



DEPARTMENT OF
COMPUTER SCIENCE

LUCAS CORREIA LOBO

B.Sc. in Computer Science and Engineering

WATER SIMULATION AND OPTIMIZATION FOR BIOMIMETIC PROPULSION IN UNDERWATER VEHICLES USING HLSL

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2025

WATER SIMULATION AND OPTIMIZATION FOR BIOMIMETIC PROPULSION IN UNDERWATER VEHICLES USING HLSL

LUCAS CORREIA LOBO

B.Sc. in Computer Science and Engineering

Adviser: Rui Pedro da Silva Nóbrega

Assistant Professor, NOVA School of Science and Technology

Co-adviser: Victor José de Almeida e Sousa Lobo

Full Professor, Portuguese Naval Academy and NOVA Information Management School

Examination Committee

Chair: Susana Nascimento

Assistant Professor, NOVA School of Science and Technology

Rapporteur: Rui Madeira

Assistant Professor, Polytechnic Institute of Setúbal

Adviser: Rui Pedro da Silva Nóbrega

Assistant Professor, NOVA School of Science and Technology

Water simulation and optimization for biomimetic propulsion in underwater vehicles using HLSL

Copyright © Lucas Correia Lobo, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Professor Rui Nóbrega, whose guidance, time and encouragement were invaluable throughout the development of this thesis. I am also grateful to Professor Victor Lobo, who first suggested the theme for the thesis and kept me focused on the final goal.

This work was carried out as part of the SABUVIS project, and I thank the entire team for their collaboration and shared vision. I also acknowledge the support of Portuguese Navy, whose resources made this research possible.

Special thanks to my family and friends, thank you for your unwavering support and patience during this journey.

ABSTRACT

Despite the large importance of the oceans and other bodies of water to our life, they remain largely unexplored. Current solutions to explore these environments include Biomimetic Underwater Vehicles (BUVs), unmanned vehicles inspired by biological life, usually in their locomotion, which can be used to navigate otherwise complicated terrain, and can be stealthier than screw-like propellers in underwater vehicles, among other benefits. Despite these benefits their development is still hampered by one common factor, the control mechanism. The European SABUVIS project, in which this thesis is included, aims to tackle this problem by training a control mechanism through a reinforcement learning algorithm.

This thesis presents the development and evaluation of a real-time fluid simulation tool designed to support the SABUVIS project. Built using the Unity3D game engine and accelerated through High Level Shader Language (HLSL), the simulation models fluid dynamics using an Eulerian (grid-based) approach. Key features include customizable resolution, multiple rendering modes, and post-processing effects to enhance visualization and user interaction.

The simulation was tested for correctness and performance under varying conditions. Results confirmed that the system accurately replicates expected fluid behavior, including interactions with static obstacles, and performs efficiently enough to support future integration with reinforcement learning algorithms. Performance benchmarks evaluated the impact of resolution, overrelaxation, and visualization features on execution time.

Keywords: Fluid simulation, Biomimetic Underwater Vehicle (BUV), High Level Shader Language (HLSL), Game-engine, GPU acceleration

RESUMO

Apesar da grande importância dos oceanos e de outras massas de água para a nossa vida, estes permanecem, em grande parte, inexplorados. As soluções atuais para explorar estes ambientes incluem os Veículos Subaquáticos Biomiméticos (BUVs), veículos não tripulados inspirados em organismos biológicos, geralmente na sua locomoção. Estes podem ser utilizados para navegar em terrenos difíceis e são mais furtivos do que veículos com propulsão por hélice, entre outros benefícios. Apesar dessas vantagens, o seu desenvolvimento continua limitado, especialmente no que diz respeito ao controlo do movimento. O SABUVIS, projeto europeu no qual esta tese se insere, tem como objetivo abordar este problema através da utilização de aprendizagem por reforço.

Esta tese apresenta o desenvolvimento e a avaliação de uma ferramenta de simulação de fluidos em tempo real, concebida para responder às necessidades do projeto SABUVIS. Desenvolvida com recurso ao motor de jogos Unity3D e utilizando a linguagem High Level Shader Language (HLSL) para acelerar os cálculos através da GPU, a simulação modela os fluidos com uma abordagem Euleriana (baseada em grelha). A ferramenta inclui, entre outras funcionalidades, resolução personalizável e vários modos de renderização, facilitando a sua utilização por outros desenvolvedores.

A simulação foi testada quanto à sua correção e eficiência em diferentes condições. Os resultados confirmaram que o comportamento do fluido é corretamente simulado, incluindo a interação com objetos estáticos, e demonstraram uma eficiência compatível com os requisitos da aprendizagem por reforço. Os testes de desempenho avaliaram o impacto da resolução, do parâmetro de overrelaxation e dos modelos de visualização no tempo de execução.

Palavras-chave: Simulação de fluidos, Veículos Subaquáticos Biomiméticos (BUVs), High Level Shader Language (HLSL), Motor de jogos, Aceleração por GPU

CONTENTS

List of Figures	viii
Listings	x
Acronyms	xi
1 Introduction	1
1.1 Motivation	2
1.2 Research Questions	3
1.3 Objectives	4
1.4 Contributions	5
1.5 Document Structure	5
2 Related Work	6
2.1 Fluid Simulation	6
2.1.1 Navier-Stokes Equations	7
2.1.2 Implementation Viewpoints	10
2.1.3 Deep Learning for Fluid Simulation	14
2.2 Simulation of Unmanned Vehicles	17
2.2.1 Underwater vehicle simulation	19
2.2.2 Biomimetic Vehicles	20
2.2.3 Learning Control Mechanisms for BUUV	22
2.3 3D Visualization and technologies	26
2.3.1 Criteria	26
2.3.2 Platforms	27
2.3.3 Chosen Platform	28
3 Data structures and HLSL framework	29
3.1 Fluid Simulator: case study	29
3.2 Data Structure	30

3.2.1	Guiding criteria	30
3.2.2	simState	31
3.3	Unity Workflow	34
3.3.1	Event driven architecture	35
3.3.2	Regular Frame Update	35
3.3.3	Object Lifecycle	36
3.3.4	UI	36
3.3.5	Physics	36
3.3.6	Generating Pixels from the Simulation State	37
3.3.7	Visualization Types	39
3.4	HLSL based systems	42
3.4.1	What is HLSL	43
3.4.2	GPU speed up	43
3.4.3	Programming for GPUs	44
4	Fluid Simulation Algorithm	45
4.1	Main Components	45
4.2	Architecture	46
4.3	Programming style	47
4.4	Simulation Steps	48
4.4.1	Clear Pressure	48
4.4.2	Projection	48
4.4.3	Advection	49
4.5	Implementation Optimizations	51
4.5.1	Simulation state data structure	51
4.5.2	Shader partition	52
4.5.3	Where is simulation state updated	52
4.5.4	Maximum and minimum in GPU	52
4.5.5	General projection improvements	53
4.5.6	Early stop of projection	53
4.5.7	Projection set split	54
4.5.8	Obstacle encoding	54
4.5.9	Force calculation	55
4.5.10	Dispatch dimensions	55
4.5.11	Dynamic time step	56
5	Fluid Simulation Interface	57
5.1	System Requirements	57
5.1.1	Hardware Requirements	58
5.1.2	Software Requirements	58
5.2	Unity Setup/Startup	59

5.3	Running a Simulation	60
5.4	Features	61
5.4.1	State Altering Features	62
5.4.2	Visualization Features	64
5.4.3	Inspection and Analysis Features	67
6	Experimental results	69
6.1	Correctness of simulation	70
6.2	Optimizations Benchmark	72
6.2.1	Rendering type	73
6.2.2	Overrelaxation	74
6.2.3	Resolution	76
6.2.4	Early Stop	76
6.2.5	Post-effects	78
6.2.6	Number of Objects	79
6.2.7	Frame	80
6.2.8	Discussion	81
7	Conclusions	82
7.1	Future Work	83
	Bibliography	84
	Appendices	
A	Appendix	91

LIST OF FIGURES

1.1	Some examples of BUVs	2
2.1	Examples of CFD applications	7
2.2	(a) Finite control volume approach. (b) Infinitesimal fluid element approach Image taken from [30]	8
2.3	Common boundary conditions [1]	9
2.4	Custom mesh example[10]	9
2.5	Representation of MAC Cells [6]	11
2.6	Fluid flagging example on MAC grid [46]	12
2.7	Obtaining the fluid surface from particles [24]	13
2.8	Evolution over time of Eulerian approaches compared to Lagrangian [67] .	13
2.9	2D prediction of velocity field magnitude [26]. First column uses a numerical solver. Second column uses the trained CNN. Last column contains the differences between the first two	15
2.10	Image from [71], that represents the U-net architecture used. Given three input "images", containing only the airfoil, the neural network will output the predicted result for pressure, velocity in the X axis and velocity in the Y axis, each in the respective output "images".	15
2.11	Image taken from [58], presenting the results of using a PINN for fluid me- chanics. (B) is depicting the concentration, (C) is depicting the pressure and (D) is depicting the streamlines	16
2.12	Examples of unmanned vehicles	18
2.13	Simulation of UAV using different platforms	20
2.14	Some examples of BUVs	21
2.15	Application of reinforcement learning	24
2.16	Images from the compared works	25
3.1	Two different BUV prototypes, part of the SABUVIS project	30

3.2	Structure of SimulationState C# class (each of these arrays have as many components as the number of cells in the simulation space, ordered by rows from bottom to top)	32
3.3	Tow different staggered grid	33
3.4	Representaion of the indexable edges of a 5x5 grid	34
3.5	Using one GameObject per cell	37
3.6	Two uses of texture, showing a checkered pattern	38
3.7	Five different types of visualization and color guides	40
3.8	The two possible overlays with an example state	42
4.1	Main component architecture	46
4.2	Update subdivision	46
4.3	Effect of projection on the velocities around a cell	49
4.4	The four sets of cells on a 5x5 grid	49
4.5	Velocity on the different points and its backtracking	51
4.6	Different interpolation sources for different properties	51
4.7	Projection shader flowchart with early stopping	54
5.1	First look into the Unity editor	60
5.2	Start of simulation	61
5.3	Customizable variables	62
5.4	Active View Types dropdown	62
5.5	Active Click Type dropdown	62
5.6	Options available in the Unity inspector	62
5.7	Example usage of some of the features	65
5.8	Example setup of tunnel enviroment	66
6.1	Smoke flowing	70
6.2	Comparison of obtained result with other simulations	71
6.3	Steps to prove reset pressure works	71
6.4	Default scenario for tests	73
6.5	Effect of rendering type on frame execution time	74
6.6	Effect of overrelaxation value on frame execution time	75
6.7	Effect of resolution on frame execution time	76
6.8	Relation between number of cells and frame execution time	77
6.9	Effect of early stop on frame execution time	77
6.10	Effect of post-effects on frame execution time	79
6.11	Effect of post-effects on frame execution time	79
6.12	Time spent on each step of an frame	80
A.1	Event functions loop in Unity	92

LISTINGS

ACRONYMS

AUV	Autonomous Underwater Vehicle (<i>pp.</i> 1 , 2 , 19)
BUV	Biomimetic Underwater Vehicle (<i>pp.</i> 1–5 , 17 , 21 , 22 , 24 , 26 , 29 , 30 , 40 , 54 , 70 , 81)
CFD	Computational Fluid Dynamics (<i>pp.</i> 6 , 7)
CNN	Convolutional Neural Network (<i>pp.</i> 14 , 15)
EDA	European Defense Agency (<i>p.</i> 29)
GLSL	OpenGL Shading Language (<i>p.</i> 43)
HLSL	High Level Shader Language (<i>pp.</i> 3 , 29 , 42–44)
SIMD	Single Instruction, Multiple Data (<i>p.</i> 43)
SPH	Smoothed Particle Hydrodynamics (<i>p.</i> 12)
UAV	Unmanned Aerial Vehicles (<i>pp.</i> 17 , 20)
UGV	Unmanned Ground Vehicles (<i>pp.</i> 17 , 19 , 20)
UUV	Underwater Unmanned Vehicle (<i>pp.</i> 1 , 2 , 19 , 20)

INTRODUCTION

Encompassing over two-thirds of the Earth's surface, bodies of water such as oceans, rivers, and lakes present a vast, resource-rich habitat for a multitude of species. Despite this, humans, not being biologically equipped to explore these extensive environments, have historically leveraged their ingenuity to navigate these waters. This ranges from ancient civilizations utilizing boats for fishing and river travel, to the contemporary ocean economy, valued at an estimated \$2.5 trillion [50].

However, it is widely acknowledged that a significant portion of these aquatic environments remains uncharted, offering potential avenues for exploration and development. Consequently, there have been numerous endeavors to create an Underwater Unmanned Vehicle (UUV) capable of investigating these aquatic environments, reducing the risk to humans and also its operational cost.

Within the broad category of UUVs, Autonomous Underwater Vehicle (AUV) have gained prominence over other control methods due to their autonomy. This is in contrast to remotely operated vehicles, which typically need a cable connection due to the challenges associated with wireless communication underwater[53]. As a result, AUVs possess the capability to traverse more challenging and intricate areas, undertake deep-sea operations, and conduct long-range surveillance missions.

The development of AUVs started in the 1970s and has witnessed substantial growth since its inception. Today, AUVs are used in a wide array of applications, from conducting oceanographic surveys and removing mines (demining), to collecting bathymetric data (measuring the seafloor depth). However, there are still areas where the capabilities of conventional AUVs could be enhanced, such as efficiency, stealth and maneuverability, which are vital in military operations.

Biomimetics, the practice of emulating biological systems, can be applied to AUVs to solve these problems, generating the subcategory of Biomimetic Underwater Vehicle (BUV)s. This involves adopting locomotion patterns similar to marine animals, such as the Manta ray [79], the tuna [72], the turtle [64], and many others [2, 14, 68, 80], some of which can be seen in Figure 1.1. After reviewing these studies and others, Wang et al. [74] highlighted in a recent survey that there is still room for improvement in the optimization

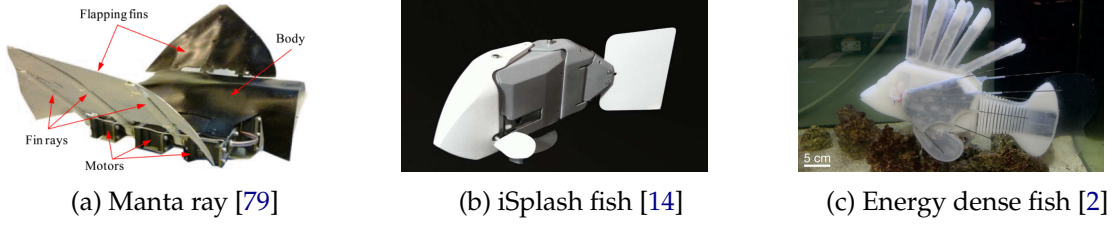


Figure 1.1: Some examples of BUVs

of propulsion mechanisms and control techniques.

The interaction between a fin of a BUV and the surrounding water is typically complex and requires a deep understanding of fluid dynamics to achieve fully optimized movement. One effective approach to studying these interactions is through fluid simulation. Possessing a high fidelity fluid simulation allows for the use a reinforcement learning algorithm to train the control mechanism of a BUV.

The ultimate goal is to develop a tool that leverages reinforcement learning to train BUV control systems. This thesis contributes to that goal by providing the theoretical background and taking the initial steps toward building such a tool, specifically through the development of a fluid simulation capable of modeling interactions between stationary objects and moving water¹. In future work, this tool can be extended to fully realize the intended objective.

The theoretical background includes topics such as reinforcement learning, which, although not implemented in the current prototype, remains valuable for future developers seeking to expand the tool. Having a complete related work for the ultimate goal also ensures any step taken is contributing to that final goal.

1.1 Motivation

The exploration of aquatic environments has been a crucial aspect of human progress, from ancient times to the current date and will likely continue to be important in the future. Despite significant advances, these environments remain largely unexplored. One such advance is the development of UUVs. Being unmanned, UUV offer significant advantages: they eliminate the risk to human life, reduce the need for large, energy-intensive designs, and lower operational costs.

However, controlling UUVs presents a challenge. There are two primary control methods: human-operated via cable or autonomous operation. The first method has major limitations due to the reliance on cable communication, which restricts maneuverability and operational range, especially in complex environments such as underwater caves. The second method, autonomous operation, is more suited for traversing difficult areas, deep-sea exploration, and long-range surveillance, leading to the development of AUVs.

¹Video with demo of tool working: <https://youtu.be/xIwYULNik8c>

Beyond control methods, propulsion mechanisms significantly impact underwater vehicle capabilities. Traditional screw-like propellers have limitations, including high noise, low energy efficiency, and poor maneuverability [74]. Recently, biomimetic propulsion, meaning propulsion inspired by biological organisms, has garnered interest. Fish achieve energy efficiencies around 70%, compared to approximately 50% for screw-like propellers [31]. Additionally, fin-based propulsion offers better maneuverability and quieter movement, making the vehicle less detectable to both marine life and human detection systems, which is crucial for underwater surveys and military operations.

Recognizing the potential of BUVs, the European Defense Agency (EDA) initiated the SABUVIS project in 2015. The project's first phase aimed to develop a BUV, and it is now in its second phase, SABUVIS II, focused on adding swarm capabilities. This ongoing project involves the Portuguese, Polish, and German Navies, together with several companies, universities and research centers, each developing their own BUVs. Currently, these prototypes use a simple sinusoidal wave with a constant frequency for fin movement, which is likely suboptimal.

Guided by the SABUVIS project this thesis aims to provide the groundwork upon which future developers can build a fully integrated environment for training control mechanisms of BUVs. Specifically, it contributes to this goal by offering an analysis of related work and presenting a prototype fluid simulation that utilizes GPU acceleration through High Level Shader Language (HLSL).

1.2 Research Questions

The main objective of this thesis is to facilitate the development of control mechanisms for BUVs. The proposed approach, uses a fluid simulation through which the interaction between objects and the water can be studied. Given this approach the primary research question is:

Is it possible to have a sufficiently realistic, physics inspired simulation of a BUV, that enables training of reinforcement learning for motion control?

To address this complex primary question, it can be divided into three secondary questions:

1. **How can we develop a physics inspired fluid simulation?**

This question involves researching various techniques for fluid simulation, such as Eulerian (grid-based) and Lagrangian (particle-based) methods, to determine which approach best suits the goals of this thesis.

2. **How can the fluid simulation be used to simulate the interaction with an object?**

This question focuses on ensuring that the implemented fluid simulation can accurately depict the forces exerted on the object by the water, which will then be applied to the body of the BUV.

3. Is it feasible to develop a BUV simulator with a reasonable computational power, using HLSL shaders?

This question examines the practicality of creating a BUV simulator that requires only a reasonable amount of computational power, thus ensuring a faster learning phase for the reinforcement learning algorithm.

By answering these secondary questions, we can develop a simulator that will, in the future, be used to train the reinforcement learning algorithm, ultimately enabling us to address the primary research question.

1.3 Objectives

Guided by the research questions, the primary objective of the SABUVIS project is to develop a fluid simulation. This simulation will calculate the forces applied to an underwater vehicle. Using these interactions, a reinforcement learning algorithm will be designed to optimize the vehicle's fins for improved movement efficiency, considering factors such as energy efficiency, acoustic footprint, maneuverability, and other criteria important to the Navy.

To achieve this primary objective, the following sub-objectives must be accomplished:

1. Identify the most suitable simulation method for this class of problems. The simulator must effectively model the forces on an underwater vehicle with reasonable computational efficiency. This involves choosing between Eulerian (grid-based) or Lagrangian (particle-based) methods and then selecting the optimal implementation.
2. Implement the chosen fluid simulation method in Unity 3D, ensuring it functions correctly and leverages the game engine's 3D capabilities.
3. Define the key characteristics for evaluating vehicle locomotion and train a reinforcement learning algorithm to optimize these characteristics. Potential factors include energy efficiency, acoustic footprint, and maneuverability, with the possibility of identifying additional relevant characteristics.
4. Assess the effectiveness of the algorithm in a real-world environment and determine the correlation between the simulated and real-world environments.
5. Using data from real-world testing, train a secondary reinforcement learning algorithm to model the forces on the vehicle. This neural network will replace the initial simulation, accounting for discrepancies between the simulated and real-world data.
6. Evaluate the final tool for its usability, intuitiveness, and adaptability to different BUVs, ensuring that the Portuguese Navy will be able to effectively utilize the tool.

These are extremely ambitious objectives for a master's thesis. Nonetheless, all objectives are still worth mentioning and discussing, as they serve as valuable guidelines to maintain focus on the final goal. They also help ensure that progress is made wherever possible, leaving clear paths for the follow-up work that will inevitably be required. Accordingly, Chapter 2 lays the groundwork for all of the objectives, but the practical work of this thesis will only focus on the first two objectives.

although for the purpose of this thesis, only the first two were fully achieved.

1.4 Contributions

The contributions achieved by this thesis include the related work studied to fulfill the ultimate goal of designing a tool capable of training the control mechanism of a BUUV, through reinforcement learning. A basic fluid simulation capable of simulating the interaction of water with stationary obstacles, taking advantage of the GPU for a faster simulation. Also, the testing of this tool to ensure the correct functioning, and provide insights into its performance.

During the writing of this thesis one paper was submitted and accepted into the "VI ENCONTRO DE INVESTIGAÇÃO E DESENVOLVIMENTO EM CIÊNCIAS MILITARES"², a conference gathering the portuguese military branches.

1.5 Document Structure

This introductory chapter presents the context of the thesis, outlining the motivation, objectives, and the key questions the work aims to address³. Chapter 2 reviews the relevant literature, covering topics such as: fluid simulation methods, particularly those involving fluid dynamics around vehicles; existing simulation systems for underwater unmanned vehicles and their relevance to this work; and a comparison of available platforms for conducting simulations. Chapter 3 discusses the technologies used and details the final implementation of the fluid simulation, including the data structures and system architecture. Chapter 4 contains the algorithms developed to achieve the fluid simulation, their implementation and explanation. Chapter 5 provides a user manual for operating the tool in its current state. Chapter 6 presents and analyzes the tests conducted to evaluate the thesis. Finally, Chapter 7 concludes the document and offers suggestions for future work.

²https://escolanaval.marinha.pt/pt/investigacao_web/avisos_web/Paginas/VI_EncontroInvestigaçãoDesenvolvimentoCiênciasMilitares.aspx

³During the drafting of this thesis, the AI tool Copilot was used piecewise to correct grammar and enhance the overall reading experience.

RELATED WORK

In this chapter, important related work will be presented. By familiarizing ourselves with the history and current developments, we can attain a more comprehensive understanding of the subject at hand, which is useful for the future implementation of the tool proposed by this thesis. Firstly, an overview of the historical and current advances encompassing the broad area of fluid simulation will be presented. This will ensure that the simulation, which serves as the foundation for the entire tool, is well-founded and used correctly. Secondly, the current techniques used for drone simulation, focusing on underwater and biomimetic vehicles, will be compiled and evaluated. Finally, various tools for visualizing the simulation will be compared using a table, providing an easy-to-read analysis.

2.1 Fluid Simulation

Fluid simulation encompasses all types of substances that assume different shapes depending on the container they are confined in. This can range from the most obvious elements, such as water and other liquids, to air, smoke, and even fire, which can also be considered fluids. Historically, fluid simulation was an important field of study even before computers had enough computational power to solve the partial differential equations governing fluid dynamics.

With the advent of computers, the use of fluid dynamics, and specifically Computational Fluid Dynamics (CFD), aids designers by allowing them to better understand the forces their creations will be subject to. This eliminates the need for real-world tests like wind tunnels, which involve the production of real-life prototypes that must be rebuilt every time an error is detected, a time-consuming and cost-intensive endeavor. Therefore, CFD can lead to a more robust final product with reduced costs [70]. Applications include calculating wind forces on high-rise buildings and bridges, optimizing the shape of an aerofoil in terms of lift and drag, and designing more efficient rockets.

For these applications CFD is only as effective as its ability to correctly predict the forces applied to the objects, but in recent years, a major use of CFD has been in the entertainment industry, which prefers visually pleasing simulations that are based on the

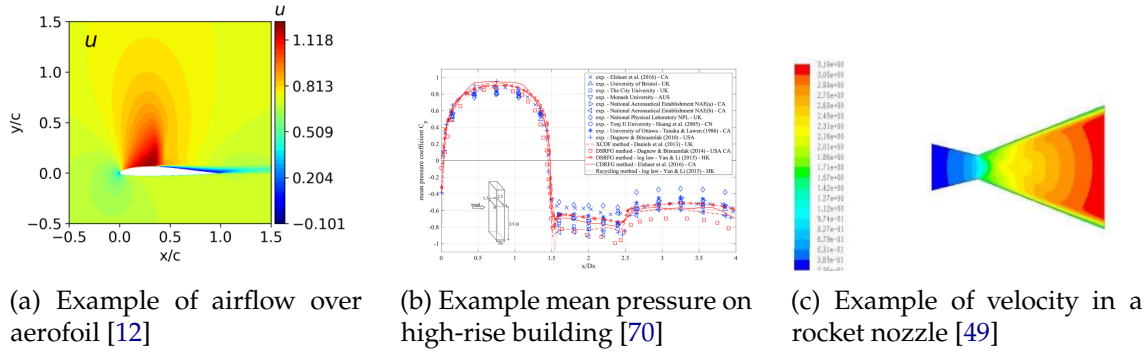


Figure 2.1: Examples of CFD applications

real world but often involve physically impossible movements. Faster simulations are also favored due to the time constraints commonly associated with the industry and the real-time nature of video games. These and other complicating factors, such as supersonic speeds, make CFD a complex subject with ample space for research. However, for the context of this thesis, some simplifications can be made to create a more manageable problem.

The basis of fluid dynamics are the continuity, momentum, and energy equations. These are the mathematical representations of three physical principles: conservation of mass, Newton's second law, and conservation of energy, respectively. Together, they form the Navier-Stokes system of equations, the derivation of which can be found in [30].

Due to the importance of the Navier-Stokes equations to CFD, a further exploration is presented in 2.1.1, followed by two possible viewpoints when implementing the equations. The fluid simulation section concludes with some recent advances made possible through the use of machine learning.

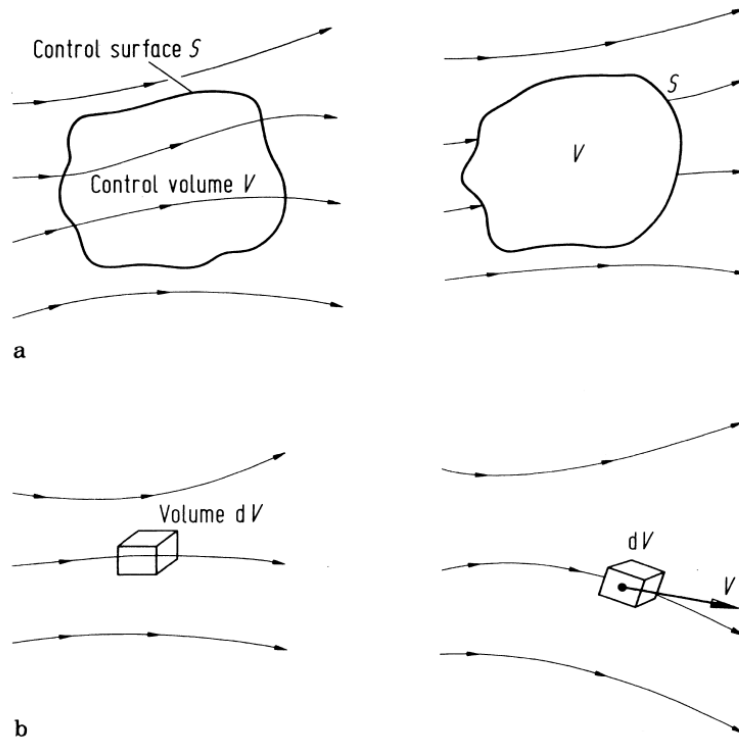
2.1.1 Navier-Stokes Equations

The Navier-Stokes equations are, as previously stated, the mathematical representation of three ¹ physics principles, that together form the fundamental governing equations of fluid dynamics. These principles are: conservation of mass, Newton's second law ($\vec{F} = m\vec{a}$) and conservation of energy, represented respectively by the continuity, momentum and energy equations.

These equations can come in many forms. In Figure 2.2 the left side images represent the view point of a control volume fixed in space, that gives rise to the conservation form of the governing equations. The right side represents when we no longer think of the volume as fixed in space, but instead it moves with the fluid, originating the non-conservation form of the governing equations. The two forms are equivalent and can be obtained from

¹Historically the Navier-Stokes equation only encompassed the momentum equation, that represents Newton's second law. However, on modern CFD literature, when referring to the Navier-Stokes equations, researchers often mean to include the whole system of equations: continuity, momentum, energy.

Figure 2.2: (a) Finite control volume approach. (b) Infinitesimal fluid element approach
Image taken from [30]



one another. However in certain CFD cases, the chosen form is important, which explains why this differentiation appears mostly in CFD literature.

Dealing with the Navier-Stokes equations is notoriously hard. One of the famous Millennium Prize Problems is given to whomever finds a solution to the equation. For further information see the formulation in [29]. This difficulty arises from several characteristics:

Nonlinearity: Because the Navier-Stokes equations are highly nonlinear, even small changes in initial conditions lead to vastly different results.

