TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Anton Jaštšuk 206000IACB

2D SHIP CONTROL SIMULATION

Bachelor's Thesis

Supervisor: Uljana Reinsalu, PhD

Co-Supervisor: Karl Janson, PhD

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Anton Jaštšuk 206000IACB

LAEVA JUHTIMISE 2D SIMULATSIOON

Bakalaureusetöö

Juhendaja: Uljana Reinsalu, PhD

Kaasjuhendaja: Karl Janson, PhD

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Anton Jaštšuk

29.12.2023

Abstract

The goal of this thesis is to create a 2D ship simulation for the purposes of testing algorithms that control real-life ships in the sea to reduce the costs of tests. To create a simulation, a platform or a software must be chosen amongst the most fitting for the task. After the comparisons between platforms are made, Godot game engine is chosen as the best and is used throughout the development process of the simulation. To limit the scope of the development cycle a set of problems and requirements were established. During the development of the simulation many different additional features were implemented to widen the functionality of the simulation to fit the specific needs. As the result, many different features were developed: ship control using the keyboard with physics interactions (object collision, be it a static or a movable object); dynamic (movable) and static objects as other 'foreign' ships moving around in the simulation and the rocks, beaches and ground respectively; created maps with different static and dynamic objects and a way to easily add or create new maps; general user interface with a guide on controls and sliders that allow to change ship parameters on the fly (which is also possible to be done using a configuration file); recording (saving) ship movement and playback of this movement from the saved file; toggleable wind force application on the ship movement; TCP (Transmission Control Protocol) server setup and connection through a client using a Python script proof-of-concept; camera follow and zoom in and out functionality; a map for demonstration purposes of a real-life location "Tilgu" port. After every feature was developed several things could be improved upon. The future development is possible with different ideas and tactics in mind.

This thesis is written in English and is 40 pages long, including 5 chapters, 35 figures and 2 tables.

Annotatsioon

Laeva juhtimise 2D simulatsioon

Selle lõputöö eesmärk on luua laeva 2D simulatsioon, et testida algoritme, mis kontrollivad päriselus meres liikuvaid laevu, vähendades sellega katsete kulusid. Simulatsiooni loomiseks tuleb valida platvorm või tarkvara, mis sobib ülesande jaoks kõige paremini. Pärast platvormide võrdlust valitakse parimaks Godot mängumootor, mida kasutatakse kogu simulatsiooni arendusprotsessi vältel. Arendustsükli ulatuse piiramiseks kehtestati probleemide ja nõuete loetelu. Simulatsiooni arendamise käigus rakendati mitmeid erinevaid lisafunktsioone, et laiendada simulatsiooni funktsionaalsust konkreetsetele vajadustele vastavalt. Selle tulemusena arendati välja mitmeid erinevaid funktsioone: laeva juhtimine klaviatuuri abil koos füüsikaga seotud interaktsioonidega (objektide kokkupõrked, olgu need siis staatilised või liikuvad objektid); dünaamilised (liikuvad) ja staatilised objektid kui teised ‘võõrad’ laevad, mis liiguvad simulatsioonis, ja kivid, rannad ning maapind vastavalt; loodud kaardid erinevate staatiliste ja dünaamiliste objektidega ning viis nende kaartide hõlpsaks lisamiseks või loomiseks; üldine kasutajaliides koos juhtnuppude juhendiga ja liugurid, mis võimaldavad laeva parameetreid lennult muuta (mida on võimalik teha ka konfiguratsioonifaili abil); laevaliikumise salvestamine (talletamine) ja selle liikumise taasesitus salvestatud failist; lülitatav tuulejõu rakendamine laevaliikumisele; TCP (Transmission Control Protocol) serveri seadistamine ja ühendus kliendiga Pythoni skripti kontseptsiooni tõestamise teel; kaamera järgimine ja sisse- ja väljazuumimise funktsionaalsus; kaart demonstratsioonieesmärgil päriselus asuva “Tilgu” sadama kohta. Pärast kõigi funktsioonide väljatöötamist võib mitmeid asju täiustada. Tulevane arendus on võimalik erinevate ideede ja taktikatega silmas pidades.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 40 leheküljel, 5 peatükki, 35 joonist, 2 tabelit.

List of abbreviations and terms

TCP	Transmission Control Protocol
IPV4	Internet Protocol version 4
IP	Internet Protocol
GUI	General User Interface

Table of contents

List of figures

List of tables

1 Introduction

A lot of ship controls nowadays become more and more automated. As the algorithms that control the autonomous ships become more technologically advanced, and the prices and risks of testing them in sea increases as the need for a solution to replace real-time testing. It is possible to save a lot of money on testing using a simulation. Instead of testing them only in real-life it is possible to first test them in the simulation to reduce risks and costs. For this reason, the simulation is created in which different technological equipment, sensors and algorithms for ships could be tested without the costs of testing it mainly in the sea or the ocean.

1.1 Problem definition and constraints

The topic of this thesis is the implementation of a 2D simulation of a ship on the water, controllable by the user. The view of the simulation is top-down, meaning the view at the ship is from the eagle-eye point of view.

The following problems will be solved in the thesis:

- Which kind of platform or a program is best suited for implementing a ship simulation?
- Easily generate maps manually for the ship to traverse through.
- In the simulation describe and simulate physics for the ship movement and maneuvering.
- Simulate real-time ship control using a keyboard on the generated maps.

Initial conditions of the thesis are – there would be created a simulation, in which a ship can be controlled by the user; the ships physics should resemble real-life physics of the ship as close as possible; the collisions between ship and different object (be they static or dynamic) is acknowledged and are happening, to prevent clipping into each other.

The overall goal of the thesis is to find out, is it possible to create a 2D simulation of a user controllable ship using a platform or engine for it to be able to test out different algorithms and testing out different sensors that operate on a real-life ship in the sea/ocean.

This could be used not only to test out existing technologies for the ship, but also the ship construction itself if enough features would be implemented in the future.

The 2D simulation has a lot of potential to be developed further into a bigger project if there is enough interest in the field for testing sailing equipment and technologies.

1.2 Thesis organization

The thesis is split into four distinct chapters.

The first chapter of the thesis talks about what platform/program was chosen to solve the initial problem and complete the goal. Different platforms are compared, and the simulation is created in the chosen platform.

In the next chapter the chosen platform is explained and introduced. Its key features and functionality would be also explained and would be shown how to set up the project. Also, there would be an instruction for the reader on how it is possible to get yourself a copy of the code and continue the development of the simulation.

Then in the third chapter the development of the simulation is explained. Described topics covering each step of development and addition of needed features to the set of functionalities of the simulation.

In the final chapter of the thesis would be explained what could have been done better/or differently, what bugs and errors were encountered and have they’ve been delt with.

At the very end of the thesis a conclusion is made on what was done and do initial conditions, goals and problems with questions were met with, what have gotten completed in the end.

2 Choosing the correct platform/software

For the creation of the 2D ship control simulation first it is important to choose the most suitable and comfortable for use platform. Through searching of different existing platforms were chosen mostly game engines. Non-game engine different engineering software available online either is too specific (like gas movement simulation or mechanics movement simulation) or too obscure for use. In the many different platforms, that were looked through, were chosen the most popular solutions on the internet: Godot game engine [1], Unity game engine [2], PyGame framework [3] and Construct 3 game engine [4]. In the next section is the tables (see Table 1 and Table 2) that compare different aspects of every platform to weigh in pros and cons of each and choose the most suitable.

2.1 Comparing different platforms/software

Table 1. The first part of the platforms comparison.

Platform/engine	Godot	Unity
Programming language	GDScript (+ C++, C#)	C#
Is it free	Yes	Yes
Pros	Very easy to use, a lot of different useful functions already built-in engine	There is a lot of functionality in the engine as well as object-oriented things, that allows to maintain code easily with a team and quickly create projects
	Everything in engine is a node of a tree, this hierarchy allows to easily manipulate, add or remove parts as well as organise and group them	There is a lot of different objects or sprites created by other people, that can be used for free in your own projects

	Very easy to understand code errors	A lot of different manuals and tutorials on how to solve some problem or create something
	GDScript is very similar to Python, and, is a very powerful scripting language, that allows to do complex stuff easily	Engines capabilities are used in engineering and cinematography
	A very fast starting engine that is also lightweight	
Cons	Might not always be an explanation of a problem or a function in documentation on how something works or how to do something, in many instances you must find out yourself	This engine takes a lot more time to get used to
		It takes a lot more time to start the engine and load everything
Can you create communication with other sources?	Yes	Yes
Cross-platform	Yes	Yes

Table 2. The second part of the platforms comparison.

Platform/engine	PyGame	Construct 3
Programming language	Python	Visual programming
Is it free	Yes	No (but there is a free version)
Pros	Not an engine but a framework	Suitable mostly for simple 2d simulations
	Suitable for easy creation of small 2d projects	You can add own blocks using javascript

	A lot of pre-made functions for different scenarios	There is a lot of tutorials on how to use the engine and how to create something
		Very easy to use for non-programmers thanks to vast built-in functionality and visual programming
		You can use the engine in the browser on the phone or tablet
Cons	You must do a lot of things manually (like create a window or load sprites) with code, because there is no user interface	Free version lacks functionality compared to bought version
		It is impossible to control every aspect of the engine
Can you create communication with other sources?	Yes	Not possible
Cross-platform	Yes	Yes

The following criterions were chosen to compare different platforms (see Table 1 and Table 2): the programming language in use; is the platform free or not; the pros and cons of the specific platform; could additional communication with other sources be made, and is the created product would be cross-platform.

