GPU-Based track visualization of multivariate moving object data

Roeland J. Scheepens
October 13, 2010
i Acknowledgements

First I would like to thank my supervisor, dr. ir. Huub van de Wetering, for his support and advice over the course of the project. I would also like to thank my tutor, ir. Niels Willems for his support and technical advice. Both my supervisor and tutor have supplied invaluable ideas and advice to help shape the course of the project.

Additionally, I would like to thank prof. dr. ir. Jack J. van Wijk, head of the visualization department at the TU/e, for supplying additional ideas.

Also, I would like to thank dr. Andrei C. Jalba and prof. dr. ir. Jack J. van Wijk for inspiring me to make the final decision to choose the visualization direction.

And finally, I would like to thank my parents for their support and financing the lion’s share of my education.
Abstract

Moving object data visualization and analysis is a growing field of research. In this thesis we present a continuation of previous work where paths of moving vessels are convolved with a kernel onto a density map to aid operators of coastal surveillance systems and analysts. The density map is visualized as a shaded height field and reveals hotspots of vessel activity.

Our improvements are twofold: First, we increase the computation speed of the density map by roughly a factor 3600 using modern graphics hardware and second, we introduce several visualization methods to reveal spatial and temporal distributions as well as relations between subsets of the data that are otherwise not visible in the traditional density map.

We present a geographic information system that allows an operator to analyze data sets containing moving objects. While the main focus in this thesis is data sets of ocean faring vessels, we show that our method is applicable to other types of moving object data sets as well.
### Contents

I  ACKNOWLEDGEMENTS .................................................................................. 3

II  ABSTRACT ................................................................................................. 5

III  CONTENTS ............................................................................................... 7

IV  LIST OF ACRONYMS & ABBREVIATIONS ................................................. 9

CHAPTER 1.  INTRODUCTION ........................................................................... 11
   1.1.  DATA SET .......................................................................................... 12
   1.2.  RELATED WORK .............................................................................. 15

CHAPTER 2.  OVERVIEW .................................................................................. 17
   2.1.  ORGANIZATION .............................................................................. 19

CHAPTER 3.  PREVIOUS WORK ....................................................................... 21
   3.1.  MOVEMENT MODEL ....................................................................... 22
   3.2.  DENSITY ......................................................................................... 23
   3.3.  GEOGRAPHIC SPACE ..................................................................... 24

CHAPTER 4.  DENSITY COMPUTATION ............................................................ 25
   4.1.  TIME WINDOW ............................................................................... 26
   4.2.  AGGREGATION ................................................................................. 28
   4.3.  DENSITY COMBINATION OPERATOR ................................................. 32

CHAPTER 5.  VISUALIZATION .......................................................................... 37
   5.1.  PROJECTION .................................................................................. 37
   5.2.  COLOR MAP .................................................................................... 39
   5.3.  SHADING ......................................................................................... 39
   5.4.  SCALING ......................................................................................... 39
   5.5.  VISUALIZATION OPERATOR ............................................................. 40

CHAPTER 6.  IMPLEMENTATION ................................................................. 45
   6.1.  INTRODUCTION TO THE GPU ....................................................... 45
   6.2.  PARALLELIZATION ......................................................................... 48
   6.3.  IMPLEMENTATION ON THE GPU ................................................... 49
   6.4.  AGGREGATION ............................................................................... 59
   6.5.  DENSITY COMBINATION OPERATOR .............................................. 59
   6.6.  VISUALIZATION .............................................................................. 59
   6.7.  VISUALIZATION OPERATOR ............................................................ 62

CHAPTER 7.  RESULTS ...................................................................................... 65
   7.1.  COMPARISON .................................................................................. 65
   7.2.  BENCHMARKS .................................................................................. 65
   7.3.  USE CASES ...................................................................................... 72
# List of Acronyms & Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIS</td>
<td>Automatic Identification System</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>Cg</td>
<td>C for graphics, Nvidia's shader language</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CT</td>
<td>Computed Tomography</td>
</tr>
<tr>
<td>Cuda</td>
<td>Compute Unified Device Architecture, Nvidia's general purpose GPU language</td>
</tr>
<tr>
<td>FBO</td>
<td>Frame Buffer object</td>
</tr>
<tr>
<td>GCD</td>
<td>Great Circle Distance</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographic Information System</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose computation on the GPU</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>IMO</td>
<td>International Maritime Organization</td>
</tr>
<tr>
<td>KDE</td>
<td>Kernel Density Estimation</td>
</tr>
<tr>
<td>MRI</td>
<td>Magnetic Resonance Imaging</td>
</tr>
<tr>
<td>OBB</td>
<td>Oriented Bounding Box</td>
</tr>
<tr>
<td>Pixel</td>
<td>Picture Element</td>
</tr>
<tr>
<td>SOLAS</td>
<td>International Convention for the Safety of Life at Sea</td>
</tr>
<tr>
<td>Texel</td>
<td>Texture Element</td>
</tr>
<tr>
<td>VBO</td>
<td>Vertex Buffer Object</td>
</tr>
<tr>
<td>VTS</td>
<td>Vessel Tracking System</td>
</tr>
<tr>
<td>WGS'84</td>
<td>World Geodetic System (1984 revision)</td>
</tr>
</tbody>
</table>
Chapter 1.

Introduction

A moving object is defined as an entity whose position changes over time [22]. These objects can represent anything ranging from airplanes to particles. Moving objects have primitive attributes such as position and time instance, primary derivatives such as direction and velocity and secondary derivatives such as spatial distribution, acceleration and approaching rate. While the primitive attributes are the most basic attributes that define the movement behavior of the objects, the derivatives are attributes derived directly or indirectly from the primitive attributes. Dodge et al. [22] divide moving objects into two major groups. The first are the geo-referenced objects which are objects that move through geographic space and as such can be referenced using geographic coordinates. This group includes moving objects such as cars, pedestrians, animals or ships. The second group is the group of non-geo-referenced objects which are objects that move through non-geographic space such as tracked eye movements. In this thesis we only deal with the former group.

The analysis of moving object data is performed in varying scientific fields such as meteorology, biology, sociology, transport engineering and analysis of animal movement behavior and migratory patterns. Where analyzing the movement patterns of the moving objects is the goal, geometric attributes such as the dimensions of the objects are ignored and as such the moving objects are treated as moving points. Datasets containing movements are often analyzed by finding and studying movement patterns. Movement patterns are spatial and temporal regularities or any other relationship between points, or even between different objects, where the definition of these relationships depends on the application domain. Through the advance of technology, increasingly larger and more complex datasets of varying types are becoming available for analysis. These datasets can be so large and complex that a human observer cannot find any structure in the data without some visual setting. This means that analysis and visualization tools are required to find structure within the data sets of moving objects.

