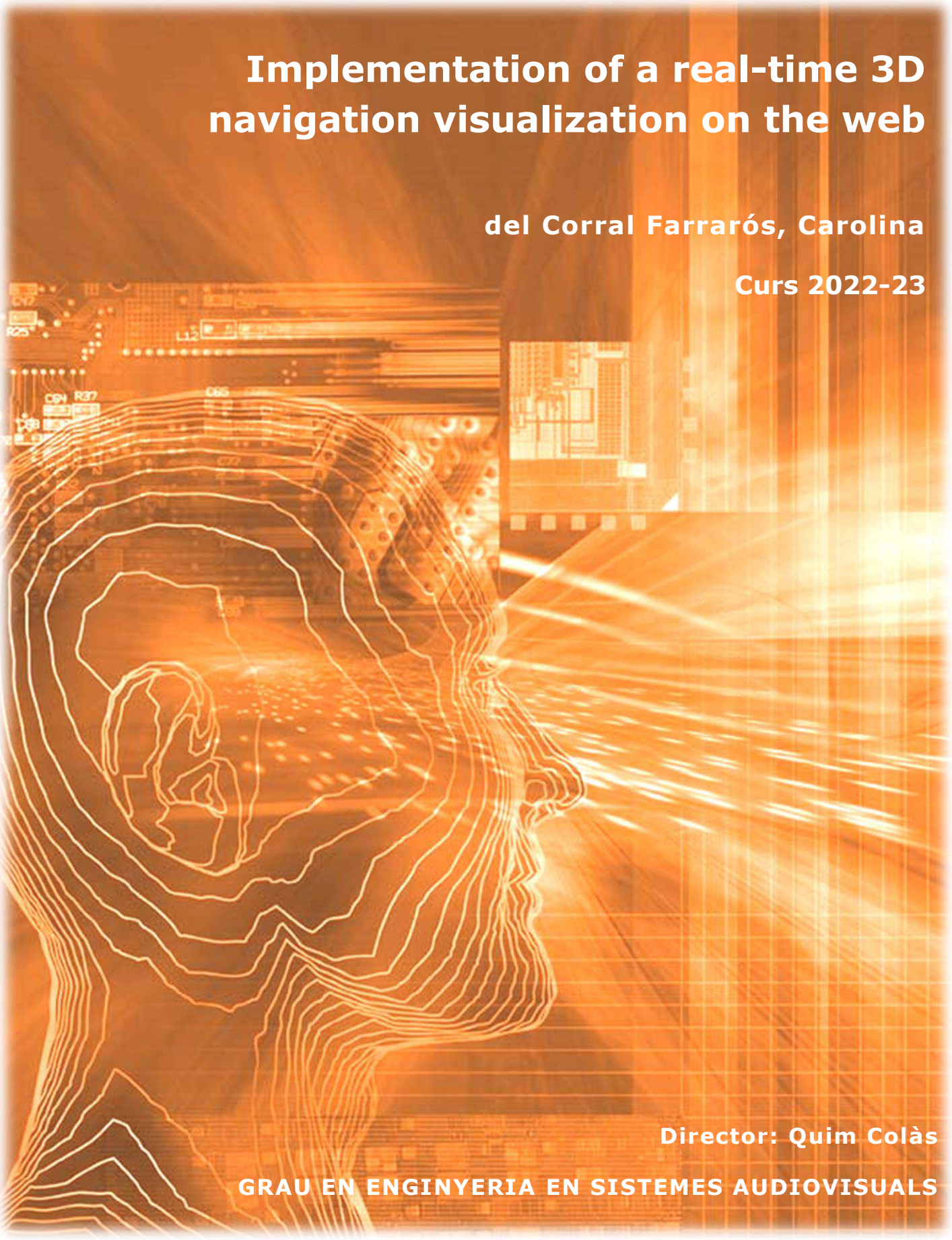
Implementation of a real-time 3D navigation visualization on the web

del Corral Farrarós, Carolina

Curs 2022-23



Director: Quim Colàs

GRAU EN ENGINYERIA EN SISTEMES AUDIOVISUALS

upf. Universitat Pompeu Fabra Barcelona
Escola d'Enginyeria

Treball de Fi de Grau

Implementation of a real-time 3D navigation visualization on the web

Carolina del Corral Farrarós

FINAL DEGREE THESIS

Bachelor's degree in Audiovisual Systems Engineering

UPF ENGINEERING SCHOOL

2023

Director: Quim Colàs
Interactive Technologies Group (GTI)

upf. Universitat Pompeu Fabra Barcelona

Que es mi barco mi tesoro,

que es mi dios la libertad,

mi ley, la fuerza y el viento,

mi única patria, la mar.

A totes les dones de la meva vida.

Acknowledgments

I would like to thank Josep Blat for trusting me and giving me the chance to work in this project and in the GTI. I also want thank my colleagues for making me feel accompanied, and especially to Jaume, for listening when I needed to vent and helping when I felt stuck.

I also want to express my gratitude to Quim for the guidance during these months. And to Javi for making me passionate about computer graphics.

Finally, I want to thank my friends for having my back all these years, to Martina for putting up with me, laughing, crying and working by my side, to Carla for continuously asking how the *"barquito"* was sailing and to Ainhoa for taking the time to help and advise me. And to my family for constantly believing in me, giving me the opportunities as well as the freedom to always choose what made me happy.

Abstract

Traditional nautical autopilot systems require ships to have complex and expensive electronic components, making its use and access difficult to the private user with small motor crafts. Under the need to cover this niche market, Searebbel was born, a start-up focusing on developing a digital rudder that connects to a mobile application. The app aims to make autonomous navigation easier and more accessible by means of route planning based on user-specified waypoints on a 2D map. In parallel it provides a 3D visualization simulating the user's boat sailing in real-time based on weather conditions, sun position, and location, and a social component where users can add information buoys and share their journey with friends.

This thesis covers the implementation of the pre-MVP with an interactive 2D map and a real-time dynamic 3D visualization of the maritime scene. The application offers the rendering of a large ocean surface in real-time with physics, as well as the support of multiple weather conditions such as rain, clouds, fog and different sea states, making it a challenging task to achieve in the context of mobile devices. This project uses the React library as the container of the frontend application, implements the MERN stack for the backend connection, Mapbox for the 2D scenery, and React Three Fiber, an extension of the Three.js library, for the graphics.

The result of this work is a web demo that fulfills all the requirements with outstanding performance while still delivering great visual results, carrying out a proof of concept that can easily be extended in the future.

Resum

Actualment, els vaixells necessiten components electrònics complexos i cars per tal de tenir sistemes de pilot automàtic, dificultant l'ús i accés a l'usuari particular amb petites embarcacions. Sota la necessitat de cobrir aquest nínxol de mercat, va néixer Searebbel, una empresa emergent orientada a desenvolupar un timó digital que es connecta amb una aplicació mòbil. L'app busca facilitar i fer més accessible la navegació autònoma a través de la planificació de rutes amb *waypoints* especificats per l'usuari sobre un mapa 2D. Paral·lelament, ofereix una visualització 3D simulant el vaixell de l'usuari navegant en temps real, basat en les condicions climàtiques, posició solar i localització. Així com un component social on els usuaris poden afegir pins d'informació i compartir la seva travessia amb amics.

Aquesta tesi cobreix la implementació del pre-MVP amb un mapa 2D interactiu i una visualització 3D dinàmica en temps real de l'escena marítima. L'aplicació ofereix la representació d'una gran superfície oceànica en temps real amb físiques, així com el suport de múltiples condicions meteorològiques com la pluja, núvols, boira i diferents estats marins, convertint-se en una tasca difícil d'aconseguir en el context dels dispositius mòbils. Aquest projecte utilitza la biblioteca React com a contenidor del frontend de l'aplicació, implementa l'stack MERN per a la connexió del backend, Mapbox per a l'escenari 2D, i React

Three Fiber, una extensió de la biblioteca Three.js, per als gràfics.

El resultat d'aquesta feina és una demo web que compleix tots els requisits amb un rendiment excepcional, oferint uns grans resultats visuals, obtenint així una prova de concepte que es pugui estendre fàcilment en el futur.

Resumen

Actualmente, los barcos necesitan componentes electrónicos complejos y caros para tener sistemas de piloto automático, dificultando el uso y acceso al usuario particular con pequeñas embarcaciones. Bajo la necesidad de cubrir este nicho de mercado, nació Searebbel, una empresa emergente orientada a desarrollar un timón digital que se conecta con una aplicación móvil. La app busca facilitar y hacer más accesible la navegación autónoma a través de la planificación de rutas con *waypoints* especificados por el usuario sobre un mapa 2D. Paralelamente, ofrece una visualización 3D simulando el barco del usuario navegando en tiempo real, basado en las condiciones climáticas, posición solar y localización. Así como un componente social donde los usuarios pueden añadir pines de información y compartir su travesía con amigos.

Esta tesis cubre la implementación del pre-MVP con un mapa 2D interactivo y una visualización 3D dinámica en tiempo real de la escena marítima. La aplicación ofrece la representación de una gran superficie oceánica en tiempo real con físicas, así como el apoyo de múltiples condiciones meteorológicas como la lluvia, nubes, niebla y diferentes estados marinos, convirtiéndose en una tarea difícil de conseguir en el contexto de los dispositivos móviles. Este proyecto utiliza la biblioteca React como contenedor del frontend de la aplicación, implementa el stack MERN para la conexión del backend, Mapbox para el escenario 2D, y React Three Fiber, una extensión de la biblioteca Three.js, para los gráficos.

El resultado de este trabajo es una demo web que cumple todos los requisitos con un rendimiento excepcional, ofreciendo unos grandes resultados visuales, obteniendo así una prueba de concepto que se pueda extender fácilmente en el futuro.

Contents

Abstract **V**

List of Figures **IX**

List of Tables **XI**

1 Introduction **1**
1.1 Context - Searebbel . . . 2
1.2 Requirements . . . 2
1.3 Objectives . . . 4

2 State of the art **5**

3 Technical Development **7**
3.1 Libraries . . . 7
3.1.1 React Native . . . 7
3.1.2 Mapbox . . . 7
3.1.3 React Three Fiber . . . 8
3.1.4 MERN Stack . . . 8
3.2 App Layout . . . 9
3.3 Map . . . 10
3.3.1 Buoys . . . 10
3.4 Scene . . . 11
3.4.1 Ocean . . . 12
3.4.2 Weather . . . 18
3.4.3 Boat . . . 24
3.4.4 Coastline . . . 25
3.4.5 Buoys . . . 25
3.5 Logbook . . . 26

4 Results **27**

5 Conclusions and Future Work **29**

A GALATEA Scope Documentation **A**

B GALATEA European Documentation **F**

C A Guide on Three.js Shader Chunks **Q**

List of Figures

1.1 Wireframes of the application. 2

2.1 Rendered image of an oceanscape *from Tessendorf's paper*. 5
2.2 Screenshot of the BRW Demo *by J. Agenjo*. 5
2.3 Screenshot of the OBSEA Demo *by G. Llorach*. 6
2.4 Screenshot of final result *by G. He and H. Wu*. 6
2.5 Screenshot of "A trip under the moonlight" *by Jérémy Bouny*. 6

3.1 Illustrative Diagram of the MERN Stack 9
3.2 App Diagram 10
3.3 Embedded map in the webpage 10
3.4 Embedded map in the webpage 11
3.5 Illustrative visualization of the 3D scene 12
3.6 Scene Diagram 12
3.7 Wave Drawing from the Uncharted Game Conference 14
3.8 Sine vs. Gerstner Wave 14
3.9 Fragment Shader Progress 17
3.10 Chart of Sky Values along the day 20
3.11 Different Sky States 20
3.12 Perlin Noise Texture 21
3.13 Difference on the edges of the cloud plane in attenuating by distance. . . . 22
3.14 Scene seen from below the surface 23

4.1 Screenshot of the frontend deployed on an AWS of Searebbel 27
4.2 Difference on the water color 28

5.1 Screenshot of Google Earth's view on the Barcelona coastline 30

A.1 Wireframes of the application. . . . . A

List of Tables

Characteristics of each context of the application. 4

3.1 Features of libraries related with each requirement 9
3.2 Skybox Shader uniforms and values along the day 19
3.3 International Visibility Code with Meteorological Range 24

Chapter 1

Introduction

The aim of this TFG is to create a pre-MVP version of an app to support easy navigation, by means of route planning based on user-specified waypoints on a 2D map, in parallel to providing a 3D visualization simulating the user's boat sailing in real-time based on weather conditions, sun position, and location, and a social component where users can add information buoys and share their journey with friends. Indeed, I implemented a web-based demo of the route planning and social component on a 2D map and their 3D simulation. It fits the pre-MVP requirements, while providing a solid basis for extending it to the full app. Finally, the work covers both the front-end and back-end aspects, therefore is considered to be a full-stack project. This work was done under the context of Searebbel, a start-up company that aims to simplify navigation by creating an autopilot system and the developed version of the app described above.

- **Chapter 1.2** discusses the functional requirements of the pre-MVP as initially established, compared with those of the full application; it also presents the evolution of those requirements, and the functionalities finally achieved in this TFG.

- **Chapter 3.1** reviews the main choices to implement the app: *React* as an overall system, *Mapbox* for 2D, *React Three Fiber*, a React renderer for Three.js on WebGL for 3D, and the specific choices of libraries within them.

- **Chapter 3.3** presents an overall view of the 2D system, and its four main components providing details of the map itself, the "buoys" or pins of information, the current route waypoints and the location of the user's boat. To add the map canvas, the application connects to *Mapbox*, as specified before. However, it also needs to get all the aforementioned data from a database, for such a task *MERN* stack was implemented, to bring together MongoDB and React using *Express - Node.js*.

- **Chapter 3.4** describes the different elements of the 3D simulation and its connection with weather services as the sea state and the climate of the scene are reactive to the forecast of the user's location, as the demo is connected to *OpenWeather API* to retrieve the corresponding data. The scene also contains other interesting elements, like the coastline, the buoys seen in the map and a changing skybox according to the hour of the day the render is being displayed.

- **Chapter 3.5** covers the implementation of the logbook element.

- **Chapter 4** exposes the results obtained and relates them with the original requirements and evaluates the client's satisfaction.

- **Chapter 5** concludes the project and explores the different lines of future work.

1.1 Context - Searebbel

Searebbel offers to make an autopilot navigation system by creating an intelligent rudder that is installed on top of the boat's manual rudder. The company also designed an application, an extension of the navigation system, that creates all the routes taking into account the waypoints the user specifies in a 2D map. However, they also wanted to add a social factor to it by giving the user the possibility to add "buoys" of information along the route so that other users can see relevant data regarding the sea state, and afterwards, share with friends their journey in a video format. To give the app an added value, they wanted to include a 3D visualization of the user's boat sailing the sea in real time, coherent with the current real world weather conditions, sun position and boat location, among others. My goal is to implement, in a web application, a demo of the 2D map, the 3D scene and the logbook recording.

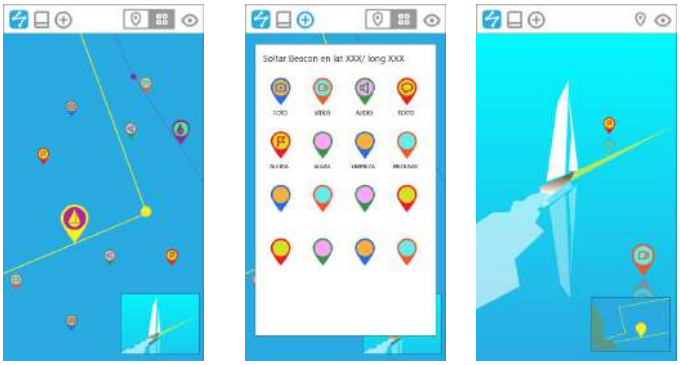


Figure 1.1: Wireframes of the application.

1.2 Requirements

The documentation of the project made a clear distinction between the two phases of the development process: the pre-MVP and the commercial app. This work addresses the first, a demo capable of showing the full potential of the application, keeping in mind that is going to be extended in the future with slightly different requisites. The requirements were the following:

Functional

- **Solar Position (geolocation + hour):** The visualization will need to read the geolocation of the boat, know its trajectory and, according to the time of the day, correctly represent the sun position. This needs to be functional at all times of the day, including the night, and be able to truthfully represent the sky state at that given moment. This requirement is final and the commercial product will be the same.

- **Coastline (geolocation):** It is not mandatory for the demo to include resources for representing the coastline in the scene. Nevertheless, the partial implementation of it would be appreciated. For instance, being able to fake the coastline of a given location to represent the capabilities the MVP will have.