Right from the start it is seen that every platform chosen creates products that are all cross-platform, meaning that the made product would run the same on every targeted system (be it Windows, Mac, Linux, etc).

Out of all 4 platforms only one of them has a paid version, which is Construct 3 game engine. Free version lacks in functionality, but overall, the engine is very simple and easy to use, because of visual programming and pre-made function blocks. This platform would not be chosen due to lack of full development control.

The PyGame is a framework, which is a Python library with a ton of functionality and a lot of pre-made functions, but it lacks a general user interface, which would make it harder to use. Also, because everything is controlled through code, even the most basic things such as loading sprites and creating a window for the project would take much more time and effort. Because of those downsides, this framework would not be chosen.

Only left are Godot and Unity game engines and we choose between two.

Unity is a very popular, versatile tool for creation of simulations, games etc. It has a lot of documentation, many different functions, and a big and vast user interface. One of the major downsides of this is that it takes much more time to get used to the engine. It suits more for the experienced type of programmers and user, partially because of the C# use inside the engine. Also, Unity takes much more time to load up the software and many errors that might occur while coding might seem obscure at times.

3 Godot engine and why it was chosen

Figure 1. Godot game engine logo and name, taken from [5].

Compared to Unity game engine, Godot game engine is a fast loading, lightweight and much easier to use game engine (see Table 1 p. 13). It uses a scripting language GDscript, that is very similar to Python, which is very robust and suits for complex problem-solving while being easy to understand and use. There is also a lot of different pre-made functions that allow to create 2D simulations easily.

Because the task doesn't require the complexity that the Unity game engine provides, and the ease of use and simplicity of the Godot game engine is the reason for the selection of the Godot game engine software for the creation of the 2D ship control simulation [6].

3.1 Godot's key features

Everything in the Godot ("Godot" pronounced with a silent 't') game engine (Figure 1) is a node of a tree, be it a physics object, a tilemap (see 4.2.3 for explanation), animation player, general user interface element, etc [7]. Every node can have child nodes, which would create trees of nodes. All grouped together nodes and trees of nodes are called scenes (Figure 2). Scenes can be instantiated to be added to other scenes (for example a 'Player' scene can appear inside a 'World' scene). You can manipulate, create, add, remove, group, and organize every node, tree of nodes and scenes manually or using code. This makes managing projects and structuring them very easy and intuitive.

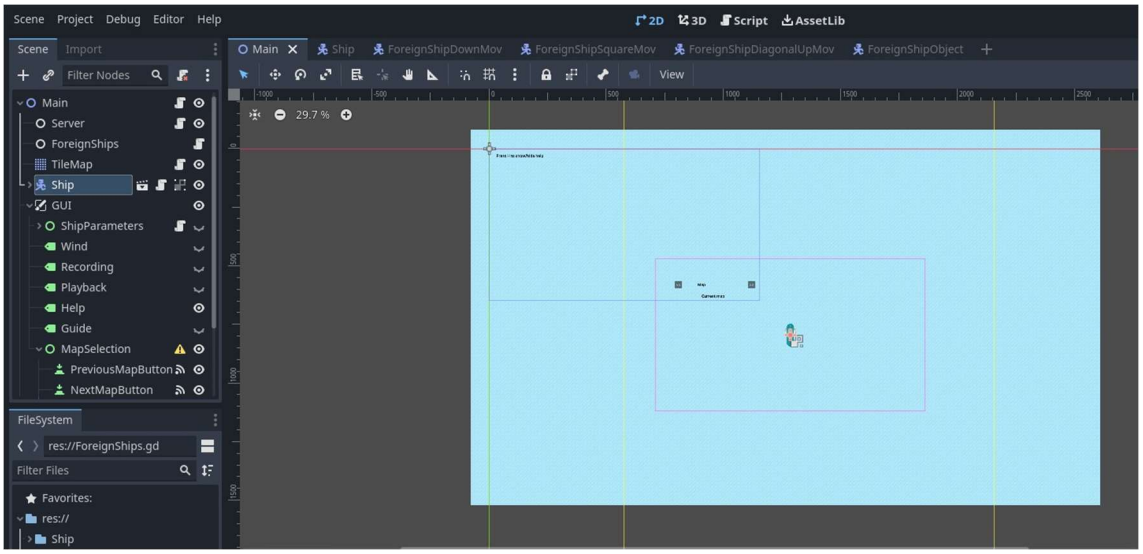

Figure 2. Inside Godot editor with the tree of nodes on the left and an opened scene on the right.

There are nodes with pre-made functionality, like a physics object node for example, which would react to gravity, other physics objects interactions, etc.

The Godot engine is also open source, which allows users to change or tweak any aspect of the engine which doesn't really suit them.

The editor inside the engine allows for code auto-completion, syntax highlighting and auto indentation. The error messages are rarely cryptic, which allows to easily understand the source of the problem. The built-in debugger allows to set breakpoints and step through code.

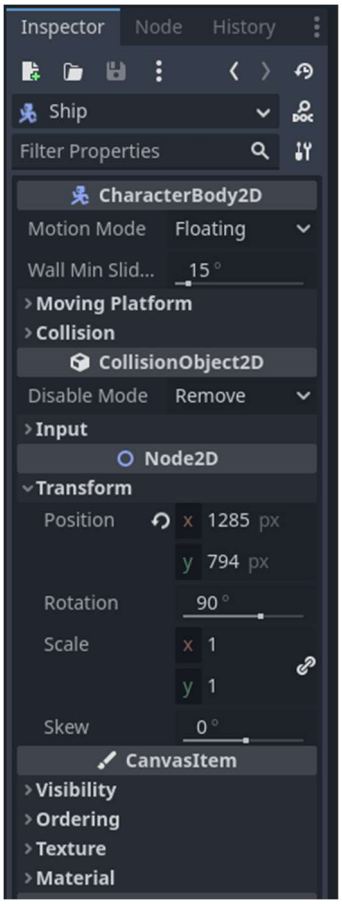

Figure 3. An example of properties of a node that could be changed in the editor.

Inside the editor you can control properties of each node (position, rotation, individual properties to each node type, etc.) (see Figure 3), which is located to the right side of the editor by default.

The scripting programming language of Godot GDScript is specifically optimized to work with Godot, allowing to control every possible aspect of the project. GDScript is a

dynamically typed and object-oriented programming language. In it, every script file can be attached to any node to control the flow of the program, properties of the node, etc. Every script file itself is a class and can extend any other class (encapsulation and inheritance).

3.2 Acquiring yourself a copy of the project for continued development or testing

The installation process for Godot is extremely simple. You must navigate to the official website and download the latest version to the specified system [1].

You will then download a container, which is an executable of the game engine. Unpack the executable and run it.

From there it is very simple to import a project you want to continue to work on. Add the project's repository to your system through GitHub and then use "Import" and add the "project.godot" file from the repository when choosing to import a project (see Figure 4).

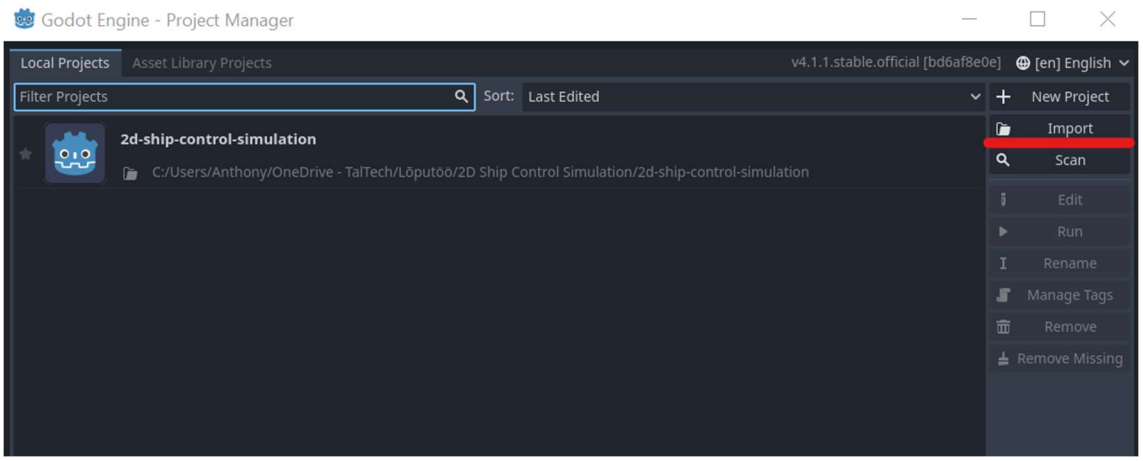

Figure 4. Locating the "Import" button when launching the Godot Engine.

4 Creating the simulation

In the upcoming subchapters would be explained every part and step in the implementation of the simulation with thorough explanation of how each feature was done and developed and how it was set up inside the Godot game engine. Any decision made would be explained on why it is the best suiting and found solutions should be interpreted as best fitting in each scenario and are not definitive answers to any possible or similar problems.

4.1 Setting up the project in Godot

Creating a new project is straightforward – after launching the engine and pressing the "New Project" button the new project gets created. GitHub repository was also set with Godot to monitor and have code version control in mind for other users to access the project. Only 2D mode is intended for the solution, so only 2D and interface nodes would be used. The main root node for the main scene is created and renamed to be "Main". It would have several child nodes and scene instances that would be discussed in greater detail in the upcoming chapters. This scene is set as the main scene whenever the project is run from the editor or from the built application.

