R}}$ and the inputs $\boldsymbol{v}$. The cost function which is minimized is shown in eq. (2.17).

$$\boldsymbol{J} = \int_0^\infty (\boldsymbol{\eta}^\top \bar{\boldsymbol{Q}} \boldsymbol{\eta} + \boldsymbol{v}^\top \bar{\boldsymbol{R}} \boldsymbol{v}) dt \tag{2.17}$$

By increasing $\bar{\boldsymbol{Q}}$ or decreasing $\bar{\boldsymbol{R}}$ the states are penalized. A common method is to set $\bar{\boldsymbol{R}} = \boldsymbol{I}$ and tune the state weights in $\bar{\boldsymbol{Q}}$. However in this implementation they were tuned in tandem. The values used are shown below:

$$\begin{aligned} \bar{\boldsymbol{Q}} &= diag([100, 2000, 100, 2000, 100, 2000, 100, 2000, 100, 2000, 100, 2000]) \\ \bar{\boldsymbol{R}} &= diag([0.005, 0.005, 0.005, 0.05, 0.05, 0.05]) \end{aligned}$$

The unit step response for all 6 degrees of freedom with this controller is shown in fig. 2.3.

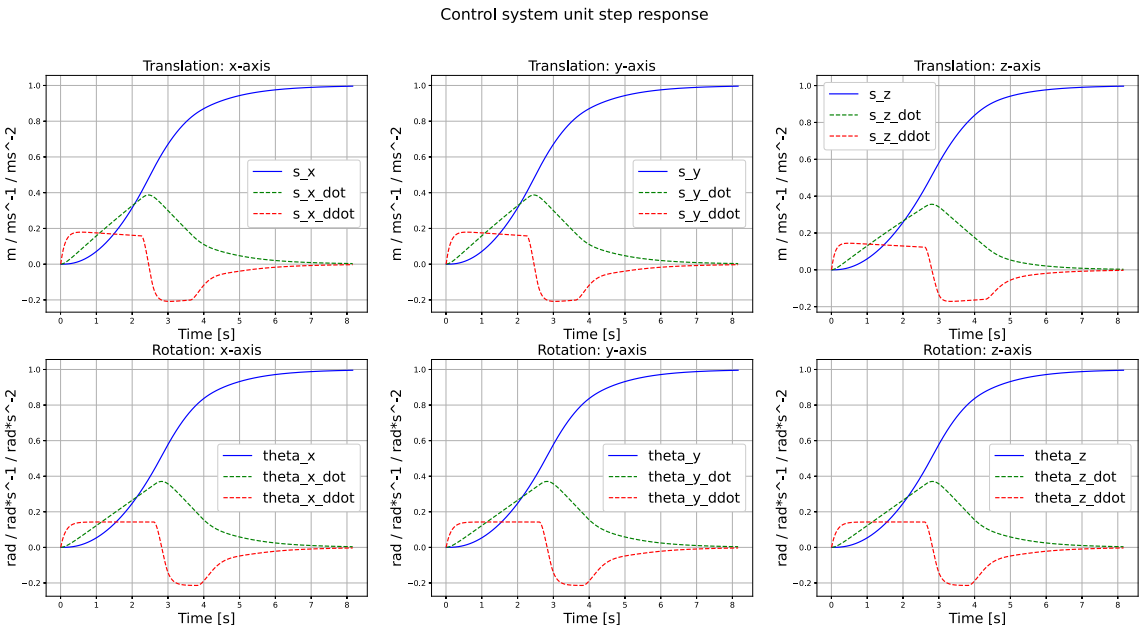


Figure 2.3: Unit step response across all 6 degrees of freedom.

2.5 Waypoint handover

As the control system moves towards a reference, using only the set keyframes as references will result in a "jagged" motion, even when using a fairly high error threshold. A soft transition between two consecutive reference poses, hereafter named waypoints, is desired. In this subsection, a method of smoothly transitioning from one reference to the next is explored.

2.5.1 Hard waypoint handover

The initial waypoint handover was a simple scheme where the next waypoint is activated when the current pose is within a sphere of the current waypoint, where the radius of the sphere is the norm of the error. Mathematically, this is expressed as

$$e_\tau = \|\boldsymbol{\eta}_r - \boldsymbol{\eta}_\tau\|_2 \tag{2.18}$$

where $\|\cdot\|_2$ is the $L2$-norm and $\boldsymbol{\eta}_\tau$ is the pose for the current timestep τ. Let $e_{threshold} \in \mathbb{R}_{\geq 0}$ be the threshold for where $\boldsymbol{\eta}_r$ transitions to the next reference $\boldsymbol{\eta}_{r+1}$. Then the handover is as follows:

$$when\ e_\tau \leq e \quad \Rightarrow \quad \boldsymbol{\eta}_r = \boldsymbol{\eta}_{r+1} \tag{2.19}$$

The resulting behaviour in this hard handover is shown in fig. 2.4, where $e_{threshold}$ is tuned to differentiate between waypoints the vehicle uses as reference for longer trajectories and waypoints for inspection of scene elements (see the figure for a specific example).

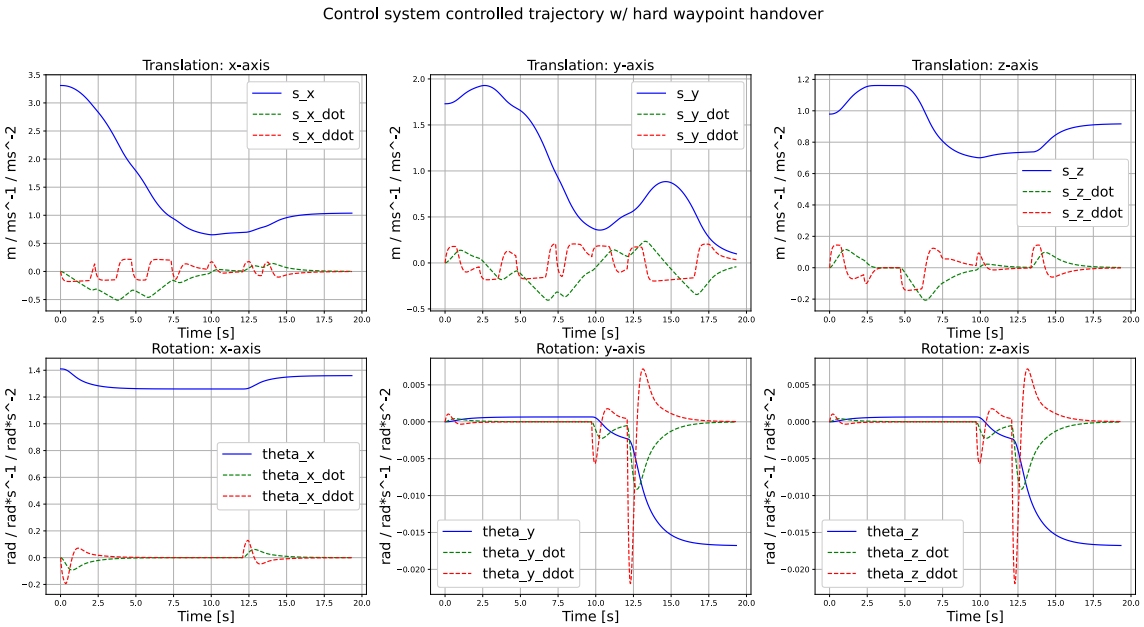


Figure 2.4: States with the control system active. The $e_{threshold}$ is set to the values (in order of appearance) $[0.4, 0.4, 0.4, 0.4, 0.125, 0.4, 0.125, 0.4, 0.05]$ depending on waypoint accuracy requirement.

Hard waypoint handover does have its issues. When the control system is approaching the reference waypoint, the velocity will decrease and the states will approach in a critically damped manner. This is previously shown in fig. 2.3. The vehicle moves fast when the reference is a distance away, and slowing down when it is closer. As a result the velocity is low when the hard waypoint handover takes place, which results in visibly uneven velocities upon playback of the image sequence. By inspecting fig. 2.4 this effect is most visible as step-responses in the $\ddot{s}_x$ curve and the resulting piece-wise linear segments of $\dot{s}_x$.

2.5.2 Soft waypoint handover

When the vehicle is moving through several waypoints, it is desired that this motion is as smooth as possible. Figure 2.5 illustrates this concept for a path consisting of three waypoints; $\boldsymbol{\eta}_{r-1}, \boldsymbol{\eta}_r$ and $\boldsymbol{\eta}_{r+1}$, and the current state $\boldsymbol{\eta}_\tau$. Starting at $\boldsymbol{\eta}_\tau = \boldsymbol{\eta}_{r-1}$, the initial reference is $\boldsymbol{\eta}_r$. However, when moving towards $\boldsymbol{\eta}_r$, the currently used reference should gradually decrease the weights on $\boldsymbol{\eta}_r$ and increase the weights for $\boldsymbol{\eta}_{r+1}$. When $\boldsymbol{\eta}_r$ is reached, the current reference is almost completely weighted to $\boldsymbol{\eta}_{r+1}$. This makes for a softer handover.

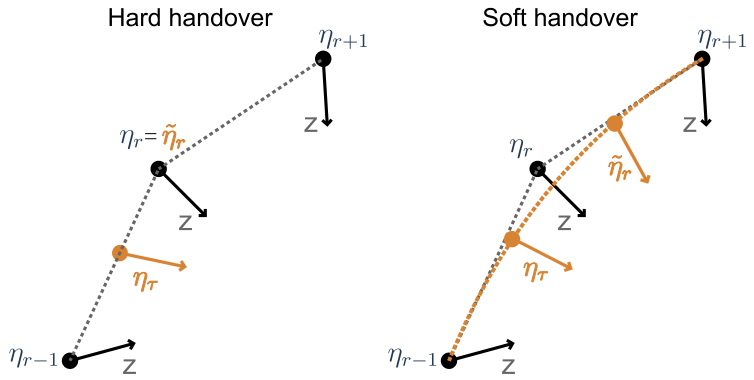


Figure 2.5: Illustration sketch of the waypoint handover function. η_0, η_1, η_2 are waypoints, η_r is the notation for the current waypoint.

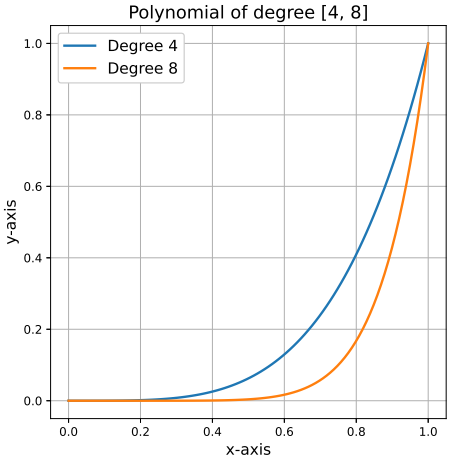


Figure 2.6: The simple n-degree polynomial used for soft transitioning between waypoints. Here a 4 and 8 order polynomial is shown.

The soft waypoint handover is built around a n-degree polynomial $p_n(x_\tau)$:

$$p_n(x_\tau) = x_\tau^n \tag{2.20}$$

where x_τ is a measure of the normalized distance from the current state and the next waypoint. This polynomial with $n = 4$ and $n = 8$ is visualized in fig. 2.6. Finding the normalized distance is done through:

$$x_\tau = \frac{\|\boldsymbol{\eta}_r - \boldsymbol{\eta}_\tau\|_2}{\|\boldsymbol{\eta}_r - \boldsymbol{\eta}_{r-1}\|_2} = \frac{\ell_{\tau,r}}{\ell_{r-1,r}}, \qquad \ell_{\tau,r} \in [0, 1] \tag{2.21}$$

The constraint on $\ell_{\tau,r}$ is applied to ensure that $p_n(x_\tau) \in [0, 1]$ from eq. (2.20). When the waypoint is updated, due to $e_{threshold} > 0$, it is possible that $\ell_{\tau,r} > 1$ depending on the states $\boldsymbol{\eta}_\tau$ when the handover took place. If $\ell_{\tau,r} > 1$, then the waypoint weighting will be outside their valid bounds. As a consequence the waypoint weighting will break down.

Let $\tilde{\boldsymbol{\eta}}_r$ denote the waypoint which is used by the control system. Let $\boldsymbol{\eta}_r$ and $\boldsymbol{\eta}_{r+1}$ be the current and next waypoint from the waypoint vector. The waypoint weighting for finding $\tilde{\boldsymbol{\eta}}_r$ is done as follows:

$$\tilde{\boldsymbol{\eta}}_r = (1 - p_n(x_\tau))\boldsymbol{\eta}_r + p_n(x_\tau)\boldsymbol{\eta}_{r+1} \tag{2.22}$$

From this equation we have that $\lim_{x_\tau \to 0} \tilde{\boldsymbol{\eta}}_r = \boldsymbol{\eta}_r$ and $\lim_{x_\tau \to 1} \tilde{\boldsymbol{\eta}}_r = \boldsymbol{\eta}_{r+1}$, which is the desired behaviour. By using identical parameters as the hard waypoint handover case, setting $n = 4$ and updating $\tilde{\boldsymbol{\eta}}_r$ at each timestep, the behaviour shown in fig. 2.7 is achieved. The amount of undesired behaviour which was found in fig. 2.4 has been significantly reduced, resulting in smoother motion. This becomes more apparent when viewing the playing back the image sequence in real-time.

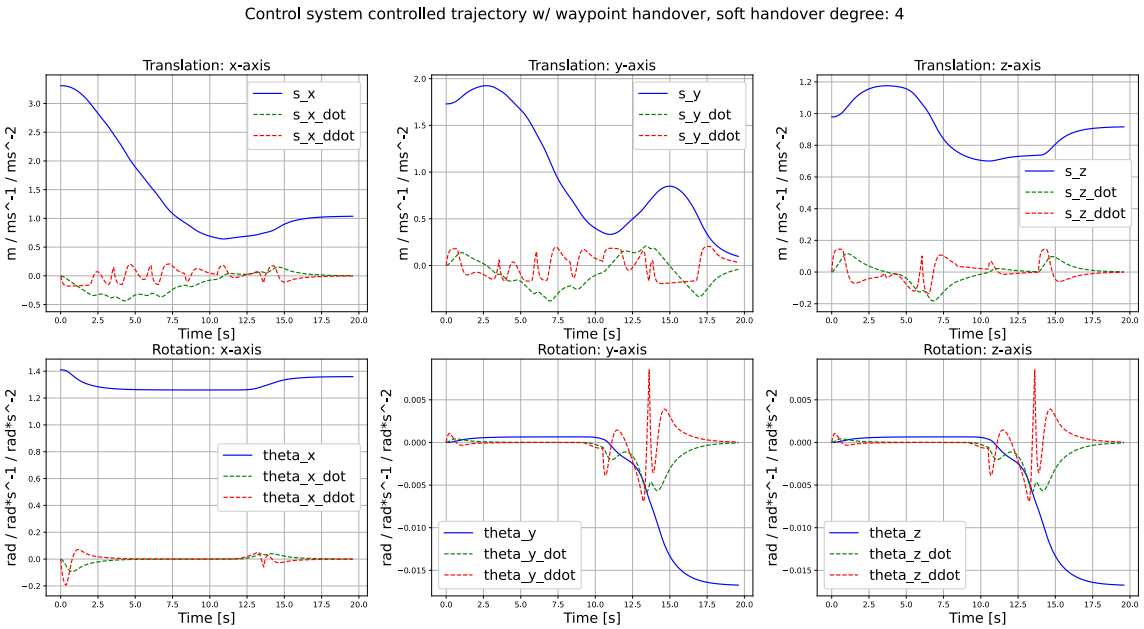


Figure 2.7: States with the control system and an active waypoint handover system with polynomial degree 4.

2.5.3 Additional waypoint handover condition

When creating a waypoint, the $e_{threshold}$ value for waypoint handover is set. If the distance between $\boldsymbol{\eta}_r$ and $\boldsymbol{\eta}_{r+1}$ is large, then the reference trajectory $\tilde{\boldsymbol{\eta}}_r$ can fall outside the error threshold $e_{threshold}$ to trigger a waypoint handover. This is illustrated in fig. 2.8. However, the VCS uses $\tilde{\boldsymbol{\eta}}_r$ as a reference, a reference it cannot follow with exact accuracy due to the dynamics of the system. Hence the actual trajectory can take the shape of any smooth trajectory within the gray area in fig. 2.8.

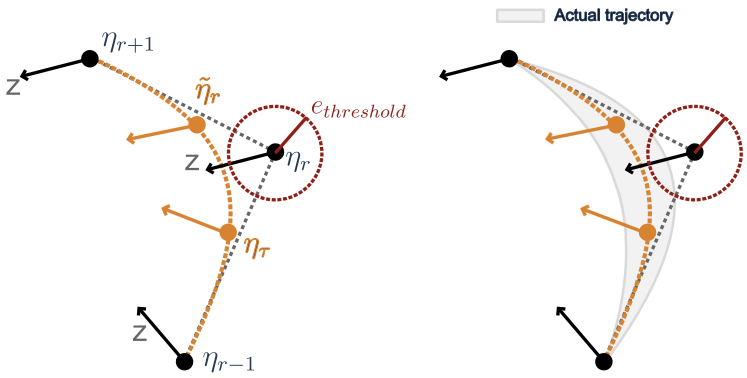


Figure 2.8: A simplified example showing the trajectory with the soft waypoint handover not intersecting with the sphere around $\boldsymbol{\eta}_r$ with radius $e_{threshold}$.

If the actual trajectory overshoots the sphere with radius $e_{threshold}$, the VCS will either reverse its direction until the error falls within this threshold or, in the theoretical worst case, converge to a point outside this sphere. Therefore a second waypoint handover condition is added:

$$\begin{aligned} \|\boldsymbol{\eta}_{r+1} - \boldsymbol{\eta}_\tau\|_2 &\leq \|\boldsymbol{\eta}_{r+1} - \boldsymbol{\eta}_r\|_2 \\ \Updownarrow & \\ \ell_{\tau,r+1} &\leq \ell_{r,r+1} \end{aligned} \tag{2.23}$$

Equation (2.23) adds a sphere around the waypoint $\boldsymbol{\eta}_{r+1}$ with radius $\ell_{r,r+1}$. If the current state $\boldsymbol{\eta}_\tau$ passes into this sphere, then a waypoint handover is triggered. This way, if the VCS overshoots with the trajectory for a given reference, it will automatically trigger the next one. An added benefit of this approach is that the VCS can pass through references at larger velocities without a direct reference for the state velocities. This is handy for parts of a sequence where the VCS is supposed to travel along a path at "travel velocity".

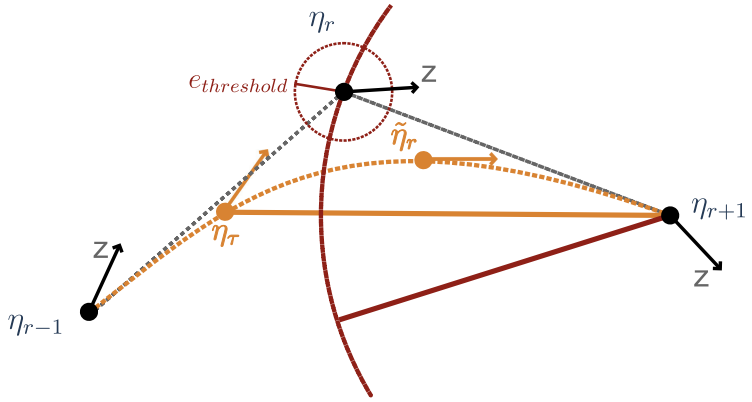


Figure 2.9: Additional distance constraint for waypoint handover, triggered if $\|\boldsymbol{\eta}_{r+1} - \boldsymbol{\eta}_\tau\|_2 \leq \|\boldsymbol{\eta}_{r+1} - \boldsymbol{\eta}_r\|_2$.

2.5.4 Creating waypoints for soft handover

When creating waypoints, there exists in general two types of waypoints; One type which serves as a "guide-rail" for the VCS trajectory with a large acceptable error, and waypoints where the error is desired to be small. Instead of manually tuning $e_{threshold}$ for every waypoint, which is labor intensive for the person creating the sequence, a less labor intensive approach yielding good results is to place two waypoints in close succession. The motion through these waypoints will be more constrained due to the proximity between the two consecutive waypoints.

Remember that the active reference $\tilde{\boldsymbol{\eta}}_r$ is calculated using the previous, current and next waypoints; $[\boldsymbol{\eta}_{r-1}, \boldsymbol{\eta}_r, \boldsymbol{\eta}_{r+1}]$. When the state $\boldsymbol{\eta}$ is approaching $\boldsymbol{\eta}_r$, the reference softly moves towards $\boldsymbol{\eta}_{r+1}$. If $\boldsymbol{\eta}_r$ and $\boldsymbol{\eta}_{r+1}$ are chosen to be close together, the VCS will converge to their general area even with a larger $e_{threshold}$. This is visualized in fig. 2.10.

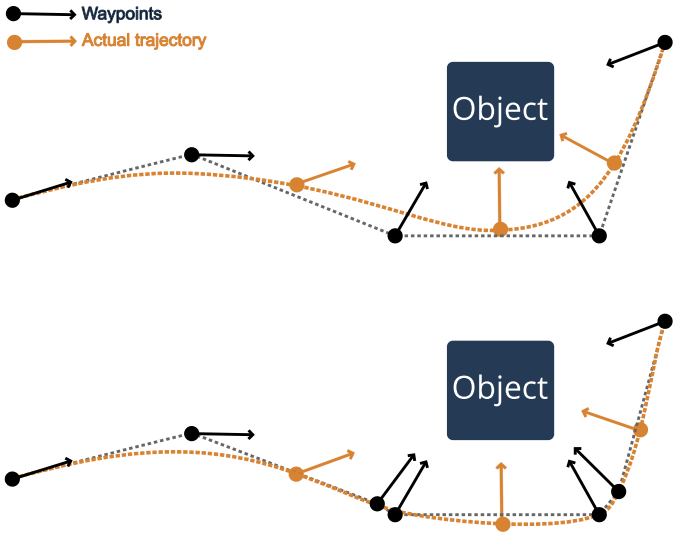


Figure 2.10: Example of the trajectory of the VCS when using close consecutive waypoints to indicate areas of interest where the VCS should closely follow the waypoints.

To illustrate the effect of this approach, five waypoints are set:

$$\boldsymbol{\eta}_0 = \begin{bmatrix} s_{x,r} \\ s_{y,r} \\ s_{z,r} \\ \theta_{x,r} \\ \theta_{y,r} \\ \theta_{z,r} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{\eta}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{\eta}_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{\eta}_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{\eta}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.24}$$

The resulting trajectories with the waypoints *eq.* (2.24) and $e_{threshold} = 0.3$ are shown in fig. 2.11. By comparing the plots, it is apparent that using double waypoints results in a smooth convergence to the reference values. The only exception is for the translation along the z-axis. With double waypoints, $s_z \approx 0.95$, while using single waypoints results in $s_z \approx 0.85$. The better convergence achieved by using double waypoints makes this method good for quickly creating points where close adherence to the waypoint is desired. Likewise, the loose convergence from using single waypoints indicates that single waypoints is sufficient for use in points along a loose trajectory where pose requirements are less strict.

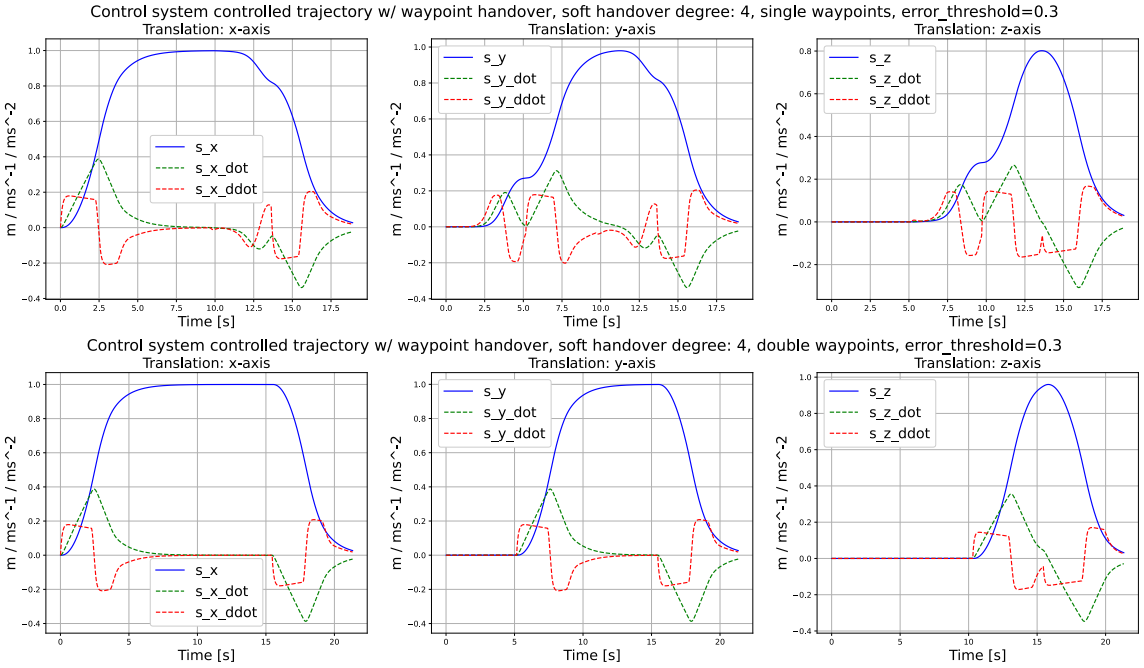


Figure 2.11: Trajectory moving through the waypoints defined in eq. (2.24) with $e_{threshold} = 0.3$. The top figure has single waypoints, whereas the last figure has double waypoints for $\boldsymbol{\eta}_1, \boldsymbol{\eta}_2, \boldsymbol{\eta}_3$ and $\boldsymbol{\eta}_4$.