Boundary conditions: Although the equations remain the same for a multitude of cases, the solutions for each case may differ greatly, for example simulating the water flow over a fish in the ocean versus that same fish but now in a rapid river. The difference in this example comes from the boundary conditions. The most common boundary conditions can be found in Figure 2.3.

Computational resources: Computing the solution in CFD applications is extremely computational expensive, especially when a high definition is required. This has led to the use of custom meshes in certain CFD programs as seen in 2.4, where the closer to the aircraft the higher the definition is, as this is likely where the definition is relevant for the overall simulation.

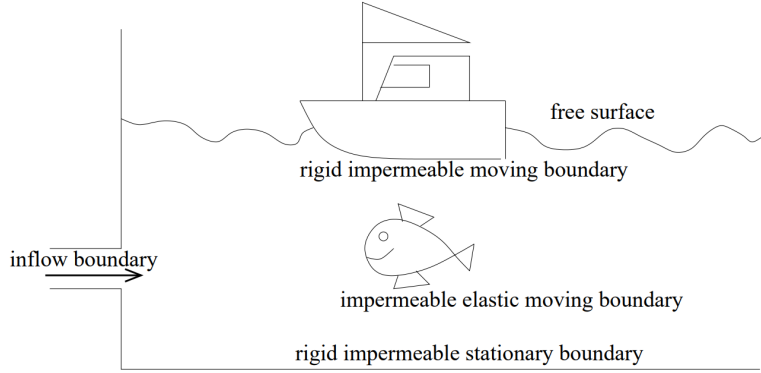
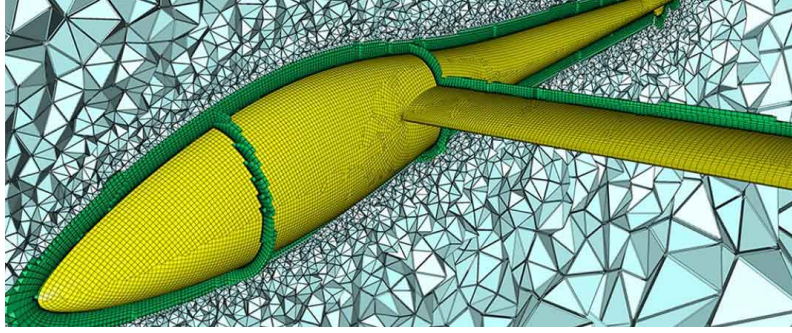


Figure 2.3: Common boundary conditions [1]

Figure 2.4: Custom mesh example[10]



When designing a CFD program, it is important to keep these challenges in mind, as certain assumptions or simplifications can be made for specific cases without significantly compromising accuracy. One such example is the use of a custom mesh, which reduces the number of calculations in less critical regions.

Some common simplifications include neglecting viscosity. Although viscosity is a fundamental component of the Navier-Stokes equations, essential for simulating fluids such as honey or tiny water droplets, in many cases, it plays only a minor role. Furthermore, even when viscosity is omitted, many numerical methods introduce errors that physically resemble viscous effects, making this simplification potentially viable[7].

Another frequent simplification is assuming the fluid is incompressible. While fluids are not truly incompressible, since that would, for instance, prevent sound transmission, this assumption is often acceptable because pressure variations in most scenarios are too small to meaningfully affect the simulation. Exceptions include cases involving acoustic waves or sonic booms, where fluid compressibility becomes essential to accurately capture the phenomena.

For the purposes of this thesis the incompressible Navier-Stokes equations will be used. They can be written as:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{F} + \nu \nabla^2 \vec{u} \quad (2.1)$$

$$\nabla \cdot \vec{u} = 0 \quad (2.2)$$

In these equations:

$\vec{u}$ represents the velocity vector of the fluid,

ρ is the density of the fluid, as an example, water has a density of around $1000\text{Kg}/\text{m}^3$,

p is the pressure, meaning the force being applied per unit area that the fluid exerts on anything,

$\vec{F}$ represents all forces applied to the fluid as a whole. Gravity is commonly included among these forces and is often the only one considered, applying a vertical force in the downward direction and accelerating objects at $9.8\text{m}/\text{s}^2$ towards the Earth's surface. However, since this thesis focuses on simulating underwater vehicles in a fully submerged environment, gravity can be disregarded.

ν stands for kinematic viscosity, representing how viscous a fluid is.

If the viscosity is dropped, we get:

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla p = \vec{F} \quad (2.3)$$

$$\nabla \cdot \vec{u} = 0 \quad (2.4)$$

These equations are generally called the Euler equations.

2.1.2 Implementation Viewpoints

When solving the Navier-Stokes equations there are two view points that are most commonly used: Eulerian and Lagrangian, often called grid based or particle based, respectively. In the following pages a brief presentation of each will be given with a comparison between them in the end, so the choice of which method is used can be well-informed and based on scientific facts.

2.1.2.1 Grid based / Eulerian

The Eulerian method, often called grid based, gets its name from the Swiss mathematician Leonhard Euler. Considering a fixed volume the Euler viewpoint chooses to fixate a fixed grid along the volume, with each cell having its own information, such as density, velocity, temperature and others. Although a simple implementation that stores all the information for the cell in the same place, would be sufficient, Harlow and Welch developed the MAC Grid technique [75], where the information is stored in different places, called a staggered grid, as seen in Figure 2.5. In both its 2D and 3D representation, we can see the pressure, represented by p is stored at the center of the cells, while the velocity, represented by u , is stored in the edges/faces, which simplify the algorithms.

Independent of the implementation choosing a correct time-step will have a great impact on the final result. If the time-step is too small, the simulation will be unnecessarily

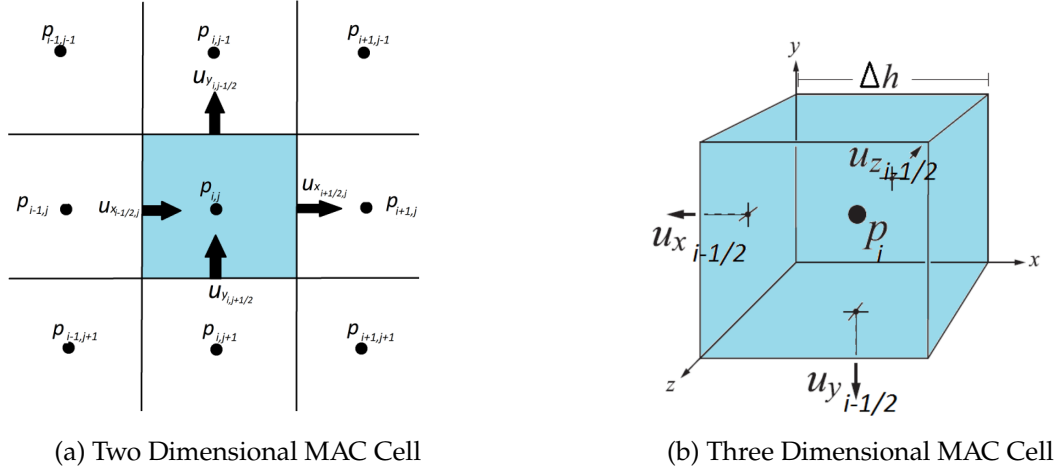


Figure 2.5: Representation of MAC Cells [6]

long. On the other hand by increasing the time-step too much we run the risk of introducing errors. So the perfect time-step is the largest possible that doesn't destabilize the simulation. The CFL condition dictates that the time-step (Δt) must be small enough for the fluid in the center of a cell to not be able to go further than the neighboring cell center. Mathematically we get:

$$\Delta t = \frac{\Delta h}{\vec{u}_{max}} \quad (2.5)$$

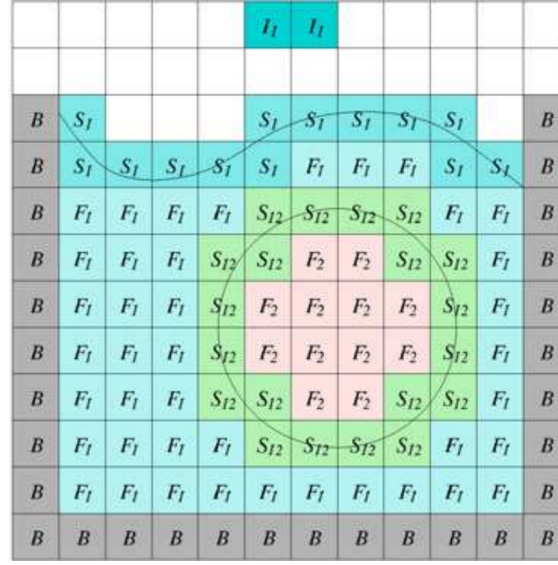
In this equation Δh is the length of a cell side, and $\vec{u}_{max}$ stands for the maximum velocity throughout the simulation. It is possible to have time-steps larger than the value given by CFL maintaining a visually plausible solution, so choosing the time-step is usually left to the user's discretion.

Advection is the ability to "transport" quantities through the simulation grid. It is used in CFD to calculate how some quantity, such as temperature, density and velocity will change after a specified Δt . We calculate this "transport" based on the velocity field, which might seem wrong at first to advect velocity based on velocity, but it is indeed right. In modern literature, implementations use a technique called *backwards particle trace*, which is considered a semi-Lagrangian advection, as it assumes the viewpoint of an imaginary particle to calculate its next position.

After advection, the incompressibility of the fluid must be assured. This step is called **Pressure solve/projection**. It's at this point that the use of the MAC grid previously discussed is advantageous. Besides ensuring incompressibility, pressure projection is used to enforce the defined boundary conditions. Two frequently used boundary conditions are Dirichlet and Neumann, used respectively for free surfaces and solid boundaries.

With all quantities calculated, the next step is to "draw" the fluid. MAC grids make it easy to simply color each cell based on it's content, as seen in Figure 2.6, where B stands for boundary, I for inflow cells, S for any surface between two fluids, including air, and F for fluids, note that there are two fluids in this example. This simple method is good

Figure 2.6: Fluid flagging example on MAC grid [46]



to convey information, but a more complicated method is necessary to have a visually compelling fluid.

2.1.2.2 Particle based / Lagrangian

In contrast to the grid based approach, the Lagrangian techniques are based on a set of particles that can move along the specified space. Each particle has its own quantities, such as position, velocity and mass, but other quantities are calculated based on more than one particle, for example density is inferred from the number of particles in a predetermined radius. The particles then interact with each other to ensure that they behave as a fluid and don't accumulate all in the bottom. The forces between particles generally have the form:

$$f(x_i, x_j) = F(|x_i - x_j|) \cdot \frac{x_i - x_j}{|x_i - x_j|} \quad (2.6)$$

Here $F()$ is an arbitrary function that returns the force magnitude given the distance between two particles. The fraction on the right is the mathematical representation of the normalized vector orientated from x_j to x_i . For the purpose of fluid simulation the function $F()$ should be derived from the Navier-Stokes equations, which was done by Müller et al. [48], using a method based on Smoothed Particle Hydrodynamics (SPH). Further work was done in [44] allowing for real-time simulation.

Having a compelling visual representation of the fluid, from the particles, can be achieved from by generating the isocontour as seen in Figure 2.7.

Figure 2.7: Obtaining the fluid surface from particles [24]

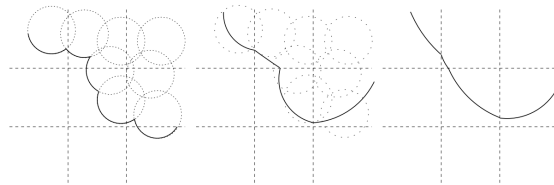
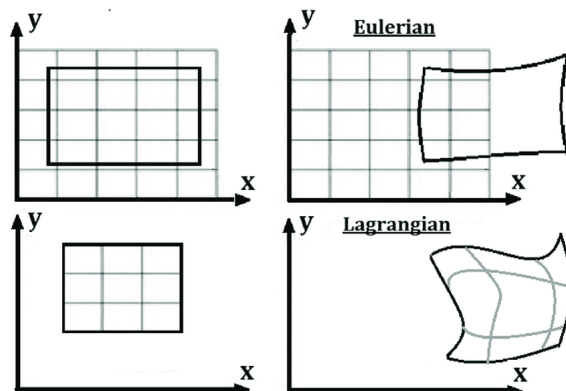


Figure 2.8: Evolution over time of Eulerian approaches compared to Lagrangian [67]



2.1.2.3 Comparison

The Eulerian approach is based on a fixed mesh through which the fluid is moving, as opposed to the Lagrangian where the particles accompany the fluid. For a visual comparison of the two approaches see Figure 2.8.

When choosing which technique should be taken for a given application, it's important to understand that both techniques are useful, and a choice should be made considering the requirements. As a rule of thumb, particle based approaches are faster but less accurate, while grid based approaches have higher accuracy, but are slower. The loss of accuracy in particle approaches can be primarily attributed to the difficulties of dealing with spatial derivatives on an unstructured particle cloud [7], although they tend to be easier to program and a more intuitive way to think about fluid dynamics. Likewise they are much faster, useful for real-time applications such as video games.

Despite being slower, grid based approaches can have higher time-steps, without compromising accuracy. The complication in the grid approaches, comes from the elevated number of edge cases in boundaries. Some of these were tackled by McKee et al. in a review of the MAC method [46]. Eulerian approaches also work better for contained areas, without influences from the exterior. The contained area with fixed points, can be very restrictive compared to the Lagrangian, but the "looseness" of the latter approach means a computationally expensive algorithm to find the neighbors must be employed.

2.1.3 Deep Learning for Fluid Simulation

Machine learning is a broad area of computer science that encompasses algorithms capable of automating tasks through the extraction and processing of data. Today, examples of machine learning can be found across most areas of our lives, including finance [3, 17], medicine [19, 59], automotive [4], robotics [8], and games [63, 66]. Deep learning is a subset of machine learning, inspired by brain neurons, forming what is now called a neural network. Deep learning is extremely popular due to its ability to closely approximate complex solutions to previously difficult tasks.

In CFD, deep learning methods can be divided into two broad categories: physics-driven and data-driven. Physics-driven methods generally use neural networks to approximate solutions for the Navier-Stokes partial differential equations [21]. This approach is both data-efficient and significantly reduces computational costs [69]. Data-driven methods rely on previously gathered data, whether physical or simulated [26], to learn the underlying physics instead of learning the governing equations.

In the next sections, an overview of these methods will be provided, along with the most common implementations and results, concluding with some insights on how the adoption of deep learning techniques can improve the simulations that this work aims to develop. For a more thorough exploration of this subject refer to Brunton *et al.* [9], Lino *et al.* [39] or Willard *et al.* [76].

2.1.3.1 Data-driven

As previously mentioned, data-driven deep learning in the context of fluid dynamics learns the physics instead of solving partial differential equations. This process can be compared to how humans learn about the world around them. Without knowing the mathematical models behind the world, humans can still predict fairly accurately how much force to kick a football with or where to position themselves on a tennis court to respond to an opponent. This knowledge comes from a large number of previous examples of what happened rather than from mathematical calculations. Similarly, data-driven approaches aim to predict the evolution of a fluid simulation based on the vast number of examples given during the training phase. Because the model's predictions are based on these examples, one of the most important parts of developing data-driven models, and indeed any deep learning model, is the quality and quantity of the examples. These examples are typically generated by numerical solvers, although real-world data can also be used. However, modern approaches suffer from poor extrapolation, error accumulation in time-dependent simulations, and difficulties in training against turbulent flows [39].

In 2016 Guo *et al.* [26] spearheaded the use of data-driven Convolutional Neural Network (CNN) for fluid simulation in a steady flow approximation, finding a speed-up of 2-4 orders of magnitude higher. In Figure 2.9, it can be observed that these predictions, despite the substantial speed-up do not introduce many errors.

Figure 2.9: 2D prediction of velocity field magnitude [26]. First column uses a numerical solver. Second column uses the trained CNN. Last column contains the differences between the first two

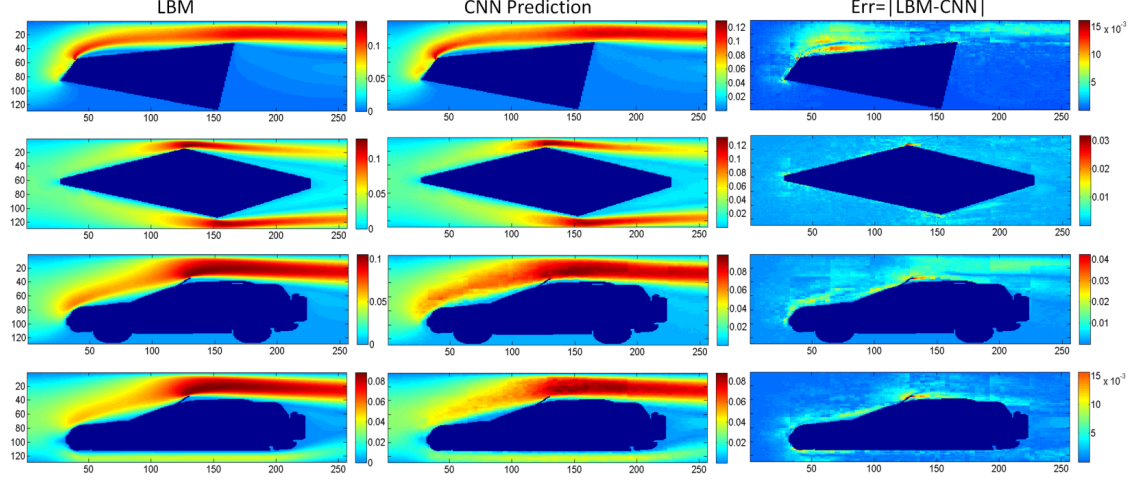
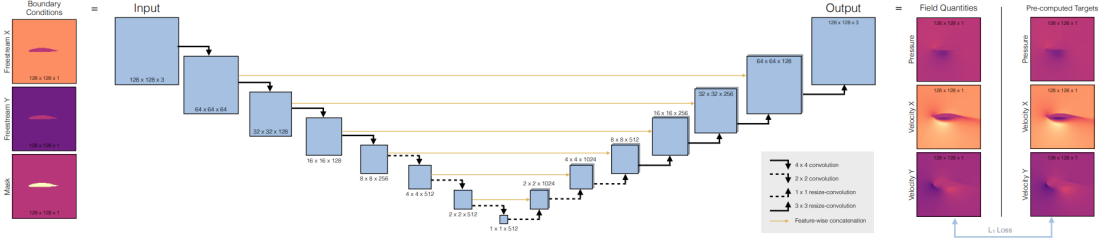


Figure 2.10: Image from [71], that represents the U-net architecture used. Given three input "images", containing only the airfoil, the neural network will output the predicted result for pressure, velocity in the X axis and velocity in the Y axis, each in the respective output "images".

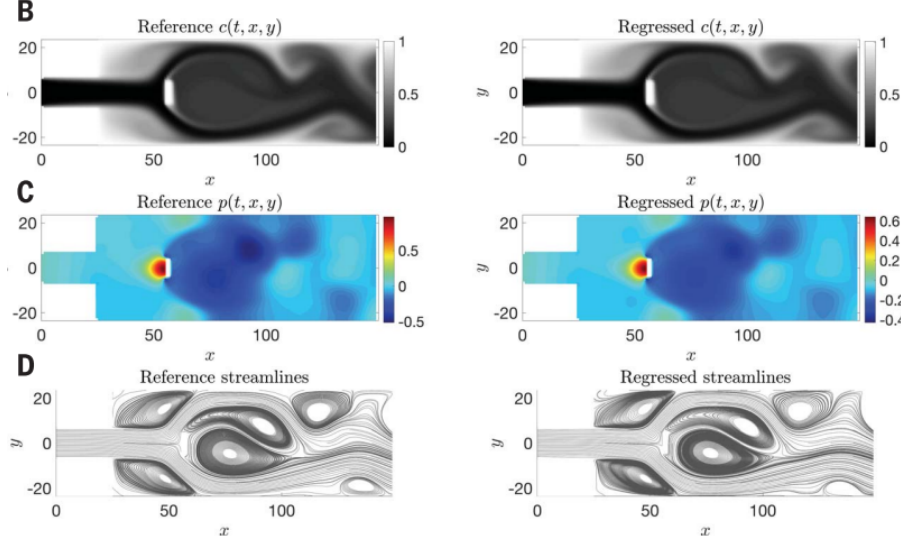


Most data-driven approaches use CNNs, with ResNet [27] and U-Net [60] being the most commonly adopted. U-Net, in particular, has seen wide adoption due to its consistently good performance [11]. This popularity can also be partly attributed to the architecture of U-Net, shown in Figure 2.10, which consists only of convolutional layers, making the neural network capable of receiving inputs of varying sizes. Besides CNNs [36], other neural networks used include Recurrent Neural Networks (RNN) [22] and Graph Neural Networks (GNN) [62]. GNNs have already proven useful in simulating fluid dynamics and other physics [54, 40, 61], while RNNs, despite being designed with time series in mind, have thus far been outperformed by CNNs [25].

2.1.3.2 Physics-driven

As already established solving the Navier-Stokes equations is hard and usually approximations, on the solution of the partial differential equations, are made in order to maintain an

Figure 2.11: Image taken from [58], presenting the results of using a PINN for fluid mechanics. (B) is depicting the concentration, (C) is depicting the pressure and (D) is depicting the streamlines



acceptable run time on the simulations. Following the universal approximation theorem proposed in [28], which states that any function may be approximated by a sufficiently large and deep network, physics-driven techniques exploit this theorem to approximate the partial differential equations.

Physics-informed neural networks (PINNs), introduced in [57], are networks that are "trained to solve supervised learning tasks while respecting any given laws of physics described by general nonlinear partial differential equations." PINNs have already demonstrated their usefulness in fluid mechanics as can be seen in Figure 2.11. Furthermore, in [58] implementation of PINNs, an accurate streamwise prediction was obtained, with less than a 3% error.

Physics-driven solvers properties make them more accurate, without necessitating a mesh for the fluid domain [39]. Additionally, the solution to the partial differential equations is obtained in a differentiable, closed analytical form, which can be useful for future calculations. This makes the evaluation of the solution in any point in time and space trivial. Although the further we extrapolate in time the higher the uncertainty becomes. Du et al. [20] developed a modified version targeting the uncertainty, but this method required an increase in computational power, negating the benefit of using PINNs. At last physics-driven methods do not suffer from stability constraints. Currently, the main problem to overcome is the high residuals of the partial differential equations, compared to numerical solvers, with possible solutions presented in [34].

2.1.3.3 Conclusions

Deep learning has many applications in fluid dynamics, but most suffer from problems plaguing deep learning as a whole, such as a high dependency on the quality of the

data provided. Because physics-driven approaches are exactly that, they produce much more reliable results when faced with vastly different problems compared to data-driven predictions, which aren't as good at generalizing. Moreover, these models generally do not take into account physical constraints and may fit the observational data very well but lack compliance with the underlying physical principles [21]. Therefore, caution should be exercised when using neural network models for extrapolation tasks [9].

Traditional CFD methods produce high-accuracy results, but they are computationally expensive and the time required to obtain results is often measured in hours, or even days for complex prototypes [26]. Thus, any speed-up is welcomed.

For this work, deep learning will be used to predict the forces applied on the BUUV. As a quick reminder, after the initial fluid simulator is in place and the machine learning algorithm has found an optimal movement for the given simulator, the BUUV will be tested in a real-world environment. The idea is to gather data from real-world experiences and the simulation, to then train a model that, given the BUUV actuators (for example, moving the back fin all the way to the left), will output the forces applied on the BUUV in six degrees of freedom: x , y , z , and the rotation over each axis. This application differs from the previously presented applications of deep learning, but due to the similarities and comparative simplicity, only having to output six values compared to predicting the pressure or any other quantity in an entire volume, it is predictable that the success can be translated to this work's application of deep learning. Additionally, because the neural network has no assumptions about the fluid, such as incompressibility and inviscidity (no viscosity), the predictions can potentially be more accurate than simulators that do make these assumptions.

2.2 Simulation of Unmanned Vehicles

Unmanned vehicles, are extremely useful. Because they do not carry any human, they can be much more versatile. The most common type of unmanned vehicles are Unmanned Aerial Vehicles (UAV), also known as drones, but there are in fact other types, such as Unmanned Underwater Vehicle (UUV) and Unmanned Ground Vehicles (UGV). Although unmanned vehicles will not likely substitute all types of vehicles, they have an important role to play in our day to day, by virtue of being cheaper than a similar vehicle that could transport humans. Take for example a commercial drone that can be bought off the shelf, at relatively low price and is capable of filming high definition videos. Another advantage is the lowered risk for humans. Because humans are not in the vehicle, it can be sent to a more dangerous place without fearing the loss of human lives. The vehicle also doesn't have to be designed with high security measures, helping to also lower costs. Applications of unmanned vehicles can be seen in Figure 2.12, which include the aforementioned filming [45], agricultural [56], scientific exploration [73], mining [52], search and rescue [43], leisure and many others.

(a) MQ-1 Predator UAV³(b) Commercial DJI drone⁴

(c) UGV used for demining [16]

(d) REMUS a military UUV⁵

Figure 2.12: Examples of unmanned vehicles

In the military, unmanned vehicles are also important, with the previously given advantages being even more important due to the heightened degree of danger a human would incur if he was physically present. In 2001 the US congress realized the advantages and mandated its Department of Defense (DoD), to have one third of all operational deep strike force aircraft be unmanned and one third of operational ground combat vehicles also be unmanned². Figure 2.12 contains some examples of military applications, such as missile strikes, demining, surveillance, reconnaissance, border patrols and the like.

One important subcategory of unmanned vehicles is autonomy, usually referring to the control mechanism. In fact most of the cited material involves autonomous vehicles. This autonomy is brought by the miniaturization of the sensors and cameras and the advances in communication and control systems [51].

Autonomous unmanned vehicles inherit all the advantages of unmanned systems and offer additional benefits. Their extended reach, both in terms of distance and the variety of environments they can access, is enhanced by autonomous capabilities. Locations that were previously inaccessible due to communication constraints can now be explored. The absence of a human controller reduces the need for outbound signals, which is advantageous for stealth missions. Additionally, autonomy lowers human resource costs by eliminating the need for active human control. In some cases, it also increases operational effectiveness, as the demand for automated, robotic, and unmanned systems is largely driven by applications that are repetitive, unpleasant, or hazardous [23]. Since computers do not experience fatigue or fear, their performance remains unaffected by such conditions.

For these reasons, autonomous unmanned vehicles play a crucial role in various domains, particularly in the military. Given the current global context, the European

²Section 220(a) (2) of the Floyd D. Spence National Defense Authorization Act for Fiscal Year 2001 (as enacted into law by Public Law 106-398; 114 Statute 1654A-38), mandates that “It shall be a goal of the Armed Forces to achieve the fielding of unmanned, remotely controlled technology such that:

- By 2010, one third of the operational deep-strike force aircraft fleet are unmanned, and
- By 2015, one-third of the operational ground combat vehicles are unmanned.”

³<https://www.defensedaily.com/general-atomics-delivers-final-two-predators-to-italy/international/>, last access on: June 2025

⁴<https://www.dji.com/pt/camera-drones>, last access on: June 2025

⁵<https://www.defenseadvancement.com/news/new-uuv-unveiled-for-naval-special-operations-forces-missions/>, last access on: June 2025

Union is increasing its investment in military development through initiatives such as SABUVIS II, in which this work is involved. One of the critical phases in designing an autonomous vehicle is testing. However, testing can be both costly and time-consuming. Indoor testing is a possible solution, but it is often impractical due to limitations in space, cost, and maintenance. Alternatively, conducting experiments in remote locations poses challenges such as potential vehicle damage, limited resources, and high costs associated with manufacturing, procurement, and upkeep [33]. While these testing methods remain important, it is beneficial during the development phase to employ faster and more cost-effective approaches.

Simulation offers a promising alternative. It can accelerate testing procedures, aid in identifying and preventing hardware malfunctions, and reduce costs. Finn *et al.* [23] argue that the goal of simulation is to aggregate complex phenomena into simpler systems. This implies that even if a simulation does not perfectly replicate real-world conditions, it remains valuable as long as it can effectively mimic the underlying complexity.

In this section, modern simulation techniques will be presented, with a special emphasis on underwater vehicle's simulation, understanding the similarities and differences to learn what can be adapted and discarded, respectively. Since biomimetic vehicles have their particularities, a more in depth study on them will be presented, followed by the possible control mechanisms, and how reinforcement learning can help find the optimal control mechanism.

2.2.1 Underwater vehicle simulation

Compared to the other types of unmanned vehicles, underwater vehicles have an added degree of difficulty. Firstly the communication is much harder, since radio communications are virtually impossible in water. When working with UUVs acoustic communication can be used, but this severely restricts the bandwidth [5]. Moreover, the characteristics of water can vary greatly along the path from emitter to receiver. For example, water type (fresh or salt), depth, temperature, pH, impurities, and water composition can all affect the propagation of acoustic waves or any other means of communication. Multipath in shallow waters ($< 100m$) and refraction in deep water also pose challenges. For these reasons, the adoption of automation becomes even more important, as it removes the need for continuous communication.

Secondly, movement has different requirements. For example, in a UUV, it might be important for the vehicle to remain still while conducting a survey of the sea floor. This task is trivial for UGVs by simply locking the wheels, and it is more complicated for AUVs due to wind. In a UUV, the difficulty is similar due to currents, but if the currents are too strong, the relatively low speeds achieved by UUV means they are more influenced by the currents.