- **Logbook:** Regarding the social component, the user has the possibility to create a logbook of the route carried out. Along the journey, the boat stops, speed, photos and other inputs will be stored, so at the end they can share their experience with a mixed 2D/3D video that sums it up.

- **Pins/Buoys of information:** As mentioned, Searebbel offers the option to add pins of information along the route; they can contain details on the water clarity, presence of jellyfish, filth... among others. To maintain the maritime style, these pins will have the appearance of buoys.

- **Boat's aspect:** In the demo, there is no need to have a customization option, being able to choose between a few basic models is sufficient. The user will have control over the aspect of the ship in further versions of the application.

- **Weather (geolocation + meteorological service):** The demo does not need to render different weather conditions. In such a case, the clime could be faked or chosen between some presets. In the future, the app will connect to a meteorological service in real time in order to display the weather in the user's location.

- **Other ships' location:** As the application wants to have a social factor to it, they want the MVP to be able to render other boats near the user's own. However, this is not needed in the demo.

Non-Functional

- **APP Integration:** For the pre-MVP, selecting the 3D view in the app will open a new window in the browser. Nonetheless, in the commercial version, the 3D view is meant to be fully integrated in its own context.

  Therefore, this introduced a challenge concerning the programming language and libraries that were required for the correct implementation and future integration of the scene in the commercial app.

  Initially, my intention was to use Javier Agenjo's libraries, Rendeer.js and Litegl.js, with the goal of reusing and enhancing the work he had done years ago for the Barcelona World Race. These libraries are designed to work in a web environment using the Javascript language, thus they were perfect for the development of the web demo.

  In spite of this, I had to keep in mind that the web demo will be adapted to the non-web version in order to view the 3D scene inside the app being developed in React Native. Doing some research on how to integrate J. Agenjo's libraries, I found that they were not compatible and there was no documentation regarding the possibility of adapting them. Consequently, I decided to switch to Three.js since

the community is much larger and there seemed to be resources concerning the integration in a React environment.

At the beginning, I intended to use Vanilla Three.js. However, it is not functional in React Native CLI, only using Expo. Looking into it, I found a React renderer for Three.js: React Three Fiber (R3F), a library that allows the creation of 3D scenes in React Native, functional in web and mobile applications. Consequently, this was the technology chosen for this project since it has future projection and it is as versatile and capable as plain Three.js. Later on, I will get into detail about the advantages and disadvantages and main differences of using the R3F library.

Find attached the original requirements table in appendix A, as well as the wireframes and mockups of the application designs, created by Quim Colàs.

1.3 Objectives

Taking into account the different requirements, this project can be divided into three main contexts regarding the principal concerns surrounding the use of the application before, during and after the experience of navigation. Each of the contexts directly corresponds to a main element of the definitive app, reflected in the following table:

Context	**Description**	**Main App Element**
Planning	A 2D visualisation in which the user can navigate the map and plan the route to follow by marking points on it.	A 2D map where the waypoints can be seen, the user can see its boat located in the map, drag it around and leave buoys of information along the way.
Navigation	Combine the 2D where users can track the ship's geoposition on the map and the 3D to view it in with weather conditions, sun position, and a coastline if visible. For an optimized demo, speed, coastline and weather during the journey will be simulated.	A 3D view corresponding to the navigation process, where the user can switch between the 2D map and the 3D scene and visualize its ship sailing the sea according to the current weather conditions, the sun location and the coastline if needed.
Experience-sharing	Generation of a video summarising the experience by combining the 2D and 3D visualisations and information (photos, comments, etc.) entered by users during the journey (which can be simulated).	A button that will access the stored route and generate the video.

Chapter 2

State of the art

A widely used approach for ocean rendering in professional productions, like the Titanic movie, is Tessendorf's FFT-based method [1]. Usually developed in engines such as Unity, Unreal, DirectX or OpenGL due to the high demand on resources, use of processes like Tessellation and compute shaders.

Figure 2.1: Rendered image of an oceanscape *from Tessendorf's paper*.

Nevertheless, there exist few implementations in WebGL, such as the technology developed by Javier Agenjo for the Barcelona World Race 2015[1]. Although his work was intended for a game, it included the same elements Searebbel requires concerning the 3D scene. Agenjo used Rendeer.js and Litegl.js for his implementation. For the wave generation algorithm he utilized Gerstner Waves, introduced by Fournier & Reeves for lightweight ocean rendering– a commonly adopted approach within web-based graphics [2]. For dealing with Level Of Detail (LOD) he used the projected grid concept, introduced by Claes Johanson [3].

Figure 2.2: Screenshot of the BRW Demo *by J. Agenjo*.

Gerard Llorach also addressed sea simulation in web environments for the OBSEA project[2], he carried it out in Three.js and also used Gerstner Waves. However, for the ocean surface mesh he modeled his own grid with different resolutions depending on the distance to the center. Both J. Agenjo and G. Llorach implemented buoyancy to the objects on the sea by computing Gerstner waves on the CPU.

[1]BWR Web Demo: https://tamats.com/work/bwr/
[2]OBSEA Web Demo: https://bluenetcat.github.io/OBSEA/

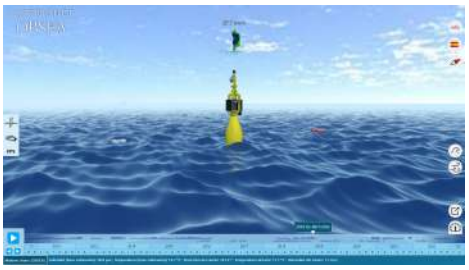


Figure 2.3: Screenshot of the OBSEA Demo *by G. Llorach.*

In 2013, Guanyu He and Hao Wu presented ocean simulation using Tessendorf's approach, creating a custom FFT shader in WebGL [4]. As mentioned earlier, the drawback of this method is its high cost for large surfaces, which cannot be supported by WebGL in real-time. To settle this problem, they join patches of grids of different resolutions. This allows to render visually infinite surfaces, however the regularity of the ocean and the gaps between patches is notable.

Figure 2.4: Screenshot of final result *by G. He and H. Wu.*

In 2015, Jérémy Bouny also provided an efficient implementation of the Fast Fourier Transform algorithm in WebGL[3], through the generation of a displacement map and a normal map applied to the vertex and fragment shader. For the ocean mesh, he used the screen space grid method by C. Johanson. Bouncy's demo achieved high frame rates while delivering satisfying visual results, however it does not resolve the issue of ocean regularity encountered by other researchers.

Figure 2.5: Screenshot of "A trip under the moonlight" *by Jérémy Bouny.*

[3]Github Repository of jbouny: https://github.com/jbouny/fft-ocean

Chapter 3

Technical Development

In this section the technical aspects, the decisions taken and the problems faced will be discussed in detail in each of the three different contexts referring to their main element (i.e. 2D map for the planning, 3D canvas for the navigation and logbook for the experience-sharing), taking into consideration the need for future scalability and implementation on a mobile application.

3.1 Libraries

3.1.1 React Native

The company is using React Native, a framework used to develop cross-platform applications in Javascript, to build their app. This decision will determine the rest of the technologies utilized; to ensure the compatibility of all the libraries used in the project the web demo will also be developed using React.

Other multi-platform architectures such as Flutter or Xamarin could have been used, however this decision was taken by the company.

3.1.2 Mapbox

As for the 2D map, we chose to use Mapbox, an online map provider, which offers a studio to customize your own map style according to your needs. It also provides a library, Mapbox GL JS, that allows you to to display your maps, add interactivity and customize the experience in your web-based application, which is exactly what is needed for the 2D context of the project.

Mapbox was also chosen because it works for web and mobile applications, bringing the scalability needed for the project. They provide separate SDKs to integrate it in Android and iOS, however they do not have tools that support hybrid frameworks, like React Native. This did not pose a problem, given that there are external plugins and integrations that allow the use of Mapbox SDKs in a React Native environment [5].

There exist other map providers such as Google Maps, Leaflet.js, react-geo or Pigeon Maps, these options were discarded as they did not provide tools for all the needs of the project, there was not enough documentation or due to lack of free pricing plans.

3.1.3 React Three Fiber

The requirements were very restricting when it came to deciding which library to use for the 3D graphics. As the demo was going to be hosted on the web but it was intended to be integrated in a mobile app environment in the future, so my work had to be the most reusable possible. For web graphics, the most extended library is Three.js, but I had to keep in mind that the main app is being developed using React Native. Doing research I found that the Three.js community has developed a renderer for React called React Three Fiber (R3F), which allows it to generate 3D graphics in any environment, web or mobile.

The Babylon.js framework also supports React Native implementation, however the community is smaller, so I decided to stick with R3F as it has more users and active developers. R3F claims to be as powerful as Three.js where you can *"Build your scene declaratively with re-usable, self-contained components that react to state, are readily interactive and can participate in React's ecosystem"*[1]. This was the most reasonable and guaranteed to work option, so I chose to go ahead with it as anything that can be done in Three.js can be mapped onto R3F, and the performance does not seem to be affected.

The evident difference between vanilla Three.js and R3F is the sintaxis, as it is expressed in JSX, a javascript extension that structures components resembling HTML. However, there are more differences than those at first sight, when creating a scene in R3F it is distinct from Three.js as everything gets encapsulated inside the `<Canvas>` component, including the renderer, scene and camera. These elements can still be accessed, but it is a complex task. Therefore, working with this tool has implied an added complexity that working directly with Three.js would not have. Having everything hidden in components is powerful and easy to use for easy tasks, but when the complexity rises it gets harder to implement and to access the lower layers. Despite these inconveniences, React Three Fiber has allowed me to achieve all the goals and milestones.

3.1.4 MERN Stack

The application needs to fetch data from the database to display the pins of information correctly in the canvas and allow users to login and register. As the app is being developed in React and the company chose MongoDB as the database program, I implemented the MERN Stack[2] to bring together the frontend and backend and database, using Javascript only. The four technologies that compose it are: MongoDB for the document database, Express.js for the Node.js web framework, React for the client side and Node.js for the web server.

In table 3.1 the features of each library and how they directly relate to the requirements can be seen for a better understanding on the choices made.

[1] React Three Fiber Documentation: https://docs.pmnd.rs/react-three-fiber
[2] MongoDB Documentation: https://www.mongodb.com/mern-stack

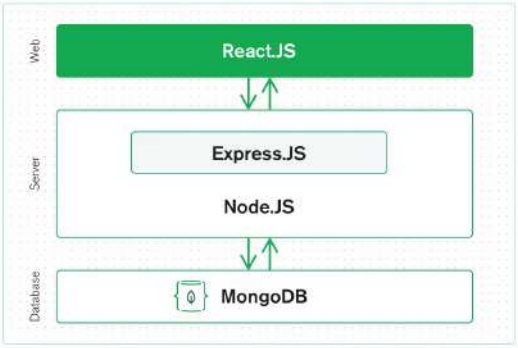


Figure 3.1: Illustrative Diagram of the MERN Stack

Requirement	Feature	Library
App Integration	Multiplatform Support	React Native
Solar Position	Get Sun And Moon Position And Sunlight Times	SunCalc.js
Boat's aspect	Loading (GLTF) models	R3F
Coastline		
Buoys Of Information		
	Markers	Mapbox
	Post and Get requests on MongoDB	MERN Stack
Logbook	Capture Stream	HTMLMediaElement
Weather	Current weather data at a given location	OpenWeather API

Table 3.1: Features of libraries related with each requirement

3.2 App Layout

Taking into account that the app is being developed in React Native, the container of the web application is also built upon such. React is a front-end Javascript library which is based on components, these are reusable and independent and work like a Javascript function. Elements can contain children and be nested with other components, thus creating a UI tree; however, information cannot be shared between children in a direct way, they need to send it to the parent node, which will be in charge of sending it to the other child component.

Being able to have independent nodes brings benefits, as well as disadvantages. It is great for assembling Single Page Applications, as each has its own functionality and logic. On the other hand, it gets complex when you want them to interact with each other, as the state of the application needs to be updated for all the elements to be aware of the changes of one another. Nevertheless, this allows the root or parent nodes to have control over the whole application or scene.

In order to better understand all the components involved in the app and the pipeline, supplementary Figure 3.2 is provided.

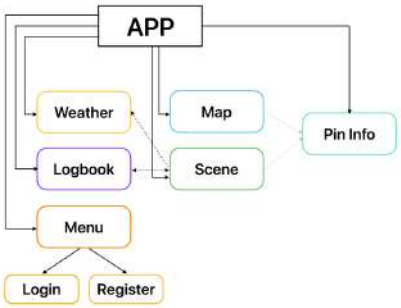


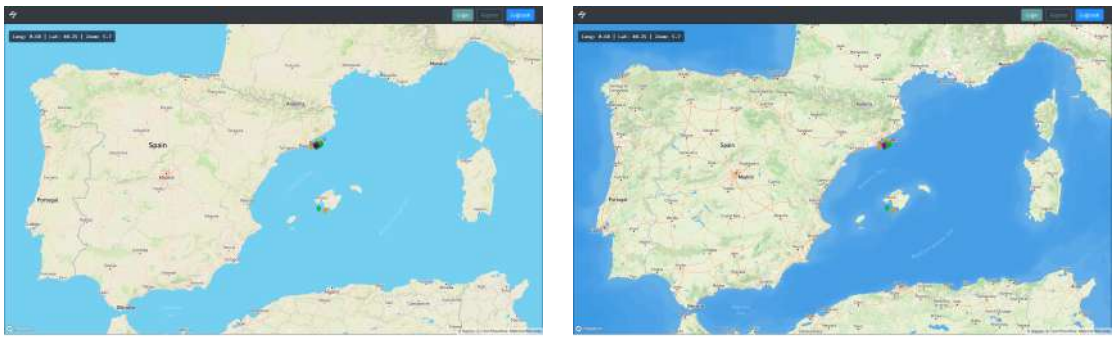
Figure 3.2: App Diagram

3.3 Map

In order to include the map to the web application I followed the Mapbox documentation, installing the necessary libraries and obtaining the access tokens required [6].

Regarding the styling of the map, in the Studio I modified some basic colors and added shadows to give the water a more natural and appealing look. I also changed the Terrain appearance, emphasizing the greener areas and enabling the shaded relief of hills. Although the application does not currently utilize this functionality, the Studio also allows the creation of layers, for example, to add place labels, which may be implemented in the future.

In the following Figures 3.3(a) and 3.3(b) the map embedded in the web application and the difference between the predefined and custom styling is shown.



(a) Predefined Map Style

(b) Custom Map Style

Figure 3.3: Embedded map in the webpage

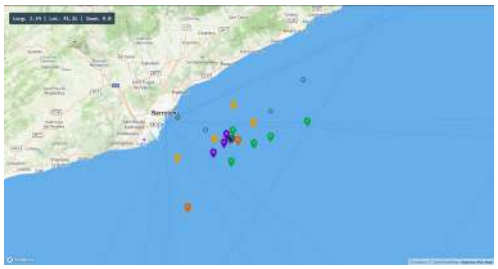
3.3.1 Buoys

Once I had included the map onto the webpage, I had to connect it to the database and implement the CRUD functionalities in order to view and add the pins of information. To do so, I followed the MERN Stack, which allows you to operate the frontend, backend and database using Javascript [7].

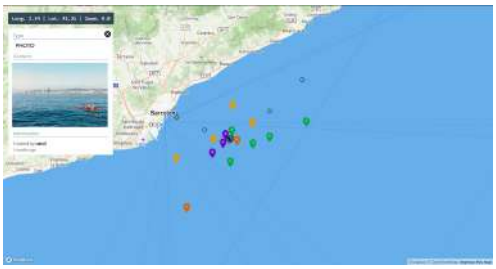
To build the server I use node.js and express to create all the routes to post and get data as the API endpoints. Once I created the routes and defined their behavior I connected my server to the database using `mongoose`, a library that allows establishing connections between MongoDB and Node.js. I also created the Schema of my data, having one for the buoys and another for the users. Once the server was up and running, I needed to host iron the web as it runs on localhost. At first I used Heroku[3], a free platform that allows developers to build and operate applications on the cloud, however it stopped being free mid-project and I had to quickly migrate to another platform, Render[4].

Now the API is running in the cloud so it can always be accessed by the users. Nevertheless, the frontend of the application was not connected to the server yet; so to send HTTP requests I used the `axios` library.

On the map component, I call my server to retrieve the existing buoys from the database and display them as markers on top of the 2D map and to post new buoys by double clicking on the map. I also make use of the API in order to login and register on the application, as users can only add markers if they have an account.