3 Generating IMU measurements

3.1 Coordinate frame overview

The inertial measurement unit (IMU) measures acceleration relative to its own sensor frame[16]. There are several frames that which are relevant for the IMU measurements[17]. Adapted to the needs of our simulation, these frames are:

World frame $\boldsymbol{w}$: The coordinate frame of the synthetic simulation environment. All data directly exported from the simulation without any transformations uses this frame of reference, and the center of the frame is defined in the center of the virtual environment. In Blender the orientation follows a right-handed frame with the positive z-direction upwards. The simulation frame is assumed to be smaller in scale such that the gravity vector is facing in the negative z-direction and is parallel with the z-axis for the entire frame.

Reference frame $\boldsymbol{r}$: Stationary frame which all motion is relative to. This is initialized as a right-handed coordinate system at the vehicles' position at time 0. Different vehicles may have their own reference frames, as is the case in the EuRoC[3] dataset. This is a deliberate choice to allow for future expansions of the simulation.

Vehicle frame $\boldsymbol{v}$: The vehicle's frame which is moving freely in the reference frame $\boldsymbol{r}$.

Sensor frame $\boldsymbol{s}, \boldsymbol{c}$: This is the frame for a sensor attached to the surface or inside the vehicle. This is the inertial measurement unit (IMU), denoted $\boldsymbol{s}$ for sensor, and the camera, $\boldsymbol{c}$. This frame is fixed with respect to the vehicle frame $\boldsymbol{v}$. While this frame primarily consists of sensors, it is denoted as the body frame to avoid confusions with the simulation frame. If several IMUs or cameras are used, $\boldsymbol{s}_k, \boldsymbol{c}_k$ can be used to address the k^{th} sensor.

Light frame $\boldsymbol{l}_k$: Coordinate frame for a light source, where k indicates the light source number.

3.1.1 Rotation and transformation matrix properties

A rotation matrix $\boldsymbol{R}$ belongs to the Special Orthogonal group $SO(3)$, which is a subspace of the special Euclidean group $SE(3)$. For a rotation matrix $\boldsymbol{R}$ belonging to the $\mathbb{SO}(3)$ group, the following properties apply:

$$\mathbb{SO}(3) = \left\{ \boldsymbol{R} \in \mathbb{R}^{3\times 3} \middle| \boldsymbol{R}\boldsymbol{R}^\top = I,\ \boldsymbol{R}^\top \boldsymbol{R} = I,\ det(\boldsymbol{R}) = 1 \right\}$$

Furthermore, $\mathbb{SO}(3)$ For a transformation matrix $\boldsymbol{T}$, with rotation matrix $\boldsymbol{R}$ and translation $\boldsymbol{t}$, $\mathbb{SE}(3)$ implies the following properties:

$$\mathbb{SE}(3) = \left\{ \boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^\top & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} \middle| \boldsymbol{R} \in \mathbb{SO}(3), \boldsymbol{t} \in \mathbb{R}^3 \right\}$$

3.1.2 Coordinate transformation introduction

A point $\boldsymbol{p}^a = [x, y, t]^\top$ in an arbitrary frame a can be rotated using a rotation matrix $\boldsymbol{R}$ and a translation vector $\boldsymbol{t}$, which are shown in eq. (3.1).

$$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \in SO(3), \qquad \boldsymbol{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{3.1}$$

The rotation matrix $\boldsymbol{R}$ can represent a rotation form frame b to a as $\boldsymbol{R}_{ab}$. The corresponding translation is denoted $\boldsymbol{t}^a_{ab}$. Here the subscript ab denotes the translation from frame b to a, and the superscript a denotes that the translation vector is given with respect to frame a. Imagine a

point $\boldsymbol{p}^b$ is given in frame b, and it is to be transformed into frame a. In order to transform $\boldsymbol{p}^b$ into $\boldsymbol{p}^a$, the following equation is used:

$$\boldsymbol{p}_1^a = \boldsymbol{R}_{ab}\boldsymbol{p}_1^b + \boldsymbol{t}_{ab}^a \tag{3.2}$$

An arbitrary point $\boldsymbol{p} \in \mathbb{R}^3$ is converted to homogeneous coordinates by following eq. (3.3).

$$\tilde{\boldsymbol{p}} = [\boldsymbol{p}, 1]^\top = [x, y, z, 1]^\top \tag{3.3}$$

By using the homogeneous points from eq. (3.3), it is possible to combine eq. (3.2) into a single expression using a transformation matrix $\boldsymbol{T}_{ab}$ belonging to the special Euclidean group $\mathbb{SE}(3)$. This transformation is shown in eq. (3.4).

$$\tilde{\boldsymbol{p}}^a = \boldsymbol{T}_{ab}\tilde{\boldsymbol{p}}^b = \begin{bmatrix} \boldsymbol{R}_{ab} & \boldsymbol{t}_{ab}^a \\ \boldsymbol{0}^\top & 1 \end{bmatrix} \tilde{\boldsymbol{p}}^b \tag{3.4}$$

However, if the point $\boldsymbol{p}^a$ is given it might be desired to find the point expressed in frame b as $\boldsymbol{p}^b$. This is done by inverting the transformation in eq. (3.4), resulting in eq. (3.5).

$$\tilde{\boldsymbol{p}}_1^b = \boldsymbol{T}_{ba}\tilde{\boldsymbol{p}}_1^a = (\boldsymbol{T}_{ab})^{-1}\tilde{\boldsymbol{p}}_1^a = \begin{bmatrix} \boldsymbol{R}_{ab} & \boldsymbol{t}_{ab}^a \\ \boldsymbol{0}^\top & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{R}_{ab}^\top & -\boldsymbol{R}_{ab}^\top \boldsymbol{t}_{ab}^a \\ \boldsymbol{0}^\top & 1 \end{bmatrix} \tag{3.5}$$

These coordinate transformations are not only valid for points $\tilde{\boldsymbol{p}}$, but also poses $\boldsymbol{P}$ such that:

$$\boldsymbol{P}^a = \boldsymbol{T}_{ab}\boldsymbol{P}^b, \qquad \boldsymbol{P} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^\top & 1 \end{bmatrix} \tag{3.6}$$

3.2 Coordinate transformations

3.2.1 A comment on transformations in this section

Near the end of May, my co-supervisor Mauhing Yip found a discrepancy in the notation used in the transformations between the vehicle- and sensor frames, and the actual transformation which were made in section 3.2.4. This section uses an implementation yielding seemingly correct results when the final plots from the IMU is viewed. However this is no guarantee that the output or the derivations are completely correct despite that the plots at the end of this subsection make sense when the actual motion is taken into consideration. This discrepancy is further discussed in section 6.2 with examples.

3.2.2 Pre-processing of imported simulation data

The pre-processing step aims to convert the data exported directly from Blender into a format and coordinate frames which can be used by the code which generates the IMU measurements. The data from the VCS is exported as a large 2D array contained in a .csv file. The first step splits the VCS data into individual vectors for each state time derivative:

$$\boldsymbol{s} = \begin{bmatrix} \boldsymbol{s}_x \\ \boldsymbol{s}_y \\ \boldsymbol{s}_z \end{bmatrix} \quad \dot{\boldsymbol{s}} = \begin{bmatrix} \dot{\boldsymbol{s}}_x \\ \dot{\boldsymbol{s}}_y \\ \dot{\boldsymbol{s}}_z \end{bmatrix} \quad \ddot{\boldsymbol{s}} = \begin{bmatrix} \ddot{\boldsymbol{s}}_x \\ \ddot{\boldsymbol{s}}_y \\ \ddot{\boldsymbol{s}}_z \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_x \\ \boldsymbol{\theta}_y \\ \boldsymbol{\theta}_z \end{bmatrix} \quad \dot{\boldsymbol{\theta}} = \begin{bmatrix} \dot{\boldsymbol{\theta}}_x \\ \dot{\boldsymbol{\theta}}_y \\ \dot{\boldsymbol{\theta}}_z \end{bmatrix} \quad \ddot{\boldsymbol{\theta}} = \begin{bmatrix} \ddot{\boldsymbol{\theta}}_x \\ \ddot{\boldsymbol{\theta}}_y \\ \ddot{\boldsymbol{\theta}}_z \end{bmatrix}$$

These vectors are then combined into two 3D arrays for easier processing, $\boldsymbol{S}$ containing $\boldsymbol{s}, \dot{\boldsymbol{s}}$ and $\ddot{\boldsymbol{s}}$ and $\boldsymbol{\Theta}$ containing $\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}$ and $\ddot{\boldsymbol{\theta}}$. then the structure is as follows:

$$\boldsymbol{S} = \begin{bmatrix} \boldsymbol{s} \\ \dot{\boldsymbol{s}} \\ \ddot{\boldsymbol{s}} \end{bmatrix}, \qquad \boldsymbol{\Theta} = \begin{bmatrix} \boldsymbol{\theta} \\ \dot{\boldsymbol{\theta}} \\ \ddot{\boldsymbol{\theta}} \end{bmatrix}$$

Using this structure, $\boldsymbol{s}$ can be accessed in code via the command $\boldsymbol{S}[0, :, :]$ for the current implementation with Python and NumPy.

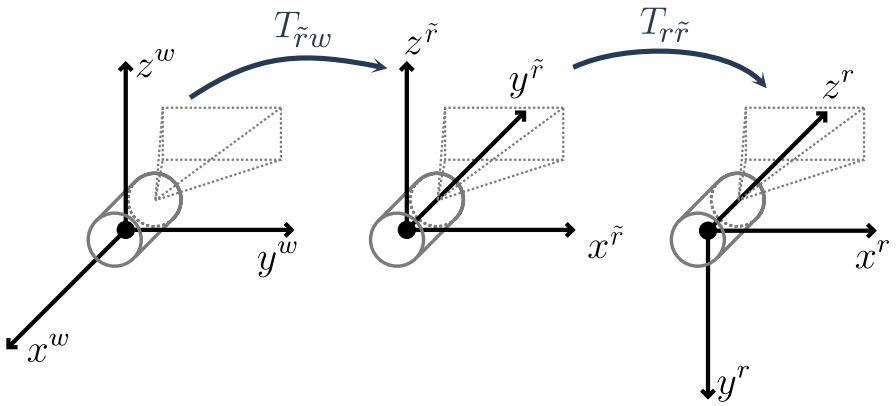


Figure 3.1: Preprocessing of IMU data illustrated for the first sample of the sequence, as $\boldsymbol{T}_{r\tilde{r}}$ and $\boldsymbol{T}_{\tilde{r}w}$ are found through the pose data for the first sample. The data is transformed from the world coordinate frame $\boldsymbol{w}$ to the reference frame $\boldsymbol{r}$. The orientation of the vehicle in $\boldsymbol{w}$ is arbitrary. The reference frame $\boldsymbol{r}$ coincides with the vehicle frame $\boldsymbol{v}$ for the first sample.

3.2.3 Converting VCS data from world to reference frames

The information used to generate the IMU measurements is extracted from the simulation in the world frame w. This data is $\boldsymbol{S}$ and $\boldsymbol{\Theta}$ from section 3.2.2.

The measurements data should be transformed such that the initial measurement is at the origin of a reference frame for the vehicle the sensor is attached. The transformation from the world frame $\boldsymbol{w}$ to the reference frame $\boldsymbol{r}$ is named $\boldsymbol{T}_{rw}$. Furthermore, $\boldsymbol{T}_{rw}$ is the product of two transforms, one that transforms the raw data in the world frame to a temporary reference frame $\tilde{\boldsymbol{r}}$, and finally a transform which transforms from $\tilde{\boldsymbol{r}}$ into the desired reference frame $\boldsymbol{r}$. These transformations are called $\boldsymbol{T}_{\tilde{r}w}$ and $\boldsymbol{T}_{r\tilde{r}}$ respectively.

$\boldsymbol{T}_{\tilde{r}w}$ is generating the pose matrix for the first timestep of the sequence, then inverting this pose matrix. Let $\boldsymbol{T}_w(t=0)$ denote the pose matrix at $t=0$. Then:

$$\boldsymbol{T}_{\tilde{r}w} = [\boldsymbol{T}_w(t=0)]^{-1} \tag{3.7}$$

Then $\boldsymbol{T}_{r\tilde{r}}$ must be determined. In this implementation, the reference frame is equal to the vehicle frame at time $t=0$. Therefore $\boldsymbol{T}_{r\tilde{r}}$ is a rotation which realigns the axes such that the z-axis points forward, the x-axis points to the right and the y-axis points downward. This is illustrated in fig. 3.1.

When $\boldsymbol{T}_{\tilde{r}w}$ is applied, the coordinate system of the vehicle is oriented as shown in fig. 3.1. In order to convert the $\tilde{\boldsymbol{r}}$ frame into the reference frame $\boldsymbol{r}$, the following passive transformation of coordinate systems is applied:

$$\boldsymbol{T}_{r\tilde{r}} = \begin{bmatrix} \boldsymbol{R}_x(\frac{\pi}{2}) & \boldsymbol{0}_{3,1} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{R}_x(\frac{\pi}{2})^{-1} & \boldsymbol{0}_{3,1} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_x(-\frac{\pi}{2}) & \boldsymbol{0}_{3,1} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} \tag{3.8}$$

The transformation from the world frame to the reference frame is then found by multiplying eq. (3.8) and eq. (3.7):

$$\boldsymbol{T}_{rw} = \boldsymbol{T}_{r\tilde{r}}\boldsymbol{T}_{\tilde{r}w} \tag{3.9}$$

When converting the data to the reference frame, it is important to distinguish between transformation of translation $\boldsymbol{s}$, and the transformation of rotation as well as rotational and translational velocity and acceleration. The former must be calculated on the normal form as $\boldsymbol{T}_{rw}$ may contain a non-zero translation term. The latter data should only be rotated into the reference frame using $\boldsymbol{R}_{rw}$.

$$\begin{aligned} \boldsymbol{s}^r &= \boldsymbol{T}_{rw}\begin{bmatrix}\boldsymbol{s}^w \\ \boldsymbol{1}\end{bmatrix} & \dot{\boldsymbol{s}}^r &= \boldsymbol{R}_{rw}\dot{\boldsymbol{s}}^w & \ddot{\boldsymbol{s}}^r &= \boldsymbol{R}_{rw}\ddot{\boldsymbol{s}}^w \\ \boldsymbol{\Theta}^r &= \boldsymbol{R}_{rw}\boldsymbol{\Theta}^w & \dot{\boldsymbol{\Theta}}^r &= \boldsymbol{R}_{rw}\dot{\boldsymbol{\Theta}}^w & \ddot{\boldsymbol{\Theta}}^r &= \boldsymbol{R}_{rw}\ddot{\boldsymbol{\Theta}}^w \end{aligned} \tag{3.10}$$

3.2.4 Generating sensor poses from vehicle poses

To reduce the amount of exported data, the pose of the sensor (i.e. IMU or camera) can be derived with respect to the exported vehicle frame. This way, only one set of exported poses are needed for generating one or more sets of IMU data and the poses for the camera which is used in 3D reconstruction or SLAM. In this IMU implementation, the transformation used assumes that the vehicle frame has its z-direction pointing in the positive forward direction, the positive x-direction points out of the vehicles' right hand side, and finally the positive y-direction points downwards, illustrated in fig. 3.2. This is chosen to comply with the standard set by the KITTI[1] dataset and the Robot Operating System (ROS)[18].

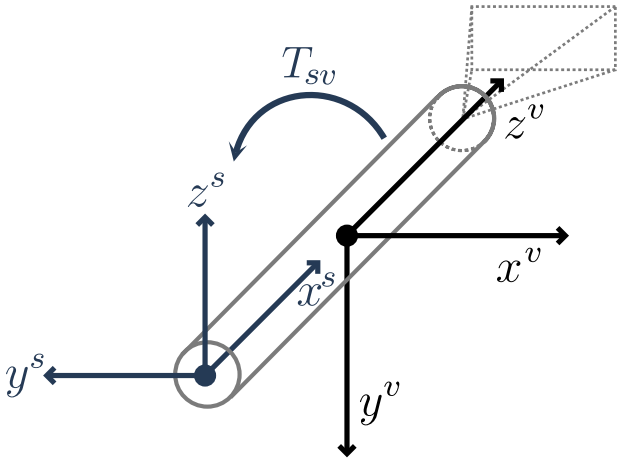


Figure 3.2: Vehicle frame $\boldsymbol{v}$ to sensor frame $\boldsymbol{s}$ transform. IMU sensor is rotated to point "upwards" in the vehicle frame.

To create the sensor coordinate frame $\boldsymbol{s}$ relative to the vehicle frame $\boldsymbol{v}$, expressed in the sensor frame $\boldsymbol{s}$, the transformation matrix $\boldsymbol{T}_{sv}$ from the vehicle frame to the sensor frame is introduced:

$$\boldsymbol{T}_{sv} = \begin{bmatrix}\boldsymbol{R}_{sv} & \boldsymbol{t}_{sv}^v \\ \boldsymbol{0}^\top & 1\end{bmatrix} \tag{3.11}$$

where $\boldsymbol{R}_{sv}$ and $\boldsymbol{t}_{sv}^v$ is the rotation and translation respectively from the source frame $\boldsymbol{v}$ to the destination frame $\boldsymbol{s}$. By looking at the previous equation, the rotation can be expressed as a rotation of 90 deg counter clockwise around $\boldsymbol{x}^v$ followed by a 90 deg counter clockwise rotation around $\boldsymbol{z}^v$. However, the coordinate transformation of the IMU data is passive. We do not wish

to change the IMU data with respect to the world frame, only the frame of which the IMU data is expressed within, the sensor frame $\boldsymbol{s}$. Therefore the described rotation is made in the fixed sensor frame $\boldsymbol{s}$. First, a rotation of $-\frac{\pi}{2}[rad]$ in the right handed coordinate frame is applied around the $\boldsymbol{x}^s$-axis, then a rotation of $-\frac{\pi}{2}[rad]$ around the $\boldsymbol{z}^s$-axis using the right-hand rule follows. The mathematical expression for this rotation is shown in eq. (3.12) and visualized step-by-step in fig. 3.3:

$$
\begin{aligned}
\boldsymbol{R}_{sv} &= \boldsymbol{R}_z(-\frac{\pi}{2})\boldsymbol{R}_x(-\frac{\pi}{2}) \\
&= \begin{bmatrix} cos(-\frac{\pi}{2}) & -sin(-\frac{\pi}{2}) & 0 \\ sin(-\frac{\pi}{2}) & cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(-\frac{\pi}{2}) & -sin(-\frac{\pi}{2}) \\ 0 & sin(-\frac{\pi}{2}) & cos(-\frac{\pi}{2}) \end{bmatrix} \\
&= \begin{bmatrix} cos(\frac{\pi}{2}) & sin(\frac{\pi}{2}) & 0 \\ -sin(\frac{\pi}{2}) & cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\frac{\pi}{2}) & sin(\frac{\pi}{2}) \\ 0 & -sin(\frac{\pi}{2}) & cos(\frac{\pi}{2}) \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}
\end{aligned}
\tag{3.12}
$$

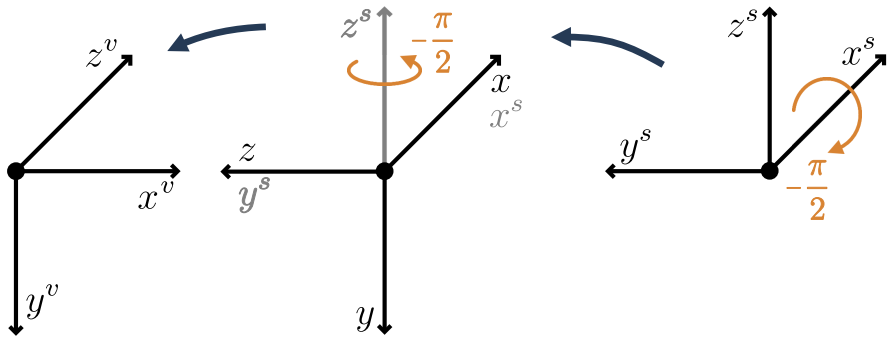


Figure 3.3: The coordinate system rotations from eq. (3.12) between the vehicle frame v and the sensor frame s. The arrows indicate the positive CCW rotation around each axis. Negative angle values is then rotated in the opposite direction.

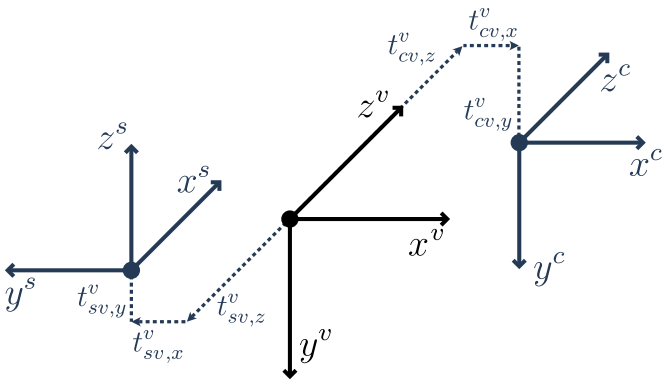


Figure 3.4: Translation of the sensor and camera frames, denoted $\boldsymbol{s}$ and $\boldsymbol{c}$ respectively, with respect to the vehicle frame, $\boldsymbol{v}$. Rotations are performed relative to the fixed frame $\boldsymbol{s}$.

The sensor is offset from the center of the vehicle by the vector eq. (3.13), where the offset is given with respect to the Origin of the vehicle frame $\boldsymbol{v}$ as shown in fig. 3.4.

$$\boldsymbol{t}_{sv}^{v} = \begin{bmatrix} t_{sv,x}^{v} & t_{sv,y}^{v} & t_{sv,z}^{v} \end{bmatrix}^{\top} \tag{3.13}$$

Combining eq. (3.12) and eq. (3.13) it is possible to write the transformation from $\boldsymbol{v}$ to $\boldsymbol{s}$, where the rotation is extrinsic and expressed in frame $\boldsymbol{s}$ and the translation from $\boldsymbol{v}$ to $\boldsymbol{s}$, as $\boldsymbol{T}_{sv}$:

$$\boldsymbol{T}_{sv} = \begin{bmatrix} \boldsymbol{R}_{sv} & \boldsymbol{t}_{sv}^{v} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & t_{sv,x}^{v} \\ -1 & 0 & 0 & t_{sv,y}^{v} \\ 0 & -1 & 0 & t_{sv,z}^{v} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

It is now possible to express the pose of the sensor $\boldsymbol{P}^{s}$ in terms of the vehicle pose $\boldsymbol{P}^{v}$ and the transformation obtained in eq. (3.14):

$$\boldsymbol{P}^{s} = \boldsymbol{T}_{sv}\boldsymbol{P}^{v} \tag{3.15}$$

3.2.5 Generating camera poses from vehicle poses

For the camera frame $\boldsymbol{c}$, the same convention as OpenCV[19] is used with a positive $\boldsymbol{z}$-direction out of the camera lens. Assuming a horizontal orientation relative to the ground, the positive $\boldsymbol{x}$-axis is pointing out to the right of the camera and the positive $\boldsymbol{y}$-axis points downward into the ground. Following this convention, the orientation of the front-facing camera is the same as the vehicle frame for a monocular camera. Hence it is only translated with respect to the vehicle frame as shown in fig. 3.4. Using this information, an expression for $\boldsymbol{T}_{cv}$ is found:

$$\boldsymbol{T}_{cv} = \begin{bmatrix} \boldsymbol{R}_{cv} & \boldsymbol{t}_{cv}^{v} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_{cv,x}^{v} \\ 0 & 1 & 0 & t_{cv,y}^{v} \\ 0 & 0 & 1 & t_{cv,z}^{v} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

3.3 State interpolation for measurements

The simulation operates with a minimum of three different frequencies, as shown in table 3.1. To ensure that all the states for the various measurements and ground truth data can be exported, f_{sim} must be chosen such that it is divisible by the other frequencies used, i.e. f_s and f_c in table 3.1.

Table 3.1: The three different type of frequencies used in the creation of sequences and the corresponding ground truth data.