4.2 Features development

For the simulation to be complete and for it to meet each goal and requirement set at the proposition of the thesis, certain features should be developed to claim that the simulation is done.

Every feature development is split into its own separate chapter with explanations on how it was done, how it was setup inside the Godot editor, what nodes was chosen and what is the logic behind each script. Each chapter follows a chronological order, but sometimes it is impossible to talk chronologically, so many additions or changes are put directly inside chapters, any references needed would be added in chapters to avoid confusion and allow to follow the development cycle.

4.2.1 Creating a ship with movement using keyboard controls

To start with the creation of the simulation, first the ship is needed to be created with a simple movement system for the user to control the ship. It needs to have a propelling motion forward where the nose of the ship is looking and the backwards movement in the direction of where the stern is. The ship would also need to have turning ability, to turn the ship left and right, turning it counterclockwise and clockwise respectively around its pivoting point. The whole ship should be able to be controlled by arrow keys on the keyboard.

Whenever creating or adding something to the project in Godot built-in nodes of different types with different pre-made functionality are used.

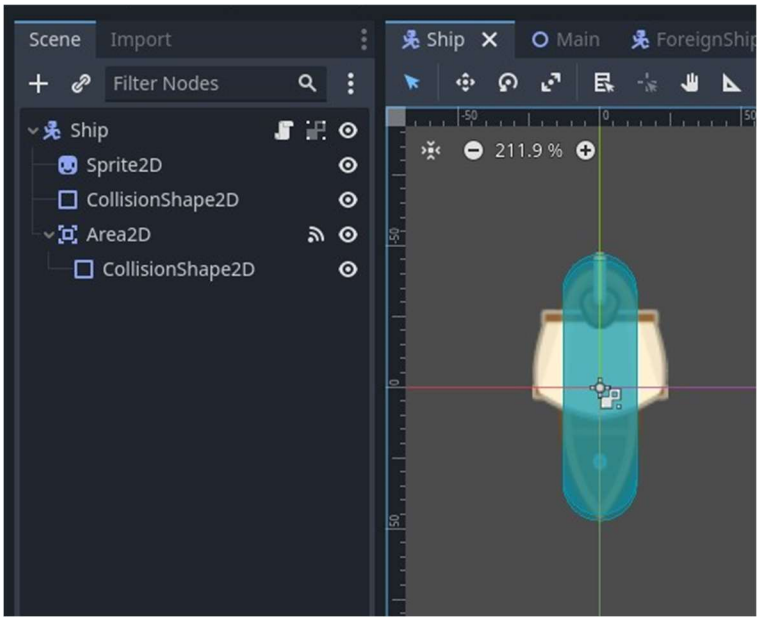

Figure 5. The created “Ship” scene with its nodes on the left and the ship on the right.

For the main controllable ship, “Ship” scene is created that uses “CharacterBody2D” node type, that allows to use its classes functions for controlling the ship with smooth movement and has physics already built-into object, so it would behave accordingly to any change (for example, “velocity” parameter would allow to control the ship easily using the script) (see Figure 5). This node would include two child nodes: the “Sprite2D” node that holds the picture of the ship [8] and “CollisionShape2D” to check for collision. The collision shape must be chosen to automatically collide with any objects. Capsule type collision shape is chosen to easily slide across any object it touches. Additionally,

“Area2D” is added with the “CollisionShape2D” node as its child to encounter any collisions that happen with the ship (later used to encounter “crashing” into static objects).

To control the ship with arrow keys, logic and all required physics are added in code through the script, that is attached to “CharacterBody2D” node.

The functions that control the movement of the ship and code flow are seen on Figure 6.

```
func _physics_process(delta):
    ship_movement(delta)
    ship_rotation(delta)
    move_and_slide()
```

Figure 6. “_physics_process” function that controls the logic for ship’s movement.

“_physics_process” function is running every frame. Ship’s functionality is split into three parts: calculating the ship forward/backward movement, ship rotation and applying the movement vector to it. To better illustrate the code associated with different functions, the “Syntax Highlighter” website [9] was used on Figure 8 and Figure 11.

First, would be talked about how the function “ship_movement” works.

The logic that controls forward and backward movement of the ship inside the “ship_movement” function is split into parts to better explain its functionality. In this subchapter only relevant parts would be explained. Other unrelated parts would be explained in the upcoming sections and subchapters as necessary. The code for the movement was taken from the video that explains the making of the smooth control motion [10].

```
var propel_forward = Input.get_axis("move_back", "move_forward")
```

Figure 7. Getting input for the ship forward or backwards movement.

The “Input.get_axis()” function saves the input from the user and does not allow for simultaneous pressing of back and forward movement (Figure 7). It would return either

1.0 for positive action as moving forward or -1.0 for negative action as moving backwards. If no input was provided, it would then return 0.0.

```
1. if propel_forward:
2.     calculated_velocity += movement_direction * propel_forward * acceleration * delta
3.     if !crash:
4.         calculated_velocity = calculated_velocity.limit_length(max_speed)
5.     else:
6.         calculated_velocity = calculated_velocity.limit_length(max_speed * crash_coeff)
7. else:
8.     if velocity.length() > (friction * delta):
9.         calculated_velocity -= calculated_velocity.normalized() * (deceleration * delta)
10.    else:
11.        calculated_velocity = Vector2.ZERO
12. velocity = calculated_velocity + wind_vector
```

Figure 8. Main logic for velocity vector calculation inside “ship_movement” function.

Whenever there is input from the user, start calculating the velocity to be applied on the controllable ship (Figure 8 lines 1-2). “movement_direction” variable saves current ship orientation, which always changes in the direction of the nose in the “ship_rotation” function, “propel_forward” variable dictates that the ship is moving forward or backwards (this variable plays more of a sign role in the equation) and the acceleration parameter (how fast the velocity would increase to the maximum speed) and delta (time elapsed since last frame) are all multiplied together and are added to the velocity vector (Figure 8 line 2).

Multiplying by delta is necessary to make any precise movement or calculation be independent from different operating systems or machines this project is run on and makes it consistent for everyone everywhere.

The velocity parameter of the ship is a 2D vector. It grows until it gets to the maximum possible speed. To limit the vector to not go over a certain value “calculated_velocity.limit_length()” function is used (Figure 8 line 4).

Whenever there is no input, gradually decrease the velocity of the ship (Figure 8 lines 8-9). While the length of the vector (magnitude of a vector) is more than the value of friction multiplied by delta, decrease the normalized vectors velocity (just get the direction in which the velocity vector is, with vector shortened to the length of 1) and multiplied by the friction parameter by delta (Figure 8 lines 8-9). If the set value is reached, the ship is forced to stay in place (Figure 8 line 11).

Whenever the ship touches the collidable object (be it rocks in the sea or shore land) a "Area2D" node is used to check for collision with its nose or stern. The code then will limit the speed (using the "limit_length()" function) to make it seem like a crash has occurred (Figure 8 line 6).

The function "move_and_slide()" would apply the velocity vector calculated and make the controllable ship move inside the simulation and collide with other objects but would never clip inside of them or get stuck on them and would "slide" around them (also thanks to the chosen collision shape being capsule-shaped with smooth corners) (see Figure 6).

To better understand the logic behind the algorithm, look at the Figure 9:

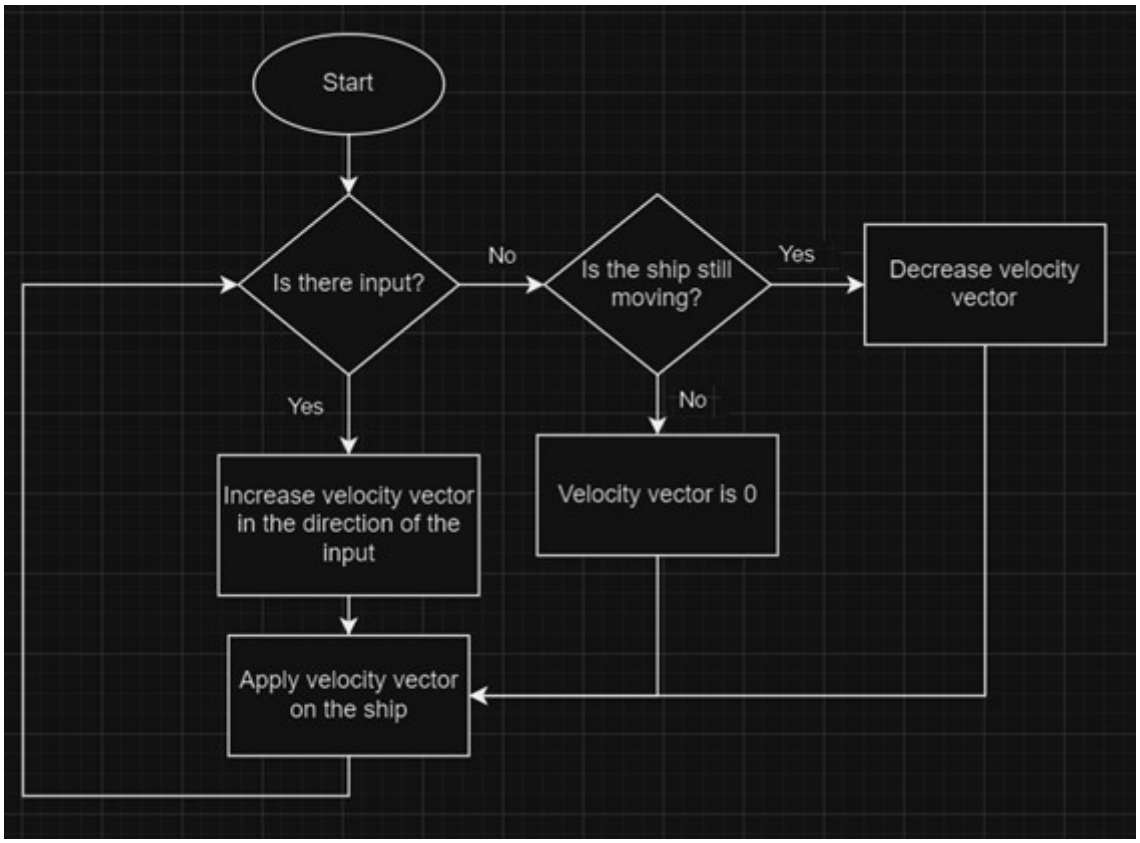

Figure 9. Simplified algorithm behind the main logic in "ship_movement" function.