Lastly, testing becomes much harder. Despite the already presented disadvantages of testing prototypes in the real world, if the prototype is an underwater vehicle, this would

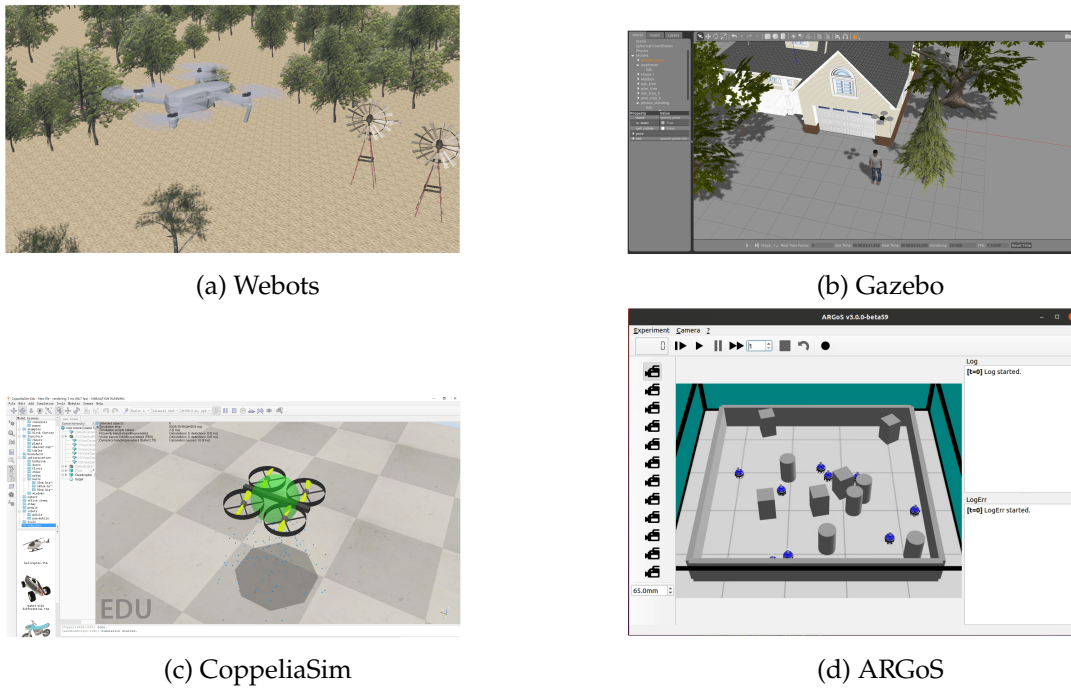


Figure 2.13: Simulation of UAV using different platforms

additionally mean relocation to a lake, pond, or the sea, which are not as readily available as the testing grounds for a UAV or a UGV. Furthermore, there are many more unknowns since testing in a real-world scenario involves a lack of communication during the mission, hoping the UUV will resurface in the predetermined location.

Therefore, in current literature there are several simulation platforms, such as Webots³, Gazebo⁴, CoppeliaSim⁵, ARGoS⁶ and more. More than one survey has been made on simulation platforms for unmanned vehicles [13, 15] these surveys focus on simulation Platforms for different types of unmanned vehicles, with most of these also having features for UUV simulation. For a more UUV directed simulation there is [37, 78].

A more thorough study on the simulators and their capabilities will be done in Section 2.3.

2.2.2 Biomimetic Vehicles

Recent technology developments have turned present-day unmanned vehicle systems into realistic alternatives to traditional marine animal survey methods. Benefits include longer deployments, improved mission safety, mission repeatability, and reduced operational costs [73]. But a typical UUV, still has its downsides, namely its acoustic signature.

³Cyberbotics. Webots User Guide. Available online: <https://www.cyberbotics.com/doc/guide/sensors/> (accessed on June 2024)

⁴Open Source Robotics Foundation. Category: Sensors. Available online: <http://gazebo-sim.org/tutorials?cat=sensors> (accessed on June 2024)

⁵Coppelia Robotics, L. CoppeliaSim User Manual. Available online: <https://www.coppeliarobotics.com/helpFiles/index.html> (accessed on June 2024)

⁶<https://www.argos-sim.info/> (accessed on June 2024)

Screw-like propellers have an unnatural acoustic signature. This is unfavorable for both scientific studies and military operations. Scientific studies monitoring any sort of marine life, would be useless if the marine life changes behavior in the presence of the monitoring devices. Therefore the higher the differences between the vehicle and its surroundings, the higher the risk of unreliable results. In military operations being hard to distinguish from the environment is an important feature and the acoustic signature is the commonly used detection mechanism by enemies, consequently, a more natural acoustic signature also has its benefit in stealth missions.

Biomimetic underwater vehicles (BUV), meaning underwater vehicles inspired by biological life, often in terms of locomotion, improve on this flaw because the source of the most characteristic acoustic signature is no longer in use. The vehicle becomes harder to detect and possibly less intrusive for the marine life around it. Additionally, the typical efficiency of a marine propeller is about 40%–50%, while it is slightly less than 70% in some fish [31], revealing a possible increase in energy efficiency, which is important due to the limited size of batteries in a BUV.

Nevertheless, bioinspired fins bring a drawback. Fish have benefited from millennia of evolution to efficiently use their bodies for propulsion, but the relationship between the movement of the fins and the propulsion generated isn't straightforward, let alone the most efficient movement, unlike screw-like propellers where the translation from actuator to movement is more direct. This has reduced and slowed the research being conducted, which is why this work chose to address it.

Biomimetic refers to the inspiration on any biological life, and there is a high biodiversity, hence BUVs can vary immensely some examples can be seen in Figure 2.14.

Thus, an all encompassing simulator for BUVs is unfeasible. As such, in context of this

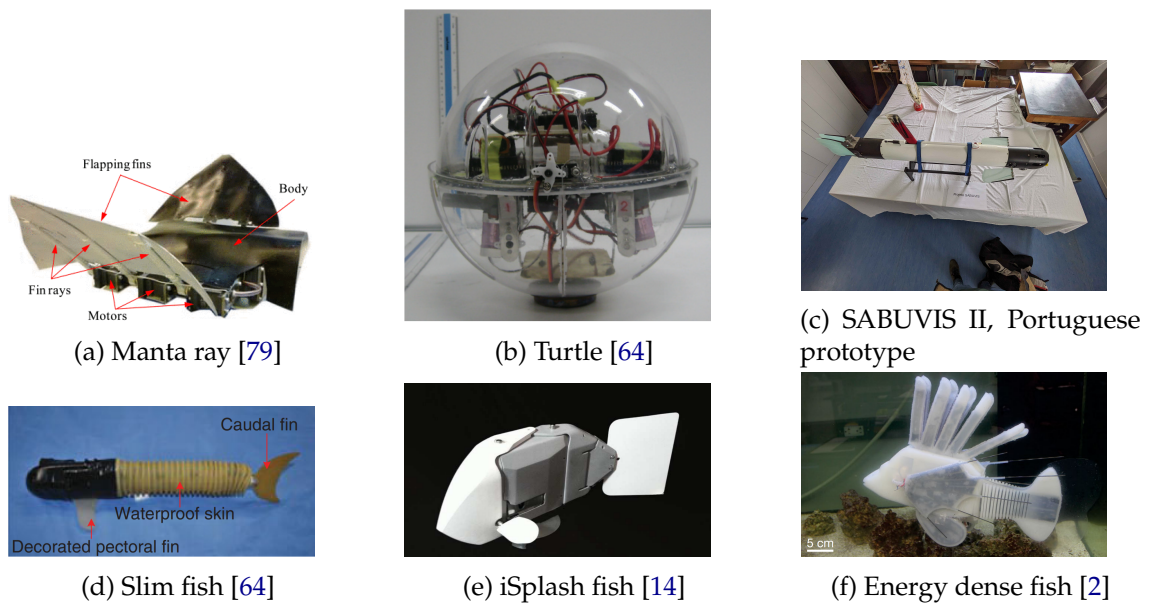


Figure 2.14: Some examples of BUVs

work the broadest possible simulator will be developed, focusing mainly on the SABUVIS II Portuguese prototype. Although some BUUV simulators already exist, they do not solve the exact problem we are trying to solve.

In 2022, Liu *et al.* [41] develops a physics-based simulation framework for underwater robot learning. It consists of three components:

Robot model - Motivated by the anatomy of the fish structure, the fish robot locomotion is modeled by its skeletal structure, that is, bones connected by joints. Covering the skeleton are the flesh and skin. In the fluid-robot simulation the flesh is assumed to be a rigid body, done for efficiency's sake. During the simulation the changing of joint angles, will be the source of the robot movement.

Simulation method for predicting fluid-robot interaction - Due to the requirements of this simulation, particularly the learning of underwater robot locomotion, performance is crucial, since the learning algorithm will often run fluid simulations many times. For this reason, instead of simulating a large fixed domain, the simulation is always centered on the robot, assuming the fluid further away from the robot by a certain distance will not influence its motion. Although the Navier-Stokes equations are considered, they are not utilized directly. Instead, a GPU-optimized LBM solver with an immersed boundary (IB) method [38] is employed for higher efficiency.

Robot learning method that leverages the simulation method - Following common practices, when training complex control policies for swimming, reinforcement learning is used. Four tasks are trained using the reinforcement learning approach: reaching a given target location, pose in order to make a U-turn, two-fish schooling, path following.

In 2023, Yan *et al.* [77], also presented a BUUV control learning method, this one focused on an in-house robotic shark. In this approach, the authors wish to overcome the observed difficulty of transferring the learned locomotion from virtual environment to the real-world, credited to the small errors introduced in simulations, due to simplifications or unaccounted effects. A previously used technique for training ground or aerial robot, involves the use of reinforcement learning, but foregoes the simulation and sources the data directly from the real-world. The use of this technique for aquatic robots has been limited. Due to the hard-to-predict environment, only small tasks had been trained previously. But their goal was to train complex tasks such as real-world exploration. After analyzing the results they conclude this is a promising approach for other underwater control problems.

2.2.3 Learning Control Mechanisms for BUUV

When designing control mechanisms for autonomous vehicles, there are at three popular options: model-based controller, PID controller or neural network. Due to complex fish

dynamic models, the control of fish-like robot swimming cannot be simply achieved by a model-based controller. In addition, since the control input is usually high-dimensional and cannot be fully decoupled, the PID controller cannot be used either [41]. On the other hand, using a neural network to represent the control policy is a good approach, as neural networks can learn the complex fish dynamics and are effective at handling high-dimensional inputs. For this type of application, reinforcement learning is often used to tune the neural network for the task at hand.

The next sections will introduce reinforcement learning, followed by a comparison of relevant applications for BUUV navigation.

2.2.3.1 Reinforcement Learning

Reinforcement learning is a branch of machine learning where learning is achieved through interaction with the environment and trial-and-error, reinforcing the good actions. Some well-known applications of reinforcement learning include the chess and Go engine AlphaZero [65] and protein folding using AlphaFold [32]. It has also been applied in many other areas, such as natural language processing, computer vision, healthcare, education, finance, and robotics [47], among others.

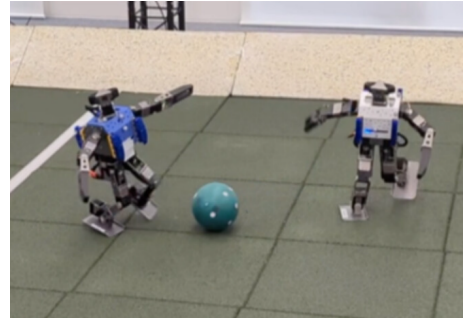
At the base of every reinforcement learning algorithm is an agent that periodically receives a state containing all necessary information to take the next step. Then, a policy, in our case represented by a neural network, will output the actions to take. This policy must be tuned in order for the agent to exhibit desirable behavior. The desired behavior is expressed through a reward function, which returns a scalar given a state. This can include rewarding positively or negatively: for example, a chess engine for taking a piece, a BUUV controller for maintaining a high speed, reaching a target location, colliding with an object, or succeeding or failing at a given task.

A good implementation of reinforcement learning depends heavily on the choices made prior to the actual learning such as the neural network architecture, the reward function, the inputs composing the state and the possible actions. For example, a neural network with sixteen nodes per layer and 1 hidden layer, is unlikely to be able to model the complexity of a chess game, but might be enough to model the snake game. The success is also highly dependent on the reward function. If the agent is rewarded for achieving a high speed, but it is unstable at that speed, it might learn that it receives a higher reward for this cycle, instead of a consistent, but lower speed that would result in a higher average speed. In the next section (2.2.3.2), the choices made in some applications will be presented.

Reinforcement learning methods often include the dilemma of exploitation vs exploration. Exploration represents the choice to deliberately preform an action, although it's not the calculated better, with the purpose of learning the effects of taking that action, while exploitation utilizes the already acquired knowledge to decide on an action. A common approach is to have a value ϵ that represents the probability of preforming



(a) Spot, a dog-like robot developed by Boston Dynamics⁷



(b) Football playing bipedal robot developed by DeepMind⁸

Figure 2.15: Application of reinforcement learning

exploration, with its value being decreased over the course of the learning, so that by the end there is little exploration. A replay memory is also an important piece of the reinforcement learning system. It stores past states, the taken action and the observed reward, allowing for the feeding of iid (Independent and Identically Distributed) data, during the learning phase, leading to improved results.

2.2.3.2 Reinforcement learning applications

Reinforcement learning has already demonstrated its use in robotics, for example Spot, a dog-like robot developed by Boston Dynamics, which leverages reinforcement learning in its locomotion control system and a bipedal robot that used reinforcement learning to play football, both of which can be seen in Figure 2.15.

Many more uses can be seen across the field of robotics, but for now there will be a focus on two uses on the development of BUVs [41, 77]. They have been presented earlier, at the end of section 2.2.2, so next is an analysis of the choices made concerning the reinforcement learning. For reference, Figure 2.16 contains a visual representation of each method.

In FishGym [41], Liu *et al.* use a state that contains the position and velocity for each joint, as well as the relative translation, velocity, and rotation between time steps. On the other hand, for their robotic shark [77], Yan *et al.* chose to include only three distances and yaw (rotation on the vertical axis). These distances are to the first object that appears directly in front, to the left, and to the right. The differences between the two approaches can be attributed to their different goals: FishGym aims to develop a path-following robot, while the goal of the robotic shark is to explore its environment.

When it comes to the reward function, both present four evaluated behaviors. In FishGym, they are: **Cruising**, which measures the ability to reach a target location; **Pose control**, for controlling its pose during a U-turn; **Two-fish schooling**, where the fish

⁷<https://bostondynamics.com/blog/starting-on-the-right-foot-with-reinforcement-learning/> (last access on June 2024)

⁸<https://www.science.org/doi/epdf/10.1126/scirobotics.adi8022> (last access on June 2024)

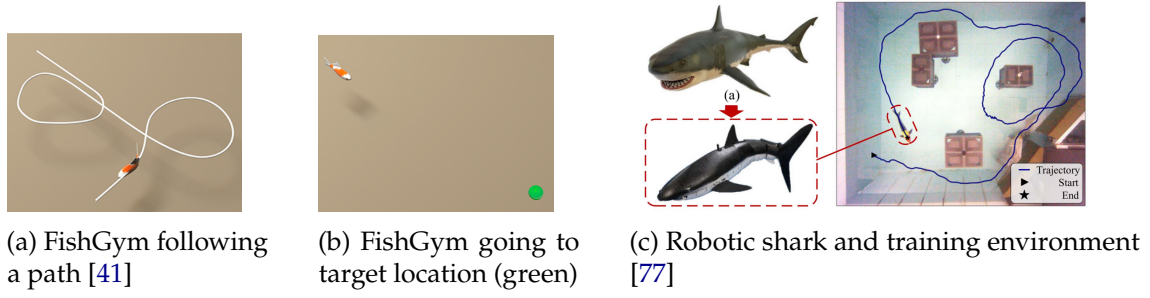


Figure 2.16: Images from the compared works

must follow a leader fish as closely as possible; and **Path following**, for following a given arbitrary path. In the robotic shark, the four measures are: **Collision-free**, which punishes the robot when it hits something; **Exploration**, which rewards the robot for exploring new areas and prevents it from standing still to avoid collisions; **Smooth turning**, which the authors explain as preventing the robot from simply doing spirals to satisfy the previous two rewards, by monitoring the yaw angle and how it changes; and finally, **Moderate speed**, which is also rewarded.

They also differ in the output of the neural network, which in reinforcement learning corresponds to the action. FishGym outputs the actuation forces for each joint and the change in the bladder's volume to control buoyancy, allowing for vertical movement. The robotic shark, instead of directly controlling the actuators, outputs the frequency and offset for the wave its tail should follow. This is fed into a Central Pattern Generator (CPG) that can smoothly change the waveform.

FishGym benefits from having a virtually infinite environment since the simulation is centered on the robot, assuming that anything outside its local environment does not currently affect the robot. The local environment had a resolution of $100 \times 100 \times 100$ and a physical time step of 0.004 seconds, resulting in a cost of around 3.5 seconds for simulating one physical second on an NVIDIA TitanXp GPU with 12 GB of memory. The policy network consisted of two layers, each containing 256 units with a ReLU activation function. The training process usually started to converge after 6 hours (1000 episodes) and achieved smooth convergence within 10 hours. They did not use a replay buffer, instead choosing to train the neural network after 2000 simulations, each with 50 time-steps.

The robotic shark enjoys a continuous environment but in the experiments was confined to a tank with dimensions $5 \times 4 \times 1.5m^3$ (*length* $\times$ *width* $\times$ *depth*), with four boxes serving as the objects to avoid. The time step for measurements is 500 ms. No architecture for the neural network is provided. Before reinforcement learning, the authors employed professionals to control the robot, recording the states and actions taken. This experience is gathered in a replay buffer, ensuring good examples to learn from. A second replay buffer stores all experiences during the reinforcement learning process. Besides running out of time, collisions also terminate the learning process.

2.3 3D Visualization and technologies

In 2007, a survey by Craighead *et al.* [15], stated that there are many available simulators and game engines that can be used to simulate robotic vehicle with high physical and functional fidelity, and therefore there was no need to build a robotic simulator from the ground up. Following his advice, instead of developing a new robotic simulator this work will leverage an already existing one. For an informed decision on which platform is most well suited for the problem at hand, this Section will present the criteria used to evaluate the platforms followed by the presentation of the platforms. Near the end there is a table summarizing the gathered data, finalizing with the explanation behind the chosen platform.

2.3.1 Criteria

In order to evaluate simulators from the perspective of a BUUV researcher, a criteria must first be formulated. These criteria will allow for an objective examination of the available platforms, and are the following:

- **Fidelity** - Fidelity refers to how accurate the simulator is at emulating the real world. This can take two forms: physical and functional. Craighead *et al.* [15] presents physical fidelity as how accurately the outer world is represented in the simulator. For BUUV simulation it can refer to the virtual environment having an expected depth, expected amount of physical obstacles such as rocks and the sensors behaving as they would in the real-world. Functional fidelity is explained as the accurateness of the forces being applied on the robot. When evaluating the different platforms a higher emphasis is placed on the functional fidelity since it has a bigger impact on the final result.
- **Ease of Development** - Ease of development also appears in Craighead's survey and refers to the general ease with which the simulator can be adapted, for example: Is it easy to modify the environment? Can the simulator be modified to simulate new equipment? Is there plenty of available documentation? Game engines will naturally score high in this criterion, as they are highly customizable and often have a big community behind them.
- **Cost** - Cost refers to both monetary costs, for example if the software is paid, but also refers the time costs when installing and other necessary processes.
- **Extensibility** - Extensibility is the only criteria that is not presented in Craighead's survey, but was deemed important due to the specific problem at hand, due to the high variance of BUUV models, a simulator that can be extended to the specificities of each BUUV is preferred.

2.3.2 Platforms

Gazebo⁷ [35] is a general purpose robot simulator. It is built upon the Ogre3D rendering engine, with the possibility of compilation to different physics engines. Gazebo has been tightly integrated with ROS (Robotic Operating System), which is used in the SABUVIS II prototype, but it is better designed for land-based robots, with no built-in way to create underwater visual effects. Some of these setbacks might be solved by plug-ins.

Unity⁸ is a cross-platform game engine, primarily used to create games. Unity provides a robust physics' engine to simulate real world physics, including fluid dynamics. Scripts can be programmed using C#. Perhaps the greatest strength of Unity lies in its extensibility, with the Unity Asset Store containing a wide variety plug-ins, for example ml-agents⁹, an open source project that enables reinforcement learning inside the unity environment. The flexibility of Unity might add some overhead in processing time, but when developing prototypes this is not a big problem.

MATLAB¹⁰ is a numerical simulation environment that supports visualization via a Virtual Reality toolkit. A standard annual license costs 900€, with additional toolboxes increasing the price. Although it has previously been used for robot simulation, not much modern literature uses it.

Unreal Engine¹¹ is a game engine with similar characteristics to the Unity game engine, although with a smaller percentage of the market¹². Unreal engines uses C++ as the scripting language.

UWSIM [55] is an open-source project designed specifically for underwater vehicles, using the Bullet physics engine. It has good fidelity and a wide range of readily available sensors, such as sonars. Similarly to Gazebo it can be easily integrated with ROS. One downside is the limited documentation available. UWSIM allows for a highly realistic visual simulation.

Simulator	Fidelity	Ease of Development	Cost	Extensibility
Gazebo	Medium	Medium	Low	Medium
Unity	High	High	Medium	High
MATLAB	Low	Medium	High	Low
Unreal Engine	High	High	Medium	Medium
UWSIM	Medium	Low	Medium	Low

Table 2.1: Table comparing the different possible simulation platforms

⁷<https://gazebo.org/home> (accessed on July 2025)

⁸<https://unity.com/> (accessed on July 2024)

⁹<https://github.com/Unity-Technologies/ml-agents> (accessed on July 2024)

¹⁰<https://www.mathworks.com/> (accessed on July 2024)

¹¹<https://www.unrealengine.com/en-US> (accessed on July 2024)

¹²<https://www.valuecoders.com/blog/technology-and-apps/unreal-engine-vs-unity-3d-games-development/> (accessed on July 2024)

2.3.3 Chosen Platform

The chosen platform was Unity because it presents the best characteristics. The only undesirable factor is the medium cost, which is due to the price of a license. However, no cost will be incurred during the prototyping phase. Additionally, Unity has one more beneficial factor that was not evaluated in the table, as it is subjective: familiarity. My previous experience with Unity means that I am already familiar with it, reducing the learning curve.

DATA STRUCTURES AND HLSL FRAMEWORK

In this chapter, an overview of the developed project will be presented. By gaining a complete understanding of the project, including how it was developed, why certain tools and decisions were made, and its advantages over other methods, further work can more easily be built upon it. This is a crucial aspect because the work presented in this document represents only part of the larger SABUVIS project.

To achieve this understanding, a case study will initially be presented. This will be followed by an explanation of the data structure used in the project to implement the fluid simulation, along with its rationale and benefits. This explanation is integral to understanding the project since the data structure serves as the foundation for the rest of the code.

The Unity workflow will then be discussed, providing an overview of how code is developed in Unity and how it was used in this project. HLSL, a language designed to run code on the GPU for faster computation, will close off this chapter.

3.1 Fluid Simulator: case study

This thesis is being developed as part of the broader SABUVIS project. SABUVIS is an European Defense Agency (EDA) initiative aimed at advancing the research and development of BUVs and equipping European countries with this type of vehicle. The project has led to the construction of multiple prototypes by various partners, some of which are shown in Figure 3.1.

To improve movement efficiency, a tool was envisioned in which the models would learn to optimize their motion through a reinforcement learning algorithm. A fundamental component of this tool is a fluid simulation. It is this fluid simulation that was developed as part of this thesis. As a result, some of the related work presented in Chapter 2, such as the sections on machine learning, is not directly applied in the current implementation. Nevertheless, this research remains valuable to ensure the final objective is kept in focus during simulation development. Additionally, future developers extending this work will benefit from the groundwork already laid.



(a) Polish prototype



(b) Seal-like UV designed by the Portuguese partners

Figure 3.1: Two different UV prototypes, part of the SABUVIS project

3.2 Data Structure

The data structure is an essential aspect of development because it can heavily affect performance. Since the aim of this project is to create a fluid simulation to aid in the development of UV, its performance is paramount.

This chapter will begin by outlining the main guiding criteria considered when deciding on the data structure. It will then present concrete examples of choices made during the development process, including the reasoning behind them, their benefits, and their drawbacks.

3.2.1 Guiding criteria

For this project, the main concerns when designing the data structure were: **Speed**, **Space**, **Intuitiveness/Code Clarity**, **Scalability**, and **Algorithms Used**. Here's why they play such a crucial role:

- **Speed** - This project will inevitably repeat the same operations multiple times, independent of the viewpoint chosen from Chapter 2.1.2. When using a Lagrangian viewpoint, each particle will calculate its new position, so the code will execute the same number of times as the number of particles in the system. In the Eulerian approach, the code will run once for each cell. Additionally, each cycle will ideally repeat many times per second. This means even a small delay in an individual operation will cascade and slow down the whole simulation. Therefore, when choosing the data structure, it is important to ensure that storing and accessing the variables is as fast as possible, which usually means using indexable data types.
- **Space** - Similar to speed, both implementation viewpoints will encounter the same issue concerning the space needed to store the information to accurately simulate the fluid. Assuming the same density of cells or particles, the space needed will grow with the square of the side length. It is therefore important to reduce redundancy

in the data and avoid storing unnecessary information. For this project, speed is valued higher than space, so some space may be sacrificed for a faster simulation.

- **Intuitiveness/Code Clarity** - Any project benefits from having intuitive and clear code, as it makes it easier to maintain, debug, and extend. An intuitive data structure is also valued for this project, as it serves as a base for future work.
- **Scalability** - Poor choices might lead to bottlenecks or inefficiencies. As the project grows in complexity, an appropriate data structure ensures the system remains scalable.
- **Algorithms Used** - The algorithms used must be taken into consideration, as their interaction with the data structure will determine which operations will be used most frequently and should therefore be faster. For example, when using a Eulerian approach, it is expected that the variables for each cell will vary, but it is unlikely that the number of cells will change. Thus, a data structure where changing values is fast, even if resizing is slow, is perfectly acceptable.

Although both Eulerian and Lagrangian viewpoints are referenced in these criteria, only the Eulerian will be discussed further, as it was the chosen implementation viewpoint, as mentioned in the previous chapter.

With these criteria in mind, the chosen data structures can be divided into three groups, which will be referred as: *simState*, **visualization**, and **debug**.

In this document, *simState* refers to the grouping of all variables needed to calculate the next step of the iteration. This includes the velocity of the fluid, the objects, and others. In the code, this is represented by the C# class named *SimulationState*. Its detailed structure can be seen in Figure 3.2. The **visualization** group encompasses system variables to control how the *simState* is visualized. For example, a boolean determines whether arrows representing the velocity at each cell are displayed, and another toggles the visibility of cell borders. Additionally, there is an enum to choose between different view modes. The last group, **debug**, includes all variables used for debugging.

In the next chapter, only *simState* will be analyzed, as both the visualization and debug groups contain data that is either rarely used or independent of the size of the simulation. Consequently, their effect on the simulation as a whole can be neglected.

3.2.2 *simState*

As referenced in the previous section, *simState* contains all the information needed to progress with the simulation. Figure 3.2 shows the properties. All properties are stored in two-dimensional arrays, and they are:

pressure: The pressure of the fluid in the cell, measured in *Pascal (Pa)*.

velocityH: Horizontal velocity, measured in ms^{-1} .

simState
pressure: float[.]
velocityH: float[.]
velocityV: float[.]
wall: float[.]
smoke: float[.]
objectsSeparated: int[.]
objectsMerged: float[.]

Figure 3.2: Structure of SimulationState C# class (each of these arrays have as many components as the number of cells in the simulation space, ordered by rows from bottom to top)

velocityV: Vertical velocity, measured in ms^{-1} .

wall: Functionally a bool that indicates whether the cell is a wall. 0 - wall | 1 - fluid.

smoke: The density of smoke in the cell. Smoke can be added to the simulation as it may be useful to visualize the movement of the fluid, similarly to it's use in wind tunnels. When the fluid is a liquid instead of a gas, such as the one developed for this project, the term dye might be more adequate.

objectsSeparated: The only integer property. Stores whether this cell belongs to a specific object. Objects differ from walls in that the forces applied to them can be measured. For example, revisiting the wind tunnel scenario, the forces applied on the walls of the tunnels are not calculated, but the effects on the foil or other objects present are calculated. Multiple objects can exist in a single simulation, each assigned an ID starting from 0. Each bit of the variable encodes whether the cell belongs to a specific object, using the ID of the object as an index. If the bit is 0 then it is part of the object, if it is 1 it isn't. With the least significant bit referring to object with index 0. (e.g., if the stored value is 110 in binary, this means the cell belongs to object 0).

objectsMerged: This merges the data stored in *wall* and *objectsSeparated* and is the property actually used in calculations, with 0 meaning it is part of at least one object or a wall and 1 meaning it is fluid. While this property could be calculated every time it is needed, it is instead stored. Having it always stored improves readability significantly, which was considered an acceptable trade-off.