$f_{sim} = 1/\Delta t_{sim}$	Internal simulation frequency
$f_s = 1/\Delta t_s$	IMU sensor frequency
$f_c = 1/\Delta t_c$	Camera frequency

The requirement for divisibility does significantly restrict the available choices for f_{sim}. In order to increase the flexibility of our simulation framework, it was proposed to use an interpolation scheme for both the rotation and translation. Sections 3.3.1 and 3.3.2 explains two different implementations of an interpolation scheme for the translation, velocity and acceleration. However, rotational interpolation turned out to be non-trivial and section 3.3.3 goes into detail as to why. Section 3.3.5 contains math which for reconstructing a the ZYX Euler angles from a rotation matrix, and is included as it can be useful in the future.

3.3.1 Translational interpolation, polynomial curvefitting

It was proposed by my co-supervisor Rudolf Mester to use a second-order curve-fitted polynomial on the available data to derive first- and second-order derivatives, where he also kindly provided the following derivation.

The offline estimator uses, for a velocity and acceleration estimation at time t, samples at $t-1$, t and $t+1$. Let $\boldsymbol{x} = [x_{-1}, x_t, x_{t+1}]^\top$ denote the sample at given times. Likewise $\boldsymbol{y} = [y_{t-1}, y_t, y_{t+1}]^\top$ denote the translation or Euler rotation exported from the simulation environment for these measurements.

The second order polynomial for a single sample $x_\tau, \tau \in [t-1, t, t+1]$, may be written as

$$y(x_\tau) = y_\tau = a + bx_\tau + cx_\tau^2 \tag{3.17}$$

where a, b and c are coefficients for the polynomial. Using this equation, it is possible to rewrite set of equations for the three measurements as the system of equations in eq. (3.18).

$$\boldsymbol{y} = \begin{bmatrix} y_{t-1} \\ y_t \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & x_{t-1} & x_{t-1}^2 \\ 1 & x_t & x_t^2 \\ 1 & x_{t+1} & x_{t+1}^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \boldsymbol{H\eta} \tag{3.18}$$

Reordering this equation in order to find $\boldsymbol{\eta}$:

$$\boldsymbol{H\eta} = \boldsymbol{y} \tag{3.19}$$

$$\boldsymbol{H}^{-1}\boldsymbol{H\eta} = \boldsymbol{H}^{-1}\boldsymbol{y} \tag{3.20}$$

$$\boldsymbol{\eta} = \boldsymbol{H}^{-1}\boldsymbol{y} \tag{3.21}$$

This requires the inverse of the $\boldsymbol{H}$-matrix to be known. Its inverse is found through the use of *SymPy*[20], resulting in eq. (3.22).

$$\boldsymbol{H}^{-1} = \frac{1}{x_{t-1}^2 - x_{t-1}x_t - x_{t-1}x_{t+1} + x_t x_{t+1}} \begin{bmatrix} x_t x_{t+1} & -x_{t-1}x_{t+1} & x_{t-1}x_t \\ -(x_t + x_{t+1}) & x_{t-1} + x_{t+1} & -(x_{t-1} + x_t) \\ 1 & -1 & 1 \end{bmatrix} \tag{3.22}$$

By using eqs. (3.19) and (3.22), the second order polynomial coefficients ***eta*** are found. It is then possible to analytically differentiate the second order polynomial function from eq. (3.17) to find an expression for its single- and double derivatives:

$$\dot{y}_\tau = b\dot{x}_\tau + 2cx_\tau\dot{x}_\tau = (b + 2cx_\tau)\Delta t_{x(\tau)}, \qquad \ddot{y}_\tau = 2c\dot{x}_\tau = 2c\Delta t_{x(\tau)} \tag{3.23}$$

Please note that $\dot{x}_\tau = \Delta t_{x(\tau)}$, where $\Delta t_{x(\tau)}$ is the time difference between the previous and current frame. For symmetric interpolation this is the time difference between the first and second sample, in a sequence of three samples.

During testing this method provided erroneous results, most likely due to an implementation error. In the interest of time, the NumPy implementation utilizing the same principle in section 3.3.2 was investigated instead.

3.3.2 Translational interpolation, polynomial curvefitting with NumPy

Due to the previously mentioned issues, curvefitting via the NumPy library is used. The curvefitting function used is:

$$numpy.polynomial.Polynomial.fit(x, y, degree, window)$$

x denotes the array for the x-axis, namely the timesteps for the three samples in question. y denotes the measurements at these timesteps, and *degree* is the degree of the polynomial which is fitted to the points. *window* is the active window for the polynomial curve-fitting. This has to be specified such that it uses a window that coincides with the current sample numbers and the surrounding samples.

3.3.3 Rotational interpolation

Interpolation of the rotation and its derivatives proved to be non-trivial. A rotation matrix must adhere to $SE(3)$, which means that it must adhere to the constraints presented in section 3.3.4. The rotation matrix can be interpolated using Lie-theory, where the velocity between the points of interpolation is assumed to be constant[21]. Herein lies the first issue, a constant velocity implies zero acceleration, which will result in invalid IMU data.

The IMU is directly dependent on angular velocity and acceleration, which requires that the angular velocity and acceleration are both interpolated. This means that the angular acceleration must be interpolated, and from this the angular velocity is derived. Finally, the interpolated rotation matrix is found, which must adhere to $SE(3)$.

Due to the added complexity of any rotational interpolation which follows the above requirements, it was decided that adapting the internal simulation timestep to directly export the states at the desired frequencies was the most efficient way to go.

While state interpolation is not used in the latest version of the IMU framework, sections 3.3.4 and 3.3.5 are included in case it is needed for future iterations of the framework that builds on this work.

3.3.4 Verifying the rotation matrix

Before converting a 3x3 matrix to Euler angles, it has to be verified as a rotation matrix. This means that eq. (3.24) has to be fulfilled for a rotation matrix $\boldsymbol{R}$.

$$\boldsymbol{R}\boldsymbol{R}^\top = I, \quad \boldsymbol{R}^\top\boldsymbol{R} = I, \quad det(\boldsymbol{R}) = 1 \tag{3.24}$$

As the two first expressions in eq. (3.24) are equal, they can be expressed as one of the equations. As such the requirements can be rewritten as:

$$\boldsymbol{R}\boldsymbol{R}^\top - I = \boldsymbol{0}, \quad det(\boldsymbol{R}) - 1 = 0$$

However, due to numerical and rounding errors a small value ϵ is introduced as a threshold. If a value is above this threshold it is treated as non-zero, and if it is below it is treated as zero. This also requires a scalar representation of the error between the matrices. To find the error, the $L2$-norm is used. Hence the code implementation is as follows:

$$||\boldsymbol{R}\boldsymbol{R}^\top - I||_2^2 < \epsilon, \quad det(\boldsymbol{R}) - 1 < \epsilon \tag{3.25}$$

3.3.5 Reconstructing the Euler angles from a rotation matrix

To convert a 3x3 ZYX rotation matrix to Euler angles, [22] is used. However, this article operates with a XYZ rotation scheme. Blender uses a ZYX convention, and the derivations of this article is rewritten to follow this convention.

Let $[\theta_1, \theta_2, \theta_3]$ denote the angle for the rotation around the $x-$, $y-$ and $z-$axes accordingly. In

addition, let c_k and s_k denote $cos(\theta_k)$ and $\sin(\theta_k)$ where $k \in [1, 2, 3]$.

$$\begin{aligned}
\boldsymbol{R}_{zyx}(\boldsymbol{\theta}) &= \boldsymbol{R}_z(\theta_3)\boldsymbol{R}_y(\theta_2)\boldsymbol{R}_x(\theta_1), \\
&= \begin{bmatrix} c_3 & -s_3 & 0 \\ s_3 & c_3 & 0 \\ 0 & & 1 \end{bmatrix} \begin{bmatrix} c_2 & 0 & s_2 \\ 0 & 1 & 0 \\ -s_2 & 0 & c_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1 & c_1 \end{bmatrix} \\
&= \begin{bmatrix} c_3c_2 & -s_3 & c_3s_2 \\ s_3c_2 & c_3 & s_3s_2 \\ -s_2 & 0 & c_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1 & c_1 \end{bmatrix} \\
&= \begin{bmatrix} c_3c_2 & -s_3c_1 + c_3s_2s_1 & c_3s_2c_1 + s_3s_1 \\ s_3c_2 & c_3c_1 + s_3s_2s_1 & s_3s_2c_1 - c_3s_1 \\ -s_2 & c_2s_1 & c_2c_1 \end{bmatrix}
\end{aligned} \tag{3.26}$$

Following the notation of [22] (with a slight index change), given a matrix $\boldsymbol{M}(\boldsymbol{\theta})$:

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{3.27}$$

The $\boldsymbol{M}(\boldsymbol{\theta})$-matrix can be viewed as a given pose of the object, where each index contains a value that is dependent upon the rotations around the x-, y- and z-axes $\boldsymbol{\theta}$. Therefore the problem can be formulated as; for a $\theta_k, k \in [1, 2, 3]$, find θ_k such that $R_{zyx}(\boldsymbol{\theta}) = \boldsymbol{M}(\boldsymbol{\theta})$.

Let $\boldsymbol{M'}$ be another rotation that is the product of the rotation around the first two axes, that is the z- and y-axes, and the $\boldsymbol{M}$ matrix.

$$\begin{aligned}
\boldsymbol{M'} = [\boldsymbol{R}_z(\theta_3)\boldsymbol{R}_y(\theta_2)]^\top \boldsymbol{M} &= \begin{bmatrix} c_3c_2 & -s_3 & c_3s_2 \\ S_3c_2 & c_3 & s_3s_2 \\ -s_2 & 0 & c_2 \end{bmatrix}^\top \boldsymbol{M} \\
&= \begin{bmatrix} c_3c_2 & s_3c_2 & -s_2 \\ -s_3 & c_3 & 0 \\ c_3s_2 & s_3s_2 & c_2 \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}
\end{aligned} \tag{3.28}$$

If the product is a pure rotation around the x-axis, then $\boldsymbol{M'}$ has to have the form:

$$\boldsymbol{M'} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & -s_1 \\ 0 & s_1 & c_1 \end{bmatrix} \tag{3.29}$$

This is only dependent on θ_1 which is the rotation around the x-axis. As we need to find a unique expression for θ_1 we need to find one expression for c_1 and s_1. By looking at the middle row of eq. (3.29), and multiplying out the middle row in the left matrix in eq. (3.28), it is found that:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & -s_1 \\ 0 & s_1 & c_1 \end{bmatrix} = \begin{bmatrix} * & * & c_3c_2m_{13} + s_3c_2m_{23} - s_2m_{33} \\ -s_3m_{11} + c_3m_{21} & -s_3m_{12} + c_3m_{22} & -s_3m_{13} + c_3m_{23} \\ * & * & * \end{bmatrix} \tag{3.30}$$

This yields two equations:

$$c_1 = -s_3m_{12} + c_3m_{22}, \qquad -s_1 = -s_3m_{13} + c_3m_{23} \tag{3.31}$$

By using the trigonometric identity $tan(\theta) = \frac{sin(\theta)}{cos(\theta)}$ and eq. (3.31):

$$\theta_1 = atan2(\frac{s_1}{c_1}) = atan2(\frac{s_3 m_{13} - c_3 m_{23}}{-s_3 m_{12} + c_3 m_{22}}) \tag{3.32}$$

This requires θ_3, and therefore this variable has to be uniquely determined.

Looking back at eq. (3.30), extract equation at index $(1, 0)$ in both matrices. This yields the equation:

$$\begin{aligned} 0 &= -s_3 m_{11} + c_3 m_{21} \\ s_3 m_{11} &= c_3 m_{21} \\ \frac{s_3}{c_3} &= \frac{m_{11}}{m_{21}} \\ tan(\theta_3) &= \frac{m_{11}}{m_{21}} \\ \theta_3 &= atan2(\frac{m_{11}}{m_{21}}) \end{aligned} \tag{3.33}$$

From eq. (3.30), extract the equation at index $(0, 2)$, yielding the equation:

$$\begin{aligned} 0 &= c_3 c_2 m_{13} + s_3 c_2 m_{23} - s_2 m_{33} \\ s_2 m_{33} &= c_2(c_3 m_{13} + s_3 m_{23}) \\ \frac{s_2}{c_2} &= \frac{c_3 m_{13} + s_3 m_{23}}{m_{33}} \\ \theta_2 &= atan2(\frac{c_3 m_{13} + s_3 m_{23}}{m_{33}}) \end{aligned} \tag{3.34}$$

From eqs. (3.32) to (3.34) it is possible to reconstruct the Euler angles:

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} = \begin{bmatrix} atan2(\frac{s_3 m_{13} - c_3 m_{23}}{-s_3 m_{12} + c_3 m_{22}}) \\ atan2(\frac{c_3 m_{13} + s_3 m_{23}}{m_{33}}) \\ atan2(\frac{m_{11}}{m_{21}}) \end{bmatrix} \tag{3.35}$$

3.4 Derivation of the gyroscope

It is assumed that the provided pose sequences are converted to the reference frame, as described in section 3.2.3. The information from which the gyroscope data is generated is located in the reference frame $\boldsymbol{r}$, whereas the gyroscope itself produces data in the IMU sensor frame $\boldsymbol{s}$. Hence the gyroscope produces a measurement of the angular velocity of the vehicle frame $\boldsymbol{v}$ with respect to the reference frame $\boldsymbol{r}$ expressed in the IMU sensor frame $\boldsymbol{s}$. In mathematical terms it is expressed as $\boldsymbol{\omega}^s_{vr}$. As the IMU sensor is fixed compared to the vehicle, the transform between the angular velocity of the vehicle frame $\boldsymbol{v}$ and the IMU sensor frame $\boldsymbol{s}$ can be expressed as:

$$\boldsymbol{\omega}^s_{vr} = \boldsymbol{R}_{sv} \boldsymbol{\omega}^v_{vr} \tag{3.36}$$

where $\boldsymbol{R}_{sv}$ is the rotation matrix from the vehicle frame $\boldsymbol{v}$ to the IMU sensor frame $\boldsymbol{s}$ expressed as a fixed rotation in $\boldsymbol{s}$.

3.5 Derivation of the accelerometer

The 3-axis accelerometer measures the acceleration of its own sensor frame. Additionally, it is affected by gravity, and at rest it will provide a measurement of $1[g]$ / $9.81[m/s^2]$ in the upward

direction[23] is the world frame $\boldsymbol{w}$. As the accelerometer is located in a virtual environment, it is assumed that this environment is sufficiently small such that the gravity vector is constant across the entire virtual environment as explained in section 3.1.

Before the following derivation, it must be stated that the rest of this section is based on section 6.4.4 of Barfoot[16] with some minor notation differences to fit the conventions used in this thesis.

The acceleration of the vehicle frame, $\boldsymbol{v}$, with respect to the reference frame, $\boldsymbol{r}$, is measured in the sensor frame $\boldsymbol{s}$. Hence it can be expressed as

$$\boldsymbol{a}_{vr}^{s} = \boldsymbol{R}_{sr}(\ddot{\boldsymbol{t}}_{sr}^{r} - \boldsymbol{g}^{r}) \tag{3.37}$$

where $\ddot{\boldsymbol{t}}_{sr}^{r}$ is the acceleration from the reference frame $\boldsymbol{r}$ to the sensor frame $\boldsymbol{s}$ expressed in frame $\boldsymbol{r}$. Likewise, $\boldsymbol{g}^{r}$ is the gravity vector expressed in frame $\boldsymbol{r}$. $\boldsymbol{R}_{sr}$ rotates the two vectors from frame $\boldsymbol{r}$ into $\boldsymbol{s}$.

The measurements from the accelerometer now depends on $\ddot{\boldsymbol{t}}_{sr}^{r}$. To find this, $\boldsymbol{t}_{sr}^{r}$ is derived:

$$\begin{aligned} \boldsymbol{t}_{sr}^{r} &= \boldsymbol{t}_{vr}^{r} + \boldsymbol{t}_{sv}^{r} \\ &= \boldsymbol{t}_{vr}^{r} + \boldsymbol{R}_{rv}\boldsymbol{t}_{sv}^{v} \\ &= \boldsymbol{t}_{vr}^{r} + \boldsymbol{R}_{vr}^{\top}\boldsymbol{t}_{sv}^{v} \end{aligned} \tag{3.38}$$

Differentiating $\boldsymbol{t}_{sr}^{r}$ with respect to time, the following is obtained:

$$\begin{aligned} \dot{\boldsymbol{t}}_{sr}^{r} &= \dot{\boldsymbol{t}}_{vr}^{r} + \dot{\boldsymbol{R}}_{vr}^{\top}\boldsymbol{t}_{sv}^{v} + \boldsymbol{R}_{vr}^{\top}\dot{\boldsymbol{t}}_{sv}^{v}, \qquad \dot{\boldsymbol{t}}_{sv}^{v} = \boldsymbol{0} \\ &= \dot{\boldsymbol{t}}_{vr}^{r} + \boldsymbol{R}_{vr}^{\top}[\boldsymbol{\omega}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} \end{aligned} \tag{3.39}$$

Looking back at eq. (3.39), t_{sv}^{v} is zero as it is assumed a constant offset from the vehicle frame to the sensors' position on that vehicle. Furthermore, $[\boldsymbol{\omega}]_{\times}$ is the *skew-symmetric* matrix of the vector $\boldsymbol{\omega}$:

$$[\boldsymbol{\omega}]_{\times} = \boldsymbol{S}(\boldsymbol{\omega}) = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}, \qquad \boldsymbol{\omega} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$$

Further differentiation with respect to time on the velocity found in eq. (3.39) yields the following:

$$\begin{aligned} \ddot{\boldsymbol{t}}_{sr}^{r} &= \ddot{\boldsymbol{t}}_{vr}^{r} + \dot{\boldsymbol{R}}_{vr}^{\top}[\boldsymbol{\omega}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} + \boldsymbol{R}_{vr}^{\top}[\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} + \boldsymbol{R}_{vr}^{\top}[\boldsymbol{\omega}_{vr}^{v}]_{\times} \cdot \boldsymbol{0} \\ &= \ddot{\boldsymbol{t}}_{vr}^{r} + \boldsymbol{R}_{vr}^{\top}[\boldsymbol{\omega}_{vr}^{v}]_{\times}[\boldsymbol{\omega}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} + \boldsymbol{R}_{vr}^{\top}[\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} \end{aligned} \tag{3.40}$$

Inserting eq. (3.40) into eq. (3.37) results in the following expression for $\boldsymbol{a}_{vr}^{s}$:

$$\begin{aligned} \boldsymbol{a}_{vr}^{s} &= \boldsymbol{R}_{sr}(\ddot{\boldsymbol{t}}_{sr}^{r} - \boldsymbol{g}^{r}) \\ &= \boldsymbol{R}_{sv}\boldsymbol{R}_{vr}(\ddot{\boldsymbol{t}}_{sr}^{r} - \boldsymbol{g}^{r}) \\ &= \boldsymbol{R}_{sv}(\boldsymbol{R}_{vr}(\ddot{\boldsymbol{t}}_{vr}^{r} + \boldsymbol{R}_{vr}^{\top}[\boldsymbol{\omega}_{vr}^{v}]_{\times}[\boldsymbol{\omega}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} + \boldsymbol{R}_{vr}^{\top}[\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v}) - \boldsymbol{R}_{vr}\boldsymbol{g}^{r}) \\ &= \boldsymbol{R}_{sv}(\boldsymbol{R}_{vr}\ddot{\boldsymbol{t}}_{vr}^{r} + [\boldsymbol{\omega}_{vr}^{v}]_{\times}[\boldsymbol{\omega}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} + [\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} - \boldsymbol{R}_{vr}\boldsymbol{g}^{r}) \\ &= \boldsymbol{R}_{sv}(\boldsymbol{R}_{vr}(\ddot{\boldsymbol{t}}_{vr}^{r} - \boldsymbol{g}^{r}) + [\boldsymbol{\omega}_{vr}^{v}]_{\times}[\boldsymbol{\omega}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} + [\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times}\boldsymbol{t}_{sv}^{v} \\ &= \boldsymbol{R}_{sv}(\boldsymbol{R}_{vr}(\ddot{\boldsymbol{t}}_{vr}^{r} - \boldsymbol{g}^{r}) + ([\boldsymbol{\omega}_{vr}^{v}]_{\times}[\boldsymbol{\omega}_{vr}^{v}]_{\times} + [\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times})\boldsymbol{t}_{sv}^{v}) \end{aligned} \tag{3.41}$$

3.6 Complete IMU model

By combining eqs. (3.36) and (3.41), the complete IMU model is found:

$$\begin{bmatrix} \boldsymbol{a}_{vr}^{s} \\ \boldsymbol{\omega}_{vr}^{s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_{sv}(\boldsymbol{R}_{vr}(\ddot{\boldsymbol{t}}_{vr}^{r} - \boldsymbol{g}^{r}) + ([\boldsymbol{\omega}_{vr}^{v}]_{\times}[\boldsymbol{\omega}_{vr}^{v}]_{\times} + [\dot{\boldsymbol{\omega}}_{vr}^{v}]_{\times})\boldsymbol{t}_{sv}^{v}) \\ \boldsymbol{R}_{sv}\boldsymbol{\omega}_{vr}^{v} \end{bmatrix} \tag{3.42}$$

From eq. (3.42) it is seen that several variables are need in order to generate synthetic IMU measurements. $\boldsymbol{t}_{vr}^{r}$, $\dot{\boldsymbol{t}}_{vr}^{r}$ and $\ddot{\boldsymbol{t}}_{vr}^{r}$ are the translation, velocity and acceleration of the vehicle frame in the reference frame, expressed with respect to the reference frame. Likewise, $\boldsymbol{R}_{vr}$ is the rotation from the reference frame to the current vehicle frame, and $\boldsymbol{\omega}_{vr}^{v}$ and $\dot{\boldsymbol{\omega}}_{vr}^{v}$ are the Euler angular velocity and acceleration from the reference frame to the vehicle frame, expressed in the vehicle frame. All the required states for the IMU derivation is extracted directly from the states in the VCS.

3.6.1 IMU measurement visualizations

To illustrate how the synthetic IMU measurements behaves, visualization of two scenarios are included. Please note that the gravity vector is nearly two magnitudes larger than the acceleration information we want to visualize. Therefore the gravity vector is excluded in this example. This visualization uses the same vehicle, $\boldsymbol{v}$, and sensor frame $\boldsymbol{s}$ as shown in fig. 3.2. The offset of the IMU to the vehicle center of mass is: $\boldsymbol{t}_{sv}^{v} = [0,\ 0,\ -0.5]^{\top}[m]$.

Figure 3.5 visualizes the acceleration measured in the IMU in frame $\boldsymbol{s}$ when the vehicle moves 1m along each of the Z, Y and X-axes of the vehicle frame $\boldsymbol{v}$. Angular velocity and acceleration during a -180degree (clockwise following the right-hand rule) rotation around the Y-axis in the vehicle frame $\boldsymbol{v}$ with no translation is shown in fig. 3.6. No translational acceleration takes place in the vehicle body, all acceleration measured in the IMU are due to centripetal forces arising from the rotation of the vehicle and the offset IMU relative to the vehicle. Figure 3.7 shows the motion in the two previous examples relative to the vehicle and IMU bodies and their corresponding coordinate frames.

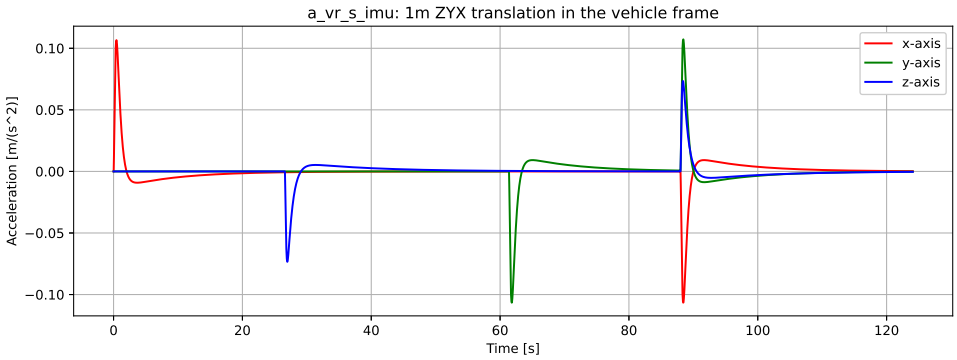


Figure 3.5: IMU measurements for 1m translation along the positive ZYX axes in the vehicle coordinate frame $\boldsymbol{v}$, in that order. The orientation of the IMU sensor frame $\boldsymbol{s}$ relative to $\boldsymbol{v}$ and the translations relative to frame $\boldsymbol{v}$ is shown in fig. 3.7. Note: The gravity-vector is not included for visualization purposes due to the low magnitude of the acceleration in the visualized data.