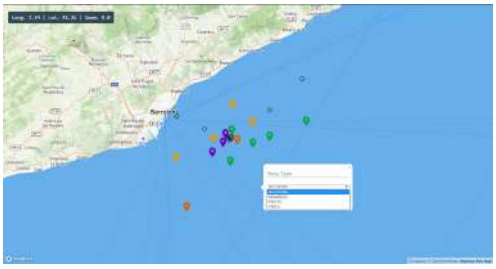


(a) Final map with buoys displayed



(b) Display after clicking on a pin

(c) Add pin popup after click twice on the map



(d) Fill form of adding new pin

Figure 3.4: Embedded map in the webpage

3.4 Scene

The main contribution to the project is the 3D scene, implemented in React Three Fiber. The objective of this element is for the user to view their ship sailing the ocean in real-time according to the sea state, weather conditions (clouds, rain, fog and sun position)

[3]Heroku's webpage: https://www.heroku.com/
[4]Render's webpage: https://render.com/

and coastline, if needed, in the corresponding location and time.

Therefore, this section covers the rendering of the sea in WebGL in real-time, with all the corresponding environment elements. The display also includes the user's boat moving with the waves and the buoys of information introduced by users in the map component, as the following pictures show:

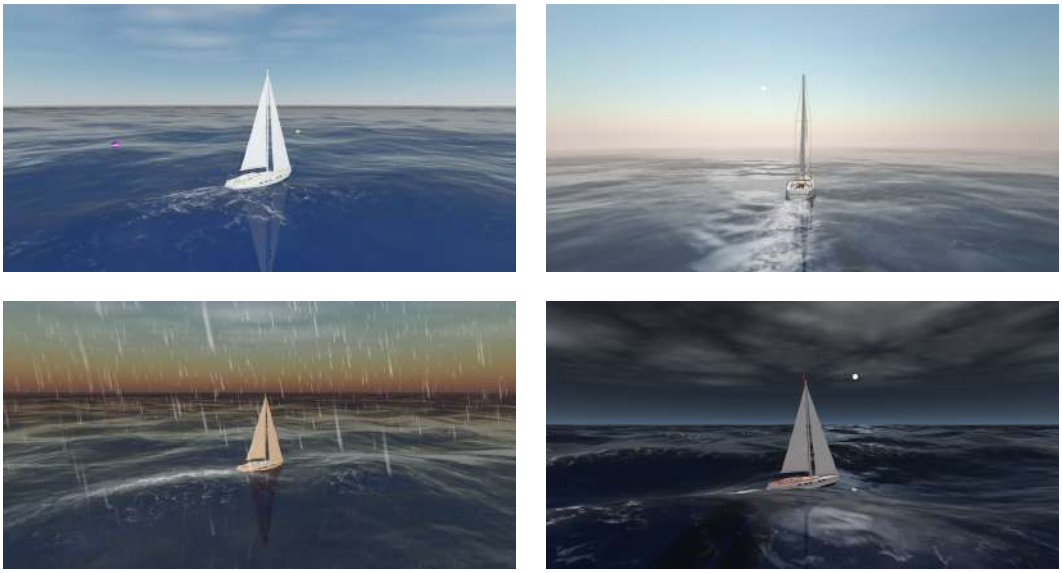


Figure 3.5: Illustrative visualization of the 3D scene

As the whole application is built upon React and follows the component structure, the scene has also been developed in a modular manner. In Figure 3.6 you can view all the elements created and how they relate to each other. They are all children of the Scene component, however, for better visual understanding they are divided according to if they are location or weather dependent.

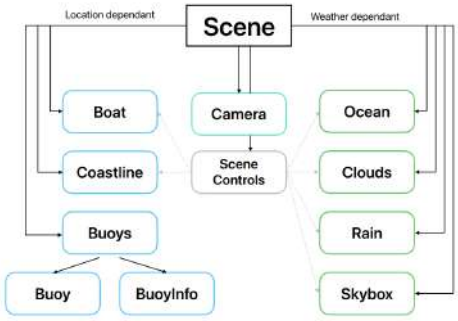


Figure 3.6: Scene Diagram

3.4.1 Ocean

As for the main element of the scene, I had to take into account different elements; the water needed to look realistic in terms of color and movement, meaning that reflections,

shadows and animation needed to be implemented. In order to carry it out, I needed to choose an adequate geometry and create its material, customizing the vertex and fragment shader.

To create custom materials in Three.js, shaderMaterial or rawShaderMaterial shall be implemented. The first one includes built in attributes and uniforms, like cameraPosition. In the second, variables must be sent to the shader manually [8].

For vertex and fragment shaders, you can write GLSL code from scratch, or you can use existing code, called Shader Chunks, created by the Three.js community. When including the Shader Chunks in my shaders I found a lack of documentation[5]. Hence, by reading the source code, I created a guide on how to use them. You can find the guide in the Appendix C.

I found an example in the official R3F webpage that implemented a basic water fragment shader with planar reflections, really useful for the project [9]. So I decided to build from there and improve the implementation.

Geometry

Said example uses a 10000x10000 PlaneGeometry, yet is not the best approach to have a uniform mesh in this case, since it will take up a lot of resources to distort the vertices further away where not much resolution is needed. However, if we decrease the overall number of vertices, the center of the scene, same as the mesh, where we need a bigger resolution will not look smooth. Increasing the resolution of the plane would be throwing resources away since the details will not be appreciated.

Looking at Gerard Llorach's work, he modeled a circular mesh that has a bigger vertex density in the center and decreases at the ends. With his permission, I loaded his geometry onto the project instead, since there are low, medium and high resolution versions. Changing the geometry also brought other benefits besides improving performance, we will discuss them in the following sections of this chapter.

Gerstner Waves - Vertex Shader

About the custom shader for the water material, as stated before, I reused the R3F example, which already had the uniforms, vertex and fragment shader created. However, I needed to animate the mesh to resemble wave motion, to do so I modified the vertex position in the shader.

The most novel method to implement realistic ocean wave simulation in real time is the FFT method by J. Tessendorf [1], it delivers admirable results. However, it is a very complex method and it runs NxN times on an NxN grid, so it scales poorly with the number of vertices, bringing down the performance of the application. Furthermore, it

[5]Three.js Documentation on ShaderChunks: https://threejs.org/docs/api/en/renderers/shaders/ShaderChunk

is mostly implemented in OpenGL since it uses methods such as tessellation to avoid artifacts and repetition when joining patches, this technology is not currently supported by WebGL. Bearing in mind that this work is being developed in a web environment and to be displayed in a mobile device, such an approach was discarded for the previous reasons.

Tessendorf's proposal relies on the trochoidal representation of waves, or Gerstner waves, using the FFT and oceanographic spectra to add more detail and get the right parameters. Therefore, I choose to implement "A simple model of ocean waves" by Fournier & Reeves [2], which is also based on Gerstner waves but with less high frequency detail. They simplify the behavior model of water waves and take up less resources, improving the efficiency of the application while still granting satisfactory results. The main difference can be seen in Figure 3.7, retrieved from the slides presented by uncharted at the Game Developers Conference in 2012 [10].

(a) Gerstner Wave (b) FFT Wave

Figure 3.7: Wave Drawing from the Uncharted Game Conference

To put it into effect, I mainly followed the Catlike Coding tutorial[6], which explains the math and logic behind the algorithm and how to execute it in Unity. They explain that the straightforward and simple way to simulate waves would be through a sinusoidal function, however, that is not realistic as it does not model the real shape of waves, looks repetitive in large areas and it is only a representation of the surface, not reflecting the movement of the water underneath on each wave. That is why Gerstner waves come into action, they model the idea that water particles move up and down but also horizontally in a wave motion, doing circular movements as it can be seen in supplementary Figure 3.8.

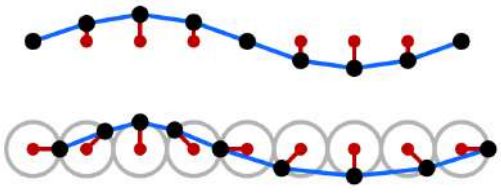


Figure 3.8: Sine vs. Gerstner Wave

The trochoidal equations are based on the sine function for displacing the Y coordinate, and they extend the X and Z coordinates to be shifted with a cosine function, such a change turns the sine wave into a circle. To avoid the plane turning into a circle, the points need to be anchored to their origin. Furthermore, each coordinate needs to move into a certain direction, the amplitude being dependent on the wave number and the

[6]Catlike Coding Waves Tutorial: https://catlikecoding.com/unity/tutorials/flow/waves/

steepness of the wave and the speed dependent on wavelength and earth's gravity. The vertex equation is as follows:

$$P = \begin{bmatrix} x + D_x \frac{s}{k} \cos f \\ \frac{s}{k} \sin f \\ z + D_z \frac{s}{k} \cos f \end{bmatrix} \tag{3.1}$$

Where the phase is:

$$f = k \left(D \cdot \begin{bmatrix} x \\ z \end{bmatrix} - ct \right) \tag{3.2}$$

Where c represents the wave speed, dependant on the wavelength: $c = \sqrt{\frac{g}{k}} = \sqrt{\frac{g\lambda}{2\pi}}$, a stands for the amplitude, also dependant on the wavelength and the steepness desired: $a = \frac{s}{k}$ and D is the direction vector.

Doing all these transformations changes the plane normals, which need to be computed and sent to the fragment shader for correct lighting. In order to get the normals, I can do the cross product of the tangent and the binormal, which are the partial derivatives of P in X and Z respectively:

$$N = T \times B \text{ where } T = \frac{\partial P}{\partial x} = \begin{bmatrix} 1 - D_x^2 s \sin f \\ D_x s \cos f \\ -D_x D_z s \sin f \end{bmatrix} \text{ and } B = \frac{\partial P}{\partial z} = \begin{bmatrix} -D_x D_z s \sin f \\ D_z s \cos f \\ 1 - D_z^2 s \sin f \end{bmatrix} \tag{3.3}$$

In my implementation, I support three main waves with distinct directions, wavelength and steepness. In order to do so, the anchor point remains and I add a summation of all the displacements with the different wave parameters, as the following equation shows for P_x:

$$P_x = x + \sum_{i=1}^{n} D_{ix} \frac{s_i}{k_i} \cos f_i \text{ and } f_i = k \left(D_i \cdot \begin{bmatrix} x \\ z \end{bmatrix} - ct \right) \tag{3.4}$$

Color - Fragment Shader

Once the mesh was animated, I adjusted the fragment to properly shade the ocean surface, following Jonas Wagner tutorial on Rendering Water with WebGL [11]. In order for

the rendering to be true to life, the water needed to have correct lighting, reflections, reflectance, scattering and foam. Water also has properties like refraction and absorption which I decided not to take into account, since in the general use case the boat is in open waters, where the absorption can be assumed to be even at all points and the waters profound. This high depth value makes refraction not noticeable, furthermore, there are only a few objects in the scene to be refracted. Implementing these two would be really complex and the results would be barely perceptible, which would lead to a misspending of resources.

The first milestone was to implement Phong shading. By storing in separate values the diffuse and the specular colors, I was able to directly alter one or another depending on each water property. The most important thing to take into account when implementing Phong was to make sure that the normals were accurate, in the previous section this was discussed, considered and corrected. It was also mentioned that Gerstner waves fail at giving high frequency details. so, I attempted to mimic the water ripples by using a normal map that gave more detail to the surface and lighting. By building in the vertex shader a TBN matrix and passing it as a varying to the fragment, I was able to mix it with the sampled texture. In spite of this, the sampled normals looked static and did not have the noisy pattern desired. To solve this problem, I sampled the texture multiple times by modifying the uv's and using the time variable in different directions, so the different texture normals intertwined creating the noise sensation.

The next step was to add the reflection, this was already done in the example I used, they apply planar reflections and send the resulting texture to the shader as a uniform. I modified the way they sample the texture, so the distortion is higher when objects are further away, otherwise the reflection looked too sharp on distant objects. Adding reflections brought many positive results, since the sky and sun color did not have to be sent as a uniform to the shader and instead were adjusted in real time according to the changing colors.

Planar reflections provided great results, however as waves got larger and taller, the reflection started to look unrealistic and distorted. Furthermore, due to performance reasons, the mirror texture is of low resolution, which may look jagged on the edges. Other approaches are commonly used in computer graphics, such as ray tracing which provides the best results, however, it is not feasible to do so for real time rendering in WebGL. Another method is to apply environment mapping, which definitely was not the suitable approach to this application, as the environment is constantly changing and procedural. Screen Space Reflections also give pleasing results, however it is quite expensive for mobile devices. Consequently, planar reflections were the best choice for a dynamic scene due to the lack of resources, despite being expensive as well, but not many elements are in the scene that need to be rendered twice.

Light not only gets absorbed by water particles, it also gets scattered into different directions underneath the surface. To simulate the scattering phenomena in the shader, I use the dot product between the surface normal and the view vector. I use the result as a weighting factor to adjust the water color. The reflection sample directly modifies the specular color, whereas the scattering is added to the diffuse component. I computed a reflectance factor, or fresnel using Schlick's approximation, in order to properly mix the

two values according to the angle of incidence, having a stronger reflection when looking from a shallow angle and a weaker one when the angle was wider.

Furthermore, I apply a shadow mask, generated automatically by R3F from the directional light in the scene simulating the sun. Finally, the sea looked realistic however it did not give the sensation of movement. To solve this problem, I sampled twice a foam texture using the world coordinates scaled as uv's and multiplied by the time and boat speed and direction, this allowed me to get the foam moving all over the sea surface in a non uniform nor static manner. To mix it with the final color I applied a linear interpolation between the sea color and the foam sample weighted by the wave height, so the foam appeared at the crests. This made the foam appear evenly along the surface and enhanced the repeatability of ocean waves, so I masked the foam texture with perlin noise, sampled taking into account the direction of the tides so the foam followed the crests and did not disappear and emerge randomly.

The sea looked complete and moving, yet when the boat was added to the scene it seemed as if it was always in the same location. To fix it, I added a wake at the boat stern. In order to do it I used two textures, a foam trail and the foam used before. The first one defined the shape and opacity and the second one was used to establish color, movement and unevenness. One challenge of the foam trail was to define the correct uv coordinates to sample the texture, since the boat can be rotated and of different dimensions, meaning that the trail should rotate with the ship and translate to match the boat size. Since the boat is always located at the [0,0,0] in the scene, I could use the world position for the initial uv's. However, to rotate them I needed to change the basis, otherwise the texture did not rotate as desired, to build the new basis axes I used the boat direction and its perpendicular. To apply the rotation I multiplied the trail uv's by the basis, afterwards I scaled the result according to the ship ratio and translated it to the center of the scene depending on the length. Using the target rotation causes some artifacts when the boat is changing direction, as the rotation speed is different from the foam trail and the boat. This causes the foam trail to mismatch the cue of the ship for a few seconds until the boat reaches its target rotation. Changing the rotation trail according to the current rotation and not the target one, would imply to update the uniform value on every frame, so I decided to not modify the current implementation for performance reasons.

Figure 3.9 displays the progress and how each of the phenomena affect the final result:

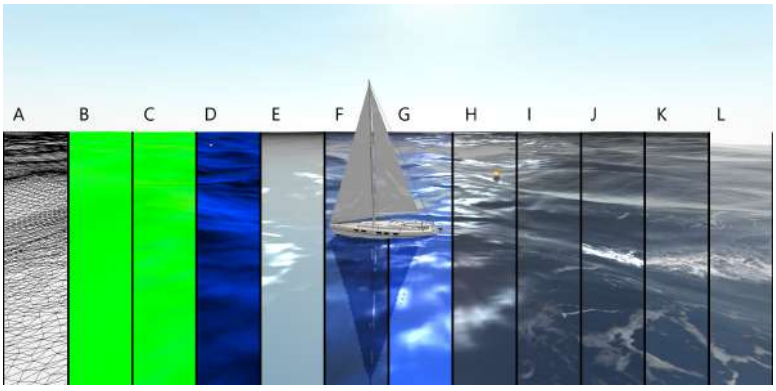


Figure 3.9: Fragment Shader Progress

A: Wireframe, B: Normals, C: Surface Normals, D: Diffuse, E: Specular, F: Phong, G: Scattering, H: Shadows, I: Foam, J: Trail, K: Tonemapping, L: Fog.

Beaufort Scale

At this point, I had a GUI that allowed me to change in real time the steepness, wavelength and direction of the three main swells. However, keeping in mind that the application had to be connected to a meteorological service and represent the current sea situation I needed a standard that could map the weather conditions to the sea state. Doing some research I found the Beaufort Scale, an empirical measure which directly associates the wind speed with the choppiness of the ocean [12].

Looking at the documentation I created some presets that mapped the different possible ranges of wind force to the proper parameters of the three waves so the scene resembled as much as possible the reality. My application maps from the beaufort number 0 (calm sea) until number 9 (strong gale), leaving out levels 10, 11 and 12 as they are storm conditions, and it is assumed that the sea is not navigable in small boats, therefore out of the use cases of the app.

