UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Interactive visualizations of unstructured oceanographic data

Simen Lund Kirkvik

INF-3990 Master's thesis in Computer Science - February 2023

Til min mor, Beate

Abstract

The newly founded company Oceanbox is creating a novel oceanographic forecasting system to provide oceanography as a service. These services use mathematical models that generate large hydrodynamic data sets as unstructured triangular grids with high-resolution model areas. Oceanbox makes the model results accessible in a web application. New visualizations are needed to accommodate land-masking and large data volumes.

In this thesis, we propose using a *k*-d tree to spatially partition unstructured triangular grids to provide the look-up times needed for interactive visualizations. A *k*-d tree is implemented in F# called FsKDTree. This thesis also describes the implementation of dynamic tiling map layers to visualize current barbs, scalar fields, and particle streams. The current barb layer queries data from the data server with the help of the *k*-d tree and displays it in the browser. Scalar fields and particle streams are implemented using WebGL, which enables the rendering of triangular grids. Stream particle visualization effects are implemented as velocity advection computed on the GPU with textures.

The new visualizations are used in Oceanbox's production systems, and spatial indexing has been integrated into Oceanbox's archive retrieval system. FsKDTree improves tree creation times by up to 4× over the C# equivalent and improves search times up to 13× compared to the .NET C# implementation. Finally, the largest model areas can be viewed with current barbs, scalar fields, and particle stream visualizations at 60 FPS, even for the largest model areas provided by the service.

Acknowledgements

I would like to thank my supervisor, Lars Ailo Bongo, for always making me optimistic after every meeting we had. Even though it all seemed daunting at times. Giving crucial feedback throughout the process, without which would make this project impossible. I would also like to thank Jonas Juselius, for starting this ambitious project and letting me take on these challenges head-on. Only responding with enthusiasm and encouragement to all the problems needing to be tackled.

Thanks to all my close ones for their continual love and support.

Contents

List of Figures

List of Tables

List of Listings

Glossary

***k*-d tree** *k*-dimensional spatial index tree. Data structure for quickly storing spatial information, where searches like nearest neighbor searches can be made.

Atlantis The front-end application of Oceanbox. Here is where the users log in to use their Software-as-a-service. A map is presented to the user, where they can choose model areas to explore, run particle simulations, and more.

Sorcerer The back-end server of Oceanbox serves archive data. All archive requests that need data go through Sorcerer. This is also where performance improvements can be made when fetching data.

List of Abbreviations

CMEMS The Copernicus Marine Environment Monitoring Service

EPSG:3857 Pseudo-Mercator - Spherical Mercator

FVCOM Finite Volume Community Ocean Model

GIS Geographic Information System

GPU Graphics Processing Unit

HPC High Performance Computing

k-NN *k*-nearest neighbor

SaaS Software As A Service

WGS84 World Geodetic System

WMS Web Map Server

WMTS Web Map Tile Server

1 Introduction

The newly founded company Oceanbox[1] is creating a novel oceanographic forecasting system to provide oceanography as a service for commercial and scientific applications. These services are built on mathematical models that generate hydrodynamic data sets in unstructured triangular grids with high-resolution model areas. This thesis describes our approach to creating interactive visualizations for oceanographic data, so consumers can easily interpret it. These visualizations include streamlines, barbs, and particle tracers. These types of applications and visualization can be found in other areas like weather forecasting[2]. Ocean forecasting has been attempted, but there are currently no large commercial enterprises focusing on oceanography-as-a-service. Oceanbox is creating a complete pipeline for running FVCOM simulations in pre-defined regions. These ocean regions hold aquacultures, whose owners can run particle simulations in their browsers, and view ocean forecasts for operational intelligence. The data amounts are large, and therefore not easily searchable within an interactive timeframe.

The hydrodynamic data sets contain an unstructured triangular grid for the structure of the model area and data points that constitute the state of the ocean. The data point types of data are velocity, depth, salt, temperature, and elevation. There are 35 layers, which is the third dimension of the grid. The grids can be significant. For example, our test grid, Napp, is a small model

1. https://oceanbox.io
2. https://windy.com

area, situated around Buksnesfjorden in Nordland, Norway, consisting of 25 136 nodes and the archive file is 3.8 GB. But it only contains 24 hours' worth of data. The Oceanbox service requires both historical and forecasting data forward 3-5 days is a challenge. Oceanography as a service requires representing the data clearly on a map to find value. Larger grids, require more data to be transferred and the gird size, therefore, limits the sizes of model areas that can be accessed in a timely fashion to ensure a good user experience.

Creating an oceanography-as-a-service solution comes with several challenges. The first is visualizations: how to represent the water's velocity and direction in a map visualization, rendering the grid on top of a map, and animating streams. Integrating custom drawing routines into existing map-drawing technologies is yet another challenge, consisting of both the triangular unstructured grid, but also the values within the grid. Finally, particle streams are an intuitive look into the movements of fluid-like substances, and will significantly increase the readability of the simulations. A second challenge is to extract data from the model area within milliseconds, which allows for real-time interactions with the visualized data. Because of the size of the grid, measures must be taken to reduce loading times, both for the servers serving data, and the clients requesting data.

Traditionally, ocean models like ROMS[10], HyCOM[2], and NEMO[9], use structured regular grids that allow for easier processing due to their regularity, as accessing positional data can be done using the known size of the rectangles. Finite Volume Community Ocean Model (FVCOM) is the ocean modeling system used by Oceanbox for ocean simulations, which uses unstructured triangular grids. This is impossible with unstructured triangular grids, so another approach must be used to query positional data fast enough. In spatial database systems, such as PostGIS, spatial indexing is used for geographical data[1]. Examples of spatial indexing data structures are R-trees, quad-trees, k-d trees. There are many examples of maps and services showing weather data for the public. Visualizations of wind bards, scalar fields, and particles, are drawn to allow clients to make decisions based on weather modeling.

The aforementioned set of challenges, lead to the following requirements for the system:

1. A client-side current barb drawing routine for a given velocity vector, and integrating them into a dynamic map tool

2. A way to effectively draw, and color triangular meshes with millions of nodes in the client's browser

3. Create a particle stream visualization given an unstructured triangular

mesh, and the water's velocities of the hour and day chosen

4. A way to access data in an unstructured triangular grid within 150ms through an API

We propose a full-stack system using F# in a purely functional paradigm that solves the above 4/5 requirements. A current barb visualization that leverages the dynamic map tool's tiling to find coordinates within the archives. We add a spatial index to the archive server, which allows it to retrieve data quickly. Together with the current barbs, we create custom dynamic map layers that use WebGL. One for rendering the FVCOM archives and their different scalar fields. And another that advects visual particles on the GPU, for a beautiful dynamic water stream visualization.

From a longer-term perspective, these solutions will be the groundwork for a new service. One that provides forecasting, and interactive analysis of transport particles and sedimentation, all within a user-friendly interface. Oceanography-as-a-service which brings ocean intelligence to your web browser. With ocean industries and activity standing vital to countries and companies, we believe this new oceanography-as-a-service platform will become the backbone of ocean intelligence and operational information.

1.1 Previous approaches

The Copernicus Marine Environment Monitoring Service (CMEMS)[8] is an open data and service provider of hindcast, nowcast, and forecast ocean data. They also provide scalar fields and current visualizations. Windy.com also has a current visualization, but where the data is provided by CMEMS. Ocean transport simulations, however, are not currently available as Software As A Service (SaaS) solutions. Most current solutions are science projects or consultancy companies. For example, Multiconsult, Intertek, and SINTEF offer marine operation consulting, hydrodynamic modeling, and other fields of oceanography expertise. Running the ocean models requires large amounts of computing power, so they are run in High Performance Computing (HPC) environments. This is a complex task. Which is part of the reason this is not a commercially available product. A problem is controlling the computing time of customers. Another is queue times for simulation jobs.

Geographic Information System (GIS) are widely used for spatial data exploration. ArcGIS[3] is such a solution, which provides a map interface. Users can

3. https://www.esri.com/en-us/what-is-gis/overview

include layers that hold any type of data with a locality. They can provide data analysis tools and visualizations to enable a greater understanding of different applications. However, this is a proprietary solution, which does not enable endless customization. And does not meet the requirements of oceanographic analysis and marine operational intelligence, which Oceanbox wishes to provide.

Spatial indexes are standard tools for maps and weather systems, but in the particular context of oceanography, there are few solutions and literature concerning interactive visualizations on the web. Most work has been done on fluid dynamics, and mathematical modeling, which is of utmost necessity to enable this new system by Oceanbox.

Particular[6] by Indreberg, 2021, created a particle simulation system for Serit IT Partner Tromsø. It was a stand-alone application that attempted to solve some of the same problems we are. In particular, Indreberg also used a k-d tree to speed up particle lookups. However, this thesis is not exploring particle simulations. Even though Oceanbox currently provides this as a service for its customers.

Meta-data archiving management tools have also been explored by Lau, in 2022. MdMt[7] keeps track of relationships in large geospatial datasets, like dependencies between archives. This system is not currently used, and instead, an RDMS is used to keep track of archives and their files.

1.2 Proposed solutions

Oceanbox has a pipeline designed and implemented to model the hydrodynamics of the ocean on their HPC cluster. Which stores hourly data in archives, and is fetched by users as they are analyzing the forecasts. Oceanbox's oceanographic systems are projected to produce hundreds of gigabytes of data daily. Users are provided a user interface to interact with these simulations, a web server called Atlantis that server a single page application (SPA). They can select which model area they want to view, and are provided with a dynamic map tool. Once in the map tool, they get widgets from where they can toggle options. Additionally, Oceanbox provides a particle simulation interface. Here users can select release points with parameters, like radius and release intervals, which get sent to the HPC cluster to be simulated. Once the results are in, the user can view the simulation, and step forwards in time to see where the particles end up. This, in conjunction with representations of that ocean's state, gives a clearer picture of what is happening. Here is where our visual contributions come in.

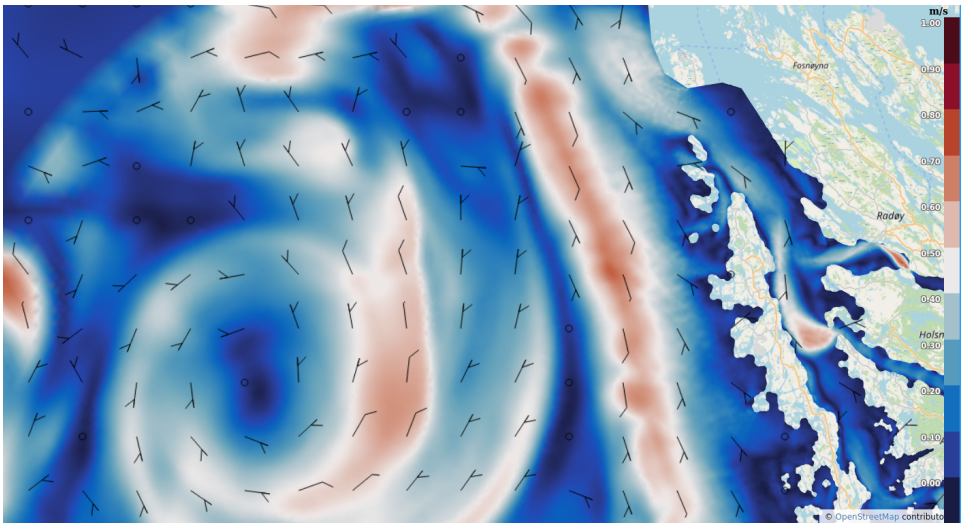

Figure 1.1: The barbs are shown as of 19/10/2022. Showing the coast of Norway in the model area called PO3. Including a color scale to indicate the value associated with the current color shown. Here is the velocity of the water shown.

We propose the design and implementation of a spatial index integrated into Oceanbox's archive delivery system. Sifting this data to extract something as trivial as the velocity of the water in a given area would – with a naive implementation – take too long to serve clients browsing the data in a web browser. To enable quick spatial look-ups into the grid archives, we propose using a *k*-d tree as the spatial index. We implemented a *k*-d tree, especially for this use-case, called FsKDTree. FsKDTree is a static 2-dimensional tree, that does not support adding points after it is made. This enables concurrent users to interactively browse the map overview, and fetch data from the system in real-time while maintaining processing times needed for a responsive user experience, which are between 30ms and 350ms.