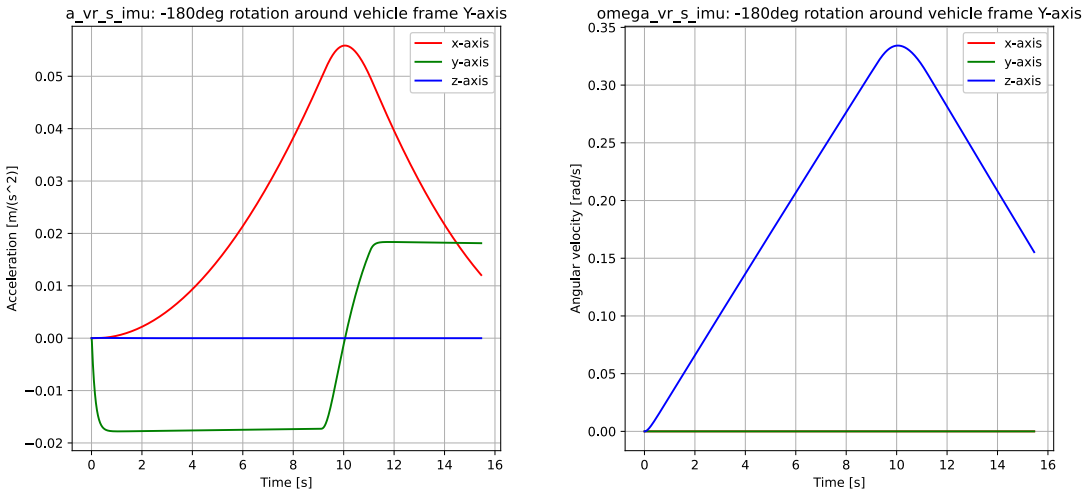


Figure 3.6: IMU measurements for a negative 180-degree rotation (clockwise following the right-hand rule) around the Y-axis in the vehicle frame $\boldsymbol{v}$, which is parallel with the Z-axis in the IMU sensor frame $\boldsymbol{s}$. The orientation of the IMU sensor frame $\boldsymbol{s}$ relative to $\boldsymbol{v}$ and the rotation relative to frame $\boldsymbol{v}$ is shown in fig. 3.7. Note: The gravity-vector is not included for visualization purposes due to the low acceleration magnitude of the visualized data.

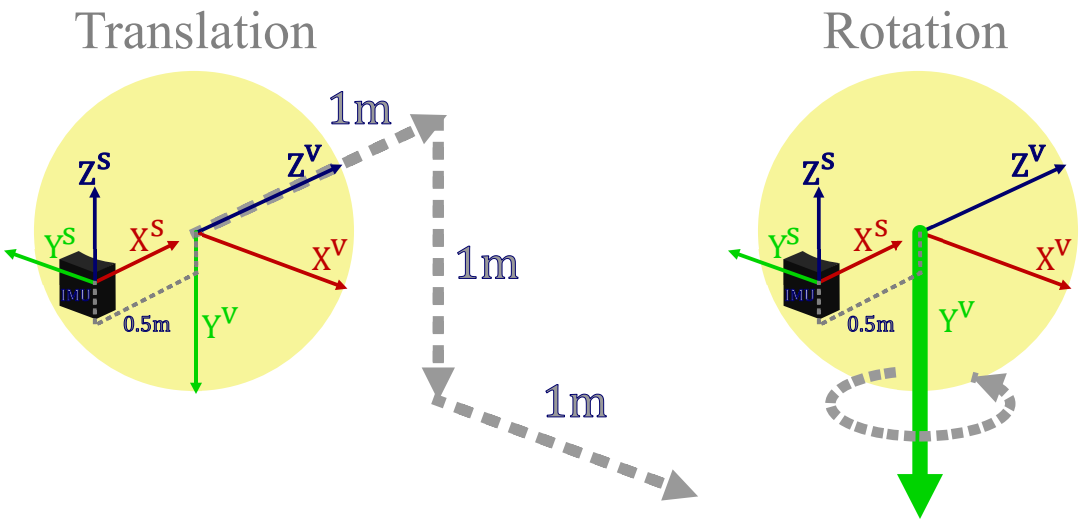


Figure 3.7: Left: The translation of the vehicle, the yellow sphere, with the attached IMU plotted in fig. 3.5 visualized. Right: Clockwise rotation around the vehicle $\boldsymbol{y}$-axis, corresponding to the plot in fig. 3.6

3.7 Pressure sensor

Most underwater vehicles are equipped with a pressure sensor which derives the depth based on the pressure the water exerts. A pressure sensor measures the pressure, and converts the measurement to SI units as $[Bar]$ or $[Pa]$. This data is then electronically converted to a depth estimate in meters based on the properties of water. The depth estimate in the simulation case is the world-frame z-axis position plus a desired depth offset as the simulation environment geometry is closely aligned to $z = 0$.

Let d_v^w be the depth of the vehicle extracted from the simulation expressed in the world frame $\boldsymbol{w}$. Let $\tilde{d}_v^w$ be the depth-corrected vehicle depth. Finally, $\tilde{d}^w$ is the virtual depth offset to achieve realistic depth levels. Note that the world- and depth sensor frames are defined with the positive z-direction upwards. Then the raw depth data can be synthesized as follows:

$$\tilde{d}_v^w = d_v^w + \tilde{d}^w \tag{3.43}$$

3.8 Measurement resolution

The exported IMU data doesn't have any constraints for its resolution to allow for the user to adapt this to their own requirements. A real sensor will have a limited resolution. This means that the measurements is not a continuous curve, but instead are confined to values that are determined by the resolution step size. First, set the desired resolution and find the residual of a given value through eq. (3.44), where % is the modulo-operator.

$$residual = value \ \% \ resolution \tag{3.44}$$

If the residual is smaller than the half of the resolution step and larger than 0, the residual is subtracted. If the residual is larger, then the residual is subtracted and the resolution step is added. In code, the logic is as follows:

Algorithm 1: Measurement resolution augmentation

Input: $Value$, $Resolution$
Output: $DiscretizedValue$

$Residual \leftarrow Value$ modulo $Resolution$
if $Residual == 0$ **then**
    $Donothing$
else if $Residual$ ¡ $Resolution/2$ **then**
    $Value \leftarrow Value - Residual$
else
    $Value \leftarrow Value - Residual + Resolution$
$DiscretizedValue \leftarrow value$

3.9 Measurement noise

IMU measurement noise is found to be white Gaussian through experiments[24]. Furthermore, this noise can be unbiased or biased, where a bias will result in measurement drift. This is dependent on the properties of the sensor used. As there are a wide variety of different IMU sensors on the market, from cheap low-end and less accurate to expensive high-end and very accurate, the sensor data is generated without any noise augmentation. Further noise augmentation is done in post processing, where a wide array of noise models can be used depending on the sensor noise characteristics that are needed.

4 Simulation data

4.1 Non-image sensor data format

4.1.1 IMU measurements

IMU measurements are given at a sample frequency of 200Hz, which includes acceleration and angular velocity of the vehicle expressed in the sensor frame $\boldsymbol{s}$, $\boldsymbol{a}^s_{vr}$ and $\boldsymbol{\omega}^s_{vr}$ respectively. These vectors consists of the following components:

$$\boldsymbol{a}^s_{vr} = [a^s_{x,vr},\ a^s_{y,vr},\ a^s_{z,vr}]^\top \qquad \boldsymbol{\omega}^s_{vr} = [\omega^s_{x,vr},\ \omega^s_{y,vr},\ \omega^s_{z,vr}]^\top$$

Every IMU measurement is then exported as a Comma Separated Value (CSV) file such that each line reads:

$$timestamp[ns],\ a^s_{x,vr},\ a^s_{y,vr},\ a^s_{z,vr},\ \omega^s_{x,vr},\ \omega^s_{y,vr},\ \omega^s_{z,vr}$$

4.1.2 Depth gauge measurements

Depth gauge measurements are given at the same sample frequency as the IMU, which is 200Hz. The depth measurement, $\tilde{d}^w_v$, is exported as a CSV file on the following form:

$$timestamp[ns],\ \tilde{d}^w_v$$

4.2 Image data format

4.2.1 RGB images

The raw RGB images are exported as OpenEXR files, which stores the value in each channel for every pixel as a 16-bit float. These images contain the uncompressed render output without discretization of the pixel values, allowing for auto-exposure in post-processing without causing banding in an image's histogram due to modification of discretized values. After post-processing, the images are exported as lossless 8bit sRGB PNG files for the benchmark sequence, corresponding to the standards used by KITTI[1], LiU[2] datasets. These images are easily converted to monochromatic 8bit PNG images that are used by datasets such as the underwater dataset AQUALOC[4] and the aerial dataset EuRoC[3]. The color management settings are shown in table 4.1.

Table 4.1: Export setup for RGB images

Filetype	16bit OpenEXR
View transform	Filmic
Sequencer	sRGB

4.2.2 Depth images

Depth images store the distance from the camera to each pixel. Following the standard format set by datasets such as the KITTI[1] dataset, this data is extracted as a 16-bit integer in 16-bit grayscale PNG images, with export settings as listed in table 4.2. Using this standard, the distance to a given pixel is mapped to a value in $[0, 2^{16} - 1] \Leftrightarrow [0, 65535]$. By choosing a start- and end distance, the resolution is determined through eq. (4.1). Here it is important to choose the start- and end distance such that the resolution remains sufficiently small. While 16bit OpenEXR files were considered, we decided to opt with 16-bit PNG as it is the established standard for the majority of the available datasets which provides depth data.

$$resolution = \frac{end\_distance - start\_distance}{2^{bit\_depth}} \tag{4.1}$$

Table 4.2: Export setup for depth images

Filetype	16bit PNG
View transform	Raw
Sequencer	Linear

4.2.3 Normal images

Normal images store the world-space surface normal in each pixel of the image. Blender uses OpenGL, hence the normal vector in world space, $\boldsymbol{n}^w$, uses the following encoding[25]:

$$\boldsymbol{n}^w = \begin{bmatrix} n_x^w & n_y^w & n_z^w \end{bmatrix}^\top \qquad \{n_x^w,\ n_y^w,\ n_z^w\} \in [0, 1] \tag{4.2}$$

Normal images following the OpenGL standard stores $\boldsymbol{n}^w$ directly in the RGB channels of the image, such that:

$$\begin{aligned} n_x^w &\to R \\ n_y^w &\to G \\ n_z^w &\to B \end{aligned} \tag{4.3}$$

An example image following eq. (4.3) is illustrated in fig. 4.1. It is important to note that many scenes will contain geometry which will have sharp edges visible in the normal image if no normal smoothing is applied before rendering. Normal smoothing is an operation done to smooth the transition between two surfaces, creating the appearance of a smooth surface with a relatively low resolution model. In Blender, the threshold angle for normal smoothing can be set according to scene requirements. The image export settings used is shown in table 4.3.

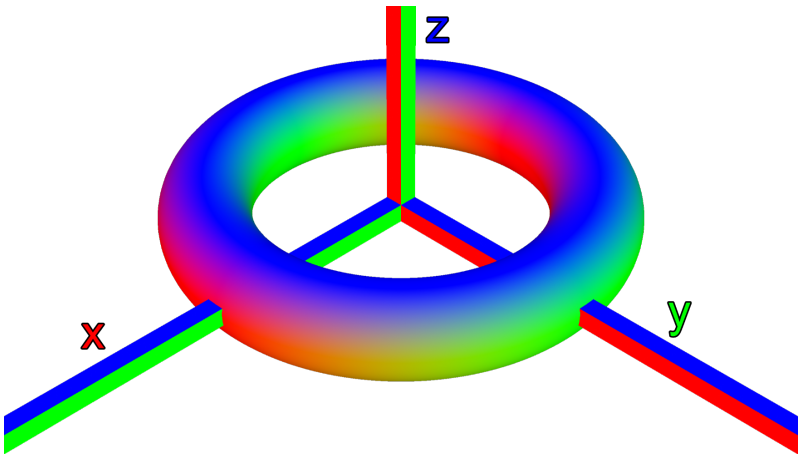


Figure 4.1: Normal map in the world space. The axes in the image correspond the the $x-$, $y-$ and $z-$axes in the world frame $\boldsymbol{w}$. Note that this world-space normal map has been post processed to remove a black background.

Table 4.3: Export setup for normal images

Filetype	16bit PNG
View transform	Raw
Sequencer	Linear

4.3 Ground truth data format

Ground truth data for the vehicle pose for every sensor sample (camera, IMU, depth) is provided in the world frame. Each sensor type will have a file with ground truth pose information in the

corresponding sensor-folder. When the stream-file format, which is discussed later in this section, is used, the ground truth information will also be baked into this file.

4.4 Image naming convention

The filename for the images in the dataset should be chosen such that the origins of the image and its placement in a given benchmark sequence can be identified outside its original folder structure. Therefore the naming scheme must fulfill the following requirements with respect to present identifiers in the filename:

- Sequence ID
- Vehicle number
- Image type
- Camera type
- Camera number
- Separator to split identifiers from frame number
- Frame number

4.4.1 Sequence ID

The sequence ID is proposed to include the leading characters "seq" to indicate that the image comes from a sequence and to make it easily legible. Following are two ASCII character for enumerating this sequence, encompassing numbers as well as lowercase and uppercase letters. This gives the available enumeration symbols listed in table 4.4. By having a total of 62 available IDs for every character, using two characters translates into $62^2 = 3844$ unique sequence IDs.

Table 4.4: Available ASCII letters for enumeration and identification purposes.

Letter type	Range	Number of IDs
Numbers	0-9	10
Lowercase	a-z	26
Uppercase	A-Z	26
Total	(0-9, a-z, A-Z)	62

4.4.2 Vehicle number

The vehicle number is included to allow for separation between multiple vehicles operating simultaneously. The vehicle number is enumerated with a single ASCII character, and allows for up to 62 vehicles as outlined in table 4.4.

4.4.3 Image type

The image type specifies what type of data is contained within each image. At the time of writing, the following image types are planned:

- RGB images with water, illuminated with vehicle spotlights
- RGB images without water, uniformly illuminated
- Depth images

- Normal images
- Direct volumetric lighting
- Indirect volumetric lighting

From table 4.4, a single ASCII character gives 62 possible IDs. When taking into account the possibility of other common data formats such as sonar or RGB-D, 62 different image types is still provides a large buffer of empty IDs. Hence a single ASCII character will be used to differentiate between the image types.

4.4.4 Camera type

The camera type specifies the camera type used to generate the image. This includes, but is not limited to, the following camera types:

- Monocular camera
- Stereo left camera
- Stereo Right camera
- Downward facing camera
- Multi-camera 1
- Multi-camera 2

Using a single ASCII character, it is possible to encode for 62 different camera types. Multi-cameras are included as the camera-number will serve as an ID to an individual camera of the multi-camera setup. This is to make it easier to differentiate between a multi-camera setup and monocular cameras placed at points of interest.

4.4.5 Camera number

The camera number is the number of a given camera type on a vehicle. For example, a single vehicle may have several monocular cameras pointed in various directions. The camera number would then point to which monocular camera the image belongs to. For a multi-camera setup the image number points to a specific camera in a given multi-camera.

4.4.6 Separator

A unique ASCII character is used in the filename as a separator between the identifiers in the filename and the frame number. The hyphen character, "-", is used as a separator since it is easily legible, unique to the filename and will not cause issues in Windows, iOS or Linux-based file systems.

4.4.7 Frame number

The frame number must contain enough digits to allow for both path-traced- and real-world image sequences. Due to the computational requirement for path-traced images, the number of real-world images will determine the digits available for frame numbering. By using 8 digits, there is room for 463 hours of footage at 60Hz.

4.4.8 Final image naming convention

Finally, the image filenames will contain two strings to make it more legible without the need of a look-up table of the ASCII character encoding. The first three characters are "seq", indicating that the sequence number follows. Likewise, the camera type and camera number is preceded by the three characters "cam" indicating that camera information follows. The image naming convention is as follows:

seq[SeqNumber]_veh[VehicleNumber]_cam[CameraType][CameraNumber]_[ImageType]-########.[Ext]

A table of codes for the image naming convention is listed in table 4.5. To demonstrate the name of an image belonging to:

- SeqNumber: 02
- VehicleNumber: 0
- ImageType: A (RGB images, with water)
- CameraType: M (Monocular camera)
- CameraNumber: 0 (First monocular camera)
- FrameNumber: 00000000 (First frame in the sequence)
- Ext: png (Image file extension)

Then the image filename would be:

seq02_camM0_A-00000000.png

Table 4.5: Description of the codes used in the image filenames to reduce the filename length.

Tag	Code	Text description
SeqNumber		
	01	Vasquez test sequence
	02	Vasquez, first proper underwater render
ImageType		
	A	RGB images, with water
	B	RGB uniformly lit images, no water
	C	World-space normals viewed by the camera
	D	Pixel accurate depth map
	E	Shadow-map
	F	Direct volumetric lighting
	G	Indirect volumetric lighting
CameraType		
	M	Monocular camera
	L	Stereo left camera
	R	Stereo right camera
	D	Downward facing camera
	A	Multicamera 1
	B	Multicamera 2
CameraNumber		
	0	First camera of a given CameraType
	1	Second camera of a given CameraType

4.5 Data structure

4.5.1 Structure motivation

The data structure of the dataset is made with inspiration primarily from the LiU[2], traffic scenes, and EuRoC[3], containing micro aerial vehicle footage, datasets. KITTI[1], traffic scenes, and AQUALOC[4], underwater dataset, was used as additional references for the format of the included data to ensure maximum compatibility with already existing datasets.

4.5.2 LiU Stream Format

A group from the Linköping University (LiU) proposed in 2013 a framework for storing dataset data named the *LiU Stream Format*[2]. The main file in the LiU Stream Format is a main file in the root folder of the dataset, a streamfile. This consists of a header containing version information for compatibility checks, and after that sequential entries containing data information. The sequential entries are composed of either pointers to external data such as images or they consist of data from sensors. We aim to include this format in future iterations of the framework such that we can provide a single sequential file with pointers to relevant data. However at the time of writing this thesis, this is not yet ready.

Table 4.6: A single message in the LiU stream file format[2].

Messagestart	ID	Length of Payload	Payload
4x uint8 '$BST'	uint32	uint32	Dependent on message

4.5.3 Folder structure

Our initial version of the stream format is supposed to be a more simple version of the LiU stream file format, consisting of a binary stream file with the structure in table 4.6, and no ceiling on the max number of files in a single folder.

It is going to contain pointers to images, and include the IMU and depth sensor data as well as ground truth data directly as a payload. However, due to time restrictions, this functionality has been put on hold. The folder structure will however remain as planned, where the IMU, depth and camera sensor data is stored in their respective folders to allow for easy extraction of data. Coincidentally this folder structure is very similar to that of EuRoC[3] as we, in contrast to LiU, wish to include support for data from multiple vehicles operating simultaneously in a single dataset.

Continuing the image filename example from section 4.4.8, the final folder structure is illustrated in fig. 4.2.

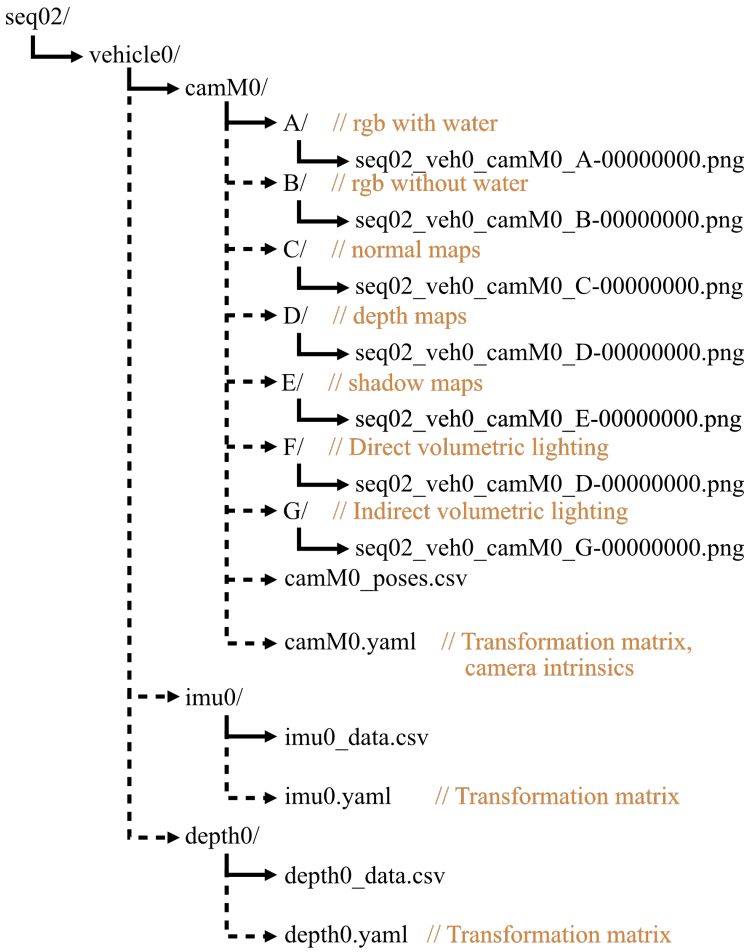


Figure 4.2: Simulation data folder structure without the sequential streaming file in the current stage of development. The sequential file containing image pointers, sensor data and ground truth will be included at a later stage in development.

5 Simulation settings and setup

5.1 Water light scattering and absorption properties

In order to properly understand the requirements of a physically correct water scattering medium, the physical properties must first be examined. Light is influenced through absorption and scattering when traveling through a volume which is not under a perfect vacuum. The following explanation is closely based on the Ocean Optics Web Book[26].

Light can be expressed as radiant flux $\mathbf{\Phi}$ $[W]$ which is the radiant energy per time unit. When an incoming radiant flux $\mathbf{\Phi}_i(\lambda)$ of a wavelength λ enters a scattering volume $\boldsymbol{V}$, three separate phenomena takes place. A part of the energy is absorbed and converted to chemical or thermal energy, denoted as $\mathbf{\Phi}_a(\lambda)$. The amount of energy which is absorbed is dependent on the properties and matter composition of the scattering medium and the wavelength of incoming light, λ.

Furthermore, a portion of the radiant flux will scatter and exit the scattering volume with an angle-difference of ψ compared to that of the incoming flux. This scattered flux is denoted $\mathbf{\Phi}_s(\lambda, \psi)$. The scattering property gives rise to a very important phenomena, namely ambient illumination. Scattered light bounces inside or outside the direct cone of light, illuminating occluded parts of the volume or surfaces in addition to the volume and surfaces inside the cone of light[27, 28]. This is illustrated in fig. 5.2.

The remainder of the radiant flux, $\mathbf{\Phi}_t(\lambda)$ is transmitted through the scattering volume. The entire scattering and absorption process is illustrated in fig. 5.1.

5.1.1 Suspended particles and absorption

The ocean contains a lot of organic matter such as plankton, algae or other biologic material. In addition, underwater currents can disturb the seabed, introducing suspended sediments in the water volume. All of these suspended particles will hereafter be referred to as suspensoids. These suspensoids will affect the absorption differently for each wavelength depending on their composition[26], altering the perceived color of the scattering medium when illuminated.

5.1.2 Suspended particles and backscattering

Suspensoids does create another important phenomena; backscattering[27, 28]. This is light bouncing back towards its source as it is reflected by microscopic particles or suspensoids in the scattering medium, or by the scattering properties of the scattering medium itself. As the camera and light source often is placed close together to minimize shadows, the effect of severely backscattered light is similar to that of driving in dense fog with high beams on.

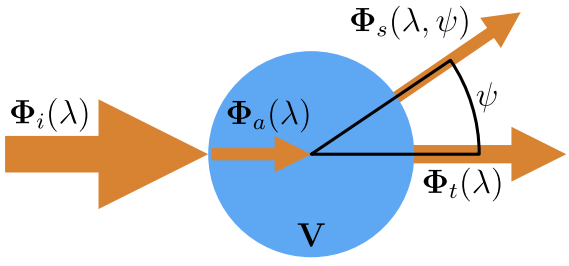


Figure 5.1: Simplified model of incoming radiant flux of wavelength λ, denoted $\mathbf{\Phi}_i(\lambda)$, being absorbed, $\mathbf{\Phi}_a(\lambda)$, and scattered $\mathbf{\Phi}_s(\lambda, \psi)$ by an angle ψ in a volume $\boldsymbol{V}$. The radiant flux which is transmitted through the volume is denoted $\mathbf{\Phi}_t(\lambda)$. Figure is inspired from [26].

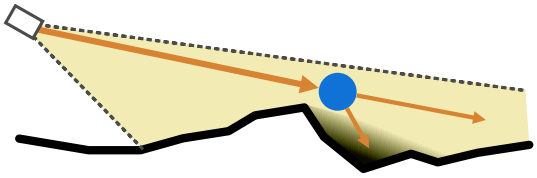


Figure 5.2: Ambient illumination outside the direct cone of light resulting from scattered light in the main light cone, which enters the occluded area providing some level of illumination.