In this thesis we present such a visual setting in the form of a Geographic Information System (GIS)\(^1\) for analyzing and visualizing the movements of ocean-faring vessels. We do this by plotting a density map containing the area usage of the vessels. The area usage of the moving vessels is computed using kernel density estimation [26]. We do this by blurring or convolving our movement data using some kernel function. We improve and extend on an existing method by Willems et al. [1], designed to compute and visualize a density map of moving vessels. As our contributions, we significantly improve the computation speed by utilizing modern graphics hardware and introduce several interactive visualization techniques that have been made possible by the increase in computation speed.

The moving objects are given by a set of points with a time and a velocity that, when connected, define a track. Using some movement model we show later, the behavior is described based on the given points in the data set with the associated attributes, time and velocity. These tracks are convolved onto a density map such that areas with more activity have a larger density. Using this map, an operator can find global usage patterns at a glance or find more subtle usage patterns using the visualization techniques we describe below in otherwise complex and large data sets. Moreover, using a density map of historical data, an operator can easily detect anomalous vessel behavior such as vessels moving outside of designated shipping lanes or vessels moving into nature reserves.

\(^1\) A GIS is a system for analyzing and visualizing data associated with geographic positional information. This data is usually displayed on a map.
In this thesis we mainly concentrate our work on moving vessels. In particular, seafaring vessels. However, our method is applicable to any moving object data that can be expressed with tracks and some movement model, such as migration behavior of birds, pedestrian movements in a city or airplane activity.

To maintain a standard of safety on the seas most countries have ratified an international treaty called the International Convention for the Safety of Life at Sea (SOLAS), which is being maintained by the International Maritime Organization (IMO). At the start of 2003 an amendment to the SOLAS came into effect that required every seafaring passenger ship or other ship with a Gross Tonnage (GT) of at least 300 tons to have the tracking system Automatic Identification System (AIS) installed [24]. Using this system, seafaring vessels broadcast their static information such as the name of the vessels and its call-sign and dynamic information such as its position, heading and speed. This data can be used by vessel Tracking Systems (VTS) or other vessels to determine the location, heading and speed of other vessels [23] in a certain area.

In the following sections we discuss the structure of our data set including a more extensive review of the movement pattern taxonomy given by Dodge et al. [22]. We conclude this chapter with an overview of related work.

1.1. Data Set

Use has been made of data sets of moving vessels around the Dutch coast and a data set of pedestrian movement in the city center of Delft. In the following, we mainly focus on vessel movements. The vessel data has been collected using automatic identification systems—see Appendix B.2.

The type of data set we work with contains a set of objects (Ω), where each object has attributes such as size and name. Each object also has an ordered set of points, where each point has at least the following attributes:

- Geographic position \( p = (\lambda, \phi) \) on the surface of the earth in longitude \( \lambda \) and latitude \( \phi \).
- Velocity \( v \) as a scalar value in \( km \) per hour at which the object moves. The velocity cannot be negative.
- Absolute time \( t \) at which the ship was at point \( p \).

Note that the velocity in our dataset is nothing more than the vessel speed, the rate of change of the vessel’s position, and as such does not contain any directional information. Also, due to the precision of the navigation satellite system and the ship’s own instruments, the velocity may not be accurate. The points are ordered by absolute time and, when connected, form a track. These tracks describe the movement of the object. We make the assumption that a moving object moves from one point to the next in a straight line. A movement model, that describes how the objects move in between their points, is applied to the tracks—see section 3.1.

The data set has a minimum time \( T_0 \) and a maximum time \( T_1 \), however, for simplicity we say the data set starts at time 0 and ends at time \( T \) where \( T = T_1 - T_0 \). The position of our points is given in geographic space. We have to use a map projection function to be able to display our tracks on a flat screen. See Appendix A for more information on geographic space and map projections.

1.1.1. Movement Patterns

Dodge et al. [22] provide a convenient taxonomy for movement patterns. Moving objects can form either continuous paths, which are curvilinear paths that flow more or less continuously, or moving objects can form discontinuous paths, which are paths characterized by short movements and stops such as observed with bees moving between flowers or eye movements. Vessel movements are best characterized as continuous paths. A number of factors that can influence the way objects move are given by Dodge et al. [22]. These factors are as follows:
**Intrinsic Properties** are properties of the moving objects that constrain its movement such as physical properties. In the case of moving vessels these are properties such as the maximum speed of the vessels, the maximum acceleration time and the maximum rate of turn.

**Spatial Constraints** are external objects that directly constrain the movement of the objects. These can be external objects such as the coastline or ocean platforms.

**Environment** in which the objects move. The weather is such an environmental factor for moving vessels.

**Influences of other agents.** The movement of an object can be directly influenced by the movements of other objects. For example when a vessel intersects a shipping lane, it may become necessary for vessels on the shipping lane to perform evasive maneuvers.

A distinction is made between **generic patterns** such as concentration or spatio-temporal periodicity, that can be found in any type of dataset, and behavioral patterns such as flock or migration, that are more specific to data sets containing the movement data of animals such as birds. For this thesis we are interested in movement patterns that can occur in moving vessel data. We present a non-exhaustive list of movement patterns we may observe in moving vessel data based on the taxonomy given by Dodge et al. [22].

