

Sveinung Aanbye Haugane

Modular Client Framework for Evaluation of Visual Navigation Systems

in Unreal Engine

Master's thesis in Cybernetics and Robotics

Supervisor: Edmund Førland Brekke

Co-supervisor: Elias Bjørne

August 2019

Sveinung Aanbye Haugane

Modular Client Framework for Evaluation of Visual Navigation Systems

in Unreal Engine

Master's thesis in Cybernetics and Robotics
Supervisor: Edmund Førland Brekke
Co-supervisor: Elias Bjørne
August 2019

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

Abstract

In the later years game engines, such as Unreal Engine, have been more and more utilized for their ability to render realistic environments, and thereby generate synthetic datasets to use in visual navigation. However, connecting these to various simulation software and hardware setups can be challenging. The problem of connecting different devices has led to the development of many types of middleware, defining standard terms of communication. While most of these are quite flexible in terms of integrability, this flexibility can, in many cases, induce unnecessary overhead and latency.

This master thesis will present the early development of a type of client software aimed towards the creation of easily interchangeable node based simulation setups in C++, with user-defined messages for inter-node communication while being as platform-independent as possible. The client differs from most other middleware in the sense that the node setup is decided at compile-time, allowing for extra optimizations and removing the need to connect the nodes via a network connection. This decision enables optimization for performance and latency. The client is presented with a real-world use case, connecting the simulation environment found in Microsoft's AirSim plugin for the Unreal Engine game engine, with ROS, in order to create simulation data for ROS-compatible visual navigation setups.

Sammendrag

I de senere åra har spillmotorer, som Unreal Engine, blitt brukt mer og mer grunnet deres evne til å gengi realistiske miljøer, og dermed kunne generere syntetiske dataset for bruk i visuell navigering. Å koble sammen disse forskjellige simuleringprogrammene med varierende maskinvare kan derimot være problematisk. Dette har ledet til mye nyutvikling av forskjellige typer mellomvare, som definerer standarder for kommunikasjon mellom enheter. Selv om de fleste av disse er fleksible når det gjelder integrerbarhet, kan denne fleksibiliteten i mange tilfeller forårsake unødvendig treghet i systemet.

Denne masteroppgaven presenterer en tidlig utgave av en klient-programvare, som er rettet mot å kunne sette opp lett utskiftbare, nodebaserte, simuleringsoppsett i C++, sammen med brukerdefinerte meldinger for kommunikasjon mellom noder, samtidig som den holder seg så platformuavhengig som mulig. Klienten skiller seg ut fra annen mellomvare ved at hele simuleringsoppsettet er bestemt før kompilering. Dette åpner for ekstra optimaliseringer og fjerner behovet for nettverkskommunikasjon mellom nodene. Dette gjør det mulig å optimalisere bort forsinkelser og øke ytelsen. Klienten blir presentert med et konkret bruksområde, ved å koble den sammen med Microsoft sin simuleringsplattform, AirSim, for spillmotoren Unreal Engine, og generere simuleringdata for ROS-kompatible oppsett for visuell navigering.

Preface

The presented work in this thesis concludes the five year Master of Science program, in the field of Cybernetics and Robotics, at the Norwegian University of Science and Technology, NTNU. The project counts towards all 30 credit points of one semester and summarizes the whole workload of the last semester.

While the development itself has been done by myself, I would like to thank my supervisors: Edmund F. Brekke and Elias S. Bjørne for their guidance and help with the task direction, report layout and finding relevant articles. I also want to direct a special thanks for Elias Bjørne for getting me up to speed with ROS, and acquiring the SKARV FPSO 3D model to use in the Unreal Engine simulations. In addition, I would like to thank the suppliers of the model, Aker BP ASA, and Cognite AS.

Contents

Abstract	i
Preface	iii
Table of Contents	v
List of Symbols	ix
Abbreviations	xi
1 Introduction	1
1.1 Report Outline	3
2 Theory	5
2.1 Interfaces	5
2.1.1 Library Interfaces	5
2.1.2 Object Oriented Interfaces	6
2.1.3 Inter-process relations	8
2.2 Optimizations	9
2.2.1 Compile Time Optimization	9
2.2.2 Threaded Programming and Asynchronous Operations	10
2.2.3 Runtime Optimization	11
2.3 Transforms and quaternion rotations	11

2.3.1	Transforms	11
2.3.2	Inverse Transforms	12
2.3.3	Quaternion Rotations	13
2.4	Modeling of cameras	14
2.4.1	Pinhole projection	14
3	Middleware and tools for VO and SLAM applications	17
3.1	Existing Middleware	17
3.1.1	Embedded middleware	19
3.1.2	Summary	20
4	Core Client Design	21
4.1	Initial Design Choices	21
4.2	Core Structure	22
4.2.1	Use Case Client Specific Setup	23
4.2.2	Node Interface and Design	23
4.2.3	Event Handling	25
5	AirSim Client Implementation	31
5.1	AirSim Client Design	31
5.1.1	AirSim Client Events	32
5.1.2	AirSim Node	33
5.1.3	ROS Node	34
5.2	Static Simulation Setup	34
5.3	Visualization	36
5.3.1	Rviz Map Frame	36
5.3.2	Ground Truth Map Visualization	37
5.3.3	Estimated Map and Transforms	37

6	Results	39
6.1	Core Client Benchmarks	39
6.1.1	Common benchmark setup	39
6.1.2	Transfer of simple message	40
6.1.3	Transfer of large data	41
6.1.4	Sample distributions for client benchmarks	43
6.2	AirSim Client Timings	44
6.3	Simulations	45
6.3.1	Unreal Engine Temple Simulations	47
6.3.2	Skarv FPSO Simulations	51
7	Discussion	53
7.1	Core Client design	53
7.1.1	Performance	55
7.2	Airsim Client Design	56
7.2.1	Performance	58
7.3	Simulations	58
7.3.1	Skarv FPSO simulations	59
7.3.2	Unreal Temple Simulation	59
8	Conclusion and Further Work	63
8.1	Conclusion	63
8.2	Further Work	64
	Bibliography	67

List of Symbols

f	focal length
Θ	Vertical field of view
Θ_H	Horizontal field of view
H	Image height in pixels
W	Image width in pixels
$\rho = \frac{H}{W}$	Image aspect ratio
u, v	Pixel coordinates
$\vec{\mathbf{p}}_{ab}^a$	Vector from the origin of frame a to the origin of frame b, given i frame a coordinates
$\vec{\mathbf{p}}^a$	Point in coordinate frame a
$\vec{\mathbf{q}}_{b,a}$	Quaternion Rotation from frame a to frame b
\vec{u}	Unit axis vector
\mathbf{R}_a^b	Matrix rotation from frame a to frame b
$\mathbf{R}_{i,\theta}$	Matrix rotation around axis i, by θ radians
R_{ij}	Matrix element on row i and column j
\mathbf{T}_a^b	Matrix transform from frame a to frame b
\mathbf{K}	Camera intrinsic matrix
\mathbf{P}	Projection matrix to pixel coordinates

Abbreviations

API	=	Application Program Interface
CAN	=	Controller Area Network
CPU	=	Central Processing Unit
CV	=	Computer Vision
DNN	=	Deep Neural Network
FoV	=	Field of View
fps	=	frames per second
FPSO	=	Floating Production, Storage and Offloading
GLFW	=	Graphics Library Framework
GPU	=	Graphics Processing Unit
HIL	=	Hardware in the Loop
IMU	=	Inertial Measurement Unit
IP	=	Internet Protocol
LIDAR	=	Light Imaging, Detection And Ranging
MAV	=	Micro Aerial Vehicle
NTNU	=	Norwegian University of Science and Technology
OpenCV	=	Open Source Computer Vision Library
Open GL	=	Open Source Graphics Library
OS	=	Operating System
RAM	=	Random Access Memory
RGB	=	Red, Green, Blue
RBGA	=	Red, Green, Blue, Alpha
ROS	=	Robot Operating System
RPC	=	Remote Procedure Call
SFML	=	Simple and Fast Multimedia Library
SLAM	=	Simultaneous Localization and Mapping
SSL	=	Secure Socket Layer
SVO	=	Semi-direct Visual Odometry
TCP	=	Transmission Control Protocol
UAV	=	Unmanned Aerial Vehicle
USB	=	Universal Serial Bus
VO	=	Visual Odometry

1 | Introduction

In simulation environments for visual navigation, there are many interconnections. The images may come from a camera, from a dataset or synthetically generated in a simulated environment. In addition to this, a simulation can contain various sensor setups to use in conjunction with the camera. All these different components require a way to communicate with the controller and estimation software.

Splitting different processes into modules or separate pieces of code is a fundamental principle within software development. Not only does this promote code reusability but also simplifies the development and maintenance of the code. Modularizing code is, however, quite challenging, as many processes not only share common parts but often have many dependencies.

One of the main ways to improve the reusability of code and better create independent modules is through abstractions. These abstractions allow for parts of the implementation details to change, while the user side of the program can remain the same. In distributed systems such as simulation systems, the modules need to exchange information. The low-level implementation details can, however, be abstracted away.

Messaging systems such as ROS[1] are popular examples of so called middleware, which is a type of software with the purpose of providing abstraction to the operating system. Usually in order to connect different operating system processes or applications through a common messaging system. Here the message distribution is handled through the message system itself. This lets each process focus on its own task, with the only external involvement being which messages to listen to, and what messages to send.

Most types of middleware are implemented through the use of a network connection, which allows the software to break programming language barriers, and communicate across different applications or programs. However, since messages sent over a network needs to be packaged for transport, distributed and reassembled by the receiver, this does induce latency. Besides, the compiler has no way of knowing what messages may be sent to it, which again disables the possibility of compile-time optimizations.

This thesis will present a modular, lightweight, node-based, heavily customizable client program, in order to make the process of setting up simulations and connecting different interfaces easier. It is not made to compete with middleware like ROS, but rather provide a performant way of connecting interfaces at a lower level, highly similar to the module interconnection found in game engines, compiling the code and producing a single binary.

To achieve this, the core client framework is written using modern C++17 in order to attain relatively high-level code abstraction, while utilizing the compile-time optimizations and move semantics added after C++11. The client includes a complete event system with the possibility of adding custom event types for semantically meaningful communication across modules and an extendable node class for the module-specific code. The node class implements a simplified interface to define the runtime operations a simulation, as well as a keyboard interface and logging system.

The thesis will also present an application for the client framework, showing how it can incorporate the API for the multicopter simulator AirSim[2] made for Epic Games' game engine Unreal Engine 4[3], together with a ROS[1] interface for publishing Image, IMU and transform data. Using this implementation, ground truth map and transform data from the AirSim simulation, will be visually compared to the estimated data gained through ORB-SLAM2[4, 5], and SVO[6] using RViz[7]. In addition to the simulations, benchmarking results will be shown for both the core client and the full AirSim client implementation.

1.1 Report Outline

The thesis is divided into four main parts: The first part consists of theoretical background on software interfaces and the application of middleware in addition to a some theory on 3D geometry and camera projection. The second part presents a small literature review on existing middleware and their application. The third part contains the implementation details of the framework and the specific extension towards ROS and Unreal Engine through AirSim. Lastly, benchmarks and simulation results will be presented and discussed.

- **Chapter 2:** Theoretical background on software development, 3D geometry and camera projection.
- **Chapter 3:** Short literature study on existing middleware used for robotics applications.
- **Chapter 4:** Implementation of the core client framework.
- **Chapter 5:** Implementation of an Unreal Engine simulation setup towards ROS and Airsim using the client framework.
- **Chapter 6:** Benchmarks and simulation results.
- **Chapter 7:** Discussion on the client's strengths and weaknesses, as well as the simulation results.
- **Chapter 8:** Conclusion and Further work, presenting limitations as well as possibilities for extension and uses.

2 | Theory

In order to modularize software and simplify its use for a programmer or end-user, abstractions are used. Not only does this increase development speed, but well thought out abstractions also help others who use the code to better understand and reason about its functionality. This abstraction is usually defined through interfaces into the underlying program, whether this is a graphical user interface, bit patterns in terms of bus communication or function definitions in programming. However, some abstractions can come at the cost of performance, and there is usually a tradeoff between performance and usability.

The first part of this chapter will look into the different kinds of interfaces and abstractions found in software, as well as different types of optimizations the compilers and processors can do. The second part will cover some basic transformations used in 3D geometry as well as a short introduction to camera projection and modeling, which is used in the simulation specific part of the thesis.

2.1 Interfaces

An interface is the set of available tools the user has to use or interact with a system or program. Interfaces directed at end-users usually consist of a graphical user interface(GUI) and are made to be used without any knowledge of the underlying software. On the other hand, developer interfaces, often denoted as application program interfaces or APIs, consist of a set of functions and tools to programmatically use a library, without concerning the developer with the implementation details.

2.1.1 Library Interfaces

Figure 2.1 shows a simplified version of an application, depending on a Graphics library and a Keyboard Input library. The abstraction provided by the two libraries can then enable the programmer to create a window, set a layout, check for key presses and print to screen, with relatively few function calls, while the actual interaction with the operating system and peripherals happen in the two libraries.

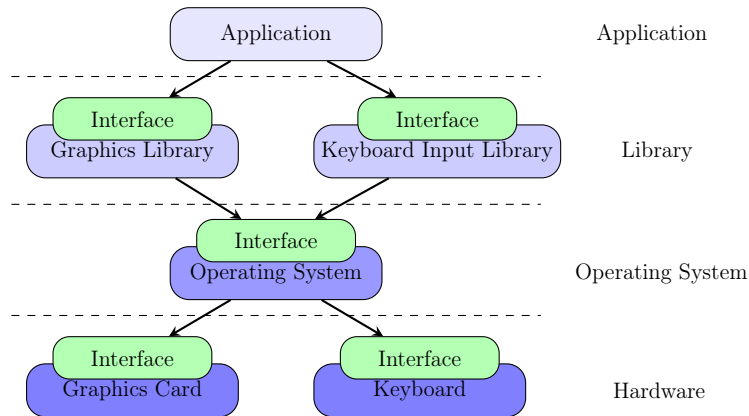


Figure 2.1: Example of connected interfaces.

The Key input library may, for example, implement the function; *isKeyPressed()*, checking the status of a keyboard key. This function may take a keycode, which is just a number but has the semantical meaning of a character. Internally this number can refer to the specific keycode enumeration the operating system uses for the same thing. What number this is may change for different operating systems. The function itself may also change behavior depending on the operating system(OS), as the OS may define specific locations to store the status of its peripherals. In order to update this status, the electrical signal of the button press is translated into a code. This code is then transmitted over USB or a similar standard, which again requires driver software to interpret.

The *isKeyPressed()* function is a part of the libraries interface, and the full interface of the library will be the types and functions available to the user. The important part is that the programmer does not need to know the details in the previous paragraph, as long as the function provides information on whether the key is pressed.

2.1.2 Object Oriented Interfaces

In languages that support object-oriented programming, functions are often tied to specific classes, structs or objects. This relationship allows different classes to implement methods with the same name, which operate differently. However, one can argue that these functions or methods should perform the same general operation. For example, a *printName()* method should print the name of the entity, independent of whether this is a Dog, an Employee or Student. This sort of behavior is called polymorphism.

In languages such as Java, C++, and C#, this is done through inheritance. Here, a generic category class is implemented as a polymorphic interface, and other more specific instances of that class are defined to be of that type. An example of this is shown in Figure 2.2. Here the *Shape* class encapsulates all of the underlying classes, saying that a *2DShape* is a *Shape* as well as a *2DShape*, and a *Sphere* is both a *Sphere*, *3DShape* and a *Shape*. Utilizing the knowledge of the underlying type, a function can take in a *Shape* and call the function *Shape::Area()*, which is then mapped to the correct *Area()* function call of the underlying type.

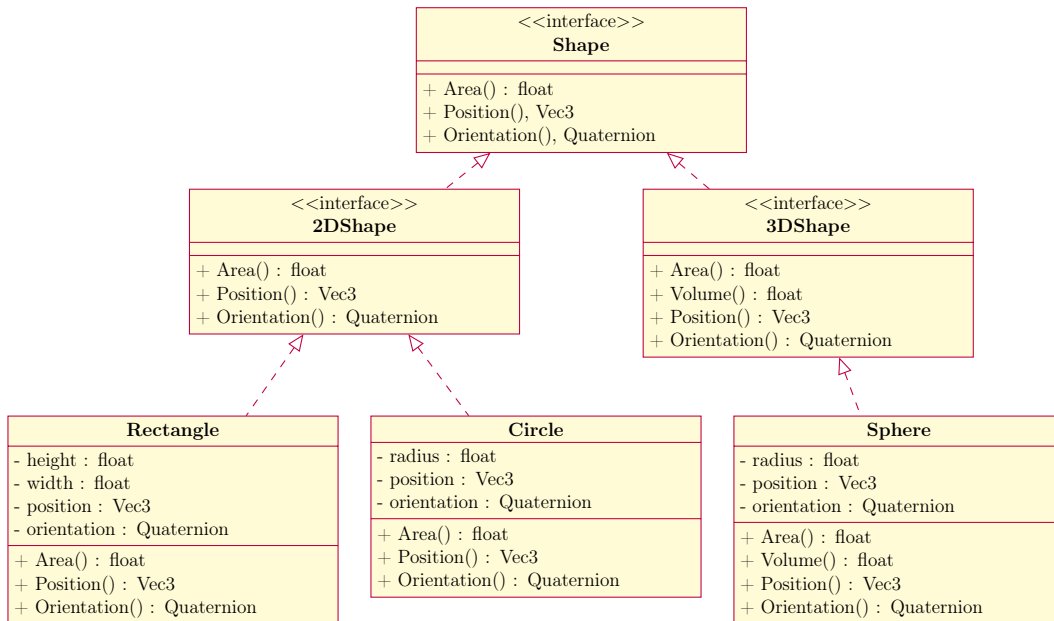


Figure 2.2: Example of a polymorphic interface for geometric shapes.

Another way of implementing functional interfaces is through the interface type, which is seen in the Go programming language. Here an interface type is defined in the core language as all types which implement a specific named function.

The *areaPrinter* function shown in Figure 2.3 will accept any type as long as the type implements the three methods shown in the Shape interface definition. This means that any of the geometric types described in Figure 2.2 would be accepted since they all implement their version of the interface.

```

type Shape interface {
    Area() float64
    Position() Vec3
    Orientation() Quaternion
}

func AreaPrinter(s Shape) {
    fmt.Println("Area: ", s.Area())
}

```

Figure 2.3: Example of a Shape interface definition in the Go programming language along with how it can be used.

While this approach is similar in many ways, there is a key difference. In the polymorphic example, all of the Geometric types are defined as shapes in addition to their other types, while in the example shown in Figure 2.3 the type is not necessarily a shape, but has the same interface as a shape, and can be treated as a shape for that specific function call. Any type could therefore be sent to the function, as long as the three functions *Area()*, *Position()* and *Orientation()* are implemented for the *Screen* type.

Both of these design approaches are incredibly powerful when it comes to operations

that have the same semantical meaning, but where the implementation differs. It also provides a way to implement common operations for different types, without writing function overloads for each type, and therefore replicating much of the code.

2.1.3 Inter-process relations

Large scale systems are often highly modularized, and may even be run across different machines and applications. In such systems, there are bound to be interprocess dependencies. An example of such a dependency is a controller waiting for new sensor data or estimation results, or a base station needing the position of an aircraft to coordinate flight traffic. Programs that formalize this kind of communication and provide abstractions that can be used across different units are called middleware.

Middleware has its name from being an abstraction of operating system functionality. This can refer to everything from hardware abstraction, simplifying communication over a CAN-bus or packaging messages for transmission over a network. The goal of the middleware is to provide a common platform or language where different types of processes can communicate the same information in the same way.