Next is the function that controls ship's rotation inside "ship_rotation()" function.

It is much more complicated than the "ship_movement" function because there would no more be vector values but the degrees of rotation (or radians). The logic in the simulation

is to rotate the ship around its pivoting point which is in the centre of the ship's body. There would be no applied forces whenever rotation happens which would intrinsically move the ship as well.

```
var rotation_direction = Input.get_axis("turn_left", "turn_right")
```

Figure 10. Getting input for the left or right rotation of the ship.

The logic and functionality from the function "ship_movement()" for getting the forward/backward movement input from the user (see Figure 7) would be applied here as well (see Figure 10).

```
if rotation_direction:
    if rotation_direction != last_rotation_direction and rotation_direction_change == false:
        rotation_direction_change = true
        rotation_before_stop = last_rotation_direction
    last_rotation_direction = rotation_direction
    if rotation_direction_change == false:
        if !crash:
            rotation += rotation_direction *
            acceleration(rotation_velocity, max_rotational_velocity, delta) * delta
        else:
            rotation += rotation_direction *
            acceleration(rotation_velocity, max_rotational_velocity * crash_coeff, delta) * delta
    else:
        if rotation_velocity > ROTATION_STOP:
            rotation += rotation_before_stop * deceleration(rotation_velocity, delta) * delta
        else:
            rotation_direction_change = false
else:
    if rotation_velocity > 0.0:
        rotation += last_rotation_direction * deceleration(rotation_velocity, delta) * delta
movement_direction = Vector2.from_angle(rotation)
```

Figure 11. Logic and functionality inside the "ship_rotation" function.

Everything else follows the same logic from the "ship_movement" function (see Figure 8) – if there is an input, move towards that input, if no input, then gradually slow down. But the logic has a few noticeable additions. First, there is no manipulation of the velocity vector but the rotation in radians, as there is no vectors involved in calculating. Also, when there is no input, "rotation_velocity" should continue to increase (this variable stores and changes the rotational speed in radians per second) to make it seem as if the rotation stops gradually, because if there are values to be subtracted from the

“rotation_velocity” variable, the ship would then start to turn in the opposite direction (see Figure 11).

Functions are created to increase (accelerate) or decrease (decelerate) the rotational velocity (see Figure 11 lines 9, 12, 15, 20). They increase/decrease the “rotation_velocity” by the same constant values from “ship_movement” function (acceleration and deceleration constants). The acceleration function also limits the maximum velocity if the current velocity is more than maximum possible.

Those functions are used to calculate the rotation of the ship gradually. The crash functionality would also be applied whenever the ship is touching any static object in the simulation (see Figure 11 lines 10-12).

The rotation velocity is only applied whenever there is input from the user. So, when there is no input, we gradually slow down the ship’s rotation until full stop (see Figure 11 lines 18-20).

User is not allowed to suddenly change the rotation. The ship also can’t retain its maximum velocity whenever the change in the rotation direction happens.

If there is a rotation change and the direction is different from the last – then last rotation direction is saved (see Figure 11 lines 2-4) to continue adding the rotation velocity in radians gradually as deceleration to the ship’s rotation when we want to change the rotation (to make it not sudden but slow down until we reach the speed that allows us to finally start turning in the direction) (see Figure 11 lines 14-15).

Last direction is always saved in which the rotation happened (either left or right direction) to remember where to continue adding the rotation as deceleration when there is no input (see Figure 11 line 5). It would also be used to check when there is a rotation change.

If there is a rotation change, decelerate rotational velocity and continue adding that velocity to the rotational value of the ship. Continue doing that until the value is lower than the “ROTATION_STOP” value. This will ensure that the ship would not make a sudden change in rotation, but gradually slow down until accelerating again in the new direction (see Figure 11 lines 13-17).

In function “ship_rotation()” the “movement_direction” vector, which was talked about earlier, is calculated from the rotation (see Figure 11 line 21). 2D vector is calculated to be in the direction in which the nose of the ship is looking at right now and it happens every frame.

To better understand the rotation function of the ship, look at Figure 12:

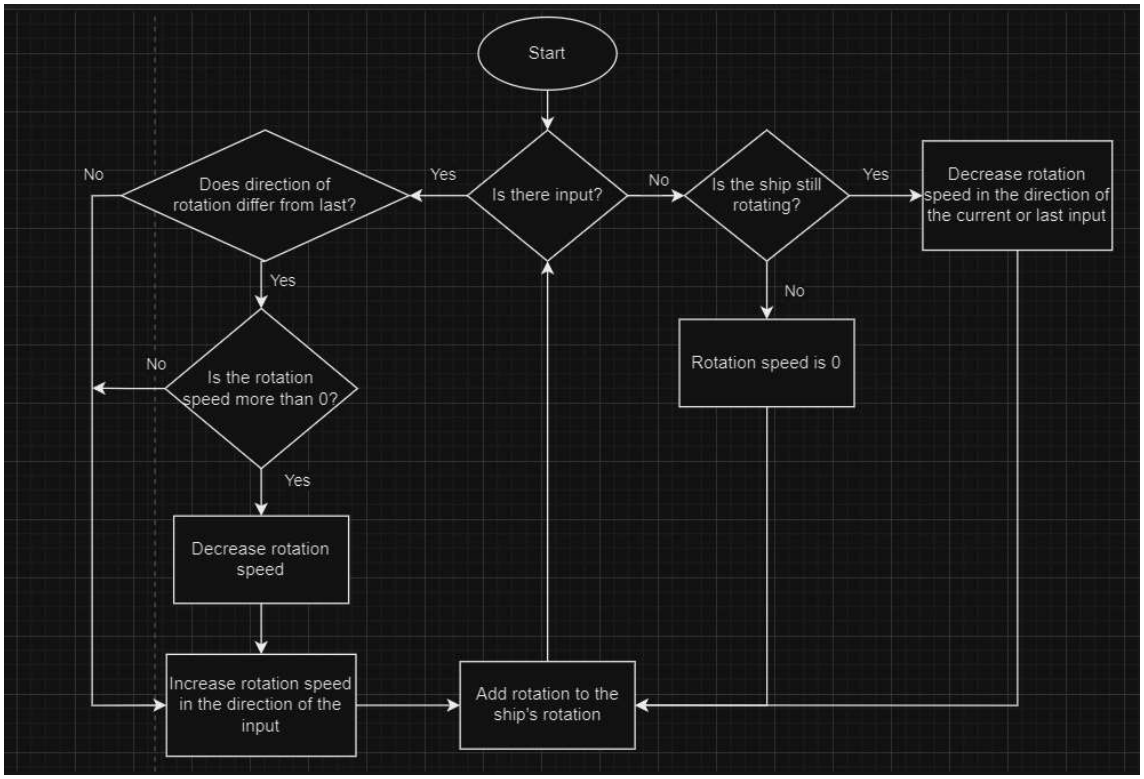

Figure 12. Simplified algorithm behind the main logic in “ship_rotation” function.

To create flowcharts for logic of both functions on Figure 9 and Figure 12, a website for diagram creation was used [11].

4.2.2 Creating a general user interface

For the user to understand the controls additional information appears on screen. To signify changes in the simulation general user interface is used. It is very bare bones but allows to understand many things for a new user.

First, a canvas layer node is created that would group all other interface nodes in it, because the canvas layer node is always on top of the screen. For example, GUI always appears on the screen, even if the camera would always follow the ship.

Next, label nodes are added to explain controls and are put as child nodes for the canvas layer node. There would always be a “Help” label on the top left corner to remind how to make the “Guide” label show up (using the ‘H’ key on the keyboard). Whenever the key is pressed, the label would appear on the screen explaining the controls and functionality (see Figure 13).

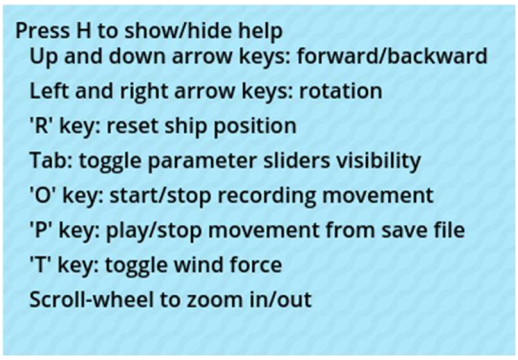

Figure 13. Guide and help labels with text inside the simulation.

In other chapters other interface parts would be added to the GUI in the canvas layer node to simplify explanation.