- **Concentration.** A movement pattern related to the spatial concentration of moving objects in a certain point in time. In our dataset this manifests itself as anchorage areas, which are designated areas near harbors in which ships are allowed to anchor, and harbors.
- **Co-Location in Space.** A movement pattern that occurs when paths have two or more positions in common. In our dataset this movement pattern can be clearly observed in the designated shipping lanes.
- **Constancy.** If the properties of the movement such as speed or direction do not change or change only very slightly for a longer period of time we can speak of constancy. This is a common movement pattern in vessel movements due to the slow acceleration and slow rate of turn of sea-faring vessels.
- **Dispersion.** This movement patterns occurs when multiple objects that have similar movement properties disperse with non-uniform motion. This pattern occurs in our dataset near the harbors. Where vessels leave the harbor with similar paths they disperse in different directions with different speeds when they reach the open sea.
- **Periodicity/Repetition** occurs when a moving object performs temporal periodic behavior. This means that the moving object repeats certain movement patterns within a regular time interval in the case of periodicity and irregular time intervals in the case of repetition. This occurs for vessels that ferry between set locations.
- **Isolated object** is an individual object that follows its own path without any influence from other moving objects. This can occur for maintenance vessels that are moving near ocean platforms where no other vessels are.
- **Convergence/Divergence** is the movement pattern occurring, respectively, when moving objects come to the same location without significantly moving their direction or when moving objects move away from a common position. This pattern can be observed at harbors where ships enter and leave the harbor.
- **Evasion** is the movement pattern that occurs when one moving object attempts to get away from or avoid another moving object. This movement pattern can occur when ships intersect shipping lanes.
- **Congestion** occurs when movement speed slows down due to the presence of other vessels. In our dataset this may occur around the harbors or on rivers.
Figure 1. An overview of the given movement patterns: A) Concentration: Several vessels are anchored in a designated anchor zone near the harbor of Rotterdam. B) Co-Location: A number of vessels follow the designated shipping lanes off the Dutch coast. C) Constancy: Vessels following a constant course with a constant velocity. D) Dispersion: A number of vessels leave a harbor and go off into varying directions. E) Periodicity/Repetition: A ferry going back and forth over the Westerschelde. F) Isolated Object: A maintenance vessel near ocean platforms visualized as black squares. G) Convergence/Divergence: Ships slowly converge as they enter the harbor of Rotterdam and slowly diverge as they leave the harbor of Rotterdam. H) Evasion: A vessel moving over a shipping lane changes course to evade a ship crossing the shipping lane. I) Congestion: Congestion occurs as the ferry over the Westerschelde has to stop to let other ships pass and passing ships have to lower their speed to evade the ferry.
1.2. Related Work

While an increase in data storage capacity is enabling the storage of larger and more complex sets of moving object data there is still little research in the field of moving object visualization, especially in the field of geo-referenced data. The focus in research appears to be largely in the area of hot spot analysis and visualization. Wang et al. [34] use taxi movement data in urban areas to find and visualize hot spots caused by taxi pickup and drop off points. These hot spots are used to support urban planning.

Meschtscherjakov et al. [31] investigate a prototype system for visualizing customer activity within a store. A map of the store is shown with hot spots of customer activity. This allows customers to see the activity behavior of other customers, which was previously exclusive to online shops. Eye movement analysis [35] is another popular field of movement analysis. Eye movements are non geo-referenced movements recorded using specialized tools. Bednarik et al. [33] use eye movement analysis and visualization to study program comprehension. A tool to visualize program structure is used where the eye movement data, such as location and duration of fixations, of subjects is recorded. This data is then used to study the program comprehension process.

1.2.1. Geographic Information Systems

Geographic Information Systems (GIS) are systems that combine analysis and visualization tools for geo-referenced data. Wang et al. [34] combine analysis and visualization tools for their taxi movement data to form a single GIS that can both analyze and visualize traffic patterns and urban hot spots. Vance et al. [36] propose a GIS to analyze and visualize particle movements generated using particle tracking models. This enables the analysis and visualization of environmental phenomena such as the path of volcanic ash ejected by an eruption.

1.2.2. Multivariate Visualization

Datasets often have multiple data values per data point. This type of data is called multivariate data. To display all the data values in a single data point, specialized techniques are required to composite and visualize these data values. If the different data values can be expressed using a color, the colors can be blended together. Gossett et al. [5] show a method of combining color information based on the mixing behavior of thin layers of paint. A noise texture is used with varying opacities to mix different colors. This avoids the ambiguity caused by directly mixing different colors. There are, however, other ways to show multiple data values. Miller [29] combines different data values represented by color maps through the use of attribute blocks. The image space is divided into a regular grid of blocks. Each block shows a different data attribute in a repeating pattern. Schneiderman [30] shows how to visualize a large number of records of a database into a single visualization using several different visualization techniques.

1.2.3. Transfer Functions

Transfer functions are a popular tool in the field of volume visualization used to highlight parts of volumetric data that are of interest. A transfer function generally maps a certain data value to an opacity and a color. For instance, in the medical field, transfer functions can be used to separate tissue types such as bone or skin in three-dimensional data obtained through a CT or MRI scan. It is generally hard to find a good transfer function. While methods exist to make finding a good transfer function [40] easier, it is generally a case of creating these functions manually using trial and error.

Bruckner et al. [32] go a step further than the classic transfer functions and introduce advanced style transfer functions that map data values to an extended style type. This enables multiple non-photorealistic rendering techniques and other complicated rendering styles in a single framework. Kniss et al. [39] show how to apply multi-dimensional transfer functions to scalar data, using derived attributes such as the gradient, or to multivariate data. Kindlmann et al. [37] use multi-dimensional transfer functions using curvature information to enhance non-photorealistic rendering techniques.
Similarly, Lum et al. [38] use a multi-dimensional lighting transfer function to enhance the visibility of boundaries in the volume.
Chapter 2.

Overview

In this thesis we concentrate on optimizing and extending the work on vessel density plots by Willems et al. [1]. Our main contributions are two-fold. First we reduce the computation time of the density map significantly using the parallel computational power of modern graphics hardware. And second, we introduce several visualization techniques intended to find patterns and correlations between different parts of the data set based on quantities such as time or velocity. Due to the speed increase we achieve using dedicated graphics hardware, we enable direct interaction on the visualization by changing the parameters of the density map computation and the visualization—see Figure 2.

The main purpose of our system is to visualize the *area usage* patterns of moving objects in the form of a *density map*—see Figure 4. A density map is a planar representation of the data set in which each point has an associated density related to the amount of time the moving object has been *near* that point. We use a *kernel function* to define the concept of *near* such that moving objects contribute more to a certain point the closer they are to it.

Our system globally works as shown in Figure 2: We extract our moving object data from our object data set $\mathcal{O}$ and define a continuous movement model to interpolate the movements the objects make—recall from section 1.1 that our data set is discrete in time. This movement model gives us smooth *tracks* that are defined for all points in time. Now that we have defined the smooth tracks, we compute a density map. We then visualize this density maps using several visualization techniques resulting in a final image.

Our data spans over a certain geographic region. Before computing the density map, we divide this region into a uniform grid of *cells* with the same geographic area and compute the density for each cell.
cell separately. On the resulting density map we apply visualization techniques to visualize the density map and make it more intuitively understandable to the human eye. Both the parameters of the density map computation and the parameters of the visualization can be adapted based on user interaction to, respectively, change how the density map itself is generated or the way the density map is visualized.

