A Reinforcement Learning Approach to Optimizing Autonomous Kite-Powered Vessel Control

A Novel Approach

By

JOSHUA CAREY

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of MASTERS OF COMPUTER SCIENCE in the Faculty of Engineering.

SEPTEMBER 2023

Word count: ten thousand and four

Abstract

Here goes the abstract

Dedication and acknowledgements

Here goes the dedication.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE: ....

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction and Motivation

Maritime travel has been a cornerstone of human civilization, facilitating the exchange of goods, ideas, and cultures across vast expanses of water. The annals of history are full with instances of seafaring civilizations harnessing the power of wind to propel their vessels across the oceans. It is posited that ancient Neanderthals embarked on maritime voyages in the southern Ionian Islands between 110 to 35ka BP [1]. The quintessence of maritime travel has predominantly been wind-powered sails, which remained unchallenged until the industrial revolution ushered in the era of fuel-powered engines.

The art and science of sailing have evolved significantly over millennia, from rudimentary rafts and canoes to sophisticated sailing ships with complex rigging systems. Ancient civilizations, including the Egyptians, Phoenicians, and Polynesians, made remarkable advancements in sailing technology, enabling them to explore and trade over larger swathes of the ocean [2]. The medieval period saw the advent of the compass and the astrolabe, which further facilitated maritime navigation and exploration. The Age of Discovery, epitomized by the voyages of Columbus, Vasco da Gama, and Magellan, was propelled by advancements in sailing technology, which enabled transoceanic voyages and the establishment of maritime empires.

The industrial revolution in the 18th and 19th centuries marked a significant turning point in maritime propulsion. The invention of the steam engine heralded the decline of wind-powered sailing and the rise of fuel-powered propulsion systems. Steam-powered ships and later, diesel-powered ships, offered greater reliability, speed, and capacity compared to their wind-powered predecessors, thus becoming the preferred mode of maritime transportation [3]. The transition to fuel-powered engines also mirrored the broader industrial and technological advancements of the era, which prioritized speed, efficiency and profit over traditional methods.

1.1 A Renewed Interest in Wind Propulsion

However, the environmental costs of fuel-powered maritime transportation have become increasingly apparent in the modern era. The shipping industry is a notable contributor to global carbon emissions, and the negative effects of pollution on marine ecosystems around the world are well-documented [4]. These challenges have rekindled interest in wind propulsion as a sustainable alternative, prompting a re-examination of the principles that guided ancient and medieval sailors. The modern iteration of wind propulsion seeks to amalgamate the age-old wisdom of harnessing wind power with contemporary technological advancements to create eco-friendly and efficient maritime transportation systems.

Contemporary wind propulsion technologies like Flettner rotors [5], wing sails, and kite systems are being revisited to mitigate the environmental impact of maritime travel. Among these, kite-powered vessel technology stands out due to its potential for higher efficiency and ease of retrofitting. Kites offer two main advantages over traditional sails: they can move relative to the vessel, generating their own apparent wind and can be flown at higher altitudes, accessing different wind systems.

The relative movement of kites generates apparent wind, allowing for maximum potential force even when the vessel is stationary. This enhanced apparent wind results in a larger force compared to a sail of equivalent area. Flying kites at higher altitudes taps into different wind systems and currents to those at sea level, making wind a potentially more reliable energy source for propulsion.

However, the effective operation of kite-powered vessels requires precise control, which is skill-intensive. To leverage the full benefits of kites as a scalable propulsion method, implementing autonomous control is crucial.

Reinforcement Learning (RL), a subset of artificial intelligence, presents a compelling avenue for optimizing the autonomous control of kite-powered vessels. RL, which is based on learning through interaction with an environment, offers a potential for developing advanced control systems and strategies that could greatly improve the effectiveness of kite-powered vessels.

1.2 Aims and Objectives

The overarching aim of this research is to develop a robust Reinforcement Learning (RL) algorithm capable of autonomously controlling a kite-powered vessel. This objective stems from the need to advance sustainable maritime travel technologies and reduce the environmental impact of current propulsion systems.

To achieve this primary aim, the objectives have been structured as follows:

Objective 1: Simulation Environment Development

- To design and implement a virtual marine environment that accurately emulates real-world maritime conditions.
- To construct a realistic model of a boat that exhibits appropriate physical movements in response to environmental forces such as wind and water currents.
  Outcome Goal: To have a physics-based boat able to be controlled and driven around a scene by a human player.

Objective 2: Kite Propulsion Modeling

- To create a physics-based model of a kite within the simulation that reflects authentic aerodynamic behaviors and integrate it onto the boat model.
- To integrate kite control mechanics into the agent's available actions, allowing for simulated propulsion through wind energy harnessing.
  Outcome Goal: To have a physics-based kiteboat able to be controlled and driven around a scene by a human player, using the agent's heuristic controls.

Objective 3: Reinforcement Learning Framework Establishment

- To formulate a set of observations, actions, and rewards that encapsulate the dynamics of autonomous kite-boat navigation.
- To deploy the Proximal Policy Optimization algorithm, leveraging its actor-critic method for effective policy learning.
  Outcome Goal: To have an agent begin training using PPO to learn to control the kiteboat.

Objective 4: Autonomous Agent Development

- To develop an RL agent capable of learning basic control and maneuvers, starting with simple navigating towards a target and maintaining a constant course.
- To refine the agent's capability to adaptively control the kite's position and angle to optimize propulsion for speed while navigating towards a target in any direction.
  Outcome Goal 1: To have an agent that can navigate towards a target in a straight line.
  Outcome Goal 2: To have an agent that can navigate towards a target in any direction, including using maneuvers to take the fastest path.

Objective 5: Efficacy and Optimization Testing

- To utilize High-Performance Computing (HPC) resources for scaling up simulations and optimizing the training process.

- To rigorously evaluate the trained agent's performance in simulating autonomous navigation in various environmental scenarios.
  Outcome Goal 1: To train an agent using the HPC resources.
  Outcome Goal 2: To have an agent that can navigate towards a target in a straight line under various environmental conditions, including wind and waves.

Objective 6: Real-World Applicability Assessment

- To extrapolate simulation findings to assess real-world applicability and propose a practical deployment of RL in kite-powered vessels.
- To provide recommendations for further research and development based on empirical results obtained from the simulation studies.
  Outcome Goal 1: To propose a practical deployment of RL in kite-powered vessels.
  Outcome Goal 2: To provide recommendations for further research and development.

These objectives pave the path towards achieving the central goal, ensuring each phase of development builds upon the last. By concluding this investigation with assessments geared towards real-world implementation, it is anticipated that the contributions made will significantly impact sustainable maritime transportation solutions.

Chapter 2

Background

The maritime domain has long been a focal point of innovation and technological advancement. Historically, propulsion mechanisms have evolved from rudimentary oars to sophisticated sails, each iteration seeking to harness nature's forces more efficiently. Today, as we stand at the intersection of technology and tradition, there emerges a compelling avenue for exploration: kite-powered vessels. This innovative approach to propulsion seeks to leverage the aerodynamic advantages of kites, offering potential enhancements in efficiency and maneuverability over traditional sails.

However, the introduction of kites as a propulsion mechanism brings forth a new set of challenges. The dynamic nature of kites, combined with the unpredictable marine environment, necessitates advanced control systems capable of real-time adaptation and decision-making. This is where the application of machine learning, and more specifically Reinforcement Learning (RL), becomes paramount. An RL agent learns to make decisions that maximize a certain objective, often framed as a cumulative reward. The potential of RL in maritime propulsion is evident: it offers a framework for developing control systems that can adapt to changing conditions and environments.

This chapter aims to provide an in depth background into the technologies that will be leveraged in this thesis. This includes a detailed examination of the core concepts of RL, the mechanisms behind the PPO algorithm, and the Unity game engine. We will also explore the current state of kite-powered vessel technologies, identify gaps in existing research, and highlight the novelty and potential contributions of the proposed work.

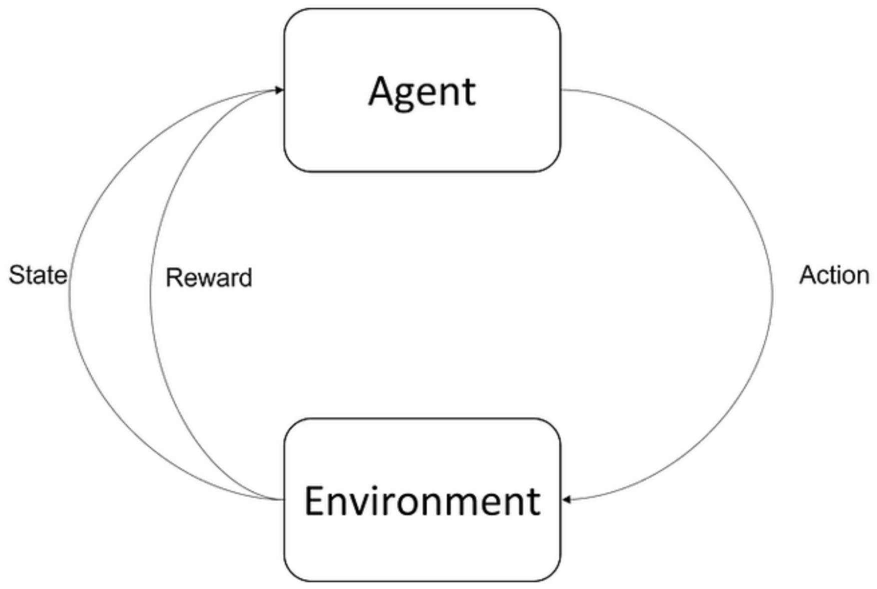

Figure 2.1: A diagram of the RL Loop

2.1 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a paradigm of machine learning that has been making waves, both literally and figuratively, in the vast sea of artificial intelligence (AI). At its core, RL is about learning by interaction: an agent takes actions in an environment to maximize some notion of cumulative reward. The agent learns from the consequences of its actions, rather than from being explicitly taught, making it a powerful tool for tasks where the optimal strategy is unknown or hard to define [6].