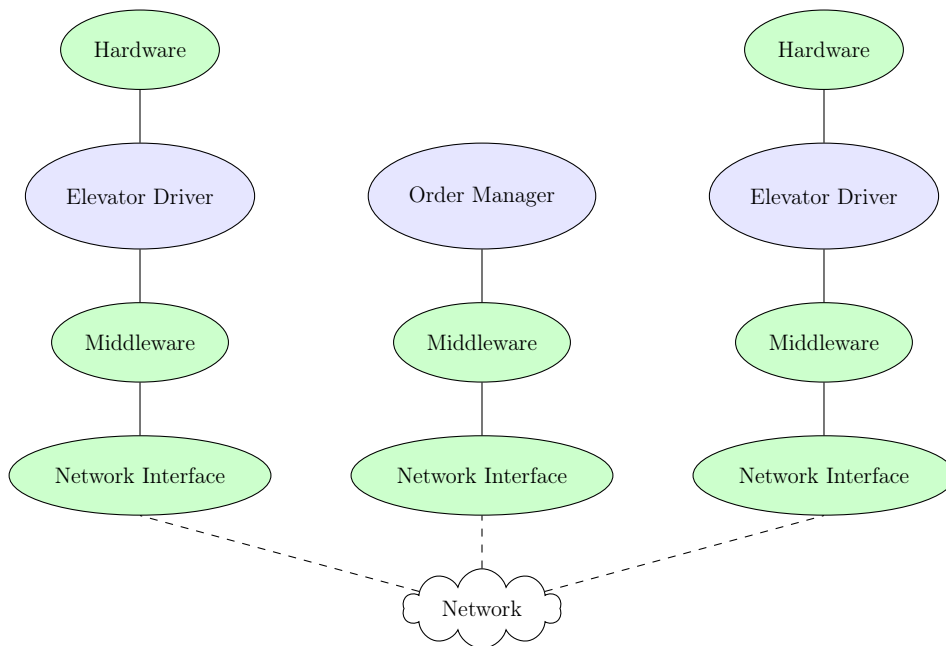


Figure 2.4: Distributed elevator system, utilizing network communication through middleware.

Figure 2.4 shows an example of an Elevator system depending on middleware in order for a centralized order manager to distribute orders between the different elevators. Not only does this enable the system to be easily scalable, in order to add more elevators in the future, but it also provides an abstraction to the specific elevator hardware, enabling the system to be used with different elevators.

2.2 Optimizations

A programming language is very different from the machine code, which is run by the operating system. In some form or another, the written code has to be interpreted or compiled into machine code. While interpreted languages like Python and Javascript usually are very readable and easy to write, they do lack some of the performance benefits of a compiled language. The reason is that each line of code has to be parsed and interpreted into machine code for each execution. While this makes the code very easy to follow, as it is executed one line at a time, it also removes the possibility to remove unnecessary operations or precalculate results. This is called compile-time optimization.

2.2.1 Compile Time Optimization

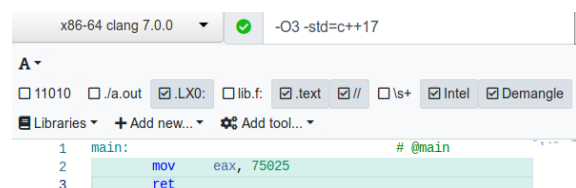
In compiled languages like C++ or Go, the code compiles into a single machine-readable executable. Code indirections, like function calls, can be interpreted once, and in many cases also omitted if all the needed information is available at compile-time.

Using Matt Godbolt's compiler explorer¹, one can compare the Assembly generated by many different C++ compilers, for different CPU architectures. Figure 2.5 shows an example of how the *n*th Fibonacci number can be found during compile time. The result shown in the figure is compiled using the clang compiler for the x86-64 architecture, which is found in most modern computers, but similar results can be found using other compilers and architectures.

This means that everywhere in the code where the Fibonacci function is used, the compiler can switch the function call with this number, and thereby reducing the execution time of the program. Comparing Figure 2.5a and 2.5b. It can also be seen that the variable *fib* in the *main()* function is also completely optimized away, as there is no real need to store it.

```
1 constexpr int fibonacci(int n) {
2     if (n == 0) return 0;
3     if (n == 1) return 1;
4     return fibonacci(n-2) + fibonacci(n-1);
5 }
6
7 int main() {
8     constexpr auto fib = fibonacci(25);
9     return fib;
10 }
```

(a) Fibonacci C++14/17 code



```
x86-64 clang 7.0.0 -O3 -std=c++17
A
11010 /a.out LX0: lib.f: .text // \s+ Intel Demangle
Libraries + Add new... Add tool...
1 main: # @main
2     mov     eax, 75025
3     ret
```

(b) Assembly of compiled fibonacci code

Figure 2.5: Comparison of C++ code and the generated assembly.

¹<https://godbolt.org/z/J8vV0W>

2.2.2 Threaded Programming and Asynchronous Operations

In modern GPUs and most modern CPUs, there are multiple processor cores, which enable simultaneous execution of different instructions. These cores have a fixed number of hardware threads available, where each thread can hold a set of instructions to compute. While only one thread can be run on each core simultaneously, thread execution can be halted in order for other threads to execute. This can, for example, be beneficial when a thread relies on reading from or writing to slow memory.

Multithreaded programs are usually not limited to the number of CPU threads. While the software threads apply the same principle of out of order execution, they do not tie directly to parallel processing. They instead propose concurrent behavior. Applying concurrency means that the programmer does not care in which order the CPU executes the instructions, and it is therefore left to the scheduler in the operating system to decide the order. The instructions are processed in parallel if there are available resources, but there are no guarantees.

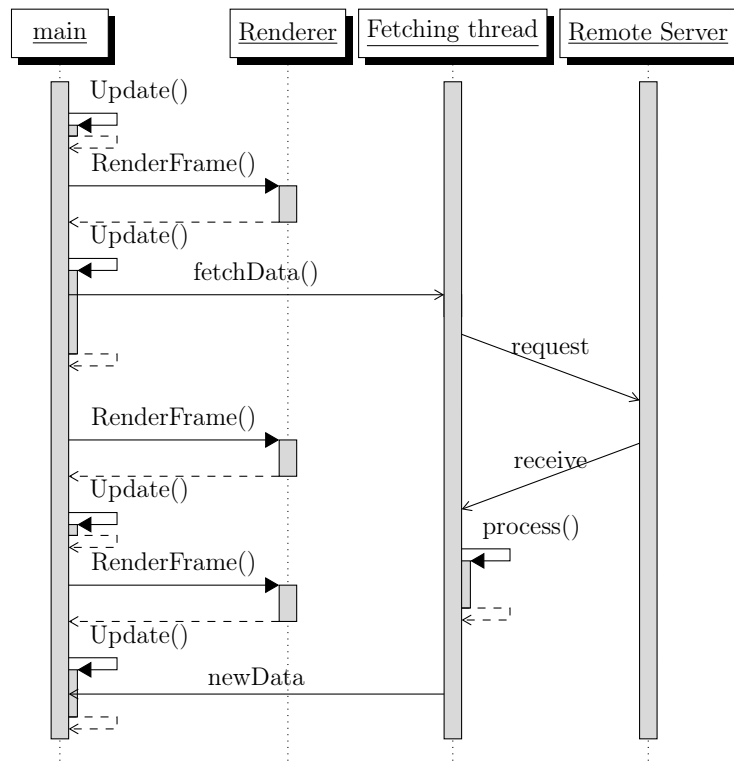


Figure 2.6: Example of asynchronous operation.

In cases where a program relies on a steady flow of updates, for example, game applications, this can be hugely beneficial, which means that slowly executing or unreliable code, can be run in a separate thread. This way, the frame rate can be held stable while still getting the remote data. Figure 2.6 shows an asynchronous operation, where the data fetching is run at its own pace, while the rest of the program executes as usual.

2.2.3 Runtime Optimization

Runtime optimizations are usually highly hardware-specific, where everything from the amount of short term memory, like cache and RAM, to the number of processor cores or process threads play a role. Most modern CPUs also perform branch prediction. Branch prediction refers to looking at the trends of conditional statements, which are comparable to if statements, and beginning a partial execution of the most likely branch. This can lead to significant performance benefits for checks that take multiple CPU cycles to execute. However, if the prediction is wrong, the processing has to start over.

Another way of improving execution time can be through instruction pipelining and by applying out of order execution. Pipelining refers to queuing instructions in such an order that the CPU utilizes most of its available resources each clock cycle. This queuing operation can be improved further by changing the order of instructions, where the result is independent of the execution order.

2.3 Transforms and quaternion rotations

It is often beneficial to define multiple coordinate systems in a control application in order to simplify computations and better reason about the parts of a system. The IMU, for example, is placed in a fixed position on a vehicle, even though the vehicle moves. This knowledge is what makes it possible to calculate the vehicle accelerations and velocities, even though the measurements of the IMU are done in a different coordinate frame.

2.3.1 Transforms

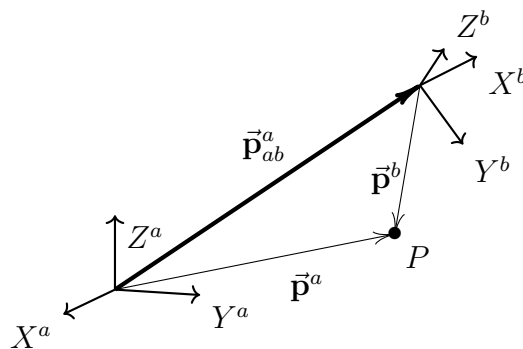


Figure 2.7: Two rotated coordinate frames set at a positional offset of \vec{p}_{ab}^a .

Two such coordinate frames are shown in Figure 2.7. Here the vectors \vec{p}^a and \vec{p}^b refer to the position of point P in the coordinate frames of \bullet^a and \bullet^b respectively. Using the vector \vec{p}_{ab}^a , the relationship between the vectors can be described as in Equation (2.1), where \mathbf{R}_b^a refers to the rotation matrix that maps the vector \vec{p}^b to the coordinates of frame \bullet^a .

$$\vec{\mathbf{p}}^a = \begin{bmatrix} x^a \\ y^a \\ z^a \end{bmatrix} = \mathbf{R}_b^a \vec{\mathbf{p}}^b + \vec{\mathbf{p}}_{ab}^a = \mathbf{R}_b^a \begin{bmatrix} x^b \\ y^b \\ z^b \end{bmatrix} + \vec{\mathbf{p}}_{ab}^a \quad (2.1)$$

Rotations can be split into a product of intermediate, and often simpler rotations. It is often beneficial to decompose the rotation into single axis rotations. This is shown in Equation (2.2). Note however that rotation matrices are not commutative, which means that $\mathbf{R}_1\mathbf{R}_2$ is not equal to $\mathbf{R}_2\mathbf{R}_1$ in general.

$$\begin{aligned} \mathbf{R}_b^a &= \mathbf{R}_1^a \mathbf{R}_2^1 \mathbf{R}_b^2 = \mathbf{R}_x(\theta) \mathbf{R}_y(\phi) \mathbf{R}_z(\psi) = \\ \mathbf{R}_b^a &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (2.2)$$

The relationship shown in Equation (2.1) can also be described by a single matrix transform. This requires the vectors $\vec{\mathbf{p}}$ to be extended to a homogeneous coordinate, $[x, y, z, 1]^\top$, in order to incorporate the translational offset, $\vec{\mathbf{p}}_{ab}^a$. The matrix transform from $\vec{\mathbf{p}}^b$ to $\vec{\mathbf{p}}^a$ is shown in Equation (2.3), with \mathbf{T}_b^a being the transform matrix, representing the transform from frame \bullet^b to \bullet^a .

$$\begin{bmatrix} \vec{\mathbf{p}}^a \\ 1 \end{bmatrix} = \mathbf{T}_b^a \begin{bmatrix} \vec{\mathbf{p}}^b \\ 1 \end{bmatrix} = \mathbf{T}_b^a \begin{bmatrix} x^b \\ y^b \\ z^b \\ 1 \end{bmatrix} \quad (2.3a)$$

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \vec{\mathbf{p}}_{ab}^a \\ \vec{\mathbf{0}}^\top & 1 \end{bmatrix} = \begin{bmatrix} R_{1,1} & R_{1,2} & R_{1,3} & p_x \\ R_{2,1} & R_{2,2} & R_{2,3} & p_y \\ R_{3,1} & R_{3,2} & R_{3,3} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.3b)$$

2.3.2 Inverse Transforms

Since a rotation maps coordinates from one frame to another, there should exist an inverse operation which does the inverse mapping. This mapping is the inverse Rotation matrix, $\mathbf{R}_a^b = \mathbf{R}_b^a^{-1}$. Since all rotation matrices are in the $SO3$ group, all rotations are orthogonal and with determinant of 1. This means that the inverse is the matrix transpose, as shown in Equation (2.4).

$$\mathbf{R}_a^b = \mathbf{R}_b^a^{-1} = \mathbf{R}_b^a{}^\top, \text{ for } \mathbf{R}_b^a \in SO3 \quad (2.4)$$

This does however not hold for the $SE3$ group, which the transform matrix \mathbf{T}_b^a is part of. However, using the relationship shown in Equation (2.5), in addition to the definition of $\mathbf{T}_a^b = \mathbf{T}_b^a$, we get the result shown in Equation (2.6).

$$\vec{\mathbf{p}}_{ba}^a = -\vec{\mathbf{p}}_{ab}^a \quad (2.5a)$$

$$\vec{\mathbf{p}}_{ba}^b = \mathbf{R}_a^b \vec{\mathbf{p}}_{ba}^a = \mathbf{R}_b^{a\top} \vec{\mathbf{p}}_{ba}^a = -\mathbf{R}_b^{a\top} \vec{\mathbf{p}}_{ab}^a \quad (2.5b)$$

$$\mathbf{T}_b^{a-1} = \mathbf{T}_a^b = \begin{bmatrix} \mathbf{R}_a^b & \vec{\mathbf{p}}_{ba}^b \\ \vec{\mathbf{0}}^\top & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_b^{a\top} & -\mathbf{R}_b^{a\top} \vec{\mathbf{p}}_{ab}^a \\ \vec{\mathbf{0}}^\top & 1 \end{bmatrix} \quad (2.6)$$

2.3.3 Quaternion Rotations

Another powerful technique used to compute rotations is quaternions. These are four-dimensional complex values on the form:

$$\vec{\mathbf{q}}_{a,b} = q_w + q_x \cdot \vec{i} + q_y \cdot \vec{j} + q_z \cdot \vec{k} = [q_w \quad q_x \quad q_y \quad q_z]^\top, \quad (2.7)$$

where $\vec{i}^2 = \vec{j}^2 = \vec{k}^2 = \vec{i}\vec{j}\vec{k} = -1$. Here, the subscripts \bullet_a and \bullet_b refers to two coordinate frames, and $\vec{\mathbf{q}}_{a,b}$ is the rotation from \bullet_b to \bullet_a .

On another form, the quaternion can describe a rotation θ around an axis unit vector \vec{u} . This relation is shown in Equation (2.8).

$$\vec{\mathbf{q}}_{a,b} = e^{\frac{\theta}{2}(u_x \vec{i} + u_y \vec{j} + u_z \vec{k})} = \begin{bmatrix} \cos(\frac{\theta}{2}) \\ u_x \cdot \sin(\frac{\theta}{2}) \\ u_y \cdot \sin(\frac{\theta}{2}) \\ u_z \cdot \sin(\frac{\theta}{2}) \end{bmatrix}, \text{ for } \vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \frac{\vec{\mathbf{p}}^b \times \vec{\mathbf{p}}^a}{\|\vec{\mathbf{p}}^b \times \vec{\mathbf{p}}^a\|} \quad (2.8)$$

Similarily, the inverse quaternion can be computed as in Equation (2.9), using the same unit vector \vec{u} .

$$\vec{\mathbf{q}}_{a,b}^{-1} = e^{-\frac{\theta}{2}(u_x \vec{i} + u_y \vec{j} + u_z \vec{k})} = \begin{bmatrix} \cos(\frac{\theta}{2}) \\ -u_x \cdot \sin(\frac{\theta}{2}) \\ -u_y \cdot \sin(\frac{\theta}{2}) \\ -u_z \cdot \sin(\frac{\theta}{2}) \end{bmatrix} = \begin{bmatrix} q_w \\ -q_x \\ -q_y \\ -q_z \end{bmatrix} \quad (2.9)$$

Equation (2.8) and (2.9) can be used in tandem to rotate any vector in \mathbb{R}^3 . This is done by defining an intermediate quaternion $\vec{\mathbf{q}}'_b = [0, \vec{\mathbf{p}}^{b\top}]^\top$, and then apply the operation shown in Equation (2.10).

$$\begin{bmatrix} 0 \\ \vec{p}^a \end{bmatrix} = \vec{q}'_a = \vec{q}_{a,b} \vec{q}'_b \vec{q}_{a,b}^{-1} = \vec{q}_{a,b} \begin{bmatrix} 0 \\ \vec{p}^b \end{bmatrix} \vec{q}_{a,b}^{-1} \quad (2.10)$$

2.4 Modeling of cameras

Section 2.4 is more or less copied directly from my project thesis[8], providing only minor alterations in order to fit the setup of this master thesis.

All cameras project a 3D scene onto a 2D plane. This projection causes information to be lost about the depth of the image. It is therefore not possible to calculate the exact placement of an object from a single picture unless there is extra information about the objects in the picture. For this reason, a projection can easily be created from a 3D scene, but it is hard to re-create a scene from a projection. Additionally, the number of pixels and the field of view(FoV) also affect the information and detail in the captured image. The most basic camera model is called the pinhole model, and it is applicable for most cameras without high distortion lenses.

2.4.1 Pinhole projection

The pinhole model replicates the capturing of a scene by projecting straight light rays through a common focal point, and a plane. The projection itself is made from where the light rays intersect the projection plane. The focal length, f , and the image plane size will decide the field of view(FoV), Θ_H and Θ , as shown in Figure 2.8.

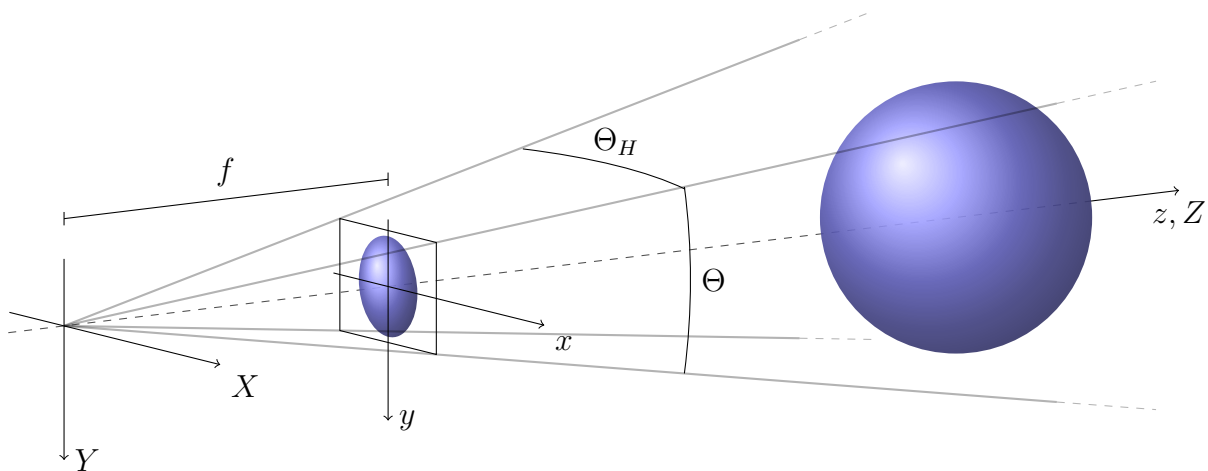


Figure 2.8: Pinhole projection with the image plane between the focal point and the object.