Note that some of the variables could have been represented by booleans, *wall* and *objectsSeparated*, but instead floats were used. This is due to the fact that HLSL does not support buffers of type boolean. Hence the use of floats.

An important design choice was the use of a staggered grid, the benefits of which were presented in Chapter 2.1.2.1. In this project, the use of staggered grids improves the efficiency of the algorithms. Specifically, since the velocity between two cell centers

is needed several times per simulation step, it is better to store these values at the edges rather than interpolating them each time. This eliminates the need for interpolation. The staggered grid used is similar to the one originally shown in Figure 2.5, and shown again in Figure 3.3a.

As a reminder: P represents the pressure of the fluid in that cell. U_x and U_y represent the velocity of the fluid at the center of the edge, but only the horizontal or vertical components respectively. For example, to obtain the velocity vector at the center, one would need to interpolate U_x using the left and right edges, to determine the horizontal velocity and the U_y from the top and bottom edges to determine the vertical velocity.

Although the concept remains the same, the implementation for this project varies slightly, as shown in Figure 3.3b. The data stored at the center C of the cell is not only pressure, but also smoke density and whether the cell is part of an object or wall. The indexing is also a bit different, in Figure 3.3a the author uses $1/2$ offset to get to the edges from the center. Since arrays can't have non integers indexes, the index (i,j) refers to the cell center when measuring the pressure or smoke density, but when measuring velocity H it refers to the edge to the left, and when measuring velocity V it refers to the bottom edge, these can be seen in Figure 3.3b, represented by "C", "H" and "V" respectively.

It is also important to note that each property is stored in a separate two-dimensional array, rather than combining all properties in a class "Cell" and using a single two-dimensional array of that object. This choice was made for two main reasons, the most important being that when sending data to the GPU, only the necessary information is accessible for each function, with no overhead caused by unnecessary data. The second reason was for code clarity, as the use of a staggered grid means that there are essentially three different coordinate systems. For instance the index $(0,0)$ refers to the center of the bottom left cell when indexing pressure, while that same index is also used to refer to its left and bottom edge when indexing velocity H and velocity V respectively. So keeping each property in its own two-dimensional array improves clarity.

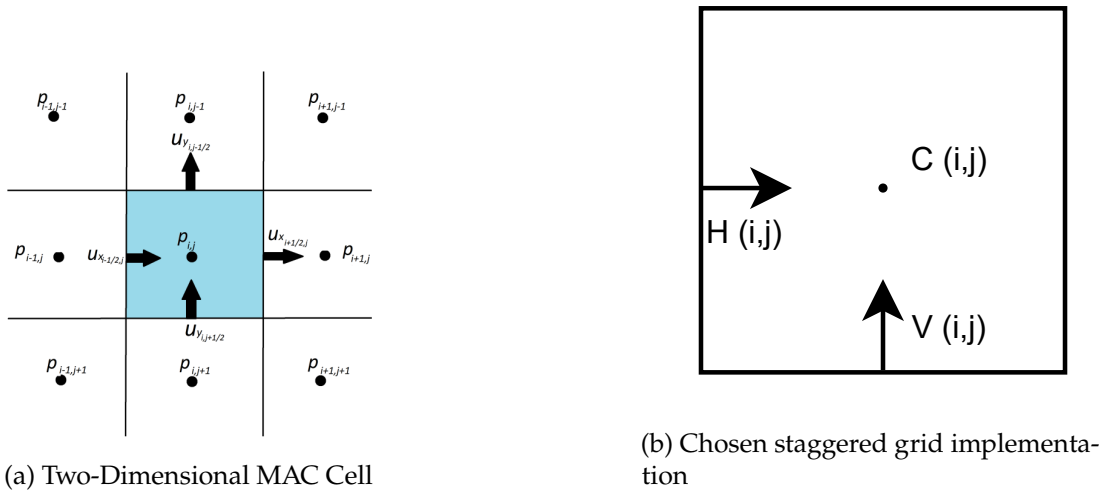


Figure 3.3: Two different staggered grid

There is a small hinderance when this is the chosen representation. Because position (i,j) always refers to the either the left or bottom edge, the right and top edges must be accessed by adding 1 to i or j respectively. But on the edges of the grid this is not possible as $i+1$ or $j+1$ can be out of bounds, thus all right most and top most edges are not represented in the data. Figure 3.4 shows what a 5x5 grid would look like if only the indexable edges were drawn.

Additionally, due to the need to send data to the GPU, a custom class was used named "Buf2<T>". This class stores both the CPU and GPU representations, using a two-dimensional array of type "T" for the CPU and a *ComputeBuffer* for the GPU, which is a flattened version of the data. This class makes it more intuitive to manage the two representations and to send the data back and forth to the GPU.

3.3 Unity Workflow

Unity or Unity3D¹ is the game engine chosen for the development of this project, as explained in Chapter 2.3.3. Game engines serve as frameworks for game developers, allowing them to quickly develop scenes without writing complex code. Unity was selected for its visualization environment, which is both intuitive and well-documented. This ease of use explains why, according to the Unity report from 2021, "Unity remains the game engine of choice with 61% of developers surveyed using it"².

Unity offers various pricing solutions depending on the intended use of its tools. For this project, a free license was used. However, this does not guarantee that the license will remain valid for any further expansion of the project.

Unity follows the component model, which provides a scalable programming architecture. The base entity in Unity is called a *GameObject*, which acts as a "container" for

¹<https://unity.com/> (accessed in July 2025)

²<https://create.unity.com/2021-game-report> (accessed in July 2025)

(0,4)	(1,4)	(2,4)	(3,4)	(4,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)

(a) Two-Dimensional indexing

20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

(b) Linear indexing

Figure 3.4: Representaion of the indexable edges of a 5x5 grid

components. These components can vary, including audio, animation, physics, scripts, and more. Scenes are used to group GameObjects logically. For example, in a traditional game, each level would be a separate Unity Scene. Typically, only one Scene is loaded at a time.

Each new Scene includes two default GameObjects: a Camera and a Light. These are essential, as without a camera nothing would be visible to the user during runtime, and without a light, all objects would appear entirely black. It is important to note that while Scenes can be viewed and manipulated in a 2D interface, they are always stored as 3D environments.

Scripts are highly valuable components, as they allow developers to add custom behaviors to GameObjects. Unity scripts are written primarily in C#, as Unity "removed boo and unityscript dlls from mono distribution"³ in 2020. Since then, C# has been the only native scripting language supported by Unity.

In this chapter, a deeper analysis of the C# language will be presented to ensure that scripts are written efficiently and follow best practices. This will be followed by a section on the process and considerations involved in generating a the texture from a Simulation State. The chapter will conclude by discussing the different types of data that can go through this transformation and be displayed to the user.

3.3.1 Event driven architecture

Unity follows an event-driven architecture. Events are a fundamental concept in Unity projects and are provided through a wide range of APIs. Leveraging the object-oriented nature of C#, any script that derives from the *MonoBehaviour* class can implement custom behavior based on these events using event functions. Event functions can be categorized into four main groups⁴: Regular Frame Update, Object Lifecycle, UI, and Physics. These groups are presented in the following sections, along with explanations of how specific events were used in this project.

3.3.2 Regular Frame Update

This group refers to events typically used to modify the properties of *GameObjects*, such as position, state, and velocity. Most of these events are invoked once per frame. It includes the most commonly used functions:

- *Update()*: Called once every frame, this is arguably the most frequently used function in *MonoBehaviour*. In this project, the majority of the code is executed within this event. It begins by updating the *simState*, followed by rendering the simulation.

³<https://unity.com/releases/editor/beta/2020.2.0a7> (accessed in March 2025)

⁴<https://docs.unity3d.com/Manual/event-functions.html>

- *FixedUpdate()*: This function is frame-rate independent, making it useful for physics calculations. Performing physics updates in *Update()* can lead to inconsistent behavior depending on the frame rate. For example, in a car simulator, if pressing forward increases the car's speed by 0.1 units per frame, the time required to reach a certain speed would vary between 240 FPS and 60 FPS. Since *FixedUpdate()* is executed a fixed number of times per second, it ensures consistent acceleration regardless of frame rate. This function is not used in the current project, as the simulation is designed to run as fast as possible. However, it may be worth considering in future implementations where a constant frame rate is desired. In such cases, simply moving the code from *Update()* to *FixedUpdate()* would suffice.

3.3.3 Object Lifecycle

These events are not called periodically but are instead triggered by specific stages in an object's lifecycle, such as its creation or destruction. Relevant functions include:

- *Start()*: Called once per script at the beginning of its lifecycle. This function is used to initialize most values, such as the *simState*, the rendering plane for the simulation, and the class responsible for handling it. The first simulation step is also calculated here.
- *OnDestroy()*: Invoked just before an object is destroyed. Since *ComputeBuffers* are not managed by C#'s garbage collector, memory must be cleared manually. *OnDestroy()* is used to release the memory associated with *ComputeBuffers* before the script is destroyed.

3.3.4 UI

UI events are handled separately from regular updates. Any changes to the UI should be performed within these functions:

- *OnGUI()*: Contains most of the code for handling GUI events. Since the Unity Inspector is used to manage simulation customization in this project, *OnGUI()* is not utilized.

3.3.5 Physics

Unity objects can have colliders, and this group of event functions manages behavior resulting from collider interactions. Functions include:

- *OnCollisionEnter()*: Called when this object's collider enters another object's collider. While this could be useful in a particle-based simulation to detect collisions between particles, it is not applicable to the grid-based simulation implemented in this project.

Only a small subset of functions have been presented here. For a complete list of the event functions and their behaviour refer to Unity documentation ⁵. Unity calls these functions according to a predetermined order which can be seen in the appendix Figure A.1.

3.3.6 Generating Pixels from the Simulation State

Since the Simulation State is stored in memory, a method is required to visualize the evolution of the simulation for the user. In Unity, this can be achieved through several approaches. The most basic method is to create a grid of GameObjects, with one GameObject per cell. At the end of each simulation step, the color of each cell is calculated and assigned as the texture color for the corresponding GameObject. This process is illustrated in Figure 3.5. As expected, the GameObjects appear in the left column, while a pixelated grid can be seen in the "Scene" window. By selecting the first GameObject, it appears highlighted in the scene and represents a "pixel," in this case, the bottom-left pixel.

This approach, however, has several downsides. First, since every GameObject is assigned a new color in each simulation step, and this operation cannot be performed in a GPU program, a "for loop" is required to assign the new colors individually. Moreover, if the calculations are performed on the GPU, the values must be transferred back to the CPU, negating the performance gains from GPU computation. This method is not scalable and, even at low resolutions, takes considerable time to execute. Another issue is the memory and CPU cost. Increasing the number of GameObjects profoundly impacts performance, even before reaching any software constraints on the number of GameObjects. This makes the method unfeasible for practical use.

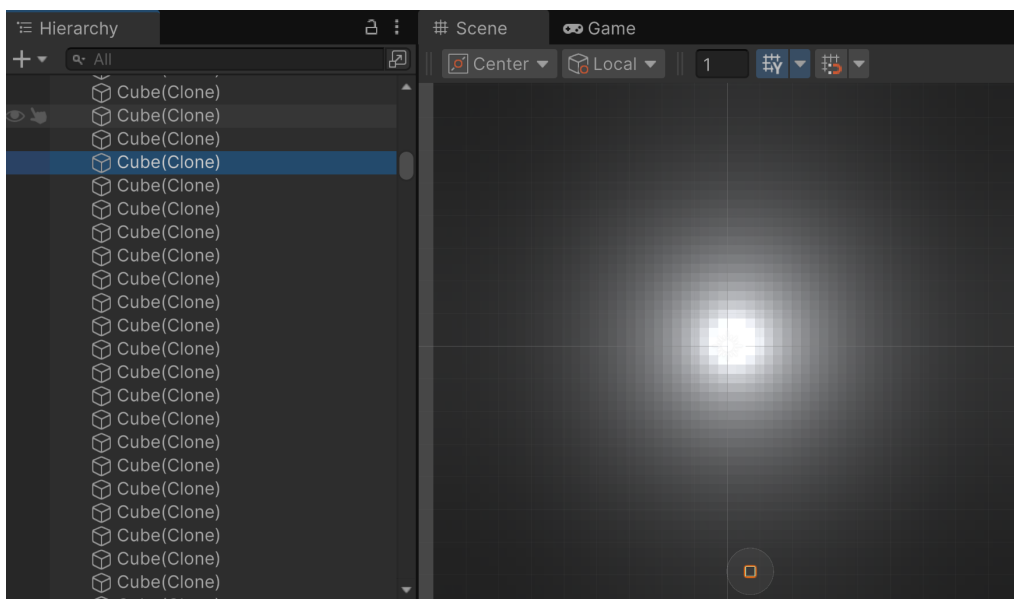


Figure 3.5: Using one GameObject per cell

⁵<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

To solve this problem, the solution must avoid relying on an increasing number of GameObjects as the resolution increases. Ideally, the solution should also run entirely on the GPU to eliminate the need for a "for loop" and prevent data transfers between the CPU and GPU during every simulation step. The method chosen involves using a single texture. Textures can be assigned to GameObjects at the start of the simulation and manipulated directly on the GPU, addressing both of the previous drawbacks.

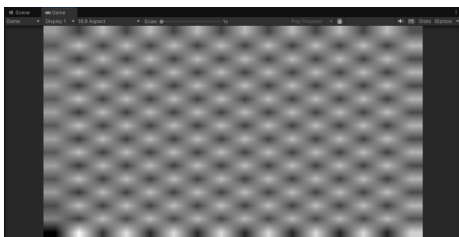
Two approaches were analyzed for using a texture: modifying the camera output or setting the texture on a plane in the 3D world. To achieve the first option, the Unity method "OnRenderImage()" ⁶ can be called on a Camera GameObject. This function allows the final image shown to the user to be replaced with a custom texture, resulting in what can be seen in Figure 3.6a, where a texture with a checkered pattern is displayed. Figure 3.6b demonstrates the second option, which involves applying the texture to a plane in the 3D world. This approach can be implemented by executing the following line, using a similar texture:

```
plane.GetComponent<Renderer>().material.mainTexture = tex;
```

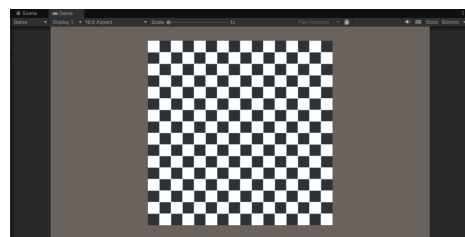
Note that two different textures are being used, which explains the fuzziness in Figure (a). Although the state they are representing is the same, a checkered pattern with 16 by 16 cells, the texture in Figure (a) is 16 by 16, with one cell in the state corresponding to exactly one pixel in the texture. In contrast, the texture in Figure (b) is 160 by 160, with each cell in the pattern corresponding to a section of 10 by 10 pixels in the texture. Since Unity interpolates the values from the texture to match the resolution of the viewing area, this interpolation creates the observed effect. A pixelated image is useful during development as it allows for a clearer understanding of the data being displayed, whereas having one pixel per cell results in a more natural image.

Since the two methods are effectively the same in terms of performance (camera and plane), the choice was made based on replication and resizable. In this context, resizability refers to the ability to adapt the texture to accommodate non-square dimensions. For example, in a wind tunnel, one axis is expected to be significantly larger than the

⁶OnRenderImage documentation: <https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.OnRenderImage.html>



(a) Replacing the camera output with the custom texture (Using one pixel per cell)



(b) Setting custom texture to plane (Using 10 pixels per cell)

Figure 3.6: Two uses of texture, showing a checkered pattern

other. In the camera approach, this can be achieved by correctly setting the aspect ratio of the camera. Figure 3.6a illustrates what happens if the aspect ratio is not adapted: the texture displays a 16x16 grid using an aspect ratio of 16:9, resulting in "squished" squares. Using the plane approach, this issue can be resolved by resizing the plane to match the aspect ratio of the texture. Both approaches are implementable, but changing the camera's aspect ratio produces less consistent results since the "window" size is constantly changing, whereas the plane approach provides a more stable and consistent user experience.

The plane approach is also superior in terms of replication, as it is much easier to add additional planes to the environment than to attempt to display multiple simulations using only the single texture outputted by the camera. This is a significant consideration, as the future use of this tool will involve leveraging the simulation to train a reinforcement learning algorithm. For this training, running multiple simulations concurrently is often faster than running them separately.

Considering the presented arguments, using a plane to display the texture was chosen as the preferred approach.

3.3.7 Visualization Types

With the display method settled, the next question is which data should be shown and what is the most intuitive way to present it. This section will delve into this topic, presenting the proposed visualization types, which can be seen in Figure 3.7, and which are pressure, smoke, type, velocity and objects. Additionally the last two images in the figure show how some colors were used. All figures represent the exact same simulation state which was captured when simulating a water tunnel, with fluid going from left to right, containing two circular objects.

Beyond this five visualization modes there are also two others that can be used as overlays, this are: arrows to show the speed and borders to more easily distinguish between cells.

3.3.7.1 Pressure

Figure 3.7a shows how pressure varies throughout the simulation space. In this mode, the color gradient used is illustrated in Figure 3.7f, where the most negative pressure is represented by blue and the most positive pressure by red. Visually, the result meets expectations, with both circular objects displaying a region of high pressure upstream and a low-pressure zone downstream.

In the current implementation of this viewing mode, the gradient's maximum and minimum values are set to constant values of 200 and -200 Pascal, respectively. This approach was chosen over using a dynamic range based on the maximum and minimum pressures from the current simulation step because a dynamic range would require comparing all the values during each simulation step to identify the highest and lowest pressures. This process would reduce the framerate.

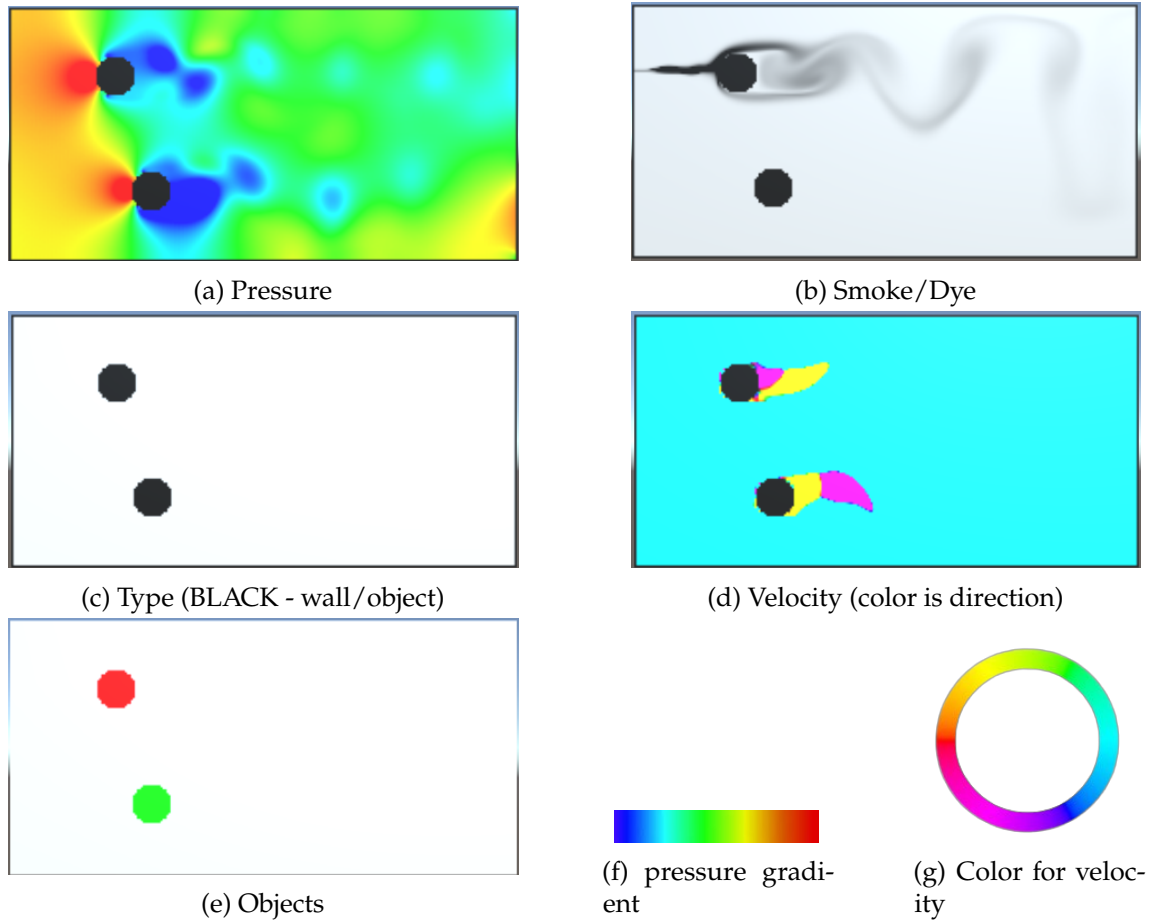


Figure 3.7: Five different types of visualization and color guides

Moreover, positive or negative pressures of more than 200 Pa would destroy some of the components of the BUV and will never be reached in a realistic situation.

3.3.7.2 Smoke/dye

In Figure 3.7b, smoke or dye is represented. In this case, prior to the snapshot, smoke was continuously added upstream of the top object, resulting in the expected visualization of turbulent flow near and downstream of the object. In this mode, the color gradient ranges from black to white, with the darkest pixels corresponding to the highest concentration of smoke or dye.

3.3.7.3 Types/Objects

There are two ways of displaying the objects and walls, as shown in Figures 3.7c and 3.7e. The "Type" mode is simpler, showing all walls and objects in black, while the fluid is displayed in white. The "Objects" mode differentiates the objects by assigning each a unique color, while the walls are shown in white along with the fluid. This distinction is important when objects are positioned very close to each other and might otherwise

appear as a single object.

3.3.7.4 Velocity

Finally, the last figure, 3.7d, shows the velocity at each cell center. Since velocity is a two-dimensional vector, a simple color gradient is insufficient to represent it. Therefore, a color wheel was used, where the color of each pixel indicates the direction of the fluid. The color wheel used is displayed in Figure 3.7g. In this case, most of the fluid velocity is represented with a cyan color. By referring to the color wheel, cyan is located directly to the right of the center, indicating that the fluid is moving from left to right. The yellow color appears slightly above and to the left of the center, meaning the fluid in the yellow regions is moving upward and slightly to the left. The magnitude of the velocity also impacts the color. The lower the velocity the darker it is, with zero velocity represented by black.

3.3.7.5 Overlays

"Overlays," as defined in this document, are visualization modes that can be used in conjunction with other modes. Two overlays were implemented: the arrows overlay and the borders overlay, as seen in Figure 3.8. Both serve a similar purpose, to make it easier to understand what is happening in the simulation, which is extremely useful for debugging.

Figure 3.8 shows a fabricated simulation state where velocities were added manually. The visualization being used is for pressure, which is displayed as green, the expected color since no simulation step has yet occurred, and green corresponds to 0 Pascal of pressure. To draw the borders and arrows, cells cannot be represented by a single pixel, as there would be no pixel available to color the space between two cells. The implemented solution addresses this by using a 100x100 grid of pixels to represent each cell. While this might seem a high number, it is necessary to accurately draw arrows.

There are two problems caused by the high number of pixels used when each cell is represented by 100x100 pixels. First, "Unity only supports textures up to a size of 16384,"⁷ so when drawing the overlays, no side can have more than 163 cells. However, this problem is less relevant due to the second issue. Similar to the interpolation Unity performs when rendering lower-resolution textures on higher-resolution screens (shown in Figure 3.6a), Unity must do the opposite when the texture resolution exceeds the screen resolution and group several pixels into one. As a result, the borders and arrows may appear distorted or even disappear entirely. Therefore, there is no practical reason to use overlays when simulating high-resolution environments.

The borders overlay draws a blue border between cells, with a thickness of 10 pixels. This value can be modified in the code. The border helps to more clearly distinguish between cells. During development, this was critical. For example, to test the display

⁷<https://docs.unity3d.com/6000.0/Documentation/ScriptReference/SystemInfo-maxTextureSize.html>

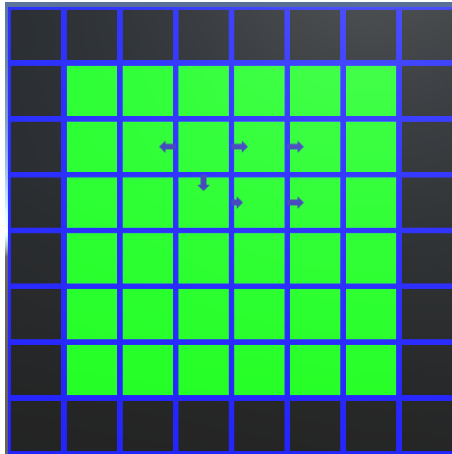


Figure 3.8: The two possible overlays with an example state

of velocities, a function was implemented that allowed the user to click a cell, adding velocity to that cell in a specified direction. Without borders, it would have been difficult to determine where each cell starts and ends.

The arrows overlay draws vertical and horizontal arrows at the edges of cells, with their size proportional to the velocity of the fluid passing through the edge. When testing projection and advection, this overlay was invaluable as it allowed for a quicker and more intuitive understanding of what was happening in the simulation. The use of 100x100 pixels per cell is essential when drawing arrows, as the arrows are rendered pixel by pixel. Instead of developing a function that could "paint" the pixels for an arbitrary size, a compromise was made for simplicity. So the function that draws the arrows assumes a resolution of 100x100 pixels per cell.

3.4 HLSL based systems

High-Level Shader Language (HLSL) is a powerful programming language designed for writing shaders that run on graphics processing units (GPUs). It is probably more well known for its use in writing shaders for the rendering pipeline, such as vertex shaders or pixel shaders. The type of shader used in this project is instead a compute shader. Compute shaders harness the parallel computing capabilities of GPUs, outside the normal rendering pipeline, and are therefore particularly utilized in physical simulations, such as the one in this project.

This chapter will explore the architecture and functionality of the HLSL systems and its use within the project. It begins with an overview of the HLSL language and its integration with Unity's workflow. Following this, a section delves into how a speed up through the use of HLSL can be achieved. Finalizing with a practical comparison between its use versus more common programming languages, and how different implementation also affect the performance.

3.4.1 What is HLSL

"HLSL is the C-like high-level shader language that you use with programmable shaders in DirectX." ⁸ Shaders are programs designed to run on GPUs to perform various tasks in the rendering pipeline, such as manipulating vertices, applying textures, or simulating lighting effects. Through HLSL, developers can harness the GPU's parallel computing power for applications such as real-time graphics, game development, and physics simulations.

HLSL was developed by Microsoft as part of DirectX 9 ⁸ with the goal of providing a more developer-friendly approach to GPU programming, similar to other languages such as OpenGL Shading Language (GLSL).

Unity can be used to develop applications for multiple platforms, and as such, its shaders must also be compatible with these platforms. For platforms that use DirectX, such as Windows, a direct compilation is performed. However, for other platforms, this compilation is followed by a bytecode translation into the respective shader language.

There are six different types of shaders that can be programmed using HLSL: pixel shaders, vertex shaders, geometry shaders, compute shaders, tessellation shaders, and ray tracing shaders. The usefulness of shaders is particularly evident when tackling massively parallelizable problems, such as those encountered in image processing or matrix calculations, and other similar problems. This is achieved through the Single Instruction, Multiple Data (SIMD) technique used by GPUs. SIMD ensures that the same instruction is performed simultaneously on multiple data elements, for example, when increasing the brightness of an image, the same operation is applied to all pixels.

3.4.2 GPU speed up

GPU's achieve their speed up over other types of processing units through a variety of different methods. Most methods revolve around the parallelization of problems. Which is reflected in its hardware. For example, opposite to CPU's relatively low number of cores, usually between 2 and 8 physical cores, GPU's have a much higher core count, which can reach over 20,000 in high-end modern commercially available units, such as the NVIDIA GeForce RTX 5090⁹. The high core count allows for more instructions to be run at the same time, through parallelization. CPU's don't adopt this as less cores allow for more complexity per core and therefore versatility. For example the CPU must be able to communicate with more components such as the RAM and I/O ports, while the GPU doesn't.

Since the GPU speed up is mostly attributed to its parallelization, massively parallelizable problems benefit the most from the speed up achieved by the GPU. To tackle the massively parallel problems, the GPU uses, among other techniques, SIMD instructions.

⁸Microsoft HLSL page: <https://learn.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hlsl> (accessed on March 2025)

⁹<https://www.nvidia.com/en-eu/geforce/graphics-cards/50-series/rtx-5090/>

By using a single instruction on multiple data, there is an improvement on processing speed because data is processed simultaneously instead of one at a time.

For a more comprehensive understanding of GPUs, William J. Dally [18], overviews the evolution of the GPU, which enabled its use in projects such as the one presented in this thesis.

3.4.3 Programming for GPUs

Given the architectural specificities of both GPUs and HLSL, several important considerations must be taken into account when writing HLSL code. These considerations not only influence the correctness of the implementation but also have a significant impact on performance optimization.

GPUs are designed with a large number of simpler cores compared to CPUs. One notable limitation of GPU cores is the absence of branch prediction, a feature commonly found in CPUs. Branch prediction enables CPUs to anticipate the flow of control in conditional statements, thereby improving execution efficiency. However, in GPUs, this mechanism would disrupt the parallel execution model, where threads operate in lockstep within a warp or wavefront. As a result, frequent use of control flow statements such as `if`, `switch`, or loops (`for`, `while`) can lead to thread divergence, reducing parallel efficiency. Therefore, minimizing or restructuring conditional logic is crucial when targeting GPU execution.