Imagine teaching a child to ride a bicycle. You don't provide a step-by-step manual; instead, the child learns by trying different actions (like pedaling or balancing) and receiving feedback (falling down or moving forward). This trial-and-error approach is the essence of RL. The agent (in this case, the child) interacts with its environment (the bicycle and the ground) and learns a policy that dictates the best action to take in any given situation based on the rewards (or penalties) it receives [7]. Figure 2.1 illustrates this process.

Historically, RL has its roots in the fields of operations research and behavioral psychology. The idea of learning optimal strategies through interaction has been explored in various contexts, from game playing to industrial optimization [8]. However, it's the recent advancements in computational power and algorithms that have propelled RL to the forefront of AI research. Games like Go, which were once considered too complex for computers to master, have now been conquered by RL agents, showcasing the immense potential of this approach[9].

Now, to explore how RL can be applied to the maritime world. Boats, with their intricate dynamics and the unpredictable nature of water, present a challenging environment for control systems. Traditional control methods often rely on predefined rules and heuristics, which might not always be optimal, especially in changing conditions. These autonomous controls vary from something as simple as a piece of bungee to PID controllers to more complex systems like Model Predictive Control (MPC). Enter RL. With its ability to learn from experience, an RL-based control system can adapt to varying conditions, ensuring smooth sailing even in turbulent waters, gusty winds [10].

But why stop at boats? The concept of using kites to harness wind power for propulsion is not new. Historically, kites have been used in various cultures for fishing, transportation, and even warfare [11]. In the modern context, kites offer an exciting alternative to traditional sails, providing more power and maneuverability. However, controlling a kite, especially in varying wind conditions, is a complex task. Kite control has only been explored in recent years, primarily in the field of renewable energy, where large ram-air kites are used to harness wind power for electricity generation [12]. However these problems are static and do not handle situations where the base of the kite is moving. This is where RL shines. By continuously interacting with the environment and adjusting the kite's position and angle, an RL agent can learn the optimal control strategy to harness the maximum wind power, propelling the boat efficiently [13].

The potential applications of RL in marine navigation are vast. From optimizing routes for cargo ships to ensuring safe navigation in crowded ports, the possibilities are as vast as the open sea. Moreover, as environmental concerns become more pressing, the need for efficient and sustainable maritime solutions becomes paramount. RL, with its ability to optimize and adapt, can play a pivotal role in addressing these challenges [14].

In conclusion, Reinforcement Learning is not just another tool in the AI toolkit; it's a paradigm shift in how we approach problem-solving. Its potential in the maritime world is just beginning to be tapped. As we venture into the future, with boats steered by intelligent agents and sails replaced by kites controlled with precision, it's clear that RL will be at the helm, guiding us towards uncharted territories and new horizons[15].

2.2 Proximal Policy Optimization (PPO)

Reinforcement Learning (RL) has witnessed a plethora of algorithms, each striving to optimize policy in its unique way. Among these, the Proximal Policy Optimization (PPO) algorithm stands out as a beacon of efficiency and simplicity [16].

PPO is a member of the policy gradient family of RL algorithms. Unlike traditional policy gradient methods that perform a single gradient update per data sample, PPO introduces a "surrogate" objective function. This novel approach allows for multiple epochs of minibatch updates, optimizing the policy over a series of iterations. The essence of PPO lies in its ability to alternate between sampling data through interaction with the environment and optimizing the surrogate objective using stochastic gradient ascent.

The inception of PPO was driven by the need for an algorithm that combined the best of all worlds: scalability, data efficiency, and robustness. While deep Q-learning and vanilla policy gradient methods have their merits, they often fall short in terms of data efficiency and robustness. Trust Region Policy Optimization (TRPO), on the other hand, although effective, is relatively intricate and lacks compatibility with certain architectures [17].

PPO seeks to bridge these gaps. It aims to achieve the data efficiency and consistent performance of TRPO but does so using only first-order optimization. The brilliance of PPO is encapsulated in its objective with clipped probability ratios. This objective provides a pessimistic estimate (or a lower bound) of the policy's performance. The optimization process in PPO is iterative, alternating between data sampling from the policy and performing several epochs of optimization on this sampled data.

Empirical evidence underscores the efficacy of PPO. When pitted against various versions of the surrogate objective, PPO, with its clipped probability ratios, emerges as the top performer. Furthermore, in head-to-head comparisons with other algorithms, PPO shines brightly. On continuous control tasks, PPO outperforms its competitors. In the realm of Atari games, PPO showcases superior sample complexity compared to A2C and performs on par with ACER, all while maintaining a simpler architecture.

The key characteristic of PPO is that it is an on-policy algorithm, meaning that it learns from the most recent experiences and that it uses an 'actor-critic' method. The actor-critic method is a combination of two neural networks, the actor and the critic. The actor is responsible for learning the optimal policy, while the critic is responsible for evaluating the actions of the actor by estimating the value function.

The actor-critic method can be broken down into the following steps:

1. **Actor**: The actor network proposes an action given in the current state. The action is drawn from a probability distribution (the policy π) parameterized by the networks weights.

2. **Critic**: The critic network estimates the value function $V(s)$ $V(s)$, which predicts the expected return (sum of future rewards) from state s under the current policy.

3. **Advantage Estimation**: The advantage function $A(s,a)$ quantifies how much better taking a particular action aa is, compared to the average action in state ss, and is computed as

   $$A(s,a) = Q(s,a) - V(s) \tag{2.1}$$

   where $Q(s,a)$ is the action-value function, which is the expected return after taking action a in state s.

4. **Objective Function**: PPO optimizes a clipped surrogate objective function to prevent large policy updates, which could lead to performance collapse. The objective function L^{CLIP} is defined as

   $$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.2}$$

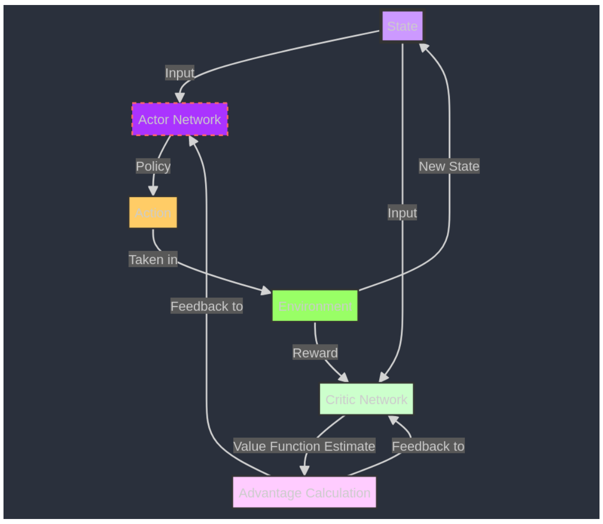

Figure 2.2: Actor-Critic Method

were $r_t(\theta)$ is the probability ratio between the new and old policy, $\hat{A}_t$ is the advantage at time step t, and ϵ is a hyperparameter that controls the size of the policy update.

5. **Policy Update**: PPO uses this objective to update the actor network's weights, maximizing the expected return while avoiding too large policy updates.

6. **Value Function Loss**: The critic network is trained to minimize the value function loss, which is typically the Mean Squared Error between the estimated value function V(s) and the observed return R.

7. **Entropy Bonus**: To encourage exploration, PPO adds an entropy bonus to the objective function, which promotes diversity in the action distribution.

PPO iterates between sampling data through interaction with the environment and optimizing the clipped objective function using stochastic gradient ascent. This optimization is typically done using minibatch updates for efficiency.

By employing PPO, the agent learns to balance exploration (trying new actions) with exploitation (taking known rewarding actions), which is particularly effective for complex tasks like sailing a kiteboat where the agent must adapt to dynamic conditions and long-term consequences of actions.

But the story doesn't end with PPO alone. Unity's ML-Agents toolkit, which is introduced in section 2.3, seamlessly integrates with PPO. ML-Agents provides a platform for training intelligent agents within the Unity environment, and when combined with the power of PPO, it paves the way for robust and efficient training regimes. This synergy between PPO and ML-Agents is particularly promising for complex simulations, such as kiteboat training, where agents can iteratively learn and refine their strategies for optimal performance.

The Proximal Policy Optimization algorithm is a testament to the continuous evolution and innovation in the field of Reinforcement Learning. Its simplicity, efficiency, and robustness make it a prime choice for a myriad of applications. As we harness the combined power of PPO with Unity's ML-Agents for kiteboat simulations.

2.3 Unity Game Engine

Unity, a name that resonates with game developers and enthusiasts alike, stands as a beacon in the realm of game development. Born in the vibrant city of Copenhagen, Denmark, in 2005, Unity has since evolved into a powerhouse, democratizing game development and breathing life into iconic games like 'Among Us' and 'Pokemon Go' [18].

At its heart, Unity is a cross-platform game engine designed to craft both 2D and 3D experiences. It offers a harmonious blend of a powerful graphical editor and the flexibility of `C#` coding, allowing developers to translate their visions into virtual realities [19]. While the engine's core is written in C++, it graciously opens its arms to developers familiar with `C#`, making the development process both intuitive and efficient.