Using the properties of similar triangles, the relationship between the world coordinates, X, Y, Z , and the image coordinates x, y becomes:

$$\tan(\phi_x) = \frac{x}{f} = \frac{X}{Z} \qquad \tan(\phi_y) = \frac{y}{f} = \frac{Y}{Z} \qquad (2.11)$$

$$x = f \frac{X}{Z} \qquad y = f \frac{Y}{Z} \qquad (2.12)$$

As seen in Equation (2.12), the relationship between the sizes is nonlinear. In order to present this in matrix form, the homogeneous coordinates, $\mathbf{p}^o = [x, y, 1]^\top$, and $\mathbf{P}^o = [X, Y, Z, 1]^\top$ are used. The matrix transformation is shown in Equation (2.13), with the intermediate step $\tilde{\mathbf{p}}^o$ being \mathbf{p}^o scaled by Z . The \bullet^o superscript refers to the optical frame of the camera.

$$\tilde{\mathbf{p}}^o = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad \mathbf{p}^o = \frac{1}{\tilde{z}} \tilde{\mathbf{p}}^o \qquad (2.13)$$

Since the light rays need to pass through the pinhole and onto the image plane, it is not possible to have a FoV larger than 180° . As seen in Equation (2.13), the projected object is also scaled by the distance, causing the points where the vertical FoV is 180° to be singular.

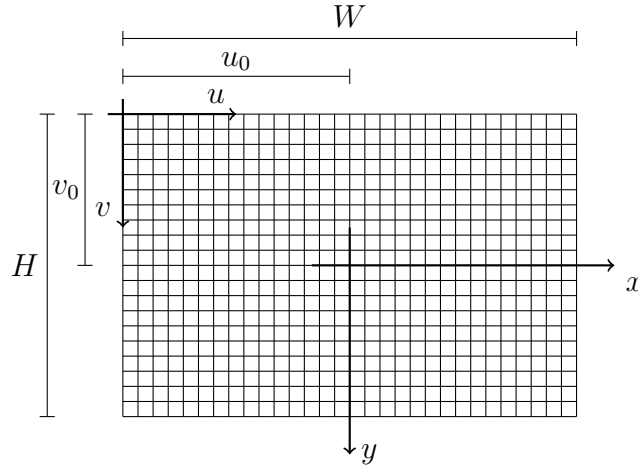


Figure 2.9: Relationship between pixel and image coordinates.

In a digital camera, the image plane consists of a small chip with a discrete number of light sensitive elements, or pixels. A new pixel coordinate frame is defined to consist of the image pixels, with the origin in the upper left corner, as shown in Figure 2.9. Using W and H as the image width and height in pixels, respectively, the transformation from image coordinates to pixel coordinates will be as shown in Equation (2.14), with \bullet^p referring to the pixel coordinate frame, with its origin at the camera center.

$$\mathbf{p}^p = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{W}{2x_{max}} & 0 & u_0 \\ 0 & \frac{H}{2y_{max}} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}^o = \begin{bmatrix} \frac{W}{2x_{max}} & 0 & \frac{W}{2} \\ 0 & \frac{H}{2y_{max}} & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2.14)$$

3 | Middleware and tools for VO and SLAM applications

As computing power continues to grow, more and more applications of visual odometry and SLAM become available. Today even pre-trained deep neural networks(DNN) can be run on small aerial vehicles, as were done in [9] using DNN for object detection along with direct sparse odometry[10] on an NVIDIA Jetson TX1 graphics processing unit. It is especially the availability of on-board graphics capabilities which have enabled this, where the team behind SVO managed to get framerates as high as 55 frames per second(fps) using an embedded Odroid U2, along with greyscale images of 752×480 pixels[6].

As eluded to in [8], there is also an increasing number of simulation tools[2, 11–13] able to create synthetic images and sensor data with more and more realistic properties, using modern game engines. According to [14], Kongsberg Digital has also integrated AirSim into their own rendering engine, COGS, to generate machine learning training data.

These game engines can replicate real-world effects such as multiple light sources in a room, and casting realistic shadows, for dataset generation. The simulation tools mentioned here have also expanded the physics capabilities of the engines in order to provide complete simulation environments.

An interesting use case for these systems are Hardware in the loop(HIL) simulations, where sensor data can be simulated and provided to a real world controller, which again can control the simulated system. Examples of this can be seen in [2, 15, 16], simulated using AirSim, VEHIL[17] and CARLA[13] respectively.

The process of connecting hardware, or other software to these simulations is however not trivial. For this reason there exists many types of middleware, which incorporates the task of creating a common communication pattern between the different parts of a system.

3.1 Existing Middleware

A survey made by Mohamed et. al.[18] compared fifteen different middleware solutions, looking at their design goals and trying to categorize them in terms of reusability, simpli-

fication of development, whether they offered real-time guarantees, how extendable they are in terms of functionality, and whether they support direct communication with a robot's peripherals.

Seeing that the applications of middleware vary as much as they do, and that the developers clearly interpret the problems differently, they conclude that the best solution probably would be some sort of component based middleware, which includes the needed functionality.

lastly, the survey also summarized some of the open issues still to be addressed in terms of robotics development. These include:

1. Lack of standardization between middleware
2. Limited self-adaptation and self-configuration possibilities
3. Investigation of security for data storage and transmission
4. Coordination and collaboration of robots

It should be noted that these were open issues as of 2008, and much has happened in the later years. Already in 2012, Elkady et.al. addressed some of these issues in their survey[19]. For example MRDS[20, 21] include frameworks for robot coordination and behaviour execution, which addresses point 4 to some extent.

Table 2 in the survey by Elkady et.al[19] also list multiple middleware solutions which adds data access security guarantees. Whether this issue has been adequately addressed can however be discussed, as this is a huge topic and they are quite brief in their explanation. However, they explicitly state that Common Object Request Broker Architecture(CORBA)[22], which is a standard used by some middleware, supports encryption through Secure Sockets Layer(SSL) for their network transfers. Among the around twenty types of middleware examined by the survey, many of them overlapped with [18]. However, they examined some extra categories such as fault tolerance, platform independability and configuration at runtime.

According to a thesis written in 2018[23], the eight most used types of the middleware solutions for multi-robot environments are:

- Carnegie Mellon Robot Navigation Toolkit(Carmen)[24]
- Miro[25, 26]
- Mobile and Autonomous Robotics Integration Environment(MARIE)[27]
- Microsoft Robotics Developer Studio(MRDS)[20, 21]
- Orca[28]
- Player/Stage[29]

-
- Python Robotics(Pyro)[30, 31]
 - Robot Operating System(ROS)[1]

The application of these systems vary quite a bit, both in terms of design goals, aims, incorporated tooling and simulation capabilities. For example Pyro, is created for teaching robotics to students without much experience in the field[31] and is therefore implemented for a handful of specific robots, while Carmen implements a system for dynamic interconnection between context aware services on a wireless network.

MRDS, Carmen, ROS and also Player/Stage implements their own simulation platform with their respective messaging protocol integrated. In addition Player and Miro also supports the Gazebo[32] simulator for the ROS platform. It should, however, be noted that these simulators are implemented towards physics simulations, and not realistic 3D-rendering, making them less powerful in terms of visual odometry and SLAM.

Most of these modules provide hardware abstraction through a common messaging system, usually over a network connection, enabling high-level features such as managing behaviour and movement patterns through predefined modules. However, as described in[18], there is a lack of standardization between the different middleware solutions. Here MARIE and ROS stand out, as they are made with the ability of easily wrapping the other standards in their own module framework, and therefore providing a common communication platform.

This is highly valuable for systems where the setup changes regularly, which it usually does in research and production environments. They also provide great flexibility both in terms of programming language support and supported platforms, relying on a network connection. As discussed in Section 2.2, splitting programs into different communicating applications can have significant performance implications. This is also the reason why many of these cannot be used in embedded systems.

3.1.1 Embedded middleware

Miro, mentioned in the previous section is an example of such middleware, which is made to run on a network of microcontrollers, connected by a bus. Using a small virtual machine run on each microprocessor, it provides high level communication abstraction for common bus technologies such as CAN or I2C. Another such middleware is Aseba[33], which provides hardware abstraction through their own scripting language and programming platform, which can be interfaced through TCP/IP.

In addition to these, the PX4 framework[34] should be mentioned. This framework is for example used for the popular Pixhawk PX4 flight controller. The PX4 is a Unix-based form of middleware, which provides a publish/subscribe type messaging protocol. It incorporates many bus technology standards, to provide low level data transfer of sensor and control messages. It also provides an interface to MAVLink[35], which is a message library designed for communication between a ground station and drones. The module

is also made to be wrapped into a ROS node if a network connection or other modules featuring ROS APIs are needed.

Common for the embedded platform is that they need to be built for specific hardware, which is why they are used for on board flight controllers or similar systems. This does however mean that there are currently only specific embedded platforms where they can be used. Many of them can however be interfaced towards general purpose computers and are therefor useful for hardware in the loop simulations. The PX4 also has an advantage in that it can be compiled to Rapberry Pi, which is highly versatile in its usage.

3.1.2 Summary

There are lots of types of middleware available for robotics applications, whether it is made to be run as a message platform on a general purpose computer, or embedded into specific hardware. Most commonly, general purpose middleware incorporate messages through a network connection. This adds lots of flexibilities, but has some implications when it comes to performance. On the other hand, the applications of embedded middleware can be quite limited, as they are targeted towards specific hardware. This does however mean that many of them can be interfaced towards a computer to do hardware in the loop simulations.

The numerous middlewares are mostly made to work independently. While systems like MARIE and ROS are made to incorporate other robotics interfaces, there are few or no platforms which let the developer use a common messaging system, to combine different APIs within one single application on a general purpose computer, meaning that there is a possible use case for the client described in Chapter 1, as long as it does not create significant overhead to the simulation.

4 | Core Client Design

The client framework described in this section is not made to be a competitor to other middleware, as there would be no way to compete with the number of features, flexibility and tooling these have, in the short timeframe of the project.

The goal of this client is, therefore, to build a lightweight and modular framework, with minimal execution overhead, seeking to be as platform-independent as possible. At the same time, it should force the programmer to think about modularity, and thereby promoting reusable code, while only paying for the latency of functionality which is needed.

It will also be made with other middleware applications in mind so that the client can be extended with their APIs in order to incorporate additional features. However, the requirement for this is that it has a C or C++ API.

4.1 Initial Design Choices

As previously mentioned, creating a single executable for the application has its benefits in the fact that the compiler can optimize the code better, most likely increasing the performance in comparison to distributed systems. The drawback, however, is that the client itself needs a resource manager, deciding when a module can operate, as the operating system cannot break the sequence of code execution within single-threaded programs. This function can, however, be replicated to some extent through multi-threaded programming, but the client would still need some sort of manager which creates and keeps track of the asynchronous operations.

Due to the nature of the application, where each module can require a different amount of data processing time, a multi-threaded approach would most likely be preferable. Especially if some modules perform heavy calculations, or slow, blocking, API calls. Delving into multi-threaded programming does however open up a whole new range of problems, such as race conditions, where multiple threads try to write to the same memory, and deadlocks, where threads are stuck waiting for each other. Delving into multi-threaded programming was seen as too big of a task to take within the short amount of time available for this project. Taking the single-threaded approach does, however, not hinder multi-threaded or asynchronous behavior of third party libraries. Redesigning the core resource manager around a multi-threaded approach would however be a reasonable

extension for later projects.

Keeping the resource manager single-threaded means that there will be some update loop, where control is given to one module at the time, iterating through them. This does create some extra work for the module developers, as they need to make sure that the operation it does is not blocking the other modules.

The language choice for the client ended up as C++. In order to be performant, it had to be a compiled language. This decision ruled out popular interpreted languages like Python. The fact that C++ is highly optimizable, while still incorporating many high-level features, has made it widespread and much used. Its ability to compile C code also allows for extensions that use C, which is the case for many embedded programs. The main contributor towards this choice is, however, that Unreal Engine, AirSim, and ROS are all written in C++ and have C++ APIs. These are all systems that will be used in the use case example shown in Chapter 5.

The following sections will present the core implementation and design choices of the client, including resource management, module design patterns, and the messaging platform. The last two will be referred to as nodes and events for the rest of the thesis, as this reflects the naming in the actual code.

4.2 Core Structure

The core design of the client is split into four main classes: The client, nodes, events and the event dispatcher. The client is the resource manager, keeping track of the active nodes, distributing events, and running the main program loop. Problem specific work is handled by the nodes, and messages between nodes are handled through dispatched events. Nodes and events are designed to be defined by the user according to a specific task. The nodes are also made to be the link to external dependencies, like external libraries or other program interfaces. In addition to this, the client also features support classes for window and Keyboard input handling through GLFW and logging with the spdlog logging library.

Figure 4.1 shows the relationship between the core classes and how the client interacts with the user-defined classes. As shown in the figure, both the node class and the event class has a pure virtual interface that must be defined for each specific node or event. The overridden functions are then called through virtual function calls in the base node and event class. In the case of this class diagram, only two user-defined nodes and one event are shown. In a realistic implementation there would most likely be multiple different user-defined events, and possibly also more nodes, where each node would interact and handle with multiple events each, while the classes shown inside the "Client core" section will stay unchanged.

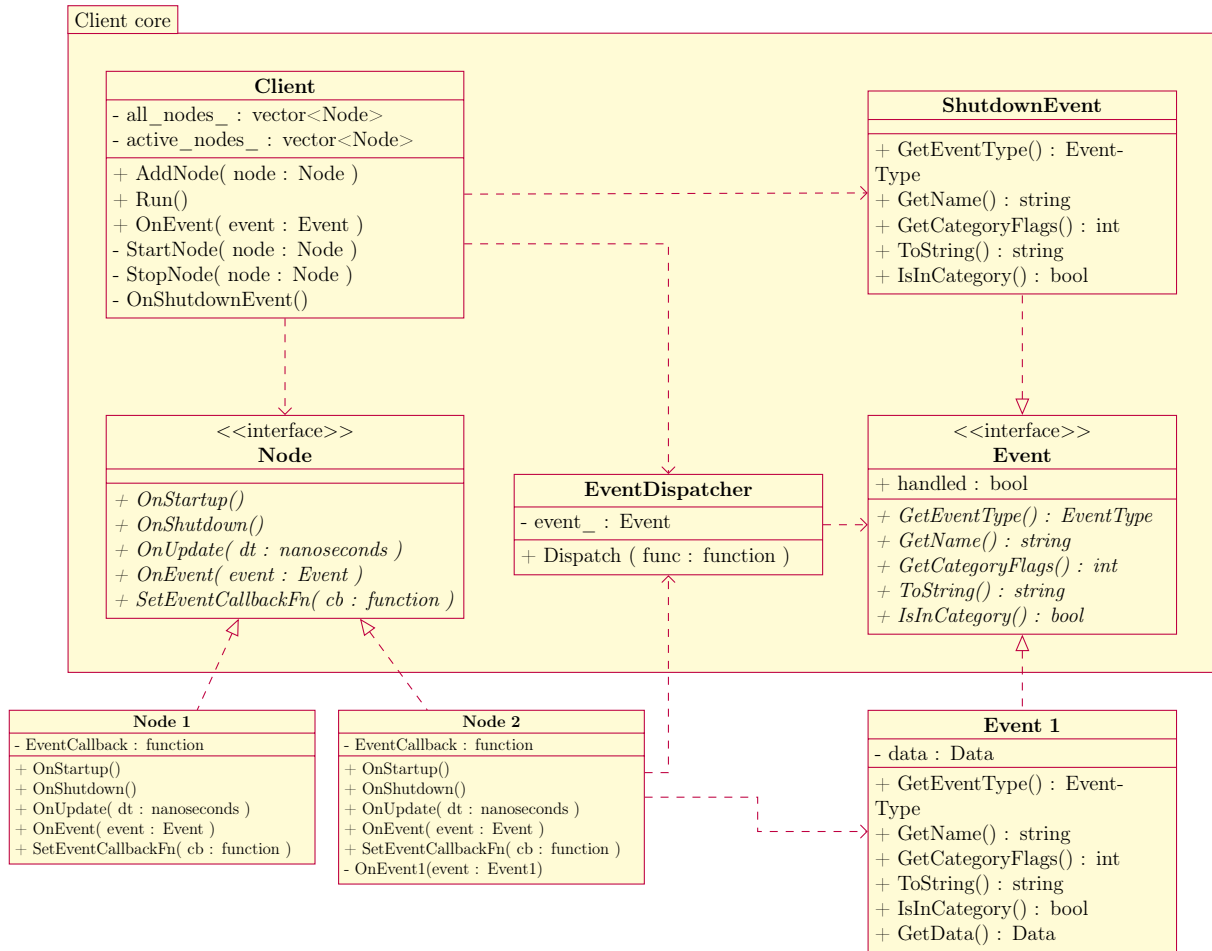


Figure 4.1: Core structure of the client

4.2.1 Use Case Client Specific Setup

As the client should be completely modular in the sense of which nodes to compile, an intuitive way to add nodes to compile is needed. This problem was solved through the function *AddNode*, which adds the node to the client’s active nodes list as well as some initial setup. This is however the only setup needed before starting the client’s run loop. The setup in the main function of the program therefore boils down to:

1. Create client.
2. Construct and add nodes via the *Client::AddNode* function.
3. Call *Client::Run*

4.2.2 Node Interface and Design

The nodes should operate independently of each other. However, there needs to be some standard interface towards the resource manager so it can decide upon the runtime oper-

ation. Through this interface, the client should be able to initialize, pass events, perform the main run loop, and stop the operation of a node. While this is possible to implement through templates, the most descriptive and easily extendable way is through a polymorphic interface. In C++, this is called virtual functions.

As shown in the class diagram in Figure 4.1, the node interface includes five virtual functions:

- *SetEventCallbackFn()*
- *OnStartup()*
- *OnShutdown()*
- *OnUpdate()*
- *OnEvent()*

Since the nodes are initialized after the client's construction, and that the client itself is dependent on the Node header file, the nodes need a way to set where to send created events, or rather which function to call when an event is created. The *SetEventCallbackFn()* is there for this reason. This function is called automatically by the client when a node is added to it through the *Client::AddNode()* function. The function overload is required to store the event function locally so that events can be sent and distributed by the client.

The next decision concerns node-specific initialization and shutdown. While most of the initialization usually can, and should, be done in the class constructor and destructor respectively. There may also be cases that require the client to be in operation. This could, for example, be important initialization, which should cause a shutdown if it fails. In this case, it might be beneficial to delay that initialization step until the client is guaranteed to respond to a shutdown event. This resulted in Two additional overloadable functions called *OnStartup()* and *OnShutdown()*.

OnStartup() is called once for each node, just before the update loop starts, as shown in the startup sequence in Figure 4.2a. This function has access to all events handled by the client, meaning that all initialization that could need to interact with the window, or initialization which could fail, but still require cleanup, should be handled here. The *OnStartup()* function also marks the point in which the node is assumed ready to handle events.

OnShutdown() has a similar role. It allows event creation as a part of the shutdown routine, creating an opportunity to handle disconnection to external dependencies or tell other nodes that it is no longer available. In terms of node state, *OnShutdown()* represents the point in which the node stops handling events.