Fix Repeatability

Another challenge was to deal with the visual repeatability of the waves, as it was noticeable that there was no randomness and the swells were the same all over the ocean. Repeatability was accentuated with the first geometry: the uniform plane, using G. Llorach's mesh, helped break the pattern.

However, it still was notable so I thought of different solutions that would help, such as adding multiple small waves. Doing so, helped by adding high frequency detail but the ocean waves still looked repetitive. A similar solution was to add perlin noise to the vertex's height, but the outcome was similar to the first approach, so I discarded both changes as I did not see any visual improvements, the performance was affected and a high resolution mesh was required for the small distortions to be appreciated.

One thing that helped was using the perlin noise mask when adding foam to the crests. As, when foam was added only using the height it enhanced the fact that the waves were the same.

3.4.2 Weather

The visualization also needed to include multiple weather conditions: clouds, rain, fog, wind, the sun at the different times of the day and the corresponding sea states. In this chapter I will break down the implementation of each of the elements.

Sun - Skybox

The most usual way to include a skybox to the scene is by adding a cubemap, a fixed image with six textures stored in a cross form that will be mapped onto the six faces of a cube geometry. It should be the first element to be rendered onto the scene without writing in the depth buffer, so it always appears to be in the background of the scene.

However, for this application I needed to have a day-night cycle and that is not feasible with precomputed images; for such a task, a procedural skybox would be more suitable. Several developers have implemented and improved the *"A Practical Analytic Model for Daylight"* paper [13], where they present an inexpensive analytic model that approximates full spectrum daylight for various atmospheric conditions.

My role was to adapt the three.js integrated model[7] onto R3F and define the different daylight conditions. The model allows you to edit different parameters in order to correctly represent the reality, perhaps the most valuable for this project was the sun position (azimuth and inclination). I separated the day into different slots that shared the same overall conditions despite having the sun in a slightly different location: dawn, sunrise, morning, noon, afternoon, dusk and night. With such a differentiation I was able to freely locate the sun in the sky according to the current hour and map the rest of parameters to match reality. Bear in mind that the presets include a default azimuth and inclination, even though these will change depending on the hour of the day the user is viewing the demo.

To get the sun path in the given coordinates, I used the SunCalc.js library[8], which returns the sun position and phases according to the given location and time, created for the SunCalc.net project. Once I had information on which sun phase the scene was being rendered I changed the skybox parameters and sun position.

In table 3.2 you can find the values used for each state, followed by Figure 3.10 which clearly shows how they evolve during the course of the day.

	Dawn	**Sunrise**	**Morning**	**Noon**	**Afternoon**	**Dusk**	**Night**
Azimuth	0,25	0,25	0,35	0,50	0,65	0,75	0,24
Inclination	0,49	0,51	0,56	0,65	0,56	0,51	0,56
Rayleigh	0,00	0,60	1,20	2,00	1,20	1,50	0,00
Turbidity	2,00	1,00	0,50	0,30	0,50	2,00	0,50
mieCoefficient	0,21	0,03	0,35	0,01	0,35	0,15	0,03
mieDirectionalG	0,77	0,60	0,35	0,90	0,35	0,00	0,00
sunIntensity	1.300,00	1.000,00	900,00	1.300,00	700,00	2.000,00	1.400,00

Table 3.2: Skybox Shader uniforms and values along the day

Using the sun position I added the different lights to the canvas, with the colors corresponding to the different times of the day. For artistic reasons I decided to include three different lights: a white ambient light to brighten the scene, a white point light with intensity directly proportional to the sun inclination used to add realism and a directional

[7]Three.js sky + sun shader: https://threejs.org/examples/webgl_shaders_sky.html
[8]SunCalc GitHub Repository: https://github.com/mourner/suncalc

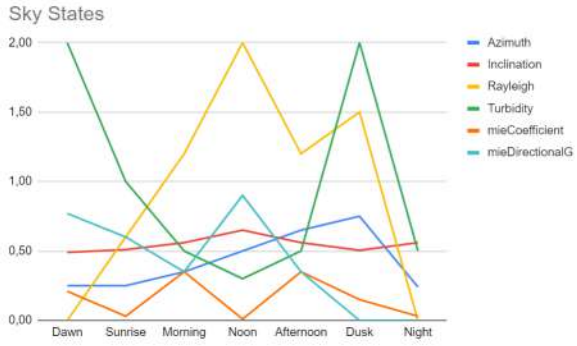


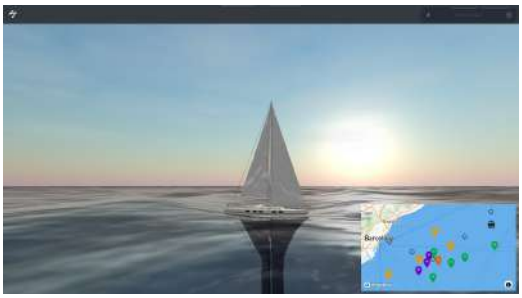
Figure 3.10: Chart of Sky Values along the day

orange light inversely proportional to the sun height used to add shadows and give a sunset effect.

Figure 3.11 shows how the sky changes at different hours near the city of Barcelona.

(a) Sky at sunrise (30/03/2023 7am)

(b) Sky at noon (30/03/2023 12pm)



(c) Sky at sunset (30/03/2023 19pm)

(d) Sky at night (30/03/2023 3am)

Figure 3.11: Different Sky States

Clouds

The most common method for displaying clouds in WebGL is using volumetric rendering techniques, such as raymarching. This approach gives the best results, however it is computationally costly. Another procedure which yields a satisfactory outcome is billboards but is expensive as well and would bring down the performance of the application.

Implementing these methods would be plausible in an environment where clouds are an

important factor or the focus of the scene, like a plane simulator to navigate through the clouds. However, for this use case, where clouds are going to be located on top, details are not going to be appreciated and using such methods would be misspending resources. Taking into account that in this context there is no need for hyperrealism, a low cost solution is to add a plane with a perlin noise texture (Figure 3.12).

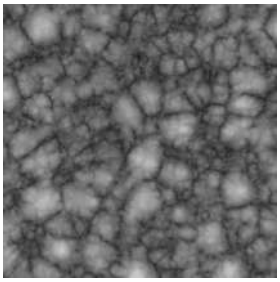


Figure 3.12: Perlin Noise Texture

In order to add it to the scene, the mesh is formed by an xz-plane geometry and a custom material. If the shader just mapped the perlin texture to the uv coordinates we would have static clouds, which would not be convincing. To solve this, the uv's needed to be modified according to the wind direction and speed:

$$cloudSpeed = 0.1 \cdot windSpeed \cdot t$$

$$uv = uv - (windDirection \cdot cloudSpeed)$$

In this case, I used `MirroredRepeatWrapping` so the texture repeats and it mirrors the edges so the change is less noticeable.

As for the fragment shader, we needed to have control over the percentage of cloudiness, this was achieved using the alpha channel of the perlin noise weighted by a cloud cover parameter sent to the shader as a uniform:

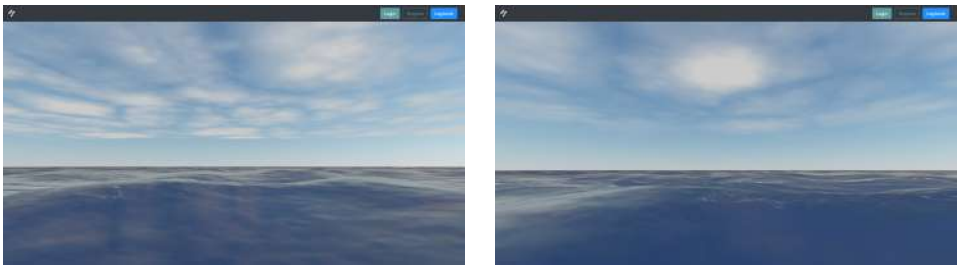
$$alpha = texture2D(perlin, uv).r$$

$$alpha = alpha \cdot cloudCover$$

The color chosen was a light gray, and changed its opacity according to the previous equation. This gave good results, however at the plane borders the cut was evident; to smooth the edges I attenuated by distance the opacity of the clouds:

$$distance = \frac{|vertexPosition - cameraPosition|}{oceanSize}$$

$$alpha = alpha \cdot (1.0 - distance)$$



(a) Without Attenuation (b) With Attenuation

Figure 3.13: Difference on the edges of the cloud plane in attenuating by distance.

Rain

To have multiple weather conditions, 3D rain needed to be included. Doing research I found many shaders that implemented it in 2D like water drops on a window [14], and for tridimensional rain the most usual way to implement it is by a system of particles. In spite of this, I found Javier Agenjo's approach really clever and I decided to carry it out. He suggests using a mesh and animate the texture in the shader, thus improving the performance and still obtaining satisfactory results.

On the geometry, he created a bunch of vertical stripes in cross form, so the drops could be seen from any perspective. As for the material, I created a custom shader, the vertex shader is responsible for distorting the vertices and moving the uv coordinates and the fragment will define the final color.

To tilt the geometry, the vertices needed to be displaced according to their height and weighted by the wind speed in order to accentuate the slope more or less. Also, the wind direction was an important factor, to incline the vertices in the correct orientation, according to the following formula:

$$vertexPosition_{xz} = vertexPosition_{xz} - windSpeed \cdot windDir \cdot vertexPosition_{y}$$

Regarding the uv's, they needed to be displaced in the y direction according to the rain speed. However, by just doing this, all the raindrops fall at the same time in each of the stripes. To solve it, J. Agenjo introduces a sinusoidal factor that displaces the uv's based on the position.

$$v = v - t \cdot rainSpeed + factor$$

$$factor = 0.1 \sin(0.1 \cdot vertexPosition_{x}) + 0.2 \cos(0.1 \cdot vertexPosition_{z})$$

To color the rain material, using the uv's I sampled the water drops texture to get the opacity and droplet form, and through uniforms I set the color. Agenjo also weighted the alpha by a fresnel factor, depending on the camera position and the surface normal.

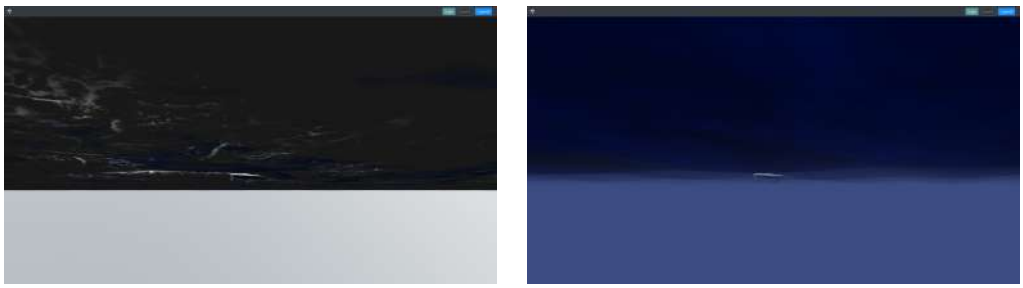
Another important thing to take into consideration, was the occlusions formed by the rain mesh and by other scene elements with it. Not only did I need to change the `depthWrite` to `false` so the mesh did not block other scene nodes, but also change the `renderOrder` of the rain component, so it got rendered after the rest of the scene.

Fog - Surface and Underwater

Concerning the visibility of the scene, Three.js offers a fog that affects all of the elements being rendered and changes its color according to their world position, the fog density and other parameters. This offers good results when you are working with a constant background color that matches your fog color, however, in this use case the skybox is constantly changing and not uniform, therefore including the default fog produced unpleasant results. One approach could have been to modify the shader to change the fog color by throwing a ray to the skybox, but since it is procedural this was not practical nor feasible.

Keeping in mind that above the surface the main element being affected is the sea, I modified its fragment shader in a very simple, yet effective way. I reduce the opacity by distance and weighted by a fogFactor, the further the vertex, the more transparent. This allows one to see what is below the surface, which in this case is the skybox, blending perfectly with the environment and giving the mist effect.

As for below the surface, rather than talking in terms of visibility, it needed to give the sensation of being underwater. Hence, the previous technique could not be applied, yet it could still be approached using fog. In this case, the default three.js fog shader worked with positive results. Nonetheless, by just adding it to the scene, the fog was applied everywhere, even above the surface which was not desired. To solve this problem, I overrode the default three.js fog shader chunks so that they only applied when the camera was below zero. In a similar manner, when watching the undersea scene, I did not render the skybox, so the blue background color can be seen and blend with the fog. For this case, the ocean needed to be rendered on both faces, front and back, so I defined the `side` rendering as `DoubleSide`. When seeing from below the surface, only the diffuse color is taken into account as in the beginning reflections were also rendered, creating artifacts and not being physically realistic.



(a) Without Underwater Effect (b) With Underwater Effect

Figure 3.14: Scene seen from below the surface

OpenWeather API

All of the prior elements were added using presets and sliders to modify the parameters on run-time. For the purpose of the application, which is to show the current weather

conditions of the user's location, this could not be final. I connected the system to a meteorological service API, OpenWeatherMap, to fetch the current weather in the desired coordinates and time.

Once I had the data, I mapped it to match the presets or variables I had within the desired range. For the rain, I used a chart provided by the USGS[9], which related rates of rainfall with millimeters per hour of water. The weather service gave the rain in mm but in my model I had a range of density from 0 to 2, and a speed from 0 to 1, to map it according to the rainfall intensities I used the following equation:

$$rainDensity = \lceil \frac{\sqrt{mm}}{2} \rceil$$

As for the haze, I added a fog factor in the shaders, yet the data retrieved from the API came in terms of visibility measured in meters. Table 3.3 helped me visualize and map the different fog states [15].

Code no	Weather Condition	Meteorological Range, *Rm*
0	Dense fog	50m
1	Thick fog	1 50m - 200m
2	Moderate fog	200m - 500m
3	Light fog	500m - 1000m
4	Thin fog	1km - 2km
5	Haze	2km - 4km
6	Light haze	km - 10km
7	Clear	km - 20km
8	Very clear	20km - 50km
9	Exceptionally clear	>50km
10	Pure air	277km

Table 3.3: International Visibility Code with Meteorological Range

I applied the inverse and mapped the values from 0 to 10 km visibility to a fog factor from 0 to 4. Due to artistic reasons, I added an offset of 0.6 to the mist so the edge of the ocean did not appear as sharp.

$$fogState = 4 \cdot \left(1.0 - \frac{visibility}{10000}\right) + 0.6$$

Mapping the clouds was straightforward, since I already worked with a percentage of cloudiness for the cloud cover factor. I changed the range of values from [0, 100] to [0, 4].

$$cloudCover = 4 \cdot \frac{cloudPercentage}{100}$$

3.4.3 Boat

The boat is the reference point of the world, centered in [0, 0, 0] and the rest of the elements of the scene are located accordingly. To have the scene dependent on the boat I stored in

[9]USGS rainfall calculator: https://water.usgs.gov/edu/activity-howmuchrain-metric.html

a variable the coordinates of the user's ship, which can be seen displayed in the 2D map. Since this is a pre-MVP the latitude and longitude are hardcoded, but the user can drag the boat icon in the map and place it anywhere, changing the surroundings of the 3D scene. The boat also changes its direction in the 2D and 3D canvas, according to the next route waypoint. To compute the angle it is pointing to, I used the Haversine Formula which computes distances and bearings between two coordinates [16]. The orientation defined the Euler Rotation in the Y axis. However, the boat also needed to move up and down and rotate based on the wave it is currently passing through.

To implement the buoyancy I computed the gerstner wave position and normal at the current vertex in the CPU. I use the height to move the boat in the vertical axis and the normal to tilt the boat in the X and Z axis, to give the sensation of movement. I apply a linear interpolation between the current Euler Rotation and the target. Ideally it would be best to work with Quaternions due to the limitations of Euler Angles, but having to work with different transformations (position, direction and rotation) at the same time made it complex.

3.4.4 Coastline

This element is highly dependent on the boat location, as it may be or not visible at all times. To place and rotate it in the scene I used a low cost approximation of the Haversine distance between the coordinates of the closest city and the boat, currently the closest city is hardcoded to Barcelona as having geolocated images was out of the scope of the project.

The straightforward way to display the city would be projecting it in the skybox, however this would only work in the case where the coastline was in the horizon, but as the boat got closer it would give an implausible outcome. One solution which yielded satisfying results was to use a semicircle geometry and map the city texture to it. It is important that the texture has an alpha channel to discard transparent pixels so as to have the correct silhouette of the town. Using this method, the mesh could be scaled, translated and rotated in the scene according to the boat location and where it is facing.