We separate our system into two distinct subsystems—see Figure 3: The density map subsystem that computes the actual density maps and the visualization subsystem that turns the density map into a picture that can be interpreted by the human eye. In the density map system we work in geographic space while in the visualization subsystem we work in screen or image space. We convert our density map in geographic space to a density map in image space using a map projection.

We can generate and visualize different types of density maps. We can show subsets of the data in time to visualize spatio-temporal relations or combine different density maps into a single visualization—see Figure 4 to Figure 7. In the density map subsystem we use the continuous curvilinear tracks resulting from a continuous movement model. From these tracks we generate one or more density maps using different techniques and parameters. These density maps are then combined using some operator we call $\oplus$. See Figure 3, for an example with two density maps. In the visualization subsystem we use the projected density map created by the density map subsystem to generate a colored

---

**Figure 3.** A schematic overview of our system. A continuous movement model is applied such that we get a set of curvilinear tracks, here a track of a single vessel. We compute one or more density maps using this movement data and combine these density maps using some operator $\oplus$. In this image it is the simple addition of two density maps. The resulting density map is projected into image space and used to compute a colored image and a normal map for shading. A final image is then generated using the visualization operator $\otimes$. In this image the result is a shading and color mapped density map.
and shaded image. The final visualization image is assembled using a visualization operator we call $\otimes$. See Figure 3.

### 2.1. Organization

In Chapter 3 we explain, in a theoretical framework, the work of Willems et al. [1] for the specific case of moving vessels and how, using their movement model, the density map is computed using convolution. Following this we give our contributions to the framework. In Chapter 4 we show a theoretical overview of how we reduced the computation time and we also show several new techniques to generate different types of density maps, each for a specific purpose such as bringing out subsets of data or showing how the moving objects behave over time. We also pay specific attention to the operator $\otimes$ and explore its possibilities.

In Chapter 5, we show how we visualize the density map and which techniques we can use to bring attention to desired details. We explore the possibilities of the visualization operator $\otimes$ and what kind of useful visualizations we can generate using this operator.

With the theoretical framework for our method in place we move on to the practical implementation using graphics hardware in Chapter 6. We start with a general overview and introduction to modern graphics hardware. We then describe the general architecture of our system followed by a detailed
description of the implementation of the density map using the methods described in Chapter 3 and Chapter 4, followed by how we implemented the visualization techniques described in Chapter 5.

Finally, we show the results in Chapter 7 including a number of use cases that illustrate how our method can be used, followed by a discussion and conclusion in Chapter 8.
Chapter 3.

Previous Work

In this chapter we give an overview of the previous work done by Willems et al [1]. Willems et al. [1] have devised a geographic visualization intended to help analysts and operators of vessel tracking systems. This is done by generating a density map of historical data that can be used as an overlay in a vessel tracking system to detect anomalous vessel behavior. A movement function is convolved with a kernel function which results in a smeared out or blurred version of the tracks—see Figure 8.

A movement model is required to turn the points of the data set into smooth tracks in the form of polylines. These tracks are then convolved or blurred onto a density map by computing the integration of a kernel along the movement line for each line segment of the polyline—see Figure 8. The geographic space in which our dataset resides, is divided into a uniform grid of cells where each cell has the same surface area. The density is then computed separately for each cell $Q$ where the density contribution of each object $o \in O$ to cell $Q$ is summed into the total density of cell $Q$. The density of each cell is recorded in a density map.

Figure 8. The paths of our moving objects are in the form of poly-lines (A). The density contribution of each line segment of the poly-line is computed separately (B). These line segments are then convolved using some kernel (C). The resulting convolved line segments are summed which gives us a convolved track.
3.1. Movement Model

In this section we describe the movement model as used by Willems et al. [1] and how it is applied to turn the discrete data points into smooth tracks see—Figure 9.

Since the moving object data consists of a set of objects moving over the surface of the earth, where each object is a set of individual points with a time and a velocity, a model is needed to describe, or rather reconstruct, the movement from one point to the next, resulting in continuous tracks. It is assumed that the moving objects move from one point to the next in a straight line and cannot reverse or change direction within that line segment.

In this movement model the velocity given in the points is taken into account, as well as the acceleration $a$ we derive from the velocities and time of these points below. This means objects can accelerate ($a > 0$), decelerate ($a < 0$) or keep a constant speed ($a = 0$) within a line segment. It is assumed that this acceleration is constant over a line segment. Since they do not want objects to reverse within a line segment, the velocity is assumed to be non-negative—see section 1.1.

For two points $p_0$ and $p_1$ with times $t_0$ and $t_1$ respectively, the movement from point $p_0$ to point $p_1$ is modeled with the parametric equation $p(t)$ as follows [1] where we call $x$ the line parameter:

$$x(t) = \frac{1}{2} a(t - t_0)^2 + v(t - t_0)$$  \hspace{1cm} (1)

$$p(t) = p_0 + x(t) \cdot \frac{p_1 - p_0}{||p_1 - p_0||}$$  \hspace{1cm} (2)

Where $x(t_0) = 0$, $x(t_1) = ||p_1 - p_0||$, $v$ is the initial velocity and $a$ is the constant acceleration over line segment $p_0p_1$. The parameters $a$ and $v$ are given as follows [1]:

$$v = \frac{||p_1 - p_0|| - v_1 - v_0}{t_1 - t_0}$$  \hspace{1cm} (3)

$$a = \frac{v_1 - v_0}{t_1 - t_0}$$  \hspace{1cm} (4)
Where $v_0$ and $v_1$ are scalar velocities of point $p_0$ and point $p_1$, respectively, as given in the data set. The line parameter $x(t)$, is monotonically increasing, which means that for all $t \geq t'$ it holds that $x(t) \geq x(t')$.

This movement model has been chosen, because it is simple and describes the movement of vessels fairly well. Vessels generally slowly change speed and/or direction. Acceleration is taken into account so that the transition between line segments is smooth. The velocity and acceleration are taken into account to show a distinction between slow moving objects, fast moving objects and objects that are changing their velocity. Slow moving objects generally contribute more to the density in an area than fast moving objects.

### 3.2. Density

With the movement function of the moving objects defined, the continuous movements can be convolved onto a density map. This is done by integrating the movement function over time, convolved with a finitely supported, radially symmetric kernel $K$, which takes a distance and a kernel radius $r$ as its parameters. Since a radially symmetric kernel function has been chosen, only a distance and a radius are required as its parameters.