The last part of the node interface concerns the normal runtime operations of the node. These functions decide what the node does when it gets to run and how it will respond

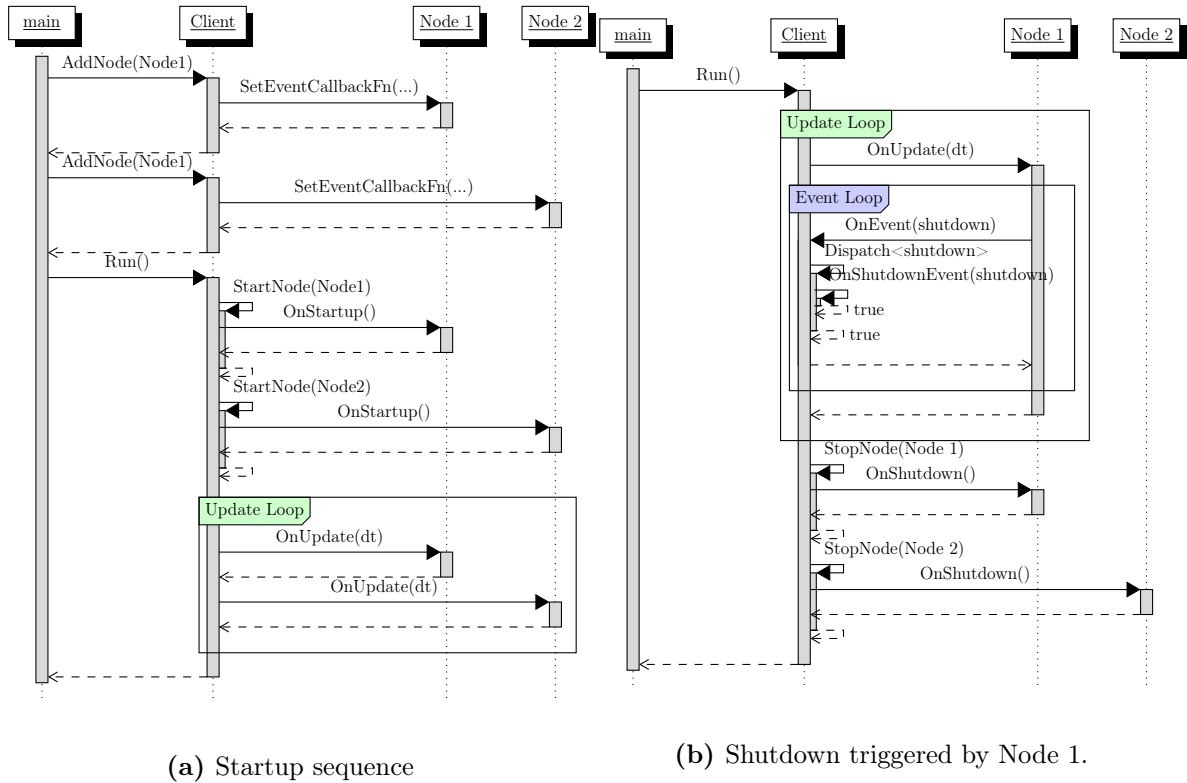


Figure 4.2: Startup and Shutdown Sequence example of the client.

to events. Since the node update is repeating in nature, while the events can occur at varying intervals these are split into two functions: `OnUpdate()` and `OnEvent()`.

The client calls `OnUpdate()` as a part of the update loop. Here the client loops through all nodes repeatedly, calling their `OnUpdate()` function until shutdown. The design of this function should, therefore, contain most of the runtime operations of the node, polling for updates from external interfaces and updating local state. In the case of a ROS node, this would, for example, mean calling `ros::SpinOnce()` in order to run its update loop for handling publishing and subscription callbacks. The update loop is also where most events should be created in order to communicate with the other nodes. `OnEvent()` function is the interface a node has to receive and handle events. This function and how it ties into the event system will be discussed in detail in the following sections.

4.2.3 Event Handling

In order to break the program sequence, and send data between nodes, the client implements a centralized event system, using the `EventDispatcher` class and the `OnEvent()` functions. The two main parts of the event system are differentiating between event types and choosing which event types a node should handle.

To solve the problems with the event handling, the event system was designed directly

based on that of the Hazel game engine¹, which is currently being developed by Yan Chernikov. The event system is only changed slightly to fit the application.

This approach defines a couple of macros to define the type and category of an event, which will be used by the Event dispatcher class to type check and call the correct event handling function defined within the specific node class. This design is quite robust and makes no assumptions on the layout of the event. However, there is some setup needed to be done by the developer in order to use it.

Part of the definition must include the *EVENT_CLASS_TYPE* macro with the specific event type enumeration as an argument. This event type must be defined within the *EventType* enumeration in the *Event.hpp* file. This is also where user-defined event types would be added as they are needed. There is also a macro called *EVENT_CLASS_CATEGORY*, which is optional. This macro adds the event to a specific category, which may be used for filtering out specific events for logging or similar operations, using the *IsInCategory()* function. Filtering on a category could also be used if the user wants to ignore certain categories of events.

There is, however, an unfortunate drawback of the single-threaded approach when it comes to events. As shown in Figure 4.5, the event loop is started when a node calls its event callback function. While this forces the client to handle the event right away, as it is supposed to do, it also blocks the update loop until the event the client or a node handles the event.

Another approach could be to implement an event queueing mechanism, where all events are handled after the update loop. However, this would only delay the problem. It would also remove some of the semantic meaning of an event, where it is supposed to be sudden and force the nodes to react.

Event Distribution and Dispatching

Since the event handling functions are defined as a part of the node definition, the event dispatcher, which chooses the correct event handling function, also needs to be defined locally. This specific implementation of the *EventDispatcher* class also needs to be created for each event and should be the first thing to happen in the *OnEvent()* function of a node. As the event dispatcher only holds a reference to the actual event, this should not amount to any performance loss.

The magic of calling the correct event handling function comes from subsequent calls to the event dispatcher's *Dispatch()* function, as shown in Figure 4.3. This templated function takes in a specific event class type as a template and an event handling function as an argument. The dispatch call then type-checks the templated type to the type of the event. If the types match, the dispatcher calls the function. In this way, the node developer can list all Event types to handle, and how to handle them by adding *Dispatch()* calls.

¹<https://github.com/TheCherno/Hazel>

```

void Client::OnEvent(Event& e) {
    EventDispatcher dispatcher(e);
    dispatcher.Dispatch<ShutdownEvent>([this]() -> bool {
        return OnShutdownEvent();
    });
    dispatcher.Dispatch<WindowCloseEvent>([this]() -> bool {
        return OnWindowCloseEvent();
    });
    dispatcher.Dispatch<WindowResizeEvent>([this](const WindowResizeEvent& e) -> bool {
        return OnWindowResizeEvent(e);
    });
}

```

Figure 4.3: Event Dispatching for ShutdownEvents and WindowCloseEvents in the core client

An advantage to the dispatch interface is that it is easy for a developer to see which events a node responds to, which also makes it easier to reason about the code. The clever type-checking mechanism and smart redirection of events were the main reason for adopting this event system.

Event Handling Function Design

As explained previously the event system type checks the event in each call to *Dispatch()*. This means that at the time of dispatching the event to the event handling function, the type is known. Using this fact, two overloads of the dispatch call have been added, where the difference is the requirement on the event handling function, the actual implementation is shown in Figure 4.4.

The first overload is the one used for events containing data. This type of event could, for example, be an image or a sensor update. In this case, the event handling function to be called needs a reference to the actual event, which is why the dispatch method requires binding to a *std::function<bool(const T&)>*.

The second overload is a special version where the event handling function does not take an argument. This overload should be used for events that carry meaning, but no actual data. An example of this is the *WindowCloseEvent* handled by the client. This event tells the client that somebody closed the window, which in itself is sufficient to describe everything that happened.

The return value of both overloads is a boolean. As seen in the function body of *Dispatch()* in Figure 4.4, this sets the *handled* flag of the event. What this does is that it gives a node the possibility to tell the client that the event needs no more handling and that the event loop should terminate. This is a way to reduce the amount of time spent in the event loop, resuming the update loop. Using this should, however, be done with caution, as lower prioritized nodes will not get the event.

Choosing whether an event is blocked or let through is therefore for the designer of the node to decide. A logging node could, for example, want to handle all events in the sense

```

class EventDispatcher {

    template<typename T>
    using EventFunction = std::function<bool(const T&)>;

    using EventFnNoArg = std::function<bool()>;

public:
    explicit EventDispatcher(Event& event)
        : event_{event} {}

    template<typename T>
    bool Dispatch(EventFunction<T> func) {
        if (event_.GetEventType() == T::GetStaticType()) {
            CLIENT_EVENT(event_.ToString())
            event_.handled = func(*(T*)&event_);
            return true;
        }
        return false;
    }

    template<typename T>
    bool Dispatch(EventFnNoArg func) {
        if (event_.GetEventType() == T::GetStaticType()) {
            CLIENT_EVENT(event_.ToString())
            event_.handled = func();
            return true;
        }
        return false;
    }

private:
    Event& event_;
};

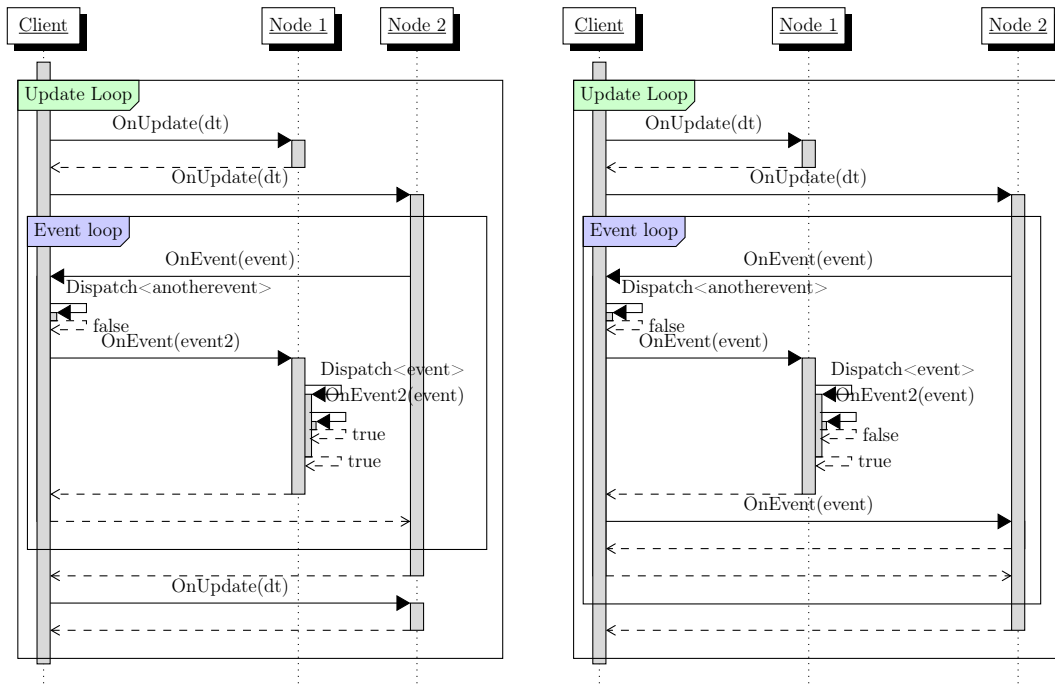
```

Figure 4.4: Implementation of the EventDispatcher class

of adding a log entry. However, it should not block further event handling and hinder the normal operation of the system. However, a node taking control inputs specific for that node would want to block further event handling, reducing the amount of time outside the update loop.

Figure 4.5 shows typical event handling. In both cases the event loop is started by a node creating an event and calling *Client::OnEvent()* through its event callback function. In Figure 4.5a the event is handled and blocked by Node 1, stopping the loop, while in Figure 4.5b the event is first handled in Node 1. However, it is handled as an unblocked event, by returning false from the event handling function, meaning that it is also passed to Node 2. After the last node has finished, the event loop finishes.

The client handles three types of events. These include *ShutdownEvent*, *WindowCloseEvent* and *WindowResizeEvent*. As of now both *ShutdownEvent* and *WindowCloseEvent* causes shutdown process to start. If other behavior is wanted this can be changed in the *Client.cpp* file. As of the current design, none of these events will be blocked. This means that they are all available to the nodes. Even though there should not be any real reasons to handle a *ShutdownEvent* in a node, because of the *OnShutdown()* function call being called as a part of Client shutdown, the decision was made to remove any surprises



(a) Handled and blocked by Node 1

(b) Handled and passed on by Node 1

Figure 4.5: Runtime loop showing Node 2 creating an event, which is handled by Node 1.

from the node developer side, as well as removing a possible reason for needing to change the core code.

Window and Keyboard Interface

One missing element during the early work with the project thesis was a way to control the multirotor in Unreal Engine using the keyboard. While the API comes with multiple API calls that can be used to control the drone programmatically, it is much easier to do quick tests with keyboard control. The main requirement for this is that polling for input cannot be blocking. In other words, a check if a key is pressed must return at once, independent of the key status. This means that the standard `iostream` C++ key interface, `cin` is out of the question, as it requires the user to confirm a keypress.

The easiest way to get key input from the operating system is by tying it to a specific window. The implementation of this is highly operating system specific and would be hard to do correctly. There are, however, many libraries which help with this. Most notable for Ubuntu are `Ncurses`, `Qt`, `GLFW`, and `SFML`. `Ncurses` is the most lightweight library, with its sole purpose being granting access to terminal input. Both `Qt` and `SFML` are quite large libraries providing a ton of functionality, most notably being graphical user interfaces, where keyboard input is only a small part of the libraries. Finally, `GLFW` is a cross-platform window context library made as a base for common graphics APIs like `OpenGL` or `Vulkan`. One can, however, create windows and get access to the keyboard interface.

Early versions of the client were implemented with NCurses, as it takes its input directly from the terminal, meaning that there is no need for an additional window. However, the implementation did not work correctly with multiple keys held down at the same time. This criterion is, however, a necessity for comfortable drone control. For this reason, GLFW was chosen. This choice also enables extending the client to include a debugging GUI in the future.

In addition to the polling-based approach, GLFW also allows for easy coupling of the keyboard inputs to the client's event system. The drawback adding GLFW is that the program now has an additional window to which the keyboard inputs are tied.

The client is currently set up with a window class, separate from the node system, which is a wrapper around the GLFW window API. The window class couples the window functions to the close, resize, and key events of the client. The key events are divided into keypresses- and releases, which are propagated in the same manner as all other events. This means that the key events are available to all nodes at all times.

Logging

String formatting and printouts are very useful in order to debug. However, the standard `cout` can be quite slow if not used correctly. In addition to this, string-formatting can be tedious, time-consuming, and provide much clutter in the code. For this reason, it was decided to use a logging library, specifically `spdlog`[36]. `Spdlog` provides a simple interface for log formatting, differentiating in log levels and messages, file logging as well as both single-threaded and multithreaded loggers. The Hazel game engine referenced in Section 4.2.3 also provided a simple wrapper for `spdlog`, which supplied useful macros for creating different log statements split into different log levels. This logger was also invaluable when while doing benchmarks.

The possible log-levels implemented are:

- Trace
- Info
- Warning
- Error
- Fatal Error
- Debug Event

The only difference between the top four is the output color. Fatal errors are special error level messages which also throw a runtime exception. Lastly, the debug event level is a warning level log entry, which is only shown if the client is compiled in debug mode. This gives the user some choice in where to add messages and what they represent. All macros support the string formatting options provided by `spdlog`.

5 | AirSim Client Implementation

This chapter will present a concrete example of how to use the client and how to implement problem specific nodes to use with external interfaces. Specifically, this implementation will include a node for interfacing towards ROS, one node for receiving sensor data and control a drone in AirSim and a simple node for calculating mean square error based on data received through events. The implementation will also be extended to show how this implementation can access data produced by SVO and ORB SLAM, through their ROS interfaces, as well as visualization of the data through RViz.

5.1 AirSim Client Design

The goal of the AirSim Client is mainly to provide a ROS interface to the AirSim API, which in turn grants access to camera output, sensor data and ground truth measurements related to the AirSim drone run inside an Unreal Engine environment. In addition to this, a simple node for comparing ground truth positional data to the estimated position given by SLAM and VO-algorithms were added, as well as client-side keyboard control of the AirSim drone.

Figure 5.1 shows a simplified view of how the different parts of the AirSim client communicate and how it is connected to ROS and AirSim. Here, the blue nodes represent the core client presented in Chapter 4, while the green nodes represent unmodified external code.

Currently, the ROS interface only supports the different image types described in the "ImageCaptureBase.hpp" file in the source code of AirSim as well as IMU data. The AirSim API itself, however, supports a barometer, GPS, magnetometer, distance sensor, and Lidar. These can be added to the client at a later date, through adding events and ROS publishers with associated topics.

The AirSim client is split into two nodes which run simultaneously: The AirSim node which handles all communication with the AirSim client API and the ROS node which handles all of the ROS publishing and subscriptions. With this project, two different versions of the AirSim node is used. One is targeting the Computer vision(CV) mode, and the other is targeting the Multirotor mode of AirSim. The following sections will get into the details of the design of each node and custom events, showing how it ties into

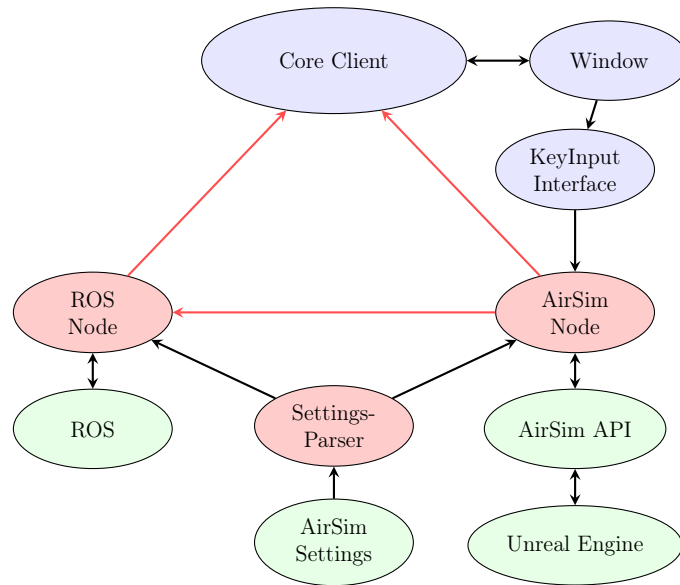


Figure 5.1: Simplified overview of the information flow of the AirSim client. Red bubbles refer to application specific code, blue bubbles represent core client functions and the green bubbles are external dependencies. black arrows represent information flow through normal function calls, while red arrows represent event based communication.

the core client runtime operation.

5.1.1 AirSim Client Events

The common factors between the nodes are that they both handle images, IMU data and Transform messages in some form, and will form the base events for this implementation. In addition to this, some extra info about the camera is transferred with the image event. This information includes the type of image sent, the intrinsic parameters of the camera, and the frame id and transform of the camera. Since this amount to very little extra data compared to the image, as well as them being so tightly connected, there was no reason to split them into separate events.

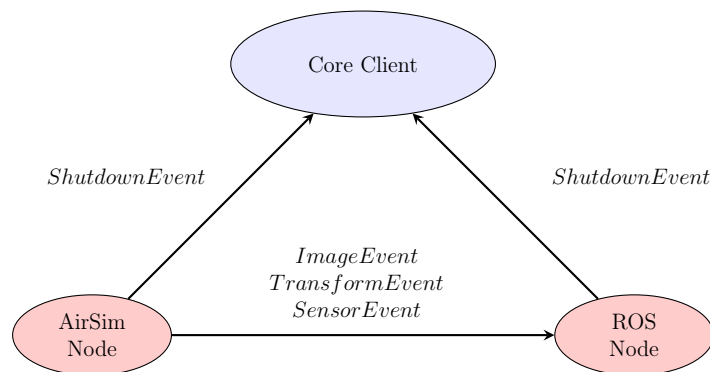


Figure 5.2: Overview of the event information flow with associated Event names.

As Figure 5.2 shows, the events are designed to follow the flow of:

-
1. Fetch data from AirSim
 2. Send data to RosNode
 3. Publish data to Ros
 4. Receive Transform estimates over ROS

5.1.2 AirSim Node

The AirSim plugin to Unreal Engine creates an RPC server, which runs alongside the environment to allow external communication over TCP/IP networks. Alongside this AirSim provides an API with a set of commands to read sensor data and control the vehicle inside of the Unreal Engine environment. It is this client API the AirSim node is connected to. This is true for both the ComputerVision node and the Multirotor node. The differences come in the form of the available functionality of the simulation modes.

In order to select simulation mode, the SimMode variable needs to be set in the settings.json file, as described by their own documentation[37]. The client is also made to parse this settings file, and select the correct node to start based on this setting, with the multirotor node being set as a default, if no simulation mode is supplied. A typical settings file configuration file setup for AirSim is shown in Figure 5.3.

```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,

  "SimMode": "Multirotor",
  "ViewMode": "SpringArmChase",
  "ClockSpeed": 1.0,

  "Vehicles":
  {
    "multirotor_1":
    {
      "VehicleType": "SimpleFlight",
      "DefaultVehicleState": "Armed",
      "EnableCollisionPassthrough": false,
      "AllowAPIAlways": true,
    }
  }
}
```

Figure 5.3: Partial setting.json file for AirSim configuration.