Part of the solution is also to create intuitive and visually pleasing visualizations. We provide current barbs, scalar fields, and particle trailing visualizations to the user. This enables clear insight into the movement of the ocean and will help several ocean-born industries in analysis and problem-solving. We provide custom dynamic map layers for rendering our visualizations. Current barbs are drawn on top of the dynamic maps, and their tiling techniques are leveraged to find the correct points to sample the model areas. WebGL renders the model area meshes directly on the user's machines, via their web browsers. We provide a WebGL tiling map layer, which renders the scalar fields of the unstructured triangular grids. This color map is calculated on the GPU to produce a color scale mapped to the grid's values.

We implement spatial indexes and front-end interfaces using F# and Fable, in a single-language full stack, using the .NET platform. Where the front-end uses the OpenLayers[4] open source library, a JavaScript library for interactive maps. The solution is run on an on-premise Kubernetes, hosted by Serit IT Partner Tromsø, the parent company of Oceanbox. The back end includes our *k*-d tree library, called FsKDTree, and contributions to the current Oceanbox solution.

1.3 Summary of results

The solutions presented in this thesis are now a part of the Oceanbox production system and are available for customers with subscriptions. We use spatial indexing to enable a responsive user experience successfully. Current barbs appear with delays between 40 ms and up to 400 ms. They quickly and evenly fill the map, while never appearing outside the model area, successfully land-masking the near-shore triangular grids. Most computers handle running Atlantis in the web browser, allowing for a responsive experience exploring the map and the model area. The largest model areas are rendered without performance issues.

1.4 Outline

The thesis is organized as follows: Chapter 2 gives background on the company, concepts, and new technologies. Chapter 3 describes the design and implementation of the *k*-d tree. Chapter 4 covers OpenLayers and how the barb visualizations are done in the browser, including a custom WebGL OL layer. Chapter 5 describes interpolation techniques, which are needed for simulations, but can be used in visualizations as well. Chapter 6 evaluates the performance of the *k*-d tree and the front end. In Chapter 7 we discuss the different solutions and the trade-offs with our choices. Finally, we conclude and outline future work in Chapter 8.

4. `https://openlayers.org`

2

Computational oceanography

This chapter covers the necessary background to understand the problem space and get insight into the chosen solutions. Firstly, we introduce oceanography and how the modeling is done. Section **??** details the ocean model used, and what kind of data it produces. Then, we create an overview of current oceanographic services, the lack thereof, and how Oceanbox is creating something new.

2.1 Oceanographic modeling

Oceanography is the field of studying the ocean. Oceanographic models are mathematical models that simulate the systems of the ocean. Oceanbox focuses on near-shore hydrodynamic properties, and therefore large spatiotemporal datasets are produced. Near-shore environments are more complex than out in the open sea. A shoreline can have thousands of small islands, requiring high-resolution grids to account for these complexities. Spatiotemporal meaning they have a geographical location and a time stamp. These properties are temperature, salinity, elevation, and velocity. Each property, then, has a location, a value for some time, and at different depths. 34 depth layers to be exact.

When a model and simulation have been created, they can run simulations, such as particle simulations. Particle simulations can be inanimate objects floating, but they can also be simulations of lice or viruses and their floating patterns. For aquaculture industries, this can provide crucial operational information for improving fish health.

Marine operations can be aided through current forecasting and hydrodynamic modelling. Reducing costs by simulating ocean conditions ahead of time. In aquaculture, hydrodynamic modeling is used to help fish farms comply with government regulations. Particle modeling of fish dung can estimate where it will land on the ocean floor, informing the placement of aquaculture infrastructure. These results will typically come as a written report, manually created by oceanographers and other consultants. Oceanbox will bring this directly to your web browser.

A critical use case is fisheries, especially fish farms, that need to know where their fish waste goes, as there are regulations to prevent too much from accumulating in a small area beneath the fish farm due to detrimental ecological effects. Oceanographic simulations forecast velocities, ocean tidings, and more. By running simulations on the fish droppings represented as particles, estimations can be calculated. Then, based on the fish farm's position, the estimated amount of droppings can be placed in the simulations, and advecting it based on the model's current stream data, predictions can be made on where it will end up.

Ocean models involve solving complicated differential equations, like Navier-Stokes equations on three-dimensional numerical grids. This requires a lot of processing power, so these simulations need HPC environments for reasonable execution times. In addition to the expertise required to maintain such a cluster, the simulations can be fickle; resulting in wrong outputs, and having to redo the simulations, which can be worth months of work for the oceanographers. Delays of such magnitude in a commercial environment can lead to additional costs in the millions.

3

Oceanbox's architecture

We outline an overview of Oceanbox's software architecture, to show the scope of this thesis. Finally, the library OpenLayers is expanded upon.

3.0.1 Motivation for oceanbox.io

Oceanographers have long deployed services for creating and running simulations for marine institutions. However, this field has fallen behind in terms of automation, and the many innovations within cloud and data-center technology that enables stream-lined data production. A considerable motivation for Oceanbox is then to streamline the process of defining, starting, running, and interacting with oceanographic simulations by creating the right tool-set to enable more efficient information gathering, and thus strengthening decision-making in marine contexts.

The goal of Oceanbox is to create a web service that oceanographers, and also laymen, can use to browse and analyze the ocean. Oceanbox provides a user interface that makes it easy to create particle simulations. These simulations, however, can become large, and may take up to minutes, hours, and even several days to execute. So a system must be in place to handle the hand-off of simulations.

Visualizations in the browser must be made so that the user can interact with the data by getting a broad picture of the ocean in their model area, or clicking

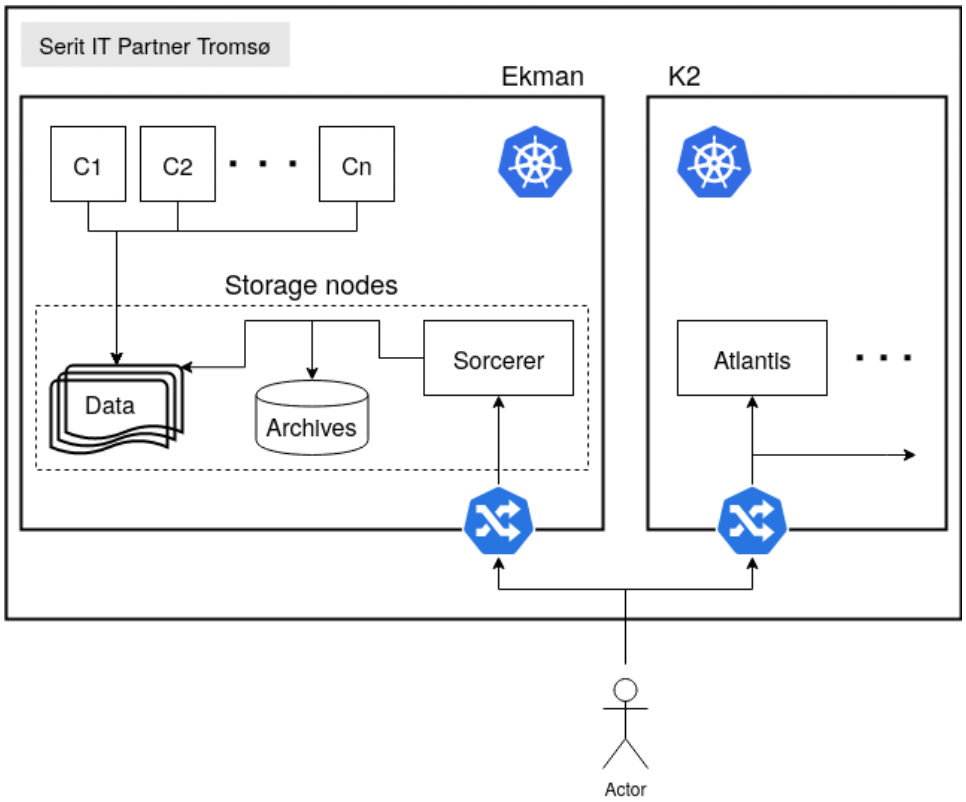

Figure 3.1: A simplified illustration of Oceanbox's architecture. All compute nodes are hosted at Serit IT Partner Tromsø. There are two Kubernetes clusters (Ekman and K2). Clients first contact Atlantis for the front end and are redirected to Sorcerer to fetch Data. The compute nodes share the same filesystem as the Storage nodes. The archive data is stored as files in the Linux file system. The archive database is an index into the filesystem which Sorcerer uses to find the correct files. All fetches pass through an Nginx reverse proxy shown on the border of the clusters.

on a river to see a forecast of its velocity at the different depths. Or tomorrow's elevation, salinity, and more.

Oceanbox was founded in April 2022. They are still in early days of launching their start-up, but they have a functional infrastructure. Able to browse model areas, inspect values, and start particle simulations.

3.0.2 Architecture

The .NET platform is the common foundation for the servers and back-end services, with Fable being used for F# to JavaScript -transpilation and Google Lit

as the reactive web component framework. The front end uses the OpenLayers[1] open source library, a JavaScript library for interactive maps. And WebGL is used for GPU acceleration on the front end. These solutions are used in live production by customers but are in continual change and incremental improvement to improve the overall solution. The solution is run on an on-premise Kubernetes, hosted by Serit IT Partner Tromsø, the parent company of Oceanbox.

Computational demands

The core of Oceanbox's solution is an HPC system where the FVCOM simulations are run. An overview of the architecture can be seen in figure 3.1. The system has 10 computing nodes, with a total of 2000 CPU cores. Two nodes are dedicated as storage nodes and run archive data servers, which are called Sorcerer. Oceanbox currently uses NetCDF as the file format for the FVCOM simulations. In a separate cluster environment, web servers are run. As the user logs on to the web service and selects a model area to view, these servers fetch data from the HPC environment. All subsequent data-specific requests are requested directly to the HPC cluster. When the user has logged in to Atlantis, they receive a URL to Sorcerer, which is used by the front-end code. The data server has direct disk access to all archive data.

When simulations are created, they must be added to the archiving system. This is a database that holds meta-data about the archives. They are stored on disk in a structured manner. Each model area has its own unique ID.

Sorcerer is a service for users to download data. It exposes endpoints for downloading entire grids, though without support for searching within the grids. When users enters the map tool, they download the archive ID's that their user id permits. In the map tool, they select an archive, and with the provided archive ID, Sorcerer can find the correct file from disk.

However, to avoid reading from disk every time, Sorcerer uses a Cache. This cache is a mailbox processor that holds the archive indcies as keys, and the grids as values. Each item has a time to live (TTL) to discard the grids after a given amount of time.

The previous version of Oceanbox did not use spatial indexes. Instead, elements were stored in a dictionary of sets where each node index maps to all it's elements, a neighbor index. Search was implemented by iterating over all nodes, and calculating whether the given point is within any of the triangles

1. `https://openlayers.org`

associated with that node. Real-time look-ups in the grid by concurrent clients was therefore not possible.

Sorcerer is deployed in a Kubernetes HPC cluster called Ekman. Sorcerer is an F# server and has mounted the grid directories to its file system, and therefore has direct file access to simulation output files. Nginx sits in front of Sorcerer as a reverse proxy.

Service delivery

Oceanbox is creating a tool for users to enter a browser that allows them to view and browse the ocean models they have produced. The model areas are split up in to smaller sections, spread across Norway's coast. The archives can holds months, and even years worth of data, with granularity down to the hour. Making the user interface intuitive is crucial to support the user in finding value within these large amounts of data.

The solution is intended to be used by clients in the web-browser, a map library called OpenLayers is used. Which is a open-source JavaScript library for drawing maps that supports many features, like drawing map tiles, vector data, and much more. We extend and leverage OpenLayers to enable the visualizations for the clients. Which are vector barbs, stream lines, and particles.

OpenLayers allows you to easily create a map application using JavaScript. With minimal configuration, you have a map that supports zooming and panning. Through many various models, you can customize your map. With clickable widgets, pop ups, and other UI elements. It has support for different types of map sources, which are where the map information and images come from. In addition to this, it has support for using the graphics card to draw on top of the map.