Diving into the basics of Unity game development, one is greeted with a plethora of tools and components that simulate real-world interactions. Unity's lighting, physics, rigidbody, and colliders work in tandem to create immersive environments. Whether it's the glint of sunlight reflecting off a surface or the realistic bounce of a ball, Unity ensures every detail is just right [20]. Developers can further enhance objects with custom `C#` scripts, paving the way for unique gameplay experiences.

Imagine crafting a game level: a dodgeball arena illuminated by a radiant light source, with a camera capturing every thrilling moment. Unity makes this possible with simple objects like planes, cylinders, and spheres. The intuitive interface allows developers to select, move, rotate, and scale objects with ease, setting the stage for an exhilarating match [21].

But what's a game without some action? Unity's rigid body component breathes life into objects, allowing them to be influenced by gravity. Combine this with the material component, and you can create mesmerizing visual effects, from the sheen of a metallic surface to the rough texture of a stone [22].

Unity's commitment to realism and smooth gameplay is further evident in its two types of updates: Update and FixedUpdate. While the former is called every frame during gameplay, ensuring fluid animations and interactions, the latter syncs with the physics engine's frame rate, making it ideal for moving objects around and applying forced in realtime [23].

Unity's ML-Agents toolkit is a game-changer for those looking to infuse artificial intelligence into their games. ML-Agents provides a platform to train intelligent agents within the Unity environment using Reinforcement Learning, imitation learning, and more. This makes it an ideal choice for complex simulations like kiteboat training, where agents can learn optimal strategies through interaction. Unity is not just a game engine; it's a platform for innovation, and a testament to the limitless possibilities of virtual worlds.

2.4 Existing Kiteboat Technologies

As kite technologies improve and kites become more mainstream and accessible, there are several companies that have started to explore the potential of kite-powered vessels, both for a commercial and recreational purpose. Wingit [24], is a German company that have specialised in creating autonomous kite control systems that can be integrated onto pleasure vessels, seen in figure 2.3. This system can use any LEI (leading edge inflatable) kite and has its own custom kite-control-unit, as well as a remote control and autopilot. It is the autopilot that is particularly interesting, as this is what will be investigated throughout this thesis. The Wingit autopilot has 2 modes:

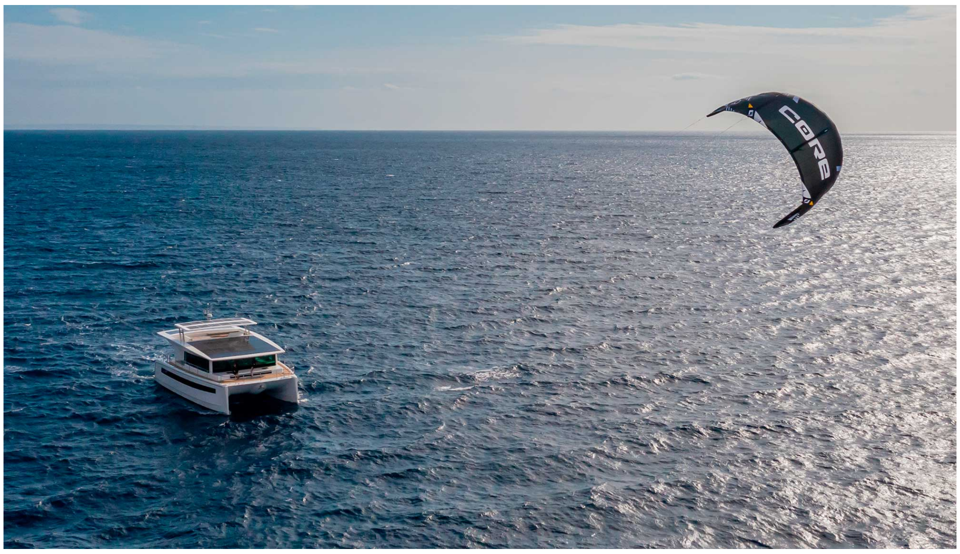

Figure 2.3: Silent 60 with Wingit Kite Control

- Lying Eight - The kite follows a horizontal figure of eight pattern, this is widely regarded as the best, most stable, and most consistent way to generate the maximum power.
- Zenith - The kite remains at 12 o'clock, directly above the boat.

The crucial thing about these modes is that they are pre-programmed. A complex control system, using line sensors to work out the position of the kite, has been created to allow the autopilot to fly the kite in these preconfigured patters and positions. This is at its core a PID controller, and while it is a good approach to controlling the kite, it is not very adaptable.

Beyond The Sea [25] is another company with kites as their primary focus. They are developing full solutions including kite and control systems for leisure and commercial applications.

Figure 2.4: Beyond The Sea 'SeaKite'

Their SeaKite seen in figure 2.4, is their latest and most curring edge innovation, which claims to utilise some 'AI' in its control mechanism.

Autonomous control systems for kites are new of the last 10 years, but have primarily focused on methods utilised by Wingit, a pre-programmed control system. Due to the technological boom in artificial intelligence and computing in the last 5 years, forward thinking companies like Beyond The Sea are just starting to explore machine learning as an alternative control method. This project aims to do just that, explore the creation of an autonomous kite and boat control mechanism using machine learning.

Chapter 3

Methodology

The following chapter will discuss the steps taken to create a robust reinforcement learning algorithm that is capable of controlling a kiteboat. It will start by explaining the tooling used to create the RL algorithm and the environment, before moving on to the implementation of the environment and the RL algorithm. Finally it will discuss the training process and optimisation. The development process discussed in this chapter can be seen in figure 3.1.

3.1 MLAgents

MLAgents is an open-source project that allows games and simulations to serve as the environment for training intelligent agents. At its core MLAgents utilizes RL, although it also supports other methods such as imitation learning.

3.1.1 Python Implementation

The neural network used in the reinforcement learning is implemented in Python. MLAgents toolkit is a Python Library that acts as an interface between the environment (gym), in this case Unity, and PyTorch [26]. PyTorch is an open-source machine learning library based on the Torch library, known for its flexibility, ease of use, and native support for GPU acceleration, which is essential for the computation-heavy processes involved in training neural networks. Torch is a Python-based scientific computing package that provides prebuilt components for machine learning and deep learning, as well as a wide range of mathematical functions. MLAgents uses a low level API to communicate directly with the Unity environment (`mlagents_envs`) frame by frame, and an entry point to train (`mlagents-learn`). This stepping process allows for the synchronous collection of observations, executions of actions and retrieval of rewards. The neural

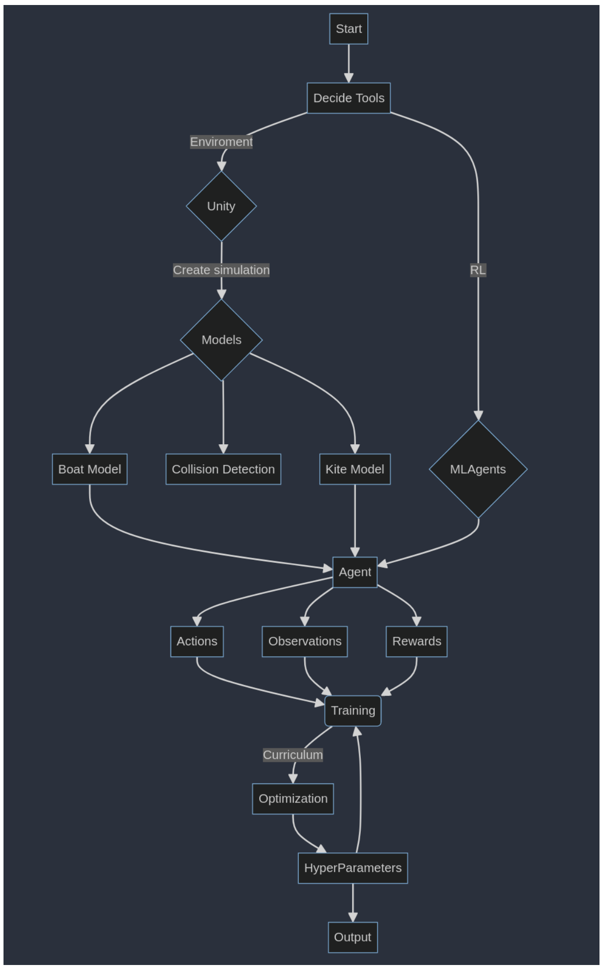

Figure 3.1: Development Process

network used in this project is a Proximal Policy Optimization (PPO) network, and so will utilise the actor-critic method as discussed in section 2.2.

RL, preveously discussed in section 2.1, is an approach to learning where an agent learns to make decisions by interacting with its environment. The fundamental components of this interaction with the environment are observations, actions and rewards.

- Observations (State): These are the pieces of information that the agent receives from the environment at each step or frame. In Unity, observations are collected through sensors or manually coded to be extracted from the game objects. They are typically fed into the neural network as a vector of floating-point numbers, representing the current state of the environment.

- Actions: Based on the observations, the agent takes actions which are the outputs of the neural network. These actions can be discrete (e.g., turn left, turn right) or continuous (e.g., change angle by a certain degree). The neural network's output layer is designed accordingly to provide the appropriate action space for the agent. (Configured as part of the behavioral parameters in Unity)

- Rewards: After taking an action, the agent receives a reward signal, which is a numerical value indicating how well the action contributed to achieving its goal. This reward is used to adjust the neural network's weights, with the aim of maximizing the total accumulated reward.

A full breakdown of the actions, observations, rewards, and how the agent script configures these for this project can be found in section 3.4.