4.2.3 Creating a tilemap with collisions, adding maps and switching between maps

To make maps easily from the scratch the node “TileMap” should be added as a child node to the “Main” node. This node allows creating tiles, add properties to them and use them as building blocks for making a map.

A tile is some graphical picture that is of certain width and length and is used as a building block for creating maps out of them. The tilemap is a collection of tiles that are grouped together in the tileset to easily create maps out of tiles. Tiles are placed on the grid section.

A premade tileset was downloaded to use in the Tilemap [8]. After adding the tileset, specify the places where each tile is (see Figure 14) in the tileset and which ones to use. As the premade tileset was made with each tile being 64 by 64 pixels, each tile would be with this size. The tile shape and the tile size is set in the Godot Editor (see Figure 15)

Figure 14. Each specified tile in the downloaded tileset inside the tilemap.

To make the tiles be somehow 1:1 representative of the real-world size the tilemap should be scaled down to make a tile be approximately 1 by 1 meters. In this case, the default ship is around 6 by 2 meters long.

After setting up tiles in the tileset the collision shapes for each tile which needs a collision shape would be set as well. It is all also done through the TileMap node. The tiles should have a collision for the ship to collide with them and not be able to move through them. This would be set through the Physics Layers collision layer to be on the 1st and the 2nd layers. 1st layer would make the ship collide with the objects that have a physics layer added to them and a 2nd layer would allow the Area2D node added to the ship to check if it encountered a collision with the tile to apply a crash coefficient to limit maximum speed and rotational velocity. Then the collision shapes are set for each individual tile in the Tileset section (see Figure 16).

Figure 15. Tilemap properties with tileset's tile shape, size and physics layers.

Figure 16. Collision shapes for each individual tile in the tileset.

After setting up the collision shapes, finalise setting up the TileMap for map creation with the layers for tiles which would be as individual "levels" for different tiles (see Figure 17). Two layers are created for map making: the water level layer, which is always water tiles as background which never changes nor interacts with any ships (does not collide), and an above water layer, which is where maps made of tiles would be at.

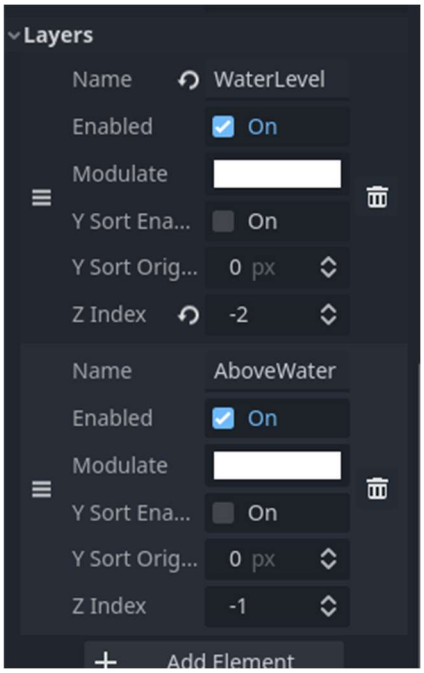

Figure 17. Tilemap property “Layers” inside the Godot Editor.

Z indexes dictate how the graphics would be rendered to the scree, meaning what is on top of what and what can be seen right now. If the Z index would be more than zero, this means that it is closer to screen. When it is lower than zero, it is more far away and would be hidden behind other objects.

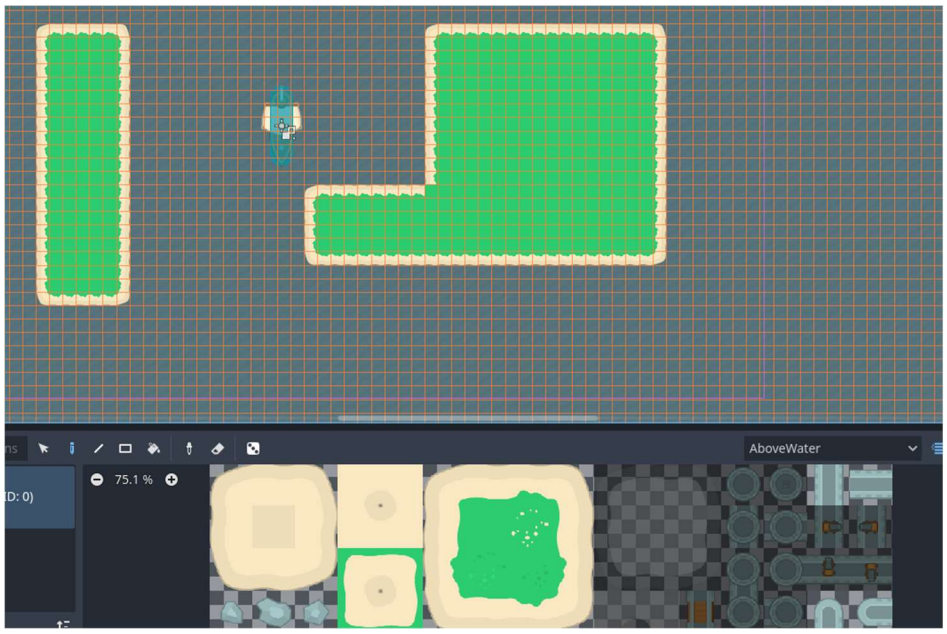

Figure 18. An example of creating a map on the “AboveWater” layer.

After everything was set up for map creation, every new map would be done on the above water layer (see Figure 18) and saved in the “Patterns” section to access them later using scripts (see Figure 19).

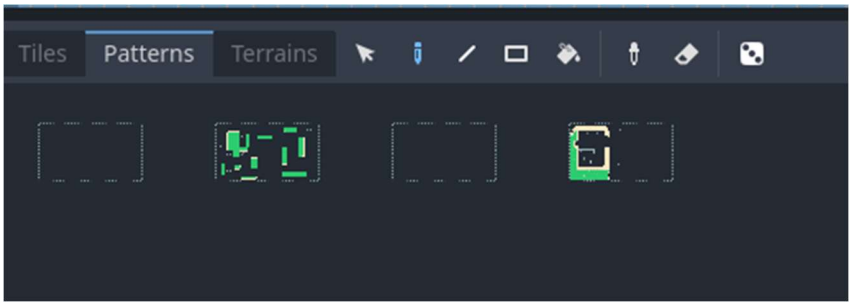

Figure 19. Different maps saved as “Patterns” inside the tilemap.

The created maps would be: the clear map for roaming around freely and getting to know the controls and features of the simulation (the simulation always starts on this map), the obstacle map for roaming around with different islands and tiles to interact with their collisions, the dynamic objects map (more on that in the later parts of the thesis), and demo “Tilgu” port map (a recreation of the real-life port, more on that in 4.3 p. 45).

After creating each pattern, the script is attached to the “Tilemap” node for changing those maps (see Figure 22). After a few different renditions of this script, first was used key bindings to numbers (from numbers 1 to 3, changing maps by pressing each number key accordingly) which did not fit. The version using GUI elements such as buttons to switch between the maps was chosen (see Figure 20).

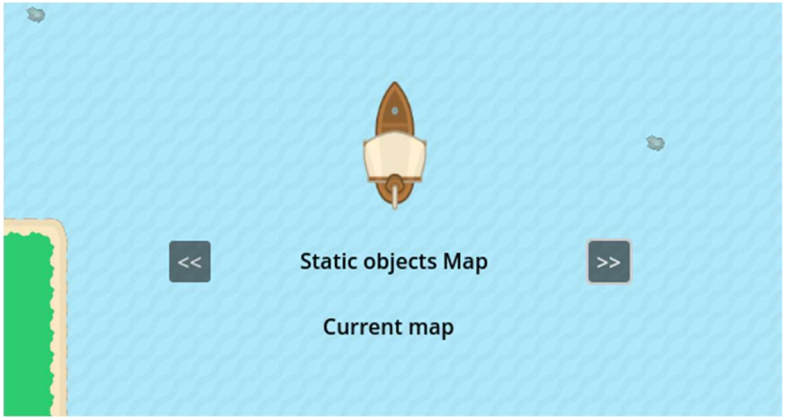

Figure 20. Buttons in GUI for changing maps.

```
var maps = {0: "Empty", 1: "Static objects", 2: "Dynamic objects", 3: "Tilgu port"}
var n_maps = maps.size()
var curr_idx = 0
```

Figure 21. Maps saved in the dictionary to keep track of what map is currently chosen.

In the script a dictionary variable “maps” is created which would hold the index of the map and the name of the map (see Figure 21).

```
func switch_map(map):
    emit_signal("map_switched")
    clear_layer(1)
    set_pattern(1, Vector2i.ZERO, tile_set.get_pattern(curr_idx))
    match map:
        "Dynamic objects":
            emit_signal("add_foreign_ships")
        "Tilgu port":
            emit_signal("add_tilgu_ships")
    map_label.text = maps[curr_idx] + " Map"
```

Figure 22. “switch_map” function logic.

Following the logic of the function “switch_map” line by line (see Figure 22), whenever a map is changed (which happens by the press of buttons on the screen added in the GUI) the signal gets emitted which will tell a function in the same or other script to activate. In this case, the controllable ship’s position would be reset to the start. The tilemap layer gets cleared (“AboveWater” layer is with an index 1) and set to the pattern according to the index (we set the pattern from the patterns list on the 1 index which is a “AboveWater” layer). All the patterns in the Patterns section start from the index 0, so each map’s index is same as in the dictionary “maps”. Depending on which map it is, dynamic objects are added (discussed in detail in 4.2.4 p. 34). The label‘s text on the screen between two buttons would also change to a current maps name.