However, because of WebGL's complexity, OpenLayers has decided to hide some of this complexity through its own API. While this provides features for accessability, it also limits some of the flexibility that is inherent to WebGL. Which is why we chose to create custom components that allows us to meet our requirements.

The map tool is a Single-page Application (SPA) written in F# using Fable Lit. Lit is googles API to create native web-components that many modern browsers support. Web components are HTML tags that can be made with internal state, and reactive functionality that allows them to update on user interaction. Fable is an F# compiler that takes F# code and transpiles it into JavaScript code that can be run in the browser. This allows for a single language in Oceanbox'

entire software stack.

3.1 The NetCDF file format

NetCDF (Network Common Data Format) is a self-describing, scientific file format. Self-describing as in, it contains meta-data that describes its form. Its form is array based storage for effective retrieval. NetCDF 4, the latest major version, supports HDF5, which allows for larger files and multiple unlimited dimensions. Oceanbox produces Terabytes of NetCDF data. Because of the array structure of the archive files, indices can be used to fetch data from disk. This will become essential in the final solution. As it is Sorcerer that has access to the NetCDF files, and when it has spatial indexing capabilities, can quickly find the correct array index based on the coordinates coming from clients.

As an example, the grid is described in the NetCDF files. Each property of the grid is its own array, and when extracting coordinates, each array are read separately, and are zipped together after. However, the grid is indexed, meaning the grid has its own property describing the elements. Elements are themselves described by three indices into the node coordinates, so element 0 has the value: $80, 81, 0$. It points to node 80, 81, and 0. These indices are shared between the x and the y coordinates for each node.

3.2 Finite Volume Community Ocean Model

FVCOM[4] is not the most popular ocean model.

The oceanographers in Oceanbox has found value in FVCOM, and has based the initial stages of the company on it. However, this is not a technological lock-in. More models are to be supported by the platform, so that in the future different models can be run by different clients. There are trade-offs with every model.

The unstructured grid allows for more precision in the model area close to the coast [3]. This effects simulations, as a orthogonal grid has sharp cut-off points. A triangular grid follows the contour of the coast line.

Spatially it will also make a difference. Since the triangles can all have different shapes and sizes, one can increase the resolution in places of interest, and decrease in it homogeneous areas.

The simulations produce NetCDF archives for the FVCOM grids. The FVCOM grid consists of triangles, made up of nodes and elements. A node is a point, and an element is a triangle with three nodes. The nodes have X, Y coordinates, and there is data attributed to both nodes and elements. For example, the speed of an area is stored in the centroid of an element. The centroids are then also coordinates into the grids. Each node, however, holds the estimated salinity for that coordinate.

There is currently no perfect coordinate system to describe the earth. There are many different coordinate systems, all with pros and cons. This is because one would want to work with a flat plane, which is what many coordinate systems do. However, this creates inaccuracies either at the poles, or other extremes. Spherical coordinates would solve this, but the mathematics involved to work with these are quite complicated.

The FVCOM simulations can output data in several projections, but are currently saved in Pseudo-Mercator - Spherical Mercator (EPSG:3857). This is a projection that gives its coordinates in meters. OpenLayers, however, can use different projections, but most commonly World Geodetic System (WGS84) is used, which has coordinates in longitude and latitude.

4

Visualizations

This chapter describes our design and implementation of the interactive Oceanbox visualizations for oceanographic simulations that provide a smooth experience for web browser clients. We provide current barb drawing routines that are imported to a map drawing browser library. The library web tiling features are utilized to query Sorcerer. Tiling functionality is utilized to draw and find appropriate coordinates for current barbs, with the help of the *k*-d tree on the server side. WebGL renders the unstructured triangular grid with the client's GPU, together with particle streams.

4.1 Usage scenario

On logging into the Oceanbox platform, the user is presented with an Atlas view as shown in figure 4.1. This shows what model areas are available to the user. To visualize the model areas on the map the user chooses which model area they wish to explore. The grid must be overlaid on the OpenLayers map which provides functions to move, zoom, and rotate the map. The drawing of the model area must therefore be integrated with OpenLayers.

On entering the map view, the user is zoomed into a pre-defined area of the model grid. They are presented with a sidebar for choosing different map layers and tools they wish to use. A timeline for the model area, how much data is available, and all the 'Drifters' simulations they can access. Finally, a

toolbox is present that has controls for adjusting the current mode. Here is also a 'Streams' switch toggle that toggles the stream's visualization on and off.

Additionally, a timeline is added which shows the timespan in which the service has available data for each model area. It also shows the user's particle simulations, which can be selected and played. This results in a workflow consisting of exploring the model area, starting particle simulations anywhere within said area, waiting for the results, and playing it back as animations over the map.

4.2 Current barbs

The first challenge was to draw barbs that could represent velocity, and encode the speed in a clear visual way. We wanted the barbs to query Sorcerer to show the grids velocity. There are two problems with this: (1) how to place the current barbs on the map, which is the coordinate to be queried to Sorcerer; and (2) how to efficiently look up this coordinate in the grid. The former challenge is solved by tiled web maps, and is described in section 4.2.1. The latter challenge is solved by spatial partitioning, and is described in chapter 5.

To draw the barbs, we took inspiration from wind barbs. We overlay a map interface with current barbs. So, when the user pans over the map, the map automatically, and evenly, fills the model area with current barbs that show the direction and magnitude of ocean's velocity. Zooming requires redrawing the barbs so that the size of the barbs scale with the viewers zoom level.

It is easy to create a single canvas with a barb in the middle that follows the mouse, and based on the distance from the mouse to the center of the barb determines magnitude of the barb. However, placing the barbs on top of a dynamic map application is not trivial. Since, when the user moves the map around, how should you keep track of the barbs and where they should sit? First, there was some thought that went into how one could evenly place barbs on the map, based on view current view extent. Then, an insight to use the web map tiling functionality came, which already fills out the screen evenly.

The barb drawing routine uses arrows described in a file, where each point of the arrow is listed out. The barb has an associated speed for when it should be displayed. The client gives the drawing routine a velocity vector, and the correct barb will be drawn in the associated canvas.

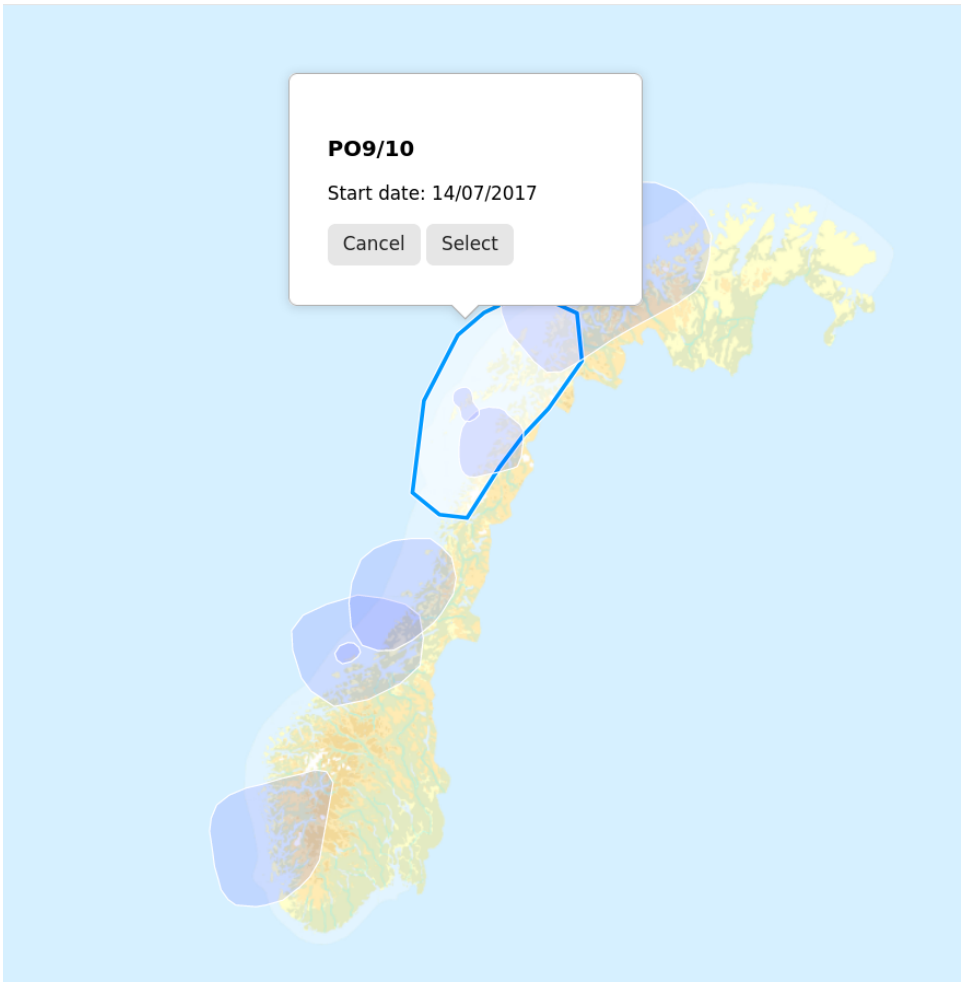

Figure 4.1: A screengrab from the map tool's model selection view. Which is what the user is initially greeted with when logged in. Here we can see that a model area has been selected. A pop-up appears showing some meta data about the grid. Currently, only when the archive started, as in the earliest available data. When confirming the selection, the user will be redirected to the map tool.

Drawing consists of looping through the pre-defined points of the arrow, as shown in the code listing 4.1. We rotate each point around the given angle, then translate is to the final position.

Listing 4.1: Barb drawing routine

```
import {XYZ} from "ol/source";

class BarbTile extends XYZ {
  \*...*\
  drawArrow(ctx, arrow, angle, pos) {
    ctx.beginPath()
    arrow.Points.forEach (p => {
      const s = Math.sin(angle);
      const c = Math.cos(angle);

      const newX = p.X * c - p.Y * s;
      const newY = p.X * s + p.Y * c;

      const x = newX + pos.X;
      const y = newY + pos.Y;
      ctx.lineTo(x, y);
    });
    ctx.stroke();
  }
}
```

4.2.1 Tiled web map

Displaying current barbs, as in a visual barb showing the magnitude and direction of the water it is placed upon, it must find the value of this exact spot. This is not done by the client, even though they have already downloaded the grid. Instead, the client queries a data server that holds the grid. We create a barb layer that extends the behavior of a tiled web map. The data Sorcerer then acts as a Web Map Server (WMS). Taking tile map coordinates and returning data for the tile the client queried. When the user pans around the map, OpenLayers finds the tiles and their tile coordinates which are used to query the data server.

When new data is available for the users to browse, they should be able to view current barbs as soon as they can download the model areas data. This requires efficient spatial indexing, to quickly look up, and find the correct elements for where the data resides as provided by Sorcerer (Chapter 5). This

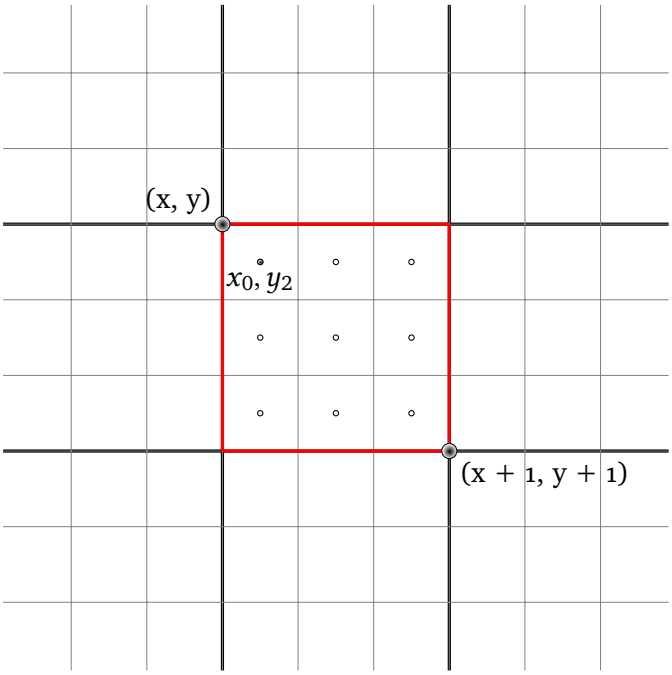

Figure 4.2: Illustration of how the web tiling in OpenLayers is leveraged to create points to fetch velocities for barbs. OpenLayers on the client-side chooses x, y, z tile coordinates which are sent to the server. The server converts the tile coordinates to lat and long coordinates and divides the bounding box into 9 boxes.

should be done in real-time, as no pre-processing is currently performed on the simulation data.