Another key consideration is the division of labor between the CPU and GPU. Since data transfer is relatively slow and can become a bottleneck. Developers should carefully decide which computations are best suited for the GPU and which should remain on the CPU. Ideally, large, parallelizable workloads should be offloaded to the GPU, while tasks involving complex branching or low-latency requirements may be better handled by the CPU. Efficient memory management and minimizing data movement between host and device are essential for achieving optimal performance.

FLUID SIMULATION ALGORITHM

This chapter delves into the developed code. As the tool is not yet in its finished state, anyone aiming to extend it will greatly benefit from this section, since a thorough understanding of the project's structure, design choices, and their underlying rationale is essential.

The chapter begins by introducing the main components, followed by an overview of the architecture to illustrate how these components interact. Next, the programming style is outlined to ensure consistency across future extensions. This is followed by a step-by-step explanation of the simulation process using the chosen algorithm. The chapter concludes with an extensive list of decisions made during development, along with the reasoning behind them.

4.1 Main Components

The components of this project can be separated into three categories that each execute once per frame: simulation state update, rendering, and GUI handling, as shown in Figure 4.1. The most important of these is the update of the simulation state.

The update of the simulation state is handled entirely by shaders, leveraging their speed since this is the most computationally intensive part of the project. The shaders are invoked by the *MainSimulationRunner.cs* script. Following the general method outlined in Section 2.1.2.1, this component begins by solving incompressibility often referred to as **projection** to ensure the fluid remains incompressible. Once incompressibility is established, properties are **advected**, with velocity being the most critical. Other properties, such as smoke, are advected in this project.

The GUI component processes user interactions during the simulation, including pausing/resuming, adding objects, and changing the visualization mode. Clicks are detected by the *PlaneClicker.cs* script, while changes made through the Unity Inspector (sidebar) are tracked by Unity's internal logic. Both types of input are handled accordingly within the *MainSimulationRunner.cs* script.

The rendering component is responsible for displaying the correct information to the

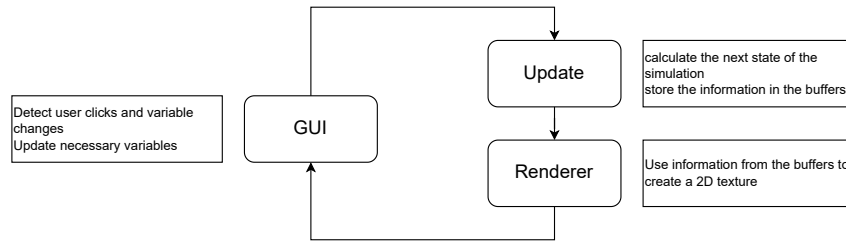


Figure 4.1: Main component architecture

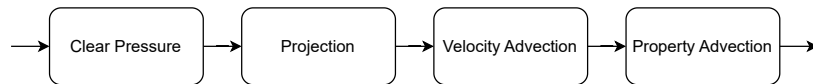


Figure 4.2: Update subdivision

user, depending on the selected visualization mode. For example, if pressure is selected, the simulation plane should be colored accordingly. This component is managed by the *RenderHandler.cs* script, which invokes the appropriate shaders based on user input. There are three steps required to convert the simulation state into the image shown to the user, each executed by a shader:

1. **Property Mapping:** The simulation state is converted into a 2D texture, where each pixel corresponds to a cell's properties. These shaders are referred to as *propertyShaders*, with examples including *PressureShader* and *ObjectShader*.
2. **Upscaling:** A single shader handles the scaling of the texture, since the property mapping step produces a texture matching the dimensions of the simulation state which may not be ideal for visualization.
3. **Post-Processing Effects:** This final step adds visual enhancements such as blue borders between cells and velocity arrows representing fluid flow between neighboring cells.

This section does not examine the code in detail, to avoid excessive information. For those interested, the full codebase is available on my GitHub¹.

4.2 Architecture

With the components clearly established, the architecture overview will highlight the flow of data between the update, rendering, and GUI components.

One characteristic that may seem counterintuitive is that while the rendering component is guaranteed to execute immediately after the simulation state update, the execution order of the GUI component is not enforced by the code. As a result, it may run either

¹https://github.com/lucascorreialobo/Unity_Project_Tese/tree/main/Project

just before the simulation update or just after rendering. However, this order remains consistent throughout the rest of the simulation. Although Unity does not guarantee script execution order, once an initial sequence is established, it remains consistent across frames. Using Figure 4.1 as reference, the start and end of the simulation cycle can be interpreted at either the top or bottom of the diagram.

An execution order can be manually enforced using Unity's script execution settings², but doing so introduces overhead. Since the relative order between the GUI and the other components is not relevant for correct functionality, it was left unmodified.

Regardless of the execution sequence, the flow of data remains consistent. The GUI directly calls functions from the *MainSimulationRunner.cs* script, and it is within these functions that internal variables are modified. Most of the functions triggered by the GUI component apply changes to or read from the simulation state, just as the other components do. This does not lead to any race conditions, as all components are executed synchronously, one after another.

4.3 Programming style

This section outlines some of the general rules that were followed when writing the code. These guidelines are important both for those reading the code, so it can be more easily understood, and for anyone intending to extend it, to ensure a cohesive style throughout the project.

The simulation state is managed using the custom `Buf2<T>` class. This class holds both a CPU and a GPU representation of the simulation state, simplifying the use of both and allowing for the translation between them. It's important to note that the class does not guarantee the data is synchronized across both representations. Synchronization only occurs after an explicit translation between CPU and GPU states, which is computationally expensive and therefore not performed routinely. For most of the time, the data remains in its GPU form, since both updating and rendering are designed to operate on this representation. One example of when translation occurs is during GUI events, when a user manually alters a value in the simulation state. In such cases, the data is translated to CPU, modified, and then converted back to GPU. This is acceptable, as these actions are infrequent.

When managing the simulation state, each cell's type is represented by a number: if the value is "0", it indicates no fluid (the cell belongs to an object); if the value is "1", it denotes a fluid cell. This simple representation was originally chosen because it facilitates calculations involving neighboring fluid cells during the projection step of the simulation.

In early development, a single shader contained all functions responsible for updating the simulation state. However, the file quickly became quite large, so it was decided

²<https://docs.unity3d.com/6000.1/Documentation/Manual/class-MonoManager.html>

that each shader should serve a single, specific purpose. This change does not impact performance but greatly improves code readability.

To simplify access to neighboring cells within a shader, the *positions.cginc* file is used. By including it via `#include positions.cginc`, its code is integrated into the shader. This file provides macros: LEFT, RIGHT, UP, and DOWN, which return the index of the neighboring cell in the specified direction, based on the original cell's index.

Throughout the code the units used were the SI, so time is measured in s, pressure is measured in Pa, velocity in m/s.

4.4 Simulation Steps

This section will be an opportunity to explain using a high level concepts the algorithms used in the simulation.

4.4.1 Clear Pressure

Pressure is actually not used to calculate the next simulation state, and is currently only used to calculate the force applied on an object by the fluid. The pressure is calculated in the Projection step and to be correctly calculated all values must be reset to 0. There is no operation on *ComputeBuffers* that sets all values to 0, besides creating one from scratch, which is very time consuming. So to achieve the reset a shader is used to simply set every value to 0.

4.4.2 Projection

The projection step guarantees the fluid is behaving incompressibly, meaning that in every cell the amount of fluid entering is the same amount exiting. For an individual cell this is done by averaging the fluid leaving the cell. The average is then subtracted to every edge, making the cell incompressible.

In the example shown in Figure 4.3, the edges (B)ottom, (L)eft, (T)op and (R)ight have values 2, 1, 4, 3, respectively. The total amount of fluid leaving is $-2 + (-1) + 4 + 3 = 4$. The average per edge would then be $4/4 = 1$. This tells us there is on average 1 unit of fluid leaving the cell per edge. So to make the cell incompressible, each edge should have 1 additional unit entering this results in $(B)-2+1 = 3$ $(L)-1+1 = 2$ $(T)4-1 = 3$ $(R)3-1 = 2$. The affects can be seen in Figure 4.3b.

Note that although all edges have positive values, the edges L and B positive value means there is fluid entering the cell, but a positive value in edges T and R, means there is fluid leaving the cell. The calculations must have this in consideration.

Although this algorithm makes a single cell incompressible, neighbouring cells are affected. To ensure global incompressibility, the algorithm is repeated multiple times to allow cells to converge. In the code, the algorithm is applied one hundred times to the



(a) Velocities before projection

(b) Velocities after projection

Figure 4.3: Effect of projection on the velocities around a cell

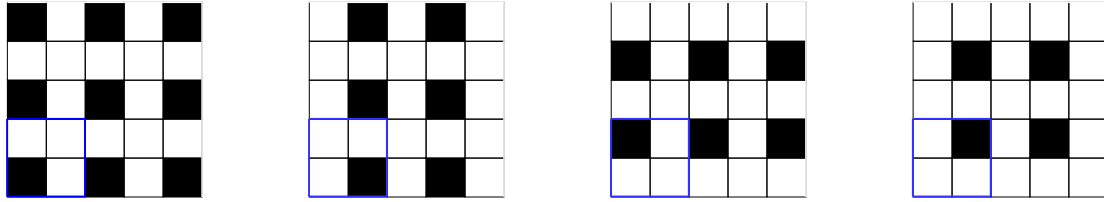


Figure 4.4: The four sets of cells on a 5x5 grid

entire grid, with each pass executed only after the previous one completes processing the entire grid.

Parallelism greatly reduces computation time, since a large number of cells follow the same algorithm. However, it introduces race conditions, as neighboring cells may try to update the same edge simultaneously, leading to unpredictable behavior. To avoid such conflicts, cells are separated into four sets. These sets are derived from any 2x2 square, take the blue square in Figure 4.4 as an example, each of the four cells in the blue square belongs to a different set. Starting from any of these cells, the rest of its set is composed of cells reachable using stride-2 movements. In Figure 4.4, each image represents a set, where black cells indicate set membership.

Within a set no two cells share an edge, whence each set can execute projection in parallel, while sets execute projection in sequence between themselves. This avoids any race condition.

4.4.3 Advection

Advection is the process by which a substance or property is transported through the motion of the fluid. In this project, the properties being advected are smoke and velocity. Other examples of properties that can be advected include temperature, salinity, and

nutrients.

Regardless of the property being advected, the algorithm used remains the same and consists of the following steps:

1. Find the velocity at the point where the property is measured.
2. Backtrack the velocity vector.
3. Interpolate the property at the resulting point (this becomes the new value at the original location).

The use of a staggered grid was helpful during the projection step but introduces complications in the advection algorithm. One issue arises immediately in the first step: since velocities are stored on cell edges, the velocity at the center of a cell must be computed by averaging its four edge values, top and bottom edges for the vertical component, and left and right for the horizontal component. This becomes more complex when velocity itself is being advected. In that case, the two-dimensional velocity should be measured at the edge center instead of cell center. Which means one axis of the velocity is already known at the measurement point, while the other must be averaged from four surrounding edges of the opposite orientation. As shown in Figure 4.5, to determine the velocity at point V, the vertical component is given, but the horizontal component must be averaged from the four dashed arrows around it. Conversely, at point H, the horizontal component is given, and the vertical component must be averaged from the four solid arrows surrounding it. In Figure 4.5 the three points V, H and C, are the points at which the velocity should be measured for the advection of vertical velocity, horizontal velocity and other properties, respectively.

In the second step, the velocity vector is inverted. This mimics the suction effect caused by the fluid's motion in the calculated direction, allowing the tip of the backtracked vector to occupy the original point. In Figure 4.5 these vectors are represented by a dotted line with a circle as the tip.

Again, the staggered grid complicates the third step, since the interpolation depends on which property is being calculated. Figure 4.6 illustrates the three possible reference configurations for interpolating a property at point P. If the vertical velocity is being interpolated, the hollow triangle serve as reference points. For horizontal velocity, the white dots should be used. Properties stored at the cell center, such as smoke, require interpolation using the black dots.

Since the advection step must be based solely on the previous simulation state, the new values for velocity and other properties are stored in a temporary buffer. These values are only copied into the simulation state after advection has been completed for the entire grid.

It is important to note that velocity advection should be performed before any other property advection to ensure that all properties are advected using the most up-to-date velocity values.

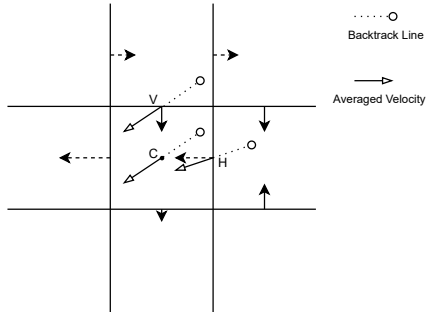


Figure 4.5: Velocity on the different points and its backtracking

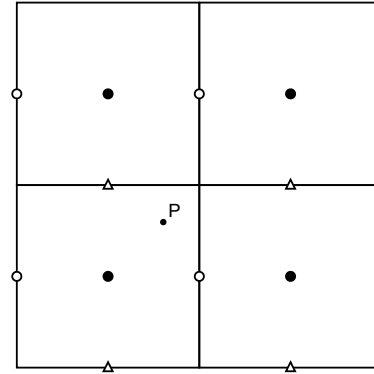


Figure 4.6: Different interpolation sources for different properties

4.5 Implementation Optimizations

Throughout the development of this project, several decisions had to be taken when it comes to implementation, either how data is stored, accessed, how to organize the code, among others. This section will explore these decisions along with the reasoning behind them so that anyone reading and possibly extending the code can have a clear understanding of why the code is written as it is.

The choices made are presented in no specific order.

4.5.1 Simulation state data structure

At the start of the project's implementation, one of the first decisions to be made was how the data would be structured. Several ideas were considered:

One option was to create a custom class or struct to represent a single cell, with the simulation state consisting of a collection of cells. However, this approach was discarded due to potential confusion, it would imply that velocity is a property of cells, whereas in reality, it belongs to the edges.

Another option was to use separate two-dimensional arrays for each property. A possible benefit of this approach was that each array could have a different size, which makes sense given that the number of edges differs from the number of cells. However, this advantage quickly diminished, as the data is flattened in the GPU representation. Consequently, indexes had to be constantly translated across the various arrays, eliminating any potential performance gain.

Ultimately, the decision was to keep the separate two-dimensional arrays for each property, but all having the same dimensions. This eliminates the need for index translation. By using separate two-dimensional arrays for each property, only the necessary data is exposed to the shaders, minimizing potential confusion.

Keep in mind that this approach assumes that a cell's index corresponds to its bottom and left edges when applied to the relevant arrays.

4.5.2 Shader partition

For the first prototype of the project, only a single shader was developed to handle the simulation update. This meant that pressure clearing, projection, and advection logic were all combined within the same shader. This approach led to several issues, mainly poor readability and conflicting variable names. From that point onward, shaders were kept as compact as possible, with each one performing only a minimal set of functions. This resulted in significantly improved code readability.

4.5.3 Where is simulation state updated

Unity has two key methods that run continuously during gameplay: *FixedUpdate* and *Update*. Both are called repeatedly until gameplay is stopped, allowing for continuous updates to variables. However, their usage differs slightly.

- *Update* runs once per frame, meaning the time between executions depends on the game's frame rate. It is best used for detecting user input, handling non-physics animations, and updating the user interface (UI).
- *FixedUpdate*, on the other hand, runs at a fixed time interval regardless of frame rate. This interval can be manually set by the developer before gameplay begins and remains constant throughout the session. As a result, it is ideal for physics-related actions, such as applying forces.

Although a simulation update aligns more closely with the *FixedUpdate* use case, it was implemented using *Update*. However, this setup causes fluids in high frame rate simulations to appear to move faster than those in low frame rate simulations, even if the recorded velocity is the same. Since the evolution from one state to the next remains consistent due to the hardcoded time step in the code.

This inconsistency is acceptable because the simulation's ultimate purpose is to be used in the training of a reinforcement learning algorithm. In such cases, updating the simulation as frequently as possible results in faster learning, despite possible visual inconsistencies.

4.5.4 Maximum and minimum in GPU

Throughout the code, it is often necessary to determine the maximum or minimum value of a given property currently present on the grid. Ideally, this should be done within shaders to avoid transferring data between the CPU and GPU. However, since GPUs are designed for massively parallel execution, finding a maximum or minimum value typically requires atomic operations and shared memory, resulting in longer execution

times that can be impractical. For this reason, calculating maximums and minimums is generally avoided unless absolutely necessary.

One example of this appears in the *pressureShader*. To color each pixel, a gradient is used, red representing the highest pressure, and blue the lowest. Accurate coloring requires knowledge of the maximum and minimum pressure values, but computing these directly in the shader negatively impacts performance. As a workaround, hardcoded values were used: 200 as the maximum and -200 as the minimum. This also creates a beneficial side effect: since the scale is fixed, different simulations can be compared visually with ease, as the same gradient color always represents the same pressure value.

4.5.5 General projection improvements

Projection is perhaps the function that would benefit the most from any performance improvement, as its code is repeated one hundred times.

One simple optimization is to multiply the divergence by an over-relaxation constant, which should be between 1 and 2. This ensures that the convergence behavior of the projection remains unchanged. The optimal value of the over-relaxation constant varies and should be tested by the developer. For this project, a value of 1.5 was used.

At the end of a cell's projection step, the velocities of its four edges are updated, but only if the neighboring cell sharing that edge also contains fluid. Translating this logic directly into shader code may introduce "if" statements, resulting in branching. To avoid this, the buffer *type* uses a value of 0 if the cell contains fluid, and 1 if it does not. By multiplying the *divergence* by the *type* of the neighboring cell, the same behavior is achieved without branching. For example, updating the left edge can be written as:

```
velocityH[simXY_1D] += type[LEFT(simXY_1D)] * avgDivergence;
```

4.5.6 Early stop of projection

The projection is currently executed one hundred times to ensure convergence. However, if the maximum divergence across the grid is already small, further iterations are unnecessary. Initially, it may seem difficult to implement this efficiently, because computing the maximum divergence after each pass is impractical due to the GPU's parallel architecture, as explained in the previous section.

The solution was to use a flag initialized to 0. During projection, if a cell has divergence above the threshold, it sets the flag to 1. After each full pass of the grid, the flag is checked. If it equals 1, the process continues to the next pass. If the flag remains 0, no cells exceeded the threshold, and further passes are skipped, even if the one hundred iterations haven't yet been completed. After each pass, the flag is reset to 0. A flowchart of this shader program is shown in Figure 4.7.

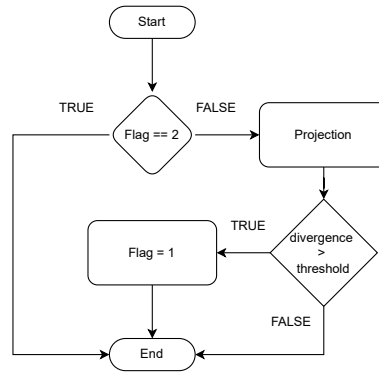


Figure 4.7: Projection shader flowchart with early stopping

Note that to keep all data within the GPU, shaders are still invoked one hundred times. However, once the flag is set to 2, all subsequent passes skip the resource-intensive portion of the code.

4.5.7 Projection set split

The invocations of the projection shader are separated into four sets, as discussed in Section 4.4.2. Since no two cells within a set share an edge, their projection steps can be executed in parallel. However, it is also possible to separate the cells into just two sets while maintaining this condition. This would follow a checkered pattern, similar to a chessboard, where no two adjacent squares—whether black or white—are the same.

In fact, a similar separation is possible for a three-dimensional grid, dividing the voxels into two sets such that no voxel shares a face with another voxel in the same set.

Ultimately, the four-set implementation was chosen for its simplicity, and because the expected performance gains of reducing to two sets were minimal.

4.5.8 Obstacle encoding

Besides representing fluid, each cell can also be either a wall or an object. Understanding the distinction between these two types is essential. Walls are immovable and act as physical boundaries; for example, in an ocean simulation, the rocks and sea floor would be labeled as "wall". Objects, on the other hand, can move within the fluid, so the BUV, for instance, would be considered an object.

To encode this information, two buffers are used: one for walls and another for objects. While it's unnecessary to differentiate between individual walls in the simulation, it is crucial to calculate the forces acting on each object separately. Therefore, a mechanism to distinguish objects is required.

The object buffer assigns an integer value to each cell, where each bit of that integer represents a specific object. Each object is assigned a unique ID from 0 to 31. To determine whether a cell belongs to object n , we examine the bit at position n .

Following the design convention used for walls, a bit value of 0 indicates that the cell belongs to the corresponding object, while 1 means it does not. The ID range is constrained by the size of an integer in Unity, which is 4 bytes or 32 bits. This limit of 32 objects is considered reasonable, as the number of objects in a typical simulation is expected to remain relatively small.

Note that, as of now the movement of the objects is not yet implemented, so the difference between objects and walls is only the ability to calculate forces on objects.

4.5.9 Force calculation

The calculation of the forces acting on an object is a crucial part of the simulation. These forces will enable future developers to extend the tool, allowing objects to move according to the forces applied to them.

The total force on an object is the sum of the forces exerted on its edge cells. Edge cells are defined as any cell of the object that borders at least one fluid cell. The force acting on an individual cell is a function of the pressure exerted by its surrounding fluid cells. Since pressure is computed during the projection step, the corresponding forces can also be determined.

4.5.10 Dispatch dimensions

When developing a shader, the "numthreads" attribute tells the GPU how many threads should be in each thread group. Most of the shaders developed for this project use "[numthreads(8,8,1)]", which means each group contains $8 \times 8 \times 1 = 64$ threads. These threads are then grouped into "waves", referred to as warps by NVIDIA and wavefronts by AMD. AMD typically uses wavefronts of 64 threads, while NVIDIA uses warps of 32 threads. Therefore, a thread group of 64 threads would generate one wave on an AMD GPU and two waves on a NVIDIA GPU.

If the thread group size is not a multiple of the wave size, the final wave will still be executed, but some of its threads will be deactivated. This wastes potential threads, so it is important to ensure all waves are "full." That's why thread groups of size 64 are commonly used, it is the smallest number that avoids waste, regardless of the GPU vendor.

Thread groups can be defined with any three-dimensional configuration, depending on the problem. For one-dimensional data, they often take the form "[numthreads(64,1,1)]". When working with a three-dimensional grid, configurations such as "[numthreads(4,4,4)]" may be used.

This project uses one-dimensional buffers to store two-dimensional data. In such cases, two-dimensional thread groups are still helpful, for example, to simplify bounds checking. However, when a two-dimensional layout offers no additional benefits, a one-dimensional thread group is used. A good example is the algorithm to find the maximum pressure on the grid, which simply involves scanning all values and does not require any two-dimensional structure.

4.5.11 Dynamic time step

The advection step of the simulation utilizes a time step. Currently, this time step is a hardcoded variable set to 1. One possible improvement would be to make the time step dynamic, following the CFL condition presented in Section 2.1.2.1: "The CFL condition dictates that the time-step (Δt) must be small enough for the fluid in the center of a cell to not be able to go further than the neighboring cell center."

Implementing the CFL condition requires knowledge of the maximum velocity on the grid to guarantee that, even in the worst-case scenario, the fluid does not travel farther than one cell. However, as explained earlier, finding maximum values is computationally expensive. For this reason, a dynamic time step was not implemented.

One possible solution would be to compute the maximum velocity every n frames and adjust the time step accordingly. This approach would distribute the computational cost of finding the maximum across those n frames. Assuming that maximum velocity does not increase significantly between frames, this method could offer performance benefits. It was not implemented due to time constraints in the current project but might be worth exploring in future iterations.

FLUID SIMULATION INTERFACE

This chapter serves as a guide for operating the fluid simulation developed in this project. The previous chapters focused on the theoretical background, the algorithms, and their implementation. In contrast, this chapter is dedicated to helping users work with the tool, configuring test environments and extracting valuable data from the simulation.

The tool is designed to support a variety of use cases, and users should be able to interact with it and customize parameters effectively. This includes selecting the visualization type, adding objects, controlling fluid speed and so on. Only with a thorough understanding of the tool's capabilities can users fully leverage its potential.

Emphasis is placed on usability and reproducibility; therefore, implementation details will not be discussed.

The chapter is organized to follow the user's journey when interacting with the tool. It begins by outlining the system requirements necessary to ensure a smooth experience. Next is a section on setting up the project on the target system. Once the project is set up, users define the initial parameters of the simulation. This is followed by a section describing how to interact with the tool during the simulation. Finally, the last section discusses how users can analyze the data generated by the simulation, either for extraction or debugging purposes.

5.1 System Requirements

To fully harness the fluid simulation, users must meet certain minimum software and hardware requirements. These requirements reflect both the resource-intensive nature of fluid simulations and the choice of development platform, Unity.

Users can create personalized environments by adjusting the resolution, adding objects, and more. This variability in complexity will significantly impact performance, but it will not produce incorrect results, provided the minimum requirements are satisfied.

In the following sections, recommended hardware specifications and supported software configurations are presented in detail. These are intended to help users prepare appropriate environments for running and analyzing fluid simulations.

5.1.1 Hardware Requirements

The fluid simulation was designed to leverage the GPU's processing capabilities; therefore, its performance is highly dependent on GPU quality. However, other system components also influence simulation efficiency. The following configurations offer reliable performance and scalability:

- **Processor:** Quad-core CPU. Since the CPU performs minimal computation, the minimum requirement is relatively low.
- **Graphics:** (e.g., NVIDIA GTX 1060+, AMD RX 580+) Any GPU that supports the DirectX 11 API (or higher). DirectX 11 introduced compute shader functionality, which this simulation heavily relies on. Given the simulation's dependence on GPU computation, this component will likely be the primary performance bottleneck. A more powerful GPU typically results in a higher frame rate. However, as long as the minimum requirements are met, the accuracy of computations remains unaffected.
- **Memory:** Minimum 8 GB RAM. While 8 GB is sufficient, higher simulation resolutions may lead to performance slowdowns. Therefore, 16 GB or even 32 GB is recommended for better responsiveness.
- **Storage:** The Unity project currently occupies around 1.5 GB. Users intending to extend the tool should ensure their system has adequate storage capacity.

In future developments, particularly when integrating reinforcement learning algorithms, the presented requirements may prove insufficient. Given the time-intensive nature of such algorithms, a low frame rate would significantly hinder the tool's overall performance.

5.1.2 Software Requirements

The software requirements are less stringent. If a system meets the hardware requirements, it will most likely satisfy the software requirements as well. The required components are listed below:

- **Operating System:** Windows 10 or 11. Linux and macOS may be compatible, but no guarantees can be made since the tool was only tested in Windows environments.
- **Installation:**
 - **Unity:** Version 6000.0.38f1. This is the version used during development. To ensure the project runs as expected, this version should be used. However, if a newer version is required to implement a new feature, Unity can upgrade the project. This conversion may lead to changes in behavior, so it is generally discouraged. From experience, the risk is minimal, especially if the major version remains the same (*i.e.*, Unity 6 or any version beginning with 6000).

- **Text Editor:** Any text editor. Microsoft Visual Studio was used during development and is recommended by Unity, but any editor with support for C# and HLSL will work.
- **Python:** Python has not yet been used in this project. However, if the tool is extended to support reinforcement learning, Python will be required.

The warning noted in the operating system bullet also applies to other points, including hardware. The tool has only been tested on a limited range of systems. While no inconsistencies were found during testing, it would be premature to assume compatibility across all platforms.

5.2 Unity Setup/Startup

If the system meets all requirements, the user can proceed to install Unity. Unity offers a free version for individual use¹.

1. **Install Unity:** Visit the Unity website² and follow the download instructions to install Unity Hub. Unity Hub is the application where Unity projects are managed. It differs from the Unity editor, which is the actual software used to develop games and applications.
2. **Clone the repository:** Clone the project using `git clone` or download it directly from the GitHub page³. It is recommended to clone the repository before installing the correct Unity editor version, as Unity will automatically detect the project and suggest the appropriate version.
3. **Install the correct editor version:** After Unity Hub is installed, open it and click "Add" → "Add project from disk." Once Unity recognizes the project, it will prompt you to install the matching editor version if it's not already available.
4. **Opening Unity:** After completing the previous steps, the project is ready to be launched. The first time opening may take a few minutes. After opened Unity will display an interface similar to the one shown in Figure 5.1.
5. **Understanding Unity:** The Unity editor consists of several tabs. As shown in Figure 5.1, six key tabs are visible:
 - The **Project** tab in the bottom left corner displays folders and files. If Unity opens a different scene, navigate to Assets/FluidSimulation/Scene and double-click the scene file.