Whenever the button is pressed on the screen to change the corresponding next or previous map, first check is the index not out of bounds, after that clear any ships from the screen and increase/decrease the index and call the “switch_map” function which was talked about previously (see Figure 22).

4.2.4 Creating movable and collidable objects as other ships

The simulation would also need to have not only static objects which collide with the controllable ship, but also have other ships moving around and could collide with the controllable ship.

For this to be done separate scenes are created which would be instanced many times either dynamically (more on that in the chapter 4.2.5) or with map changes.

Three different ships with different properties and functionality are made. One version of dynamic movement is to move diagonally up, the second is to go down and the last one is to move in a square pattern, repeatedly. Each ship version has different speed, reset timers, size and so on. Individual scene for each version is created with its own script attached (see Figure 23).

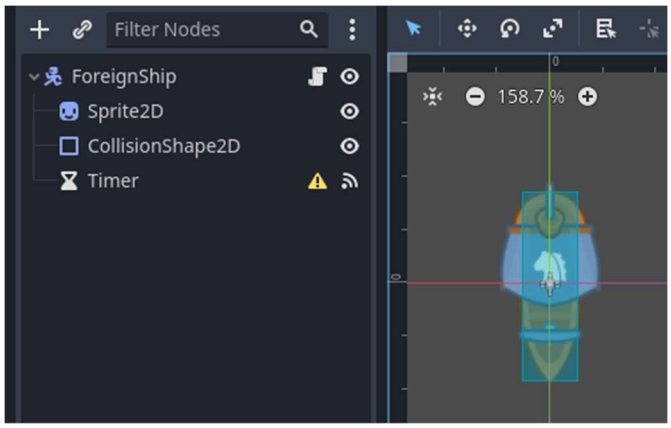

Figure 23. An example of an individual scene with its nodes for a foreign ship.

To differentiate between the controllable ship and other ships that are uncontrollable, the other ships would be referred to as “foreign” ships from now on.

Each individual scene consists of a “CharacterBody2D” node with “Timer”, “CollisionShape2D” and “Sprite2D” nodes as its child nodes (see Figure 23). The sprite would be set to the same sprite for all foreign ships to be a different from the controllable ship picture from the downloaded tileset. The collision shape for each version is the capsule and collides with the controllable ship and static objects (tiles from the tilemap).

Each individual script is attached to the “CharacterBody2D” node. Whenever the foreign ship version appears on the screen, its scale, speed, rotation is set, and it begins to move constantly until the timer reaches 0.0. Whenever timer reaches 0.0, the ship version resets its position back to the original (for the square movement it would turn 90 degrees and continue movement).

To be able to add these foreign ships to the simulation whenever a change to the dynamic objects map happens, the script for adding foreign ships is created. It is then attached to the empty node, which would be a child node to the “Main” node in the main scene (see Figure 22).

The signal to add these ships is emitted whenever the index of the map matches to the index of the second dynamic objects map.

```
# Preload each individual foreign ship
var foreign_ship_v1 = preload("res://Ship/ForeignShipDownMov.tscn")
var foreign_ship_v2 = preload("res://Ship/ForeignShipSquareMov.tscn")
var foreign_ship_v3 = preload("res://Ship/ForeignShipDiagonalUpMov.tscn")

# Declare new foreign ships types here...

# Foreign ships dict, their corresponding position on the map and type of foreing ship
var ships = {Vector2(550, 50): "1", Vector2(1600, 200): "2", Vector2(1100, 1350): "3", Vector2(790, 50): "1"}
```

Figure 24. Declared different versions of foreign ships with dictionary of positions and versions.

The dictionary is created inside the script, with foreign ships corresponding positions inside the simulation as dictionary keys and their values being their version type (see Figure 24). Whenever the signal gets emitted, the ships are added by their version type and their position would be the one from their keys inside the dictionary. Whenever the map gets changed the signal gets emitted to remove any foreign ships from the screen.

```
func add_new_foreign_ship(speed, starting_rotation_deg, reset_time_s, scale, position):
    var new_foreign_ship = foreign_ship_vX.instantiate()
    var foreign_ship_object = new_foreign_ship.init(speed, starting_rotation_deg, reset_time_s, scale, position)
    add_child(foreign_ship_object)
    new_ships[new_ships_key] = foreign_ship_object
    new_ships_key += 1
    #remove_new_foreign_ship(0) # Testing purposes
    return foreign_ship_object

func remove_new_foreign_ship(idx):
    new_ships_key -= 1
    var obj = new_ships[idx]
    obj.queue_free()
    new_ships.erase(idx)
```

Figure 25. Dynamic addition and deletion of foreign ships code.

The code for dynamically creating foreign ships during run-time is also created (see Figure 25). For this, a template foreign ship is made as a separate scene the same as others where – with "CharacterBody2D" node and its child nodes "Sprite2D", "CollisionShape2D" and "Timer".

Additional functions for adding and removing ships dynamically (see Figure 25) is added to an existing script that controls foreign ship loading on the map.

Whenever the function to add a new ship is accessed, it would instantiate [12] a new foreign ship scene and add its object in the simulation and save it into the array to keep track of all the objects that were created dynamically. The count of objects inside the array is also tracked, increasing, and decreasing the amount depending on what function was accessed (either ship adding or removing).

Whenever the function to remove a ship from the array where each object is kept track of, the index of that object inside the array is passed as an argument whenever the function is accessed and is removed from that array, removed from the simulation and the number of objects inside the array are reduced.

4.2.5 Creating ship parameter manipulation through config files and parameter sliders

The controllable ship has multiple distinct parameters that should be manipulatable to fit the need to test different types of ships with varying speed, size, deceleration, friction, acceleration, and such. To be able to load in different parameters whenever the simulation is run, the config file configuration is added [13]. It would take parameters and values

entered in the file to load them in the ship parameters whenever the simulation is started (see Figure 26). If the simulation is run first time on any machine the file is created with default values. The config file is created in the path “user://” which is a user’s project folder inside the Godot folder in a “Roaming” folder (on Windows operating system). The file should be delt with precaution because the broken file or incorrect values could in turn give unpredictable behaviour. Also, any values could be entered outside of possible (see Figure 26) which in turn would make the ship not being able to move or anything could break the simulation.

```
1  [Ship_Parameters]
2
3  weight=1                      ; 1-2.2
4  max_speed=150.0               ; 30-250
5  acceleration=400.0            ; 0-800
6  deceleration=100.0            ; 0-500
7  friction=150.0                ; 0-300
8  max_rotational_velocity=0.8 ; 0.05-1.5
```

Figure 26. Ship’s parameters inside the config file with default values on the left and ranges on the right.

The config file should be used only for configuring or loading parameters from different ships and should be used to debug or test things for the developers. Any other user could use another feature, such as sliders for the parameters in the range to change real-time (see Figure 27) to see changes to the physics of the ship’s movement.

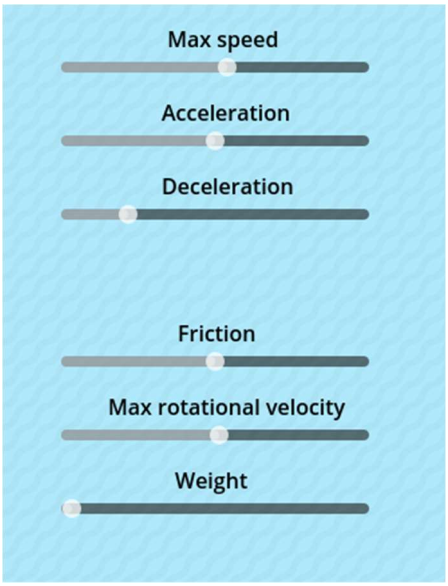

Figure 27. Sliders for changing parameters real-time inside the simulation.

The idea for slides creation comes from a Youtube video explaining volume sliders creation in Godot [14]. For this to be able to be done built-in sliders from Godot Editor is used. These are nodes for each possible changeable parameter. These are all added into the GUI node with labels for each slider to show what parameter it would change. They can change any ship parameter whenever the simulation is running. Built-in sliders can be hidden or shown with the press of a 'TAB' key. Whenever the slider changes values, the signal (a handler of events that gets raised) gets emitted which then applies the changed value on the ship's parameter. Only when the size of the ship is changed, this would recalculate each other parameter, because the ships size dictates other parameter values as well – the bigger the ship, the slower it is and the slower the deceleration, acceleration and so on.

Whenever the simulation is run, the ship's parameters get loaded from the config file and then put as current values on the sliders. The sliders, as it was mentioned, are only visible whenever the appropriate key is pressed and can be manipulated with only when they are visible.

4.2.6 Creating ship movement recording and playback

For the possibility to test out different algorithms it should be possible to record movement that are done by the ship. Recording starts from the moment the "start record"

action is being done until stop action. Then it is possible to recreate the same movement done during recording at the playback action.

For this to be possible the ship's functionality must include additional functions to record and playback movement. Two new functions are created to suit these needs and added inside the ship movement (see Figure 28 and Figure 30).

```
1. func record_input(input_matching, input1, input2):
2.     if is_recording:
3.         var string
4.         match input_matching:
5.             1.0:
6.                 string = input1
7.             -1.0:
8.                 string = input2
9.             0.0:
10.                string = "NONE"
11.        input_file.store_line(string)
```

Figure 28. A function for recording input for movement.