4.2.2 Design

In the selection screen, the available model areas are displayed with a rough outline of their actual positions. The client can click the area of interest, and then be redirected to the map application. Controls are presented to choose what kinds of data the client wants to explore. Whether it should be the velocity of the water, salinity, bathymetry, and more. An example of showing velocity can be seen in figure 1.1.

Showing the current barbs is a choice for the client, as a checkbox in the floating control panel. When enabled the model area will be evenly filled with current barbs, displaying the velocity and direction of the water at the locations from their origins. When the time is incremented, either by the animation, or manually stepping forwards in time, the barb layer has to query the server

again to retrieve the values for the new time step.

The barb tiles are an extension to OpenLayers, with a custom drawing routine for the barbs. The barbs are specified as a JSON file that the server defines. The client will then request the arrow definitions, which it looks up when drawing barbs. It defines the vectors for drawing the arrow, and at what speeds. So when a coordinate is found to have a velocity of $1\,\mathrm{m\,s^{-1}}$, it will draw a specific type of arrow.

Panning or zooming the map will move the client viewport, or extent, of the map, telling OpenLayers to fetch new tiles from the WMS of choice. Users can choose, from our pre-defined selection, what map providers they want to use, as different map tile providers host different types of tiles. They differ stylistically and how detailed they are. The coordinate the client is viewing will correspond to a tile coordinate, which is defined as X, Y, and Z coordinates. The X, Y, Z coordinates of the tile is placed on its north-western corner, and is a geographic coordinate, which can be translated. This is done by the Sorcerer service, and is described in Chapter 5. As tiles are rectangles covering some real geographic space, they can be used as bounding boxes for finding the points from where to sample the current model area. An example of this can be shown in figure 4.2. Which shows the web tiles, and the subsequent partitioning of these to find the barb search coordinates.

This results in a scheme for evenly placing current barbs over the model area. By bootstrapping the tile fetching procedures of web tiling libraries, in this case OpenLayers, the coordinates and positioning of current barbs are found. Additionally, OpenLayers does caching to avoid redundant downloads of web tiles, and since the current barbs are drawn onto a canvas, they are cached for reuse.

The WMS in case the “Sorcerer”, which is a F# server running in the same cluster as the HPC cluster running the simulations. This means that it has data locality, as it can mount the file system that stores the simulation data, and the model areas. “Sorcerer”, then, exposes an API that takes web tile coordinates, an archive id, and a time step, which it uses to fetch velocity data from the correct model area.

4.2.3 Implementation

The barb layer is implemented in JavaScript where it extends a XYZ layer. Which is one OpenLayer class from the assortment of different layers and functionality that comes with OpenLayers. An XYZ layer is a map layers that expects to be pointing at a WMS, which has an API that takes XYZ tile coordinates. This

layer class has a tile loading function, which takes a tile and a URL. The tile is a canvas, and the URL is the WMS the layer should fetch its map images from. However, this function is overwritten in the barb layer, where a fetch is done manually to “Sorcerer”.

Listing 4.2: Tile drawing routine

```
drawArrows(tileVelocities, tile, coords) {
  const tileSize = this.tileGrid.getTileSize(coords);
  const gridSize = tileSize[0] / this.arrowsPerTile;
  const midGrid = gridSize / 2;
  // Create a canvas for this tile
  const ctx = createCanvasContext2D(tileSize[0],
                                    tileSize[1]);

  tileVelocities.forEach((tileRow, i) => {
    tileRow.forEach((vel, j) => {
      const pos = {
        X: midGrid + j * gridSize,
        Y: midGrid + i * gridSize
      };
      const arrowVec = {
        X: vel.X,
        Y: -vel.Y
      };
      const velMag = this.vectorLength(arrowVec);
      const arrowVecN = this.vectorNormalize(arrowVec);
      const angleRad =
            this.arrowAngleRad(arrowVecN) + Math.PI;

      if (velMag < 0.1) {
        this.drawCircle(ctx, pos.X, pos.Y);
      } else {
        const arrow = this.findArrow(velMag);
        this.drawArrow(ctx, arrow, angleRad, pos);
      }
    });
  });
  (tile).setImage(ctx.canvas);
}
```

Instead of Sorcerer returning an image, it returns a set of vectors. These are the velocities of the barbs. The client then does a drawing procedure in each of the HTML5 canvases that OpenLayers produced, drawing the correct barb based

on the magnitude, and in the correct direction. This can be seen in Listing 4.2. An example of this can be seen in figure 1.1. The colored areas of the map show the outline of the model area. It is colored using WebGL which is done with another OpenLayers layer, but is described further in section 4.3. The client can choose the density of barbs, but the default value is 9 barbs per tile.

The web service Atlantis is meant to be used by many clients concurrently. Since different archives and files are read at the same time. Unnecessary file io should be avoided. This is partly ameliorated through a cache agent. There are two agents, one for opening and reading from the archive files stored as NetCDF files on disk, and a cache agent, which holds data structures, and mainly spatial indices, which are also stored to disk when stale.

4.3 Model area mesh with WebGL

This section describes the use of WebGL to draw the model area, and enable efficient interactive visualizations by solving the challenge of the data amounts involved in oceanographic simulations, and techniques used to overcome this. This allows us to place the whole model area on the grid, without any pre-processing. Drawing and coloring the grid is done by the GPU, which makes the rendering time negligible, but fetching times are cause of some delay. The FVCOM unstructured triangular grid corresponds well to the grids normally drawn by GPUs, enabling the use of well known optimizations.

4.3.1 Design

The client downloads the grid in its entirety. But because of the size of the grids, they are stored on the clients machine using the IndexedDB browser API. When the user starts the web application, and chooses a model to view, the browsers database is checked against the archive id, and the cached version of the grid is used.

OpenLayers handles map visualizations and navigation, and already has layers using the GPU to render objects overlaying the map, these features are used and extended to fit the needs of Oceanbox. OpenLayers supports WebGL layers, so objects can be drawn as sprites at given coordinates on the map. The GPU is very efficient at drawing thousands of such objects at the same time. However, there are no pre-defined layers that allows including the whole mesh (FVCOM grid) to be drawn. The current largest grid, called ‘LT3’, has over 1 million nodes in its mesh. Therefore, we create a new layer by extending the functionality of OpenLayers. However, drawing a triangular grid using the CPU

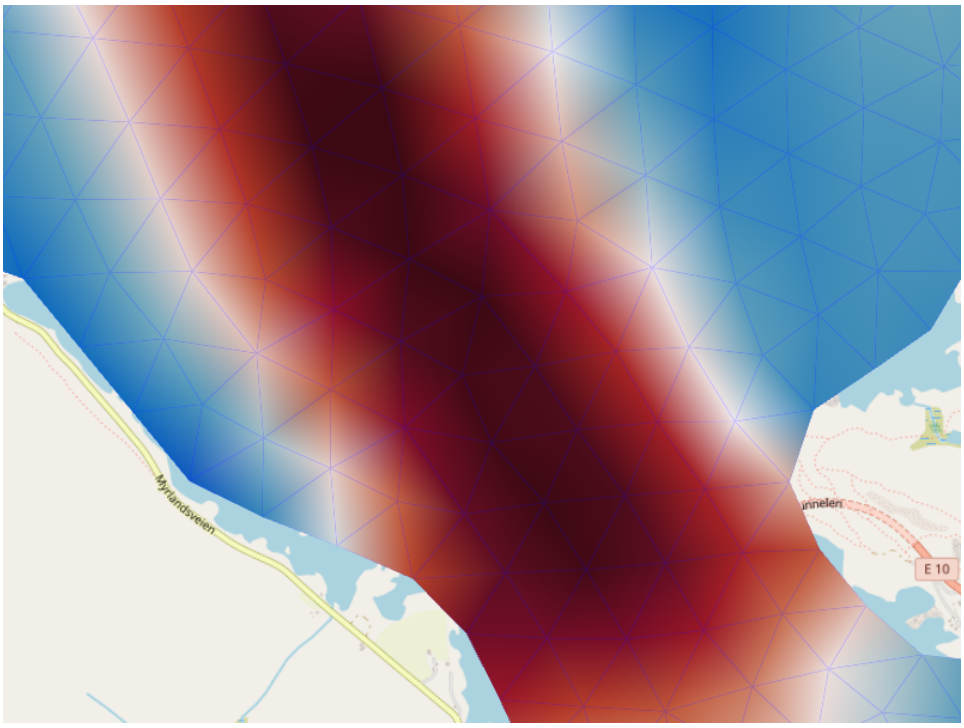

Figure 4.3: Close-up of model grid with wireframe rendering turned on that shows the triangles of the grid and how the coloring is done. Since each nodes gets a value, the color is interpolated by the graphics card between the three corners of the triangle.

is unfeasible.

WebGL is an API for the browser that interfaces with the users graphics card (GPU). So, we can utilize the GPU's triangle rasterization technologies to effectively draw triangle meshes in real time, and therefore provide the responsive interactions with the scientific model that are integral in the Oceanbox service. As OpenLayers is a map drawing library, we use a custom layer that overlay the model drawing on top of the map, even while the client is navigating the map by zooming, panning, and even rotating. This custom layer uses WebGL to draw and color the grid.

Colors

We want to display the information stored in the model area. When looking at the grid, the user can choose what properties they want to see. These are: salt, elevation, bathymetry, temperature, and velocity. Since this information is stored either on the grid's nodes, or in the elements, we can color the elements

based on the values they hold. Which is to say, with velocity as an example, we want to show fast regions of the ocean red, and calm ones blue. However, we want different colors based on the properties being visualized. We can then create color maps, which are pre-defined color palettes. Color maps are calculated on the GPU using the values of the grid. Which means that the grid can be drawn and colored based on the state of the ocean at the given point of time.

Users should be able to tweak the color palette. Different properties can vastly different value ranges. Changes in the color properties should update the model instantly, and a color range should be present to indicate what the colors represent.

The properties must then be fetched from the Oceanbox servers as the client views a given time step.

OpenLayers

As the custom components are written in JavaScript to interface with OpenLayers, we need a way for the F# code to interface with our new modules. Bindings are created in F# as a module that can be imported into F# code. This is done in the style of Feliz[1]. This lets us get typed JavaScript code, that – once transpiled – our F# code gets access to.

Wireframe

To view the grid, we need a wireframe view of the model areas. This is disabled by default, and can be switched on by the user. There is no native functionality in WebGL to enable a wireframe view, so it must be calculated manually by the GPU. An approach is to send barycentric coordinates together with the grid when rendering it. However, these must be correctly aligned with the vertices, so the grid must therefore be unindexed. An alternative would be to not include the barycentric coordinates, and calculate the length from the edge of the triangles, but this reduces rendering quality. We therefore went for this approach.

However, because of the changes between wireframe and normal rendering, the wireframe is created as a separate layer. This makes it optional to add the wireframe rendering to the map application. Another approach would be to bake the wireframe functionality into the original layer, but this would increase

1. `https://zaid-ajaj.github.io/Feliz/`

the complexity within the shader code. One could send an attribute to the GPU whether wireframe was enabled, or change the shader program used with the view. For simplicity, they were split into separate layers overlayed on top of each other.

Summary

We create functions that returns the new WebGL layer, and with arguments that lets you specify the grid and the inital values for the grid. One for viewing props, and another layer that renders the wireframe of the model area. The class also exposes methods for updating the grid's properties for when the time changes, the user adjusts the opacity of the grid, or changes the color palette to one of the many made available to them.