In ComputerVision mode AirSim does not simulate any vehicle, and there are therefore no sensors associated with it except for a camera. The camera position is also controlled through the keyboard inside of Unreal Engine, without any involvement from the client. The API used for controlling the multirotor from the client is disabled for the CV mode, which means that there is no way to enable remote control.

The node itself is set up to receive a single RGB image each update loop and create an ImageEvent with the image. This means that the ComputerVision node operates as a channel supplying a constant stream of images from AirSim.

From this, it can be seen that the Multirotor node supports more features than the CV node. It supports multiple cameras, client-side multirotor control, Imu data, and fetching

of ground truth kinematics of the multirotor. However, the CV node is not tied to the physics engine of AirSim, meaning that it can, for example, be used for camera calibration or gathering datasets quickly. Just like the multirotor node, this node also parses the full settings.json document to find the specific vehicle setup in AirSim, to find which cameras are available.

One significant problem encountered with the AirSim node is that the amount of time it takes to get a picture is around $80ms$. This low framerate hinders the regular operation of the node. The fact that this call to AirSim is synchronous, also means that the control loop for the multirotor is delayed, causing unresponsive controls. This is unfortunately tied to the Unreal Engine side of the simulation, meaning that there is no easy fix.

5.1.3 ROS Node

The ROS node is designed around ROS-operations, with its node handle, subscribers and publishers. It keeps track of the different publishers needed to send images, transforms and camera info topics over ROS, in order to supply these to ORB-SLAM2 and SVO.

In order to create the correct publisher for publishing sensor data from AirSim, the ROS node also parses the settings.json file and then creates separate publishers and topics based on the vehicle setup. In the Event handling functions for sensor data, the correct publisher is chosen based on sensor type, sensor name, and associated vehicle.

5.2 Static Simulation Setup

As discussed, the goal of the AirSim client setup provides images for VO and SLAM algorithms through ROS messages, while also providing ground truth data for comparison. Since RGB images, depth images and vehicle transforms are available directly through the AirSim API, the only setup needed is to define the coordinate frames, and set their relative transforms.

In order to reduce the number of slow calls to the AirSim API, we can store some transforms that are stationary throughout the simulation. The map frame will be used as the reference frame, and for simplicity, the multirotor will be set to spawn at the origin. This means that the transform from the map frame to the multirotor spawn frame is equal to the identity matrix. This is shown in Equation (5.1), with m referring to the map frame, and s referring to the spawn frame.

$$\mathbf{T}_s^m = \begin{bmatrix} \mathbf{R}_s^m & \vec{\mathbf{p}}_{ms}^m \\ \vec{\mathbf{0}}^\top & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.1)$$

Another stationary transform is the camera frames relative to the multirotor. Based on the settings file described in Section 5.1.2, the camera is known to be positioned 46cm along the north axis of the multirotor throughout the simulation. Using this information, two frames relating the camera to the multirotor will be defined.

The camera body frame, \bullet^c , which will keep the NED convention, and a rotated optical frame, \bullet^o , which follows the most common optical frame convention. This frame will have its x-axis to the right in the picture, y-axis downwards and z-axis into the picture. The optical frame rotation is shown in Equation (5.2), and the full transform from optical to vehicle frame is shown in Equation (5.3).

$$\mathbf{R}_o^c = \mathbf{R}_x\left(\frac{\pi}{2}\right) \mathbf{R}_y\left(\frac{\pi}{2}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (5.2)$$

$$\mathbf{T}_o^v = \mathbf{T}_c^v \mathbf{T}_o^c = \begin{bmatrix} \mathbf{R}_c^v & \vec{\mathbf{p}}_{vc}^v \\ \mathbf{0}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_o^c & \vec{\mathbf{p}}_{co}^c \\ \mathbf{0}^\top & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0.46 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{for } \vec{\mathbf{p}}_{vc}^v = \begin{bmatrix} 0.46 \\ 0 \\ 0 \end{bmatrix} \quad (5.3)$$

The last thing which is needed is the intrinsic parameters of the camera as these are needed to calculate the projection matrix. While the AirSim API does provide a projection matrix, it is the one used by Unreal Engine for its graphics processing, projecting a view frustum into a known volume. While they are related, they are not directly convertible. Seeing that the AirSim settings file gives the field of view, image resolution, and aspect ratio, it is just as easy to calculate the projection matrix once and store it.

$$\begin{aligned} \vec{\mathbf{p}}^p &= \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z} \mathbf{K} \vec{\mathbf{p}}^o = \frac{1}{Z} \begin{bmatrix} \frac{W}{2} & 0 & \frac{W}{2} \\ 0 & \frac{H}{2} & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\tan(\frac{\theta}{2})} & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta}{2})} & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{\mathbf{p}}^o \\ &= \frac{1}{Z} \underbrace{\begin{bmatrix} \frac{W}{2\text{tan}(\frac{\theta}{2})} & 0 & \frac{W}{2} \\ 0 & \frac{H}{2\text{tan}(\frac{\theta}{2})} & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \vec{\mathbf{p}}^o = \frac{1}{Z} \underbrace{\begin{bmatrix} \frac{W}{2\text{tan}(\frac{\theta}{2})} & 0 & \frac{W}{2} \\ 0 & \frac{H}{2\text{tan}(\frac{\theta}{2})} & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \end{aligned} \quad (5.4)$$

Equation (5.4) shows the intrinsic matrix \mathbf{K} in relation to the projection from a point $\vec{\mathbf{p}}^o$ given in the camera's optical frame, to pixel coordinates, denoted $\vec{\mathbf{p}}^p$. Here W and H refer to the width and height of the image in pixels and θ begin the field of view. The projection matrix \mathbf{P} given by Equation (5.5). Note that the translation portion is the zero vector. This is because $\vec{\mathbf{p}}^o$ is given in the optical frame.

$$\mathbf{P} = [K \quad \vec{\mathbf{0}}] = \begin{bmatrix} \frac{W}{2 \tan(\frac{\theta}{2})} & 0 & \frac{W}{2} & 0 \\ 0 & \frac{H}{2 \tan(\frac{\theta}{2})} & \frac{H}{2} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (5.5)$$

5.3 Visualization

There are four main elements of this simulation which is to be visualized:

- Ground truth map point cloud
- VO/SLAM estimated map point cloud
- Ground truth position in the simulated map
- Estimated position in the map coordinate frame

The client itself does not provide any visualization tools, so in order to show the live operation, RViz is used. RViz is a visualization tool for ROS topics. It supports visualization for many of the common ROS messages, for data such as point clouds, transforms, and images. In order to create a 3D visualization, it uses the frame id found in the standard ROS message header to keep track of relative positions.

This section will focus on the setup related to the visualization of data, while the results will be shown in Chapter 6 along with the client benchmarks.

5.3.1 Rviz Map Frame

RViz uses a north-west-up coordinate representation for its visualizations, while AirSim uses a north-east-down representation. This means that the visualization will appear flipped upside-down if the map frame from AirSim is tied directly to RViz. The easiest way to solve this was through a *static_transform_publisher* provided by the *tf2* library to ROS. This can be launched as its own ROS node providing an additional rotated coordinate frame to use as the map frame in RViz. Using Equation (2.8), with $\theta = \pi$ and $\vec{u} = [1, 0, 0]^\top$, we get the rotation shown in Equation (5.6). Here R represents the Rviz map frame, and A represents the AirSim map frame.

$$\vec{\mathbf{q}}_{R,A} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos(\frac{\pi}{2}) \\ u_x \cdot \sin(\frac{\pi}{2}) \\ u_y \cdot \sin(\frac{\pi}{2}) \\ u_z \cdot \sin(\frac{\pi}{2}) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (5.6)$$

The launch file for the static transform publisher is shown in Figure 5.4, where the arguments are given in the order:

```

<launch>
  <!-- Transform between NWU and NED -->
  <node name="ned_to_nwu_pub" pkg="tf" type="static_transform_publisher"
        args="0 0 0 1 0 0 0 world_ned world_nwu 100"/>
</launch>

```

Figure 5.4: ROS launch file for publishing transform between North-East-Down and North-West-Up frames. Arguments in order: $p_x, p_y, p_z, q_x, q_y, q_z, q_w$, parent frame id, child frame id, republish period.

5.3.2 Ground Truth Map Visualization

While the ground truth pose of the simulated multirotor is directly available with the AirSim API, the map itself is not. Ground truth depth images are however available. This means that the distance from the simulated camera, to the objects in the map, is known. Using this as well the transform to the camera frame, one can make a ground truth point cloud for the map.

Through the package, *depth_image_proc* for ROS, a nodelet is defined to do just this. The *point_cloud_xyzrgb* takes a depth, image, a point cloud, and a camera info ROS message, and publishes a point cloud as a *PointCloud2* ROS message, which can be shown directly in RViz. The launch file for this nodelet is shown in Figure 5.5, with added remapping of topics to match the ones published by the client.

```

<launch>
  <node name="depth_to_pointcloud_manager" pkg="nodelet" type="nodelet" output="screen" args="manager" />

  <node name="depth_to_pointcloud_node" pkg="nodelet" type="nodelet" args="load depth_image_proc/point_cloud_xyzrgb
  depth_to_pointcloud_manager" >
    <remap from="depth_registered/image_rect" to="/multirotor_1/front_center/depth_planner/image_raw"/>
    <remap from="depth_registered/points" to="/multirotor_1/front_center/depth_planner/points"/>
    <remap from="rgb/image_rect_color" to="/multirotor_1/front_center/scene/image_raw"/>
    <remap from="rgb/camera_info" to="/multirotor_1/front_center/scene/info"/>
  </node>
</launch>

```

Figure 5.5: ROS launch file for point cloud publisher.

5.3.3 Estimated Map and Transforms

Both SVO and ORB-SLAM2 have ROS interfaces that can be run through the ROS launch file system. This means that most of the setup needed in order to run these with AirSim is complete. The only needed is to match coordinate frames, set which cameras to use and supply the camera's intrinsic parameters. For this simulation, the intrinsic parameters in the config files will be set to match the ground truth parameters of the camera. In this case, that is equal to the undistorted pinhole model, with the image dimensions of 512 by 512.

Using the matrix K from Equation 5.4, with a FoV of 90° and image dimensions of 512×512 , we get:

$$K = \begin{bmatrix} \frac{W}{2 \tan(\frac{\theta}{2})} & 0 & \frac{W}{2} \\ 0 & \frac{H}{2 \tan(\frac{\theta}{2})} & \frac{W}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad (5.7)$$

6 | Results

The first two sections of this chapter will contain the benchmarks for the core client and the AirSim client, respectively. Here the benchmarks are run more vigorously for the core client, as that is the main topic of this thesis. For the AirSim implementation, only the main update loop will be timed, together with bottleneck findings. The last section shows the operation of the AirSim client while applying ORB-SLAM2 and SVO to the generated image data.

6.1 Core Client Benchmarks

This section will present the benchmarks of the core client. For each category, they will be qualitatively compared to a similar ROS implementation. Since the two systems do not use the same messaging protocol or design approach, there are bound to be differences in the implementation. However, the tests aim to apply semantically similar operations in both systems, even though the underlying code is different.

ROS also provides a node type called nodelets, which provide about the same interface but can be compiled together, in order to allow for some optimization. The comparisons towards ROS will be made towards both ROS nodes and ROS nodelets, in order for the benchmark comparisons to be realistic.

6.1.1 Common benchmark setup

All benchmarks have been done on a computer running Ubuntu 18.04, and the client code has been compiled using the clang 6.0 compiler. In order to limit external factors, the PC was not used for any other purposes while the benchmarks were running. Some relevant computer specs are posted in Table 6.1.

The tests performed in this section will measure the time it takes for a node to generate an event, send the event to another node and receive a confirmation from the other node, in the form of a reply message or event.

Each test is run in groups of 1.000.000 events, where the test is rerun between two and

Table 6.1: Hardware setup for benchmarks

Type	Specification
Processor	AMD Ryzen 5 1600X 6x3.6GHZ
RAM	HyperX Fury DDR4 2666MHz
GPU	Nvidia GeForce 1080Ti

five times to average out the runtime specific differences as best as possible. This is also true for the ROS nodelets. However, the number of messages sent had to be reduced for the ROS node data transfers with many nodes spawned, as some of the tests would have taken multiple days for each run. The results which deviate from the setup described here will be pointed out in the later sections.

One thing to note is that the timing results were significantly slower for the first 1-5000 cycles of the tests. Some were up to ten times as slow. Based on this observation, it was decided to discard the first 10000 timing results for each run, as the relevant part is the long-time operation of the system and not the start-up process.

Node setup

In each of the benchmarks, only one node will be created as an initial publisher. Since the node which sends the message is also the one that receives the confirmation, it is also the only node that does the actual logging. This was done to reduce the amount of effect the actual logging has on the test itself. As an additional effort to assure as little external impact on the tests as possible, the nodes are set up to do only one of three tasks:

1. Publish initial message and wait for response
2. Listen for message, and not respond
3. Listen for message, and respond

Also, there is some difference in the types used for timestamps for ROS nodes and Client events. For time handling, the client utilizes the Chrono library, while ROS uses its own time library. While the implementation of these differs, it felt most natural to use the time library, which would most likely be used for the different applications.

6.1.2 Transfer of simple message

In order to test how both systems handle the transfer of small messages, the first test is using a ping-type message and a response-type message. Both messages consist of:

- sender id (int32)

- receiver id (int32)
- message id (int32)
- timestamp (chrono time point / ros Time)

Figure 6.1 shows the average time to send the simple message and receive confirmation. This can be seen to scale linearly with the number of nodes created. Compared to the actual message transfer time, this effect is quite significant, as the transfer time is almost doubled in the sixteen nodes test, as opposed to two. For the ROS node, this happens as early as for ten nodes.

An interesting result is the fact that Figure 6.1b shows that the client has a significant outlier at 50 created nodes, while an outlier at 32 nodes can be seen in Figure 6.1a for the ROS node. These are not common across the different implementations but are consistent in all five runs done for each type.

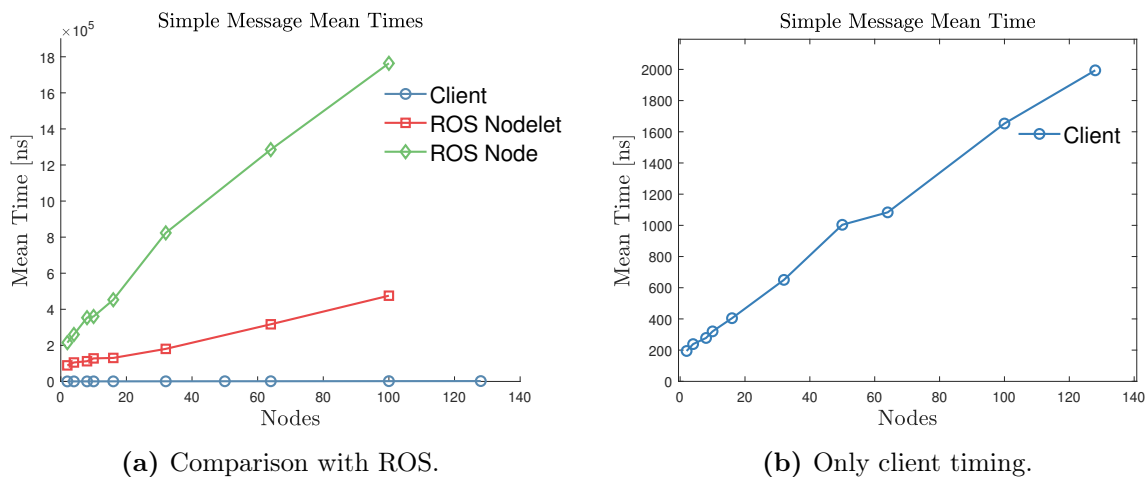


Figure 6.1: Timing comparison of simple message mean time, between ROS and the client.

Figure 6.1a also show that the client implementation is significantly faster than the ROS node implementation and that it scales better with the number of nodes. The nodelet approach, while being much closer to the client in implementation and speed, also contains significant overhead, and scales worse than the client.

6.1.3 Transfer of large data

To test the speed of large data transfers, a similar test to the simple message transfer was used. However, the message was changed to also contain a 1080x1080 size vector, in addition to the other information. This vector can, for example, represent a 1080x1080 greyscale image, or a 512x512 RGBA image, which would be a common task for the client to transfer. The whole message contains:

- sender id (int32)

- receiver id (int32)
- message id (int32)
- timestamp (chrono time point / ros Time)
- data (vector / variable size array)

Note the difference in the data container between the event and the ROS message. The client uses a vector of floats, while ROS uses its variable size array type defined for the ROS message system. These are the most comparable large containers, as they both support an arbitrary amount of elements, decided at runtime. Note that the test only transfers the vector one way and that the response is the same simple message type described in the previous section.

Table 6.2: Amount of samples taken each run and amount of runs per implementation type.

	Runs	Amount of samples				
Nodes		2	4	8	10	16
ROS node	2	1.000.000	500.000	200.000	200.000	100.000

As shown in Figure 6.2, this took significantly more time across all implementations. Due to this, only the client tests were run for 128 nodes. As seen in Figure 6.2a the ROS nodes scaled really poorly with this test. Seeing that the average transfer time of the message for 16 nodes was close to $25\mu s$. Since each run of a million samples for 16 nodes would take around 7 hours, the number of samples had to be reduced. The number of samples and the number of individual runs are shown in Table 6.2. The rest of the nodes were run with one million samples each run.

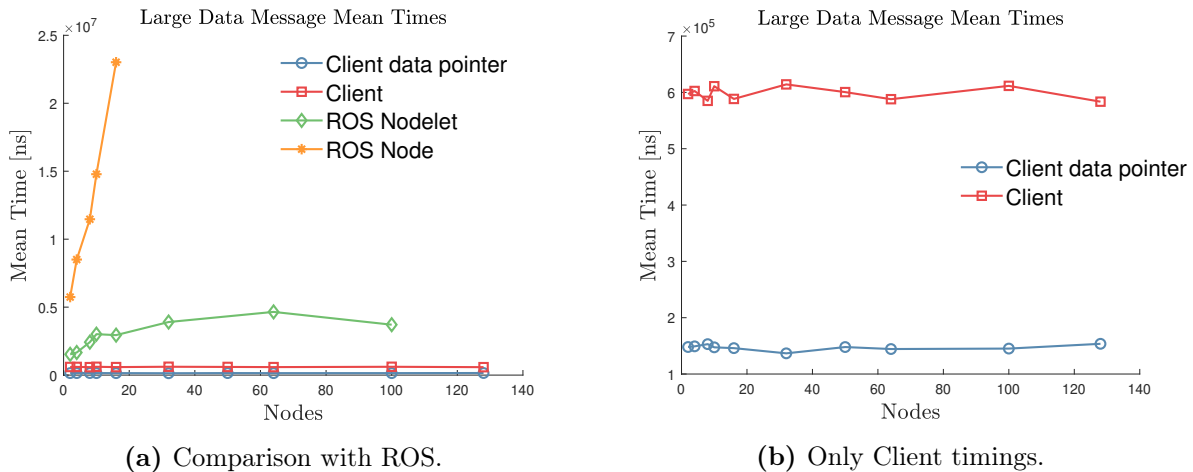


Figure 6.2: Timing comparison of large data transfer mean time, between ROS and the client.

In all cases, except for the ROS nodes, we see that the amount of nodes has little to no impact on the performance. This behavior is more shows more easily in Figure 6.2b, where the deviations are sporadic rather than dependent on the number of nodes created. The scaling of the ROS nodelets also seems to be flattening out at around the 32 node point, even decreasing a bit at 100 nodes created.

6.1.4 Sample distributions for client benchmarks

Looking at the distribution in Figure 6.3, we see much of the same as in Figure 6.1b. However, looking at the box plot in Figure 6.3b, an increase in slope can be seen. One should also note that while the plot shows very consistent results, the consistency decreases with the increased number of nodes. However, even in the worst case of 128 nodes, it can be seen that the first and third quartile is within $250ns$ of each other. Another interesting observation in Figure 6.3a is that the outliers of the benchmarks for 50 nodes are a lot more spread than for the rest.