The first function is for recording input (see Figure 28), and it is done on each frame. The file gets created and saves every input from the forward/backward movement as well as separately from the rotation to the right and left right after the user enters or doesn't enter any input from the keyboard. It translates the inputs -1.0, 0.0 and 1.0 to appropriate string values from the function it is being run from. For this, the function would accept three arguments, two of which are the desired strings to be converted to and the last one being the value to compare to the floating values mentioned before (see Figure 28 line 1). In the case when the movement is translated to string, -1.0 and 1.0 are both "DOWN" and "UP" strings respectively. For the rotation -1.0 and 1.0 floating values are "LEFT" and "RIGHT" strings respectively. In the case of 0.0 would indicate the "NONE" string. The file (see Figure 29) is saved to the same user path as the config file (see 4.2.5 p. 36) using the file saving system and ideas from the official documentation [15]. There is a restriction for the recording file to not go over the size of 30 MB. Also, whenever the button key for recording is pressed, the label with text "Recording" appears on the top side of the screen and disappears whenever the same button was pressed, or the file reached its maximum size.

```
125  UP
126  NONE
127  UP
128  NONE
129  UP
130  RIGHT
131  UP
132  RIGHT
133  UP
134  RIGHT
135  UP
136  RIGHT
137  UP
```

Figure 29. An example of saved input inside the recorded file.

```
1. func playback_input(input1, input2):
2.     var input = input_file.get_line()
3.     match input:
4.         input1:
5.             input = 1.0
6.         input2:
7.             input = -1.0
8.         "NONE":
9.             input = 0.0
10.    return input
```

Figure 30. A function for playing back the recorded input from the saved file.

For the ship to recreate and playback all the recorded input the second function must be used (see Figure 30). The function appears after taking the input from the user, so no matter what user would press on the keyboard, no inputs for controlling the ship would manipulate it in any way aside from the playback function. The function opens the aforementioned file and reads saved string input for the movement first and then for the rotation. It takes two arguments as two different string inputs to translate to floating values (see Figure 30 line 1). Both arguments are the input values from the file to compare to. The file consists of lines of single strings of either every odd line being “UP”, “DOWN” or “NONE” strings and every even line of strings in the file being “LEFT”, “RIGHT” or “NONE” (see Figure 29). So, for the movement the arguments would be “UP” and “DOWN” and for the rotation it would be “RIGHT” and “LEFT” strings that all would be translated to 1.0 and -1.0 respectively (see Figure 30 lines 5, 7). After the translation happens the function returns the floating value, and it is applied as if the user has made

the input himself inside the function to move the ship in the simulation. The label with text to indicate that the playback is happening appear at the top side of the screen. After that the ship would repeat every frame what was recorded beforehand, and it does not place the ship back at the centre of the map, so it would repeat all the actions from the last place the playback was invoked from. It would repeat all the inputs saved in the file until it finishes the whole file, or if the playback was stopped.

Recording and playback actions cannot happen simultaneously. The recording or playback action can be stopped at any time by the press of the same key button that invokes the action.

4.2.7 Creating TCP server in the simulation through Godot and TCP client in Python

To setup the connection between the client and the server using TCP sockets first the client connection is created using Python programming language to test out the TCP connection working. All the code and functionality was used from the video and would be explained in this chapter [16]. The Python programming language is used to create the TCP client/server connection because of simplicity and ease of creation from ground-up.

```
import socket

HEADER = 64 # How many bytes the length of the message is

PORT = 5050
SERVER = "192.168.0.248"
ADDRESS = (SERVER, PORT)
DISCONNECT_MESSAGE = "DISCONNECT"
```

Figure 31. Constant variables declaration and an imported ‘socket’ library inside “client.py” file.

Inside the “client.py” file first import a library called “socket” which has all the functions needed for the creation of the connection (see Figure 31). This library comes with the Python installation and doesn’t require any additional setup or library installation.

Then, for the client to establish the connection, it needs to declare the port that would be used and the address to which to connect to. For this, the port that is not in use right now

on the network must be used so in this case 5050 is a safe choice, as it is usually not occupied (see Figure 31). Then, check the IPV4 address on the usable device that would run the Godot simulation. Then, use this IP address to connect to, in this case, the server is started and connected to on the same device, so the local IP would be 192.168.0.248 (see Figure 31). After that, a Python tuple is made with the server IP address and the port to which connection would happen to. Tuples in python are unchangeable collection of data, so they are a constant variable that holds the whole IP address with the port. A disconnect message would be declared which is a string that would, if sent to the server, tell it to disconnect the client from the connection established. The part of the header variable and its value would be talked about in the upcoming paragraph in this section.

To make it possible for other devices around the world to be able to connect to the server with the “client.py” code and send out messages, ports should be opened on the network.

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(ADDRESS)