3.4.5 Buoys

In chapter 3.3.1 I talked about fetching the buoys from the database and how they are pinned in the map, in this chapter I will discuss how this data is mapped onto the 3D canvas. As the ship can be surrounded by multiple buoys and the mesh used is the same, only the location changes, I decided to use instancing to reduce render calls and optimize the application. The buoys hold information, such as their location, to place them in the scene with respect to the boat I used the Haversine formula as well, and the type of pin they are (jellyfish, photo...) defines the mesh color to match the 2D map pins.

Buoy instances also had to float with the sea waves. I used their position in the scene to get the corresponding ocean geometry vertex and compute the trochoidal wave at said

point. As opposed to the boat buoyancy, where the X and Z position were fixed and only height changed, all three coordinates of the buoy's position are linearly interpolated to their target value as they are not motorized and move freely with the surface. In the case of buoys no rotation was applied.

3.5 Logbook

As for the last requirement, the application needs to create a logbook as the user checks in the route waypoints and at the end create a video showing the different states at that moment.

To do so, I store in a JSON file the different waypoints with their coordinates and time the user got there, change the scene state after a few seconds and use the captureStream method to record the canvas element. Unfortunately, this functionality is not compatible with all browsers, as it is not currently supported on Safari, but it was the best choice when it came to capturing and HTMLMediaElement [17]. As the user can still be navigating and seeing the current state when they choose to record a video, I created a hidden canvas with the same scene and changed its state while keeping the main canvas visible and displaying the current information. The problem with having to canvas was that the performance got directly affected as I was rendering the scene twice, on and off screen. That is why I only render the second canvas on demand once the user clicks on the button to create a logbook video [18].

Once the video was recorded, I needed to include photos the user captured along the route. I decided to store this information in the json file and to render the picture as a screen quad on top of the rest of the scene. A problem faced was that images have different aspect ratios and the quad distorted the images or cropped them. The solution I found was to change the vertex quad shader to change its size according to the image aspect ratio.

Chapter 4

Results

When validating the work carried out during these months, the most important thing was to assure the correct integration and deployment of the application in the start-up and to get the client's feedback and satisfaction. After many changes, adjustments to the requirements and supervision on the final result, the company accepted the final deliverable and deployed it onto their cloud service correctly.

The application accomplishes all the requirements and part of the requisites for the commercial application. The initial requirements included a series of necessary and complementary things; during the project, priorities have changed, but the objective has been fully achieved to the satisfaction of the company. Regarding the requirements exposed in Chapter 1.2, these are the respective outcomes:

- **App Integration** The demo is implemented on the web[1], with the possibility of working in a mobile app, as it has been developed in React, a multi-platform technology. The company is already working on deploying the application; a first cloud portability was tested by the start-up with a successful outcome:

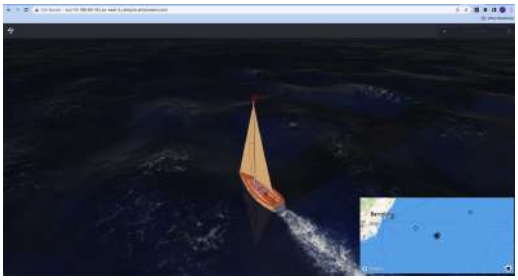


Figure 4.1: Screenshot of the frontend deployed on an AWS of Searebbel

- **Boat's Aspect** The demo has a "Control Panel" that allows the user to choose between 3 boat models. Refer to Figure 2.2.1 and 2.2.2 in Appendix A to see the changes.

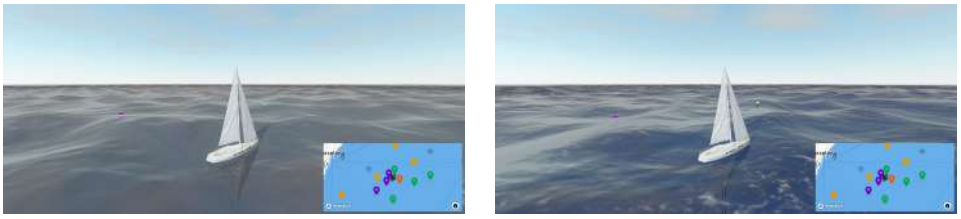
- **Solar Position** The app allows the user to simulate their location in the map and drag the boat to change it. The display reads the geoposition of the ship, knows its trajectory, and according to the position and time of the day, the skybox changes and displays the sun or moon in the correct location with respect to the 3D model of the ship, as well as the height above the horizon and the aspect of the light. This is functional at all hours of the day and all locations, as it can be seen in Figures 2.3.x of Appendix A.

[1]Web Demo: https://webglstudio.org/users/cdelcorral/searebbel/

- **Coastline** The demo fakes this case in the shoreline of Barcelona, displaying a texture in the horizon. However, it is geo dependent and the location of the city in the 3D scene changes subject to the boat coordinates. This can be seen in Figures 2.4.x of Appendix A.
- **Weather** The canvas is connected to a real time weather data service in order to render the scene accordingly. Refer to Figures 2.5.x in Appendix A for more information.
- **Other ships' location** The demo only displays the current user's boat.

The last meetings were focused on refining the project and implementing the desired changes, demonstrating that the project was delivered within the designated time frame. Several rounds of improvements have been done, including:

- Enhancements to the sea color, making it more vibrant.



(a) Before Changes (b) After Changes

Figure 4.2: Difference on the water color

- Creating a sense of speed through the foam and the improved wake (Chapter 3.4.1).
- Achieving an underwater look (Chapter 3.4.2).

These modifications prove that the final demo has gone beyond what could have been expected.

Find attached in Appendix B Sections 4 and 6 of the documentation delivered to the European Commission which contains a table with a comparison side by side with the requirements exposed in Chapter 1.2 and the actual outcome.

Besides the final satisfaction round with the client, I decided to carry out an internal test, a significant validation was to check the frame rate of the application as it needs to run on mobile devices. On my development computer, an *MSI GS63 Stealth 8RE with an NVIDIA GeForce GTX 1060*, the application runs at a constant framerate of 120fps. Whereas in the mobile phone *Redmi Note 8 Pro with a Mali G76MC4 GPU* it runs at 60fps. This proves that the demo delivers an excellent performance on home devices, ensuring portability to a mobile application in the future.

Chapter 5

Conclusions and Future Work

In this thesis different computer graphics techniques have been used in order to achieve the final product. Custom shaders have been implemented to change the geometry vertex of the sea and develop my own fragment shader for it; taking into account WebGL limitations to make the most out of the resources available. Instancing was also utilized to improve the web-based demo performance, keeping in mind that this was being developed in a React Native environment.

Despite the things that could be improved on the technical aspects, my goal for this thesis has been accomplished, demonstrating that it can be totally functional outside of the pre-MVP. Keeping in mind that many features had to be implemented in a short period working part-time, therefore the end product is not the best result that could have been achieved had the development team been larger or had I had more time to carry it out.

Regarding my thesis, I would have liked to do more research on sea rendering techniques however there was little to no time as the demo had to be functional. Along the project, the workflow has been to build up from the skeleton and improve features once they were all operative. In the end, this demo is a proof of concept to demonstrate that all the requirements could be done, so the prioritization of quantity has taken precedence over the quality. Bearing in mind the previous statement, I am pleased with my work, even though I would have liked to further elaborate on the quality.

Furthermore, working in a React environment has been a challenge that took over a lot of effort. Having used another technology, like vanilla Three.js would have made things easier and straightforward. Working with React Three Fiber was almost mandatory to guarantee the functionality of the future commercial application and therefore an implicit hurdle that needed to be overcome. Overall, I think R3F is a powerful and easy to use tool for easy tasks but as the complexity of the assignment increases, the more difficult it gets to implement as many functionalities get encapsulated. That is why R3F is great for artists or novice developers, nonetheless for more complex commissions other tools should be explored before working in React Native.

The future iterations of this work would be to implement breaking waves for a more realistic marine experience. As well as developing better waves besides the Gerstner algorithm, and attempting to support Tessendorf's approach for an excellent visualization. Furthermore, the repeatability of the ocean waves could be explored and overcome using simple methods such as attenuation by distance or fancier approaches like patch stitching could be looked into. Along the lines of the work done regarding the water surface, foam particles could be added to bring realism to the scene and improve the sensation of movement. Working with particles could imply the enhancement of surface fog. Additionally, rain drops hitting on water could be added to the fragment shader of the ocean.

The next work streams could also involve creating an underwater scenery, improving the current submarine effect and implementing caustics. As for the ship, better buoyancy could be supported; right now it focuses on the center of the boat and rotates on every crest, making the boat move excessively. To add more realism to the boat, a mesh with sails modeled separately could be found and move them in accordance to the wind. Finally, resources could be found or created to fully support multiple coastlines, for instance Google Earth's system could be explored and use libraries like Maps Models Importer by Elie Michel to import the shore models into Blender and add them to the application [19].

Figure 5.1: Screenshot of Google Earth's view on the Barcelona coastline

Bibliography

[1] Jerry Tessendorf. "Simulating Ocean Water". In: (2001). URL: https://people.computing.clemson.edu/~jtessen/reports/papers_files/coursenotes2004.pdf. (Accessed: 11th september 2022).

[2] Alain Fournier and William T. Reeves. "A Simple Model of Ocean Waves". In: *SIGGRAPH Comput. Graph.* 20.4 (1986), 75–84. ISSN: 0097-8930. DOI: 10.1145/15886.15894. URL: https://doi.org/10.1145/15886.15894.

[3] Claes Johanson. *Real-time water rendering - Introducing the projected grid concept.* URL: http://habib.wikidot.com/projected-grid-ocean-shader-full-html-version. (Accessed: 10th august 2022).

[4] Guanyu He and Hao Wu. "WebGL Ocean Simulation using Fast Fourier Transformation". In: *JCGT* 2.1 (2013). URL: http://www.wuhao.co/uploads/2/6/0/1/26012804/paper_final.pdf. (Accessed: 3rd may 2023).

[5] Mapbox. *Mapbox Documentation - Hybrid frameworks.* URL: https://docs.mapbox.com/help/getting-started/mobile-apps/#hybrid-frameworks. (Accessed: 3rd august 2022).

[6] Mapbox. *Mapbox Documentation - Getting Started.* URL: https://docs.mapbox.com/help/tutorials/use-mapbox-gl-js-with-react/#getting-started. (Accessed: 3rd august 2022).

[7] Nikita Duggal. *All You Need To Know About MERN Stack.* URL: https://www.simplilearn.com/tutorials/mongodb-tutorial/what-is-mern-stack-introduction-and-examples#:~:text=MERN%20stack%20is%20a%20collection,stack%20are%20all%20JS%2Dbased. (Accessed: 1st september 2022).

[8] Leanne Werner. *ShaderMaterial vs RawShaderMaterial.* URL: https://medium.com/@leannewerner/shadermaterial-vs-rawshadermaterial-f1f0def5722. (Accessed: 19th June 2023).

[9] Paul Henschel. *Water Shader.* URL: https://codesandbox.io/s/1b40u. (Accessed: 21st November 2022).

[10] Carlos Gonzalez and Doug Holder. *Water Technology of Uncharted.* URL: https://cgzoo.files.wordpress.com/2012/04/water-technology-of-uncharted-gdc-2012.pdf. (Accessed: 10th December 2022).

[11] Jonas Wagner. *Rendering Water with WebGL.* URL: https://29a.ch/slides/2012/webglwater/#title. (Accessed: 4th June 2023).

[12] Windy App. *Learn to measure wind speed and power conditions by the Beaufort scale.* URL: https://windy.app/blog/wind-speed-beaufort-scale.html. (Accessed: 4th June 2023).

[13] A. J. Preetham, Peter Shirley, and Brian Smits. "A Practical Analytic Model for Daylight". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '99. 1999, 91–100. URL: https://doi.org/10.1145/311535.311545.

[14] Lucas Bebber. *Rain Water Effect Experiments*. URL: https://tympanus.net/codrops/2015/11/04/rain-water-effect-experiments/. (Accessed: 4th June 2023).

[15] Devyani S. Deshpande. "Analysis of the atmospheric visibility Restoration and fog attenuation using gray scale image Ms". In: 2016.

[16] Movable Type Scripts. *Calculate distance, bearing and more between Latitude/Longitude points*. URL: https://www.movable-type.co.uk/scripts/latlong.html. (Accessed: 5th June 2023).

[17] MDN. *HTMLMediaElement: captureStream() method*. URL: https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/captureStream. (Accessed: 5th June 2023).

[18] React Three Fiber. *Scaling performance*. URL: https://docs.pmnd.rs/react-three-fiber/advanced/scaling-performance. (Accessed: 5th June 2023).

[19] Elie Michel. *Maps Models Importer*. URL: https://github.com/eliemichel/MapsModelsImporter. (Accessed: 5th June 2023).

Appendix A

GALATEA Scope Documentation

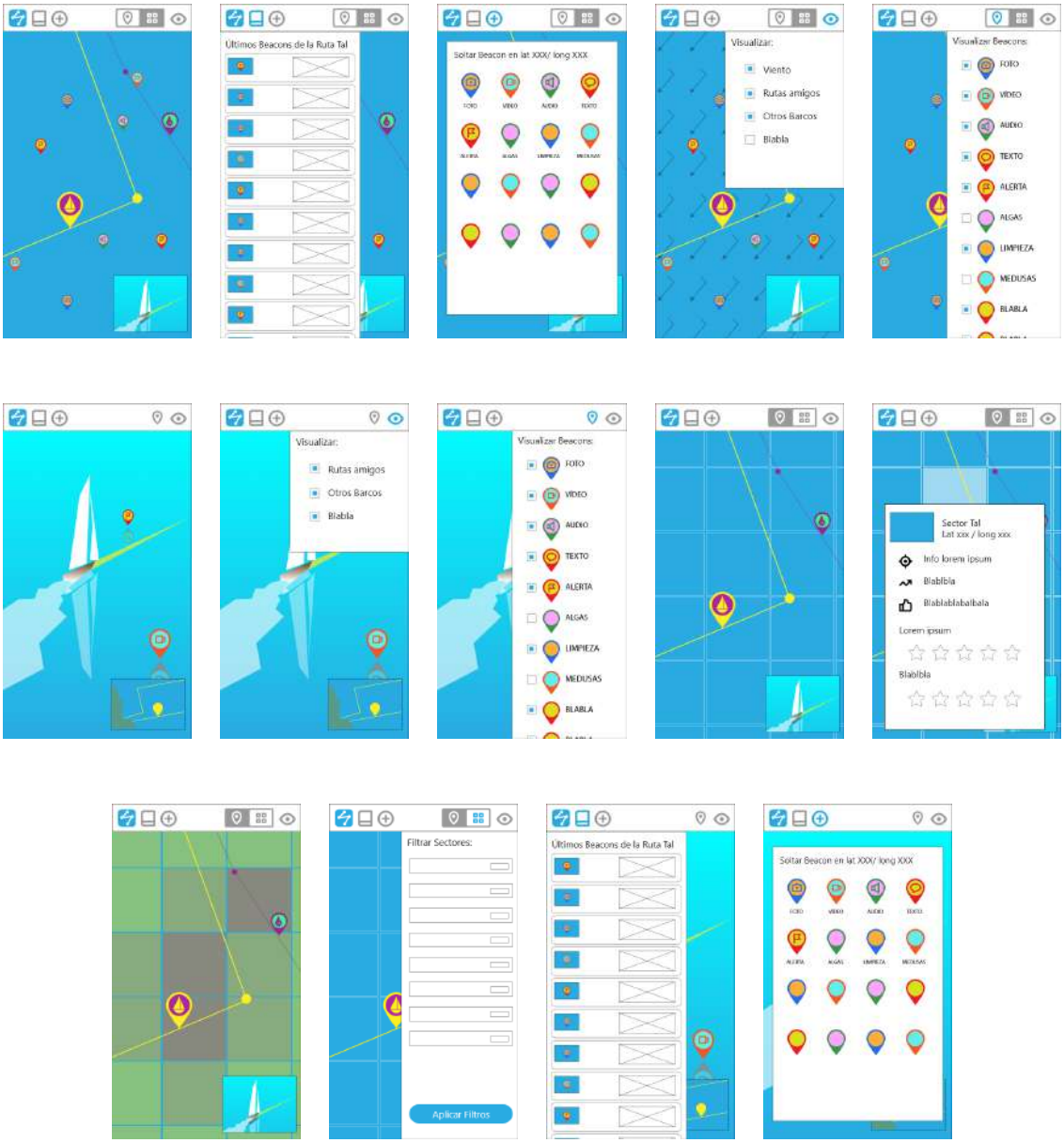


Figure A.1: Wireframes of the application.

Gamificación en SeaRebbel App - Desarrollo a plantear en el contexto de Galatea

SeaRebbel desea cambiar la forma en la que navegamos, haciendo la navegación mucho más sencilla y accesible para todas las personas.