4.3.2 Implementation

OpenLayers integration

The WebGL layer is created in JavaScript as a class that extends the OpenLayers Layer class. This allows us to add this layer to our own map. A custom renderer is created for the overrideable method that gives the layer its renderer. By creating a custom renderer, we choose how the layer looks, and what should be on it. The custom renderer is also an extension of a pre-existing class within OpenLayers, the WebGLLayerRenderer.

In the WebGLLayerRenderer class, the shaders, array buffers, attributes, and uniforms can be manually created with the help of OpenLayers WebGL helper functions. However, OpenLayers exposes the utilities they use internally for their WebGL layers. These are used when possible. The renderer has two central methods that can be overridden, which are `prepareFrameInternal` and `renderFrame`. Most importantly, in `renderFrame`, since this custom layer has been added to a map, from where the user moves their view around the world, it provides the projection matrix which translates coordinates to screen pixels. By uploading the projection matrix to the GPU, and multiplying every node in the grid, we move the model grid to the correct position on the map.

Color palette

The color palette calculations are done by uploading the selected color palette to the GPU as an attribute. The vertex shader is shown in 4.3, where some

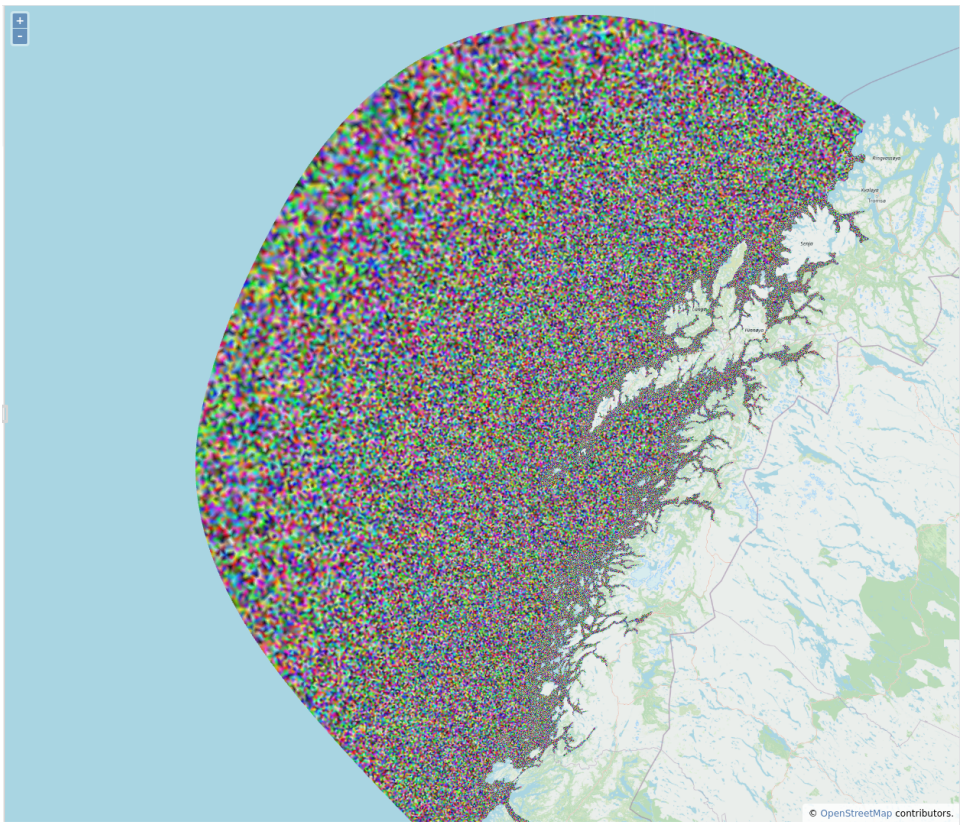

Figure 4.4: An early rendition of drawing the grid. Random colors were used to draw every triangle. The model area is called 'LT3', and is located on the coast of Norway, encompassing Lofoten.

variable declarations are omitted. The color palette is therefore shared between every vertex in the vertex shader. Together with the vertices, we upload the properties, which is the chosen grid property that the client wants to view. Then, for every vertex in the grid, we also get the node's property. With the property's value, we can normalize it to index the correct color in the color palette. The color is passed on to the fragment shader, which is where the triangles are colorized.

Listing 4.3: Shader for model area

```
attribute vec2 a_position;
attribute float a_prop;

uniform mat4 u_projectionMatrix;
uniform vec3 u_palette[256];
uniform float u_paletteLength;
uniform float u_paletteRange[2];
uniform float u_opacity;

varying vec4 v_color;

void main() {
  gl_Position = u_projectionMatrix
                    * vec4(a_position, 0.0, 1.0);
  float v = a_prop;
  if (v < u_paletteRange[0]) {
    v = u_paletteRange[0];
  } else if (v > u_paletteRange[1]) {
    v = u_paletteRange[1];
  }
  float colorRescale = ((u_paletteLength - 1.0)
                        / (u_paletteRange[1]
                            - u_paletteRange[0]));
  int idx = int((v - u_paletteRange[0]) * colorRescale);
  v_color = vec4(u_palette[idx][0], u_palette[idx][1],
                 u_palette[idx][2], u_opacity);
}
```

Wireframe rendering

By giving each vertex in our grid a barycentric coordinate[5], we can let the GPU interpolate between the corners, and thus see how far from the edge of each triangle we are. By knowing how far we are from the edge of the

triangle, we can decide what color it should have. In the case of a wireframe, we can make the middle of each triangle transparent, and give the colors to the edges.

To create barycentric coordinates, we take the indexed grid and unindex it. This can be done by iterating over every element and expanding the elements out into a new unindexed vertex array. This creates an array with twice as many entries as the original index array.

After we have a flat unindexed vertex array, we can create the barycentric coordinates. This is done by mapping each vertex coordinate to its index in the array modulo three. This gives us the index into another array of barycentric coordinates, which are uploaded to the GPU as an uniform. Meaning, every vertex shares the same three barycentric coordinates, but each pair of vertex coordinates get assigned an barycentric index.

In the fragment shader, with the barycentric coordinate, we can calculate the pixels opacity, based on its distance from the triangles edge.

4.4 Particle streams

We designed the particle streams to visualize velocities that are inspired by weather forecasts[2], which often have wind particles as a visualization tool. The particle streams were implemented as an independent application in JavaScript by Daniel Stødle of NORCE Norwegian Research Centre and were integrated into the map tool afterward by the thesis Author. Not to be confused with particle transport models, where in visualizations, actual speeds are not of primary concern, only the aesthetics. However, the land-masking problem requires us to avoid getting ocean particles on land. By uploading the vector fields to GPU textures, velocities can be sampled to move only the particles where there is motion.

4.4.1 Design

Particle streams can be visualized by releasing thousands of fluorescent particles into a stream of water and then tracking the particles to see the water streams. However, keeping track of these particles is computationally expensive. Therefore, the visualization and computation to render the animation is done on the GPU.

2. `https://windy.com`

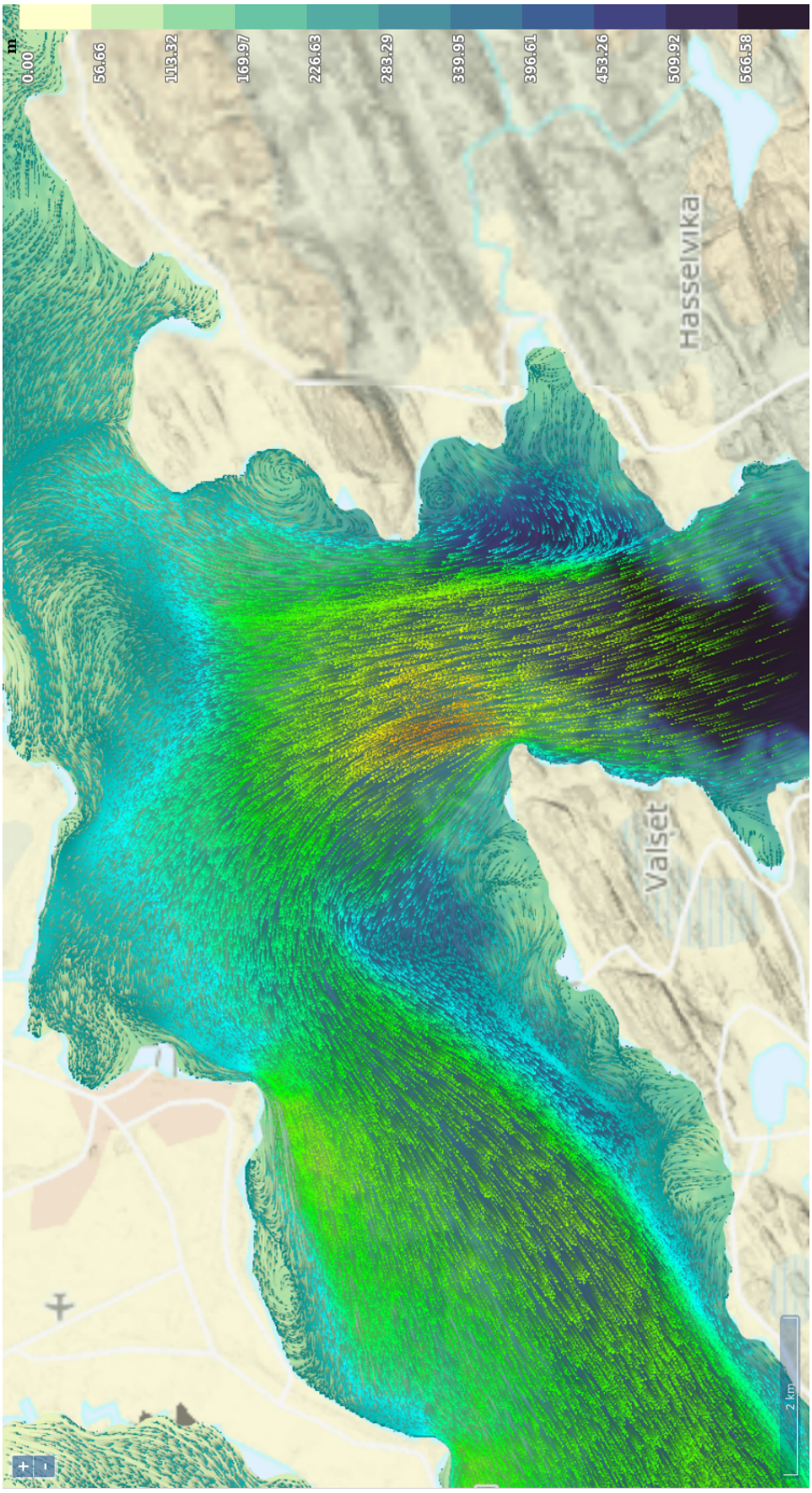

Figure 4.5: An image of particle streams visualizations in Atlantis showing Saltstraumen, a famous tidal current in Northern Norway near Bodø.

The particles should be beautiful and have a trailing tail that fades behind the particles as they move. They should follow the properties of the water they happen to have landed in, and therefore give a general idea for the user of the velocity of the water. These visualizations are therefore mostly cosmetic.

The streams layer is created as an extension to OpenLayers, which allows OpenLayers users to add these graphics to their maps. What must be supplied is the model, which consists of vertices and indices, and the velocity for each element in the model. Then, a canvas will be overlaid on top of the user's map, following the user's movements.

Since the user has already selected a model area, the grid is already fetched from 'Sorcerer' and loaded into memory. On the initial load, the streams layer is created and initialized but set to be invisible. In the toolbox, the user can toggle the streams layer. Which will fill the model area with particles that will start moving. When the user steps forward in time, the particle system must have its vector field updated, so that it can create a new particle simulation. This initiates a fetch from Sorcerer, which returns the velocities for the model in a new timestep.

Particle stream animation is done in 4 steps:

1. Generate particles in random locations

2. Look up the particles on their positions and the underlying vector field, and advect them

3. Reset particles after a 'lifetime'

4. Fade the previous positions and draw the new position on top

We create interfaces both for F# and JavaScript. The JavaScript ones wrap the streams application in OpenLayers, which allows us to view the particles on a map, and which allows other programmers to utilize this layer in their map applications. The F# bindings are made to be used in our application. An interface to update the particle stream layer's velocities is provided.