A sequence diagram of the interaction between the Unity environment and the Python neural network can be seen in figure 3.2.

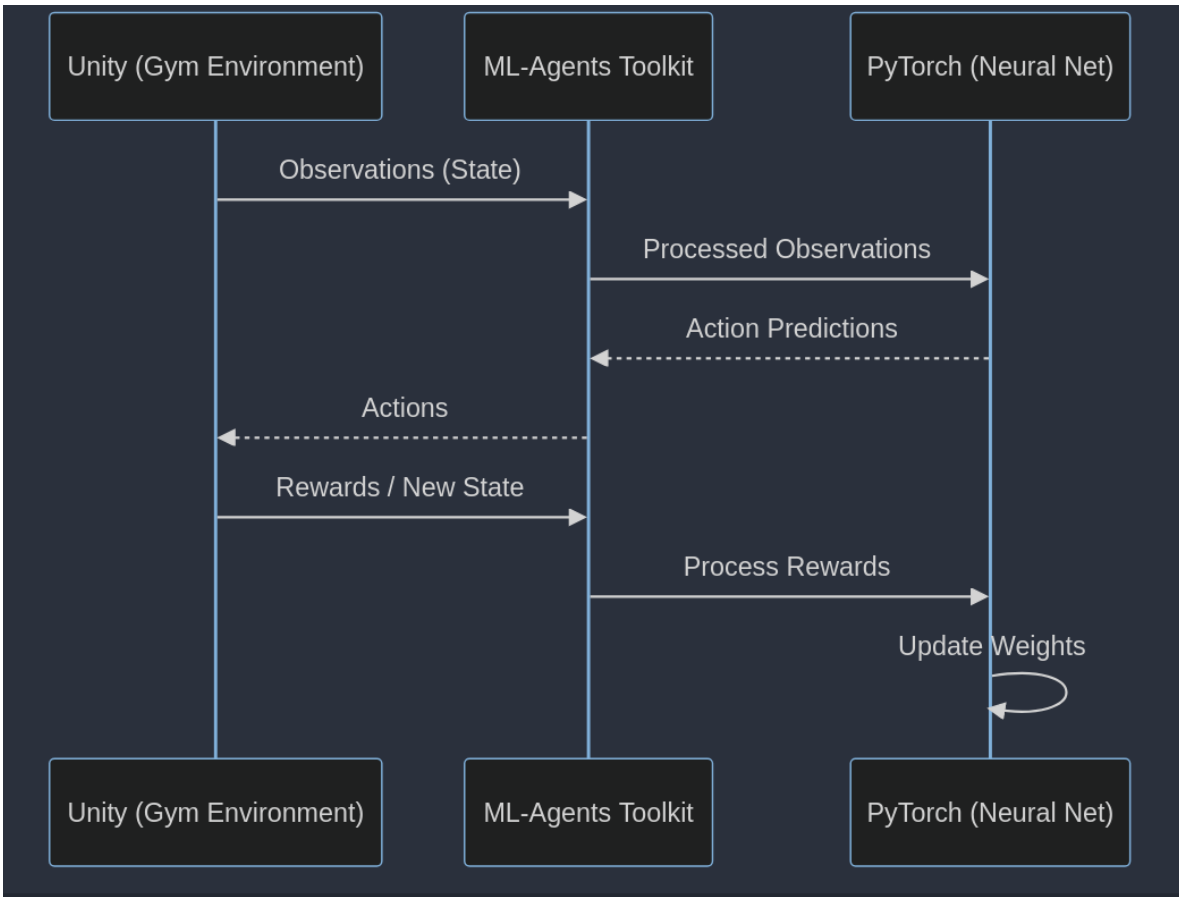

Figure 3.2: MLAgents Sequence Diagram

The technical instructions to setup MLAgents in Python and Unity are as follows:

- Initialise a python virtual environment
- Install MLAgents with the command `pip install mlagents`

- Configure Unity project to use MLAgents by importing the MLAgents package, see section 3.4 for more details.

3.2 The Environment

The first step to this process is to create the environment, which the agent will use to train. In this case that will need to look something like a sailing game; it will have some form of water, a boat, a kite and a course. For any machine learning endeavor, especially one with such intricate physical dynamics, the choice of simulation environment is paramount. Not only does it provide the playground for our AI agent to learn and make mistakes safely, but it also serves as a litmus test for the robustness and realism of the designed model.

Given the myriad of choices available, the Unity game engine emerged as the most suitable platform. Beyond its reputation in gaming, The inherent support for mesh bodies, colliders, and a variety of joints made it an attractive option for simulating the kite-boat system, which comprised a complex dance of forces, and counterforces. Unity is recognized for its potent physics engine, however for this project the use of Unity's physics engine will be kept to a minimum. Unity will be used primarily for its visual capabilities and the ability to be used as the gym environment for machine learning simulations.

Central to our simulation is the depiction of water, the medium in which our boat will navigate. Here, the Unity HDRP Water System 16.0.3 [27] came to the rescue. Bundled with Unity 2023.2.0b9, this water system provides a realistic representation of water with its undulating waves, refractions, and reflections. The alternative option to using Unity's water system was to model an entire particle fluid simulation, this would have had its advantages, however it would have been a lot more computationally expensive and would have taken a lot longer to implement. As this project was primarily focused on creating a RL algorithm for controlling a kiteboat it was decided that the Unity water system would be sufficient for this project.

The boat part of the kiteboat had two main component scripts, the buoyancy and the rudder, allowing the boat to float and be steered. The implementation of Buoyancy and the rudder are discussed in more detail in section 3.2.1. The kite had a single script that defined the physics and its explanation can be found in section 3.2.2. The final training environment can be seen in figure 3.3.

3.2.1 Boat Model

Boat Assumptions

- The Archimedes force is uniform across all submerged sections of the boat.
- The rudder forces of lift and drag could be aproximated to a torque applied about the rear of the boat.

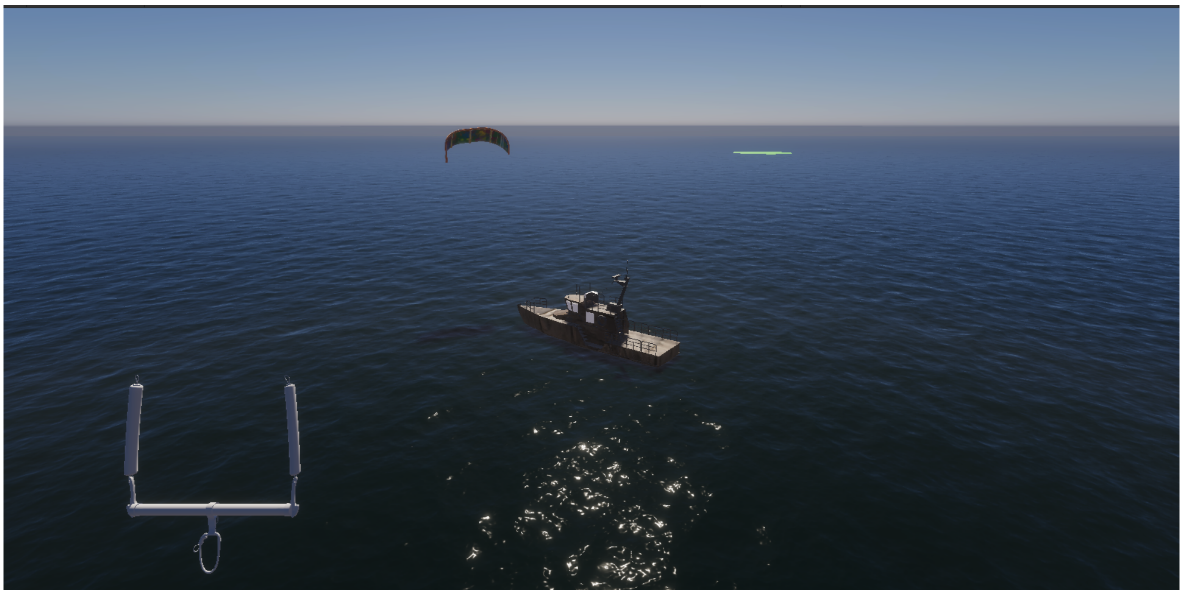

Figure 3.3: Training Environment

- Once the boat is moving at a speed greater than 0.25m/s the lift and drag forces of the keel are equal to the downwind component of the kite's resultant force- essentially providing a non'slip condition.

Buoyancy, the force that allows ships to float, was the first physical property to be addressed. Rooted in Archimedes' Principle, it dictates that the buoyant force exerted on a submerged body is equivalent to the weight of the fluid displaced by that body. In our Unity environment, the boat's hull, represented as a 'mesh' with an associated 'mesh collider', was divided into many small triangles or Voxels. These Voxels became the fundamental units for calculating buoyancy, allowing for a granular and realistic representation of the boat's interaction with water. This was achieved by first calculating the total Archimedes force (AF) of the entire boat using equation 3.1, followed by a local AF at each Voxel. The water level, y component, was then computed at each voxel's (x,z) coordinates to determine if it was above or below the surface. If below the surface the component of the AF was applied vertically at each voxel. This implementation can be viewed in the buoy.cs script in the project REF IN APENDIX.

$$F_B = \rho_w g V \tag{3.1}$$

While buoyancy ensures our boat doesn't sink, it's the rudder that grants it direction. The Rudder.cs script handles the implementation of the rudder and the keel. The equation for the torque applied about the rear of the boat is shown in equation 3.2.

$$\tau = \alpha v R \tag{3.2}$$

where τ is the torque, α is the angle of the rudder, v is the speed of the boat and R is the rotation scale.