For some point $q$ the density contribution of a moving object $o$ with movement function $p : [0, T] \rightarrow \mathbb{R}^2$ to point $q$ is then given by the following function [1]:

$$
C_o(q) = \int_0^T K(||p(t) - q||, r) \, dt
$$

(5)

The input to the kernel function $K$ is the distance between the sample point $q$ and the position given by the movement function $p(t)$ at time $t$.

#### 3.2.1. Multiple Objects

Only a finite amount of samples of our density map can be computed, so the density must be sampled per region which is called a cell, as described by Willems et al. [1]. A cell is a partition in geographic space. Each cell has a certain geographic surface area so to get accurate results the entire region must be sampled. The density contribution of an object $o$ to a cell $Q$ is then as follows:

$$
C_o'(Q) = \iint_Q C_o(q) \, dq
$$

(6)

The density $D(Q)$ of a cell $Q$ is the sum of all density contributions of all objects to that cell:

$$
D(Q) = \sum_{o \in \Omega} C_o'(Q)
$$

(7)

Where $C_o'(Q)$ is the density contribution of object $o$ to cell $Q$ as in equation (6) and $\Omega$ is the set of all moving objects. Since we have chosen to divide our space into uniform cells with equal area, however, we can approximate equation (7) by simply sampling in the center of the cells and multiplying by the cell area $A$. The density of a cell $Q$ then becomes as given by the following approximation:

$$
D(Q) \approx \sum_{o \in \Omega} C_o(q) \cdot A
$$

(8)

Where point $q$ is the center point of cell $Q$. 

---

This information is derived from the aforementioned source and is provided to offer a comprehensive understanding of the topics discussed.
3.3. Geographic Space

Real-world data is supplied with coordinates in geographic space. This means the movement model has to be slightly adapted to the properties of geographic space as explained in Appendix A.1. First, the assumption is made that, when an object moves from point \( p_0 \) to point \( p_1 \), it moves with a constant bearing. This means our line segments are expressed as Rhumb line segments which we denote by \( \overline{p_0 p_1} \)—see Appendix A.3. We now have to change our movement functions of equations (2) and (3) to the following:

\[
\dot{v} = \frac{\|\overline{p_0 p_1}\|}{t_1 - t_0} - \frac{v_1 - v_0}{2}
\]

(9)

Where \( \|\overline{p_0 p_1}\| \) is understood to be the Rhumb line length as given by equation (51). The acceleration \( a \) remains unchanged. Since a Rhumb line is actually a curved line in geographic space, we cannot simply apply equation (2) to get a point on the line with parameter \( x(t) \). To be able to parameterize the movement as a straight line, the line segment must first be projected into a space where Rhumb lines are mapped as straight lines. There is such a space, namely, Mercator space—see Appendix A.2. Using the Mercator projection function \( M_p \)—see equation (49)—we map the Rhumb line to a straight line. Due to the distortion inherent to the Mercator projection, however, the length of a Rhumb line in Mercator space is equal to or greater than the actual length of the Rhumb line. To compensate for this distortion we normalize the line parameter \( x(t) \) into the range \([0,1]\) by dividing the line parameter \( x(t) \) by the Rhumb line length \( \|\overline{p_0 p_1}\| \). The normalized line parameter \( x'(t) \) for geographic space is then as follows:

\[
x'(t) = \frac{x(t)}{\|\overline{p_0 p_1}\|}
\]

(10)

Once we have a point in Mercator space for some time \( t \), we can project it back into geographic space, as shown in Figure 10, using the inverse Mercator projection \( M_p^{-1} \)—see equation (50). For some time \( t \), the point \( p(t) \) is now given as follows:

\[
p(t) = M_p^{-1}(p'_0 + x'(t) \cdot (p'_1 - p'_0))
\]

(11)

Where \( p'_0 = M_p(p_0) \) and \( p'_1 = M_p(p_1) \). With \( p(t) \) given in geographic space we can accurately compute the geographic distance between \( p(t) \) for any \( t \) and some geographic point \( q \). While the parameterization of the line segment in Mercator space is not linear due to the latitude distortion of the Mercator projection, we find that it suffices for small line segments such as those found in the data set of the Dutch coast (approximately 400 × 400 km). The Mercator distortion is near to linear further away from the poles (e.g. Around the Dutch coast) and only becomes significantly non-linear near the polar regions—see Appendix A.2.
Chapter 4.

Density Computation

The main problem of the method of Willems et al. [1] is its computation time. Generating a density map of approximately 1000 by 1000 cells, which is not unusual over an area of 400 by 400 km, may take up to several hours. This makes applying real-time visualization techniques, even for smaller density maps, infeasible.

In this chapter, we start by proposing a fundamental change in the way the density map is computed, leading to an increase in speed, that opens up new possibilities. We discuss some of these possibilities for generating density maps for analyzing spatio-temporal relations and generating density maps that are aggregates of multiple density maps of subsets of the data, generated with different kernel sizes.

The most important difference between the computation model as used by Willems et al. [1] and the computation model proposed in this thesis is parallelization—see Figure 11. The final density in one cell does not depend on the density of other cells. Conversely, the density contribution of one object does not depend on the density contribution of another. And even further, the density contribution of a single line segment does not depend on the contribution of any other line segment at all. This means that we can process our density contributions completely independently in parallel.

There are two ways to achieve this parallelism. We can either parallelize from the perspective of the moving objects or we can parallelize from the perspective of the cells. For parallelization from the perspective of the moving objects, as shown in Algorithm 1, we can follow roughly the same procedure as used by Willems et al. [1]. Only, for each object we process each line segment independently in parallel. An oriented bounding box is constructed around each line segment. We then compute the density contribution of the line segment to each of the cells intersected by the OBB in parallel. For parallelization from the perspective of the cells—see Algorithm 2, we process each cell Q independently in parallel. For each cell Q, we search our data set for line segments that have an oriented bounding box intersecting the cell Q. If the oriented bounding box intersects the cell Q, we know that

```
Sequential: C_i(q_1) ----> C_i(q_2) ----> ... ----> C_i(q_N) ----> Density Map
Parallel: C_i(q) ----> Density Map
```

*Figure 11. We use a model in which we compute our density contributions in parallel whereas Willems et al. [1] sequentially compute the density contributions.*
it is possible that the line segment has some positive contribution to the density of the cell. For each
line segment with an intersecting OBB we then compute the density contribution to cell \( Q \) and add it
to the final density \( D(Q) \) of the cell. This second method requires a spatial subdivision data structure
such as a quad-tree or a kd-tree [27] to efficiently look for line segments with a bounding box that
intersect a given cell. We would have to generate an oriented bounding box for each line segment
when we instantiate the data structure. Both methods would require some form of synchronization
when adding intermediate density contributions to the final result of a cell. Improper synchronization
could lead to erroneous results.