5.2 Blender interface

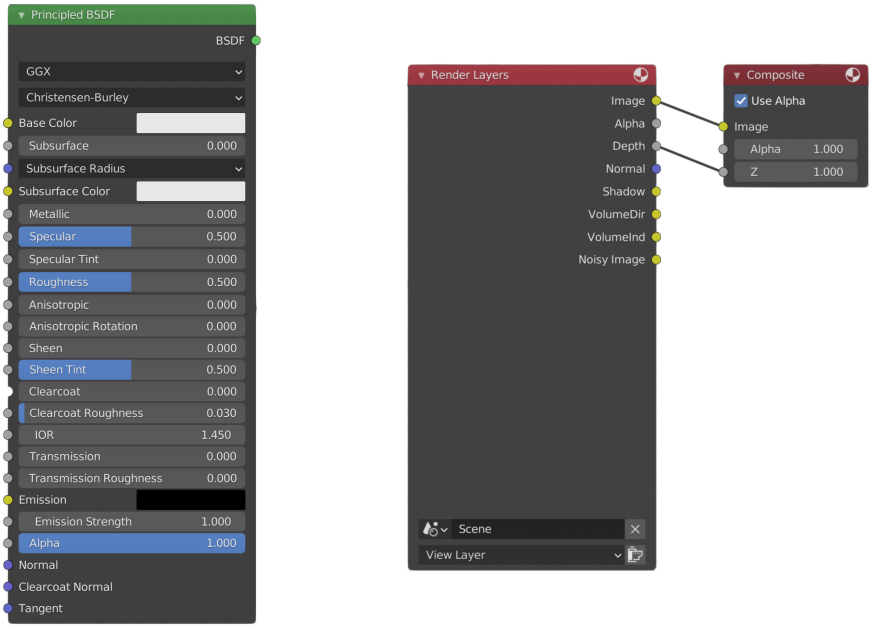
Blender[10] 2.81 and 2.92 has been used during development of the simulation environment. The Blender releases of 2.8x and 2.9x both share a very similar interface, where 2.9x versions feature various render optimizations. One key aspect of Blender is that both materials for 3D objects and compositing of export data is node-based. Materials for objects are used using material nodes, discussed in section 5.2.1, and image export of multiple image requires use of the compositor nodes, discussed in section 5.2.2.

5.2.1 Material nodes

Materials in Blender is what decides the visual properties of an object. Materials are node based, and are found in Blenders shading editor. The default node for materials in Blender 2.8+ is the ”Principled BSDF” shader as seen in fig. 5.3a. This node can either use image textures or procedural colors made from combining noise and object parameters as inputs to ”Base Color”, ”Metallic”, ”Roughness” and ”Normal”. The base color is the true color of the object, and the metallic property is the metallicity. Roughness reflects all light at a value of 0, and is completely non-reflective at a value of 1. The normal input takes a map of the objects surface normal vectors, and uses this to create shadows or highlights independent of the underlying mesh.

5.2.2 Compositing nodes

Compositing post-processes the raw data from a render, and is used for setting up multiple image outputs if this is required. This is done with the compositor editor in Blender. The main nodes in the compositor is the ”Render Layers” and the ”Composite” node, pictured in fig. 5.3b. By default, Blender uses the Composite node for the RGB image output. The ”Render Layers” makes it possible to access the various render layers produced during the render process. These passes are further discussed in section 5.6.1.



(a) Shader editor: Principled BSDF material node

(b) Composite editor: "Render Layers" and "Composite" node.

Figure 5.3: The shader editor material node (left) and the composite editor nodes (right).

5.3 Environment requirements

5.3.1 Water properties

The scattering volume must be able to recreate the physical properties of water as discussed in section 5.1, while also being able to create images in a reasonable timeframe as we aim to produce hundreds to thousands of images in different image sequences.

5.3.2 Object textures

The textures used on models in the scene should have colors corresponding to the true color of the object. If an object is viewed underwater, the colors can be distorted if the water is very turbid or has a high concentration of algae. This color warping should be performed indirectly through the use of a scattering medium instead of altering the texture colors. Object textures is also required to be physically accurate with respect to color, uneven surfaces that cast very small amounts of shadows, amount of light reflected and metallicity.

5.3.3 Environment geometry

The underwater environments we aim to replicate should contain both flat areas that are similar to a simple seafloor, and more complex geometry building in all three dimensions. By including these different environment characteristics, it is possible to both use the same environment for different shorter sequences of each individual area, as well as longer sequences visiting each area. The latter would allow for creating a single sequence with quite different scenarios for testing of 3D reconstruction, VO and SLAM algorithms.

5.3.4 3D asset geometry

3D assets used in the environment should also have as realistic shapes as possible. If a 3D model has a sharp 90-degree corner or edge, no light will be reflected from this edge. In the real world, edges are slightly rolled over from the manufacturing process or due to paint layers. As a result, 90-degree edges on structures will reflect some light as shown in fig. 5.4 This reflection has to be included as the images of these 3D assets will be used for 3D reconstruction, VO and SLAM purposes. Therefore care has to be taken to ensure that the assets used have beveled edges.

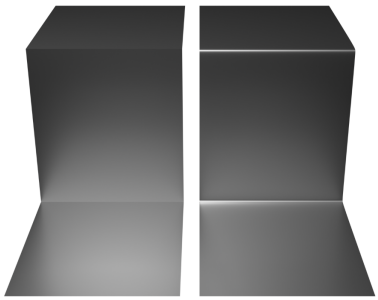


Figure 5.4: Two models with different edge geometry to showcase the effect different effects it has on shading and light reflections. Model on the left features sharp 90-degree edges, and the model on the right features smooth rounded edges.

5.4 Environment setup

5.4.1 Photogrammetry 3D model

Photogrammetry is process of creating 3D objects from images, and is synonymous with 3D reconstruction with textures in this thesis. 3D reconstructions can yield very good models with both geometry and textures that are a good representation of the real-world object with modern camera equipment and software.

There are many photogrammetry models posted online of different scenes, many of which are posted under a Creative Commons Attribution 4.0 license[29]. This license requires that credit must be given to the original author(s), and we are free to adapt the material to our needs.

A photogrammetry model which fulfilled our requirements was a model of the Vasquez Rocks in California USA by Austin Beaulier[30], visualized in fig. 5.5. The model measures $632m \times 598m \times 57.9m$ (width $\times$ length $\times$ height) before scaling. Due to the limited speed of an underwater vehicle, the environment was reduced in size to $191m \times 161m \times 17.5m$ using the scaling factors $(0.302147 \times 0.268642 \times 0.302147)$. The length was scaled down more to increase the steepness of the angled surfaces present in the model. Hereafter the original Vasquez Rocks model will be referred to as the *original mesh*.

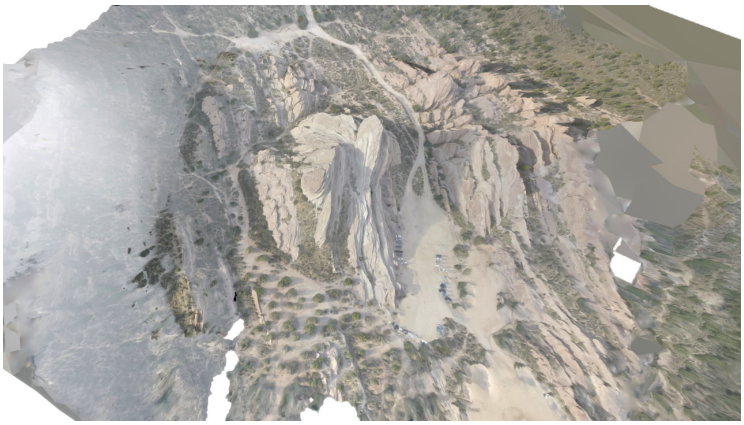


Figure 5.5: The Vasquez Rocks photogrammetry model from [30] scaled to $191m \times 161m \times 17.5m$. Completely white areas of the model are holes.

5.4.2 Importing the photogrammetry model

Blender: Merge photogrammetry objects into one object. Two common file formats for 3D reconstructed objects using photogrammetry are glTF[31] or Wavefront OBJ[32] files. glTF files often consists of several separate meshes, and must be merged into one object. In Blender, this is done by importing the glTF file, and merging the different meshes in the model. Then, the glTF orientation matrix, model correction matrix and model.obj.cleaner.materialmerger.gles is deleted. This will reset the orientation of the model, so it may have to be rotated back into the desired orientation. If any rotations are applied, make sure to apply the rotation, such that the objects current pose becomes the new "Origin pose" without any transformation applied. A step by step example from Blender's UI is shown in fig. 5.6.

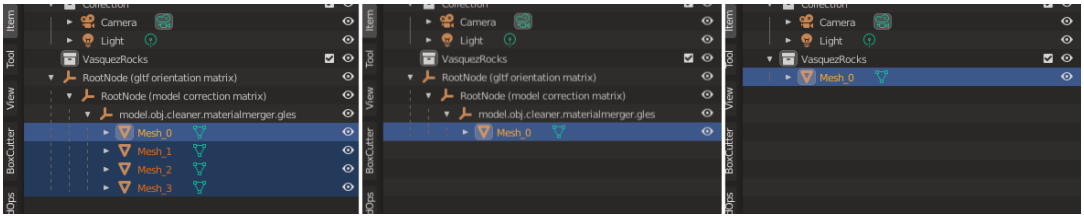


Figure 5.6: From left to right: Merging of the different sub-meshes in the imported glTF model into a single mesh, then deletion of the various orientation matrices present in the glTF file.

Blender: Merge vertices to make a watertight mesh. When the sub-meshes are merged into a single object, the vertices of each sub-mesh remain unchanged if Blender is used. This will result in edges where the mesh surface looks continuous, but in reality there is a small separation of the vertices at that location. In order to merge all vertices of an active object in Blender, ensure that the original mesh is selected. Then add a "Weld" modifier from "Add Modifier → Deform column → Weld". Set the merging distance "Distance" to a value large enough such that near overlapping vertices are merged, but yet small enough so no mesh details are lost due to merging. Making the mesh watertight means that all faces are properly connected. When surface smoothing is applied, it creates a smooth transition between the surface normal of two connected faces. As a result, the lighting visible on the model becomes more smooth without sharp transitions. This is illustrated in fig. 5.7. The mesh with merged vertices is then exported to a .obj or, preferably, .fbx file.

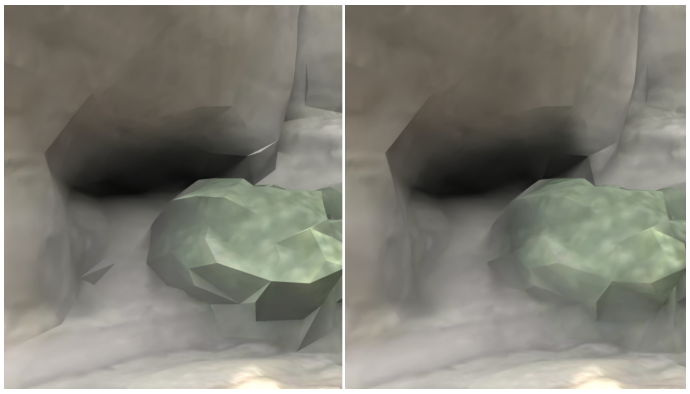


Figure 5.7: Left: the original model, right: model where vertices closer or equal to $0.1m$ distance are merged. Smoothed normals are enabled, smoothing the transition between two faces.

5.4.3 Post-processing: re-meshing the photogrammetry model

With a watertight model of the environment available, it will be re-meshed to increase the mesh resolution, and to create an optimized UV map.

What is UV unwrapping and mapping? UV mapping is the mapping the faces on a 3D model on to a corresponding set of faces on a 2D texture. This is used to store information about textures, roughness (or reflectivity), metallicity, surface normals and height displacement. The different types of information needed is dependent on the 3D rendering application. To use the 2D texture information, the 3D object has to be laid flat on a 2D plane through UV unwrapping. To exemplify the procedure, use a tetrahedron as illustrated in fig. 5.8. In order to lay this object flat on the plane of its base, three cuts must be made to separate two or more faces. In fig. 5.8 these are visualized as dotted lines. The same principle applies for any 3D surface or object.

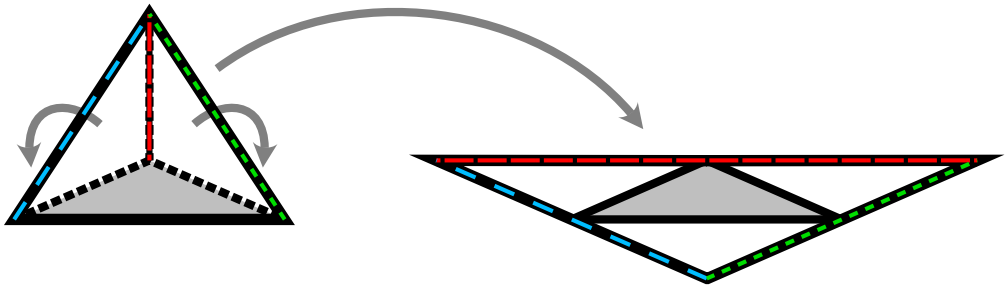


Figure 5.8: UV unwrapping process of a tetrahedron. The dashed lines indicate seams along the edges. These seams are cut to place the faces flat onto a planar surface.

Zbrush: Creating a new high-resolution mesh. The original mesh is triangulated, and will heavily distort if it is subdivided[3] further if we want to add more complexity. A mesh consisting of only faces with four corners, named quadrilaterals or "quads" for short, will subdivide without distortion.

To get a very high-resolution mesh, the digital sculpting software Zbrush[33] is used as it supports manipulation of very dense meshes with tens of millions of vertices. The original mesh with merged

[3]Subdivision is the process where a face is split into sub-faces. A triangle will split into three faces with four sides each, and a quadrilateral will split evenly into four faces.

vertices is imported, and a new flat mesh is created and scaled to the size of the imported mesh in Zbrush. The new mesh is UV-unwrapped at the lowest subdivision level before it is subdivided to approximately 250.000 vertices. A version of the new mesh at each subdivision stage is stored automatically by Zbrush in what can be described as a resolution pyramid[4]. At the different resolution levels, the mesh is either sculpted akin to digital clay or manipulated with dragging and pulling to roughly match the shape of the original mesh. Doing this step will reduce the probability of surface artifacts in the next step.

Next the features of the original mesh is projected to the newly created mesh. Starting at the lowest subdivision level, project the details from the original mesh to the new mesh. By doing this first at a low resolution ensures that the new mesh follows the surface of the original mesh as closely as possible. If the distance between the original mesh and the corresponding surface in the new mesh is too big, the projection algorithm can mistakenly use another surface as a reference. This is especially common for areas with a lot of 3D structure, like cliffs or pits in the ground with steep edges. If errors from projection arise, then adjust the maximum projection distance or modify the new mesh accordingly. After the highest subdivision level is reached, the mesh is further subdivided to make it more smooth, but the original mesh is no longer projected on to it. Doing so can make the new mesh project several polygons on to a single polygon in the original mesh, reducing the smoothness of the model. For the Vasquez Rocks, a total of 16.6million vertices was used at the highest subdivision level.

Before exporting, the mesh has to be decimated. This is the process of triangulating the mesh, where detail is preserved such that detailed areas contains a higher number of vertices than areas without significant details. Figure 5.9 showcases the effect of the decimation operation. As a result, the mesh has a much smaller memory footprint while the majority of the detail is preserved. Render setup times are impacted by the density of the model, hence the model should be decimated as much as possible while retaining the desired details.

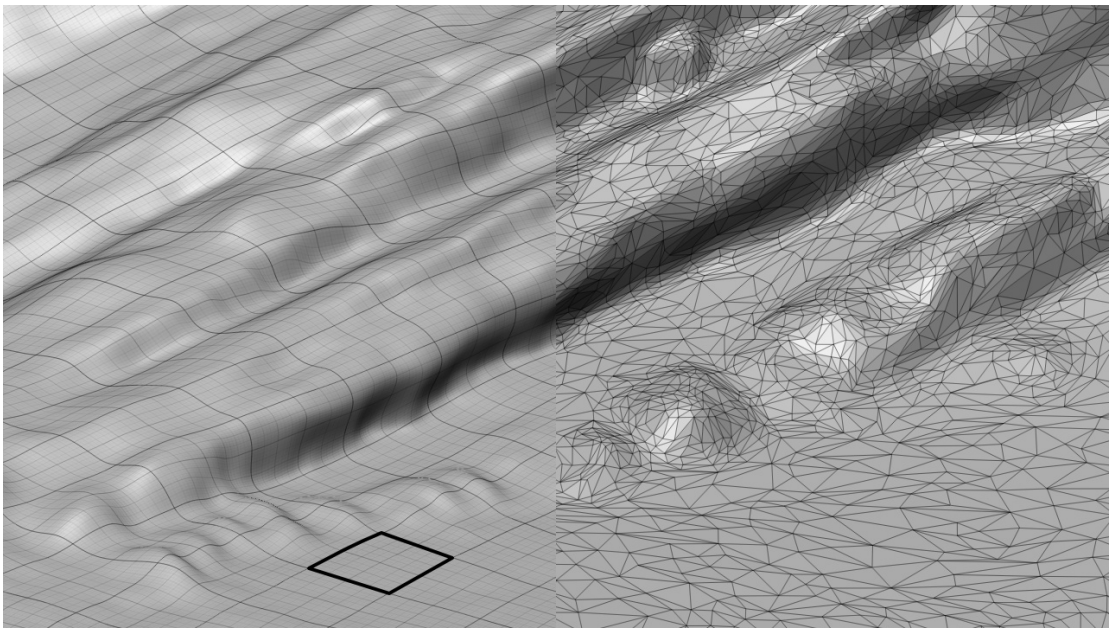


Figure 5.9: From the viewport in Zbrush. Left: All quad-mesh with a total of 16.6million vertices, where each large black square is an area of 32×32 vertices. Right: Decimated with a total of 250.000 vertices. Notice that the curved surfaces contain a larger number of vertices in a given area than the surrounding flat areas. Note: Only a small section of the mesh is imaged.

[4]As Zbrush is closed source, this is only an assumption to better explain how it works in practice.

5.4.4 Post-processing: Using a photogrammetry model directly

As a much simpler alternative to section 5.4.3, it is possible to use the photogrammetry model directly given that the original mesh has a sufficient resolution to allow for smooth surface shading. For reference, the model in fig. 5.7 has a too low resolution for our use, resulting in several areas with 90-degree changes in the face orientation in parts of the mesh. The sharp angles can cause shading artifacts which can create unrealistic features in the environment which can be detected by computer vision algorithms.

Blender: UV unwrap If the model is fairly flat such that it doesn't contain significant geometry in the z-direction, it is possible to UV unwrap the model directly in Blender. This is done by selecting all vertices and then selecting the standard UV unwrap option. If the model does have a lot of overlapping geometry, some seams will have to be placed in order to lie the model down flat without texture distortions. Here care has to be taken such that these seams are placed in areas where they are not easily visible as the textures rarely properly line up where these seams are placed.

Blender: Smart UV Project Smart UV Project is an alternative UV wrapping method which automatically creates seams if the angle between two faces are more than a set threshold. If the standard UV unwrap procedure fails, then this can be used. Again, care should be taken to ensure that the threshold angle is set high enough to avoid unnecessary seams.

5.4.5 Surface normal-weighted texturing

Texture requirements: The underwater environment consists of two different textures for the ground; sand for flat areas, and a rocky surface for angled surfaces. The textures should ideally not contain repeating patterns. This is a problem which must be considered due to the size of the environment. Using a single non-tiled texture for the entire area at its current size will require a square texture with a size of minimum 40960×40960 pixels, ideally 102400×102400 pixels or more for sufficient resolution when passing close to the surface. Even if a GPU had enough memory for these textures, the render times would slow down to a halt, making such large textures impossible to use in practice. Instead, 4096×4096 pixel tile-able textures are tiled 50 times to achieve a sufficient resolution when the camera is close to the surfaces with a relatively small memory and performance footprint. The blending between sand and rock textures is done by using a surface normal-weighted mask.

If the environment consists of a smaller with highly area specific textures, then it is possible to manually paint the environment using proper UV unwrapping and a texture painting suite such as Substance Painter[34]. This is very time consuming, but should be mentioned as an alternative should it be needed by future requirements.

Creating a surface normal-weighted mask: The surface normal vectors of the faces on a given object can be used to extract information whether or not a face is lying horizontally in the xy-plane, vertically along the z-axis, or somewhere in between. The normal vector of an object is given in the object's local coordinate system. Therefore, for consistency in this method, any rotation done to the object should be applied. This aligns the object's local coordinate system with the world coordinate system. Any normal vectors belonging to a flat surface should now contain a large z-component in the normal vector:

$$\boldsymbol{n} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^\top \longrightarrow \begin{bmatrix} R & G & B \end{bmatrix}^\top$$

The z-component, n_z, is visible in fig. 5.10 as a blue surface. This component is separated and modified for greater contrast between horizontal and vertical areas in the "ColorRamp" node. By mapping the values in $n_z \in [0, threshold] \rightarrow 0$, the areas with $n_z \leq threshold$ is excluded from

the mask, represented as the color black. For this specific environment object, $threshold = 0.836$ is used. Nodes for these operations is shown in fig. 5.11.

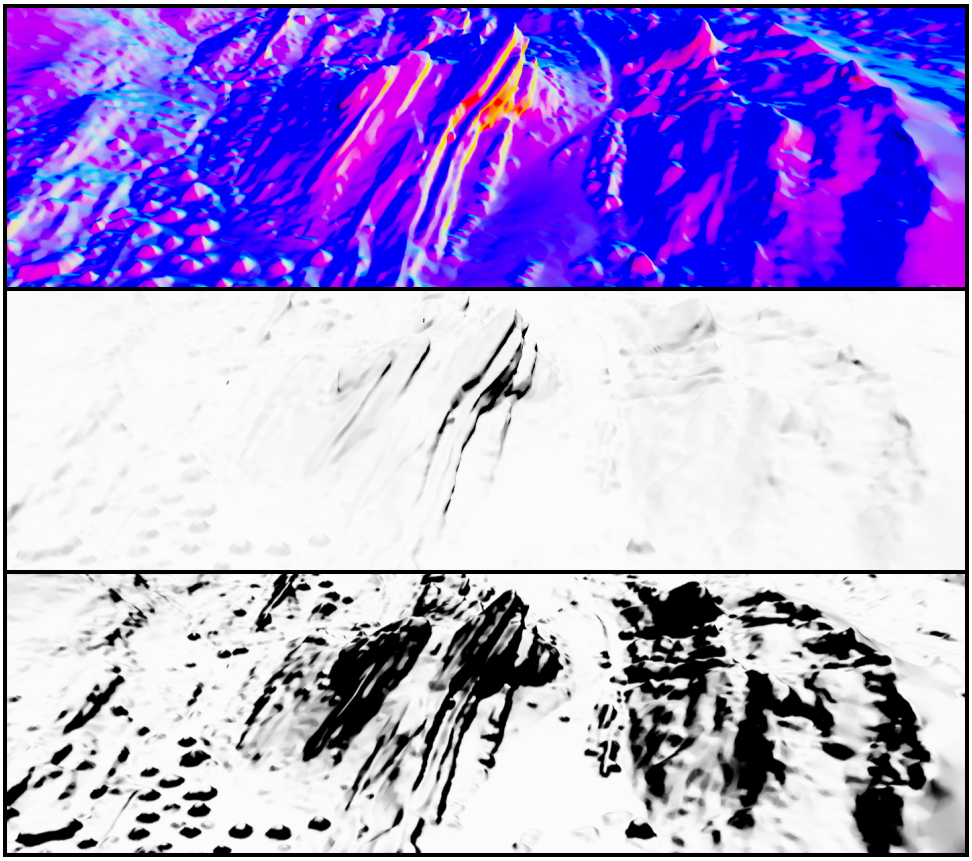


Figure 5.10: Upper: Surface normal vectors of the object used for the environment, where a normal vector $\boldsymbol{n} = [0, 0, 1]^\top \rightarrow [R, G, B]^\top$ indicates a planar horizontal surface. Middle: Green channel isolated, where white is value 1, and black is value 0. Bottom: Mask from the middle with adjusted value-mapping (output from fig. 5.11). This is the normal based mask for separating between two different materials. Note that the images have been enhanced for increased contrast.