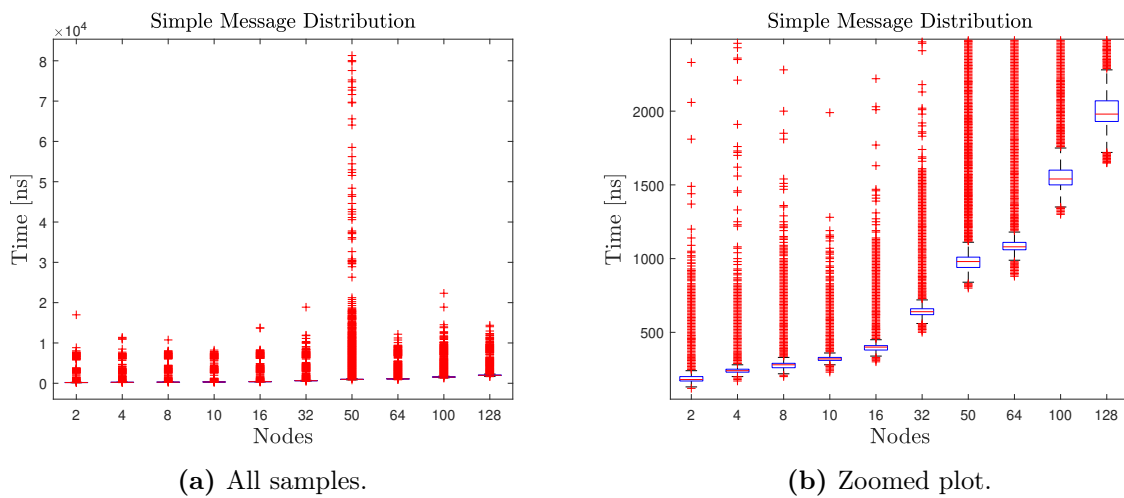
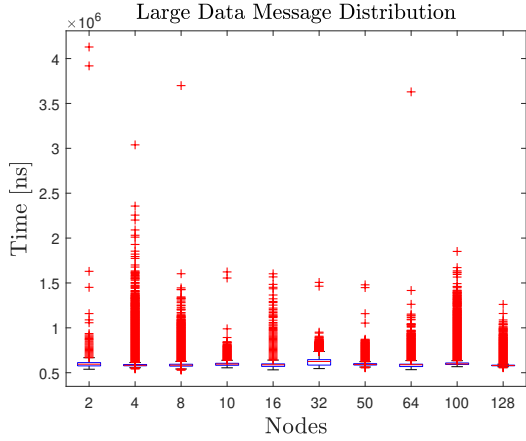


Figure 6.3: Boxplot of the timing sample distribution for small message transfer.

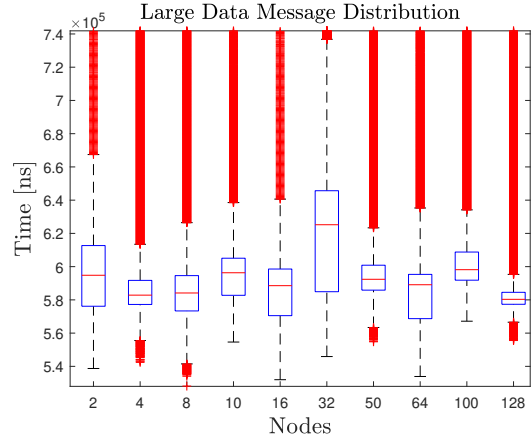
For the large data benchmarks, in Figure 6.4b, it can be seen that there is no significant increase in processing time based on the number of nodes. One should, however, note that while the values deviate more in the large message tests, the relative deviation scaled the median is smaller than for the small message tests. Another observation is that the results are more inconsistent with each other, showing a significantly slower response with 32 nodes. However, one should take into account that this test was run only twice, as opposed to 5 times, which was done for the small messages and the data-pointer messages.

In Figure 6.5 way more consistent results can be seen, both in terms of quartile placement, median and outlier boundaries. As with the mean values shown in Figure 6.2b, it can be seen that the performance of the data-pointer message is about four times faster than that of the data copy.

An observation concerning all three tests is that there does not seem to be any correlation between the outliers in the figures and the number of nodes created. However, for 32 nodes, we see a high median in both Figure 6.4b 6.5b, showing that there is a higher spread in the lower 50% of the values.

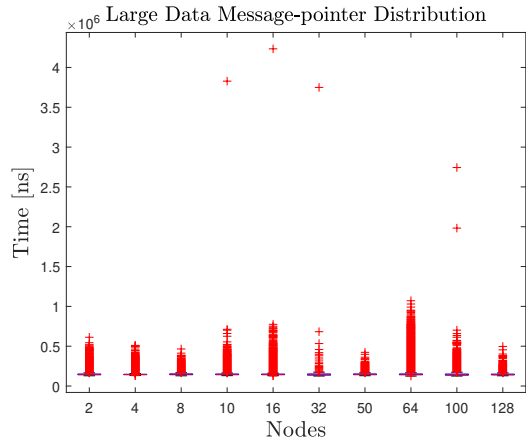


(a) All samples.

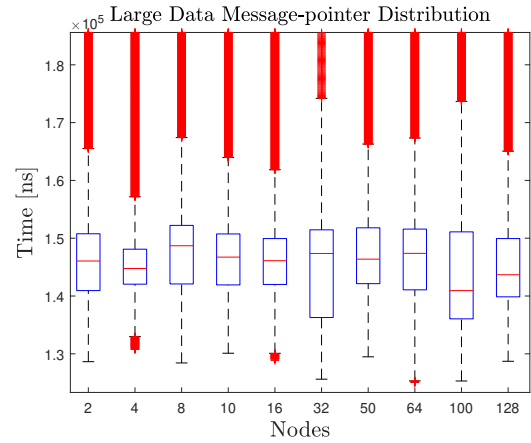


(b) Zoomed plot.

Figure 6.4: Boxplot of the timing sample distribution for large data transfer.



(a) All samples.



(b) Zoomed plot.

Figure 6.5: Boxplot of the timing sample distribution for transfer of data by a shared pointer.

6.2 AirSim Client Timings

The Airsim client implementation is heavily reliant on the external operations of AirSim, Unreal Engine, and ROS. Operations which are uncontrollable from the client’s perspective. The Airsim client implementation also needed to run at a very low framerate in terms of images delivered to the ROS interface. For this reason, there was an incentive to find the bottleneck or bottlenecks in the implementation.

For these tests, the framerate was set to five frames per second, while the update loop was unconstrained. This means that the control loop, ROS publishing and subscription, Transform calculation, and event system runs as fast and often as it is able.

For these tests, only 50000 samples were gathered, where the first 5000 were discarded, for the same reasons as explained in Section 6.1.1. The mean, standard deviation, and median for both measurements are shown in Table 6.3. Note that the measurements for

Table 6.3: Comparison of statistical data for the image fetch times, compared to the update loop of the system.

Type	Image Fetching Delay	Update Loop
Mean	80, 12ms	2, 01ms
Standard deviation	5, 60ms	9, 39ms
Median	80, 39ms	0, 32ms

the update loop also contains fetching of two images every 200ms.

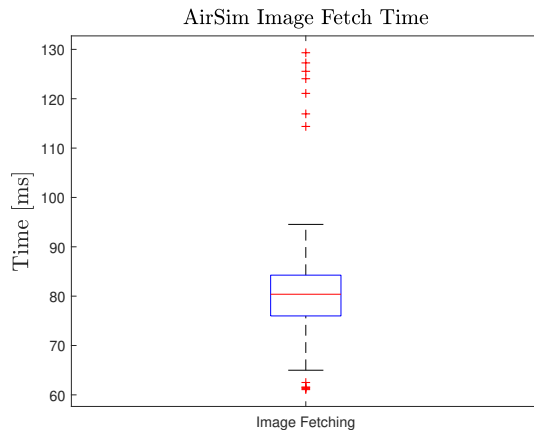


Figure 6.6: Timing data for the fetch time of two images through the RPC interface of AirSim.

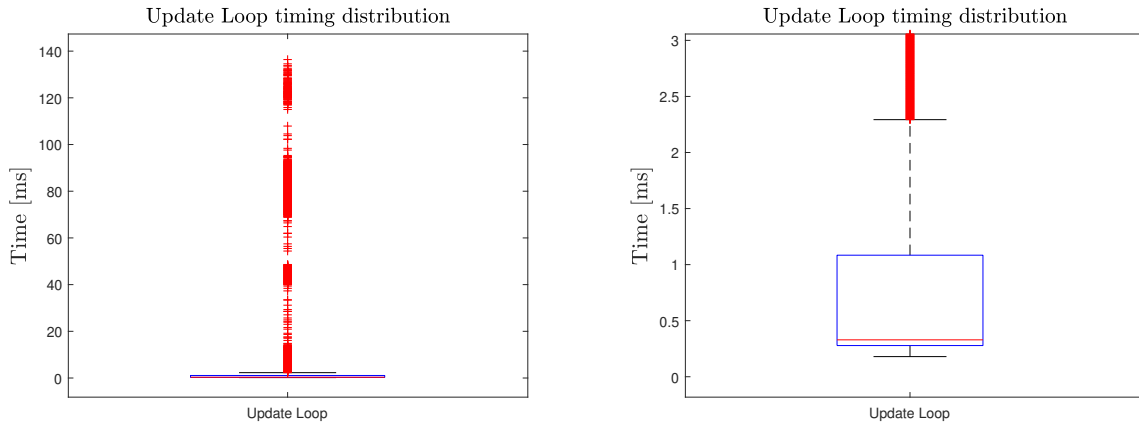
The average delay in Table 6.3 show a considerable bottleneck in terms of the image-fetching. As the time it takes from the RPC call starts to the RGB and depth image is received is around 80ms on average. Looking at Figure 6.6, the fetch times also seem to be normal-distributed around this mean, never going below 60ms. The rest of the update loop is comparatively much faster.

Figure 6.7b shows that the first quartile and the median are very close to each other. This result is to be expected, as based on the 200ms delay between each image fetch call and the update loop median in Table 6.3, there should be about 250 samples of the fast update loop per update loop with image fetching.

This does, however, mean that there are many samples of the fast loop which lie above the third quartile. Comparing Figure 6.6 and 6.7a a significant portion of samples lie between the 60ms mark of the fetch times, and the 1.2ms mark around the third quartile.

6.3 Simulations

Both Orb-SLAM and SVO were attempted to run in two different environments: The Unreal Engine Temple example environment, seen in Figure 6.8, and on a full-size 3D model of the Skarv FPSO[38], seen in Figure 6.9. As in-depth error metrics is outside the scope of this thesis, only a visual comparison will be performed.



(a) All samples.

(b) Zoomed.

Figure 6.7: Timing data for the whole update loop of the client, including image fetching.

All simulations were done in real-time, with AirSim connected to ROS through the client setup described in Chapter 5, using the standard multirotor model and flight controller API, provided by AirSim. SVO was run with one monocular camera pointing downwards, as they recommend on their wiki pages [39], and ORB-SLAM2 was run with both monocular RGB and RGB + depth.



(a) Outside pavillion



(b) Inside

Figure 6.8: Unreal Engine temple environment.



(a) Whole model in UE4.



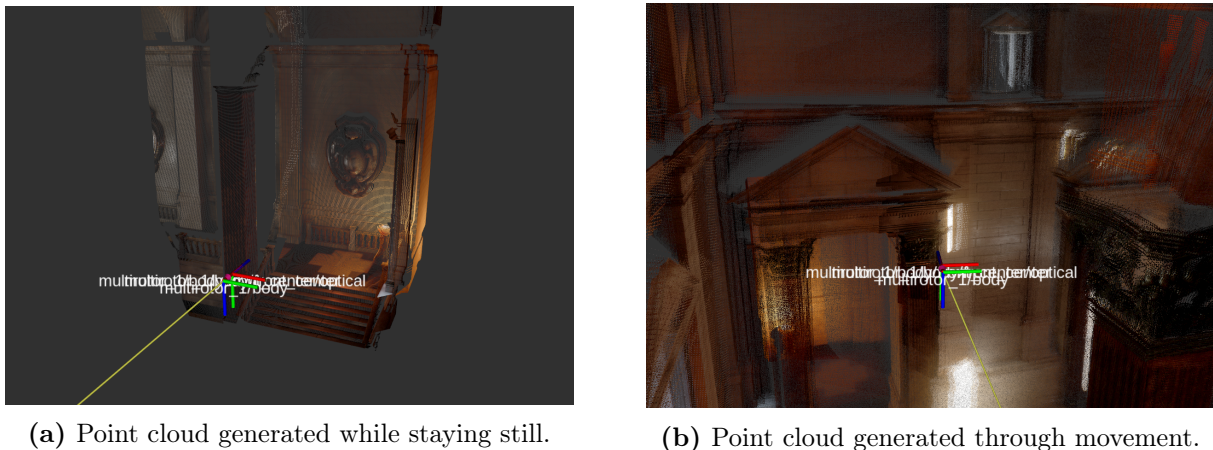
(b) Detail view

Figure 6.9: Live size 3D model of Skarv FPSO

6.3.1 Unreal Engine Temple Simulations

The temple environment is shown in Figure 6.8 is a relatively small scene, providing both an indoor environment, as well as a semi-outdoor environment. This allows for an excellent basis to see how SLAM and VO algorithms perform in low-light conditions, as well as direct or indirect sunlight.

In order to establish a visual comparison for the mapping of ORB-SLAM2, a point cloud was created from the ground truth data gained from AirSim. Using a depth image and the pose of the multirotor, an RGB image can be mapped to a dense point cloud, as described in Section 5.3.2.



(a) Point cloud generated while staying still.

(b) Point cloud generated through movement.

Figure 6.10: Point cloud generation from ground truth image and pose.

The technique does, however, not provide an accurate map, as can be seen in Figure 6.10. On the left-hand side, in Figure 6.10a, the image can be seen projected onto a point cloud while standing still in the air. This allows the depth image and pose measurements to synchronize, providing an accurate result. In Figure 6.10b, however, one can observe significant blur. Especially around the pillar on the bottom right. This effect is induced whenever there is significant movement of the multirotor. Generating a dense point cloud of this size also affected the performance of the flight controller. In order to counter this, a decay time was set up, so that old points were deleted.

Using the same technique, one can generate a map for the whole scene. The parts of the map used in these simulations are shown in Figure 6.11. As mentioned above, the point cloud itself cannot be used as ground truth. It is, however, sufficient to use as a visual comparison to the estimated data.

ORB-SLAM2 RGB

The map in Figure 6.12 is generated by the ORB-SLAM2, without any depth information. Comparing it to Figure 6.11, it can be seen that the shape of the map is quite similar. The relative sizes of the rooms and corridors are also quite correct. The only clear discrepancy is the corridor just below the picture center.

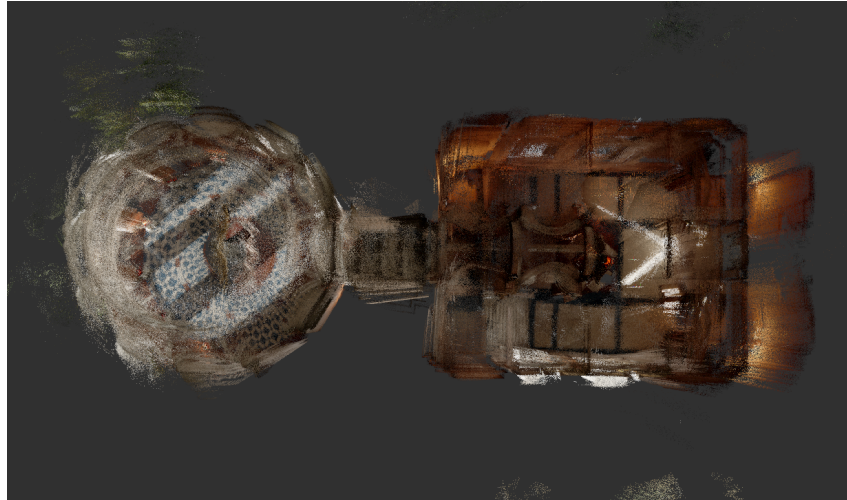


Figure 6.11: Map point cloud overview, generated directly from ground truth depth image and pose

There are a couple of points to take note of in the map: These are the topmost circular wall, the right-hand side wall and the corridors on the bottom half of the map. These are all in which ORB-SLAM2 struggles to place map points. It shall, however, be said that there was little to no problems when it came to feature tracking, as long as the multirotor did not induce any quick rotations.

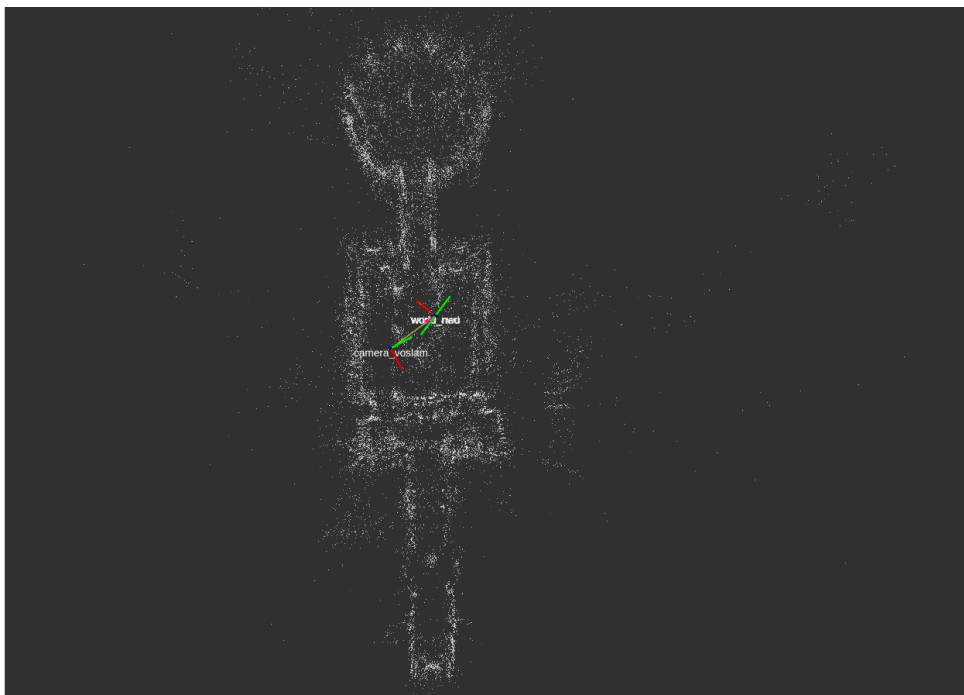
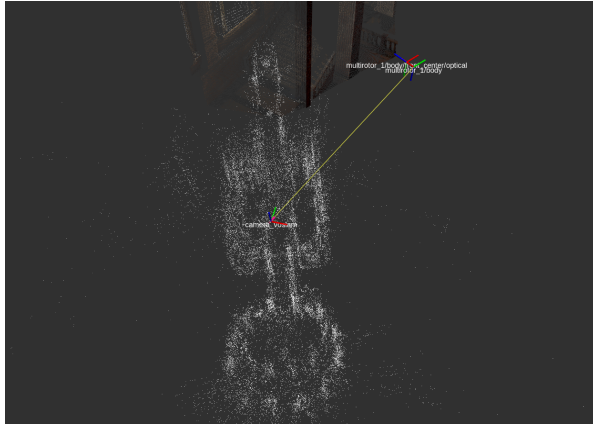


Figure 6.12: ORB-SLAM2 generated map, using a monocular RGB camera.

Figure 6.13 shows how the map and estimated pose compares to the ground truth data. The multirotor is flying at the position shown in Figure 6.13b, while the estimate is shown in Figure 6.13a. It can also be seen that the scale and rotation is off. However, it is not possible to estimate this using monocular SLAM.



(a) Estimated pose in generated map.



(b) Multirotor placement in temple environment.

Figure 6.13: Comparison of estimated pose and actual pose.