El uso de cartas náuticas es aburrido, contienen mucha información de detalle difícil de interpretar para aquellos usuarios menos expertos, y menos para la familia u amigos, quienes también son parte muy importante de la experiencia.

Navegación en 3D

El uso de una vista alternativa de la navegación utilizando un visor 3D, con embarcaciones varias y divertidas (Llaüt, súper yate, colchoneta hinchable, barco volador, etc…), con vistas a la costa y simulando las condiciones metereológicas existentes en el estado del mar, la situación del sol, etc.. permiten una original e innovadora vista que es además fácilmente entendible por todos. En el siguiente post, el desarrollador principal, Javi Ajenjo explica cómo realizó esta tecnología https://www.tamats.com/blog/?p=715 en el contexto del juego del Barcelona World Ocean Race del año 2015.

Para disponer de esta vista de navegación en 3D, se contemplan necesarios los siguentes desarrollos en los que se proponen dos fases diferenciadas de desarrollo: Pre-MVP y la aplicación comercial.

Item	**DEMO (Pre-MVP)**	**Commercial APP**
Integración en APP	Seleccionar la visualización 3D abrirá la demo 3D en una ventana de navegador.	La visualización 3D deberá verse integrada en un contexto de la propia APP.
Aspecto de la nave	Aspecto básico / elegir entre tres modelos simples.	La APP debería permitir seleccionar /personalizar hasta cierto grado el aspecto de la nave (por ejemplo permitiendo elegir entre diferentes modelos 3D de las embarcaciones más comunes -diferentes clases y tamaños- y elegir entre un rango de colores/texturizados). Esta “feature” no sería estrictamente necesaria en la primera versión comercial de la APP, las opciones de personalización podrían incluirse en actualizaciones posteriores.

Posición solar (geoposición + hora)	La visualización deberá leer la geoposición del barco y conocer su trayectoria, sumado a la hora del día, para representar correctamente la posición sol en el cielo respecto al encaramiento del modelo 3D de la nave, así como la altura del mismo sobre el horizonte y el aspecto de la luz. Del mismo modo, si es de noche deberá representarse así en la visualización.	La visualización deberá leer la geoposición del barco y conocer su trayectoria, sumado a la hora del día, para representar correctamente la posición sol en el cielo respecto al encaramiento del modelo 3D de la nave, así como la altura del mismo sobre el horizonte y el aspecto de la luz. Del mismo modo, si es de noche deberá representarse así en la visualización.
Línea de costa (geoposición)	No/Parcial (puede funcionar solo en una zona concreta o utilizar algún recurso para "simular" una línea de costa genérica).	La visualización deberá usar la geoposición del barco para representar una visualización de las líneas de costa visibles desde el mismo. La tecnología actual permite cargar una textura como línea de costa representada en el horizonte. Se necesitará encontrar/generar un banco de recursos visuales para poder generar estas líneas de costa (podría realizarse por fases, empezando a un nivel local), así como generar el algoritmo que permita seleccionar la línea de costa a representar según la geoposición.
Clima (geo + servicio meteo)	No/Simulado(fake)	La visualización se conectará a un servicio de datos meteorológicos en tiempo real para (actualmente la tecnología ya se encuentra preparada para esto).
Posición de otras naves	No	La APP podrá representar en la visualización 3D las naves (con su aspecto personalizado) de otros usuarios de la APP que se encuentren en un radio cercano.

Gamificación y socialización

Pero SeaRebbel no únicamente quiere facilitar el uso, sino que pretende crear una experiencia única. Una ruta empieza en su planificación, compartiéndola con las personas que nos van a acompañar, se disfruta también durante la navegación, y posteriormente nos deja un recuerdo que podemos compartir con otras personas.

En la planificación SeaRebbel quiere ayudar a los usuarios a descubrir nuevas zonas y rutas de acuerdo a las preferencias de cada uno, del tiempo disponible, el tipo de embarcación, las condiciones meteorológicas, y otras variables de mucho valor, como el lugar de fondeo, estado de nitidez del agua, etc…

Por tanto, los datos juegan un papel importantísimo para el futuro de SeaRebbel. Muchos de estos datos que nos permitirán proponer experiencias únicas son públicos, tales como las previsiones metereológicas, algunos datos de batimetría e incluso de tipo de fondo marino, etc.., sin embargo, existe otra información que es muy interesante poder contar con la colaboración de los usuarios: claridad o nitidez del agua en un momento concreto, si hay medusas, plásticos, riqueza de su fondo para hacer submarinismo, o qué tipo de actividad hemos realizado: nadar, bucear, pescar…

Por ello, en el alcance del proyecto, pretendemos desarrollar una aplicación donde los usuarios nos informen de esta información de valor y para ello les vamos a tener que gratificar. Una forma muy directa de gratificación es poder disfrutar de los datos compartidos por los demás usuarios, y una segunda forma es entregarles un resumen de la ruta realizada en un formato que les genere un muy buen recuerdo y que lo deseen compartir con sus familiares y amigos.

El concepto es sencillo, durante la ruta se irá almacenando un logbook con la ruta, velocidades, pausas, etc…, y se podrán añadir fotografías e inputs de los usuarios.

Por ejemplo, al finalizar la ruta, la aplicación nos solicitará si deseamos compartir la experiencia con un vídeo de la ruta. Si le decimos que sí, nos solicitará, si no lo habíamos hecho, que etiquetemos la calidad las aguas en los lugares que hemos realizado parada, y a continuación se generará un vídeo de 20-30 segundos en un formato mixto 2D y 3D, para compartir por whatsapp o publicar en las redes.

Además, para ampliar el efecto de viralización podremos configurar notificaciones push para que se nos informe en un radio determinado de publicaciones de otros usuarios si hay medusas o plásticos.

Qué información podemos pedir a los usuarios:

Nitidez del agua: Escala entre 1 y 10
Medusas: Ninguna, alguna aislada, algunas, muchas, imposible nadar
Plásticos y suciedad: muy limpio, muy poca suciedad, zonas de suciedad aisladas, suciedad general
Otras embarcaciones: Estábamos solos, muy pocas embarcaciones, algunas embarcaciones, muchas, imposible fondear

Si has venido otras veces: Hoy estaba mejor que normalmente, estaba igual, estaba peor
Cómo valoras en general la zona para:
-Nadar: No puedo valorar, 1-10
-Bucear: No puedo valorar, 1-10
-Pescar: No puedo valorar, 1-10

Propuesta de proyecto a desarrollar:

El proyecto se planifica a 8 meses, empezando el 1 de Mayo.

La DEMO se presentará como un software todavía no integrado en la APP definitiva, sino que podrá utilitzarse como un demostrador independiente de las capacidades interactivas y sociales futuras de la APP a posibles inversores.

Planteamos la DEMO como una aplicación que constará de tres contextos, correspondientes a los 3 concerns principales alrededor de los que que se articulará el uso de la aplicación antes, durante, y después de la experiencia de la navegación:

Contexto de Planificación	Funcionará como una visualización 2D en la que el usuario podrá navegar el mapa y planificar la ruta a seguir marcando puntos en el mismo.
Contexto de Navegación	Combinará la visualización 2D con la visualización 3D. Podremos seguir la geoposición de la nave durante el trayecto en el mapa, y en su contrapartida tridimensional veremos el avatar de la embarcación en su contexto, con las condiciones climáticas, la posición del sol relativa, y la representación virtual de la línea de costa si es visible. Para la demo, se elegirá una zona/ línea de costa concreta, o se buscará una manera alternativa de simular la misma si no se dispone de los recursos definitivos. Del mismo modo, la velocidad a la que transcurre el trayecto, así como la climatología durante el mismo, etc., podrá ser simulada/alterada para obtener una experiencia más óptima para una demostración (en contraposición a un uso real).
Contexto de Experience-sharing	Este último contexto es el que permitirá generar un clip de video que resumirá la experiencia combinando las visualizaciones 2D y 3D y posible información (fotografías, comentarios, etc.) introducida por los usuarios durante el viaje (que, nuevamente, en el contexto de la demo podrán ser simuladas).

Se facilitará al desarrollador un documento de diseño funcional con wireframes de la interfaz y explicaciones detalladas del funcionamiento de la DEMO. Dicho esto, no tiene por qué tratarse de un diseño cerrado, y nuestra propuesta ideal para el desarrollo pasaría por contar con un desarrollador que quisiera participar activamente del co-diseño aportando su feedback e ideas propias al proyecto partiendo de su expertise previo en materia de visualización y navegación..

Appendix B

GALATEA European Documentation

Progress performance final report template for the GALATEA vouchers

[…]

Section 4. Project progress in reporting period

Describe briefly the actions taken and the results obtained during the reporting period regarding the planned work packages, deliverables and milestones detailed in your project proposal.

4.1 Work plan

Explain the work carried out during the reporting period and include an overview of the results in line with the Annex I of the GALATEA Sub-grant Agreement.

The work carried out during this period is made of three main Work Packages, which are part of the requirements and development of the pre-MVP, which is a web application demonstrating all the interactive and social capabilities of the app. Additionally, the development carried out is extensible and scalable, as it has been done in React, which easily adapts the interface to the user's environment, including mobile. The milestones and requirements of each WP will be detailed in the next section.

WP1: Planning context

This WP is carried out through the 2D map, and it mainly corresponds to the planning of the route, where the user can select its waypoints. It is also available during the actual navigation, where the user can see their boat located in the map and leave *buoys* of information along the way. The latter is a key ingredient of the demo, and has been successfully achieved. The web application connects to a MongoDB database, in order to add and retrieve the pins of information. Currently, the system of stored information is basic, as it is just a proof of concept, keeping in mind that it can be extended in the future. The outcome of this WP is a map, provided by Mapbox - implemented in React using Mapbox GL JS -, where the buoys of information are displayed on top, and the user can click once to view its content and double click on the map to add a new pin. The annex shows screenshots of the results in Figures 1.1, 1.2, 1.3 and 1.4.

WP2: Navigation context

A 3D view is also available during the navigation process, where the user can switch between the *2D map* and the *3D map*. The latter map visualises its ship sailing the sea according to the current weather conditions, the sun location and the coastline if needed. These months work revolves around this WP as it technically is the most complex one and the main objective of the proof of concept. The scene has been implemented using React Three Fiber, a React renderer for Three.js, giving scalability to the demo for a practical integration in the commercial App. In the milestones section the different requirements will be discussed and explained. As an overview of the results, refer to Figure 2.1 in the annex.

WP3: Experience-sharing context

As for the experience sharing part, as the user reaches the route waypoints and interacts with the application, a logbook is stored. Once the journey is finished, a video combining the 3D view during the journey and information (pictures, comments…) introduced by the user along the trip is generated. In the demo, there is a button to create and download such a video. In this case predefined routes and weather conditions are used to generate the video, to prove that the creation of such a file is possible. Refer to figure 3.1, 3.2 and 3.3 of the annex to see the workflow, user experience and results of this goal.

4.2 Milestones' status

WP1 and WP3 do not contain specific milestones besides the main outcome, already mentioned, documented and shown in the annex. Within WP2 a number of milestones exist that needed to be reached to achieve an excellent result. The following table provides a summary of the work, showing the functionalities planned for the pre-MVP, its relationship to the commercial app, and the actual outcomes developed.

Item	Demo (pre-MVP)	Commercial App	Actual outcomes
App integration	Selecting the 3D display will open the 3D demo in a browser window.	The 3D visualisation should be integrated into the context of the APP itself.	The demo is implemented on the web, with the possibility of working in a mobile app, as it has been developed in React.
Boat's Aspect	Basic appearance / choose from three simple models.	The APP should allow users to select/customise to a certain degree the appearance of the ship (for example allowing them to choose between different 3D models of the most common ships -different classes and sizes- and to choose between a range of colours/textures). This feature would not be strictly necessary in the first commercial version of the APP, the options of customization options could be included in later updates.	The demo has a *"Control Panel"* that allows the user to choose between 3 boat models. Refer to Figure 2.2.1 and 2.2.2 to see the changes.
Solar position (geoposition + time)	The display shall read the geoposition of the ship and know its trajectory, plus the time of day, in order to correctly represent the sun's position in the sky with respect to the 3D model of the ship, as well as the height above the horizon and the aspect of the light. Likewise, if it is at	Idem.	The display reads the geoposition of the ship, knows its trajectory, plus the time of day, in order to correctly represent the sun's position in the sky with respect to the 3D model of the ship, as well as the height above the horizon and the aspect of the light. This is functional

	night, it should be represented as such in the visualisation.		at all hours of the day and all locations, as it can be seen in Figures 2.3.x. Refer to milestone 1 for more details.
Coastline (geoposition)	No/Partial (may work only in a specific area or use some resource to "simulate" a generic coastline).	The visualisation shall use the ship's geoposition to represent a visualisation of the shorelines visible from the ship. Current technology allows loading a texture as a shoreline represented on the horizon. It will be necessary to find/generate a bank of visual resources to be able to generate these shorelines (it could be done in phases, starting at a local level), as well as to generate the algorithm that allows to select the shoreline to be represented according to the geoposition.	The demo fakes this case in the shoreline of Barcelona, displaying a texture in the horizon. However, it is geo dependent and the location of the city in the 3D scene changes subject to the boat coordinates. This can be seen in Figures 2.4.x.
Weather (geo + meteo service)	No/Simulated (fake)	The visualisation will be connected to a real-time weather data service for (the technology is now ready for this).	The canvas is connected to a real time weather data service in order to render the scene accordingly. Refer to milestone 2 and Figures 2.5.x for more information.
Other ships location	No	The APP will be able to represent in the 3D view the ships (with custom appearance) of other users within a nearby radius.	The demo only displays the current user's boat.

Milestone 1 – Solar Position (geoposition + time)

From the geoposition of the ship and its trajectory and the time of the day, the display should correctly represent the sun position in the sky with respect to the 3D model of the ship, as well as the height above the horizon and the aspect of the light. Likewise, if it is at night, it should be represented as such in the visualisation.

The web application connects to the *Suncalc.js* library, which, given some geographic coordinates and desired time, computes and returns the sun position as well as the sunlight phases (sunrise, sunset, dusk, etc). The 3D canvas displays these sky conditions, according to the retrieved information. In order to verify the correct working and results of this milestone, refer to Figures 2.3.x, where the scene is rendered at different times of the day. In the case of night time, the visualisation includes the moon location.

Estimated date according to project proposal

31st of December 2022

Has the milestone been reached in the planned date?

X Yes
☐ No

Milestone 2 – Weather (geoposition + meteorological service)

The visualisation will be connected to a real-time weather data service for realistic visualisation (the technology is now ready for this).

The web application connects to the *OpenWeather API*, which, given some coordinates and desired time, returns the atmospheric conditions, such as percentage of clouds, visibility, wind direction and speed, and rain density. Bringing together all the data, the application renders the scene environment accordingly. Furthermore, the sea state is also dependent on the weather, creating a more realistic view as it can be calm, with a moderate breeze, gale, etc. The waves parameters are designed in function of the wind speed, following the Beaufort Scale; the faster the wind, the rougher the sea. In order to verify the correct working and results of this milestone, refer to Figures 2.5.x, where the scene is displayed in different weather conditions.

Estimated date according to project proposal

31st of January 2023

Has the milestone been reached in the planned date?

X Yes
☐ No

Milestone 3 – Sensation of movement

The visualisation represents a boat sailing the sea, therefore the 3D scene needs to simulate the boat movement and make it look like it is moving forward toward the direction of the next route waypoint.

Different elements and components have been included in the scene to give the sensation of motion. For instance, the foam trail the boat leaves is created, as well as some foam on top of the peak of the waves. Furthermore, even though each wave has its own direction and properties, they are rotated in the opposite direction of the boat, so that the boat appears to be going against the sea motion, creating the optical illusion. Moreover, the boat should not appear static, as it should react to its surroundings and the waves. Thus the boat position and rotation are updated every frame according to the wave they are sailing through. In the annex screenshots allowing the reader its verification appear as Figures 2.6.x.

Estimated date according to project proposal

31st of March 2023

Has the milestone been reached in the planned date?

X Yes
☐ No

[…]

Section 6. Annex

Please include at least two images in jpg format (minimum 2000 pixels). If you have a project logo, you can share it too (hd format). Any additional document supporting the current report can be uploaded here.

Note: if any changes have been made to the project description, company logo, etc., please send updated information about these changes as well as any new logos, pictures, etc. in jpg, png, or illustrator format.

This annex contains the screenshots representing the outcomes of the project, organised according to the three work packages.