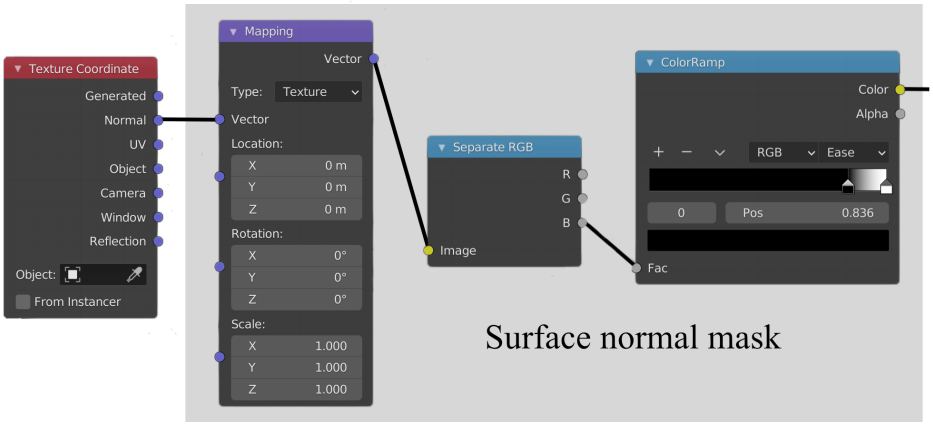


Figure 5.11: Material node setup for generating a normal based texture mask. The blue channel from an object's normal vector, stored as a 3-dimensional "RGB" vector, is extracted. The ColorRamp adjusts the value mapping for increased contrast in the final mask.

Using the surface normal-weighted mask for texture blending: The normal-weighted mask are connected to "MixRGB" nodes, denoted as "Mix_[TextureType]" in fig. 5.12, through the factor input, abbreviated "Fac" in the figure. If $Fac = 0$, then the first color input "Color1" is used as output. Likewise, if $Fac = 1$, the second color input "Color2" is used. The surface normal-weighted mask from fig. 5.11 is a 2D mask for the entire environment, and weighs small patches of the area corresponding to these factor values. Figure 5.13 shows how the final seafloor after the normal based texturing is applied. The blending of the two textures reduces the perceived presence of a repeating pattern.

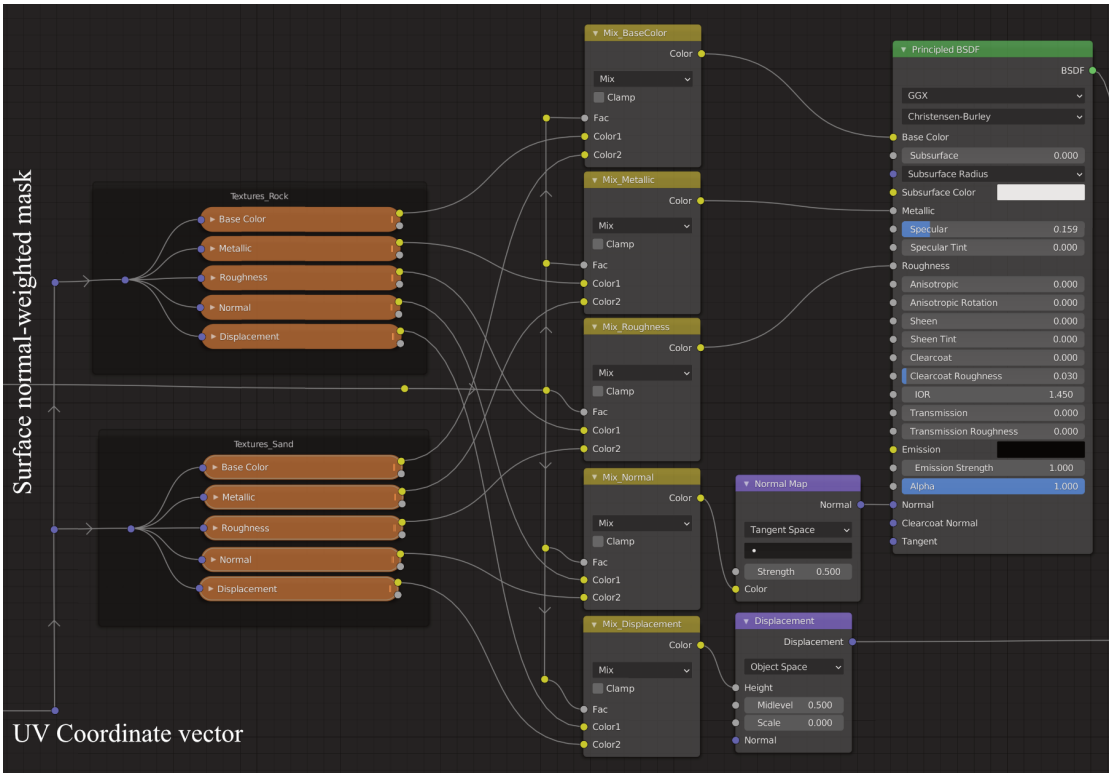


Figure 5.12: Node setup in Blender's material editor for the normal-weighted textures of the sea floor. The collections "Textures_Rock" and "Textures_Sand" both contain base color, metallicity, roughness (inverse reflectivity), normal and displacement (height) images for the corresponding textures. The "Mix_[TextureType]" nodes mixes between the textures based on the surface normal-weighted mask. The "Principled BSDF" node creates the actual material from the different texture inputs. The complete node setup is shown in fig. A.1.

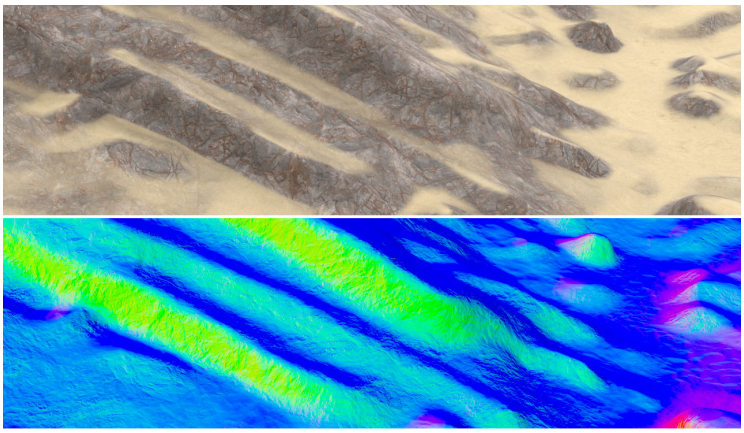


Figure 5.13: Top: Seafloor with normal based texture blending. Bottom: World-space normal image as seen by the camera, where blue areas contain surface normal vectors pointing in the positive z-direction.

5.4.6 Seafloor textures

The textures used on the seafloor has to be complete with image files for base color, roughness (equivalent to inverse reflectivity), metallicity, normal and is possible, height maps (named displacement in Blender). For acquisition of quality textures, the material library Substance Source in the Substance 3D suite[34] is used. This library provides access to hundreds of high-quality photoscanned materials and textures, ensuring that the textures we use properly represents the real-world material.

The main issue of using these textures is that there are little to no textures available from underwater scenes. Therefore we use fairly smooth and wavy sand pattern together with an eroded rock coastal cliff texture, shown in fig. 5.14. In Substance Source, these are named "Wavy Sand Beach 02" and "Eroded Coastal Boulder Rock" respectively. Both textures are photo-scans of real textures, that has been post-processed to allow for seamless tiling.

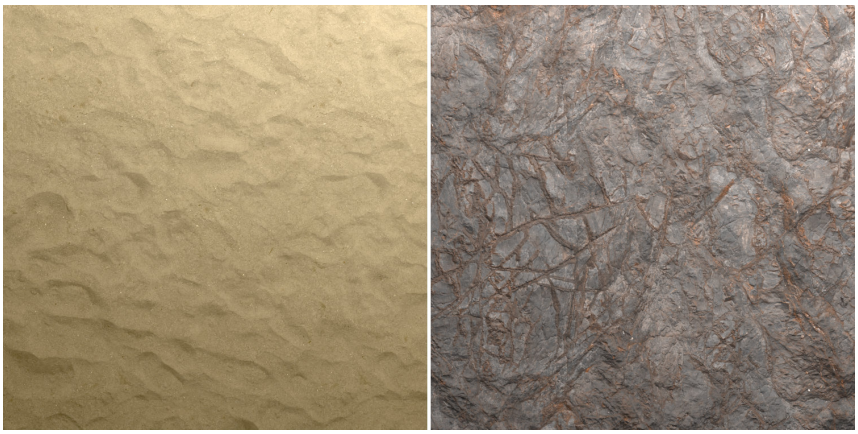


Figure 5.14: Textures used from the Substance Source[34] library. The images are created in Blender Cycles using these textures on a planar surface and a tilted area light.

5.4.7 3D assets

Due to the need for proper edge geometry as introduced in section 5.3.4, it was more efficient to create 3D assets tailored to our needs instead of downloading assets found online. A total of six assets was created, as listed in table 5.1. All pipes, bolts and support structures are made such

that they can be tiled with an array-modifier in Blender. This makes the process of populating a scene time efficient.

The materials used are mostly procedural, with the exception of the concrete for the support pillars and the ArUco markers on the marker platform. A procedural material is created using noise patterns to generate texture, roughness, surface normal and height features. While it requires more effort to set up, they can create these features at a very high resolution for a low memory footprint compared to an image texture with a corresponding level of detail. Most importantly however: by including the location of an object in the noise generation, two identical objects with the same material placed side by side will contain different texture features. When these features are used to generate dirt or rust, this translated into non-repeating high-resolution patterns. These non-repeating patterns are used for rust on the pipes and dirt on the concrete support pillars in fig. 5.15.

The textures for the 3D pipe assets was created following a tutorial[5] for procedural rust, only modified to fit our needs. The paint was made to be procedural with slight roughness variations, and the rust has been modified such that the procedural rust is only applied in a procedural mask created by a larger, more blocky, musgrave noise texture. The final material nodes for the pipe texture is shown in fig. A.2 and the material nodes for the pipe supports is shown in fig. A.3.

Table 5.1: Assets created and used in the simulation environment. Procedural is abbreviated "Pr.".

Asset type	Textures	Dimensions
Pipe, straight	Pr. paint and rust	3m long, 0.5m pipe diameter
Pipe, 90deg bend	Pr. paint and rust	1m long in each bend direction, 0.5m diameter
Pipe support	Concrete texture, Pr. dirt	2m high, 0.8m wide, 0.2m deep
Bolts, pipe joints	Pr. metal and rust	
Bolts, pipe support	Pr. metal and rust	
ArUco markers	ArUco markers, Pr. paint	ArUco marker size (black area): 0.5m x 0.5m Square of 4 ArUco markers: 2.0m x 2.0m

Figure 5.15: Assets listed in table 5.1 viewed with uniform lighting.

5.4.8 Water shader

It is important that the water shader has the properties of a true scattering medium, as outlined in section 5.1. The path-tracing render engine in Blender named Cycles traces paths out of the camera, where light intensity and color values are calculated each time the path hits a surface or a scattering medium and bounces. As such, it can emulate the phenomena discussed in section 5.1 with a sufficient amount of samples per pixel.

[5]Procedural rust tutorial available at: https://www.youtube.com/watch?v=YTMdF6lOOsw

The scattering medium is created by applying a "Principled Volume" volume shader to the world-shader. This "submerges" the entire scene in a uniform scattering volume. This volume contains four parameters of interest; color, density, anisotropy and absorption color. The final parameters for the scattering volume is listed in table 5.2.

Scattering volume color: The volume color determines the color of the scattering volume itself. For a completely clear volume, this color is completely white, where the red, green, blue and alpha (RGBA) channels have an intensity of 1. In order to replicate the visual properties of the reference image in fig. 5.17, a saturated dark blue with $RGBA = (0.014061, 0.146473, 0.34509, 1)$ is used.

Scattering volume density: The density in the scattering volume determines the amount of light attenuation. If the medium is very turbid, then this property is increased. A higher density will make the scattering volume color more intense, which must be taken into consideration. To achieve a sufficient level of light attenuation compared to the reference image in fig. 5.17, a density value of 0.125 was used.

Scattering volume anisotropy: The anisotropy value determines the amount of forward light scattering. Looking back at fig. 5.1, a larger positive anisotropy value is equivalent with the scattering angle ψ covering a smaller interval in the forward direction, where the majority of the light scatters forward. A negative anisotropy value scatters light primarily back in the direction where it came from, and an anisotropy value of zero scatters light uniformly, as illustrated in fig. 5.16.

For our use, the anisotropy value is important for achieving the desired backscattering effect. A lower value will increase the amount of backscattering. We are using an anisotropy value of 0.8, which is lower than that of clean water as we want to include the effect of microscopic suspensoids which are too small to simulate as individual objects.

Scattering volume absorption color: The absorption color emulates the effect where the perceived color of objects further away is changed due to absorption of different wavelengths in the scattering volume. Water absorbs red and green wavelengths to a larger degree than blue, hence objects further away appear blue. Various microscopic particles absorb different wavelengths of light and can further alter the absorption spectrum. In Blender the absorption color works the opposite way around. Instead of defining which colors are absorbed, the color tint given objects further away is defined. Using the reference image in fig. 5.16, it was found that the absorption color $RGBA = (0.0, 0.394458, 1.0, 1)$ provided the desired effect.

Scattering volume limitations: It must be noted that the scattering medium is only an approximation of the real-world. There is no built-in functionality for taking into account water density changes or temperature. Therefore the scattering volume is constructed to be valid only for a smaller region of operations. It is possible to merge different scattering volumes, using 3D fractal noise as a mask, which can open up more complex non-homogeneous volumes in later iterations of the environment.

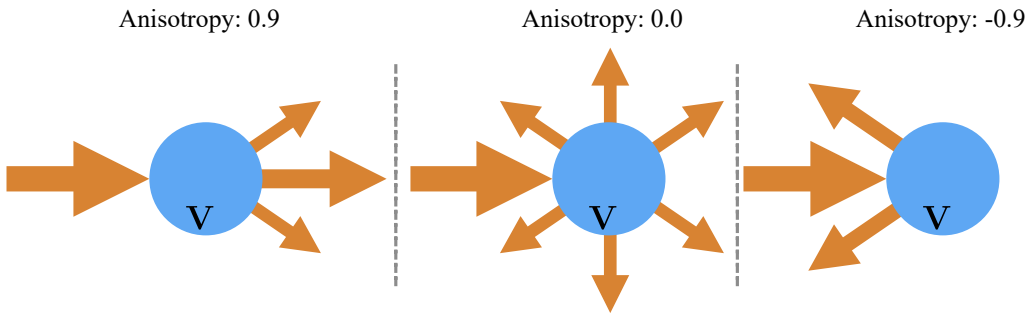


Figure 5.16: Light scattering for three different levels of anisotropy, $a \in (-1, 1)$. Left: Mostly forward light scattering with $a = 0.9$. Middle: Uniform light scattering in all directions with $a = 0$. Right: Backward scattering with $a = -0.9$.

Table 5.2: Scattering volume parameters for a uniform water shader.

Parameter	Value	Description
Color	(0.014061, 0.146473, 0.34509, 1)	(R, G, B, A)
Density	0.125	In range [0, 1]
Anisotropy	0.8	In range (0, 1)
Absorption color	(0.0, 0.394458, 1.0, 1)	(R, G, B, A)

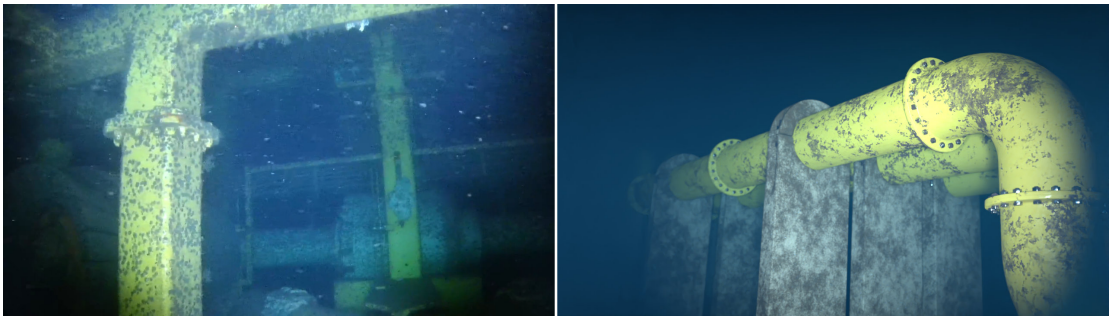


Figure 5.17: Left: Reference footage from NTNUs EELY500 robot provided by [11, 35, 36], recorded at a depth of $90m$ in the Trondheim Fjord. Right: Simulation environment with a water shader designed to closely match the properties of water in the reference footage.

5.5 Camera setup

The front-facing camera is modeled after the intrinsic parameters of the camera present on NTNUs EELY500 robot supplied by Eelume[11]. This camera consists of two parts, the image sensor is a CIS VCC-HD3 camera[37] with a HF3417D-12MPIR wide angle lens[38]. The intrinsic parameters of the front facing camera system is listed in table 5.3, and provides all necessary information to set up the camera parameters in Blender.

Table 5.3: Front facing camera parameters of NTNU's 6m EELY500 robot.

Parameter	Value	Citation
Pixel size	$3.45\mu m \times 3.45\mu m$	[37]
Effective pixels	2064×1544	[37]
Output resolution	1920×1080	[37]
Output format	1080p	[37]
Shutter type	Global Shutter	[37]
Focal length	$3.4mm$	[38]
Focal-stop	1.7	[38]

5.5.1 Auto-exposure

Images taken by a real camera is subject to exposure correction by the camera itself. Exposure correction aims to use the entire value range of an image, while at the same time retaining details in areas of high and low light intensities. It is possible to use methods that naively fills the entire histogram or more sophisticated methods such as estimating a non-linear intensity mapping curve based on the intensity of several regions in the images[39].

Blender does not contain automatic exposure correction, but it is possible to add exposure correction through post-processing on lossless raw data. The raw data from underwater RGB images are exported as 16-bit OpenEXR files where each pixel consists of a 16-bit float for each channel. This way, there is no discretization which is present in other image formats where the intensity is mapped to a predetermined range, i.e. $[0, 255]$ in 8-bit files or $[0, 65535]$ in 16-bit files. Therefore heavy exposure correction with OpenEXR files results in no banding in the histogram, which may occur in images operating with a discretized value range, illustrated in fig. 5.18.

An added benefit gained from storing the raw image data is that various exposure correction algorithms and their impact on computer vision algorithms can be tested. This is important, as demonstrated by [39], exposure correction is a control problem of its own.

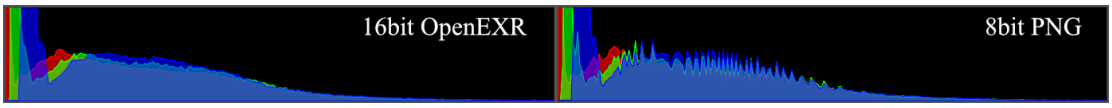


Figure 5.18: Illustration of the banding effect when applying exposure correction post-process, brightening the image. 8bit PNGs are used as a counter example to show the banding effects in the form of sharp peaks and valleys. This effect will be much less severe for 16bit PNG files.

5.5.2 Focus and depth of field

The camera has a limited depth of field which has to be taken into consideration. By default, all rendered images are completely sharp. Adding depth of field in post processing using the compositing system in Blender yields sub-par results. Therefore the active camera in Blender is given DoF parameters to accurately simulate this effect. The depth of field is determined by the focal length, distance from the lens to the focal point in front of the sensor, and the Focal-stop (F-stop) value of the lens. The F-stop value is the ratio found when dividing the focal length by the diameter of the aperture opening in the lens. A smaller F-stop value will cause a greater level of blur outside the forward focus distance. The focal length and F-stop used is listed in table 5.3

5.5.3 Active sensor size

Based on the provided pixel size and output resolution in table 5.3, the active sensor size is found:

$$sensor\_width = pixel\_size\_width \cdot output\_resolution\_width = 3.45\mu m \cdot 1920 = 6.624mm$$
$$sensor\_width = pixel\_size\_height \cdot output\_resolution\_height = 3.45\mu m \cdot 1080 = 3.726mm$$

The derived sensor size of $6.624mm \times 3.726mm$ is corresponding to the sensor size of the images with resolution 1920×1080 from the front-facing camera of the EELY500 robot. Due to computational complexity images with a resolution of 1280×720 pixels are used, as they contain less than half of the pixels of a 1920×1080 image and nearly halves the rendering time requirement. Shrinking the sensor-size to just contain a sensor array of 1280×720 without changing the lens parameters, results in the image zooming in - effectively cropping the original 1920×1080 image. Instead, the parameters for a 1920×1080 image is used to ensure that the field-of-view remains constant with a render-resolution set to 1280×720. In practice, this means that the output is essentially 1920×1080 pixel images re-scaled to 1280×720 pixels.

5.5.4 Downward facing camera

The downward facing camera, a Sony FCB-EV7520 camera[40], is not included in the initial simulation environment, and the intrinsic parameters are not derived here. However, the same principles apply for this camera.

5.6 Render setup

In order to produce the different images needed for the sequences, Blender's rendering settings must be set up properly. Settings related to the needed render layers, which are direct outputs from the layered render process, and the settings used in Cycles and Eevee is discussed in this subsection.

5.6.1 Render passes

A computer generated image in Blender is a composite of several different render passes, where each pass contribute with different types of information. By default, the final image, the alpha channel and the depth-buffer, hereafter named the Z-buffer, is made available. All render passes needed in our setup is listed in table 5.4.

Table 5.4: Overview of the different render passes that are enabled for image export in our setup. Names under 'Render pass' is corresponding to the names used in Blender's node-based image compositing system.

Render pass	Description
Image	Denoised image, default output
Noisy Image	Noisy image, no denoiser applied
Depth	Image Z-buffer, depth to each pixel in scene units
Normal	World-space surface normals for each pixel
Shadow	Mask containing only light from light-sources
VolumeDir	Direct volumetric lighting present in the image

5.6.2 Cycles settings

Cycles is the path-traced renderer i Blender, and can utilize either GPUs or CPUs to render images. Path-tracing uses Monte Carlo simulation to repeatedly calculate the path of a ray out from a single pixel, the ray picks up light intensity and color information from each bounce. The average of all the bounces is then used as the final pixel intensity and color. Settings for number of path tracing integrator, samples per pixel and the tile size is listed in table 5.5. The integrator determines the bounce characteristics of a path-traced ray. Pure path tracing follows a single ray until it terminates. Branched path tracing is an alternative, where the ray splits into several rays upon a bounce for increased sample accuracy. This is currently not supported with GPU rendering, but it is a strong tool to consider if CPU rendering is used. As we utilze GPUs for our task, leveraging the GPGPU cluser IDUN[41], pure path tracing is used. In order to optimize the render times for each image, a maximum number of bounces per ray is set as shown in table 5.6.

The tile size determines the size of a tile in the image to be rendered, which is delegated to an available device. One tile is assigned per GPU, where all GPU cores work in parallel on a single tile, hence the tile size is relatively large. The tile size is chosen as

$$(TileX,\ TileY) \leq (256,\ 256)$$

where the total tiles in the image is divisible by the number of available GPUs. This is done to avoid idling, where GPU0 may render the final tile, while GPU1 idles and waits for GPU0 to finish.

During CPU rendering, each CPU thread is assigned a tile. Hence the tile size is chosen to be approximately (16, 16), which is the default value.

Table 5.5: Cycles render pixel sampling settings.

Setting		Value	Description
Integrator		Path tracing	Pure path tracing, no branching
Samples	Quality	375	Path-traced samples per pixel
Samples	Fast	20	Path-traced samples per pixel
TileX	GPU	256	Tile size width, GPU rendering
TileY	GPU	180	Tile size height, GPU rendering
TileX	CPU	16	Tile size width, CPU rendering
TileY	CPU	16	Tile size width, CPU rendering

Table 5.6: Maximum number of bounces for each pure path-traced ray in cycles.

Bounce type	Number of bounces
Total	16
Diffuse	4
Glossy	4
Transparency	2
Transmission	2
Volume	14

5.6.3 Depth-, normal- and uniformly lit RGB images

Cycles is used to generate depth-, normal and uniformly lit RGB images, as the quality of the world-space normals are higher than that of Blenders hybrid rasterization render engine Eevee. Cycles is set to use the "Fast" preset for samples per pixel in table 5.5. Using the "Fast" preset, the render time per image using Cycles is comparable to that of Eevee when time for scene setup is included, resulting in approximately 10 seconds per render.

5.7 Automated image rendering

In order to expedite the water setup and rendering process, an automated render script was created. The idea behind the script is to use the dictionaries in table 5.7 as inputs. These provide all the necessary information for the setup of:

- Water parameters
- Uniform light color and brightness
- Render parameters
- Camera parameters for depth of field effects
- Output folder- and name structure

Due to limitations in Blender's Python API the water and uniform light world shaders must be setup each time the script is run. This is due to the inactive world shader is being removed if the Blender process is stopped in the middle of a render job, which is very likely to happen if you only can render in batches of a given time duration. While bugs for other settings has not been encountered yet, all other parameters are also set each time the script is run as a precaution - overwriting the settings in the Blender file. Therefore it is important that the global variables in the automated script is updated with the most recent values from the Blender file prior to runtime.

The automated render script contains a lot of verbose code to set all the required parameters. The main functionality of the program is shown in algorithm 2. Functions for rendering images of the underwater scene is shown in algorithm 3, and uniformly lit scene in algorithm 4.