4.4.2 Implementation

We create an OpenLayers class that extends the OpenLayers Layer class, which has its own renderer, which is the Streams renderer. This custom renderer calls the streams application and gives the application its canvas instance and the frame extent from where the streams can sync up with the map view. Finally,

we wrote F# bindings for the map tool front end. Which allows us to create the streams layer and add it to the map tools map.

The particle streams are implemented as a custom OpenLayers layer. This layer gives us a canvas to draw on, and an extent that tells us the world space coordinates, based on where on the map the Client is currently viewing. When the layer is initialized and the canvas is created, it can be handed off to the streams layer application, together with the FVCOM grid and the current velocity. For the sake of brevity, mainly the vector field computation will be discussed.

With the vertices, indices, canvas, and velocities as input, the streams layer will draw onto its own canvas in an independent animation loop. We create a mesh out of the model grid, the same as the model area grid layer. Then, a vector field is created from the model mesh, but the velocities are moved from the centroids of the elements, to the nodes of the mesh. This is done by averaging the velocities by the contributing triangles, since many triangles can be connected to each node. The vector field consists of the model grid, and a texture which holds the velocities.

Particles are created when the stream application is initialized and placed randomly within the bounding box of the grid. The particles are also transcribed into a texture, where their properties are written as pixels. Each particle gets two floating point pixels to describe its position in x, y, their remaining lifetime in t, the velocities magnitude in v, and the starting x, y, t in sx, sy, st. Which are the starting position when the lifetime of the particle expires.

With the vector field and particles initialized, the animation can begin. It loops at 60 frames per second. Each frame, the particles are computed and moved based on their positions, including both the particle texture, and the vector field texture. Then, the graphics card does a texture look-up to find the current particle. It checks to see if the lifetime of the particle has passed or not. If not, it will do another look-up in the vector field texture, and move the particle based on the found velocity. This is written back into the particle texture.

When rendering the particles, points are used as the geometries for the rendering. There are allocated the same number of point geometries as particles in the texture. The particle texture is uploaded together with the points. Then, in the vertex shader, each point geometry looks up their respective value in the particle texture, which there are two pixels per particle, as described earlier. The point can then be placed in the x, y coordinates described by the texture. The color can also be found by looking at the particles velocity in conjunction with the chosen color gradient. To create a fading effect, the lifetime of the particle decides the opacity of the point geometry, fading away as its lifetime

approaches its maximum.

5

Spatial indexing: Finding things fast

In this chapter, we describe how spatial indexing allows for efficient data queries into an unstructured triangular grid. We use a *k*-d tree to store the grid information of FVCOM files to index them based on spatial coordinates. Given a coordinate, the tree finds the correct index in the FVCOM archive, letting us retrieve data without searching further. The “Sorcerer” service uses the library to load the grid and exposes an API to let clients fetch and search for model data while browsing the data in a web browser.

5.1 The k-d tree

A *k*-d tree is a *k* dimensional tree, where *k* is *k*-dimensional space. It is a space-partitioning data structure mainly used for holding points. Specifically, a *k*-d tree bisects a list of points, constructing a tree. On construction, it alternates the axis on what it bisects. That is, sorting first on the x-axis, then the y-axis, in a typical example where we want to partition points in space. Initially, our implementation only supports two dimensions, so it is a static 2D tree. We can search some space using either range search or nearest neighbor search. Range search takes a bounding box, and returns all elements stored in the tree within that box. The nearest neighbor search takes a coordinate, and returns –

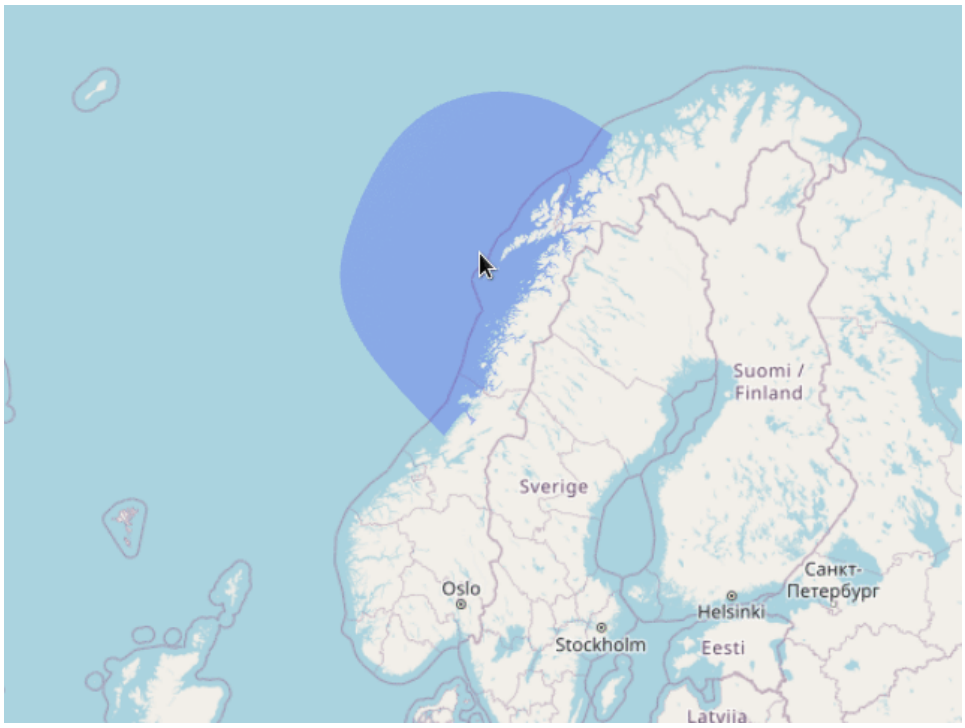

Figure 5.1: Zoomed out view of an FVCOM grid

in order – the closest items in the tree to the given point.

Two trees are created: a node index tree, where the FVCOM node coordinates are the leaves making up the tree; and an element index tree, where the coordinate of the element's centroid is used. Each leaf holds the corresponding index in the grid archive. The library has two functions: range search and nearest neighbor search. The tree can index any set of points. Here we index the triangular unstructured grids, using their real geographic coordinates as the points, and the index back into the NetCDF file as the data.

The user can then use the tree to for example search for the velocity at some area within the grid. Doing a range search will result in a list of leaf nodes satisfying the bounds of the area supplied. With the indices returned, the user can then index the NetCDF archive and read the velocities of the points found. A search has successfully been done.

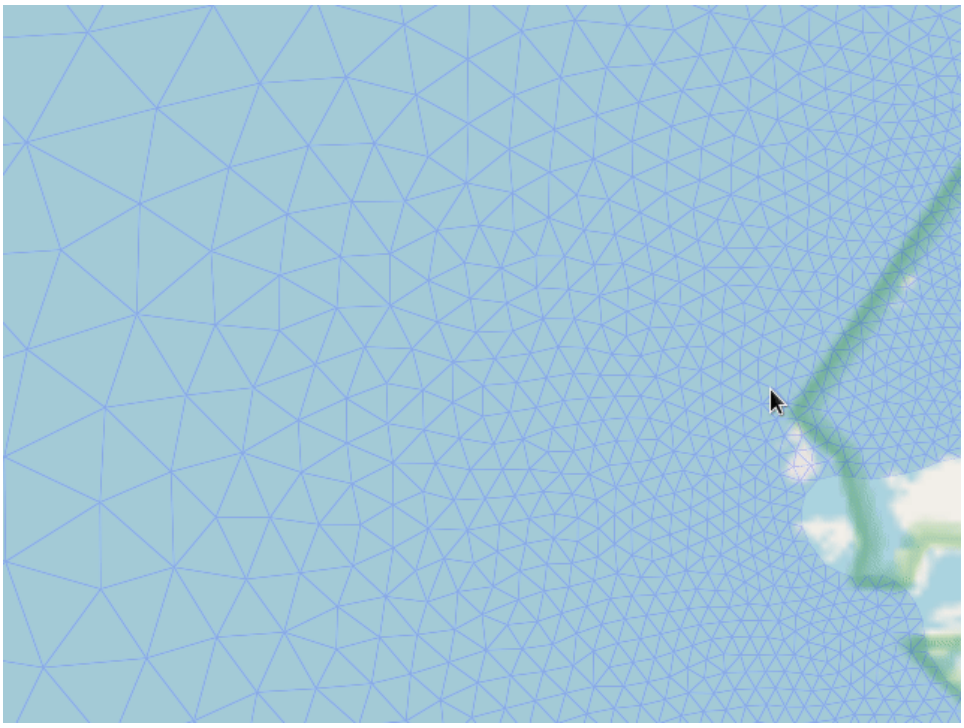

Figure 5.2: Close up view of the FVCOM grid shown in 5.1

5.2 Design

The FVCOM simulations produce NetCDF archives, which Sorcerer serves the clients browsing archives in the map tool. However, no searching endpoints were previously exposed in Sorcerer. With the *k*-d tree, Sorcerer lets users search the archives. Like in the case of current barbs, where we want to sample grid velocities.

An example grid is a model area called “Napp”, which has 25136 nodes, and then 48332 elements.

There is no correlation between elements in the grid and their coordinates. So to look up a point within the model area without a spatial index is not trivial. In an orthogonal grid, binary search can be used since the grid sizes are known, and the coordinates are therefore sorted. This is not possible with an irregular triangular grid, as traversing the grid would not result in a straight path (as shown in Figure 5.2.)

To enable real-time look-up in the unstructured triangular grid we use the *k*-d tree in Sorcerer. Two search functions are given for the tree: nearest neighbor search, and range search. The *k*-nearest neighbor (k-NN) search

takes a coordinate and returns the closest point it finds. Range search takes a bounding box, and returns all points within. Each leaf in the tree has a position and associated data. The data is generic, but in our use case the indices into the neighbor index previously described is stored.

5.2.1 Barb look-ups

To find the current barb values, Sorcerer uses the *k*-d to do a search in the archive. Since the barbs needs the speed of their position, the search is done on the elements (triangles) of the grids. A nearest neighbor search is performed for each coordinate found within the bounding box of the tile. When the index of the element is found, a check is done to see if the barb's point is inside the element. This, in essence, is how the land masking for the barb visualization is done.

When users enable current barbs, the layer will send an array of coordinates for Sorcerer to look up. Together with the coordinates, the client will include what archive they are looking at, how many barbs they want, and the current time step. This information is used by Sorcerer to find the correct archive, and find the information the client is looking for. Then a list of velocities is returned to the client, which they use to draw current barbs.

The current barb search endpoint uses nearest neighbor search, and will therefore always return a value, even if it misses an element. To avoid this, we must additionally test if the search point is within the element the search returned.

5.2.2 Caching

Multiple users use the same *k*-d tree. So multiple model areas, and their respective grids, can be queried simultaneously. The trees therefore must be loaded only on-demand, and discarded when deemed stale. The indices should then be cached to avoid recreating the trees. Additionally, since the grids never change, the indices can be serialized to binary, and written to disk. Then, when the cache misses when a user selects an archive, Sorcerer can read the trees from the disk.

Since users can query specific points for any of the data types, and they are stored on either the nodes or the elements, two types of spatial indices need to be made. They are either node indices or element indices. We, therefore, expose separate API endpoints for locating properties stored on nodes or elements.

5.3 Implementation

This section describes the implementation of a *k*-d tree in F#, why this spatial index was chosen, and the motivation for implementing a new F# library.

5.3.1 The k-d tree

A server that has mounted the file system with the model data (NetCDF files) reads at start-up the grid coordinates. The array of points is given to the *k*-d tree module, which constructs a tree. The points are sorted in turn by each of the *k* axes and split into two halves. Each half is then sorted on the next axis and split again, and so on. This continues until the length is smaller or equals the leaf length specified. So the *k*-d tree is a tree of arrays. Each node holds the coordinates of the mid-point for the previous sorts. This point is used to navigate the tree later in search.

5.3.2 Location endpoints

Sorcerer is the name of the service close to the data. It creates tree indices of the grid data. It exposes end-points for searching for nodes, elements, or a Web Map Tile Server (WMTS) type search, as described in Chapter 4. Two index tree are created with the *k*-d trees, and stored for each model grid. An element tree, for the triangles, and a node tree.