¹<https://unity.com/pt/products>

²<https://unity.com/download>

³https://github.com/lucascorreialobo/Unity_Project_Tese/tree/main/Project

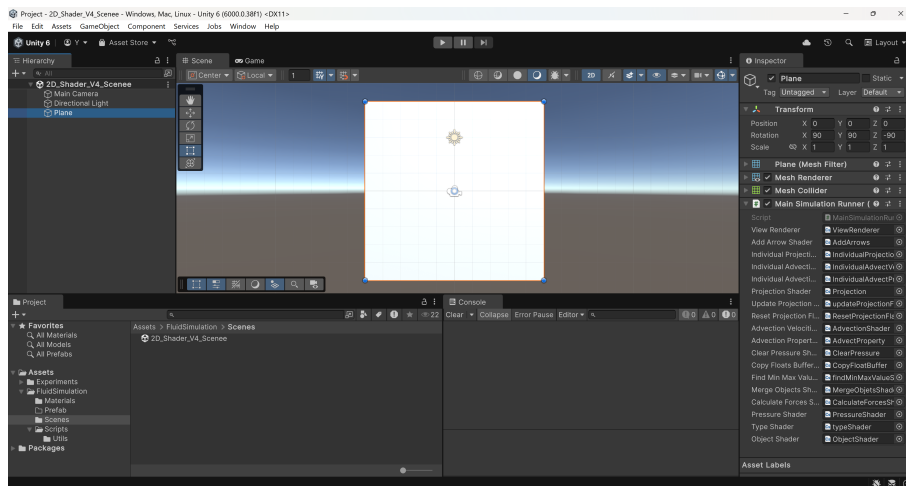


Figure 5.1: First look into the Unity editor

- The **Hierarchy** tab, located in the left column, lists all game objects present in the scene. At minimum, it includes a "Camera" and "Light Source." Additionally, a "Plane" game object is used to display and host components for the fluid simulation.
- The **Console** tab, located at the bottom, right of the Project tab, displays debug messages and other logs.
- The **Inspector** tab on the right shows the properties of selected game objects. In Figure 5.1, the Inspector displays details about the "Plane" object.
- The central area contains the **Scene** and **Game** tabs. The Scene tab provides a customizable view of the environment that the developer can freely navigate. The Game tab displays the camera's perspective and is typically used during simulation runtime.

5.3 Running a Simulation

Before running a simulation, a few parameters can be configured. The most important is the simulation resolution, which defines the size of the grid used for the simulation. The default resolution is 128×128 , which is large enough to create interesting scenarios but not so large that individual cells become indistinguishable, helpful when debugging.

To run the simulation, click the "Play" button at the top of the Unity window. After a few seconds, the simulation will begin. Visually, this is indicated by the colouring of the plane into the pressure view mode, as shown in Figure 5.2.

The Play button transforms into a Stop button, which, when pressed, will end the simulation. To its right are the "Pause/Resume" and "Step" buttons, both of which only function while the simulation is running. The Pause button halts Unity's execution until pressed again, while the Step button advances the simulation by one frame each time it

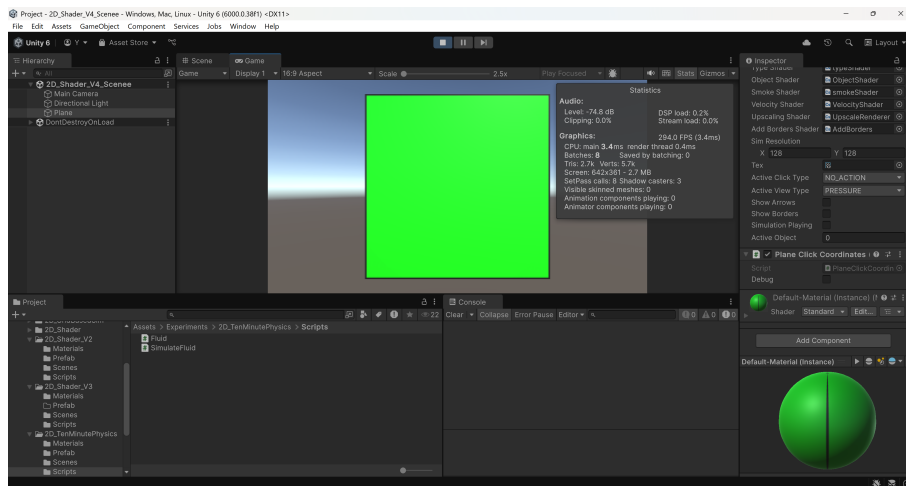


Figure 5.2: Start of simulation

is clicked, executing the Update function once per press. These buttons will serve little purpose in this project, since similar features have been implemented that better suit its requirements.

To pause or resume the simulation, the user can press the letter “P.” This bypasses the function in the script responsible for updating the simulation. To advance a single update cycle while the simulation is paused, press the letter “N.” These controls differ from Unity’s built-in buttons by skipping the update of the simulation state but still allowing rendering, so the effects of any changes to the simulation state can be seen instantly.

By default, Unity does not open the Stats window shown in Figure 5.2, but enabling it can help monitor simulation performance. It displays metrics such as frames per second (FPS) and frame execution time (shown in parentheses next to the FPS). To open or close the window, click the “Stats” label in the top-right corner of the Game tab.

5.4 Features

The default environment when running the simulation is an enclosed two-dimensional box (all borders are walls), with no objects, all velocities initialized to zero, and no smoke.

To help developers customize the simulation according to their own needs, several features have been added. These include adding velocity, calculating the forces applied to an object and switching between pressure view and smoke view.

In this chapter, the features will be categorized into three groups. The first group encompasses features that modify the simulation state. The second group includes features that change how the simulation state is displayed. The final group consists of features used to inspect the simulation state.

Every feature has been implemented in a script, and since in Unity the scripts must be associated to a game object, the “Plane” was selected as the holder of such scripts. These means that to access any of the features the “Plane” game object, must be selected and

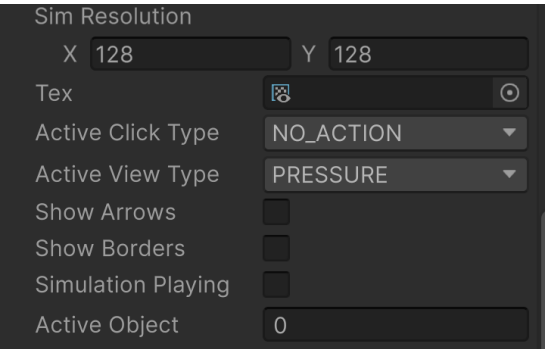


Figure 5.3: Customizable variables

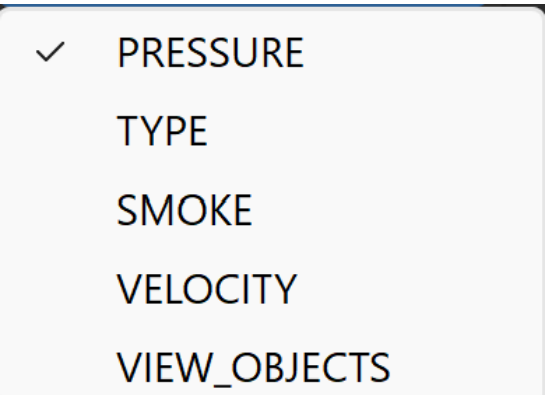


Figure 5.4: Active View Types dropdown

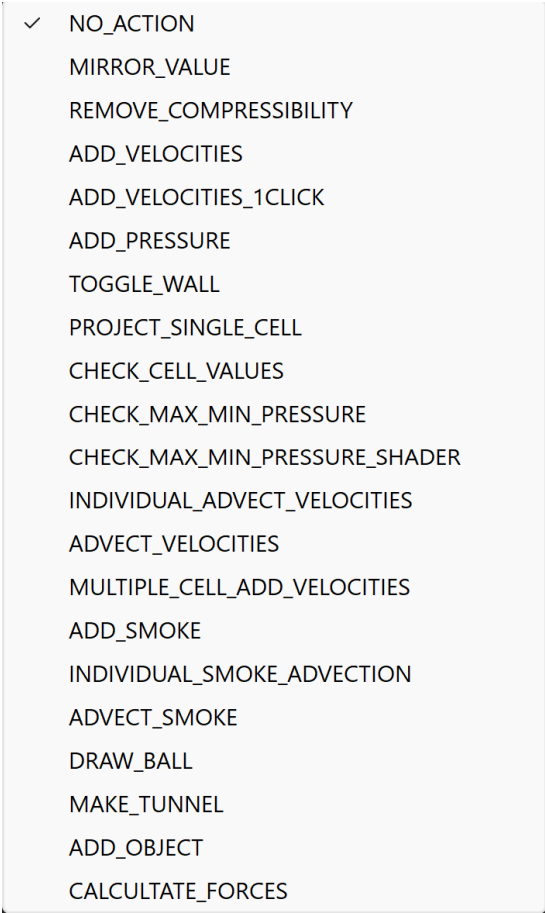


Figure 5.5: Active Click Type dropdown

Figure 5.6: Options available in the Unity inspector

displayed in the "Inspector" tab, which is the case in Figure 5.2.

To apply the effects of each feature, the user must click on the simulation. In cases where mouse position is relevant, it will be taken into account. In other cases, the click will produce the same outcome regardless of its position. Only one feature can be selected at a time.

5.4.1 State Altering Features

The simulation state includes pressure, velocity, objects, and smoke across the entire grid. Features in this group modify at least one of these properties.

All features in this group are implemented in the *MainSimulationRunner* and are initially accessed by selecting an option from the dropdown menu labeled *Active Click Type*. From this menu, choose the appropriate option to modify the desired property.

The following list presents the options from this group in the dropdown, in order of appearance:

- **REMOVE_COMPRESSIBILITY** executes the first step of the simulation, also known

as projection. This modifies velocity and pressure throughout the grid and is primarily useful for debugging errors in this part of the algorithm. To execute the projection, click anywhere on the grid. The effects of this feature can be seen on Figure 5.7b. In the previous Figure, velocity had been added, causing some cells to "lose" more fluid than they were "gaining", or the other way around, but and after executing the REMOVE_COMPRESSIBILITY function, the velocities were updated to ensure they no longer contain this unbalance.

- **ADD_VELOCITIES** adds velocity to a single edge. To use it, click one cell and then a neighboring cell. Velocity will increase by 0.3 in the direction from the first cell to the second.
- **ADD_VELOCITIES_1CLICK** allows continuous velocity addition with a single click. By clicking and holding, velocity is continuously added in a fixed direction. The direction is hardcoded; to change it, go to the *MainSimulationRunner* and modify the *direction2D* variable in the function *AddVelocities1Click(gridCoord)*. Figure 5.7a was obtained by having the ADD_VELOCITIES_1CLICK active and clicking and dragging the mouse on the simulation space.
- **TOGGLE_WALL** toggles the clicked cell between wall and fluid.
- **PROJECT_SINGLE_CELL** executes the projection step for a single cell. Click the cell to apply the projection. Like REMOVE_COMPRESSIBILITY, this is mainly used for debugging.
- **INDIVIDUAL_ADVECT_VELOCITIES** advects the four velocities surrounding a single cell. Click the cell whose surrounding velocities should be advected.
- **ADVECT_VELOCITIES** executes the velocity advection step for the entire grid. To use it, click anywhere on the grid. Figure 5.7d shows the effects of using this function a few time on the previous Figure. Note that only the velocities were advected, so the fluid may now contain cells which have more fluid coming in than going out. Which is the sharp edge between pixels at the edges of velocity.
- **MULTIPLE_CELL_ADD_VELOCITIES** adds velocity to a group of cells within a radius, in a specified direction. Like ADD_VELOCITIES, the first click defines the center, and the second click determines the direction. The direction is calculated by comparing the x and y distances between the two points; whichever is greater determines the direction. This allows the second click to be anywhere in the general direction of the intended velocity.
- **ADD_SMOKE** increases the smoke amount in the selected cell by one. Click the cell to apply the change. This function is useful to visually understand how the fluid is moving. It was used in both Figure 5.7e and Figure 5.8i. In the first case the

simulation was stopped during the use of the `ADD_SMOKE`, while it was running in the second.

- **INDIVIDUAL_SMOKE_ADVECTION** advects the smoke of a single cell. Click the cell whose smoke should be advected.
- **ADVECT_SMOKE** advects smoke across the entire grid. To activate it, click anywhere on the grid. By applying the `ADVECT_SMOKE` function to Figure 5.7e the Figure 5.7f is obtained. Since only the smoke is being updated the velocities remain the same throughout the simulation space.
- **DRAW_BALL** is the first feature that requires more than just a click. Before creating the "ball," an index must be selected. By default, the index is zero, but this can be changed in the inspector by entering the desired index into the "Active Object" text box. After setting the index, clicking a cell will transform all cells within a ten-cell radius into part of the object with the specified index. `DRAW_BALL` was used to create the obstacle seen in Figure 5.8f.
- **MAKE_TUNNEL** transforms the grid into a wind tunnel. This means the fluid enters from the left edge with an initial velocity and can "exit" through the right edge. `MAKE_TUNNEL` was used in Figure 5.8b, causing the pressure to spike in both the intake and exit edges. After a few frames (at 128x128 it takes no more than 30 frames) the simulation regains stability as can be seen in Figure 5.8e.
- **ADD_OBJECT** converts a single cell into part of the object with the index specified in the "Active Object" field.

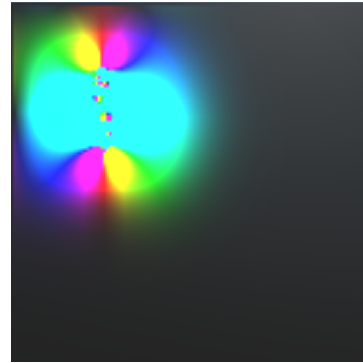
5.4.2 Visualization Features

Visualization features do not alter the simulation state. Instead, they change how that data is displayed to the user. To use any of the features in this group, open the dropdown menu labeled *Active View Type*. The options in this dropdown represent the different visualization modes. Each cell is colored according to a specific rule, which varies depending on the selected option. The following list explains the rule used for each visualization type: (All view types already appear on Figure 3.7)

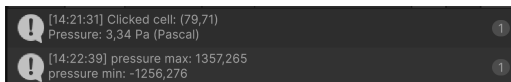
- **PRESSURE** is displayed using a three-color gradient that transitions from blue to red, with green as the midpoint. This is a common color mapping in scientific visualization for representing scalar fields. The gradient implementation uses fixed maximum and minimum values, so the same color always corresponds to the same pressure—even across different simulations. The downside is that if the actual pressure exceeds the maximum, it will still be shown as red, and similarly, values below the minimum will appear as blue. To modify the range, go to `PressureShader.compute` and adjust the values passed into the `getSciColor()` function.



(a) Adding left to right velocity with ADD_VELOCITIES_1CLICK

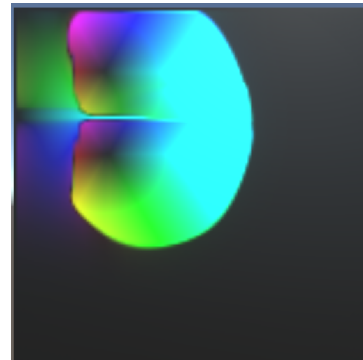


(b) Executing the first step of the simulation update, using REMOVE_COMPRESSIBILITY

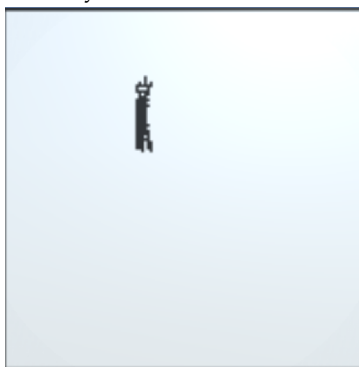


(c) Logs produced by CHECK_CELL_VALUES^a and CHECK_MAX_MIN_PRESSURE

^aClicking the CHECK_CELL_VALUES log, will show more information such as the cell type (0 - Wall/Object | 1 - Fluid), density of smoke and the velocity of the fluid at its four edges



(d) Advecting only the velocity, with ADVECT_VELOCITIES



(e) Changing to SMOKE view and adding smoke with ADD_SMOKE



(f) Advecting only the smoke with ADVECT_SMOKE

Figure 5.7: Example usage of some of the features

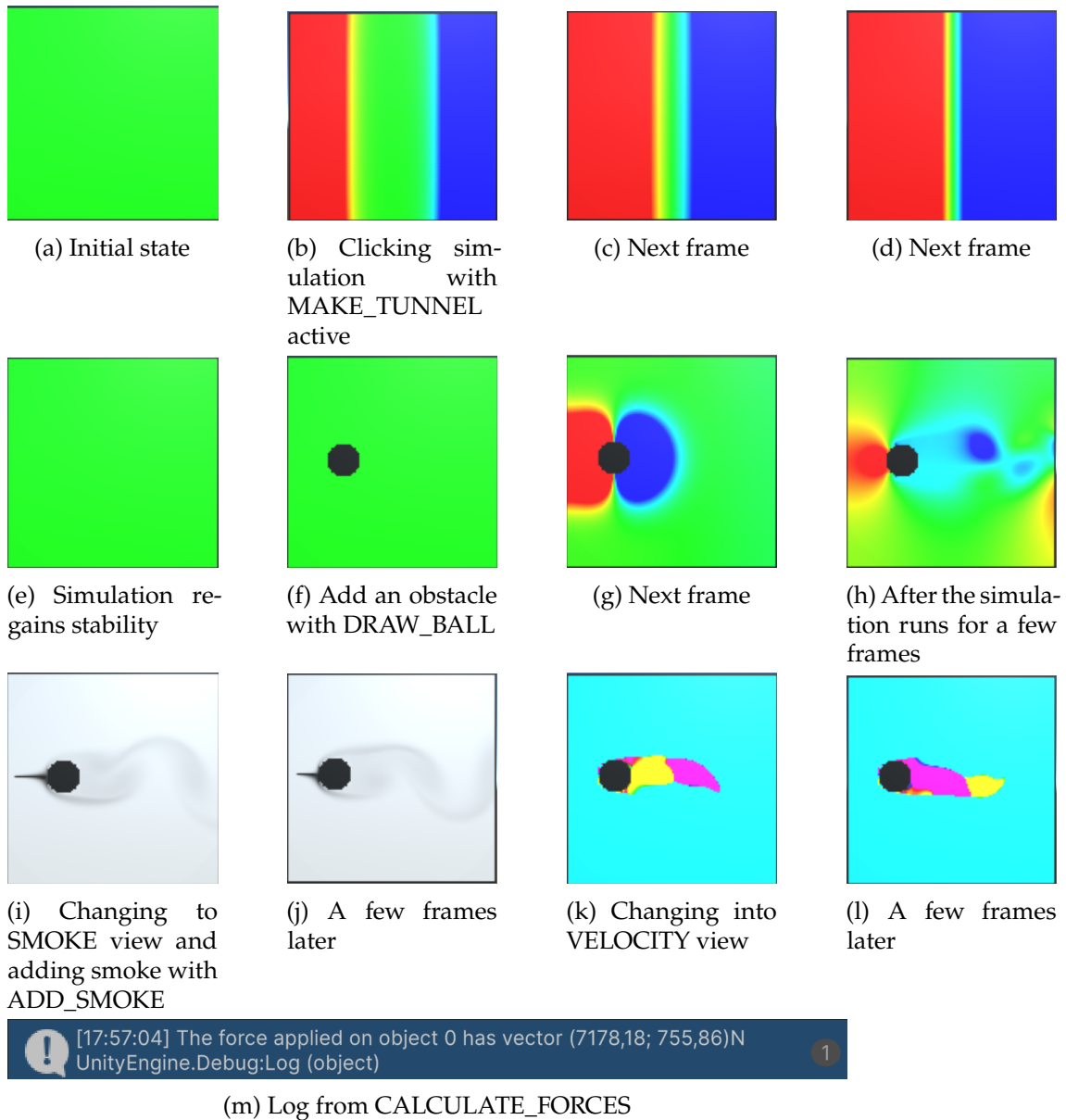


Figure 5.8: Example setup of tunnel environment

- **TYPE** colors fluid cells white and all other cells black.
- **SMOKE** uses a gradient from white to black. The more smoke present in a cell, the darker it appears. As with pressure, the darkest color (black) corresponds to a hard-coded value. To change the range, modify the relevant values in *SmokeShader.compute*.
- **VELOCITY** visualization first calculates the velocity at the center of each cell, since velocity is stored at the edges. The resulting two-dimensional velocity is then mapped onto a color wheel, as shown in Figure 3.7g. Lower velocities appear darker. For example, if the velocity in a cell points to the right, the cell will be colored blue, since blue represents the rightmost direction on the wheel.
- **VIEW_OBJECTS** colors cells similarly to the TYPE option, but with a distinction: walls are shown in black, while objects are colored based on their index. For example, an object with index 0 is colored red, index 1 is green, and so on.

Unless explicitly stated otherwise, walls and objects are displayed in black.

Additionally, there are two other visualization features that are not included in the dropdown menu but are instead available as checkboxes located just below it. These are **Show Arrows** and **Show Borders**.

The **Show Borders** option toggles blue borders between each cell, helping to clearly distinguish neighboring cells. The **Show Arrows** option displays a blue arrow on each edge, indicating the magnitude and direction of the velocity at that edge. Both of these features function correctly only if the rendering step uses a 100×100 box to represent each cell, as the arrows and borders rely on fixed pixel dimensions for drawing. To increase the number of pixels per grid cell, the variable *viewResMult* in the *MainSimulationRunner*, must be set to 100.

5.4.3 Inspection and Analysis Features

Finally, the last group of features neither alters the simulation state nor changes how the data is displayed. Instead, these features help the user better understand what is happening in the simulation by writing detailed information to the "Log" tab. This group includes the following features, all accessible from the *Active Click Type* dropdown:

- **CHECK_CELL_VALUES**: After selecting this option and clicking a cell in the grid, all of the cell's properties will be printed, including the velocities of its four surrounding edges. The printed data includes: cell index, pressure, type (fluid or wall/object), smoke amount, and total divergence (fluid leaving the cell), followed by the velocities on each edge in the order: left, down, right, up. This was the function that generated the first log message seen in Figure 5.7c.

- **CHECK_MAX_MIN_PRESSURE:** Writes the maximum and minimum pressure values of the current simulation state. This feature is executed entirely on the CPU. This was the function that generated the second log message seen in Figure 5.7c.
- **CHECK_MAX_MIN_PRESSURE_SHADER:** Performs the same task as the previous feature, but the calculations are done on the GPU. This was useful for comparing performance between the two processing units.
- **CALCULATE_FORCES:** Calculates the force being applied to the object with the index specified in the *"Active Object"* text box. Displaying this information in the *"Log"* tab. This was the function that generated the log message seen in Figure 5.8m.

EXPERIMENTAL RESULTS

This chapter evaluates the implementation of the fluid simulation. A thorough evaluation is essential to ensure that the final product meets the goals outlined in this thesis and the requirements of the SABUVIS project. The evaluation aims to address the following research questions:

- How can we develop a physics-inspired fluid simulation?
- How can the fluid simulation be used to simulate interaction with a fin?
- Is it feasible to develop a BUV simulator with reasonable computational demands using a game engine?

To answer these questions, various tests and metrics were developed. The first two research questions can be addressed through comparison with expected results. Specifically, we assess whether the fluid behaves in a natural and realistic manner. One informal but insightful test involves "fooling" a human observer, if the simulation appears fluid-like, it is likely at least partially physics-inspired. A more rigorous approach involves comparing the results with those from established fluid simulations. Such comparisons can validate the accuracy of the simulation, but it is also important to ensure that the algorithms perform correctly under edge-case conditions.

The third research question requires a different evaluation metric. To determine whether the simulation operates within reasonable computational limits, performance indicators such as frames per second (FPS) or frame execution time are more appropriate. These metrics should be recorded across varying parameters, including resolution, post-processing effects, rendering methods, and others.

This chapter is divided into three sections. The first presents tests assessing the correctness of the simulation, which help answer the first two research questions. The second section focuses on performance tests, serving a dual purpose: addressing the third research question and identifying which components most significantly affect performance. Finally, the third section discusses the results of these tests, how they relate to the research questions, and other insights gained during the evaluation process.

6.1 Correctness of simulation

For the fluid simulation to be useful it must reflect the behaviour of water. Some test were created, to prove the simulation is in fact replicating a fluid's movement. This section contains the methods and tests conducted.

The first method used to evaluate the correctness, was a visual test. Humans, through their experience with the natural world gain an intuition on how fluids should behave. So if a fluid simulation, looks visually correct it is a good initial indication of its quality. As seen in Figure 6.1 (which already appeared as Figure 3.7b), the smoke is being dispersed in a seemingly realistic manner. Visual appearance is enough for some applications, such as in animation, but since this fluid simulation was designed to be used as the training enviroment of a BUV, simply appearing correct is insufficient.

Hence, the next step is to compare our fluid simulation with other well-established simulations. Figure 6.2 presents this comparison, showcasing our results alongside those of Raissi [58]. The depicted simulation features a rectangular obstacle placed at the center of a "wind tunnel" setup. Although the colors might not match exactly, this is due to variations in color grading rather than discrepancies in the simulation itself. For example, Raissi uses light blue to represent zero pressure, while our simulation uses green, explaining the difference in background color.

Despite these apparent differences, the simulations are fundamentally identical. This is evident from the concentration of pressure upstream of the obstacle and the formation of vortices downstream, as indicated by the pressure drop.

Beyond similar results to others simulations, it is also important to guarantee, the implementation of the algorithms does not contain any errors as small errors may remain hidden when comparing results. The performed tests, presented next, aim to establish that the various simulation steps follow, the presented algorithm. The tested steps were: (All the tests were preformed on an 8x8 simulation grid)

- **Pressure reset:** As part of the algorithm the pressure of each cell must be set to zero, before the projection step, where pressure is calculated. To prove the pressure is in fact being set to zero, the click action `REMOVE_COMPRESSIBILITY` was used, since it starts by setting the entire grid's pressure to zero. To start the velocity of one edge was set to 0.9, then `REMOVE_COMPRESSIBILITY` was applied twice, the first

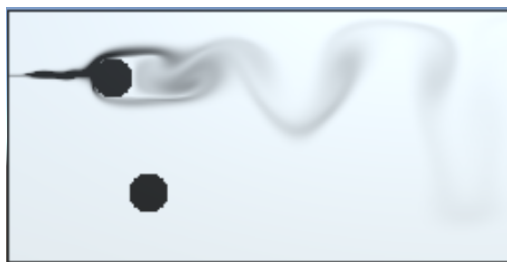


Figure 6.1: Smoke flowing

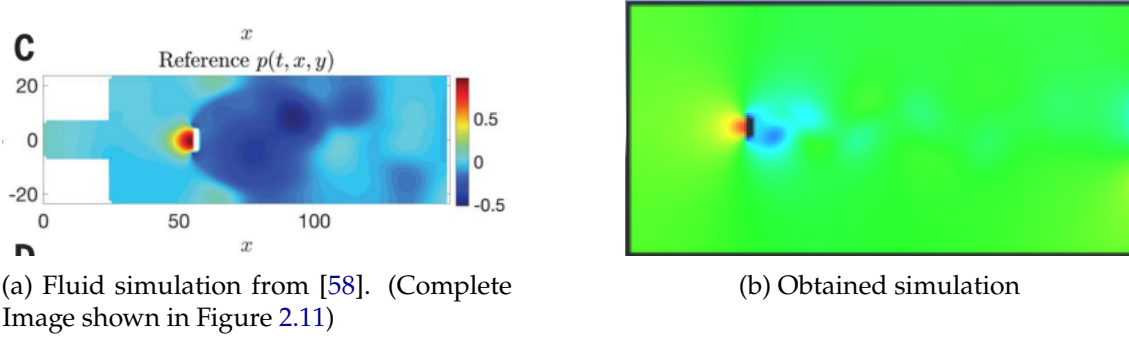


Figure 6.2: Comparison of obtained result with other simulations

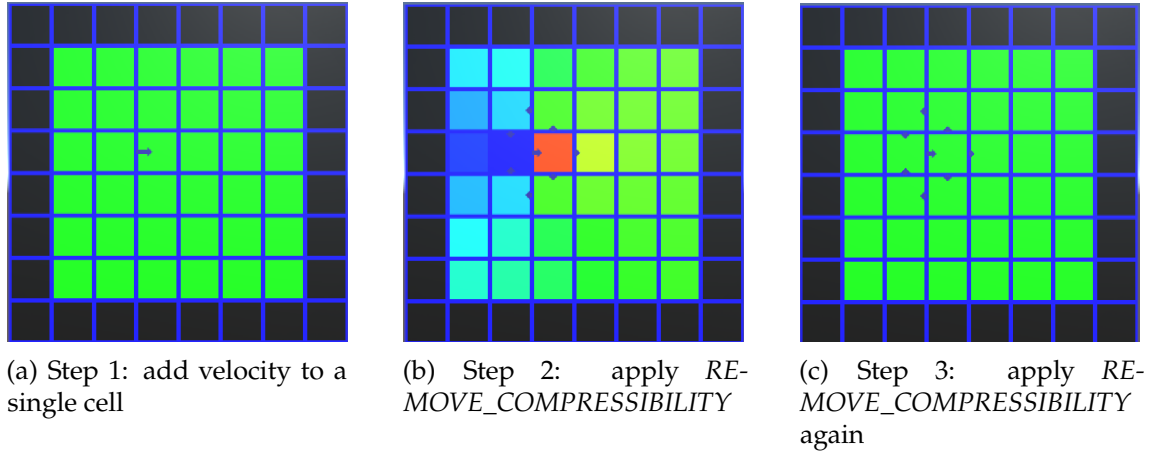


Figure 6.3: Steps to prove reset pressure works

time the pressure of surrounding cells changes drastically, but on the second one the pressure is again zero. This is only possible if the reset works, since the grid is already incompressible on the second execution of *REMOVE_COMPRESSIBILITY*, thus only the pressure reset effects can be seen. Figure 6.3 shows the evolution of the grid when using this method.