5.7.1 Compositor node setups

The compositor setup will mainly be shown through images as it is trivial, with one exception. The underwater RGB image will not use the denoised "Image" output, as that denoiser takes the surface albedo and normals into account. For parts of the image occluded by the scattering medium, this will introduce denoising artifacts where details of the environment is "stamped" on to parts of the image which should be without details. Instead, only the "Noisy Image" output is used as input to a "Denoise" node which removes the noise in the image, yet does not introduce any significant artifacts. The three different node setups used are shown in fig. A.4. It is important to note that the three different node structures are automatically generated through code for every rendered frame in order to extract the various types of image data.

Table 5.7: Parameters used to automate uniform light, water and render initialization.

Dictionary name	Key	Value	Description
PARAMS_UNIFORM_LIGHT			
	Color	(1,1,1,1)	RGBA values
	Strength	1	Brightness, range [0,1]
PARAMS_WATER			
	Color	(R,G,B,A)	RGBA values of the scattering volume color
	Density	0.1	Determines light attenuation, range [0,1]
	Anisotropy	0.8	Forward light scattering, range (-1,1)
	Absorption Color	(R,G,B,A)	RGBA values of the volume absorption color
PARAMS_RENDER			
	ResolutionX	1280	With of the image in pixels
	ResolutionY	720	Height of the image in pixels
	ResolutionScaling	1.0	Resolution scaling factor, 1.0 $\Leftrightarrow$ 100%
	CyclesRenderDevice	GPU	If GPU or CPU is used during rendering
	CyclesSamplesCPU	30	Samples per pixel, CPU rendering
	CyclesTileXCPU	16	Tile size width, CPU rendering
	CyclesTileYCPU	16	Tile size height, CPU rendering
	CyclesSamplesGPU	375	Samples per pixel, GPU rendering, Quality preset
	CyclesSamplesGPU	20	Samples per pixel, GPU rendering, Fast preset
	CyclesTileXGPU	256	Tile size width, GPU rendering
	CyclesTileYGPU	256	Tile size height, GPU rendering

Algorithm 2: Automatic render script

Output: Image output, timestamped filenames

```
FrameStartGlobal ← 0                                  // Starting frame of the entire sequence
OverrideDefaultImageNumbering()                       // Allow for custom image enumeration
SetRenderParameters(PARAMS_RENDER)
CreateUniformLightShader(PARAMS_UNIFORM_LIGHT)
CreateWaterShader(PARAMS_WATER)
Timestamps ← Timestamps from file with Blender camera poses
ImageNames ← Data structure of filenames
ExportTimestampedFilenames(Timestamps, ImageNames)
if RestartRender == TRUE then
    SetCurrentFrame(FrameStartGlobal)
FrameEnd ← Last frame of active Blender scene timeline
for FrameCurrent in [FrameStart, FrameEnd + 1] do
    RenderWater()
    RenderUniformLighting()
    SetSceneFrameCurrent(FrameCurrent)
    if FrameCurrent ≤ FrameEnd then
        SetSceneFrameStart(FrameCurrent)              // Set the animation start to FrameCurrent
    SaveBlenderFile()                                 // Save blender file for automatic resumption of FrameCurrent
```

Algorithm 3: RenderWater(): Render images of the underwater scene, shadow maps, and volumetric scattering with path-tracing in Cycles.

```
Activate water shader
Enable vehicle lights
Delete all compositor nodes
Setup and create compositor nodes for water output images in Cycles
Set Cycles to "Quality" samples per pixel preset.
Set color management for sRGB images
Render images and write to disk
```

Algorithm 4: RenderUniformLighting(): Render uniformly lit images, depth- and normal maps with Cycles.

```
Activate world shader with uniform light
Disable vehicle lights
Delete all compositor nodes
Setup and create compositor nodes for depth- and normal images
Set Cycles to "Fast" samples per pixel preset.
Set color management for non-color images
Render images and write to disk
Delete all compositor nodes
Setup and create compositor nodes for RGB images
Set color management for sRGB images
Render images and write to disk
```

6 Discussion, shortcomings and future work

The thesis is split into three distinct parts: Derivation of the vehicle control system, generation of synthetic IMU measurements and finally setup of the simulation environments. This section is split into three separate corresponding parts, and is discussing the intention behind each section as well as potential shortcomings and future work.

6.1 Vehicle control system

Section 2 introduced a control system which uses waypoints consisting of desired vehicle poses as a reference for the generated trajectory. The aim of this approach is to reduce the human effort needed to create sequences with physically correct motion. Using this system, the user sets a number of waypoints creating a desired trajectory. Then the VCS creates a physically correct trajectory from these waypoints.

In its current state, the vehicle control system, VCS, is dependent on careful placement and orientation of waypoints in order to produce motion which is to be expected from an underwater vehicle. The reason is the simplicity of the vehicle model, where each degree of freedom is controlled independently with no dependencies on other states and exerts an input directly along or around each axis. This is a significant simplification of a standard underwater remotely operated vehicle (ROV) or Eelume's EELY500 robot which is shown in fig. 6.1. Therefore the vehicle model itself is a weakness of the current implementation. In the next iteration of the simulation framework, this model should be replaced with a more complicated model which is correct with respect to an existing underwater vehicle. Not only will this result in motion corresponding to the vehicle WP1 and WP2 of the AROS project will be using in live tests, but it will also reduce the probability of human error during waypoint creation.

A different approach which allows for greater movement flexibility is to create trajectories by manually piloting the vehicle, such that the user sets a velocity reference for the VCS using a joystick and allowing the VCS to set the inputs accordingly. This approach was used in the creation of the COnGRATS[7] dataset with great success, where Unity[42] served as an interface for creating trajectories with direct user input.

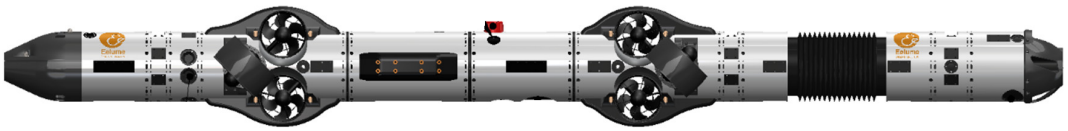


Figure 6.1: Image of the EELY500 robot provided by Eelume[11]. Note that the modules attached to the robot in this image is for illustration purposes and is not a 1:1 representation of NTNU's EELY500 robot.

6.2 Generating IMU measurements

6.2.1 Coordinate transformations

In section 3.2.1 there was briefly mentioned a discrepancy in the transformations used to transform data in the vehicle frame $\boldsymbol{v}$ to the sensor frame $\boldsymbol{s}$. More precisely, this discrepancy revolves around eqs. (3.11) to (3.14) and the corresponding figures in figs. 3.2 to 3.4.

Rotation and translation frame of reference discrepancy: The rotation $\boldsymbol{R}_{sv}$ in eq. (3.12) rotates from the source frame $\boldsymbol{v}$ to the destination frame $\boldsymbol{s}$, and this rotation is expressed with respect to the *fixed* frame of $\boldsymbol{s}$. Deviating from the notation in [16], $\boldsymbol{R}_{sv}$ can be rewritten as $\boldsymbol{R}^{s}_{sv}$, where the superscript denotes the frame of reference for the rotation. Meanwhile the translation

in eq. (3.13) is given as $\boldsymbol{t}_{sv}^{v}$, which is the transformation from frame $\boldsymbol{v}$ to the destination frame $\boldsymbol{s}$, expressed in frame $\boldsymbol{v}$. Rewriting $\boldsymbol{T}_{sv}$ in eq. (3.14), we now have to possibilities; use either frame $\boldsymbol{v}$ or frame $\boldsymbol{s}$ as the reference for the transformations.

Using the vehicle frame as reference: First, let us examine using the vehicle frame $\boldsymbol{v}$ as the frame of reference. Now we have:

$$\boldsymbol{T}_{sv}^{v} = \begin{bmatrix} \boldsymbol{R}_{sv}^{v} & \boldsymbol{t}_{sv}^{v} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} \tag{6.1}$$

where $\boldsymbol{t}_{sv}^{v}$ is already given in eq. (3.13), and $\boldsymbol{R}_{sv}^{v}$ is expressed as

$$\boldsymbol{R}_{sv}^{v} = \boldsymbol{R}_{y}(-\frac{\pi}{2})\boldsymbol{R}_{x}(\frac{\pi}{2}) \tag{6.2}$$

where the order of rotations is visualized in fig. 6.2 and the resulting IMU measurements are plotted in fig. 6.3.

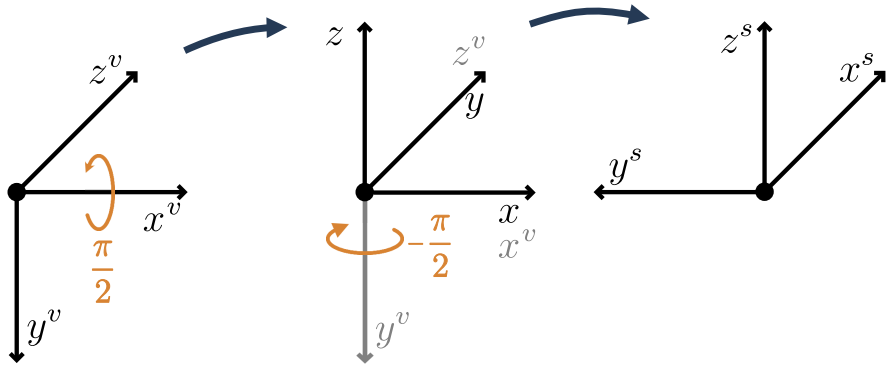


Figure 6.2: Visualization of $\boldsymbol{R}_{sv}^{v}$, the rotation from frame $\boldsymbol{v}$ to frame $\boldsymbol{s}$ expressed in frame $\boldsymbol{v}$. Rotation order is left to right, and the corresponding equation is found in eq. (6.2).

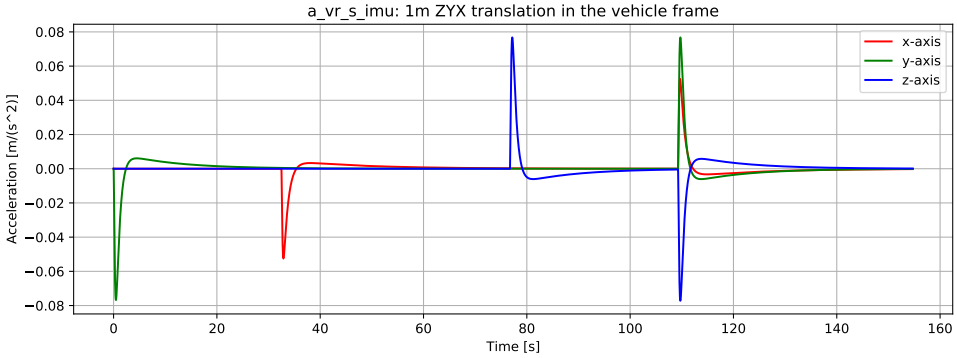


Figure 6.3: IMU acceleration measurements when using the transformation matrix $\boldsymbol{T}_{sv}^{v}$ from eq. (6.1). The actual translation and coordinate frames for the IMU is shown in fig. 3.7. Note: The gravity-vector is not included for visualization purposes due to the low acceleration magnitude of the visualized data.

Using the sensor frame as reference: Now, let us examine the case where the sensor frame $\boldsymbol{s}$ is the frame of reference. We already have $\boldsymbol{R}_{sv}^{s} = \boldsymbol{R}_{sv}$ from eq. (3.12). The translation vector $\boldsymbol{t}_{sv}^{v}$ is expressed in the vehicle frame $\boldsymbol{v}$, but it must be converted to the sensor frame $\boldsymbol{s}$. This is done by reversing its direction and rotating it into the sensor frame. Hence we have that

$$\boldsymbol{t}_{sv}^{s} = -\boldsymbol{R}_{sv}^{s}\boldsymbol{t}_{sv}^{v}$$

and it is now possible to express $\boldsymbol{T}_{sv}^{s}$ as:

$$\boldsymbol{T}_{sv}^{s} = \begin{bmatrix} \boldsymbol{R}_{sv}^{s} & \boldsymbol{t}_{sv}^{s} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_{sv}^{s} & -\boldsymbol{R}_{sv}^{s}\boldsymbol{t}_{sv}^{v} \\ \boldsymbol{0}_{1,3} & 1 \end{bmatrix} \tag{6.3}$$

After applying this transformation, the acceleration measurements for linear motion without rotation is correct, as shown in fig. 6.4. However, when viewing the plots where the vehicle is rotated with a negative 180-degree rotation clockwise around the Y-axis in the vehicle frame $\boldsymbol{v}$, the plot in fig. 6.5 does not correspond to the actual motion of the IMU when $\boldsymbol{t}_{sv}^{v} = [0, 0, -0.5]^{\top}$ is used. For reference, the plots in fig. 3.6 correspond to the IMU at the intended placement. Whereas the acceleration in fig. 6.5 corresponds to a different IMU placement.

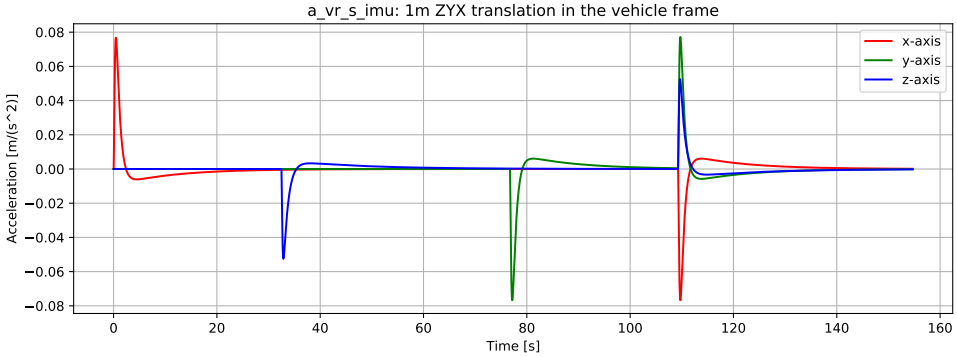


Figure 6.4: IMU acceleration measurements when using the transformation matrix $\boldsymbol{T}_{sv}^{s}$ from eq. (6.3). The actual translation and coordinate frames for the IMU is shown in fig. 3.7. Note: The gravity-vector is not included for visualization purposes due to the low acceleration magnitude of the visualized data.

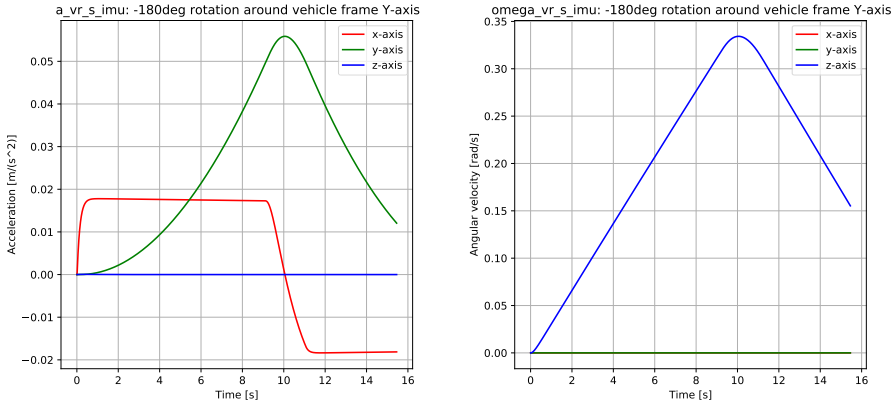


Figure 6.5: IMU measurements for a negative 180-degree rotation (clockwise following the right-hand rule) around the Y-axis in the vehicle frame $\boldsymbol{v}$. The orientation of the IMU sensor frame $\boldsymbol{s}$ relative to $\boldsymbol{v}$ and the rotation relative to frame $\boldsymbol{v}$ is shown in fig. 3.7. Note: The gravity-vector is not included for visualization purposes due to the low acceleration magnitude of the visualized data.

Discussion of the observations above: By using the same transformation convention as in eq. (3.7) for finding $\boldsymbol{T}_{rw}$, it would seem reasonable that using $\boldsymbol{T}_{sv}^{v}$ from eq. (6.1) should provide correct results. However, as shown previously, this is not the case. As both eqs. (6.1) and (6.3) provide wrong results, there is an error somewhere in my implementation or mathematical derivations of the coordinate transformations. Where this error is located is unknown at the time of writing. Due to the presence of this discrepancy, it is even more important that the correctness of the IMU

data is verified by calculating the vehicle's pose and trajectory from the IMU data alone, and compare it to the ground-truth trajectory. Using plots for verification is subject to human error, likewise human error is a factor in this proposed verification method during implementation of the testing framework. However cross-checking the verification results from these methods should be done, as that could quickly point out potential flaws which are difficult to spot with just using one method.

6.2.2 Sensor noise augmentation

The derivation of synthetic measurements are introduced in section 3. As the data used to generate these measurements is noise-free data straight from the VCS, the IMU data does not contain any noise. At the time of writing, whether or not noise-free IMU and depth data that can be augmented with noise by the end user, or data with pre-augmented noise should be included is still a point of discussion. Allowing the end user to augment the data themselves would allow them to customize the data to match the noise characteristics of the equipment they will be using. However, it would make it difficult to compare results from VSLAM systems tested on the dataset using IMU and depth data, as the noise levels will vary between different users. This could lower the benchmark value of the dataset due to the inconsistency in the IMU data used for testing in published results using this dataset.

Publishing the dataset with IMU data augmented with noise will ensure that all published results using this dataset is using IMU measurements with identical noise. This will keep the data used in published results consistent, increasing the validity of the dataset as a benchmark. However, this approach will then limit the customizability of the data for the user.

While this topic is still undecided, we are inclined to publish the dataset with sensor data augmented with noise. The reason is that providing a dataset which can serve as a benchmark will provide useful for comparing different VSLAM approaches if our dataset is a part of the datasets used for testing.

6.3 Simulation settings and environment

6.3.1 Scattering properties of water

The most important aspect of the simulation environment is the scattering properties of the water volume, as this is the main distinction between an above-water and underwater scene. The render engine Cycles in Blender uses path-tracing to generate imagery. However, in practice it is an approximation to the real world, as already discussed under scattering volume limitations in section 5.4.8. In short, the parameters set for the scattering volume are approximations, set by using a reference image and a general range of acceptable values for the parameters involved. However, these simplifications allow for lower rendering-times compared to a very accurate physically based model. Given a high enough number of samples per pixel, the Monte Carlo sampling process in Cycles will return a good approximation of the actual color and value of a given pixel. As we may require several thousand images for a single sequence, reasonable render-times are a necessity as hardware accessibility for extended periods of time is limited.

6.3.2 Scattering properties of water using rasterized render methods

Rasterization is a render technique commonly used in games due to the very low computational complexity compared to path-tracing which is used by the Eevee renderer in Blender. Briefly explained, rasterization maps the 3D objects to a 2D plane and changes the pixel color and value intensity based on the lights in the scene. While it is significantly faster, it is not feasible to create physical correct light scattering akin to what is possible through Cycles. Without proper scene setup effects such as ambient illumination and light reflections are missing. It is possible to enable screen-space reflections and screen-space global illumination (the latter through a community ad-

don), however these effects are based on what is visible in the camera. If scene geometry which is reflecting light exits the camera frame, then the reflected light from that geometry is completely removed from the scene, resulting in sudden illumination changes. Due to the importance of lighting in computer vision algorithms, this inconsistency is unacceptable for use in benchmark sequences.

6.3.3 Marine growth

Another point of interest with respect to accurately replicate underwater scenery is that underwater structures are prone to be covered in marine growth, i.e. barnacles, algae or other growths protruding from the surface of an object as seen in fig. 6.6. This is not currently modeled on the objects in the simulation environment, and could be included in the future. However, objects such as vegetation growth that grow to several centimeters in length has to be modeled as separate objects or as particle systems (in essence as modified hair). When applied to a large surface in a reasonable density, this has a significant impact on the required VRAM[6] in the GPUs used to render the images. If the VRAM budget is limited, this may be more feasible to use on smaller and more specialized "inspection scenes", where the simulation environment only contains one structure of interest.

Figure 6.6: Marine growth present on a subsea structure. Footage from NTNU's EELY500 robot [11, 35, 36].

6.3.4 Marine snow

Marine snow is not currently included in the current environment. In some scenarios, marine snow can be the dominating visual features in an image, as illustrated in fig. 6.7. This poses a serious challenge for computer vision algorithms due to their high contrast and motion when the vehicle is moving. The SLAM framework developed by the AROS group will have to work with these type of images at a later stage in the development, and as such the simulation framework should be expanded to allow for generating synthetic images with various levels of marine snow.

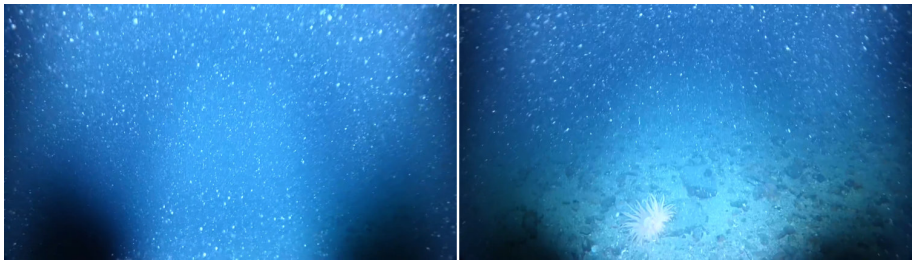


Figure 6.7: Presence of heavy marine snow in an underwater environment. Footage from NTNU's EELY500 robot [11, 35, 36].

[6]Video Random Access Memory, memory on-board the graphics card directly accessed by the GPU.

6.4 Dataset availability and render times

The dataset will not be ready at the time of finishing this thesis due to the time consuming renders required for proper imagery. Using Blender 2.92 with the OptiX Cycles render back-end together with an Nvidia RTX 3080 GPU and a 16core AMD R9 3950X CPU, an image rendered on the GPU with the settings from table 5.5, a resolution of 1280×720 and depth-of-field parameters as listed in table 5.3 takes 1 minute and 7 seconds to complete.

Our resources for rendering the sequence is primarily the AIRLARON[43] project's server, which has $2x$ Nvidia RTX 2080Ti GPUs available, and the NTNU IDUN/EPIC computing cluster[41], which has more than 70 nodes and 90 GPGPUs. Every node is equipped with two Intel Xeon cores, a minimum of 128GB main memory, and they are connected to an Infiniband network. The storage consists of two storage arrays and a Lustre parallel distributed file system. From IDUN we asked for 4 GPGPUs, where we were allocated either an Nvidia P100 or V100 GPU. Both come equipped with 16GB of VRAM. Generating the images for the dataset will be done over the summer when the load on these services are lower.

6.5 Future articles

Two articles are planned to be created from this thesis. First is the simulation benchmark paper, detailing how to use the dataset and what information is included. The second paper will explain the implementation of the simulation framework and how the data was generated. The outline for these papers are included as attachments to the final thesis. It is important to note that these papers are currently only drafts, and their content are subject to change before the respective submission deadlines.

6.5.1 Benchmark paper

This paper will include all relevant information to allow any user to properly use the data included in the dataset. It will give an overview of the vehicle and the coordinate frames used by the camera, vehicle and sensor. Then, the sensor data, image data and ground truth data will be explained in detail, detailing what the data is and how it is structured in the included files. Finally we will give a short overview of the underwater environment, and discuss the model used and the properties of the scattering medium used to simulate the water in the scene. This paper is planned to be submitted to the 2022 IEEE International Conference on Robotics and Automation, ICRA[44].

6.5.2 Simulation backend paper

This paper delves into detail of how the data in the benchmark paper was created. The vehicle model is introduced, followed by an explanation of the vehicle control system and the waypoint handover system. Then a in-depth explanation of the sensor data generation follows, before a recap of the included simulation data is given. Finally the simulation environment is discussed, where the environment requirements and setup, as well as the render settings are discussed. This paper is planned to be submitted to the 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS[45]

7 Conclusion

I have with this thesis created a framework for creating benchmarks image sequences accompanied with sensor and ground truth data. The framework takes trajectory waypoints from Blender and uses a control system complete with a vehicle model to generate a physically correct trajectory. This trajectory is then used to generate sensor data and it is imported back into Blender to move the vehicle in the simulation environment.

The benchmark sequences created with this framework is intended to be published such that they can be used by the underwater community at large for development of algorithms for underwater computer vision, egomotion estimation and environmental awareness. At the time of writing, no other synthetic benchmark sequence possesses physically correct path-traced images combined with simulated IMU data. As a result, we aim to publish two articles based on this thesis. The first is a short paper detailing the benchmark and included data, which we aim to submit to the 2022 IEEE International Conference on Robotics and Automation, ICRA[44]. The journal paper detailing the simulation framework, including the trajectory generation using the vehicle control system and IMU data generation, is planned for submission to the 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS[45].

Bibliography

[1] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[2] P. Koschorrek, T. Piccini, P. Öberg, M. Felsberg, L. Nielsen, and R. Mester. A multi-sensor traffic scene dataset with omnidirectional video. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 727–734, 2013.