Finding the nearest node is simple. As nodes are just points, a k-NN search on the FVCOM grid can be done, and the nearest neighbor will be the node. Finding the element, however, requires figuring out whether or not the coordinate is inside the element that is found.

Firstly, to do a k-NN search on the elements, a new set of points must be generated, namely the centroids of each triangle. These are calculated by taking the three nodes and averaging their position vectors. These points are stored in the element tree. On look-up, the given point is then compared to the centroids of the triangles, but to check whether the point is within an element, there needs to be a check whether it is inside any of the triangles. Because of the constraints of the triangles in the FVCOM grid, there are edge cases in the nearest neighbor, and within triangle -tests. There is no guarantee that a point within one triangle, but that is on the edge, or near one of its corners, will actually be closer to the center of a neighboring triangle. This can be due to early precision errors, as single-point precision was initially used, and later switched to double precision. However, to ameliorate this, a more comprehensive test is done; when searching for an element within the grid,

all surrounding elements are also tested. Because of the neighbor index, the neighboring elements of the missed element can be used. So, a k-NN search, followed by testing if it is inside that element, can fail. Therefore, in that case, all surrounding elements are fetched and sequentially tested to determine if they contain the point. This enables clients to find specific elements on the map, and they are given an index back. This index can be used to search for data contained in the grid. Like the velocity in the area of the element. This will also be used to sample the grid for information, enabling visualizations like barbs, and point-clicking to look up statistics about the grid. The k-d tree is also used in simulations, as one needs to look up information and velocities from the grid when advecting particles.

To retrieve the velocities of barbs, a look-up is done into the model area. This means that the point found for every barb is searched for in the triangle mesh, to find what element the barbs are within. When the element is found, it can be read from the NetCDF archive.

Sorcerer converts the tile coordinates to UTM33, which the FVCOM grid is stored in. This is done with the formulas

$$lon = \frac{x}{2^z} \times 360 - 180$$

$$lat = \arctan\left(\sinh(\pi - \frac{y}{2^z} \times 2\pi)\right) \times \frac{180}{\pi}$$

Which gives the coordinates of the north-western corner of the web tile. Adding one x gives the east-ward tile, and y the one below. This gives you two coordinates that are the bounding box of the current tile you are looking at. Finally, the tile is divided into n sub-boxes, which can be specified by the client, as shown in 4.2. This is done by dividing the tile into however many barbs the client wants, stepping over the tile with the length given by the barb amount, and finding the middle of each sub-box.

The indices of elements are stored in the k-d tree together with the position of either the element's centroid or the nodes of each element. So depending on the information being searched for, different trees must be used. As mentioned earlier, the velocity of the model area is stored in the elements, with the placement of the velocity being in the centroid of the element. A k-NN search is done against the grid's elements, and an index is given if the barb's point is within any element in the grid. If the search misses, no value is returned.

Additionally, when searching for an element, which again, is the centroid of a triangle, the nearest neighbor search can miss in edge cases. When search points hit close to the border between two triangles, the k-d tree will find the centroid of one element, but fail when testing whether it is inside it. This

results in missing barbs, causing holes where there should be none. To fix this, we perform a double search for every barb. Using the neighbor index, we can look up the neighbors of the element we find. We then perform the triangle hit test for all the surrounding elements. Then, these elements are iterated over, and the element index of the first elements the search point falls within is returned.

When the element indices for the given tile have been found, the u, v of the element can be read from the FVCOM archive on the given time step. The reading operation is given to a "DatasetAgent", which is a mailbox processor that holds the file handles to the different archives available. As concurrent clients can view the same archives at a time, there must be a way to handle file io efficiently.

6 Evaluation

The two most important questions are if the static *k*-d tree implementation was worth it, and how fast the look-up times are. Grid property download times are also evaluated. We evaluated the solutions using benchmarks run on the thesis authors personal computer, and the production system by requesting services directly.

6.1 FsKDTree performance

In this section, we compare the performance of the C# *k*-d tree implementation (KdTree[1]) against our new F# implementation. Tree creation and nearest neighbor searches are critical paths in Oceanbox's stack, and we will therefore evaluate these two functions.

6.1.1 Experiment setup

The experiments are run on a personal laptop. The hardware specifications are in Table 6.1. The data used for the experiments is the largest FVCOM grid available at Oceanbox: 'LT3'. It has 1 000 083 nodes and 1 930 679 elements. The vertices and indices comprise around 30 mb. The archive's simulation date

1. `https://github.com/codeandcats/KdTree`

Host	HP 8549
CPU	Intel Core i5-8265U (8) @ 3.900GHz
RAM	16GB 2667MHz DDR_4
Harddrive	SAMSUNG 256GB NVMe SSD w/ 1300.0/3000.0 Mbps W/R
OS	NixOS 22.11.2345.af96094e9b8 (Raccoon) x86_64
Kernel	Linux 5.15.90

Table 6.1: Hardware and software specifications of the author's personal laptop. Note: hard drive performance is taken from productz.com - Samsung PM981

.NET SDK	6.0.403
.NET Runtime	6.0.1122.52304
BenchmarkDotNet	v0.13.2

Table 6.2: Software versions used in benchmarking

start is 2017/07/14 00:00, and it has time steps of 10 minutes each, lasting for 9 hours, totaling 72 steps. These 9 hours culminate in a total of 13GB of data.

The .NET platform is used to run the code. The versions used can be seen in table 6.2.

The experiments were run using 'BenchmarkDotNet'[2]. To run benchmarks, one can implement a method for a BenchmarkDotNet class. Annotating the method as a benchmark, the library will methodically run the method many times to evaluate the algorithm properly. Data is prepared outside the method, and only the functions of interest are called within it. The framework will therefore decide the start and end -times of the function call.

6.1.2 Creation

We want to measure tree creation times, for both grid nodes and elements. That is, when the grids' points have been loaded into memory, and added to the *k*-d tree, how long does this take? Even though new model areas are not created often, and are often done once and written to disk, these can be deleted, and must therefore be created a-new. This happens on demand, when a user selects a model area, and can block requests. Reducing any user experience delays will therefore be desirable.

This can be measured by timing the creation of the tree. .NET NetCDF SDKs

2. `https://benchmarkdotnet.org/`

Method	Mean	Memory Allocated
C# Node KdTree (N=33)	4.165 s	83.93 MB
C# Elem KdTree (N=15)	9.637 s	294.61 MB
F# Node FsKdTree (N=13)	1.068 s	774.65 MB
F# Elem FsKdTree (N=15)	2.281 s	1731.43 MB

Table 6.3: *k*-d tree creation benchmarking results on 'LT3'. A 13 GB FVCOM archive with 1 000 083 nodes.

are used to read data from disk. When the arrays are read into memory, we can time the data transformation into the right format, and call the creation function. Node vertices do not need any transformation. Elements however must be converted from centroids into nodal coordinates. Going through every element we read its vertices, and calculate the centroid which is loaded into the trees.

FsKDTree has a faster construction time than the C# *k*-d tree. The results of the benchmarks can be viewed in Table 6.3. We see that the node tree construction takes less time in both cases than the element tree, as expected. The F# node tree takes 1 second, and the C# tree takes 4.1 seconds. The element tree takes 2.3 and 9.6 seconds for the F# and the C# trees, respectively.

We can see that the F# tree's creation time is around 4× faster than the C# tree. The F# tree takes a whole list which it iterates over to construct the tree. While it does array splitting, which might allocate objects, this only happens for as many leaves as there are. The default leaf array sizes are 64 elements. The C# tree, on the other hand, creates a node object for every point inserted into the tree. This means memory allocation for every point inserted.

FsKDTree is also static, as one cannot add new points to the tree after creation. The tree must therefore not handle keeping itself balanced, which reduces code complexity and logic within the creation code. The C# tree supports both *k* dimensions, and adding points to the tree. Though, it does not balance itself automatically on point additions, as this must be done manually after creation.

6.1.3 Search

Search functions are used by Sorcerer when users request location-specific information from the grids. When clients activate barbs visualization, they send a request per tile in their viewport. The tile images are 512×512 large, so there are a minimum of 8 tiles on the screen at all times. However, these images can be scaled down by the clients. Here we assume the minimum case.

Method	Mean	Memory Allocated
C# node k-NN search (N=28)	260.363 µs	206.344 kB
C# elem k-NN search (N=33)	274.780 µs	216.544 kB
F# node k-NN search (N=14)	20.583 µs	11.584 kB
F# elem k-NN search (N=14)	20.351 µs	10.768 kB

Table 6.4: *k*-d tree searching benchmark results on the large 'LT3' grid.

Sorcerer subdivides each tile at default into 9 coordinates, this becomes 72 searches into the grid to find the closest element index. The performance of the search will therefore have an effect on the clients. We want to see the difference between the C# tree and our new F# tree.

We use the same hardware and experimentation setup as in the previous section. We measure the time to take a pre-built tree and run k-NN search on both trees. We include both node tree searches and element tree searches. The searches use the same point for both the element and the node trees. When the functions return the point, the benchmark ends.

The results of the search benchmarks can be seen in table 6.4. We can see there are negligible differences between node and element searches, so we consider only the difference between element searches. When indexed into the trees, they are both simply points. k-NN search resulted in 274.780 µs, and 20.351 µs for the KdTree vs. FsKDTree, respectively. This is an improvement on a factor of $13x$.

The C# *k*-d tree uses an internal priority queue to handle returning a list of the nearest neighbors, in addition to a hyperrect that it keeps track of while testing points in the different dimensions. Which does additional distance tests on searches, and reordering the queue on finding closer neighbors. FsKDTree does not have a priority queue as it only returns the single closest neighbor, decreasing the logic with the function call.

6.2 Grid prop fetching times

We want to measure how long it takes to download grid properties, since they are not currently tiled. Currently, as no pre-processing is done on the archives, we want to see how long it takes to fetch archive props. Users can step forwards in time, and on the larger model areas, this causes some delay. This is because when a scalar field is selected, all properties for the new timestep must be fetched from Sorcerer.

Host	Lenovo SR645 (7D2XCTOLWW)
CPU	AMD EPYC 7713 (256) @ 2.000GHz
RAM	16× 16 GB 3200MHz DDR_4
Harddrive	349TB XFS storage with RAID 60
OS	NixOS 22.11 (Raccoon) x86_64
Kernel	Linux 5.15.91

Table 6.5: Storage node hardware specifications.

Property	10 fetch avg.	Max	Min
Speed	723ms	836ms	451ms
Temperature	385ms	802ms	241ms

Table 6.6: Results of sampling property downloads. Taken from Oceanbox's production system. The model area is PO11, whose grid size is 22.72 MB. The properties are 3.02 MB.

This is measured by sampling download times in a browser's network console. Each call to Sorcerer is logged, and timings are displayed in milliseconds. Sampling these timings can show back-end performance.

Sampling was done in Oceanbox's production system, which is currently unavailable to the public. API calls are requested to Sorcerer, which is running on Ekman as a Kubernetes pod, which means a containerd[3] container. Its hardware specifications can be seen in table 6.5. The model area called 'PO11' is used to test prop fetching, as shown in figure 6.1. It is located in Troms and Finnmark, Northern Norway. The previous example 'LT3' is not available in the production system. 'PO11"s grid is 22.72 MB large.

We manually increment the time on model area 'PO11' with speed and temperature properties selected, read the time taken by each call, and finally average the times. Ten calls are sampled from each of the props from the production instance of Atlantis. The results are in table 6.6. This gives us an average request time of 723 ms for speed, and 385 ms for temperature. Speed has a max duration of 836 ms, and a minimum of 451 ms. Temperature has 802 ms, and 241 ms.

These are noticeable waiting times, especially for speed properties. When Sorcerer fetches the grid's velocities, it must convert every value from being located on the element to its nodes.