- **Projection:** To test whether the projection step is working correctly apply the *REMOVE_COMPRESSIBILITY* on any grid, then use *CHECK_CELL_VALUES* to check the divergence of each cell. This was done the after Figure 6.3a, and in fact the divergence was 0 or very close to it on every cell.
- **Velocity Advection:** Testing the velocity advection was done using the *INDIVIDUAL_ADVECT_VELOCITIES*. First the some velocities were added to the grid, then the advections were done on paper and by the simulation and the observed result was the same.
- **Smoke Advection:** Smoke advection was tested in the same manner as the velocity advection, and the conclusion was the same, the algorithm is correct.

Another good test is to experiment with the simulation under extreme scenerious,

to understand at which point it longer reflects the real-world. To test the limits of this simulation the velocities when "transforming" the grid into a wind tunnel, were increased from 0.6 to 10.6. This resulted in the simulation breaking. The behaviour was expected, since one of the limitations is that the fluid, must never travel more than one cell's width in a single time step, which is not respected when the velocity increases this much.

Ideally the results of the simulation would also be compared to real-world tests, but this was not possible for this project.

6.2 Optimizations Benchmark

With the simulation confirmed to exhibit the expected behavior, the next focus is on efficiency. Given the ultimate goal of the SABUVIS project, efficiency is paramount, the faster the simulation runs, the faster the reinforcement learning algorithm can operate.

The following tests aim to demonstrate that the optimizations made during development do, in fact, result in faster execution. They also serve as a reference for future developers seeking to improve performance or design scenarios using the simulation.

Accordingly, the tests measure the impact of various features on the execution time of a single frame. The features tested are: resolution, early stopping, `resMult`, borders, arrows, rendering types, number of objects, overrelaxation, and finally, a breakdown of the time spent on each step of the simulation.

For example, when testing early stopping, the simulation is first run with early stopping enabled, and the execution time of each frame is recorded. Then, the simulation is run again with early stopping disabled, and the execution time is recorded once more. The two averages are then compared, in this case a reduction in execution time is expected when early stopping is enabled.

To ensure comparability, all tests were conducted using the base scenario shown in Figure 6.4, with the following configuration:

- Resolution: 128×128
- Early Stop: ON
- ResMult: 1
- Border: OFF
- Arrows: OFF
- Rendering Type: Pressure
- Number of Objects: 1
- Wind Tunnel: ACTIVATED
- Overrelaxation: 1.5

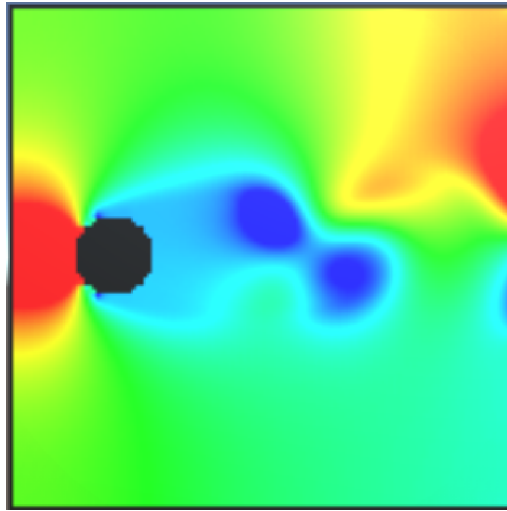


Figure 6.4: Default scenario for tests

Some variation may occur between scenarios, as the ball is placed manually and cannot be positioned in exactly the same location each time. All other variables remain constant unless they are the subject of the test.

Each test is introduced with a brief explanation, followed by graphs showing the results. An analysis and discussion of the findings are also provided.

Most of the graphs presented are violin plots, chosen for their ability to convey both the distribution of the data and embedded box plot in a compact manner. When applied to simulation data, these plots often exhibited long upper tails caused by outliers. These outliers made the graphs harder to interpret and did not contribute meaningful insights. As a result, they were removed from the dataset. While this means the graphs no longer represent the raw data with complete accuracy, the impact on the overall results was minimal, and readability was significantly improved. Since the drawbacks do not apply to tables, these reflect the entire data.

6.2.1 Rendering type

To give users maximum flexibility in how the simulation is viewed, five rendering types are offered: **PRESSURE**, **TYPE**, **VIEW_OBJECTS**, **SMOKE**, and **VELOCITY**. These rendering types do not affect the simulation itself; they are simply different ways of visualizing the same underlying data. For a deeper analysis, Section 5.4.2 explains the data displayed by each visualization type, while Section 3.3.7 describes the design and data structures behind them.

If implemented correctly, the visualization types should not have a significant impact on frame execution time. This test was designed to confirm that assumption. Frame execution time was recorded across more than one thousand frames for each rendering type, and the results are presented in Figure 6.5 and Table 6.1.

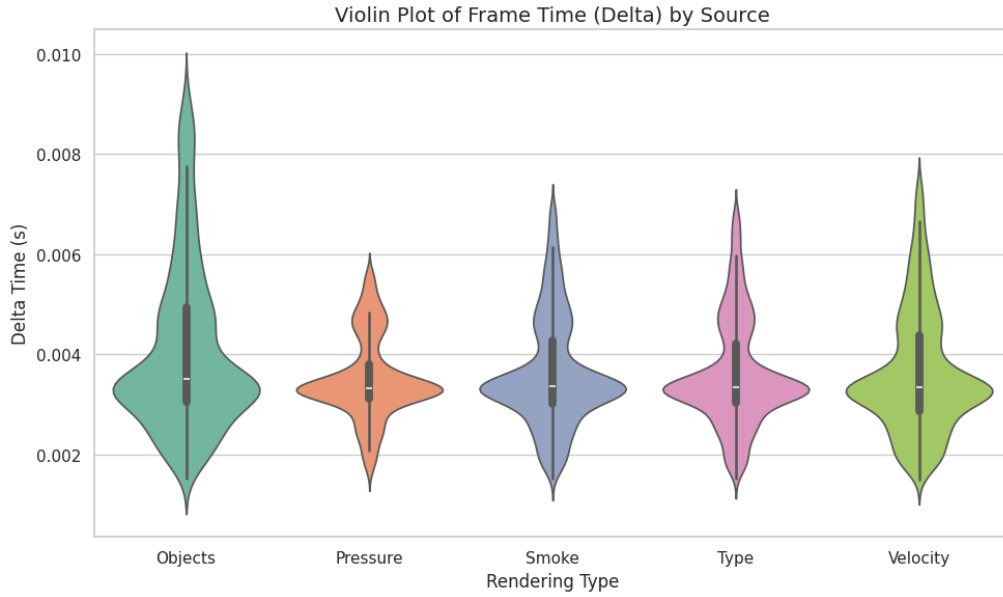


Figure 6.5: Effect of rendering type on frame execution time

Rendering type	Average Frame (ms)	Max Frame (ms)	Min Frame (ms)
Objects	4.894	32.193	1.512
Pressure	3.683	21.338	1.545
Smoke	4.103	29.854	1.527
Type	4.064	26.721	1.529
Velocity	4.195	35.977	1.502

Table 6.1: Summary of Frame Time Statistics by Rendering Type

As shown in Figure 6.5 and Table 6.1, the impact on execution time is indeed minimal. The two visualization types with the largest deviations are those that require precomputation.

In the **VIEW_OBJECTS** visualization, determining which object a pixel belongs to involves multiple `if` statements, which are relatively slow on GPUs. The **VELOCITY** visualization also introduces a slight deviation, as it must average the velocities from the four surrounding edges to compute the cell's velocity. However, since this process does not involve conditional branching, the performance impact is smaller.

6.2.2 Overrelaxation

Overrelaxation is a constant used during the projection step to accelerate convergence toward an incompressible fluid. Its value should lie between one and two to ensure convergence, but the optimal value for fastest results must be determined through trial and error. In this test, six different values were compared to identify the most effective one. The potential speedup stems from the activation of early stopping, if the fluid is already incompressible, the projection step can terminate earlier, resulting in faster execution.

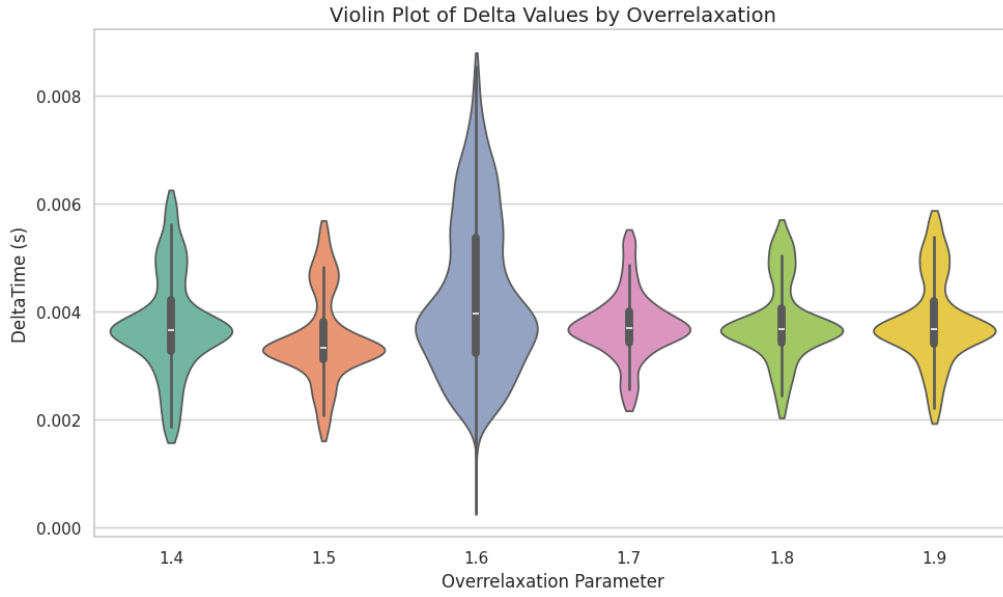


Figure 6.6: Effect of overrelaxation value on frame execution time

Overrelaxation Value	Average Frame (ms)	Max Frame (ms)	Min Frame (ms)
1.4	4.047837	17.574	1.487
1.5	3.682641	21.338	1.545
1.6	4.554454	27.726	0.257
1.7	4.329458	111.908	1.572
1.8	3.940792	17.368	1.505
1.9	3.951248	18.009	1.537

Table 6.2: Frame time statistics for different overrelaxation values

The results shown in Figure 6.6 and Table 6.2 indicate minimal differences between the various values of the overrelaxation constant. This is likely because, despite the faster convergence enabled by overrelaxation, early stopping is not triggered frequently. Although the average execution times are similar, the lowest average frame time was recorded with the constant set to 1.5. This suggests that, while the improvement is modest, this value allows early stopping to activate more often.

Table 6.2 also reveals an extremely abnormal frame execution time when using a constant value of 1.7. In this case, the longest frame took over one hundred and ten milliseconds, more than twenty five times the average. This anomaly was most likely caused by factors beyond the developer's control and not by an implementation error. One plausible explanation is the temporary diversion of computational resources by the CPU to another application, leaving the simulation with reduced processing power.

Such events highlight the importance of collecting a large number of data points and removing outliers when presenting violin plots.

6.2.3 Resolution

In this context, resolution refers to the number of cells used in the simulation. This value can be adjusted before the simulation begins. Since higher resolution results in a greater number of cells, it is expected to have a significant impact on simulation performance. To evaluate the effect of resolution on performance, several simulations were run, each with a different resolution, and frame execution times were recorded over a sufficient period to ensure more than one thousand frames were captured.

This test examines not only the impact of the total number of cells but also whether the grid structure, square versus rectangular, affects performance. The tested resolutions range from 64×64 to 512×512 , with two non-square configurations: 128×64 and 512×128 . The latter has the same total number of cells as a 256×256 grid, allowing for direct comparison.

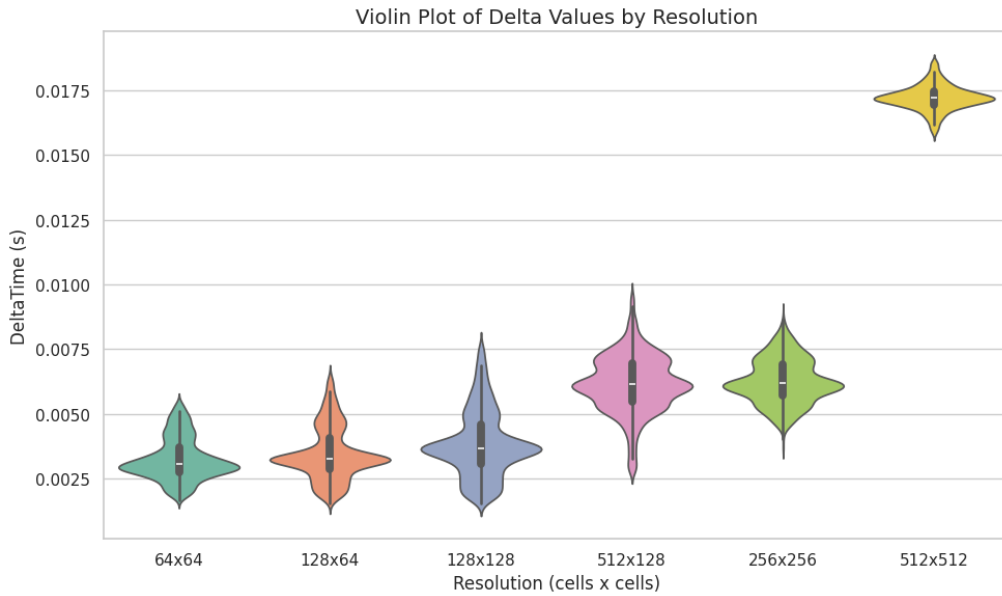


Figure 6.7: Effect of resolution on frame execution time

As expected, Figure 6.7 shows that frame execution time increases with resolution. By comparing the two configurations with the same number of cells, 512×128 and 256×256 , we can conclude that the shape of the grid has no significant impact on overall performance.

Figure 6.8 demonstrates that frame execution time increases linearly with the number of cells. This may not be immediately apparent from the first figure, where the number of cells grows quadratically due to the square grid dimensions.

6.2.4 Early Stop

Early stopping in this simulation refers to terminating the projection step early if the fluid is already incompressible, *i.e.*, all cells have equal amounts of fluid entering and exiting. Early stopping does not alter the maximum number of projection cycles, which remains set at one hundred. One of the key challenges in implementing early stopping, already

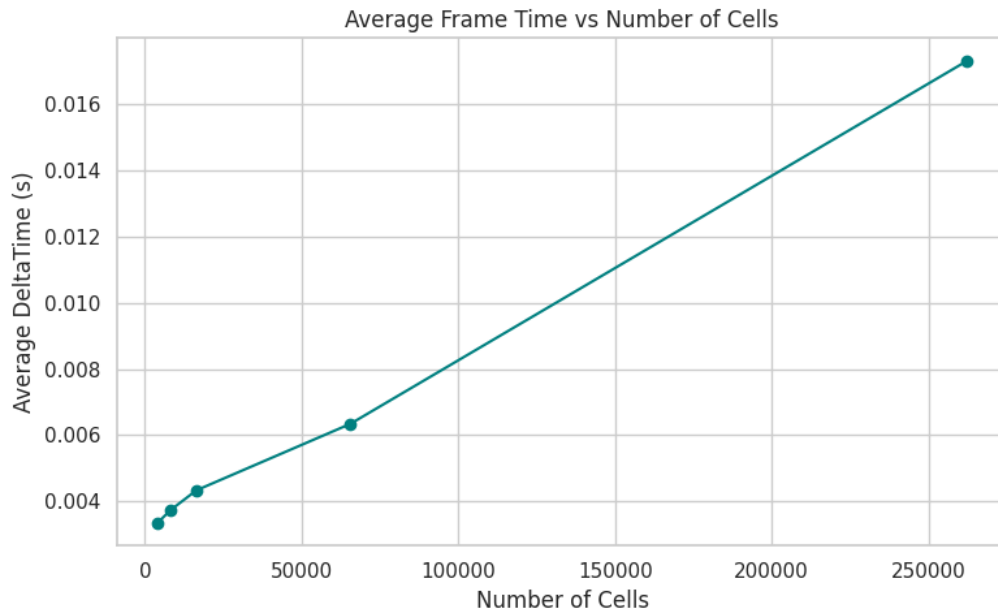


Figure 6.8: Relation between number of cells and frame execution time

discussed in Section 4.5.6, is achieving it without transferring data to the CPU, as doing so would negate the intended performance benefits.

For this test, six simulations were conducted, two for each of the three resolutions: one with early stopping disabled and one with it enabled. The results compare the implemented early stopping mechanism against a fixed approach that always performs one hundred projection cycles, regardless of the fluid's state.

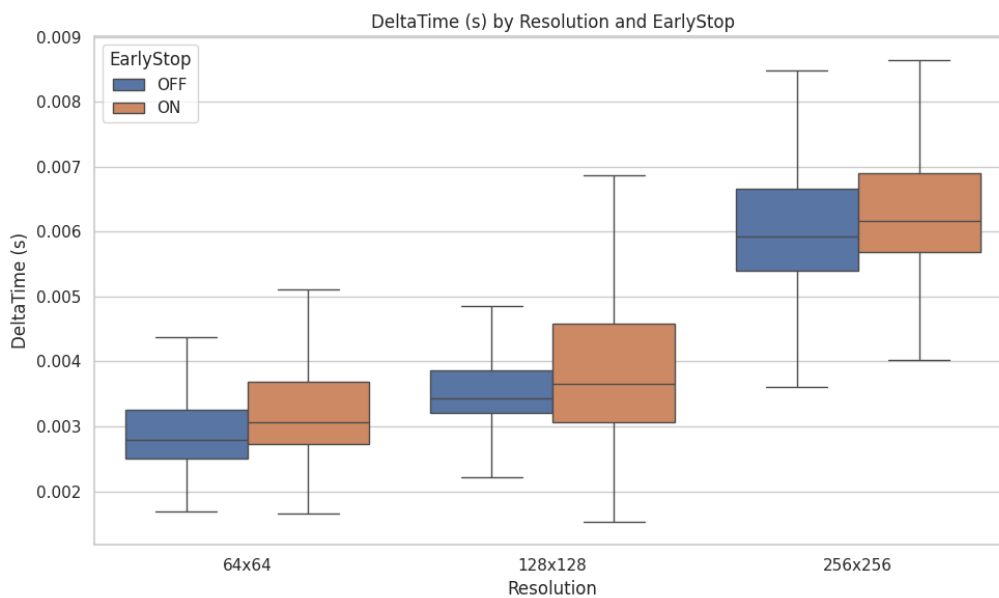


Figure 6.9: Effect of early stop on frame execution time

Contrary to expectations, Figure 6.9 shows that early stopping did not improve performance; in fact, it made the simulation slower across all tested resolutions. This likely indicates that the fluid does not reach the incompressibility threshold even after one hundred cycles. One possible solution would be to increase the threshold at which a cell is considered incompressible, though this must be done carefully to ensure the simulation continues to behave correctly.

For the purpose of this test, the threshold was set to 0.001. As a quick experiment, two other thresholds, 0.1 and 0.01, were visually tested. The 0.1 threshold resulted in incorrect behavior, as the vortices downstream of the object were no longer generated. While the 0.01 threshold appeared visually correct, further testing would be required to confirm its validity.

6.2.5 Post-effects

In addition to the five rendering types, two supplementary features were implemented to help users better understand fluid behavior: arrows and borders. Arrows are placed along cell edges to indicate the amount of fluid flowing between adjacent cells, while borders add blue outlines around each cell to make them clearly distinguishable.

Both post-processing effects—arrows and borders—require the variable `viewResMult` to be set to 100, as explained in the visualization features section (Section 5.4.2). This value represents the number of pixels per side of each cell, meaning a value of 100 results in a resolution of 100×100 pixels per cell. In the graph, this variable is labeled as "ResMult."

Although the underlying calculations are relatively simple, the GPU must process 10,000 times more pixels, since $100 \times 100 = 10,000$, which naturally leads to a decrease in performance. Additionally, the post-effects themselves contribute to the overall performance impact. This test aims to quantify their effect.

For this test, the simulation was first run with `viewResMult` set to 1 and no additional features enabled, establishing the baseline. Four more simulations were then conducted: the first increased `viewResMult` to 100 without enabling any post-effects; the next two enabled each post-effect individually; and finally, a simulation was run with both post-effects active.

Figure 6.10 shows that simply setting "ResMult" to 100 substantially increases the frame execution time, though perhaps not as drastically as a ten-thousand-fold increase in pixels might suggest. This is because the GPU shader responsible for upscaling performs only a multiplication and a division, followed by a single buffer access. These are simple operations that GPUs handle efficiently.

The performance drop from borders to arrows can be attributed to the more complex logic in the arrows shader. It draws both the line segment and the arrow tip using conditional statements, which are less favorable for GPU execution.

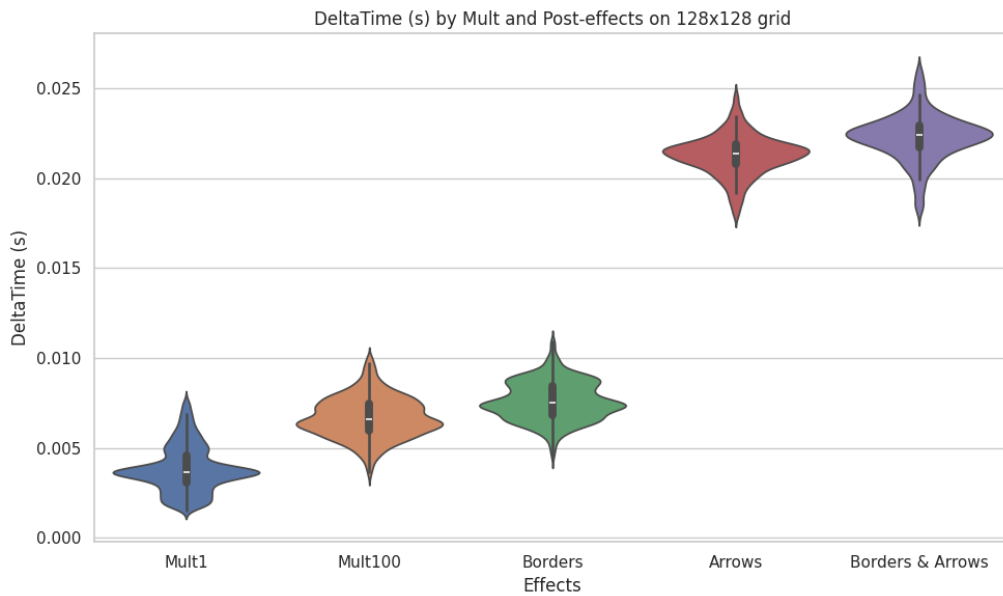


Figure 6.10: Effect of post-effects on frame execution time

6.2.6 Number of Objects

The user has the ability to add objects to the environment, such as the "ball" seen in the base simulation setup in Figure 6.4. In this test, the objective was to understand how the number of objects in a scene affects the overall performance of the simulation. To set up the test, the simulation was started in a wind tunnel configuration with no objects, and performance was measured. Then, objects were added one by one, with frame execution time measured after each addition. Each object was added using the *DRAW_BALL* function.

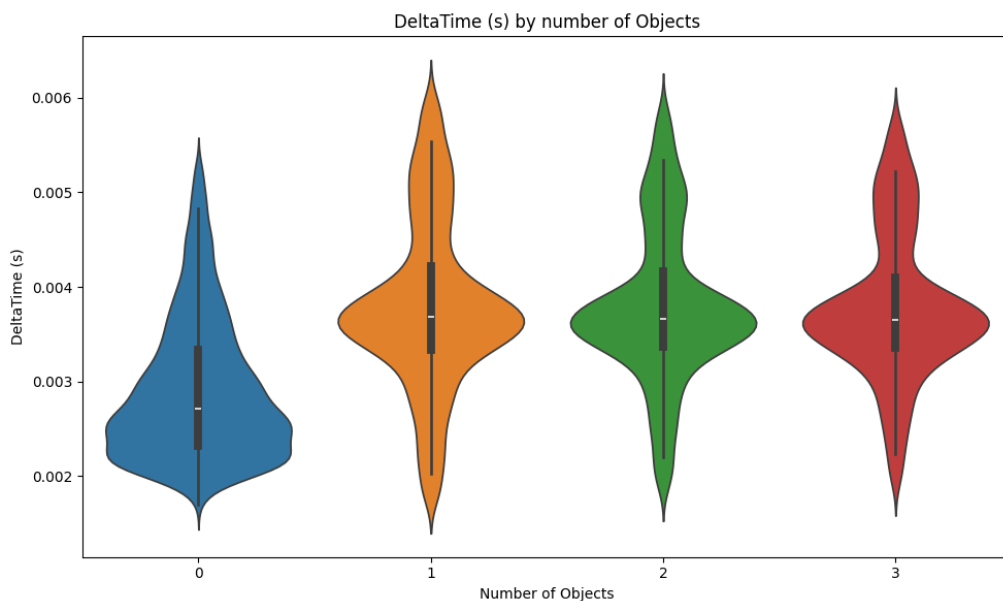


Figure 6.11: Effect of post-effects on frame execution time

From Figure 6.11, it appears that, aside from the case with zero objects, adding

Objects	Average Frame (ms)	Max Frame (ms)	Min Frame (ms)
0 Objects	3.33	25.54	1.70
1 Object	3.98	18.08	1.51
2 Objects	3.96	20.47	1.56
3 Objects	3.90	17.84	1.54

Table 6.3: Frame time statistics by number of objects

additional objects does not significantly impact performance. Which supported by the values shown in Table 6.3. The drop in frame execution time when simulating the wind tunnel with no objects is likely due to early stopping. Although early stopping did not prove faster in its dedicated test, in this scenario there are no objects causing turbulence, so the fluid is already incompressible. As a result, the projection cycle can terminate earlier.

6.2.7 Frame

The final test was conducted to determine how much of the frame execution time is dedicated to each simulation step. In this test each step: pressure reset, projection, velocity advection, smoke advection, merge objects and rendering was timed, over more than one thousand frames. Figure 6.12 shows the average time for each step.

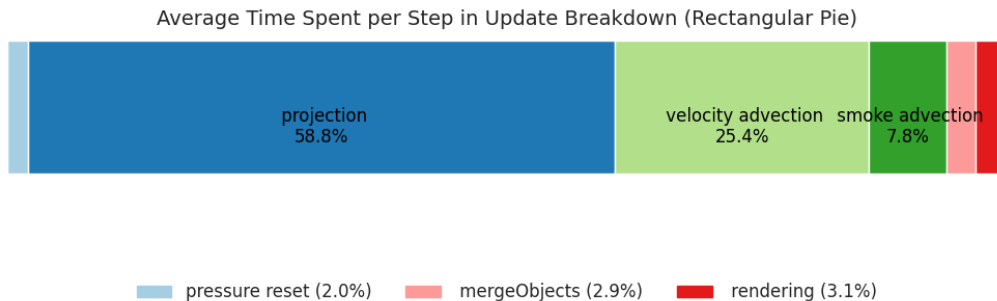


Figure 6.12: Time spent on each step of an frame

Step	Average Time (ms)
Pressure Reset	0.0078
Projection	0.2245
Velocity Advection	0.0969
Smoke Advection	0.0297
Merge Objects	0.0111
Rendering	0.0120

Table 6.4: Average time per simulation step

Figure 6.12 shows the percentage of time each step contributes to the total frame execution time, while Table 6.4 presents the exact values. Both clearly indicate that the

projection step is by far the most time-consuming, followed by velocity advection, which executes in less than half the time.

From these results, we can conclude that anyone aiming to improve the performance of the simulation should focus their efforts on optimizing the projection and advection steps.

6.2.8 Discussion

It is now important to take a comprehensive view of the project, considering the developed tool, the research questions and, especially for this discussion, the tests to which the tool was subjected.

The key findings extracted from the tests confirm that a physics-based fluid simulation can indeed be developed using a GPU, as proposed in the first research question. Furthermore, the fluid simulation can be considered suitable for a BUV simulation, as it maintains a high frame rate even on a large grid—running at nearly 60 FPS (approximately 0.017 seconds per frame) at a resolution of 512×512 , which is a common FPS benchmark for simulations.

Most of the tests produced expected results, with the exception of the early stopping feature, which proved to be unnecessary. To ensure that the measured values accurately reflect the tool's capabilities, each test was based on the average of a large number of measurements, always in the order of thousands, though the exact number varied. Despite efforts to ensure accuracy, one unavoidable source of error remains: the act of recording execution times introduces a small overhead. However, this error is minimal and considered insignificant.

All graphs exhibit a similar distribution, with most execution times clustering around the median, though a notable number of values fall significantly above or below it. Some of this variation stems from the algorithm itself, which contains branching logic that leads to different execution paths depending on the initial state. Additionally, system resource availability contributes to performance variability, since the simulation runs alongside other system processes, even if just basic ones. The distribution of available resources likely accounts for the most extreme values, which, as previously noted, were removed from the dataset to improve chart readability. This further underscores the importance of using thousands of samples to obtain an accurate representation of execution times.