[3] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10):1157–1163, 2016.

[4] Maxime Ferrera, Vincent Creuze, Julien Moras, and Pauline Trouvé-Peloux. Aqualoc: An underwater dataset for visual–inertial–pressure localization. *The International Journal of Robotics Research*, 38(14):1549–1559, 2019.

[5] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[6] Angelos Mallios, Pere Ridao, David Ribas, Marc Carreras, and Richard Camilli. Toward autonomous exploration in confined underwater environments. *Journal of Field Robotics*, 33(7):994–1012, 2016.

[7] Daniel Biedermann, Matthias Ochs, and Rudolf Mester. Evaluating visual adas components on the congrats dataset. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 986–991, 2016.

[8] Mario Prats, Javier Perez, J Javier Fernandez, and Pedro J Sanz. An open source tool for simulation and supervision of underwater intervention missions. In *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*, pages 2577–2582. IEEE, 2012.

[9] Amanda C Duarte, Guilherme B Zaffari, Rômulo Thiago S da Rosa, Lucas M Longaray, Paulo Drews, and Silvia SC Botelho. Towards comparison of underwater slam methods: An open dataset collection. In *OCEANS 2016 MTS/IEEE Monterey*, pages 1–5. IEEE, 2016.

[10] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Blender Institute, Amsterdam, 2021.

[11] Eelume, Ture-Fronczek-Munter, and Pål Liljebäck. Eelume. https://eelume.com, 2020.

[12] Michael E Mortenson. Mathematics for computer graphics applications. page 264, 1999.

[13] M. Elimelech, J. Gregory, X. Jia, and R.A. Williams. Chapter 4 - colloidal hydrodynamics and transport. In M. Elimelech, J. Gregory, X. Jia, and R.A. Williams, editors, *Particle Deposition Aggregation*, pages 68–109. Butterworth-Heinemann, Woburn, 1995.

[14] Joseph Kestin, Mordechai Sokolov, and William A Wakeham. Viscosity of liquid water in the range- 8 c to 150 c. *Journal of Physical and Chemical Reference Data*, 7(3):941–948, 1978.

[15] California Institute of Technology. Python control systems library. https://python-control.readthedocs.io/en/0.9.0/, 2021.

[16] Timothy D Barfoot. State estimation for robotics. 2021.

[17] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. Using inertial sensors for position and orientation estimation. *Foundations and Trends® in Signal Processing*, 11(1-2):1–153, 2017.

[18] Paul Bovbel. Conventions for imu sensor drivers. https://www.ros.org/reps/rep-0145.html, 2015.

[19] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[20] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[21] Trym Haavardsholm. A handbook in visual slam. 2021.

[22] M. Day. Extracting euler angles from a rotation matrix. https://d3cw3dd2w32x2b.cloudfront.net/wp-content/uploads/2012/07/euler-angles1.pdf.

[23] P. Corke. *Robotics, Vision and Control*. 2017.

[24] Nirmal Kj, A. Sreejith, Joice Mathew, Mayuresh Sarpotdar, Ambily Suresh, Ajin Prakash, Margarita Safonova, and Jayant Murthy. Noise modeling and analysis of an imu-based attitude sensor: improvement of performance by filtering and sensor fusion. page 99126W, 07 2016.

[25] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[26] Curtis Mobley, Emmanuel Boss, and Collin Roesler. Ocean optics web book. https://oceanopticsbook.info/, 2021.

[27] Mohit Gupta, Srinivasa G. Narasimhan, and Yoav Y. Schechner. On controlling light transport in poor visibility environments. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4319–4326, 2008.

[28] Mark Sheinin and Yoav Y. Schechner. The next best underwater view. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3764–3773, 2016.

[29] Creative Commons. Attribution 4.0 international. https://creativecommons.org/licenses/by/4.0/legalcode, 2021.

[30] Austin. Beaulier. Vasquez rocks (photogrammetry). https://sketchfab.com/3d-models/vasquez-rocks-photogrammetry-79a310e006fa406a8291319bf6f9c9e7, 2020.

[31] Fabrice Robinet, Rémi Arnaud, Tony Parisi, and Patrick Cozzi. gltf: Designing an open-standard runtime asset format. *GPU Pro*, 5:375–392, 2014.

[32] Wavefront Technologies. Object files (.obj), appendix b1 of the advanced visualizer software version 3.0 manual. https://coefs.uncc.edu/arwillis/programming-and-documentation/object-files-obj/ — https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml, 1992.

[33] Pixologic. *Zbrush - The all-in-one digital sculpting solution designed for pursuit of art*. Pixologic, 2021.

[34] Adobe. Substance 3d suite. https://www.substance3d.com, 2021.

[35] NTNU AMOS and Asgeir Sørensen. Ntnu amos. https://www.ntnu.edu/amos, 2020.

[36] NTNU AURlab and Martin Ludvigsen. Ntnu aurlab. https://www.ntnu.edu/aur-lab, 2020.

[37] CIS Corporation. Cis vcc-hd3, 1920x1080, 60p, full hd camera. https://www.ciscorp.co.jp/product/detail$_e$n.html|https : //aegis − elec.com/cis − vcc − hd3 − dcc − hd3 − 1920x1080 − 60p − full − hd − camera.html, 2021.05.20.

[38] Hikvision. Hf3417d-12mpir fixed focal dc auto iris 12mp ir lens. https://www.sourcesecurity.com/hikvision-hf3417d-12mpir-cctv-camera-lens-technical-details.html, 2021.05.20.

[39] L. Yuan and J. Sun. Automatic exposure correction of consumer photographs. In *European Conference on Computer Vision*, pages 771–785. Springer, 2012.

[40] Sony. Fcb-ev7520 full hd 30x colour camera block. https://www.image-sensing-solutions.eu/FCB-EV7520.html — https://www.image-sensing-solutions.eu/FCB-EV$_S ERIES_1 1 - 15_H R_W DR.pdf$, 2021.05.20.

[41] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. Epic: An energy-efficient, high-performance gpgpu computing research infrastructure, 2020.

[42] Unity Technologies. Unity game engine. https://unity3d.com/, 2021.

[43] Aya Saad, Annette Stahl, Andreas Våge, Emlyn Davies, Tor Nordam, Nicole Aberle, Martin Ludvigsen, Geir Johansen, João Sousa, and journal = Oceanography year = 2020 month = September volume = 33 url = https://doi.org/10.5670/oceanog.2020.307 Kanna Rajan, title = Advancing Ocean Observation with an AI-Driven Mobile Robotic Explorer.

[44] ICRA. Ieee international conference on robotics and automation. https://icra2022.org, 2022.

[45] IROS. Ieee/rsj international conference on intelligent robots and systems. https://iros2022.org, 2022.

Appendix

A Simulation environment

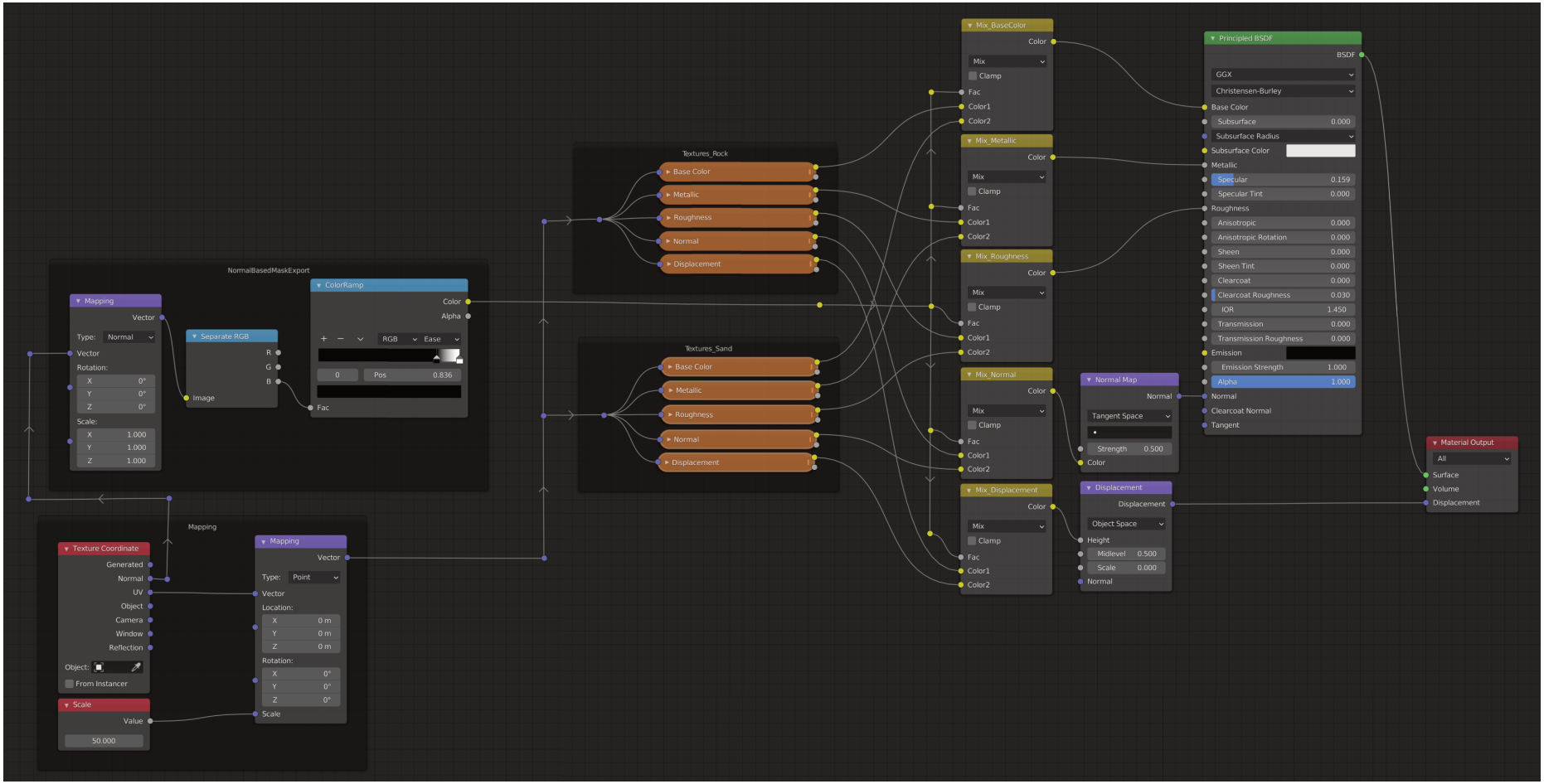


Figure A.1: Complete node setup in Blender's material editor for the normal-weighted textures of the sea floor.

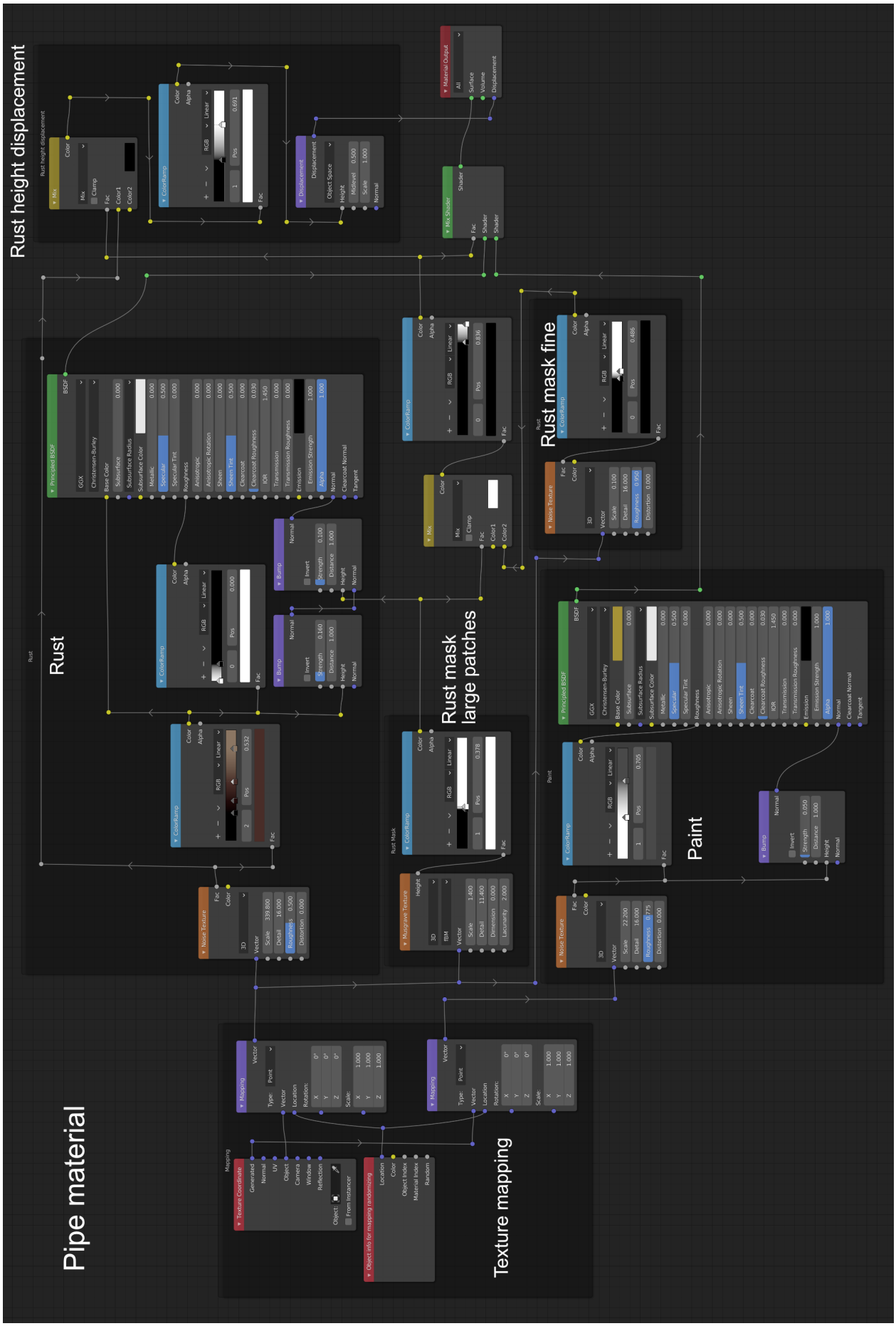


Figure A.2: Blender node tree of the procedural paint and rust material as seen in fig. 5.15.

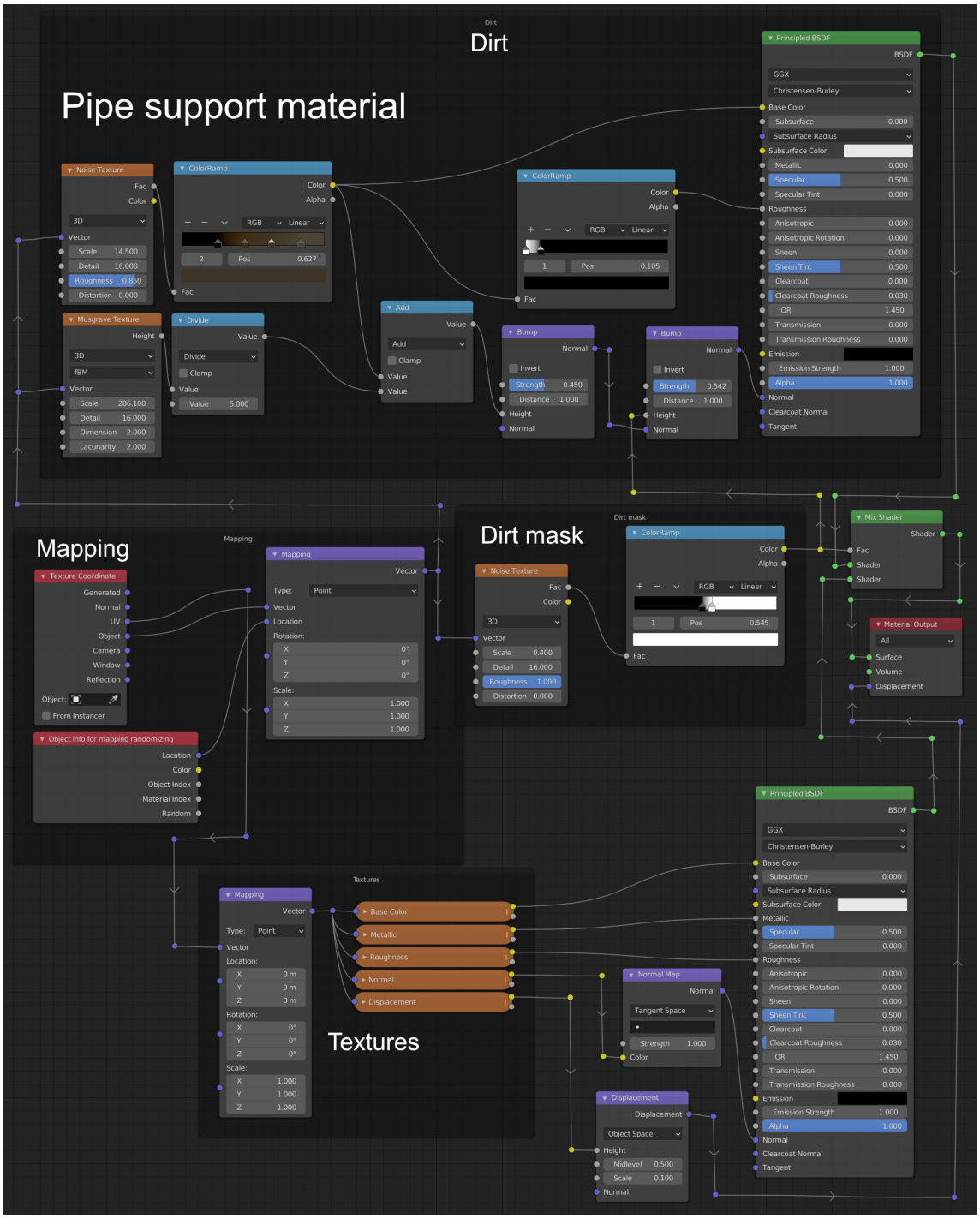


Figure A.3: Blender node tree for the procedural pipe support material as seen in fig. 5.15.

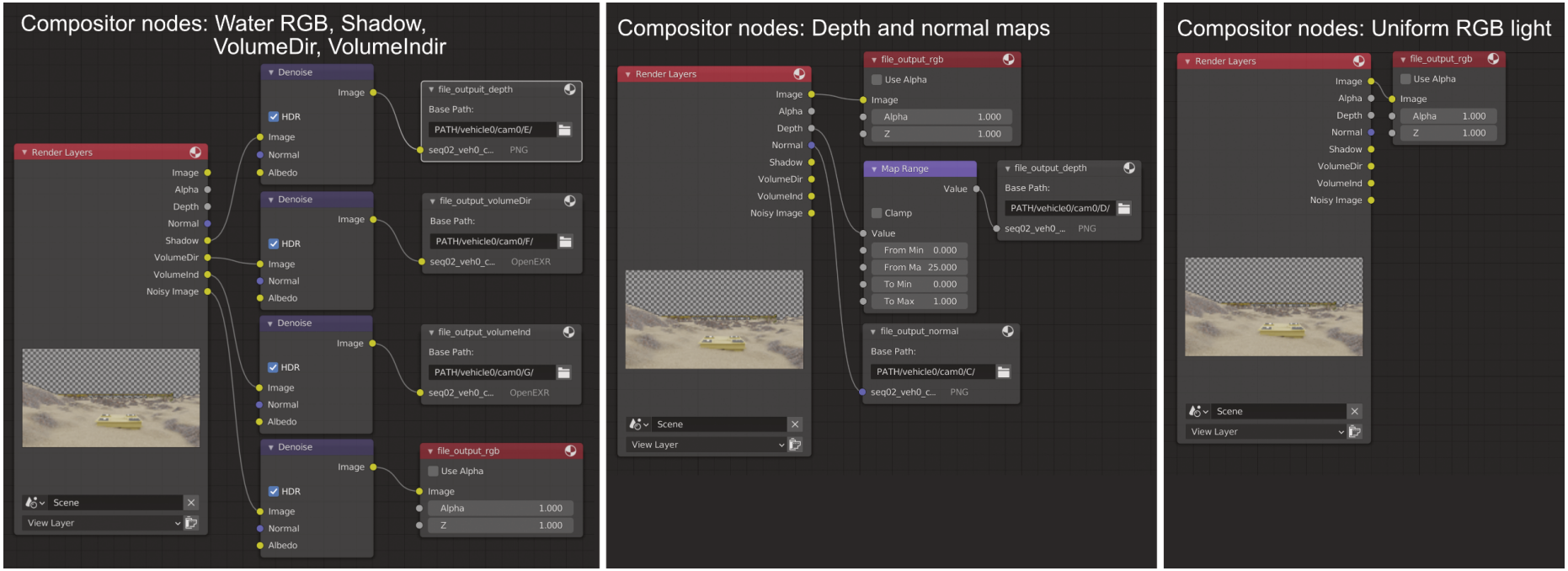
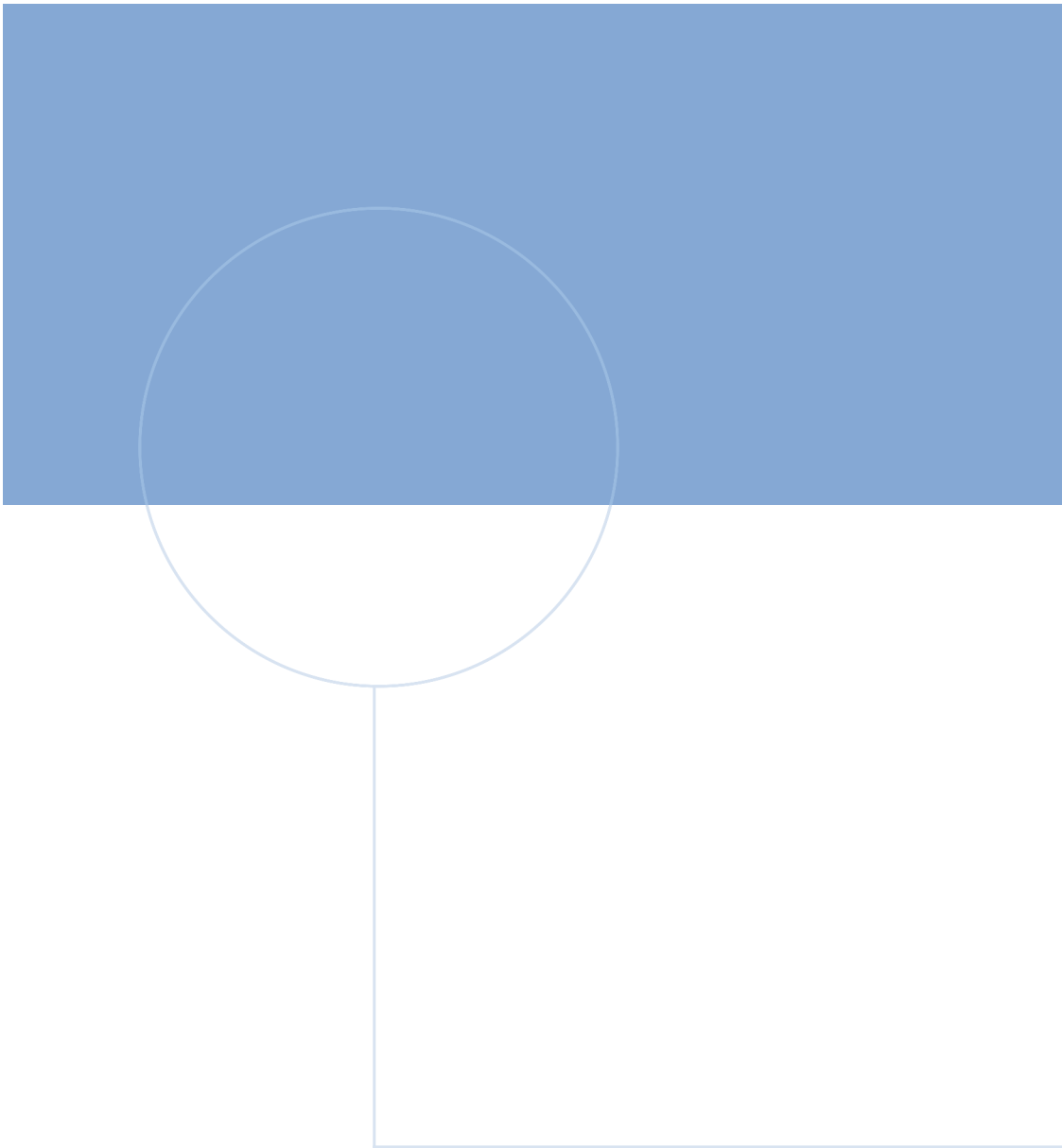


Figure A.4: Blender compositor node tree of different setups used for image export of the various image types discussed in section 5.7.



NTNU

Norwegian University of
Science and Technology