3. `https://containerd.io/`

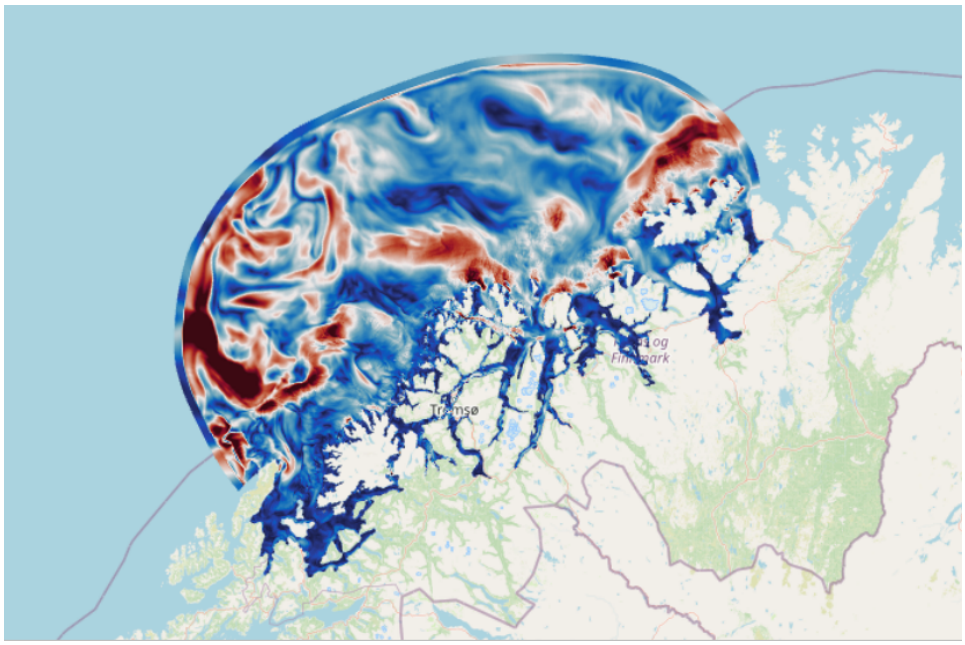

Figure 6.1: Image of PO11 with speed scalar field visualization enabled. It has a grid size of 22.72 MB, and for every props' time step one layer has 3.02 MB of data.

6.3 Grid and Scalar field rendering with WebGL

Drawing the smallest grids with the HTML Canvas API took up to 60 seconds per frame. With WebGL, 60 frames per second are achieved reliably.

6.4 Summary

Overall, our results show the following:

- FsKDTree outperforms the C# equivalent up to 4× in creating the tree. This comes from knowing the size beforehand, which avoids dynamic object allocations.
- Current barbs are deployed to production and perform sufficiently
- Model areas with up to one million elements can be viewed in their totality, though some latency can be experienced when time-stepping
- The particle streams draw up to 250000 particles moving at 60 frames per

second, which would not be possible with CPU rendering, as attempted with the HTML canvas API

7

Discussion

7.1 Current barb drawing

It is unnecessary to give the barb point file its API. The barbs are described in a file, which the user downloads after entering the website. This could have been avoided by simply including the geometries in JavaScript. Since the clients must download the web content anyway, they do not need to do a separate fetch for the barbs.

Currently, the barbs are drawn procedurally by connection lines between points described in a JSON file. An alternative to procedurally drawing current barbs is using images. The current barbs and their velocity representations could have images that clients download, rotate, and blit to their screens. However, this can be considered wasteful. An image will most likely be a rectangle, so the transparent space in the picture is hard to exclude from the request. In addition to this, doing what is essentially texture mapping onto the canvas is an expensive operation. Procedurally drawing lines based on a small text file might be less expensive. These costs could have been measured, but the overhead can be considered negligible. Here, some computation is moved onto the client, where they procedurally draw the barb instead of downloading a resource stored on Oceanbox's servers.

7.2 Spatial indexing

Many engines and systems work with spatial data[1]. Was it worth it to build ours from scratch? In the Alam survey, we can see many databases and systems for storing geographical data. We can also see these systems have their own spatial partitioning and indexing schemes. Oceanbox does not use any of these pre-existing systems to store oceanographic data. Instead, data is stored in a standard Linux filesystem, and a database keeps track of the archives and their files. Integrating these solutions into Oceanbox's system would mean duplicating component responsibilities. Custom components had already solved the storing and fetching of the data. It was also not within the scope of this paper to investigate such infrastructure problems. Since Sorcerer was already loading the data into memory, placing the spatial index here was a natural choice.

There are other spatial partitioning algorithms, like the R-tree. It creates minimum bounding rectangles around close elements. When a rectangle holds enough items, it can partition itself further by reducing the amount each rectangle bears. However, since we are indexing the nodes of a polygon and not many polygons. Therefore, choosing the *k*-d tree seems more intuitive. The complexity of the algorithm, contrasted with the net benefit, seems negligible. There are no performance issues identified with a *k*-d tree yet.

7.2.1 The k-d tree implementation

The nearest neighbor search algorithm seems to be lacking. It does not currently utilize a 'hyperrect' to ensure that the points found can be within the distance found.

The algorithm currently uses the exact distance instead of the squared distance. Instead, by using squared distances, we square operations, which are done per current barb in the searches. Still, it performs better than the C# *k*-d tree that uses squared distance comparisons.

There are ways to create more optimal implementations. One example is reducing the size of the data structures used and the organization of the data. An alternative is to use flat arrays for the points and a separate item array. This would be more cache friendly as the points being iterated over are more tightly packed. However, this can be a more complex implementation, as keeping track of pairs of points when moving them around is more prone to error. Then, with an idiomatic F# implementation that improves performance, it can be viewed as a success.

Currently, we use a recursive discriminated union that is either a leaf or a node that holds a tuple of sub-trees. A leaf is also a struct that holds another struct, the 'Pos' struct. This creates higher memory fragmentation, as more unnecessary data is used in the tree operations, invalidating more of the cache. Though, how the discrimination is represented in memory is somewhat opaque. The .NET platform creates a sizeable abstract layer on top of the programs and with garbage collection, which makes knowing how memory is handled more difficult.

The FsKDTree should have used an array for the points. This would allow a more straightforward implementation of a *k* dimensional tree. With an array, the number of dimensions could be its length instead of explicitly defining a position type for each supported dimension.

Existing implementations

The C# *k*-d tree already exists, so implementing our own could have been avoided. However, for development ergonomics, there are benefits to using a single language for the entire technology stack. In the case of C# is easy, as they share the same .dotnet runtime. C# is an object-oriented programming language first (or that is how it is mainly used). F#, on the other hand, is a functional first language, where immutability and having no side effects are central tenants in the language. Creating the C# involves adding each point separately through an `Add` method on the tree. The method does not return anything and is, therefore, a side-effect. This trivial example is not a problem in and of itself, but it breaks with the usual programming flow one has with F#.

k-NN bugs

Because this is a new implementation of a *k*-d tree, the stability of the solution is not sufficient for production use. Using the FsKDTree in a development environment and enabling the current barbs gave a fast response. However, zooming into the model area far enough will sometimes yield gaps in the current barb rendering. Near select triangle borders, there will be missing barbs. A possible explanation for this could be a bug in the nearest neighbor algorithm. If the k-NN search returns the centroid of the neighboring element, then testing whether the barb coordinate is inside another, this would naturally fail.

Meanwhile, the C# *k*-d tree is used in the production system to ensure correctness.

7.3 Model grid tiling

There are possibilities for pre-processing of model properties. We can render the model areas server-side and create our WMTS that serves tiled images of the scalar fields. This would reduce the data loads needed to download for the clients. Since the client would no longer need to download the entirety of the grid but only images that fit the 512x512 standard resolution of WMTS tiles.

This would not work, as large amounts of data are continually created, with 34 layers per time-step, usually one hour. Images must then be stored at different resolutions, per grid property, at each time step, and the different depths. Culminating in large amounts of data to produce and store. An alternative, however, would be to produce tiles only for forecasting data. The forecasting is only five days, which would keep the data amounts manageable. In addition, forecasting data are arguably in the highest demand, which would decrease the data throughput significantly on the data that is already the most sought-after.

7.4 Grid downsampling

To reduce the data load, a possibility would be to downsample the grid. This would involve collapsing the unstructured triangular grid. As the FVCOM triangle specifications enforce some attributes for the grid, we know that collapsing the triangles in half would still preserve the integrity of the model area. The grid resolution would degrade, but it would be sufficient for visualization.

7.5 Future Work

In addition to the improvements mentioned above, like ensuring the correctness of the FsKDTree. Oceanbox is still in the early development of its services. The company was founded in April of 2022 and has created its stack from scratch. Many optimizations can be made, but the system's overall architecture and feature set must settle before these are attempted. Features that still need to be added are things like grid tiling. Especially for forecasting data. The front end must be continually improved, and specialized to specific fields, to ensure a good user experience.

8

Conclusion

This thesis has described the design and implementation of visualizations for Oceanbox's Oceanography-as-a-Service web application. A *k*-d tree was introduced for spatial indexing when sifting through large amounts of hydrodynamic data. It enables responsive current barbs that sample large model areas and is available as a feature in the map tool. Together with the current barbs, the model areas are viewed and colored using GPU rendering. Finally, a stream visualization application is integrated into the map tool for a highly dynamic and visually pleasing view of the ocean's complex nature.

Together with the entire F# stack, the *k*-d tree was created from scratch. Compared to the .net C# *k*-d tree implementation, which provides a performance difference of up to 4× for tree creation and up to 13× for search operations. However, the C# implementation is more mature and more widely used, which increases its reliability, which is why our new implementation is not currently used.

Our work aims to show a new way to work with scientific data and applications. We are providing highly available and easy-to-use interfaces in any web browser. We can show large triangular unstructured hydrodynamic data sets in real time by leveraging computing graphics rendering techniques. Without image preprocessing before viewing results, an engaging feedback loop for those seeking operational intelligence in these valuable datasets is created.

Bibliography

[1] Md Mahbub Alam, Luis Torgo, and Albert Bifet. "A Survey on Spatio-Temporal Data Analytics Systems." In: *ACM Comput. Surv.* 54.10s (Nov. 2022). ISSN: 0360-0300. DOI: 10.1145/3507904. URL: https://doi.org/10.1145/3507904.

[2] Eric P. Chassignet et al. "The HYCOM (HYbrid Coordinate Ocean Model) data assimilative system." In: *Journal of Marine Systems* 65.1 (2007). Marine Environmental Monitoring and Prediction, pp. 60–83. ISSN: 0924-7963. DOI: https://doi.org/10.1016/j.jmarsys.2005.09.016. URL: https://www.sciencedirect.com/science/article/pii/S0924796306002855.

[3] C Chen, R.C. Beardsley, and G Cowles. "An Unstructured Grid, Finite-Volume Coastal Ocean Model (FVCOM) System." In: *Oceanography* 19(1) (Mar. 2006), pp. 64–77. URL: https://doi.org/10.5670/oceanog.2006.92.

[4] Changsheng Chen et al. *An unstructured-grid, finite-volume community ocean model: FVCOM user manual*. Sea Grant College Program, Massachusetts Institute of Technology Cambridge . . ., 2012.

[5] Tomasz Czajęcki. *Wireframes with barycentric coordinates*. Jan. 5, 2019. URL: https://tchayen.github.io/posts/wireframes-with-barycentric-coordinates.

[6] Marius Indreberg, Jonas Juselius, and John Markus Bjørndalen. "Particular: A Functional Approach to 3D Particle Simulation." Master thesis. UiT The Arctic University of Norway, May 2021.

[7] Ka Hin Lau, Jonas Juselius, and Lars Ailo Bongo. "Management of large geospatial datasets." Master thesis. UiT The Arctic University of Norway, May 2022.

[8] Pierre Yves Le Traon et al. "From Observation to Information and Users: The Copernicus Marine Service Perspective." In: *Frontiers in Marine Science* 6 (2019). ISSN: 2296-7745. DOI: 10.3389/fmars.2019.00234. URL: https://www.frontiersin.org/articles/10.3389/fmars.2019.00234.

[9] Gurvan Madec et al. *NEMO ocean engine*. URL: http://hdl.handle.net/2122/13309.

[10] Alexander F. Shchepetkin and James C. McWilliams. "The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model." In: *Ocean Modelling* 9.4 (2005), pp. 347–404. ISSN: 1463-5003. DOI: https://doi.org/10.1016/j.ocemod.2004.08.002. URL: https://www.sciencedirect.com/science/article/pii/S1463500304000484.