WP1 –

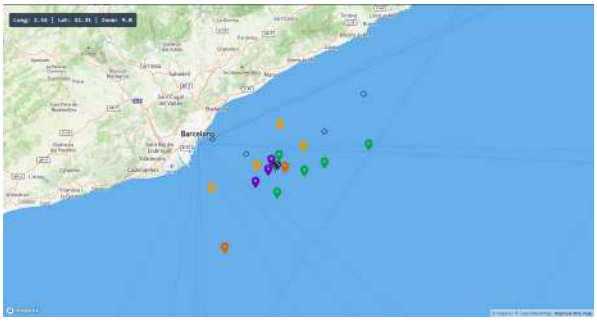


Figure 1.1: Final map with buoys displayed

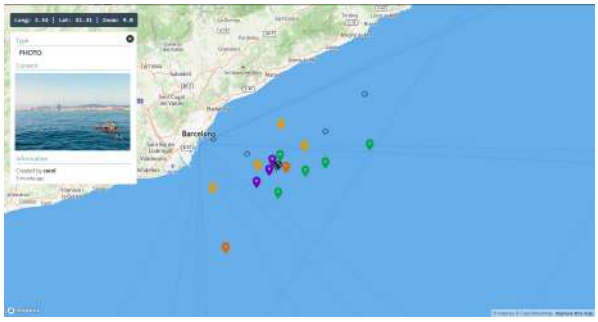


Figure 1.2: Display after clicking on a pin

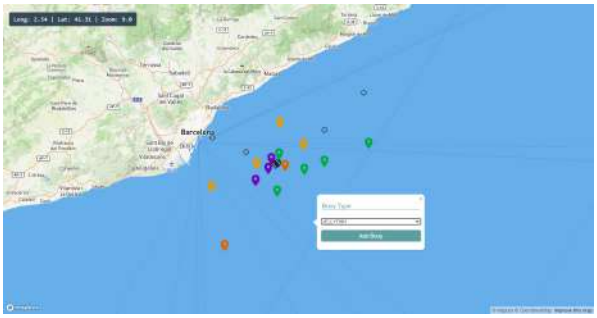


Figure 1.3: Add pin popup after click twice on the map

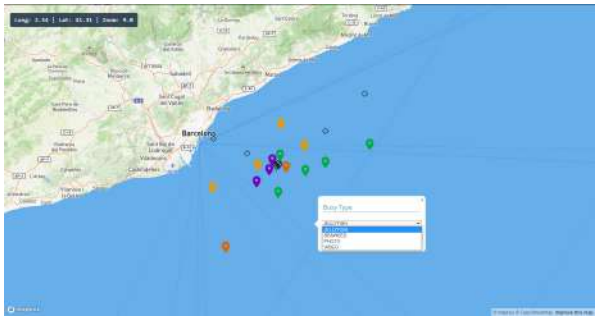


Figure 1.4: Fill form of adding new pin

WP2 –

Figure 2.1: Illustrative visualisation of the 3D scene

Figures 2.2 – Boat aspect:

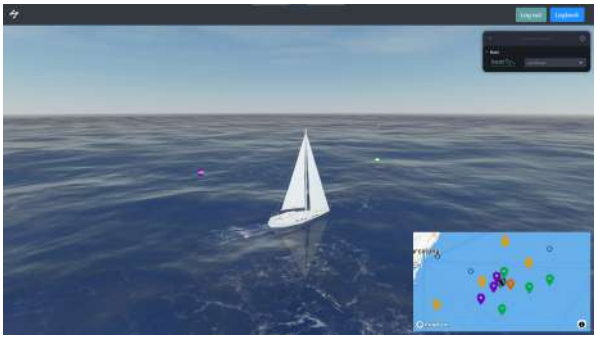


Figure 2.2.1: Sailboat model

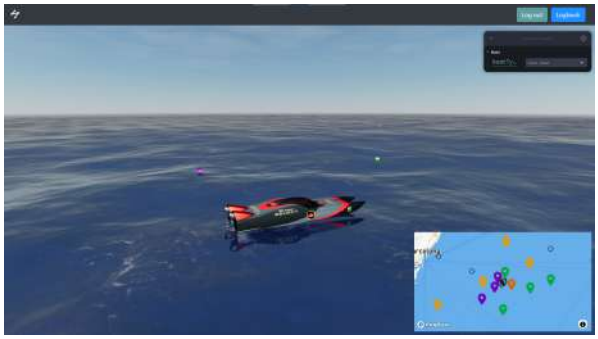


Figure 2.2.2: Race boat model

Figures 2.3 – Sun position:

These screenshots have been taken with the boat facing North, so that it can be clearly seen that the sun is located correctly at all times.

Figure 2.3.1: Sky at sunrise (30/03/2023 7am)

Figure 2.3.2: Sky at noon (30/03/2023 12pm)

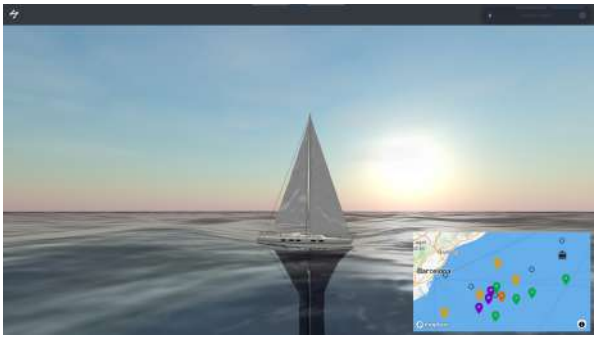


Figure 2.3.3: Sky at sunset (30/03/2023 19pm)

Figure 2.3.1: Sky at night (30/03/2023 3am)

Figures 2.4 – Coastline:

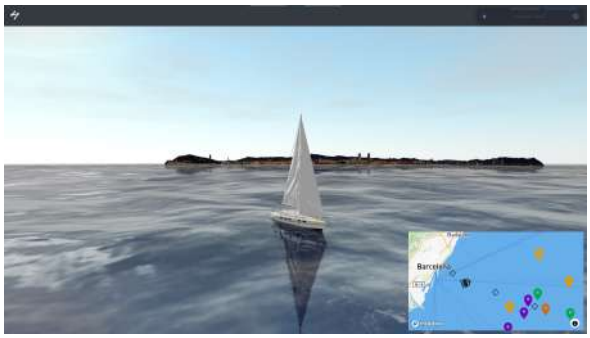


Figure 2.4.1: Boat close to Barcelona

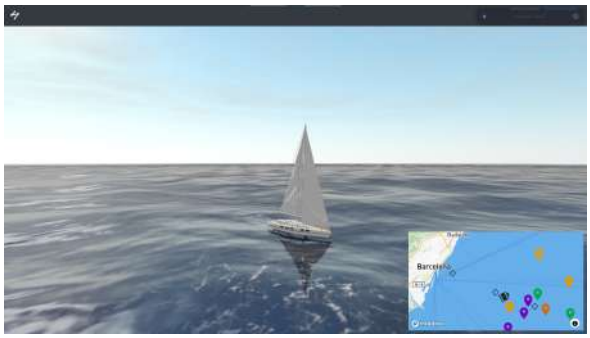


Figure 2.4.1: Boat far from Barcelona

Figures 2.5 – Weather:

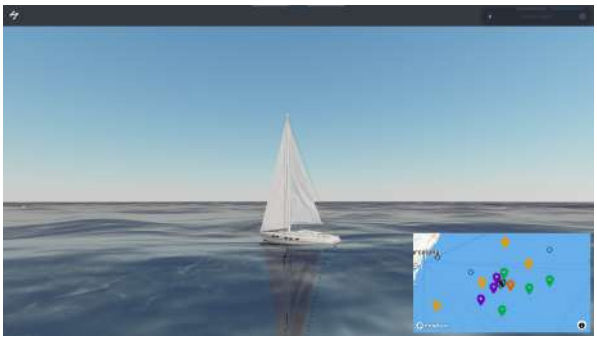


Figure 2.5.1: Cloudiness 0%

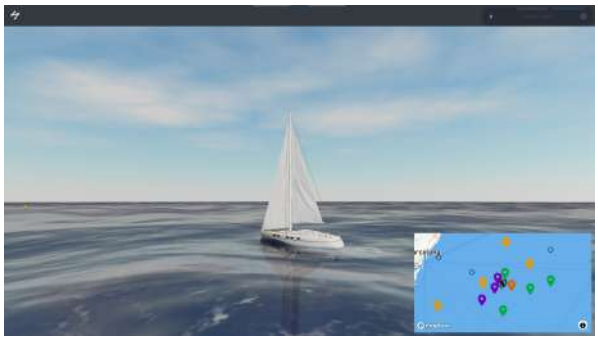


Figure 2.5.2: Cloudiness 70%

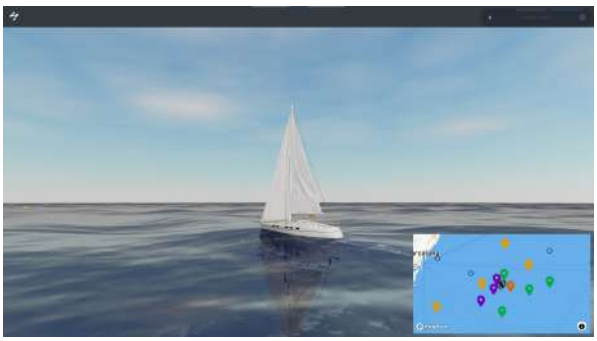


Figure 2.5.3: Visibility 10km

Figure 2.5.4: Visibility 1km

Figure 2.5.5: Rain 0 mm/h

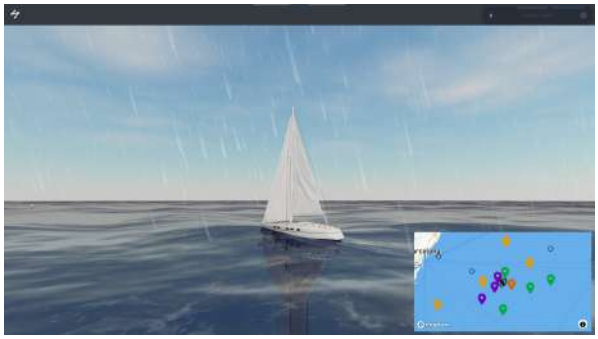
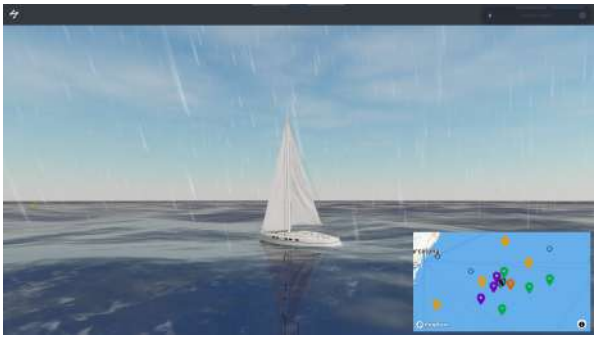


Figure 2.5.6: Rain 1 mm/h



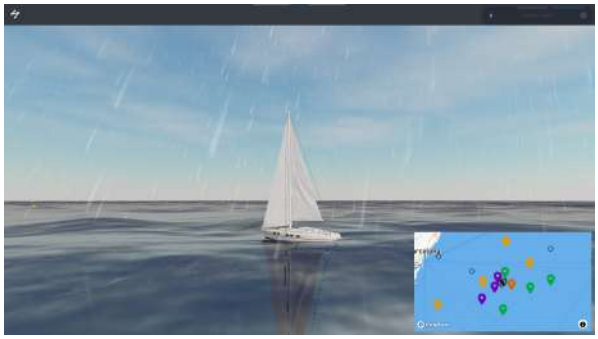
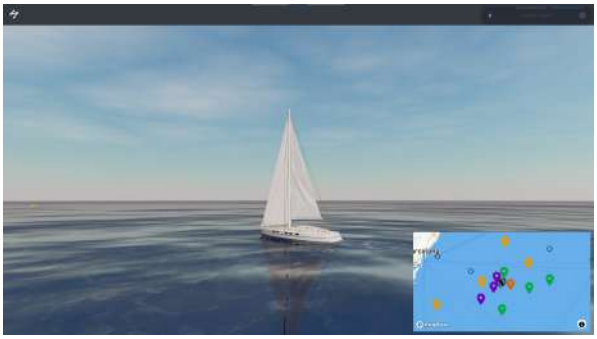


Figure 2.5.7: Wind from NE

Figure 2.5.8: Wind from SW



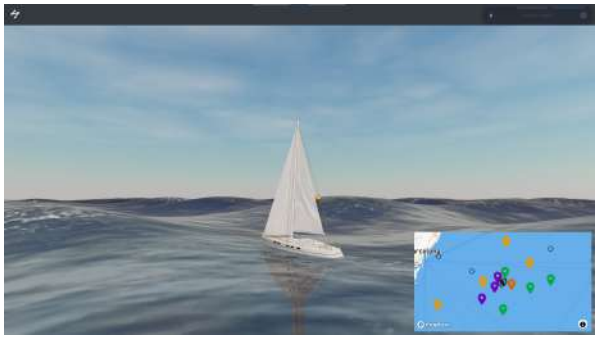
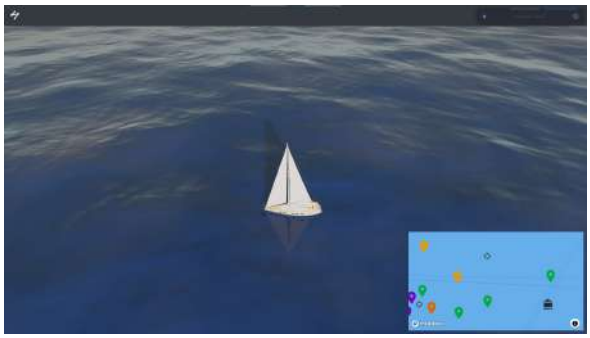


Figure 2.5.9: Calm sea

Figure 2.5.10: Gale sea

Figures 2.6 – Sensation of movement:



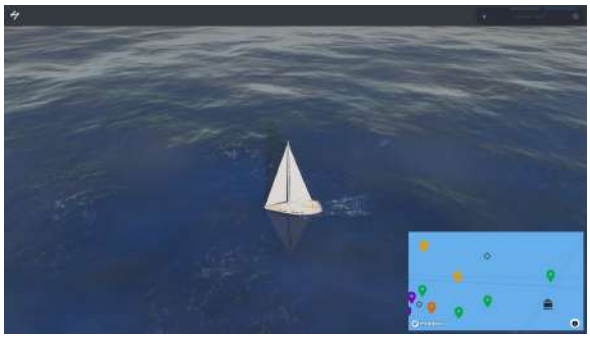
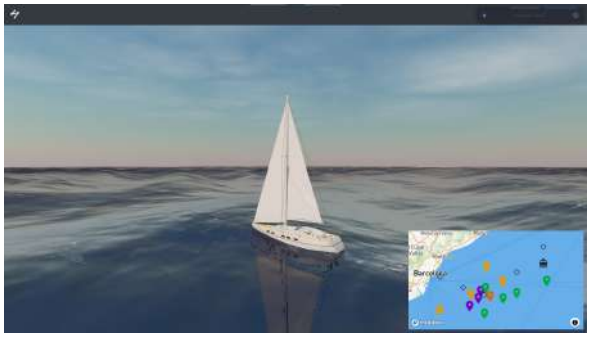