ORB-SLAM2 RGBD

Adding a depth image as an input to ORB-SLAM2 made the initial mapping very close to the ground truth measurements, as seen in Figure 6.14. Comparing these simulations to the previous one, it is also seen that the rotation has been correctly estimated. This is also shown in Figure 6.15b, where the corner is placed almost perfectly.

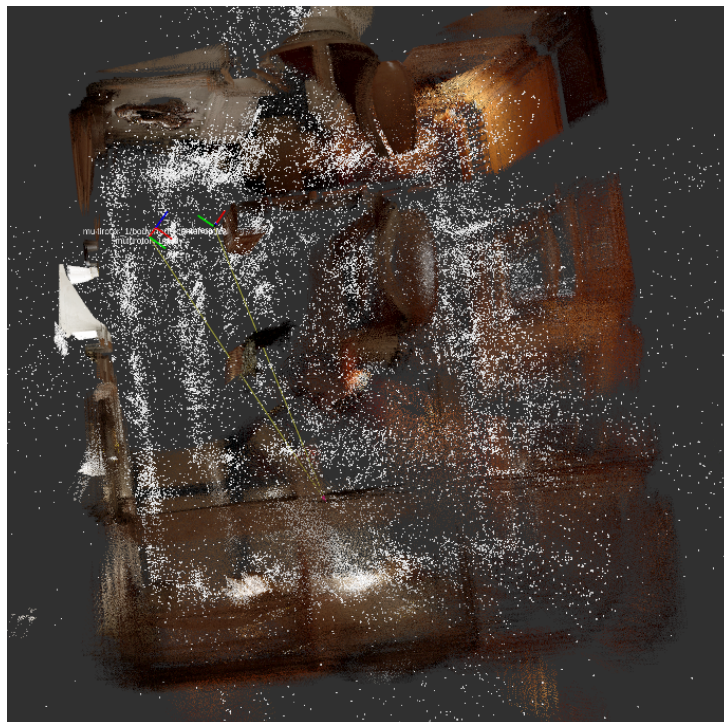


Figure 6.14: Comparison of the scale and placement of the estimated map.

However, during tracking, it was significantly easier to lose the tracked features in this setup. Especially around the corridors on the bottom half of Figure 6.12. The results of this were generation of smaller maps within the map, as seen in the left part of Figure 6.14

and the right side of Figure 6.15a. The longer the simulations lasted, the more of these nested maps were generated.

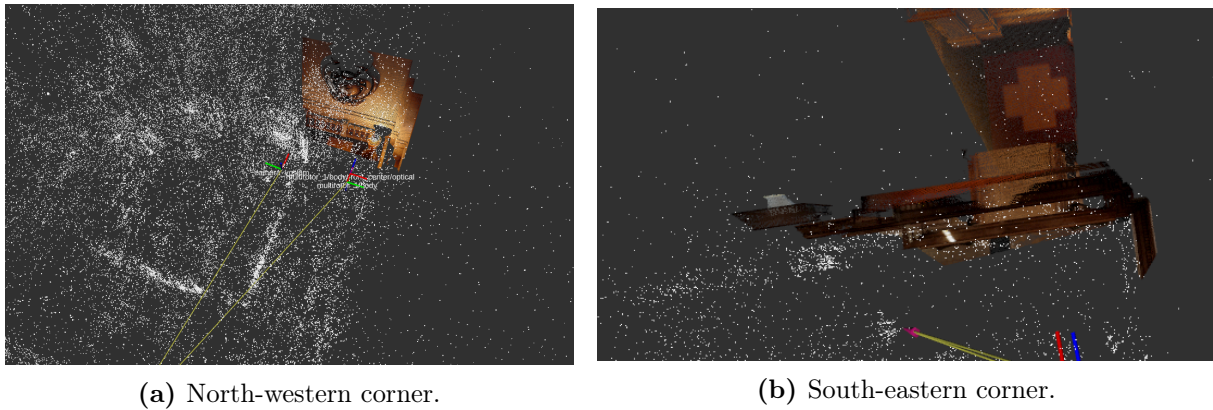


Figure 6.15: Corner wall placement comparison on opposite sides of the room.

SVO

SVO had a lot more trouble with feature tracking in this environment, where the hardest part was getting a proper initialization. Using the standard setup for the FAST feature tracker, it was not able to track anything aside from the initial points. While decreasing the search grid size helped a bit, applying the accurate preset setting provided the best results. The FoV was also increased to 110° , with corrected intrinsic parameters, and setting the image to greyscale. The fps could, however, not be increased above ten fps due to the limitations in the AirSim API. These changes allowed for decent tracking in some areas.

In the indoor parts of the temple environment, the number of features it was able to track was constantly around the minimum threshold of 100 features, which usually meant that it went below the threshold as soon as the multirotor moved. Decent tracking was achieved in the pavilion shown in Figure 6.8a, where the light conditions were better. This produced the track shown in Figure 6.16 and 6.17.

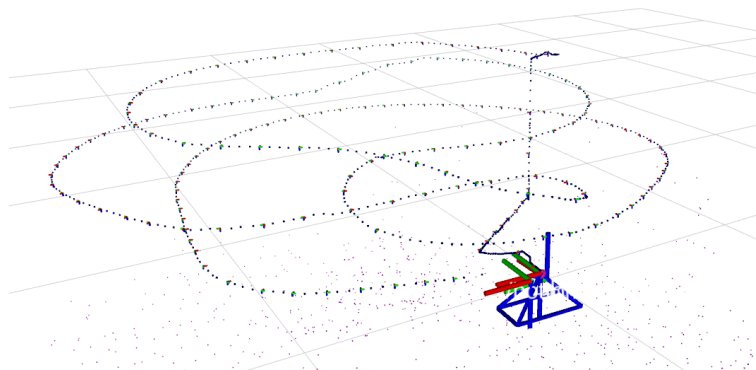


Figure 6.16: SVO pose estimate close to the origin.

Looking at Figure 6.16 it can be seen that there is very little drift in the pose estimation, even after significant movement, and in Figure 6.17b we see that the estimated direction is correct, even though the absolute scale is off. As mentioned earlier, it is not possible to compute the actual scale with monocular SLAM, which means that this result is quite good even at the reduced framerate.

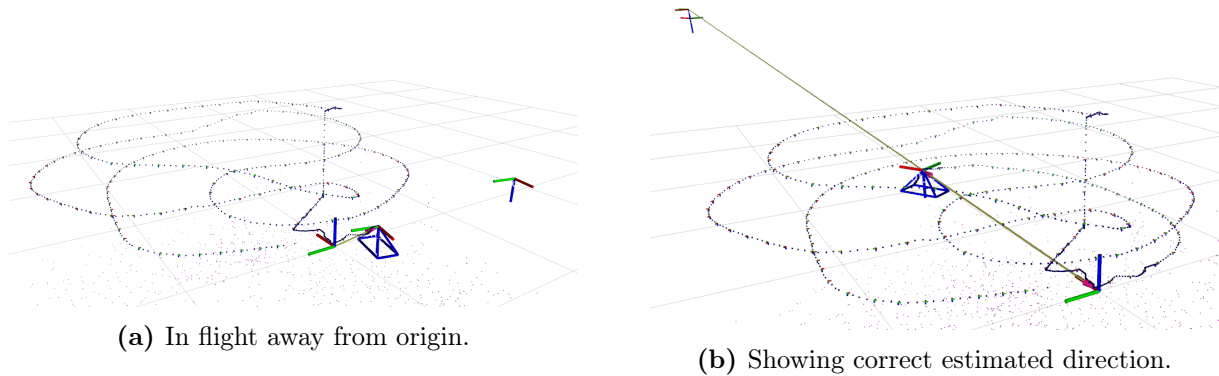
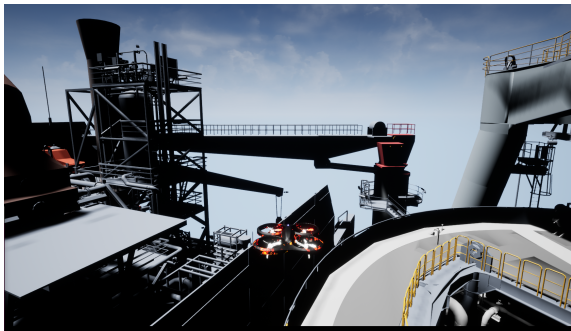


Figure 6.17: SVO pose estimate away from the origin.

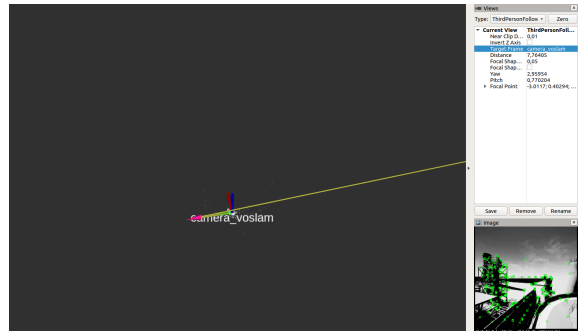
6.3.2 Skarv FPSO Simulations

There is one significant difference between the temple environment and the Skarv environment: The temple environment is made in Unreal Engine and optimized in terms of details and lighting calculations, while the Skarv model is not. It is directly imported into the scene from a production-ready model. This turned out to be too computationally intensive for both of the computers available for testing. The number of calculations needed to render the scene slowed down the simulation drastically, causing severe stuttering and even making the flight controller unstable at times. This made SVO fail the initialization step altogether, as the image frame rates were not consistent enough. For this reason, all attempts of SVO simulation on the Skarv FPSO model was discarded.

ORB-SLAM2 was actually able to find features and initialize, as seen in Figure 6.18. However, the only way to keep track of the features was to move at very incremental steps. In many cases, this led to the map points provided by ORB-SLAM to stacking on top of each other.



(a) In flight view.



(b) RViz view.

Figure 6.18: Attempt at running ORB-SLAM2 in the Skarv FPSO environment.

7 | Discussion

As presented in Chapter 1 and 3 there are a lot of existing middleware, which incorporates modularization techniques and messaging protocols across modules, all the way from small embedded applications to large scale distributed systems across multiple machines. However, one small gap exists between the network-based middleware solutions, aimed for general-purpose operating systems, and the embedded middleware. Namely a way to connect the different interfaces found in the various simulation setups, in a way that promotes modularization and code reuse, while still being compiled into a single application, in order to enable compile-time optimizations.

7.1 Core Client design

The core client was made to be minimalistic by design, in itself only consisting of about a thousand lines of code, trying to make it as small as possible, while making the possibilities of expansion as simple as possible. The implementation provides minimal functionality at its core, enabling the application to process what is needed for the specific use and little else. For this reason, all the client does during the main parts of the runtime is: Keeping track of the time between updates, deciding when a node is allowed to run its update loop, distributing current events and updating the GLFW window for the keyboard interface.

These operations alone provide extremely little overhead, as seen in the benchmarks for the simple message passing, Figure 6.1, where both the message overhead itself, and the scaling of additional nodes is significantly better than the ROS equivalents. The fact that the difference in processing time towards ROS is so large means that there is virtually no implications in terms of latency when using the client together with ROS.

The strength of the client comes in the form of how the events and nodes work together. Since the event system is the only way the nodes can pass information between each other, the events can be created to have semantic meaning such as "image" and "transform." This abstraction allows for modularization, where all nodes that know how to handle images can handle them, regardless of which node sent it.

The downside of the event system does not protect the programmer in any way against creating slow implementations. Since the user of the client is free to design the nodes and events however they want, they may also design events inefficiently. The effect of this

is shown quite well in Figure 6.2b, where the performance of managing the vector as a shared resource, is about four times faster on average than creating and sending copies. This drawback is, unfortunately, something that comes with using the C++ language, where it trades high performance with an increased amount of pitfalls.

Something ROS messages do very neatly is that message files define structures outside of the code. The compiled message code is then shared to the nodes that need their definition. This functionality is something the core client should have surrounding the event system. At this point, to create readable events, it is easiest to define a struct and include a header in each node that needs to know about it. While this is possible to define in the event definition file, it would be most readable to do this behind the scene.

The nodes provide a clear interface for the programmer, with a set number of descriptive functions that need to implement, reflecting the phases in operation: construction/destruction, startup/shutdown, during an update and on an event. This setup forces the programmer to split the code up in smaller chunks of more readable code, instead of long, convoluted initialization scripts ending in a while loop. A lacking feature in the core client is, however, synchronization between nodes, in between startup and the first update loop. The implications of this will be discussed further in Section 7.2.

The explicit shutdown mechanic creates a way to tell external APIs that the client will no longer be available and therefore shut down cleanly. Note that there is no inherent exception safety, meaning that unhandled exceptions thrown by the nodes or their connected APIs will terminate the application. As the client should not assume that every exception is a non-breaking exception, this is intentional.

C++ is also one of the more portable languages, as there exist compilers for C++ available in most operating systems, and even for a wide range of microcontrollers. As of now, one should note that the client relies on GLFW for input handling, as well as spdlog for logging. GLFW is locked to Windows, macOS, and Linux, while spdlog also supports Android and a couple more systems. However, these systems could be easily disabled to port the client to other platforms.

An improvement to implement is a way of making GLFW windows behave more like nodes, and change the input system accordingly. This way, windows could be added if needed, reducing the overhead of the client further.

When it comes to the flexibility of the client, it is less flexible than many other types of middleware. The decision to make it a single C++ application means that all connected interfaces need a C or C++ API. The constraint of compiling a single binary also makes the possibility of inherently supporting multiple programming languages a lot harder. However, since interfacing towards ROS is easily done via the node system, there are options to extend the flexibility. This approach where the middleware can work together, rather than compete is a step in the right direction.

The main drawback of the client is its single-threaded, synchronous client design. Even though asynchronous function calls are possible, there is no way of protecting the update loop from slow, blocking, function calls. While this makes it easy to reason about the

execution order of both updates and events, it also means that a node that uses a long time to update will block all other nodes from doing their task. This delay can be detrimental to real-time applications, such as control loops. Future additions to the client should, therefore, include a scheduler, which can set some execution on hold to allow execution of time-critical code.

7.1.1 Performance

The performance goal of the core client is to impact the latency of the application in a way that is of magnitudes lower than other types of middleware, so that they may be used together with the client without applying any additional latency. Looking at the resulting benchmarks in Figure 6.1 and 6.2, it can be seen that the results satisfy this criterion, at least when it comes to ROS. According to this thesis [23], ROS also has a small overhead compared to many other types of robotics middleware, which means that comparing the client to other types of middleware would give similar results. This statement does, however, not include embedded types of middleware, which are highly optimized for the platforms they support.

The graphs plotted in Section 6.1 are based around the processing time scaling based on the number of nodes. While scaling is an important factor, one should note that creating multiple nodes, all subscribing to topics containing large data messages, is not realistic in terms of applications for visual navigation. Large data structures like the ones benchmarked here are in most cases images, which should only be processed for VO/SLAM or AI purposes. In other words, very few nodes will listen to this kind of data messages, which in turn makes the results for 2 – 4 nodes most relevant.

The ROS nodelets scale considerably better than the ROS nodes, which most likely is because the underlying message used for ROS nodelets only contain pointers to a shared data resource, while the ROS nodes copy the message contents to each node. The reason why the copy version of the large data event example in Figure 6.2b scales just as well as the pointer-based approach, is because only event transfer only performs one extra copy. While all the nodes receive the event, the event system itself is created so that only one single instance of the event itself exists, and the nodes get a reference to this event instance.

In the client timings for the simple message, there was one outlier in Figure 6.1b, where the response time for 50 nodes was slower than expected when compared to the other results. This result also shows in the distribution in Figure 6.3a, where a significant amount of outliers lie way above the outliers for the other tests. These results were consistent between all five timed runs, and the outlier samples were distributed sporadically in the datasets. However, since the tests were run in quick succession, the most likely cause is that the operating system was doing other work while running the tests. Since these tests take such a small amount of time, any processes which delay the execution in the scale of microseconds could have a significant effect on the timing.

Outliers are not uncommon in any of the plots. In fact all plots in Figure 6.3 , 6.4 and 6.5

show a significant amount of outliers. These outliers are most likely caused by the operating system's background tasks influencing the measurements. As eluded to in the previous paragraph, many OS tasks are running in the background affecting the measurements. In addition to this, memory access time differences between registers cache and RAM may also show in these kinds of tests.

This fact does not mean that the outliers are discardable. While these numbers are low enough for the simple messages to affect the overall operation, larger spikes could be detrimental to control applications. In the case of large data transfers, seen in Figure 6.4a and 6.5a, as there are samples which reach the $2 - 4ms$ range. One should note that this is the case for about $0 - 2$ samples per 5.000.000 for the shared-pointer approach, and $0 - 2$ samples per 2.000.000 in the case of the vector copy approach. In other words, it happens extremely rarely. More extensive benchmarks, accounting for varying amounts of computation, should, however, be done if the system is extended to include real-time guarantees.

It can also be seen that the box plots vary way more in Figure 6.4b than in Figure 6.5b. This result is likely caused by the fact that each test was run only two times, as opposed to five, which could mean that these benchmarks ran too few tests, for the large data case.

All in all, the benchmarks for the core client turned out satisfactory, showing high performance compared to the ROS equivalent. The box plot distributions also showed that 75% of the samples stayed within $100ns$ of each other, except for the case of 128 nodes, which is not a practical use case of the client. For the large data timings, 75% of the samples stayed within $60\mu s$ of each other. This results in a smaller relative variance than for the smaller size data, which is good.

The fact that the barebones implementation is so fast also shows potential in terms of embedded systems. There is at least potential for compiling it to work on a Raspberry Pi or similar systems, where low-level external interfaces are available. This possibility could create interesting applications to use the client for HIL purposes. Whether this is possible would require further testing, and there is also a possibility that existing embedded middleware is more suited to this kind of application.

7.2 Airsim Client Design

The AirSim Client was designed to show how one can incorporate the drone simulation of AirSim, in Unreal Engine, together with ROS based VO/SLAM algorithms, using the client framework. This operation was achieved by designing a ROS node and two AirSim nodes, and a common set of events for transferring images, transforms and camera info between the nodes. The two AirSim nodes were responsible for each of the simulation modes of AirSim: Multirotor mode and Computer vision mode, and which one to run was decided by reading the AirSim settings file during initialization.

In terms of the initial design, it was easy to split the different roles. The ROS node, would handle all communication, publishing and subscribing to topics, and keeping the info needed for ROS local. The same was true for AirSim, where it could keep track of the RPC connection to AirSim, and the information needed to access the cameras and vehicle. This way the ROS node never kept any information about the state of AirSim, and the other way around.

However, one problem arose when the setup in the AirSim node needed to change. In the case of ORB-SLAM2, it has two modes of monocular SLAM. One requires a color image, while the other also needs a depth image. Moreover, in the case of SVO, the setup needs a downward pointing camera producing greyscale images.

For the AirSim node, the change was simple. After adding a downward pointing camera to the AirSim simulation and API, this could be set active in the AirSim settings file, discussed in Section 5.1.2. This setting meant that the AirSim node could be made general, by merely parsing the settings in this file, to find the current setup. This change solved the problem without adding any new dependencies.

In the ROS node, however, the published topics were set initially to be static. This could, however, no longer be the case, as the number of publishers needed to change to fit the setup of ORB-SLAM and SVO. The way this was solved was through parsing the same settings file, and setting up a variable amount of publishers based on this. Looking at this in hindsight, it is a violation of the whole principle behind the client structure, as it added a dependency between the ROS node and the AirSim node. It also shows that the client lacks post initialization possibilities.

One could argue that one could pass these as ROS parameters in a launch file. However, this would make the user need to keep the ROS parameters up to date with the change in the AirSim settings file. While this would have solved the problem, a cleaner solution would be to add an initialization event to the ROS node, which should hold the topics to publish to, as well as the message types. However, due to the lack of post-initialization possibilities, there is currently no good time to create this event. During startup, the only available events are the client events, and in the update loop, one would have to create extra logic to ensure a single event instance, and that the ROS node handles it before anything is to be published.