3.2.2 Kite Model

Capturing the intricate movements of a kite as it fly's through the air involves a complex balance between theoretical aerodynamics and the unpredictability of real-world conditions. In this model several assumptions were made to streamline the complexity into a more manageable form and are shown below.

Kite Assumptions

- The kite is modeled as a symmetrical aerofoil, with constant lift and drag coefficients.
- Constant wind angle and laminar flow over the entire kite.
- The kite is always in the air, for the initial model the case where the kite crashes and requires relaunching was not considered. This would require adding buoyancy to the kite.

This sections outlines a kite model that, while simplified, serves as an effective tool for designing and testing the RL algorithm. The model is geared towards a realistic representation of the kites behavior and its response to control inputs. The kite chosen for this project was a Leading Edge Inflatable (LEI) kite, which is the most popular and mass produced recreational style of kites that exists. These kites connect to a control bar via 4 dyneema kite lines. Two center power lines take the load of the kite, while the outside two are responsible for steering, as shown in figure 3.4.

The kite model was implemented using the kite.cs script. The kite was modeled as a symmetrical aerofoil, and implemented as a rigid body with constant lift and drag coefficients. The lift and drag forces were calculated using the equations shown in equation 3.3 and equation 3.4.

$$F_L = \rho C_L A \frac{v^2}{2} \tag{3.3}$$

fixed at certain lengths, designed to permit movement across all rotational axis without any

$$F_D = \rho v^2 C_D A \frac{v^2}{2} \tag{3.4}$$

where F_L is the lift force, F_D is the drag force, ρ is the density of air, C_L is the lift coefficient, C_D is the drag coefficient, A is the area of the kite and v is the velocity of the kite relative to the wind.

In order to replicate the kite's mechanics and bar configuration, the model uses 4 configurable joints. These joints were fixed at certain lengths from the deck of the boat, designed to permit movement across all rotational axis without any 'bounce' effect. This design enables the kite to descend or 'fall' in conditions of low wind, mirroring real-world behavior where the kite may lose

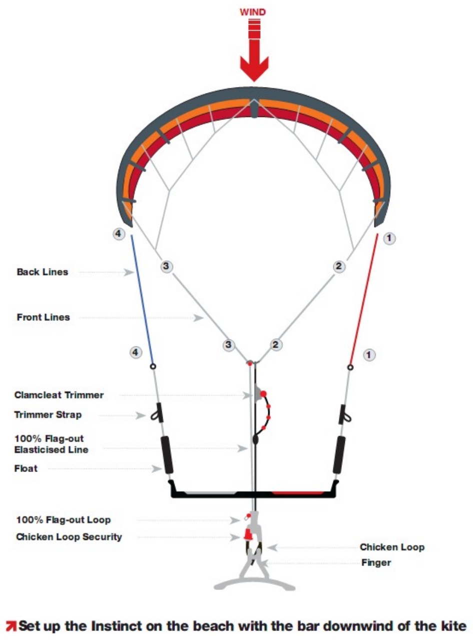

Figure 3.4: Kite Diagram

altitude but can be maneuvered back into position. To simulate the bar being pulled in, the lift and drag coefficients were increased, this has the effect of increasing the angle of attack of the kite.

3.2.3 Collision Detection

The Gilbert-Johnson-Keerthi (GJK) [28] algorithm is a sophisticated method for collision detection between convex shapes. This algorithm was the approach taken to detect weather the kiteboat had reached the waypoint during training, and later, to work out if it had rounded the marks of the racecourse. The implementation of the GJK algorithm can be found in the GJK.cs script in the project REF IN APENDIX. The GJK algorithm operates by iterative refinement of a simplex, which is a set of points that can define a line segment, triangle, or tetrahedron. The algorithm progresses by assessing whether the simplex contains the origin, which would imply an overlap between the two shapes. The initial direction d is determined by the normalized vector from mf1Pos to mf2Pos, constrained in the x-z plane by nullifying the y component, meaning the algorithm is only concerned with the horizontal plane.

Within the GJK method, the Support function plays a pivotal role, calculating the Minkowski difference between the two shapes in a specified direction. This is achieved by finding the farthest points along that direction on both shapes and then subtracting them to obtain a single point in the Minkowski space.

The HandleSimplex function is a recursive strategy that adjusts the simplex and direction d based on whether the current simplex is a line or triangle. For a line, the LineCase function is invoked, and for a triangle, the TriangleCase is employed. These functions adjust the simplex and direction of search to move closer to the origin, if it is not already contained within the simplex.

3.2.4 Course Generation

3.3 Controls

In order to ensure the AI would be able to learn to sail the kiteboat a playable game version was created. As discussed above the kite was modeled using 4 configurable joints to replicate the line system. There are several ways of configuring these so that a simple control input will result in the desired movement of the kite.

3.4 RL Implementation

In order to use Unity game engine as the environment for RL it must be setup correctly, the steps are as follows:

1. Enable MLAgents in Unity: Window → Package Manager → MLAgents → Install
2. Create an empty game object in the scene.
3. Create a new `C#` script of with class of type 'agent' and attach it to the game object, include the methods discussed in subsection 3.4.1.
4. Add the Behaviour Parameters component to the game object.
5. Create a new agent config file (`.yaml`) and set the 'Behavioural Name' in the Behaviour Parameters component to the name of the config file.

A visual representation can be found in section A.1 of the appendix.

3.4.1 The Agent Script

The kiteboat agent sets up the scene for learning and in order for Unity to correctly process the script it must have the following 4 functions implemented:

- OnEpisodeBegin()
- CollectObservations(VectorSensor sensor)
- OnActionReceived(ActionBuffers actions)
- Heuristic(in ActionBuffers actionsOut)

These are the minimum requirements for the agent setup, however there are several other functions that can be implemented to further customize the agent, handle errors and provide additional information. The agent starts by taking in the Kite and the Boat rigid bodies, as well as the rudder and kite scripts. A new gjkCollisionDetection is also initialised. The script starts with the OnEpisodeBegin method, which in turn starts by ensuring the training environment has been reset. The reset method sets the velocities and angular velocities of all rigid bodies in the scene to 0 and returns the kiteboat to a starting position. The remaining methods will be discussed in more detail below.

3.4.2 Observations

CollectObservations provides the network will all the information about the State of the environment, and so aims to provide all the required information for the agent to make an informed decision. This means fully describing the movement of the kite and boat, as well as the position of the boat relative to the waypoint, allowing it to gain an idea of direction. It is easy to see how the number of observations could rapidly increase, but this would also increase the complexity of the network and the likelihood of the agent becoming 'confused' by the data its receiving. With this in mind the goal is to provide all the required information about the state in the minimum number of observations possible, this is achieved by combining vectors where appropriate. Another consideration when thinking about the observations was that they should be possible to collect if this system were to be created in the real world. This means that the observations should be possible to collect using sensors, such as GPS, wind speed and direction, and accelerometers. The observations used in this project are shown in table 3.1.

Observation	Vector Size	Description
Distance to the waypoint	1	The distance to the waypoint from the boat
Boat Speed	1	The speed of the boat in the forwards direction
Wind vector	3	The direction of the wind
Relative Boat Angle	1	The angle of the boat relative to the wind
Kite Position	3	polar angles of the kite relative to the wind
Relative velocity of the kite	3	The velocity of the kite relative to the boat
Kite altitude	1	The height of the kite above sea level
Rudder Angle	1	The angle of the rudder

Table 3.1: Observations

The total number of observations passed to the network is 14, this is a relatively small number of observations and so the network should be able to process them quickly. When providing observations to an agent is is essential they have no discontinuity in their values, as this can cause the network to become unstable. Observations that may be discontinuous are normalized to a value between 0 and 1. Normalised observations can increase the learning speed of algorithms, as the network does not have to learn the scale of the observations.

3.4.3 Actions

When the agent starts to train it has no idea what to do, it is essentially a blank slate, so starts by randomly flicking around the actions. The agent was given a discrete action space of 6 actions, these are shown in table 3.2. The actions are passed to the network as a vector of 6 values, each value is a float between 0 and 1. The network then interprets these values and outputs the appropriate action. The actions are then passed to the OnActionReceived method, which in calls the methods that control the kite and boat. Discrete actions were chosen because then the rate of change of the angles that control the rudder and kite is not the choice of the network. This gives the network an easier job and means the amount of freedom the network has can be configured. i.e. if the network could pick any value between 0 and 1 for the bar position, it would see far more aggressive controls making it harder for a stable flight to be achieved. On the other had the discrete action space encourages the agent to make a decision and stick with it while the action is being applied.

These actions were tested as part of making a playable game so that the agent receives the same actions as a human controlling the kiteboat simulation. The Heuristic method that is one of the required functions for the agent script, allows the agent to be controlled by a human. This is useful for testing the environment and the controls, as well as for playing the game. The Heuristic method is called when the agent is not training and so the agent can be controlled by a human. The Heuristic method takes in the action vector and sets the values of the actions to the values of the keyboard inputs. The keyboard inputs are set in the Unity editor and are shown in table 3.2.