For this thesis we have chosen to parallelize from the perspective of the objects. The motivations and
details behind this choice are explained in Chapter 6.

Vessel density() in Parallel from objects

\[
\text{for} \quad o \in \text{Objects} \quad \text{do in parallel} \\
\quad \text{for} \quad i \in \text{Trajectory Segments}[0, N] \quad \text{do in parallel} \\
\quad \quad p_0 \leftarrow p_o(t_i); p_1 \leftarrow p_o(t_{i+1}) \\
\quad \quad \text{for} \quad (u, v) \in \text{OBB at distance } r \quad \text{to } p_o p_i \quad \text{do in parallel} \\
\quad \quad \quad q \leftarrow \text{Center of the cell } (u, v) \\
\quad \quad \quad D(u, v) \leftarrow D(u, v) + C_{p_0 p_i}(q)
\]

Algorithm 1. Parallelization of the density computation from the perspective of the moving objects.

Vessel density() in Parallel from cells

\[
\text{for} \quad all \quad cells \quad Q \quad \text{do in parallel} \\
\quad S \leftarrow \text{All line segments that have an OBB intersecting } Q \\
\quad \text{for} \quad (p_o, p_i) \in S \quad \text{do in parallel} \\
\quad \quad q \leftarrow \text{Center of the cell } Q \\
\quad \quad D(Q) \leftarrow D(Q) + C_{p_o p_i}(q)
\]

Algorithm 2. Parallelization of the density computation from the perspective of the cells.

### 4.1 Time Window

To see how the moving objects behave and how their movements evolve over time, we isolate certain
interest points in time. In other words, we want the density of a certain moment in time instead of the
density over the entire time range of the data set. A single moment in time, however, does not give
any information on how the movement behavior evolves over time. To this end we introduce the sliding time window \([t_{ip} - T_w, t_{ip}]\) defined by an interest point in time \( t_{ip} \) and a window size \( T_w \). Only
the movements of the objects within this time window are convolved onto the density map. This gives
the interest point \( t_{ip} \) a context in the form of its immediate history. In Figure 13 A we show an example
of three vessels within a time window of two hours. There is one vessel moving between the
islands of Terschelling and Vlieland and a vessel anchored off the coast of each island. While we can see
the spatio-temporal distribution of the three vessels, we cannot see the direction in which the moving
vessel is moving within the time window.

To give a sense of direction, a weight function can be applied to the objects in the time window that
scales the size of the kernel and the resulting density through time. We now change the density func-
tion of equation (5) to the following:

\[
\int_{t_{ip} - T_w}^{t_{ip}} w_z(t)K(||p(t) - q||, w_z(t)r) \, dt
\]

(12)

Where \( w_z(t) \) is a time weight function that is zero outside\(^2\) of the time window \([t_{ip} - T_w, t_{ip}]\) and
describes some weight inside the time window. In the above function we use the same weight func-

\(^2\) The weight of the time weight function \( w_z \) is equal to zero outside of the time window \([t_{ip} - T_w, t_{ip}]\) such that movements
outside of the time window never have a non-zero contribution to the density.
tion for both the kernel radius and the density. This is, however, not necessary and is only a practical consideration since the function will only need to be evaluated once if both weight functions are the same. If we now apply a linearly increasing weight function to the vessels in the previous examples, we can actually see that the moving vessel has moved from Terschelling to Vlieland within the two hour time window—see Figure 13 B.

We use a generic weight function that is monotonically decreasing, such that the weight has an order over time and has a parameter that controls the rate of decline of the weight over time. Our time weight function with parameter $\alpha$ is given as follows:

$$
W_t(t, \alpha) = \begin{cases} 
(t - (t_{ip} - T_w))/T_w & \text{if } t_{ip} - T_w \leq t \leq t_{ip} \\
0 & \text{otherwise}
\end{cases} \tag{13}
$$

Where parameter $\alpha$ controls the rate of decline of the weight based on the input time.
To give a better insight into how the spatial distribution of the vessels evolves over time, we make the interest point movable over time. This allows us to animate the movements in the data set to see how the individual vessels behave over time, which is information that is generally lost within a density map of the entire data range.

4.2. Aggregation

A density map consists of several layers of movements at different times and with different velocities. Also, each moving object has different properties such as size or vessel type—see Appendix B.2. These layers are hard, if not impossible, to distinguish on a density map containing a large amount of movements. We introduce a special type of density map within our density convolution framework to emphasize points or layers of interest. We divide the data set into several subsets which we call bins based on some criteria. A bin is a subset of the data based on some data filter function. As an example, if a dataset ranges over an entire day the dataset can be segmented uniformly in bins such that each bin contains two hours of object movements. For each bin we generate a density map. These density maps are then aggregated using some to be defined operator. Each density map may have used different convolution parameters such that some bins show up stronger or in more detail in the aggregated density map. This can be done by varying the kernel radius per bin or attaching a weight to the density per bin.

4.2.1. Density Map Aggregation

The dataset is divided into a set of $N$ bins. Each bin $i$ has an associated filter function $F_i$ that, for each object in the dataset, returns a subset of line segments of the object. A set of bins can therefore be a collection of sets ranging over an attribute such as time or any other continuous or even discrete attribute. For example, we can filter on objects using ship types or filter on line segments individually using velocity. Even further, the binning of the objects need not be based on any attribute at all, but can be any arbitrary set of subsets. Each bin also has an associated kernel radius given by the kernel radius transfer function $r : [1..N] \rightarrow \mathbb{R}$ and a weight given by the weight transfer function $w_{bin} : [1..N] \rightarrow \mathbb{R}$, that is multiplied with the resulting density. Both the kernel radius function and the weight function can be any arbitrary function based on which bins should stand out. We extend the density function $D(Q)$ of equation (8) such that it also takes a kernel radius $r$ and a subset of objects $O \subseteq \mathbb{O}$ as input, as follows:

![Figure 14](image-url). Density aggregation occurs with a single Compute Density Map block. For each bin the dataset is filtered and a density map is computed using the bin’s parameters. The resulting density maps are then combined using the operator $\bigcirc$. 

\[ D(Q) = \sum_{i=1}^{N} D_i(Q) \]
Where \( r \) is a kernel radius as used by the kernel function \( K \) of equation (5). The density of bin \( i \) for some cell \( q \) is then computed as follows:

\[
D_i(q) = w_{\text{bin}}(i) \cdot D(q, r(i), F_i(\mathcal{O}))
\]

(15)

Where \( \mathcal{O} \) is the entire data set of moving objects, \( r(i) \) is the kernel radius for bin \( i \) and \( F_i(\mathcal{O}) \) is the filter function used to extract the subset of moving objects associated with bin \( i \) from the dataset \( \mathcal{O} \). To get the total density for some cell \( Q \), the individual densities for each bin can be blended together using some operator \( \bigodot \)—see Figure 14. The total density \( D(Q) \) for some cell \( Q \) is then given as follows:

\[
D(Q) = \bigodot_{i=1}^{N} D_i(Q)
\]

(16)

The above aggregation as shown in Figure 14 occurs within a “compute density map” block in Figure 3.

In Figure 15 we show an example of aggregation on a dataset of a single day. We created four bins of six hours dividing the day into four consecutive six hour blocks. We then apply a transfer function that assigns decreasing kernel radius and increasing weights to the bins such that we highlight the activity that takes place in the last six hour block of the day. The three other blocks now serve as a context in

Figure 15. An aggregation of four density maps using addition. The data set of a single day is divided into four consecutive bins of six hours each. Using decreasing kernel sizes and increasing weights we can highlight activity in the last six hours of the day and find unusual activity in the evening.
which we can detect anomalous behavior during the last six hours of the day. We use simple addition as the aggregation operator and visualize the result as a single density map.

We control the density weight transfer function and the kernel transfer function using an editable bar chart where each bar represents the transfer value for a bin—see Figure 16.

### 4.2.2. Binning

For creating the bins and their filters we have created a specialized widget. We first make a selection of two axes from a set of available axes such as time, velocity, vessel type or vessel area. These axes define a square with $x$ and $y$ parameter ranges on which we show the distribution of our data over these ranges using a distribution map. This distribution map indicates the data density for each point.

Figure 17. A number of distribution maps generated for time and velocity (A), Velocity and Vessel Area (B), Time and Vessel Area (C), Velocity and Vessel Type (D) and Time and Vessel Type (E). The axes have between 10 and 64 samples depending on the number of different data elements. Where needed, bilinear interpolation is applied to fill in additional pixels.
on the square and can show where movement activity takes place and as such can help with the creation and selection of bins. As can be seen in Figure 17, we can observe certain patterns forming. For example, in distribution map A it is clear at which velocities the majority of the objects move. Also, in distribution map D one can see which types of vessels tend to be slow and which can go faster. In distribution map E the relation between the size of a vessel and its velocity becomes clear. Larger vessels tend to be slow while smaller vessels can go faster as well.

We create regions on the above discussed square. A region is a two-dimensional range on the axes, which means regions have a rectangular shape on the square. All movements contained within a region are accepted by the filter function $F_t$. Each bin can have zero or more regions. Movements outside all of these regions are not accepted by $F_t$.

![Figure 18](image1.png)

*Figure 18. The binning interface where the selected axes are time and velocity. Three bins have been created with a total of four regions. Note that a bin can have multiple regions and regions of different bins can overlap. A region of the bin marked with the yellow color is selected.*

![Figure 19](image2.png)

*Figure 19. A density map created using the binning of Figure 18*
To make a distinction between the bins, we color code the bins such that the regions of each bin have their own color. The binning interface now looks as shown in Figure 18 for a data set of a single day. The yellow bin contains a region of slow movements during the first four hours of the day and a region of slow movements during the last four hours of the day. The teal bin contains all fast movements and the pink bin contains all movements during the middle eight hours of the day. In Figure 19 we show a density map generated using these bins. We give the yellow bin a normal weight and a medium kernel radius, the teal bin a normal weight and a large kernel radius and the pink bin a low weight and a small kernel radius. We can see, as the distribution map suggests, very little fast movements shown as thick lines.

4.3. Density Combination Operator

If multiple density maps have been generated they can be combined using the density combination operator $\oplus$—see Figure 20. This is a generic operator with one or more density maps as input and as output a single combined density map. The density combination operator is left as a generic operator such that a large variety of functions can be plugged into the operator. This opens up possibilities for multiple density map combination techniques for different purposes. We have defined a weighted addition operator, an anomaly detection operator and an absolute difference operator, which we discuss in the following sections.

4.3.1. Weighted Addition

The most basic density combination operator we have implemented is a weighted addition operator. The density values of the maps are multiplied by a weight $w_i$ assigned to each density map $\mathcal{D}M_i$. The weighted densities are then added into a single density map as follows:

$$D(Q) = \sum_{i=1}^{N} w_i \cdot \mathcal{D}M_i(Q)$$  (17)

Where $\mathcal{D}M_i(Q)$ is the density value for density map $i$ in cell $Q$.

In Figure 21 we show an example of weighted addition of two density maps. The first density map has a kernel radius of 3 kilometer while the second density map has a kernel radius of 300 meter. The density map with the larger kernel has a weight of one and the density map with the smaller kernel has a weight of 20. Using this combination, the first density map shows the general area usage of the
vessels while the second density map, due to its smaller kernel radius, reveals the detailed individual movement patterns of the vessels.

4.3.2. Anomaly Detection

For operators of coastal security systems it may be important to quickly find anomalous behavior. For example, ships that are in areas where there is normally only very little traffic. To achieve this, we require a density map $H$ containing historical data and a second, sparse density map $S$ in which the operator wants to find anomalous behavior defined as vessel movements in the second map that significantly differ from those in the historical map. To have anomalies show up in the combined density map, we defined our anomaly operator as follows:

$$
D(Q) = \max(\tau S(Q) - H(Q), 0),
$$

where $\tau > 0$ is the scaling factor of the sparse map and $S(Q)$ and $H(Q)$ are the density contributions of the sparse and historical density map, respectively, to cell $Q$. Since the historical data will generally span over a longer period of time than the data in the second density map, the densities found in the historical density map will tend to be some factor larger. We use the parameter $\tau$ to find an appropriate scaling factor, which can be seen as the proportion of weights assigned to each density map. The larger the densities in the second density map $S$ are, compared to the historical density map $H$, the sooner a movement pattern will show up as an anomaly.