A feature for which should for the future would, therefore, be a post initialization step, and another type of events made for initialization between nodes. The client should run this step after all the nodes have run the *OnStartup()* function, and before the first update. This functionality would then create another layer to the node design, enabling dynamic configuration based on which nodes are active. While it would add a small dependency between the nodes, it would remove the need for the AirSim nodes to include ROS header files or the other way around.

7.2.1 Performance

In regards to performance, the single-threaded approach of the client showed negatively in a couple of cases. As seen in Table 6.3, the image fetching from AirSim takes a considerable amount of time, with a median of around $80ms$. Since the update loops of the nodes are run in succession, synchronously, the image fetching would hold the operation of the ROS node, as well as the rest of the AirSim update execution. Because of this, the multirotor controls became less responsive, and the transforms were updated and published at a slower rate than they should. This problem was partially solved by setting using the delta-time, or time since the last update, to only fetch an image every $100ms$. However, it did not solve the problem with the blocking function call.

While the image fetching is way too slow in general, this should not affect the rest of the system in any significant way, and could be solved through asynchronous function calls, where the fetching of images is run concurrently with the rest of the program, to allow multiple updates to be run for each image fetched, and also allow updates while fetching the image.

The reason behind the slow function call to fetch images was tracked down to be a specific part of Unreal Engine, which reads the pixel values from the texture in the camera object's render target[40]. After trying Unreal Engine version 4.18, 4.20, 4.21, and 4.22, this issue is still there. Changing the Unreal Engine source code would also complicate the usage for others at a later time.

Another less obvious flaw of the single-threaded execution shows in the event handling functions. Since the client gives the events to the nodes sequentially, their event handling function can also halt the execution, as it interrupts the update loop. Here, one can argue that the events should only be used to pass quick messages and data and that the processing should happen during the update loop. However, multi-threading also solves this problem.

7.3 Simulations

After solving the problems surrounding image publishing, adding point cloud generation from the ground truth depth images and pose was quickly done through the depth image processing package available to ROS. However, even though the simulation provides ground truth data, there is still some error in the generated point cloud, as seen in Figure 6.10b. The cause of this is a mismatch in time. The multirotor pose update and the image fetch, unfortunately, happens at different times. Since getting the image takes a long time, the multirotor can move a significant distance, causing the difference.

In hindsight, this could have been solved in the client through timestamp matching. By buffering the pose data instead of just storing the current one, and then matching the timestamps, one would guarantee that the depth image and the camera info message for the same ROS timestamp would contain the related pose and depth image.

Another solution would be to create a map generation node, which creates the point cloud, and ensures that it uses the correct pose. This addition would likely be a better solution overall, as one could control the density of the created map, to tune for performance. Adding a mapping node would, however, require the client to be adopted to a multithreaded asynchronous approach, as point cloud generation is a demanding task.

For this simulation, it does provide a good enough estimate as it is just used to visually compare it to the map generated by ORB-SLAM2, and not used in calculations as ground truth data. The reason this was disabled in the SVO simulations, is that the depth camera was disabled in order to increase performance. Since the SVO algorithm produces a trajectory and not a map, it is also less applicable for the comparison.

7.3.1 Skarv FPSO simulations

The aspect of importing 3D models into Unreal Engine and then use it for image data generation is interesting. Unfortunately, this did not turn out so well in these simulations. Since the FPSO model was so detailed, the simulations could not run smoothly on the available computers. This performance issue caused both the flight controller to become unstable. It also made it impossible for either ORB-SLAM2 or SVO to keep track of the features in the images collected. This issue does not rule out the possibility of importing large-scale models for simulation. However, one has to do significant work to the model, and the simulation to make it run smoothly.

One optimization to apply would be to reduce the amount of detail in the model. Looking at Figure 6.9b, many details would be unnecessary to capture in terms of visual navigation. However, some details are needed in order to keep the realism of the simulation. Another technique that could be applied would be to reduce the number of triangles composing the model. One can do this without reducing the visual quality while significantly reducing the amount of calculation needed to compute the lighting. As the model itself is real size, applying fog some distance away from the camera, or reduce the view distance all together may also help. It is, however, uncertain how this would affect the depth camera.

7.3.2 Unreal Temple Simulation

As this scene is more optimized for real-time rendering, the simulations went smoothly. In terms of scene variety, the temple environment was complex enough to provide a varied test environment, containing both areas of varying light conditions as well a fair share of varied textures. The realistic-looking design also made it pass as a location that could exist in the real world.

ORB-SLAM2

The ORB-SLAM simulations were done in two turns, once with a single RGB camera, and once with an RGB-D camera, utilizing the new additions to the ORB-SLAM2 algorithm.

For the monocular SLAM example, the result shown in Figure 6.12 show a mapping which has a relative size that is very close to the ground truth map shown in Figure 6.11. however, some areas are of particular note:

The first to note is the top circular area of the estimated map. Here we see a much lesser dense point cloud. This effect is caused by the large arcs placed here, as seen in Figure 6.8a. For this reason, the resulting map is accurate in this regard.

Another thing to note is the adjacent corridor just below the main room. In regards to the rest of the map, this area appears skewed. The loop closure of SLAM algorithms typically corrects this problem. However, in this area, ORB-SLAM has not been able to do this. The reason is most likely the poor lighting in the area, as well as the monotone textures. In this corridor, there are no windows nearby, so the room is therefore only lit by two torches, as well as indirect lights from the main room. In addition to this, the corridor consists of brick walls with little to no additional details. This uniformity caused problems for the feature tracking, and it was not uncommon that the algorithm had to relocate after flying through this corridor. Since ORB-SLAM generally performed well in other areas, both in terms of distance to features and rotation of features, it may also have been a combination of poor lighting and bad framerate.

Comparing the estimated map position in Figure 6.13, we see that it estimates its pose in the map correctly, relative to the position of the drone seen in the picture to the right. An improvement the test setup would be to scale the estimated map point cloud to the size of the actual map and feed it back to ORB-SLAM, with the correct rotation, for a better localization test.

Adding depth images to ORB-SLAM2 improved both the rotation and the scale of the map, as can be seen in Figure 6.14. Note here how the angle of the point cloud causes the map placement to look worse than it is. As can be seen in Figure 6.15b, looking at the image from above actually shows a very accurate corner placement.

One interesting observation in the RGBD tests, which did not happen in the RGB test was the creation of duplicate maps with a smaller scale within the map. This can be seen in both Figure 6.14 and 6.15a. This would appear quite frequently, across multiple runs, and most often after having to relocate. Since the map is generated from the ground truth depth image, one would think that this should not happen.

The ORB-SLAM2 paper[5] does state that the RBGD approach creates a simulated stereo camera setup, by treating the RGB picture as the left picture, and projecting the features found there to a right image, using the depth information. Further, it states that they split the uncertainties of the depth estimates based on the baseline. Where the split between close and far features is 40 times the baseline. The close features are assumed accurate and used for triangulation, while the far features are mainly used to correct for

rotation.

However, in the simulation setup, the camera capturing the depth and RGB image has the same transform, which ultimately would cause the baseline to be zero. This would mean that all depth estimates would be classified as far away, and therefore mainly used to correct for rotation, and only used for triangulation if supported by multiple views. This problem, in turn, removes a huge benefit of the RGBD approach and should be noted by others who try to use similar setups.

SVO

For the SVO simulations, there was, unfortunately, a problem with the feature tracking. This problem caused the simulation to be limited to run in the pavilion area shown in Figure 6.8a, where the scene was well lit. The low light conditions are, however, not believed to be the main cause of the problems. On the creator's wiki pages[39] they state that the framerates should be set as high as 70 frames per second. In these simulations, only ten fps was achievable in order to assure a stable frame rate. Only having a seventh of the recommended framerate can have significant implications for the tracking, as the rotation of the multicopter can change the camera view significantly in $100ms$.

In order to counteract the low framerate, the accurate feature tracking settings described in the paper[6] was be applied. The problem, however, was first solved by increasing the FoV to 110° in both directions. The last change was enough to track enough features to get a suitable run. Flying at a higher altitude also seemed to help, which is to be expected, as features will stay in the picture for a longer time.

Even though the tests were limited, one can extract some information. One interesting observation is that SVO is extremely good at estimating the pose, with very little drift. This observation can be seen in Figure 6.16, where the estimated position is accurate near the origin, even after significant flying. This observation is in line with their results, where they show that the drift in estimations, increases very little with time.

8 | Conclusion and Further Work

8.1 Conclusion

The client is still in early development, and it therefore naturally lacks in features. However, in its core design it is not supposed to provide much flexibility, like other middleware, but rather force the developer into creating modular code to use across changing simulation setups, as well as being easily integrated with other middleware that provides additional features, if those are needed.

Regarding integration, the core design works quite well. The node structure should be integrable with all C++ interfaces, as long as they do not have operating system dependencies. Looking at the setup with AirSim and ROS, both interfaces integrated without much problem. However, the event system needs to be powerful enough to force the programmer to not work around it. At this point, the easiest way to use the event system for data messages is through adding common dependencies between nodes. This created dependency does remove some of the points of using this client in the first place.

That said, the code written for the master's thesis is much cleaner than that of the project thesis, and the AirSim node itself is independent enough to be combined with for example an ORB-SLAM node, without any significant changes to it.

The client does also have significant drawbacks to address in order to be viable for visual navigation, or other real-time critical tasks. As seen with the image fetching, the update loop for the control inputs was blocked while waiting for the image to be received. This delay made the multirotor controls unresponsive, which would not have been viable at all if the flight controller itself had been implemented as a node.

Another feature that was lacking was the ability to do initialization between nodes. In the case of the AirSim client, one should have been able to initialize the ROS publishers based on the AirSim setup, without needing to rely on the parsed AirSim settings file in the ROS node. For this to be possible, another initialization step is needed. This step must come after node initialization towards external APIs, and before the first update loop.

Even though the setup was not optimal, the AirSim client was shown to work with ORB-SLAM, and to some extent SVO, managing to see the features these algorithms supply.

The main issue was the reduced image update rate caused by the slow RPC call to AirSim and Unreal Engine. While this does show that ORB-SLAM is more tolerant towards low framerates than SVO, it did not give SVO a fair chance. Using real cameras, with this resolution, one could have benefited from a substantially higher framerate.

In the case of ORB-SLAM, the simulation produced an interesting result: Based on how it handles depth data for monocular RGBD SLAM, it depends on the fact that there is a distance between the depth camera and RGB camera. This effect is something to note for artificial camera setups in simulation environments, where one can place cameras on top of each other.

All in all, one can conclude that the client shows potential, in terms of usability in simulation setups for visual navigation. However, it is not usable in its current state. Where the most significant problem to address is the synchronous and blocking behavior of the update loop. This problem must be handled before anyone can use the client in a real-time system.

In addition to this, if this is possible to fix the slow image fetching from Unreal Engine, there are great possibilities when it comes to using it for real-time visual navigation simulations.

8.2 Further Work

There are various areas where the client to improve, without adding any latency inducing overhead. An important note to the following section is that none of the features here should be added to the core unless they add performance benefits, or add much-needed features to visual navigation simulations. Most additional features should also be on an opt-out or opt-in basis, where they may be disabled or added to the compilation as a choice for the developer.

The first, easily implementable addition should be to add a post initialization step between the startup routine and the first update loop. This addition would be beneficial in order to support internode dependencies and dynamic setups, without adding unnecessary dependencies between them.

Another feature that is sorely needed is multithreading and scheduling. Modern CPUs have multiple cores available to use for processing. Utilizing these would be vital for the performance-critical task visual navigation is. Providing a scheduler would also negate the problem with one update loop blocking the others, which is a must for real-time critical systems.

One feature which may improve performance and portability would be to redo the keyboard input and window system to behave more like nodes, where the developer can add them to the client id needed. While converting the window system is simple, the keyboard input system would need some more work.

In addition to this, some restructuring of the event system could be beneficial. One exciting approach to modular event systems is through the usage of `std::variant`¹ instead of a polymorphism based event system. This change could reduce the amount of virtual function calls needed, and therefore improve performance. One must, however, benchmark this to be sure. In addition to this, it is uncertain how to handle expansions of the system for new events.

Another way to enforce performant events may be to provide base event types for different data sizes, to expand as needed, but where the underlying data structure is optimized for transport between nodes. This optimization is also integrable with a ROS message-like system and might be the correct way to move forward.

Additional wanted features include:

- A customizable debugging user interface.
- Provide a base package of events for developers to use.
- More example setups.
- Integration with other simulation environments and middleware.

One exciting aspect worth exploring would be compiling towards a Raspberry Pi as this is a relatively general-purpose UNIX-based computer, while still having many low-level interfaces. If it would be reasonable to integrate the client, this could open up a whole new range of applications, especially in the form of hardware in the loop simulations.

¹<https://en.cppreference.com/w/cpp/utility/variant>

Bibliography

- [1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: An open-source robot operating system”, in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.
- [2] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”, *CoRR*, vol. abs/1705.05065, 2017, Github repository: github.com/Microsoft/AirSim. arXiv: 1705.05065. [Online]. Available: <http://arxiv.org/abs/1705.05065>.
- [3] (2004). Unreal Engine 4 Game Engine, [Online]. Available: <https://www.unrealengine.com>.
- [4] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, “Orb-slam: A versatile and accurate monocular slam system”, *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.
- [5] R. Mur-Artal and J. D. Tardós, “Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras”, *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, Oct. 2017, ISSN: 1552-3098. DOI: 10.1109/TR0.2017.2705103.
- [6] C. Forster, M. Pizzoli, and D. Scaramuzza, “Svo: Fast semi-direct monocular visual odometry”, in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 15–22. DOI: 10.1109/ICRA.2014.6906584.
- [7] (May 16, 2018). Rviz ros visualization library package roswiki page, [Online]. Available: <http://wiki.ros.org/rviz>.
- [8] S. A. Haugane, “360-degree camera simulator for realistic imaging using unreal engine”, Project thesis, Norwegian University of Science and Technology, NTNU, 2019, Unpublished.
- [9] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield, “Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness”, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 4241–4247.
- [10] J. Engel, V. Koltun, and D. Cremers, “Direct sparse odometry”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 3, pp. 611–625, Mar. 2018, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2017.2658577.
- [11] M. Mueller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem, “Ue4sim: A photo-realistic simulator for computer vision applications”, *CoRR*, vol. abs/1708.05869, 2017. arXiv: 1708.05869. [Online]. Available: <http://arxiv.org/abs/1708.05869>.

-
- [12] W. Meng, Y. Hu, J. Lin, F. Lin, and R. Teo, “Ros+ unity: An efficient high-fidelity 3d multi-uav navigation and control simulator in gps-denied environments”, in *Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE*, IEEE, 2015, pp. 002 562–002 567.
- [13] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “Carla: An open urban driving simulator”, *arXiv preprint arXiv:1711.03938*, 2017.
- [14] C. Dyken. (Jun. 21, 2019). Adventures in the machine-learning land of drones and lidars, part 1, [Online]. Available: <https://medium.com/kongsberg-digital/adventures-in-the-machine-learning-land-of-drones-lidars-2d087df6d689>.
- [15] O. Gietelink, J. Ploeg, B. D. Schutter, and M. Verhaegen, “Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations”, *Vehicle System Dynamics*, vol. 44, no. 7, pp. 569–590, 2006. DOI: 10.1080/00423110600563338. eprint: <https://doi.org/10.1080/00423110600563338>. [Online]. Available: <https://doi.org/10.1080/00423110600563338>.
- [16] C. Brogle, C. Zhang, K. L. Lim, and T. Bräunl, “Hardware-in-the-loop autonomous driving simulation without real-time constraints”, *IEEE Transactions on Intelligent Vehicles*, pp. 1–1, 2019, ISSN: 2379-8904. DOI: 10.1109/TIV.2019.2919457.
- [17] L. Verhoeff, D. J. Verburg, H. A. Lupker, and L. J. J. Kusters, “Vehil: A full-scale test methodology for intelligent transport systems, vehicles and subsystems”, in *Proceedings of the IEEE Intelligent Vehicles Symposium 2000 (Cat. No.00TH8511)*, Oct. 2000, pp. 369–375. DOI: 10.1109/IVS.2000.898371.
- [18] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, “Middleware for robotics: A survey.”, in *RAM*, 2008, pp. 736–742.
- [19] A. Elkady and T. Sobh, “Robotics middleware: A comprehensive literature survey and attribute-based bibliography”, *Journal of Robotics*, vol. 2012, 2012.
- [20] J. Jackson, “Microsoft robotics studio: A technical introduction”, *IEEE Robotics Automation Magazine*, vol. 14, no. 4, pp. 82–87, Dec. 2007, ISSN: 1070-9932. DOI: 10.1109/M-RA.2007.905745.
- [21] K. Johns and T. Taylor, *Professional microsoft robotics developer studio*. John Wiley & Sons, 2009.
- [22] (2012). Common object request broker architecture specifications, [Online]. Available: <https://www.omg.org/spec/CORBA/>.
- [23] S.-G. Chitic, “Middleware and programming models for multi-robot systems”, Theses, Université de Lyon, Mar. 2018. [Online]. Available: <https://hal.archives-ouvertes.fr/tel-01809505>.
- [24] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, “Context-aware middleware for resource management in the wireless internet”, *IEEE Transactions on Software Engineering*, vol. 29, no. 12, pp. 1086–1099, Dec. 2003, ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1265523.
- [25] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar, “Miro-middleware for mobile robot applications”, *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 493–497, 2002.

-
- [26] S. Enderle, H. Utz, S. Sablatnög, S. Simon, G. Kraetzschmar, and G. Palm, “Miro: Middleware for autonomous mobile robots”, *IFAC Proceedings Volumes*, vol. 34, no. 9, pp. 297–302, 2001.
- [27] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, “Robotic software integration using marie”, *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 10, 2006. DOI: 10.5772/5758. eprint: <https://doi.org/10.5772/5758>. [Online]. Available: <https://doi.org/10.5772/5758>.
- [28] A. Makarenko and A. Brooks, “Orca: Components for robotics”, in *In 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS’06)*, 2006.
- [29] B. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems”, in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.
- [30] D. Blank, D. Kumar, L. Meeden, and H. Yanco, “Pyro: A python-based versatile programming environment for teaching robotics”, *J. Educ. Resour. Comput.*, vol. 3, no. 4, Dec. 2003, ISSN: 1531-4278. DOI: 10.1145/1047568.1047569. [Online]. Available: <http://doi.acm.org/10.1145/1047568.1047569>.
- [31] D. Blank, L. Meeden, and D. Kumar, “Python robotics: An environment for exploring robotics beyond legos”, in *SIGCSE ’03 Proceedings of the 34th SIGCSE technical symposium on Computer science education*, vol. 35, 2003. DOI: 10.1145/792548.611996.
- [32] (2014). Gazebo simulator, [Online]. Available: <http://gazebo.org>.
- [33] S. Magnenat, V. Longchamp, and F. Mondada, “Aseba, an event-based middleware for distributed robot control”, *Workshops and Tutorials CD IEEE/RSJ 2007 International Conference on Intelligent Robots and Systems*, 2007. [Online]. Available: <http://infoscience.epfl.ch/record/111860>.
- [34] L. Meier, D. Honegger, and M. Pollefeys, “Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms”, in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 6235–6240. DOI: 10.1109/ICRA.2015.7140074.
- [35] (2009). Mavlink communication protocol home page. Part of the Dronecode Project, [Online]. Available: <https://mavlink.io/en/>.
- [36] (2014). Spdlog github repository, [Online]. Available: <https://github.com/gabime/spdlog>.
- [37] S. Shah and C. Lovett. (Nov. 29, 2018). Airsim settings documentation, [Online]. Available: <https://github.com/Microsoft/AirSim/blob/master/docs/settings.md> (visited on 08/05/2019).
- [38] (2017). Skarv fpso article at aker bp’s homepage, [Online]. Available: <https://www.akerbp.com/produksjon/skarv/>.
- [39] (2014). Obtaining best performance. svo github wiki pages, [Online]. Available: https://github.com/uzh-rpg/rpg_svo/wiki/Obtaining-Best-Performance.
- [40] (Jul. 10, 2017). Airsim github repository issue#329, [Online]. Available: <https://github.com/Microsoft/AirSim/issues/329>.
-