Action	**States**	**Description**	**Keyboard Input**
Kite Bar Position	Left, Off, Right	The different states change the bar position	Arrow left/right
Rudder Angle	Left, Off, Right	The angle of the rudder	Keys 'A' and 'D'
Kite Bar Power	0, 1	The power of the kite bar	Space Bar

Table 3.2: Actions

3.4.4 Rewards

The rewards provided to the agent each step are the fundamental input that influences the future actions taken by the neural network, and so must be carefully applied when the agent performs positive actions and penalized when it performs negative actions. The reward function is the most important part of the RL algorithm, as it is the only way the agent can learn. The reward function is also the most difficult part of the RL algorithm to get right, as it is very easy to create a reward function that does not encourage the desired behavior, moreover it is common to create a reward function that helps the agent fall into local maxima while training, especially with complex tasks. Table 3.3 shows the rewards used in this project, These rewards were however not all applied at the same time, they were applied in stages as the agent progressed through the curriculum. This is discussed in more detail in the following section 3.5.

Reward	Condition	Value	Timing	Description/Impact
	>1m/s	+0.005	Every Step	
Boat Forward Speed	>4m/s	+0.01	Every Step	Optimise the boat for forward speed
	<0.1m/s	-0.001	Every Step	
Rudder Angle	> 60°	-0.001 * abs(rudderAngle-60)	Every Step	Penalise the agent for aggressive steering
Distance to Waypoint	current < previous[1]	$+\frac{10}{current}$	Every Step	Encourage the agent to move towards the waypoint
	current > previous	-0.01 * (current - previous)	Every Step	
Waypoint Reached	Waypoint Reached	+10	On End	Encourage the agent to reach the waypoint
Kite Crashes	Kite Hits Water	-10	On End	Penalise the agent for crashing the kite
Kite Flying	Kite Angle > 15°	+0.01	On Action	Encourage the agent to keep the kite in the air
	Kite Angle > 45°	+0.05	On Action	
Keep Alive	Always	-0.001	Every Step	Encourage the agent to explore the action space
Decision Making	Current Actions != Previous Actions	-0.1	On Action	Encourage the agent to be decisive in its actions

Table 3.3: Rewards

3.5 Training

Before commencing the training of the kiteboat agent, a simpler test agent was created to check the workflow and ensure the environment was setup correctly. This test agent was a simple cube that was trained to move towards a waypoint, following a pacemaker. This was a good test of the environment as it was a simple task that could be easily visualized as shown in figure 3.5 However this test agent proved more tricky than initially anticipated and after some additional research some core RL training concepts that improve the quality of training were discovered. The first of these being make the problem a 'Keep Alive' i.e. do the correct thing or die. In the context of the path follower this meant that should the cube ever be further away from the pacemaker than its original distance of 2m the episode was ended and a large negative reward given. This method encourages the agent to do the correct process if it wants to stay alive. Making the problem a keep alive problem is a simple way to improve the quality of training and is a common practice in RL. The path follower changed from not making much progress over the course of 500000 steps to being able to complete full laps of the course in around 100000 steps.

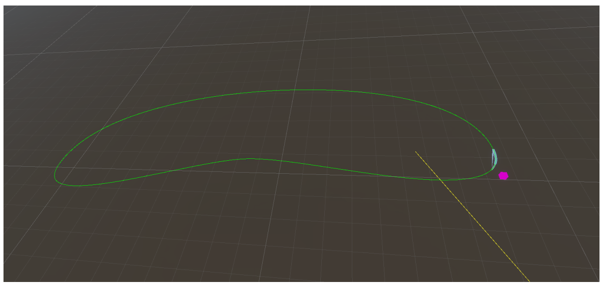

Figure 3.5: Keep Alive

[1]'previous' is the previous distance to the waypoint at the previous time step, and 'current' is the current distance to the waypoint.

3.5.1 Curriculum Learning

The next RL technique to help improve training is called Curriculum Learning (CL) [29]. CL is an instructional strategy that structures the learning process, much like how a school curriculum guides human learning. It involves organising the leaning tasks from simple to complex, facilitating the agent's ability to incrementally acquire, transfer and refine knowledge. In essence it breaks down large complex tasks into more manageable bitesized chunks that the agent can progress through. CL was not utilised in the path follower but was used in the kiteboat agent. The kiteboat agent was trained in several stages shown below.

1. Master the controls- a keep alive problem
2. Sail downwind towards a target
3. Progressively move the waypoints upwind with each completed waypoint
4. Randomly generate waypoints in any direction

Step 1 in the curriculum ensures the agent has a fundamental grasp of the controls and is able to keep the kite in the air. This is a keep alive problem and so the agent is rewarded for keeping the kite in the air and penalised for crashing. The direction the boat sails is not important at this stage. Step 2 is a simple task that the agent can learn quickly, having now gained a grasp of the kite control it must learn to steer straight towards the waypoint. This was also turned into a keep alive problem, move closer to the next waypoint or end the episode. Step 3 is where the agent starts to learn to sail its own path, initially this will still be a straight line until the waypoints are spawning upwind of the kiteboats initial location. At this point the agent must start to experiment with finding the VMG (velocity made good), which is a measure of the speed at which a vessel is moving directly towards its destination, considering both its heading and wind direction. In stage 3 the waypoints will spawn more and more upwind until they are directly upwind, this is in an effort to encourage the agent to find the emergent property of Tacking, a maneuver by which the nose of the boat transitions through the wind while it turns around.

Figure 3.6 shows the difference in the paths at stages 2 and 3 of the curriculum. The path in stage 2 is a mostly straight line towards the waypoint, while the path in stage 3 is a zigzag upwind.

3.5.2 Hyperparameters

Hyperparameters are critical configurations that govern the training process of machine learning models. In reinforcement learning, particularly within the scope of PPO, they play a pivotal role in determining how effectively the model learns from its environment. The hyperparameters are defined within the boatAgent.yaml config file. A variety of different configurations were tested to find the a set of hyperparameters that yielded the best results. It is also possible to schedule

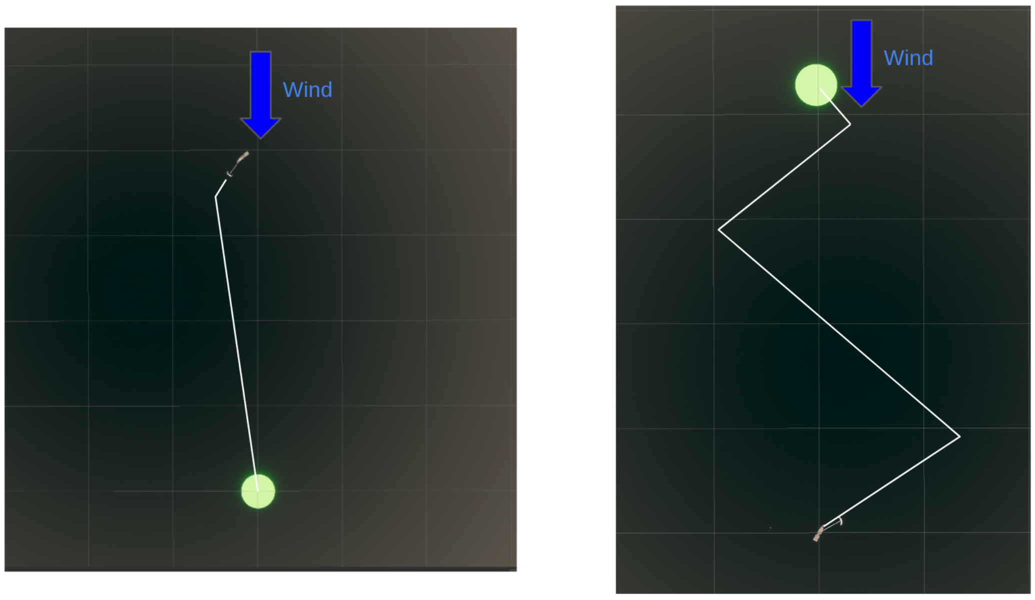

Figure 3.6: Stages 2 and 3 of the curriculum

the hyperparameters to change over the course of the training, this can potentially lead to better results as the agent can adapt its learning behavior during different stages of training. To test which hyperparameters were the most effective a grid search was performed. Section A.2.2 of the appendix shows the python script used to generate the 100 config files to be tested. The performance metric of the average reward per episode with a max steps of 1,000,000 was used to determine the best hyperparameters to then be used for the final long training runs. The following hyperparameters were experimented with:

- **Normalised network inputs**- Adjusting normalized inputs ensures that the model receives input data within a consistent scale, which can aid in stabilizing and speeding up the training process.
- **Number of layers in the network**- The depth of the network, defined by its number of layers, determines the level of abstraction the model can achieve, impacting its ability to learn complex patterns.
- **Number of nodes in each layer**- The width of each layer, given by the number of nodes, influences the model's capacity to capture nuances in the data, with more nodes allowing for more complex representations.
- **Beta**- The beta hyperparameter balances the exploration-exploitation trade-off by controlling the strength of the entropy term in the objective function, which affects the agent's policy diversity.

- **Curiosity**- This motivation factor drives the agent to explore unseen states, enhancing learning in sparse reward environments by providing an internal reward signal.
- **Epsilon**- Epsilon values dictate the degree of policy randomness, serving as a threshold for exploration in the agent's decision-making process, promoting the discovery of new strategies and techniques.
- **Learning rate**- The learning rate controls the magnitude of updates to the model's weights, with too high a rate risking overshooting minima and too low delaying convergence.