Figure 2.6.1: No foam

Figure 2.6.2: Waves foam + boat trail



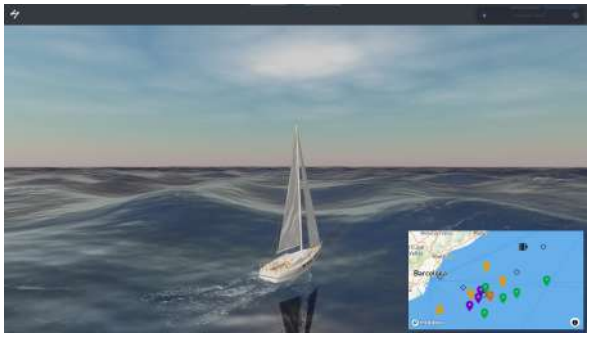


Figure 2.6.3: Boat rotation + waves direction

Figure 2.6.4: Boat rotation + waves direction

WP3 –

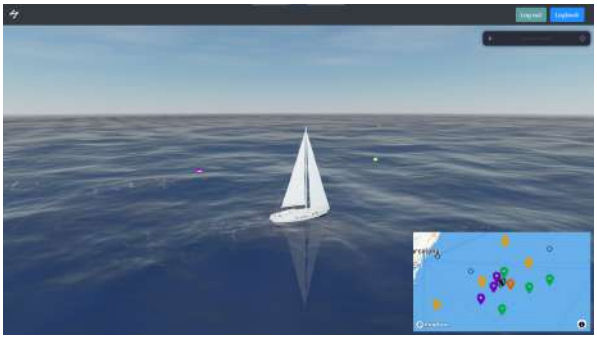


Figure 3.1: Main screen + Logbook button

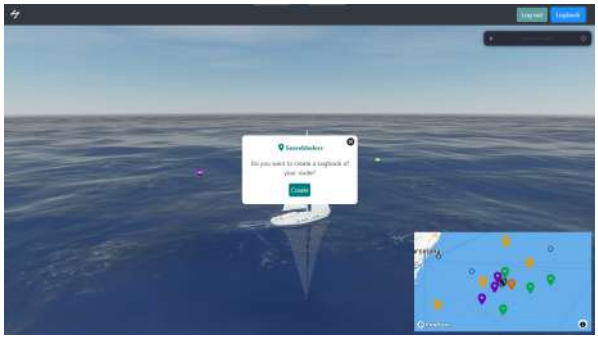


Figure 3.2: Display after clicking on Logbook button

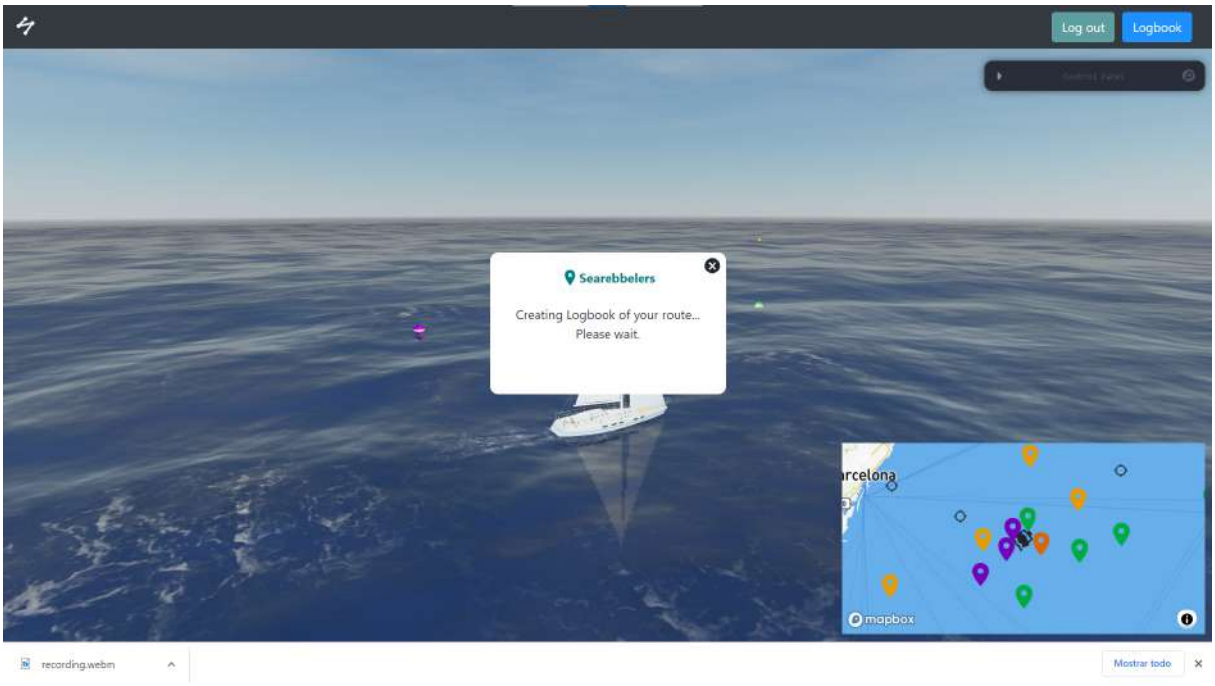


FIgure 3.3: Result after video is recorded and downloaded

Appendix C

A Guide on Three.js Shader Chunks

	#	FILE NAME	DEFINITION	PLACE	DEPENDENCIES	UNIFORMS
ALPHA MAP	1	alphamap_pars_fragment	defines texture uniform	header		common (alphaMap)
	2	alphamap_fragment	changes diffuseColor alpha value depending on texture	main()	vUv, diffuseColor	
ALPHA TEST	3	alphatest_pars_fragment	defines float uniform	header		common (alphaTest)
	4	alphatest_fragment	discards pixel if alpha below threshold	main()	diffuseColor	
AMBIENT OCCLUSION	5	aomap_pars_fragment	parameters (uniforms)	header		aoMap, aoMapIntensity
	6	aomap_fragment	reflectedLight *= ambientOcclusion	main()	22, vAoMapUv	
	7	begin_vertex	transformed = position	main()		
	8	beginnormal_vertex	objectNormal = normal	main()		
BSDF	9	bsdfs	**functions for BSDF**	header	22	
BUMP MAP	10	bumpmap_pars_fragment	**functions bumpmap**	header	vBumpMapUv	bumpMap
CLEAR COAT	11	clearcoat_pars_fragment	parameters (uniforms)	header		...
	12	clearcoat_normal_fragment_begin	defines variable clearcoatNormal	main()		
	13	clearcoat_normal_fragment_maps	maps normal from texture	main()	80, vClearcoatNormalMapUv	
CLIPPING PLANES	14	clipping_planes_pars_vertex	parameters (varying)	header		
	15	clipping_planes_vertex	vClipPosition = - mvPosition.xyz;	main()		
	16	clipping_planes_pars_fragment	parameters (varying and uniforms)	header		clippingPlanes
	17	clipping_planes_fragment	discards clipped pixels	main()		
COLOR	18	color_pars_vertex	parameters (varying)	header		
	19	color_vertex	vertex color (vColor)	main()		
	20	color_pars_fragment	parameters (varying)	header		
	21	color_fragment	diffuseColor *= vColor	main()		
	22	common	**useful defines, structs, functions…**	header		
CUBEMAP REFLECTION	23	cube_uv_reflection_fragment	**functions for cubemap reflection**	header		
DEFAULT	24	default_vertex	basic vertex shader (projects position)			
	25	default_fragment	basic fragment shader (red gl_FragColor)			
	26	defaultnormal_vertex	transformedNormal *= normalMatrix	main()	8	
DISPLACEMENT MAP	27	displacementmap_pars_vertex	parameters (uniforms)	header		...
	28	displacementmap_vertex	displaces position (transformed)	main()	7, 8	
DITHERING	29	dithering_pars_fragment	**function (adjusted color)**	header		
	30	dithering_fragment	gl_FragColor = dithering()	main()		

EMISSIVE MAP	31	emissivemap_pars_fragment	parameters (uniforms)	header		emissiveMap
	32	emissivemap_fragment	totalEmissiveRadiance *= emissiveColor	main()	vEmissiveMapUv	
ENCODINGS	33	encodings_pars_fragment	**functions LinearTosRGB**	header		
	34	encodings_fragment	gl_FragColor = lineartoout	main()		
ENVIRONMENT MAP	35	envmap_pars_vertex	parameters (varyings, uniforms)	header		...
	36	envmap_vertex		main()		
	37	envmap_pars_fragment	parameters (varyings, uniforms)	header		...
	38	envmap_common_pars_fragment	parameters (uniforms)	header		...
	39	envmap_physical_pars_fragment	**functions**	header		
	40	envmap_fragment	envColor, outgoingLight = …	main()		
FOG	41	fog_pars_vertex	parameters (varying)	header		
	42	fog_vertex	vFogDepth = …	main()		
	43	fog_pars_fragment	parameters (varyings, uniforms)	header		...
	44	fog_fragment	mixes final color with fog color	main()		
	45	gradientmap_pars_fragment	**function getGradientIrradiance**	header		
IRIDESCENCE	46	iridescence_pars_fragment	parameters (uniforms)	header		...
	47	iridescence_fragment	**functions for Iridescence**	header		
LIGHTMAP	48	lightmap_pars_fragment	parameters (uniforms)	header		...
	49	lightmap_fragment	reflectedLight += lightMapIrradiance	main()	22, vLightMapUv	
LIGHTS	50	lights_pars_begin	uniforms, **functions**	header	22, 54/56/58/60	...
	51	lights_fragment_begin	computes direct light (reflectedLight)	main()	55/57/59/61	
	52	lights_fragment_end	computes indirect + specular	main()	55/57/59/61	
	53	lights_fragment_maps	irradiance and radiance if ENVMAP	main()		
LAMBERTIAN LIGHTING	54	lights_lambert_pars_fragment	defines RE functions + material	header		
	55	lights_lambert_fragment	creates material	main()		
PHONG LIGHTING	56	lights_phong_pars_fragment	defines RE functions + material	header		
	57	lights_phong_fragment	creates material	main()		
PHYSICAL LIGHTING	58	lights_physical_pars_fragment	defines RE functions + material	header		
	59	lights_physical_fragment	creates material	main()		
TOON LIGHTING	60	lights_toon_pars_fragment	defines RE functions + material	header		
	61	lights_toon_fragment	creates material	main()		

LOG DEPTH BUFFER	62	logdepthbuf_pars_vertex	parameters (varyings, uniforms)	header		logDepthBufFC
	63	logdepthbuf_vertex	changes gl_Position.z	main()		
	64	logdepthbuf_pars_fragment	parameters (varyings, uniforms)	header		...
	65	logdepthbuf_fragment	logarithmic depth value	main()		
MAP	66	map_pars_fragment	uniform texture	header		map
	67	map_fragment	updates diffuse color with texture	main()	vMapUv	
	68	map_particle_pars_fragment	parameters (varyings, uniforms)	header		...
	69	map_particle_fragment	updates diffuse color with texture	main()	vUv	
METALNESS MAP	70	metalnessmap_pars_fragment	uniform texture	header		metalnessMap
	71	metalnessmap_fragment	updates metalness with texture	main()	vMetalnessMapUv	
	72	morphcolor_vertex	morphs vertex color	main()		
	73	morphnormal_vertex	morphs vertex normals	main()		
MORPHING	74	morphtarget_pars_vertex	uniforms, **functions**	header		
	75	morphtarget_vertex	morphs vertex position	main()		
NORMAL	76	normal_pars_vertex	parameters (varying)	header		
	77	normal_vertex	computes vNormal	main()		
	78	normal_pars_fragment	parameters (varying)	header		
	79	normal_fragment_begin	computes normal	main()		
NORMAL MAP	80	normalmap_pars_fragment	normalmap uniforms	header		...
	81	normal_fragment_maps	updates normals with texture	main()		
	82	output_fragment	gl_FragColor = (outgoingLight, a);	main()		
	83	packing	**functions to pack and unpack data**	header		
	84	premultiplied_alpha_fragment	blending	main()		
	85	project_vertex	projects position	main()		
ROUGHNESS MAP	86	roughnessmap_pars_fragment	uniform texture	header		roughnessMap
	87	roughnessmap_fragment	updates roughnessFactor with texture	main()	vRoughnessMapUv	
SHADOW MAP	88	shadowmap_pars_vertex	varyings and uniforms of scene lights	header		...
	89	shadowmap_vertex	fills variables	main()		
	90	shadowmap_pars_fragment	varyings, uniforms, **functions to getShadows**	header	83	
	91	shadowmask_pars_fragment	getShadowMask()	header		

SKINNING	92	skinning_pars_vertex	**function getBoneMatrix**	header		
	93	skinbase_vertex	gets bone matrices	main()		
	94	skinning_vertex	transforms position skinned	main()		
	95	skinnormal_vertex	transforms normals skinned	main()		
SPECULAR MAP	96	specularmap_pars_fragment	uniform texture	header		specularMap
	97	specularmap_fragment	updates specularStrength	main()	vSpecularMapUv	
TONE MAPPING	98	tonemapping_pars_fragment	**functions for ToneMapping**	header		
	99	tonemapping_fragment	tonemapped gl_FragColor	main()		
TRANSMISSION	100	transmission_pars_fragment	**functions for transmission**	header		...
	101	transmission_fragment	updates diffuse with material transmission	main()		
UV'S	102	uv_pars_vertex	parameters (varying)	header		
	103	uv_vertex	transforms UVs	main()		
	104	uv_pars_fragment	parameters (varying)	header		
WORLD POSITION	105	worldpos_vertex	gets worldPosition	main()		

When creating custom materials in Three.js (ShaderMaterial or RawShaderMaterial), you can either program from scratch or use already existing code created by the community, called Shader Chunks. These pieces of code are meant to be included in your GLSL shaders to help you achieve a task. However, there is a lack of documentation on how to use them. Hence, I created this document by reading and inspecting the source code, understanding what each block of code does.

This is a guide on how to use Three.js Shader Chunks, a notation on their main dependencies, objective and content. Keep in mind that this document is only to introduce you to the basics and I highly encourage you to check the source code of the chunks you want to use, in order to better understand what they do and what uniforms they need.

The file name gives you an idea on what they are intended for and where they should be located: in the vertex or fragment shader. If the file also includes the word *pars*, that chunk is meant to be included in the parameter section; otherwise, it goes inside the main function.

To use most of the shader chunks you have to define the corresponding flags (#ifdef) in your application code. To check the *defines* needed, go to the source code. Here are some examples:

ShaderChunks 18-21: Color

Basic shader to update the diffuseColor according to the vertex color.

When you create your ShaderMaterial:

```
const defines = {
    USE_COLOR: true,
  };
```

In your vertex and fragment shader:

```
// color
export const color_vertex = glsl`
  #include <color_pars_vertex> // vColor

  void main() {
    #include <color_vertex> // vColor = color

    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
  }
`;

export const color_fragment = glsl`
  #include <color_pars_fragment>

  void main() {
    vec3 diffuseColor = vec3( 1.0 );
    #include <color_fragment> // diffuse = vColor
    gl_FragColor = vec4(diffuseColor, 1.0);
  }
`;
```

Multiple ShaderChunks: Phong

When it comes to lighting, using already existing equations is easy with built in materials like MeshPhongMaterial. However, when you want to include said lighting to your material, but adding some customization, many problems and dependencies arise. The following code shows the dependencies and steps to follow in order to include Phong lighting to your custom shader. This example can be extrapolated to other lighting equations, like the Lambertian model.

To create your shaderMaterial:

```
const PhongMaterial = new shaderMaterial(
      //uniforms
      THREE.UniformsUtils.merge([
      THREE.UniformsLib.lights,
      {
       emissive: { value: new THREE.Color() },
       specular: { value: new THREE.Color() },
       shininess: { value: 30 },
       diffuse: { value: new THREE.Color() },
       opacity: { value: 1.0 },

       uvTransform: { value: new THREE.Matrix3() },
       uv2Transform: { value: new THREE.Matrix3() },

       u_color: { value: new THREE.Color() },
       },
      ]),
      vertex,
      fragment
);
```

In your vertex and fragment shader:

```
export const vertex = /* glsl */ `
  #define PHONG

  varying vec3 vViewPosition;

  #include <common>
  #include <uv_pars_vertex>
  #include <uv2_pars_vertex>
  #include <normal_pars_vertex>
  #include <shadowmap_pars_vertex>

      void main() {
        #include <uv_vertex>
        #include <uv2_vertex>
        #include <beginnormal_vertex>
        #include <defaultnormal_vertex>
        #include <normal_vertex>
        #include <begin_vertex>
        #include <project_vertex>

        vViewPosition = - mvPosition.xyz;

        #include <worldpos_vertex>
      }
`;
```

```
export const fragment = /* glsl */ `
      #define PHONG

      // phong uniforms
      uniform vec3 diffuse;
      uniform vec3 emissive;
      uniform vec3 specular;
      uniform float shininess;
      uniform float opacity;

      uniform vec3 u_color; // base color

      #include <common>
      #include <packing>
      #include <uv_pars_fragment>
      #include <uv2_pars_fragment>
      #include <lightmap_pars_fragment>
      #include <bsdfs> // BRDF functions
      #include <lights_pars_begin> // get lights info
      #include <normal_pars_fragment>
      #include <lights_phong_pars_fragment> // RE
      #include <bumpmap_pars_fragment>
      #include <shadowmap_pars_fragment>

      void main() {
        vec4 diffuseColor = vec4( diffuse * u_color, opacity );
        // total light
        ReflectedLight reflectedLight = ReflectedLight( vec3( 0.0 ), vec3( 0.0 ), vec3( 0.0 ), vec3( 0.0 ) );
```

```
        vec3 totalEmissiveRadiance = emissive;

        #include <normal_fragment_begin>

        // accumulation
        float specularStrength;
        #include <lights_phong_fragment> // create material
        #include <lights_fragment_begin> // directional light
        #include <lights_fragment_end> // indirect + specular

        // modulation
        vec3 outgoingLight = reflectedLight.directDiffuse + reflectedLight.indirectDiffuse +
      reflectedLight.directSpecular + reflectedLight.indirectSpecular + totalEmissiveRadiance;

      #include <output_fragment> // gl_FragColor
      }
`;
```