# First, send the amount of bytes the message will be, and
# pad the length with the amount of bytes in the header
# Example '5    ', then 'hello' etc
def send(msg):
    message = msg.encode()
    msg_length = len(message)
    send_length = str(msg_length).encode()
    send_length += b' ' * (HEADER - len(send_length))
    client.send(send_length)
    client.send(message)
```

Figure 32. Main driving logic of “client.py” file for sending messages.

First and foremost, establish the connection to the address that was made using the IP address and the port from the tuple. A basic socket is made for the client and connects to the made address (see Figure 32).

Next, a function for sending out messages is created. A variable header is created (see Figure 31), which has a value that would tell the program how long the message is sent out in bytes. This header length is used in the function to send out messages. The “msg” parameter is the string that we want to send to the server.

First, the message that is desired to be sent out is encoded using UTF-8 encoding (the function “encode()” would encode in UTF-8 encoding format if no parameter was specified). The length of that encoded message is saved in the “msg_length” variable.

The header length in bytes is needed to send out any messages to the server, because the exact number of bytes needs to be specified whenever expecting a message of some sort of length. For this to be possible with different lengths of messages, first the length of the message is sent and then the message of that exact length. Here, the length of the message with the size of the header value is sent, padded with whitespace to the right that many bytes how many are declared in the header. This is made possible by converting the length of the message from a number to a string and encoding that said string. Now, add to the end of the byte encoded variable that many whitespace characters to fill in the variable until it is of 64 bytes (as in the example Figure 31). That created string is sent out and after that the encoded message (see Figure 32).

Again, the idea is that the server side would receive the first message with specified length (as provided in the header). It would first get the length of the second message and then use that length to receive and decode the next, second message to finally get the sent message.

Now, the server side would be implemented in the Godot engine using an empty “Server” node and an attached script for handling the connection using the aforementioned logic (materials were used from the source on setting up server side in Godot [17]).

The same variables are declared as was done in the Python file to know the IP address with the port on which the server would start. Also, the header length in bytes and the disconnect message are present as well.

Whenever the simulation is started, the “Server” node appears in the tree and the code to start the server is run. A new TCP server is created and starts listening on the port and address provided.

Whenever a connection is encountered, accept it, and save it in the variable and set that the client has connected. Then check if the client has lost connection at any point by checking its status.

Get the message length, decode it and check if it is not empty. Then convert the value from string to integer and use this value to determine the length of the upcoming message with the actual string sent out. Whenever the client would send out the disconnect message, that would mean for the server to close the connection to the client.

The idea of using TCP client/server connection is for the future development and usage in the simulation it would be theoretically possible to send out different signals to the simulation to control the ship remotely. It means that another human on the client side could control the ship as well as an algorithm could control the ship entirely, as we can send info both ways – from the server side to client and vice versa.

4.2.8 Creating a constant wind vector toggler

This additional feature is added in the way to enhance the simulations capabilities to mimic the real-world physics and add another variable that can change and manipulate the way the ship is controlled. The constant wind vector addition is simple mathematical equation to calculate how the wind impacts the ships real direction (see Figure 33) but it is important that the velocity vector of ship is only modified, but not altered. So, the wind vector is only added to the calculated velocity vector and only then applied to the ship.

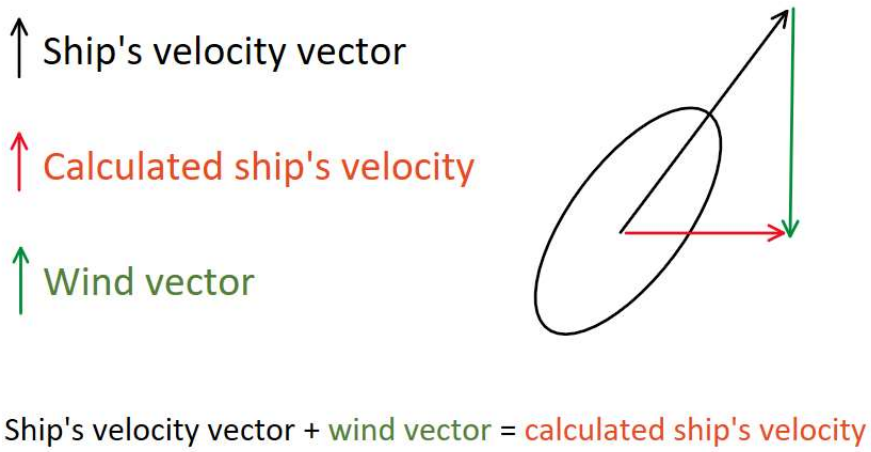

Figure 33. Wind vector application on the ship's velocity vector calculation.

The ability to toggle the wind vector could be added whenever the corresponding key is pressed, the label with the text appears, showing that the wind vector is being applied to the ship's velocity vector. The wind vector becomes non-zero valued vector and gets added to the ship's velocity to make a calculated velocity vector. Whenever the key is pressed again, the wind vector becomes a zero-length vector and whenever the vector of zero-length gets added to another vector, then the resulting vector is the initial vector.

The value of the wind vector could be changed inside the script and is constant.

4.2.9 Creating camera follow, screen resize and scroll-wheel functionality

To be able to zoom in and out to see the bigger part of the screen a camera node is added in the "Main" node as a child. The camera would follow the "RemoteTransform2D" node added as a child to the ship scene in the "Main" node. So, the camera follows any movement the ship makes. It is also possible to set limits for the camera to not go out of bounds in the simulation [18]. The camera node would have a script attached to it that would allow to zoom in and out using the built-in functionality [19]. Whenever the scroll-wheel on the mouse is turned, the "_input()" function gets invoked and checks what press it was – either scroll-wheel down or up. If the value of the zoom goes outside of the declared constants, then the zoom in or out would not happen. If the zoom happens, it is happening by a constant value of the speed as declared in the variable.

Also, screen resizing is set to make the objects and things inside the simulation to remain their size, but the screen would show more of the simulation world around the ship (it is the default sizing option whenever creating a new project inside the Godot game engine).

4.3 Creating a real-life replica of the "Tilgu" port for showcase purposes

For demonstration purposes the map based on the real-life port named "Tilgu" is reconstructed. The simulation uses all created during this work features. First, the reference picture is taken from the location. The picture (see Figure 34) used is taken from the Google maps location [20]. It is later placed in the simulation on the background layer and tilemap is used to draw the tiles over it to make it be a 1 to 1 representation (see Figure 35). Three different ships with different timer resets, speeds, sizes, and angles at which they move as well as positions, are added to this map using functions for dynamical

addition of foreign ships with different parameters (see 4.2.4 p. 34). Those functions are invoked inside the script that controls the addition of foreign ships to the maps where they should be and the fourth and final map pattern in tiles is created as well as a name for the label to change whenever the final map is chosen on the screen.

Figure 34. ‘Tilgu’ port picture from the satellite on the Google Maps [20].

Figure 35. A close representation of a ‘Tilgu’ port inside the simulation.

This map is a good example because the real ship frequently visits this port. This map showcases the capabilities of the simulation and how could algorithms or equipment be tested on the real-life places after making and designing the location.

4.4 Discovered bugs and errors

After completion of the simulation some problems were discovered inside the editor or the simulation itself that were not resolved due to time limitations or the lack of knowledge on the subject.

One of which was that the ship’s rotation’s sudden change is impossible after gaining some reasonable speed but repeated pressing of different directions (such as right-left keyboard pressing) would still turn the ship instantaneously and would remain with the same speed (which is relatively low but still is more than 0.0).

The other discovered problem was inside the Godot editor itself. For any particular reason after adding each GUI element under the “CanvasLayer” node, every first letter in any label with text would not move together with the camera zoom functionality added.

Also, the anchor point for every GUI element should match the screen resizing method, but they are somehow connected to the viewport inside the Godot editor the size of which could not be changed, so the position of the GUI elements for the full-screen and windowed screen is the same. So, if the map change buttons and sliders are on the right side, in the full-screen version they should be on the right side as well, but they remain in the middle, which is wrong.

5 Summary

The goal for this thesis was to create a 2D simulation of a ship on the sea from a top-down view, which allows the user to control the ship using a keyboard. A set of problems and requirements were defined. First, simulation engine was chosen, the best suitable for our goal turned out to be Godot game engine. During the development of the simulation, different features were implemented to match the requirements and expand the functionality of the simulation. The following implemented features expanded the simulations functionality: static objects, groups of which would form maps (the static objects could be collided with); dynamic objects (meaning moving around) of different kind labelled as 'foreign' ships, which are also possible to collide with; GUI elements to control the ship's parameters and change made maps; ship movement recording and playback for the saved movement from the file; TCP server setup with a possibility of a Python script client connection to transfer data; wind power applied to the ship movement which could be toggled on and off; camera follow and zoom in and out functionality. The result of the development is a simulation that is capable to control the simulating ship with relatively close to real-life physics and with many different features for testing out navigation algorithms or equipment.

The created simulation met all set requirement. Every feature was added and developed to fit the specific needs of the simulation.

Few problems and errors were encountered and described.

5.1 Future work

It is possible to improve on the created features and continue development of this simulation to expand and make it more fitting to a ship's equipment and algorithm testing.

One of the possible things is that the set of pictures used for this simulation might be too childish or game-like looking and could be switched in the future for a more realistic look.

After the TCP connection development and proof of concept done, it is possible to continue developing this feature so that the client could remotely control the ship. The strings of text containing the movement commands could be interpreted as input the same way that playback feature translates strings to input.

Future development might also include an improvement in the recording and playback functionality. Right now, the input is saved every frame. This could be improved by adding the sample rate at which input is saved. This would drastically decrease the size of the input file saved and make it possible for the user to choose the desired sample rate.

Overall, a lot of different additions and improvements could be made to expand the simulation. This would also increase the scope of the project and the added functionality and features would make this a fully-fledged out software for ship control simulation.

References

[1] “Godot Engine,” Godot Foundation, 2023. [Online]. Available: https://godotengine.org/. [Accessed 26 September 2023].

[2] “Unity Engine,” Unity Technologies, 2023. [Online]. Available: https://unity.com/. [Accessed 28 September 2023].

[3] “PyGame development library,” 2023. [Online]. Available: https://www.pygame.org/wiki/about. [Accessed 28 September 2023].

[4] “Construct 3 Game Making Software,” Scirra Ltd, 2023. [Online]. Available: https://www.construct.net/en. [Accessed 28 September 2023].

[5] A. Calabró, “Godot logo,” Wikipedia, 8 November 2015. [Online]. Available: https://ru.m.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Godot_logo.svg. [Accessed 2 October 2023].

[6] “Godot vs Unity: Which Game Engine is The Best One?,” 16 September 2023. [Online]. Available: https://www.hitberrygames.com/post/godot-vs-unity-2023-which-game-engine-is-the-best-one#:~:text=Unity%20is%20a%20highly%20acclaimed,visually%20impressive%20and%20immersive%20games.. [Accessed 28 September 2023].

[7] “Godot (game engine),” Wikipedia, 21 December 2023. [Online]. Available: https://en.wikipedia.org/wiki/Godot_(game_engine). [Accessed 28 September 2023].

[8] Kenney, “Pirate Pack asset,” 12 February 2017. [Online]. Available: https://www.kenney.nl/assets/pirate-pack. [Accessed 8 November 2023].

[9] A. Shevchuk, “Syntax Highlighter,” [Online]. Available: https://highlight.hohli.com/index.php. [Accessed 24 November 2023].

[10] DevWorm, “How to Create SMOOTH Player Movement in Godot 4.0,” 17 March 2023. [Online]. Available: https://www.youtube.com/watch?v=KceMokK2qFA&ab_channel=DevWorm. [Accessed 3 October 2023].

[11] J. Ltd, “drawio - Free online diagram software,” 2023. [Online]. Available: https://app.diagrams.net/. [Accessed 15 November 2023].

[12] J. Linietsky and A. Manzur, “Godot Engine 4.2 documentation in English: Nodes and scene instances,” Godot Community, [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/nodes_and_scene_instances.html. [Accessed 5 November 2023].

[13] J. Linietsky and A. Manzur, “Godot Engine 4.2 documentation in English: ConfigFile,” Godot Community, [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_configfile.html#class-configfile. [Accessed 10 October 2023].

[14] The Shaggy Dev, “Creating volume sliders in Godot 4,” 23 May 2023. [Online]. Available:

https://www.youtube.com/watch?v=aFkRmtGiZCw&ab_channel=TheShaggyDev. [Accessed 20 October 2023].

[15] J. Linietsky and A. Manzur, “Godot Engine 4.2 documentation in English: Saving games,” Godot Community, [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html. [Accessed 25 October 2023].

[16] Tech With Tim, “Python Socket Programming Tutorial,” 5 April 2020. [Online]. Available: https://www.youtube.com/watch?v=3QiPPX-KeSc&ab_channel=TechWithTim. [Accessed 2 November 2023].

[17] Kermer, “TCP connection tutorial,” 4 October 2021. [Online]. Available: https://github.com/Kermer/Godot/blob/master/Tutorials/tut_tcp_connection.md. [Accessed 10 November 2023].

[18] Chris' Tutorials, “How to Setup Easy Camera Limits and Invisible Walls for Game Map in Godot 3.3.4,” 11 November 2021. [Online]. Available: https://www.youtube.com/watch?v=oBggwTeDMZY&ab_channel=Chris%27Tutorials. [Accessed 15 November 2023].

[19] LucyLavend, “How to zoom using the scroll wheel in Godot 3.2,” 17 January 2021. [Online]. Available: https://www.youtube.com/watch?v=dFd9fINGJRo&ab_channel=LucyLavend. [Accessed 15 November 2023].

[20] Google Maps, “Location of the 'Tilgu' port,” 2023. [Online]. Available: https://www.google.com/maps/@59.455247,24.4887455,243m/data=!3m1!1e3!5m1!1e4?authuser=0&entry=ttu. [Accessed 16 November 2023].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Anton Jaštšuk

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “2D ship control simulation”, supervised by Uljana Reinsalu,

   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;

   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.

2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.

3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

27.12.2023

[1] The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the University's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – GitHub link for the repository of the simulation

https://github.com/ZaRdEr15/2d-ship-control-simulation