After training the first 100 config files for 1,000,000 steps it could be seen the training was very inconclusive, so further experimentation was conducted. 500 config files were randomly picked from a pool of unique files that varied 10 different parameters by 3 options, comprised of the 7 hyperparameters, 2 network parameters and 2 reward signals. The 500 config files were trained for 10,000,000 to ensure the agent had enough time to learn. The limitation of this approach was that there were a posable 3^{10} combinations of hyperparameters, which is 59,049 unique different configurations, meaning the random 500 selected only represented 0.8% of the possible combinations. However as it is a random selection from the sample it should help identify some the of better hyperparameter configurations, after the best configs from this run were found they were tuned manually and retrained again. The comparison between these different configurations can be viewed in chapter 4, section 4.2.1. The final config can be found in section A.2.3 of the appendix and is further explained in chapter 4. Given more time it would have been prudent to conduct a larger grid search of the hyperparameters to ensure that the best combination was found for this project.

3.6 Optimization

3.6.1 Blue Crystal HPC

The Blue Crystal HPC, operated by the University of Bristol, offers significant computational resources tailored for intensive tasks such as machine learning simulations.

To utilize Blue Crystal for MLAgents simulations, the following steps were undertaken:

- **Access and Security:** Gained access to the university's HPC and set up SSH keys for secure communication.
- **File Preparation:** Built the .x86_64 Unity build file and uploaded it, along with the necessary config files and credentials, to Google Drive.
- **Automation with Shell Script:** Developed a shell script to automate the process. This script:

- – Retrieves the build files from Google Drive.
- – Sets up a virtual environment on Blue Crystal.
- – Installs the ML-Agents.
- – Manually runs the training for a given number of steps.
- – Upon completion, uploads the results back to Google Drive.

The shell script can be found in section A.2.3 of the appendix. Although the process appears simple there were a number of difficulties encountered during the usage of the HPC. First and foremost was handling the files required for training, Google Drive proved very useful, and using a link to a public folder the build files were easily accessible using `wget`. The results dir, which is created during the training and contains the trained `.onx` model was zipped up and uploaded back to google drive using the credentials.json file. As this process had to be entirely autonomous and the HPC node could not perform any mouse clicks, the credentials.json was created by uploading the file to google drive locally and performing the manual authentication process, then uploaded to google drive. Again as no authentication could be completed the drive folder had to be public so anyone with the link could download it, this presented a security risk albeit not major. To mitigate this security risk the OAuth 2.0 Client ID was deleted after each training session. To handle the running of many hundreds of config training runs another shell script was created that configured the setup of the runs. This included downloading the config.zip (that contained the unique config files), and the build.zip that contained the prebuild Unity files. The shell script then unzipped these files and ran the training script for each config file, adding it to the *SLURM* queue. *SLURM* (Simple Linux Utility for Resource Management) is an open-source, scalable cluster management and job scheduling system for Linux clusters, and is the queue management system used on the hpc.

Useful commands for working with Blue Crystal:

- **sbatch:** Submits a job to the queue.
- **sacct:** Checks the status of a job.
- **scancel:** Cancels a job.

Parallelization and Optimization

Initially, a single node on Blue Crystal was employed to run the simulation. This node with 28 CPUs was responsible for both hosting the environment and executing the model. However, Blue Crystal's architecture allows for more advanced parallelization strategies. Distributing the simulation across multiple nodes can enhance efficiency. Additionally, offloading the ML-Agents toolkit to a GPU core can further accelerate the learning process.

However, it's worth noting that the demand for GPUs on Blue Crystal is high. For tasks that don't necessitate the power of GPUs, relying on CPUs, even if they take longer, is a practical choice given the limited GPU availability.

CHAPTER 4

Results and Validation

Introduction Briefly summarize the goals of the RL experiments. Outline the structure of the results chapter.

4.1 Final Experimental Setup

The training scene used for the final training runs can be found in:
`Assets/Scenes/kiteboat_training`. The final experimental setup was a combination of the best performing elements from the previous experiments. The model was configured with the config file shown in section A.2.3 of the appendix.

Outline the training procedure, including any pre-training steps.

4.2 Training Results

The following section will present the results of the training runs, and discuss the overall performance of the agent.

4.2.1 Hyperparameter Tuning

Show the graphs with the results of the grid search and explain why the config set was chosen

4.2.2 Final Training

The final training consisted of 6 runs with different config files. The first was the manually created, which

4.3 Agent Performance

Difficulties Encountered

There were several difficulties encountered when trying to get the agent to learn anything let alone the combination of directionally sailing a kiteboat. One of the most common local maxima that the agent fell into was for the agent to steer on 'hard lock' with the rudder at˜$70-90°$, shown in figure 4.1 where the target can be seen as the green area in the distance. This behavior allowed it to learn to fly the kite very reliably with the boat in a more consistent and stable position. These episodes provided false positives in the training data because as soon as the agent started to explore the rudder space more it was not able to fly the kite. To try and combat this behavior a large negative reward was added for aggressive steering as shown in table 3.3. This went some way to discouraging this behavior but it was still observed in some of the later training runs. After this rudder reward was added it was observed that the agent took almost 5 times as long to learn to fly the kite with some reliability.

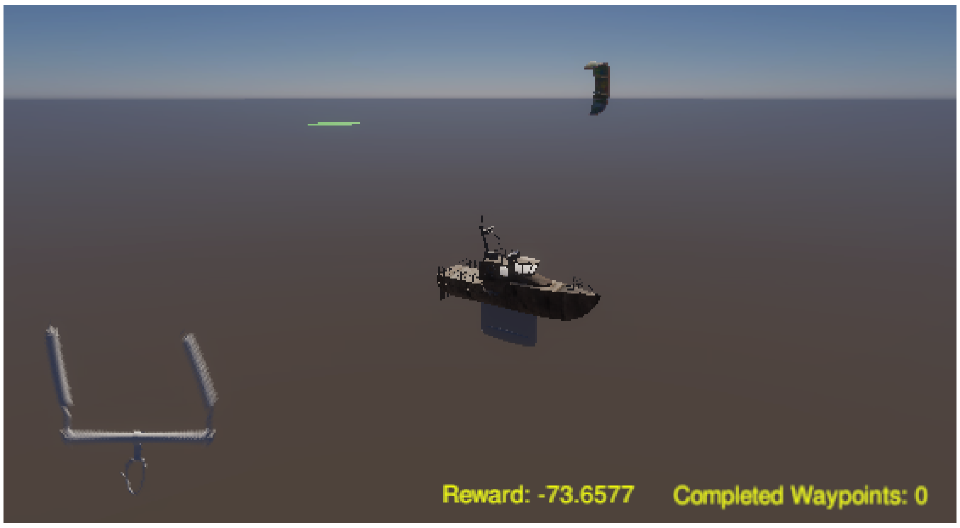

Figure 4.1: The agent steering on hard lock

4.3.1 Human Comparison

Earlier it was mentioned that a heuristic playable game was created to make sure the model felt sensible and could be played by a human player. To create a baseline for the agent's performance 5 human players were asked to play the game for 5 minutes each in heuristic mode and their results were logged. The top 5 performing agents were then run for the same time and the same metrics recorded.

4.3.2 Hardware

There were several limitations that affected the quality of the training and thus the trained model. First and foremost was compute, as expected when conducting any machine learning training, the more compute available the better. The local machine used for training was a 16-core i9 with 32GB of RAM and a T2000 nvidia graphics, and took approximately 2 hours per million steps completed. This was not a viable option for training the agent to a high level of performance, and so the training was attempted to the university HPC. The HPC has 525 Lenovo nx360 m5 nodes each with two 14 core 2.4 GHz CPUs, and 32 GPU nodes with two cards each. At face value this looks wonderful and training should be a breeze, this was not quite the case. Unity does not support native multi threading, due to the complex nature of its physics engine, so it runs on a single CPU unless manually specified. Manual threading was possible but only for separate tasks that could be called independently of the model, such as the collision detection algorithm. As expected this severely limits the speed of training, and using the hpc did not change these run times. However, the HPC was not without its advantages, the main being that it was possible to run multiple training runs in parallel, and so the hyperparameter tuning was conducted on the HPC. This would not have been possible on the local machine as it would have taken far too long to test all the combinations. The ability to submit a large batch of jobs to be run in parallel and then the results collection automated was a huge time saver. The HPC also had the advantage of being able to run the training for longer periods of time without causing inconvenience, so even though the training speed was limited it was not a problem to let them run for many days.

The wait time for resources to be allocated increased the longer the job was scheduled to take

Thus this project wouldn't have been possible without the HPC.

4.4 Critical Evaluation

4.4.1 Objective's Met

Table opf objectives and outcome goals and wheather they were met or not

Chapter 5

Future Work

Curriculum

It is easy to imagine to the limitless possible combinations of rewards available, and as some of the desired behaviour was observed in the simulation, it is clear that the reward function was on the right lines. However, as explained earlier, due to the complex nature of the challenge at hand, a curriculum was used to split the reward function into smaller, more manageable stages, which proved beneficial but there was room for dissecting the task into even smaller stages. This would have allowed for a more gradual learning process and hopefully given the agent a better chance at retaining what it had learnt in the previous stage. It would also have been good to try multiple different curriculum's to see if the agent could learn the task in a different order.

Evaluation of Performance

It would be advantageous to setup a comprehensive evaluation process that could be conducted in between each training run. This would provide clear incite into the runs success and help provide direction for the next training run. While training a complex system it is easy to get distracted in nuances of the training process as there are so many potential different factors effecting the outcome.

Simulation Environment

The simulation (training environment), albeit not perfect, was a good representation of the real world, and was a good starting point for the project. In future it would be good to create a more accurate simulation, representing the real world as closely as possible. This would allow for easier integration into a physical system, should the training reach a point where it is ready to be tested in the real world. To create a better simulation, the first aspect to be addressed would be the water model. Remove any water system and create a particle simulation from scratch, this would allow for proper mechanics and realistic water behaviour.