These tests cover the currently implemented features. Future developments should also be evaluated to determine how they affect simulation performance and to ensure the system maintains the necessary efficiency for its ultimate goal: training a reinforcement learning algorithm to control a BUV. For instance, the algorithm responsible for moving objects (such as the fin) based on water forces and propulsion dynamics will likely have a significant impact on performance. Additionally, further improvements to the simulation of a flexible fin, used in the SABUVIS prototype, will also influence overall performance.

CONCLUSIONS

This thesis presented the development and evaluation of a fluid simulation tool designed to support the SABUVIS project, an initiative aimed at advancing research and deployment of biomimetic underwater vehicles (BUVs) across Europe. The simulation serves as a foundational component for future reinforcement learning systems, enabling autonomous agents to optimize their movement through fluid environments.

The tool was developed using the Unity3D game engine. During its development, High Level Shader Language (HLSL) was leveraged to accelerate part of the computation on the GPU, specifically the calculations for the next state of the simulation. The simulation was thoroughly tested for both correctness and performance. Special attention was given to ensuring that it remained efficient under varying conditions, including different resolutions, rendering modes, and post-processing effects.

Testing confirmed that the simulation behaves as expected, accurately modeling fluid interactions with fixed objects. Its performance was also found to be promising, potentially enabling the training of reinforcement learning systems.

These tests allowed us to address the research questions posed in this thesis:

- **First, how to develop a physics-inspired fluid simulation?** This was achieved using an Eulerian (grid-based) algorithm, demonstrating that such a simulation is indeed feasible.
- **Second, how can the fluid simulation be used to simulate interaction with objects?** The results confirmed this, as simulations such as a water-filled "wind tunnel" with an obstacle produced the expected behaviour, including vortices downstream of the object.
- **Third, is it feasible to develop a BUV simulator with reasonable computational requirements using HLSL shaders?** While this question is more difficult to answer definitively—since the prototype has not yet been tested with a full BUV model—the knowledge gained suggests a positive outlook. The use of HLSL enabled a highly performant prototype, which is likely to scale effectively to more complex scenarios such as full BUV simulation.

In conclusion, this thesis successfully delivered a robust and adaptable fluid simulation tool tailored to the needs of the SABUVIS project. While certain features require further refinement to achieve the ultimate goal, the current implementation meets its core objectives and lays the groundwork for future advancements in autonomous underwater navigation, control, and simulation-based learning.

7.1 Future Work

As this thesis forms part of the broader SABUVIS project, it is expected to be extended in future stages. The ultimate goal is to develop a fully integrated fluid simulation environment in which control mechanisms for biomimetic underwater vehicles (BUVs) can be trained using reinforcement learning.

The next logical step is the implementation of **movable objects**, which is essential for simulating the dynamics of a BUV navigating through fluid. This will allow the simulation to respond to object motion and enable realistic feedback for learning algorithms. Following this, the addition of **deformable objects** will be necessary to accurately model flexible components such as bendable fins, which are a key feature of current BUV prototypes.

Once these physical capabilities are in place, the final milestone will be the **design and integration of a reinforcement learning algorithm**. This will allow autonomous agents to interact with the fluid environment, learn control strategies, and optimize movement through simulation-based training.

Together, these extensions will transform the current prototype into a comprehensive simulation platform capable of supporting advanced research in underwater robotics, control systems, and bio-inspired design.

BIBLIOGRAPHY

- [1] https://personalpages.manchester.ac.uk/staff/matthias.heil/Lectures/Fluids/Material_2019/Chapter4.pdf. [Accessed 23-05-2024] (cit. on p. 9).
- [2] C. A. Aubin, S. Choudhury, R. Jerch, L. A. Archer, J. H. Pikul, and R. F. Shepherd. “Electrolytic vascular systems for energy-dense robots”. In: *Nature* 571 (7763 2019). ISSN: 14764687. DOI: [10.1038/s41586-019-1313-1](https://doi.org/10.1038/s41586-019-1313-1) (cit. on pp. 1, 2, 21).
- [3] J. O. Awoyemi, A. O. Adetunmbi, and S. A. Oluwadare. “Credit card fraud detection using machine learning techniques: A comparative analysis”. In: vol. 2017-January. 2017. DOI: [10.1109/ICCNI.2017.8123782](https://doi.org/10.1109/ICCNI.2017.8123782) (cit. on p. 14).
- [4] C. Badue et al. *Self-driving cars: A survey*. 2021. DOI: [10.1016/j.eswa.2020.113816](https://doi.org/10.1016/j.eswa.2020.113816) (cit. on p. 14).
- [5] M. J. Bocus, A. Doufexi, and D. Agrafiotis. “Performance of OFDM-based massive MIMO OTFS systems for underwater acoustic communication”. In: *IET Communications* 14 (4 2020). ISSN: 17518628. DOI: [10.1049/iet-com.2019.0376](https://doi.org/10.1049/iet-com.2019.0376) (cit. on p. 19).
- [6] C. Braley and A. Sandu. “Fluid Simulation For Computer Graphics : A Tutorial in Grid Based and Particle Based Methods”. In: *Computer* 0 (1 2009) (cit. on p. 11).
- [7] R. Bridson. *Fluid simulation for computer graphics, Second Edition*. Book on fluid simulation. 2015 (cit. on pp. 9, 13).
- [8] L. Brunke, M. Greeff, A. W. Hall, Z. Yuan, S. Zhou, J. Panerati, and A. P. Schoellig. *Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning*. 2022. DOI: [10.1146/annurev-control-042920-020211](https://doi.org/10.1146/annurev-control-042920-020211) (cit. on p. 14).
- [9] S. L. Brunton, B. R. Noack, and P. Koumoutsakos. *Machine Learning for Fluid Mechanics*. 2020. DOI: [10.1146/annurev-fluid-010719-060214](https://doi.org/10.1146/annurev-fluid-010719-060214) (cit. on pp. 14, 17).
- [10] cadence. *Cadence CFD for Aerospace and defense*. https://www.cadence.com/en_US/home/tools/system-analysis/computational-fluid-dynamics/aerospace-and-defense.html. [Accessed 25-06-2024] (cit. on p. 9).

- [11] J. Chen, J. Viquerat, and E. Hachem. “U-net architectures for fast prediction of incompressible laminar flows”. In: *arXiv preprint arXiv:1910.13532* (2019) (cit. on p. 15).
- [12] L. W. Chen and N. Thuerey. “Towards high-accuracy deep learning inference of compressible flows over aerofoils”. In: *Computers and Fluids* 250 (2023). ISSN: 00457930. DOI: [10.1016/j.compfluid.2022.105707](https://doi.org/10.1016/j.compfluid.2022.105707) (cit. on p. 7).
- [13] Z. Chen, J. Yan, B. Ma, K. Shi, Q. Yu, and W. Yuan. *A Survey on Open-Source Simulation Platforms for Multi-Copter UAV Swarms*. 2023. DOI: [10.3390/robotics12020053](https://doi.org/10.3390/robotics12020053) (cit. on p. 20).
- [14] R. J. Clapham and H. Hu. “iSplash: realizing fast Carangiform swimming to outperform a real fish”. In: vol. 12. 2015. DOI: [10.1007/978-3-662-46870-8_7](https://doi.org/10.1007/978-3-662-46870-8_7) (cit. on pp. 1, 2, 21).
- [15] J. Craighead, R. Murphy, J. Burke, and B. Goldiez. “A survey of commercial and open source unmanned vehicle simulators”. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. 2007, pp. 852–857. DOI: [10.1109/ROBOT.2007.363092](https://doi.org/10.1109/ROBOT.2007.363092) (cit. on pp. 20, 26).
- [16] G. de Cubber, H. Balta, and C. Lietart. “Teodor: A Semi-Autonomous Search and Rescue and Demining Robot”. In: *Advanced Concepts in Mechanical Engineering I*. Vol. 658. Applied Mechanics and Materials. Trans Tech Publications Ltd, 2014–11, pp. 599–605. DOI: [10.4028/www.scientific.net/AMM.658.599](https://doi.org/10.4028/www.scientific.net/AMM.658.599) (cit. on p. 18).
- [17] R. Culkin and S. R. Das. “Machine Learning in Finance: The Case of Deep Learning for Option Pricing”. In: *Journal of Investment Management* 15 (4 2017) (cit. on p. 14).
- [18] W. J. Dally, S. W. Keckler, and D. B. Kirk. “Evolution of the Graphics Processing Unit (GPU)”. In: *IEEE Micro* 41 (6 2021), pp. 42–51. ISSN: 19374143. DOI: [10.1109/MM.2021.3113475](https://doi.org/10.1109/MM.2021.3113475) (cit. on p. 44).
- [19] R. C. Deo. “Machine learning in medicine”. In: *Circulation* 132 (20 2015). ISSN: 15244539. DOI: [10.1161/CIRCULATIONAHA.115.001593](https://doi.org/10.1161/CIRCULATIONAHA.115.001593) (cit. on p. 14).
- [20] Y. Du and T. A. Zaki. “Evolutional deep neural network”. In: *Physical Review E* 104 (4 2021). ISSN: 24700053. DOI: [10.1103/PhysRevE.104.045303](https://doi.org/10.1103/PhysRevE.104.045303) (cit. on p. 16).
- [21] H. Eivazi, M. Tahani, P. Schlatter, and R. Vinuesa. “Physics-informed neural networks for solving Reynolds-averaged Navier-Stokes equations”. In: *Physics of Fluids* 34 (7 2022). ISSN: 10897666. DOI: [10.1063/5.0095270](https://doi.org/10.1063/5.0095270) (cit. on pp. 14, 17).
- [22] J. L. Elman. “Finding structure in time”. In: *Cognitive Science* 14 (2 1990). ISSN: 03640213. DOI: [10.1016/0364-0213\(90\)90002-E](https://doi.org/10.1016/0364-0213(90)90002-E) (cit. on p. 15).
- [23] A. Finn and S. Scheduling. “Developments and Challenges for Autonomous Unmanned Vehicles: A Compendium”. In: *Intelligent Systems Reference Library* 3 (2010). ISSN: 18684394. DOI: [10.1007/978-3-642-10704-7](https://doi.org/10.1007/978-3-642-10704-7) (cit. on pp. 18, 19).

- [24] N. Foster and R. Fedkiw. "Practical animation of liquids". In: 2001. DOI: [10.1145/383259.383261](https://doi.org/10.1145/383259.383261) (cit. on p. 13).
- [25] S. Fotiadis, E. Pignatelli, M. L. Valencia, C. Cantwell, A. Storkey, and A. A. Bharath. "Comparing recurrent and convolutional neural networks for predicting wave propagation". In: *arXiv preprint arXiv:2002.08981* (2020) (cit. on p. 15).
- [26] X. Guo, W. Li, and F. Iorio. "Convolutional neural networks for steady flow approximation". In: vol. 13-17-August-2016. 2016. DOI: [10.1145/2939672.2939738](https://doi.org/10.1145/2939672.2939738) (cit. on pp. 14, 15, 17).
- [27] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: vol. 2016-December. 2016. DOI: [10.1109/CVPR.2016.90](https://doi.org/10.1109/CVPR.2016.90) (cit. on p. 15).
- [28] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2 (5 1989). ISSN: 08936080. DOI: [10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8) (cit. on p. 16).
- [29] C. M. Institute. *EXISTENCE AND SMOOTHNESS OF THE NAVIER-STOKES EQUATION*. <https://www.claymath.org/wp-content/uploads/2022/06/navierstokes.pdf>. [Accessed 10-05-2024] (cit. on p. 8).
- [30] J. J.D. Anderson. *Governing Equations of Fluid Dynamics*. <https://www.eng.auburn.edu/~tplacek/courses/fluidsreview-1.pdf>. Online; accessed 20-06-2024. 2013 (cit. on pp. 7, 8).
- [31] A. S. Jaya and M. W. Kartidjo. "Thrust and efficiency enhancement scheme of the fin propulsion of the biomimetic Autonomous Underwater Vehicle model in low-speed flow regime". In: *Ocean Engineering* 243 (2022). ISSN: 00298018. DOI: [10.1016/j.oceaneng.2021.110090](https://doi.org/10.1016/j.oceaneng.2021.110090) (cit. on pp. 3, 21).
- [32] J. Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596 (7873 2021). ISSN: 14764687. DOI: [10.1038/s41586-021-03819-2](https://doi.org/10.1038/s41586-021-03819-2) (cit. on p. 23).
- [33] P. Katara, M. Khanna, H. Nagar, and A. Panaiyappan. "Open Source Simulator for Unmanned Underwater Vehicles using ROS and Unity3D". In: 2019. DOI: [10.1109/UT.2019.8734309](https://doi.org/10.1109/UT.2019.8734309) (cit. on p. 19).
- [34] E. Kharazmi, Z. Zhang, and G. E. Karniadakis. "Variational physics-informed neural networks for solving partial differential equations". In: *arXiv preprint arXiv:1912.00873* (2019) (cit. on p. 16).
- [35] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: vol. 3. 2004. DOI: [10.1109/iros.2004.1389727](https://doi.org/10.1109/iros.2004.1389727) (cit. on p. 27).
- [36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86 (11 1998). ISSN: 00189219. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791) (cit. on p. 15).

-
- [37] J. H. Li, M. G. Kim, H. Kang, M. J. Lee, and G. R. Cho. “UUV simulation modeling and its control method: Simulation and experimental studies”. In: *Journal of Marine Science and Engineering* 7 (4 2019). ISSN: 20771312. DOI: [10.3390/jmse7040089](https://doi.org/10.3390/jmse7040089) (cit. on p. 20).
 - [38] W. Li, Y. Chen, M. Desbrun, C. Zheng, and X. Liu. “Fast and scalable turbulent flow simulation with two-way coupling”. In: *ACM Transactions on Graphics* 39 (4 2020). ISSN: 15577368. DOI: [10.1145/3386569.3392400](https://doi.org/10.1145/3386569.3392400) (cit. on p. 22).
 - [39] M. Lino, S. Fotiadis, A. A. Bharath, and C. D. Cantwell. “Current and emerging deep-learning methods for the simulation of fluid dynamics”. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 479 (2275 2023). ISSN: 14712946. DOI: [10.1098/rspa.2023.0058](https://doi.org/10.1098/rspa.2023.0058) (cit. on pp. 14, 16).
 - [40] M. Lino, S. Fotiadis, A. A. Bharath, and C. D. Cantwell. “Multi-scale rotation-equivariant graph neural networks for unsteady Eulerian fluid dynamics”. In: *Physics of Fluids* 34 (8 2022). ISSN: 10897666. DOI: [10.1063/5.0097679](https://doi.org/10.1063/5.0097679) (cit. on p. 15).
 - [41] W. Liu, K. Bai, X. He, S. Song, C. Zheng, and X. Liu. “FishGym: A High-Performance Physics-based Simulation Framework for Underwater Robot Learning”. In: 2022. DOI: [10.1109/ICRA46639.2022.9812066](https://doi.org/10.1109/ICRA46639.2022.9812066) (cit. on pp. 22–25).
 - [42] J. M. Lourenço. *The NOVAthesis L^AT_EX Template User’s Manual*. NOVA University Lisbon. 2021. URL: <https://github.com/joaomlourenco/novathesis/raw/main/template.pdf> (cit. on p. i).
 - [43] M. Lyu, Y. Zhao, C. Huang, and H. Huang. *Unmanned Aerial Vehicles for Search and Rescue: A Survey*. 2023. DOI: [10.3390/rs15133266](https://doi.org/10.3390/rs15133266) (cit. on p. 17).
 - [44] M. Macklin and M. Müller. “Position based fluids”. In: *ACM Transactions on Graphics* 32 (4 2013). ISSN: 07300301. DOI: [10.1145/2461912.2461984](https://doi.org/10.1145/2461912.2461984) (cit. on p. 12).
 - [45] I. Mademlis, N. Nikolaidis, A. Tefas, I. Pitas, T. Wagner, and A. Messina. “Autonomous Unmanned Aerial Vehicles Filming in Dynamic Unstructured Outdoor Environments [Applications Corner]”. In: *IEEE Signal Processing Magazine* 36 (1 2019). ISSN: 15580792. DOI: [10.1109/MSP.2018.2875190](https://doi.org/10.1109/MSP.2018.2875190) (cit. on p. 17).
 - [46] S. McKee, M. F. Tomé, V. G. Ferreira, J. A. Cuminato, A. Castelo, F. S. Sousa, and N. Mangiavacchi. *The MAC method*. 2008. DOI: [10.1016/j.compfluid.2007.10.006](https://doi.org/10.1016/j.compfluid.2007.10.006) (cit. on pp. 12, 13).
 - [47] S. S. Mousavi, M. Schukat, and E. Howley. “Deep Reinforcement Learning: An Overview”. In: vol. 16. 2018. DOI: [10.1007/978-3-319-56991-8_32](https://doi.org/10.1007/978-3-319-56991-8_32) (cit. on p. 23).
 - [48] M. Müller, D. Charypar, and M. Gross. “Particle-based fluid simulation for interactive applications”. In: 2003 (cit. on p. 12).
 - [49] P. Natta, V. R. Kumar, and Y. V. H. Rao. “Flow Analysis of Rocket Nozzle Using Computational Fluid Dynamics (Cfd)”. In: *International Journal of Engineering Research and Applications (IJERA)* 2 (5 2012) (cit. on p. 7).

- [50] *Ocean economy offers a \$2.5 trillion export opportunity: UNCTAD report* — [unctad.org. https://unctad.org/news/ocean-economy-offers-25-trillion-export-opportunity-unctad-report](https://unctad.org/news/ocean-economy-offers-25-trillion-export-opportunity-unctad-report). [Accessed 10-05-2024] (cit. on p. 1).
- [51] A. Ollero and L. Merino. “Control and perception techniques for aerial robotics”. In: *Annual Reviews in Control* 28 (2 2004). ISSN: 13675788. DOI: [10.1016/j.arcontrol.2004.05.003](https://doi.org/10.1016/j.arcontrol.2004.05.003) (cit. on p. 18).
- [52] S. Park and Y. Choi. *Applications of unmanned aerial vehicles in mining from exploration to reclamation: A review*. 2020. DOI: [10.3390/min10080663](https://doi.org/10.3390/min10080663) (cit. on p. 17).
- [53] L. Paull, S. Saeedi, M. Seto, and H. Li. *AUV navigation and localization: A review*. 2014. DOI: [10.1109/JOE.2013.2278891](https://doi.org/10.1109/JOE.2013.2278891) (cit. on p. 1).
- [54] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. “LEARNING MESH-BASED SIMULATION WITH GRAPH NETWORKS”. In: 2021 (cit. on p. 15).
- [55] M. Prats, J. Perez, J. J. Fernandez, and P. J. Sanz. “An open source tool for simulation and supervision of underwater intervention missions”. In: 2012. DOI: [10.1109/IROS.2012.6385788](https://doi.org/10.1109/IROS.2012.6385788) (cit. on p. 27).
- [56] M. F. F. Rahman, S. Fan, Y. Zhang, and L. Chen. *A comparative study on application of unmanned aerial vehicle systems in agriculture*. 2021. DOI: [10.3390/agriculture11010022](https://doi.org/10.3390/agriculture11010022) (cit. on p. 17).
- [57] M. Raissi, P. Perdikaris, and G. E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019). ISSN: 10902716. DOI: [10.1016/j.jcp.2018.10.045](https://doi.org/10.1016/j.jcp.2018.10.045) (cit. on p. 16).
- [58] M. Raissi, A. Yazdani, and G. E. Karniadakis. “Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations”. In: *Science* 367 (6481 2020). ISSN: 10959203. DOI: [10.1126/science.aaw4741](https://doi.org/10.1126/science.aaw4741) (cit. on pp. 16, 70, 71).
- [59] A. Rajkomar, J. Dean, and I. Kohane. “Machine Learning in Medicine”. In: *New England Journal of Medicine* 380 (14 2019). ISSN: 0028-4793. DOI: [10.1056/nejmra1814259](https://doi.org/10.1056/nejmra1814259) (cit. on p. 14).
- [60] O. Ronneberger, P. Fischer, and T. Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: vol. 9351. 2015. DOI: [10.1007/978-3-319-24574-4_28](https://doi.org/10.1007/978-3-319-24574-4_28) (cit. on p. 15).
- [61] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia. “Graph networks as learnable physics engines for inference and control”. In: vol. 10. 2018 (cit. on p. 15).
- [62] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. “The graph neural network model”. In: *IEEE Transactions on Neural Networks* 20 (1 2009). ISSN: 10459227. DOI: [10.1109/TNN.2008.2005605](https://doi.org/10.1109/TNN.2008.2005605) (cit. on p. 15).

-
- [63] J. Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588 (7839 2020). ISSN: 14764687. DOI: [10.1038/s41586-020-03051-4](https://doi.org/10.1038/s41586-020-03051-4) (cit. on p. 14).
 - [64] L. Shi, S. Guo, S. Mao, C. Yue, M. Li, and K. Asaka. "Development of an amphibious turtle-inspired spherical mother robot". In: *Journal of Bionic Engineering* 10 (4 2013). ISSN: 16726529. DOI: [10.1016/S1672-6529\(13\)60248-6](https://doi.org/10.1016/S1672-6529(13)60248-6) (cit. on pp. 1, 21).
 - [65] D. Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362 (6419 2018). ISSN: 10959203. DOI: [10.1126/science.aar6404](https://doi.org/10.1126/science.aar6404) (cit. on p. 23).
 - [66] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (7587 2016). ISSN: 14764687. DOI: [10.1038/nature16961](https://doi.org/10.1038/nature16961) (cit. on p. 14).
 - [67] H. A. Soliman. "Investigation of process parameters in orthogonal cutting using finite element approaches". In: *Heliyon* 6 (11 2020) (cit. on p. 13).
 - [68] Z. Su, J. Yu, M. Tan, and J. Zhang. "Implementing flexible and fast turning maneuvers of a multijoint robotic fish". In: *IEEE/ASME Transactions on Mechatronics* 19 (1 2014). ISSN: 10834435. DOI: [10.1109/TMECH.2012.2235853](https://doi.org/10.1109/TMECH.2012.2235853) (cit. on p. 1).
 - [69] Y. Sun, Q. Sun, and K. Qin. "Physics-based deep learning for flow problems". In: *Energies* 14 (22 2021). ISSN: 19961073. DOI: [10.3390/en14227760](https://doi.org/10.3390/en14227760) (cit. on p. 14).
 - [70] M. S. Thordal, J. C. Bennetsen, and H. H. H. Koss. "Review for practical application of CFD for the determination of wind load on high-rise buildings". In: *Journal of Wind Engineering and Industrial Aerodynamics* 186 (2019). ISSN: 01676105. DOI: [10.1016/j.jweia.2018.12.019](https://doi.org/10.1016/j.jweia.2018.12.019) (cit. on pp. 6, 7).
 - [71] N. Thuerey, K. Weißenow, L. Prantl, and X. Hu. "Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows". In: *AIAA Journal* 58 (1 2020). ISSN: 1533385X. DOI: [10.2514/1.J058291](https://doi.org/10.2514/1.J058291) (cit. on p. 15).
 - [72] S. W. Tolkoff. "Robotics and power measurements of the RoboTuna". In: *Department of Ocean Engineering* (1999) (cit. on p. 1).
 - [73] U. K. Verfuss et al. *A review of unmanned vehicles for the detection and monitoring of marine fauna*. 2019. DOI: [10.1016/j.marpolbul.2019.01.009](https://doi.org/10.1016/j.marpolbul.2019.01.009) (cit. on pp. 17, 20).
 - [74] R. Wang, S. Wang, Y. Wang, L. Cheng, and M. Tan. "Development and Motion Control of Biomimetic Underwater Robots: A Survey". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 52 (2 2022). ISSN: 21682232. DOI: [10.1109/TSMC.2020.3004862](https://doi.org/10.1109/TSMC.2020.3004862) (cit. on pp. 1, 3).
 - [75] J. E. Welch, F. H. Harlow, J. P. Shannon, and B. J. Daly. *The MAC method-a computing technique for solving viscous, incompressible, transient fluid-flow problems involving free surfaces*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 1965 (cit. on p. 10).

- [76] J. Willard, X. Jia, M. Steinbach, V. Kumar, and S. Xu. “Integrating Physics-Based Modeling With Machine Learning: A Survey”. In: *ArXiv200304919 Phys. Stat* 1 (2020) (cit. on p. 14).
- [77] S. Yan, Z. Wu, J. Wang, Y. Huang, M. Tan, and J. Yu. “Real-World Learning Control for Autonomous Exploration of a Biomimetic Robotic Shark”. In: *IEEE Transactions on Industrial Electronics* 70 (4 2023). ISSN: 15579948. DOI: [10.1109/TIE.2022.3174306](https://doi.org/10.1109/TIE.2022.3174306) (cit. on pp. 22, 24, 25).
- [78] Z. Zhang, W. Mi, J. Du, Z. Wang, W. Wei, Y. Zhang, Y. Yang, and Y. Ren. “Design and Implementation of a Modular UUV Simulation Platform”. In: *Sensors (Basel, Switzerland)* 22 (20 2022). ISSN: 14248220. DOI: [10.3390/s22208043](https://doi.org/10.3390/s22208043) (cit. on p. 20).
- [79] C. Zhou and K. H. Low. “Design and Locomotion Control of a Biomimetic Underwater Vehicle With Fin Propulsion”. In: *IEEE/ASME Transactions on Mechatronics* 17 (1 2012-02), pp. 25–35. ISSN: 1083-4435. DOI: [10.1109/TMECH.2011.2175004](https://doi.org/10.1109/TMECH.2011.2175004). URL: <http://ieeexplore.ieee.org/document/6104154/> (cit. on pp. 1, 2, 21).
- [80] C. Zhou and K. H. Low. “Optimization of swimming locomotion for fish robots with multi-actuation”. In: 2011. DOI: [10.1109/ROBIO.2011.6181605](https://doi.org/10.1109/ROBIO.2011.6181605) (cit. on p. 1).

A

APPENDIX

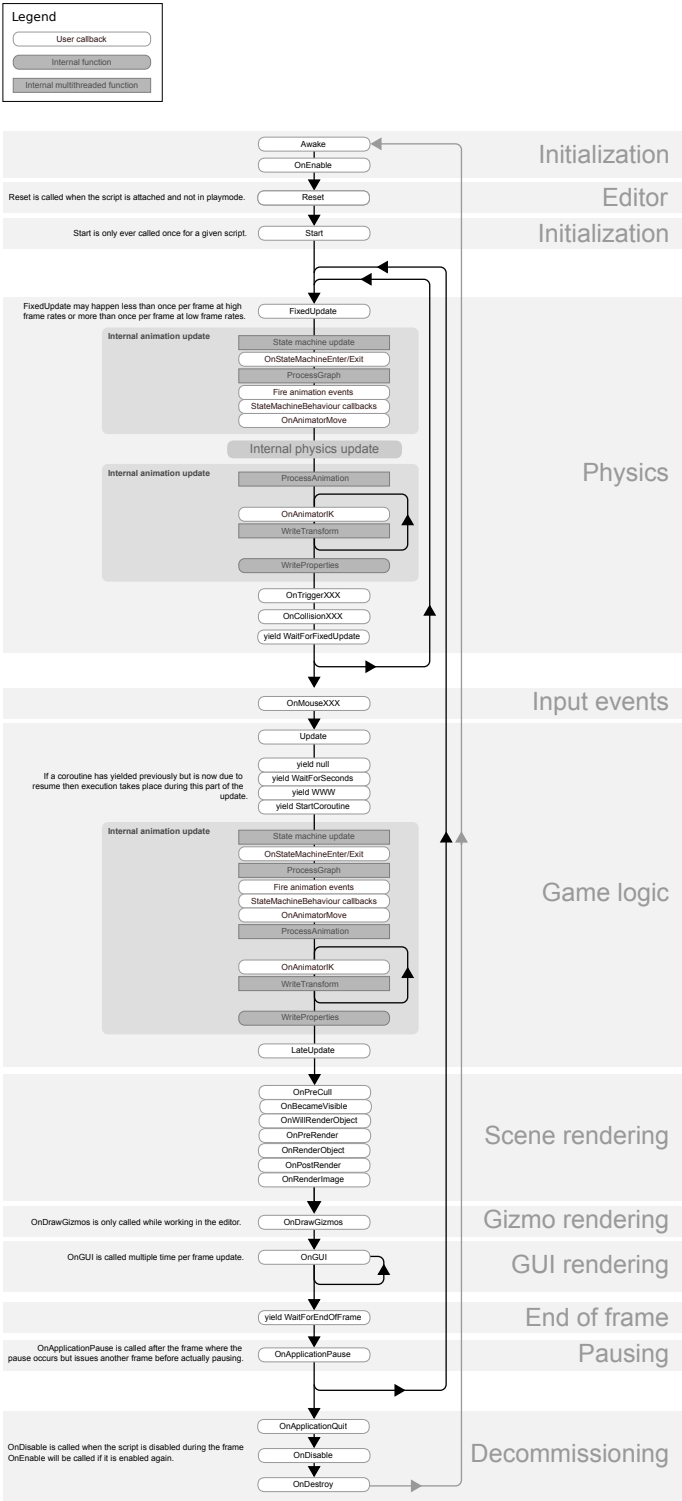


Figure A.1: Event functions loop in Unity





2025 Water simulation and optimization for biomimetic propulsion in underwater vehicles using HLSL