Once the agent has learnt the simpler task of the kiteboat controls, it would be good to start adding in adverse weather conditions. This would be in the form of non laminar wind, i.e. gusty shifty wind, and varying waves. These conditions must be added to the model if the agent is ever going to progress to the point of becoming reliable in the real world, as these are the conditions that the agent will be faced with. Once a reward function has been found that allows the agent to learn the task, it is at this point where the conditions should be added and the agent retrained from scratch; it would be interesting to observe weather this same reward function is still able to learn the task in the new conditions, or if a new reward function is required.

Another part of improving the simulation (the water system), would be to add the ability for water relaunching. LEI kites have the ability to relaunch from the water, and this is a key reason for their popularity in the kiteboarding community. Adding the mechanics for water relaunching into the simulations and training the agent to perform this task would be a good next step, improving the reliability and robustness of the agent. One of the main concerns for the viability autonomously controlled kites is what happens if the kite crashes into the water. It is easy to imagine many dangerous and difficult situations that could arise, and so it is important to tray and mitigate these risks as early as possible to improve the change of success in the real world.

APPENDIX A

APPENDIX A

A.1 Unity Setup

A.2 Scripts

A.2.1 GJK Collision Detection

A.2.2 Grid Search

```
import itertools
import os

# Define the ranges for each hyperparameter you want to vary, with fewer options
batch_size_options = [256, 512]
buffer_size_options = [2048, 4096]
learning_rate_options = [1.0e-4, 3.0e-4]
beta_options = [1.0e-4, 5.0e-4]
epsilon_options = [0.2, 0.3]
lambd_options = [0.95, 0.99]
num_epoch_options = [3, 4]
hidden_units_options = [128, 256]
num_layers_options = [4, 5]

# Create a product of all the hyperparameter options
grid_search = list(itertools.product(
    batch_size_options,
    buffer_size_options,
    learning_rate_options,
    beta_options,
    epsilon_options,
    lambd_options,
    num_epoch_options,
    hidden_units_options,
    num_layers_options
))

# If there are more than 100 configurations, randomly sample 100 from them
import random
if len(grid_search) > 100:
    grid_search = random.sample(grid_search, 100)

    # Create a directory for the config files if it doesn't exist
config_directory = "config_files"
os.makedirs(config_directory, exist_ok=True)

# Function to generate the config file content
def generate_config_content(batch_size, buffer_size, learning_rate, beta, epsilon,
lambd, num_epoch, hidden_units, num_layers):

    return f"""behaviors:
  BoatAgent:
    trainer_type: ppo
    hyperparameters:
    batch_size: {batch_size}
    buffer_size: {buffer_size}
    learning_rate: {learning_rate}
    beta: {beta}
    epsilon: {epsilon}
    lambd: {lambd}
    num_epoch: {num_epoch}
    learning_rate_schedule: constant
    network_settings:
    normalize: false
```

```
hidden_units: {hidden_units}
num_layers: {num_layers}
reward_signals:
extrinsic:
gamma: 0.99
strength: 1.0
curiosity:
strength: 0.1
gamma: 0.99
learning_rate: {learning_rate}
max_steps: 1000000
time_horizon: 64
summary_freq: 10000
checkpoint_interval: 50000
keep_checkpoints: 25
"""

# Generate and save the config files
for idx, config in enumerate(grid_search):
batch_size, buffer_size, learning_rate, beta, epsilon, lambd, num_epoch,
hidden_units, num_layers = config

config_content = generate_config_content(batch_size, buffer_size,
learning_rate, beta, epsilon, lambd, num_epoch, hidden_units, num_layers)

config_filename = f"config_{idx+1:03d}.yaml"
config_filepath = os.path.join(config_directory, config_filename)
with open(config_filepath, 'w') as file:
file.write(config_content)

print(f"Generated␣{len(grid_search)}␣configuration␣files␣in␣the␣directory
␣␣␣␣'{config_directory}'.")
```

A.2.3 Config

```
   behaviors:
 BoatAgent:
   trainer_type: ppo
   hyperparameters:
     batch_size: 256
     buffer_size: 4096
     learning_rate: 3.0e-4
     beta: 5.0e-4
     epsilon: 0.3
     lambd: 0.99
     num_epoch: 4
     learning_rate_schedule: constant
   network_settings:
     normalize: false
     hidden_units: 256
     num_layers: 6
   reward_signals:
     extrinsic:
       gamma: 0.99
       strength: 1.0
     curiosity:
       strength: 0.1
       gamma: 0.99
       learning_rate: 3.0e-4
   max_steps: 50000000
   time_horizon: 64
   summary_freq: 10000
   checkpoint_interval: 50000
   keep_checkpoints: 25
```

HPC Shell Script

BIBLIOGRAPHY

[1] G. Ferentinos, M. Gkioni, M. Geraga, and G. Papatheodorou.
Early seafaring activity in the southern ionian islands, mediterranean sea.
Journal of Archaeological Science, 39(7):2167–2176, 2012.
doi: 10.1016/J.JAS.2012.01.032.
URL `https://dx.doi.org/10.1016/J.JAS.2012.01.032`.

[2] Lionel Casson.
Ships and Seamanship in the Ancient World.
Princeton University Press, 1995.

[3] Robert Gardiner.
The Advent of Steam - The Merchant Steamship before 1900.
Conway Maritime Press Ltd., 1993.

[4] J.J. Corbett et al.
Mortality from ship emissions: A global assessment.
Environmental Science & Technology, 41(24):8512–8518, 2007.

[5] M. Vahs.
Retrofitting of flettner rotors – results from sea trials of the general cargo ship "fehn pollux".
Proceedings of the Royal Institution of Naval Architects - Part A: International Journal of Maritime Engineering, 2020(A4):641, 2019.
doi: https://www.intmaritimeengineering.org/index.php/ijme/article/view/1146.

[6] Richard S Sutton and Andrew G Barto.
Reinforcement learning: An introduction.
MIT press, 2018.

[7] Christopher J Watkins and Peter Dayan.
Q-learning.
Machine learning, 8(3-4):279–292, 1992.

[8] Richard Bellman.
Dynamic Programming.

Princeton University Press, 1957.

[9] David Silver et al.
Mastering the game of go with deep neural networks and tree search.
Nature, 529(7587):484–489, 2016.

[10] Chen Yang and Lin Wu.
Reinforcement learning-based control for autonomous boats: A survey.
Journal of Marine Science and Technology, 25(1):1–10, 2020.

[11] Richard Hallion.
Taking flight: inventing the aerial age, from antiquity through the First World War.
Oxford University Press, 2003.

[12] Uwe Fechner.
A Methodology for the Design of Kite-Power Control Systems.
PhD thesis, Delft University of Technology, 2016.

[13] Moritz Erhard and Hauke Strauch.
Control of towing kites for seagoing vessels.
IEEE Transactions on Control Systems Technology, 21(5):1629–1640, 2013.

[14] Marielle Christiansen, Kjetil Fagerholt, and David Ronen.
Ship routing and scheduling in the new millennium.
European Journal of Operational Research, 228(3):467–483, 2013.

[15] Volodymyr Mnih et al.
Human-level control through deep reinforcement learning.
Nature, 518(7540):529–533, 2015.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov.
Proximal policy optimization algorithms.
arXiv preprint arXiv:1707.06347, 2017.
URL `https://arxiv.org/pdf/1707.06347.pdf?fbclid=IwAR1DwRSkBzhqXRhVh3XQ_Nu_XYvN_sMR4ppYM28h3qAtx9EK03Lrl0cF7Dg`.

[17] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, F. Janoos, L. Rudolph, and A. Madry.
Implementation matters in deep rl: A case study on ppo and trpo.
In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.
URL `https://dblp.org/rec/conf/iclr/EngstromISTJRM20.html`.

[18] Merlin.

Unity in 100 seconds, 2023.
URL `https://www.youtube.com/watch?v=iqlH4okiQqg`.

[19] Unity Technologies.
Unity User Manual, 2021.

[20] Will Goldstone.
Unity 3.x Game Development Essentials.
Packt Publishing Ltd, 2010.

[21] Ashley Harrison.
Mastering Unity 2D Game Development.
Packt Publishing Ltd, 2013.

[22] Sue Blackman.
Unity 3D Game Development by Example Beginner's Guide.
Packt Publishing Ltd, 2012.

[23] Unity Technologies.
Unity scripting api: Update and fixedupdate, 2022.
URL `https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html`.

[24] Wingit.
Wingit, 2023.
URL `https://www.kite-boat.com/en/`.
Accessed: 2023-11-20.

[25] Yves Parlier.
Beyond the sea, 2023.
URL `https://beyond-the-sea.com/en/seakite/`.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al.
Pytorch: An imperative style, high-performance deep learning library.
In *Advances in Neural Information Processing Systems*, 2019.
URL `https://ar5iv.org/abs/1912.01703`.

[27] Unity Technologies.
Unity high definition render pipeline water system.
Unity Documentation, 2023.
URL `https://docs.unity.cn/Packages/com.unity.render-pipelines.high-definition@16.0/changelog/CHANGELOG.html`.

Accessed: 2023-11-20.

[28] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi.
A fast procedure for computing the distance between complex objects in three-dimensional space.
IEEE Journal on Robotics and Automation, 1988.
The seminal work on the GJK collision detection algorithm.

[29] Yoshua Bengio, Jerome Louradour, Ronan Collobert, and Jason Weston.
Curriculum learning, 2009.
URL `https://qmro.qmul.ac.uk/xmlui/handle/123456789/15972`.
Accessed: 2023-11-20.