![Figure 21. The weighted addition of a density map with a larger kernel (3 km) and a density map with a smaller kernel (300 m).](image-url)
In Figure 22 we show an example of the anomaly detection operator. We use a historical density map $H$, generated from a data set of six days, and a second density map $D$, spanning a single day not contained within the dataset of the historical map. The historical density map $H$ is shown as a context in white and the anomalies are shown using a green to red color map. As we can see, the designated shipping lanes show no anomalies since these are heavily used in the historical map $H$. We do, however, see some unusual patterns. The left encircled vessel is following a periodic movement pattern that is unusual in the context of the historical map $H$. It turns out this vessel is a patrol ship of the Dutch navy. The second encircled pattern turns out to be a cargo vessel that is supplying several ocean platforms in the area. Since vessels usually stay well away from ocean platforms, the movements of this vessel show up as an anomaly.

4.3.3. Absolute Difference

To analyze the relation between two density maps, the absolute difference of two density maps could be taken. By showing the absolute difference between two density maps, areas where the behavior of the moving objects between the two density maps differs are highlighted. This can be used to find anomalies as shown in Figure 23. Anomalies show up as areas surrounded by valleys, where the density map that has the higher density changes. This operator highlights anomalies in otherwise already busy areas, but cannot highlight anomalies in areas where there is no activity on one of the maps. We define our absolute difference operator between two density maps $\mathcal{D}M_0$ and $\mathcal{D}M_1(Q)$ as follows:

$$D(Q) = |w_0 \cdot \mathcal{D}M_0(Q) - w_1 \cdot \mathcal{D}M_1(Q)|$$  \hspace{1cm} (19)

Where $w_0$ and $w_1$ are weights associated with each of the density maps.
Chapter 4 – Density Computation

Figure 22. The anomaly detection operator on a historical density map $H$ of six days and a second density map $D$ of a single day. The historical map is shown as a context in white and the anomalies are shown using a green to red color map.

Figure 23. The Absolute difference operator used to detect anomalies corroborated by the same density map using the anomaly detection operator in the lower right image.
Chapter 5. Visualization

If datasets are sufficiently large, a human observer can no longer easily detect or recognize relevant patterns and some analysis tool is required. Since humans can reason and learn more effectively in a visual setting [22], a visual analysis tool can be powerful. In this chapter we describe such a tool and we discuss the visualization techniques employed on our moving vessel dataset. Our density maps are visualized using a continuous or discrete color map and a lighting model applied on a height map. How exactly the color map and the shading are applied depends on the choice of visualization operator, $\otimes$.

We start by describing how and why we convert our density maps in geographic space into density maps in screen space. We then discuss some standard techniques we apply to visually support the human observer, such as scaling, coloring and shading. Finally, we discuss our implementation specific visualization techniques plugged into the visualization operator—see Figure 24.

5.1. Projection

Since our density map has been computed in geographic space and our display surface is planar we need to convert our density map to screen space. We do this by using an equal-area map projection as described in section A.2. It does not matter exactly which map projection is used as long as it is an area-preserving map projection that has a well-defined inverse. We have chosen Lambert’s cylindrical equal-area projection—see section A.2. The main reason for choosing an area preserving projection is that we can project our cells without having to compensate for changes in area of the cell caused by distortion due to the projection function. This would require having to calculate the area of a cell on the earth which is a fairly expensive operation [12].

![Figure 24. An overview of the visualization subsystem. The density maps of the density computation phase are projected into screen space. A normal map and a colored image are generated and fed into the visualization operator which returns the final visualization.](image-url)
Figure 25. A modified black body color map applied to a density map of vessel activity around the harbor of Rotterdam. A continuous color map is used to the left and a discrete color map, with 6 samples, to the right.

Figure 26. A discrete color map with 5 distinct color samples is applied to the density map (Left). Shading is applied to the same density map bringing out more subtle height differences that are lost otherwise (Right).

Figure 27. A density map of vessel activity around the harbor of Rotterdam over a day. Visualization with a linear scale function (left) and with a logarithmic scale function (right). Using the logarithmic scale function gives a clearer image where differences in height are more easily recognizable.
We project our density map by, for each pixel on the screen, applying the inverse projection of equation (48) to find the appropriate geographic coordinates to, in turn, look up the appropriate cell in the density map.

5.2. Color Map

To visualize the densities, we apply a color map to the density map to help distinguish between different values. This color map can either be continuous or discrete with a finite number of samples—see Figure 25. The advantage of a discrete color map is that it shows contours as a side-effect as can be seen in Figure 25. The color map is applied based on a given data range \([r_0, r_1]\). The entire color map is applied to the given range and every density value smaller than \(r_0\), giving the lowest color, and every density value greater than \(r_1\), giving the highest color—see Figure 28 as follows:

\[
clamp(x) = \begin{cases} 
C[0] & \text{if } x < r_0 \\
C \left[ \frac{x - r_0}{r_1 - r_0} \right] & \text{if } x \in [r_0, r_1] \\
C[1] & \text{if } x > r_1 
\end{cases}
\]  

(20)

Where \(C[x]\) is a color map defined for the range \(x \in [0,1]\).

Figure 28. The color map \(C\) (below) is clamped to a given range \([r_0, r_1]\) on the entire data range (above). Values outside the given range are clamped to the extreme colors.

5.3. Shading

We model our density map as a height-map where we interpret each density in a point as a height and apply lighting to the height map to give the illusion of actual height differences. Shading reveals subtle differences in density that may otherwise not be visible. This is especially the case if a discrete color map is used—see Figure 26. Also, the height map used to derive the lighting does not necessarily have to be the same as the density map our color map is applied to. This means the lighting becomes another way of visualizing data on top of the color maps. We use a variation of the Phong Shading Model [15] with a single white light source that is located at infinity; Specular highlights are not used. The lighting is applied by the visualization operator in Figure 24.

5.4. Scaling

The density range in the density maps may be very large, especially around stationary vessels. To bring the actual density range into a smaller, more manageable, range we introduce a scaling function \(S\). This scaling function \(S\) controls the way the density is scaled which can either be linear with \(S(x) = x\) or logarithmic with \(S(x) = \log(x + 1)\). Using a logarithmic scale, it becomes easier to see height differences when applying a color map.
5.5. Visualization Operator

In the visualization operator—see Figure 24—the visualization image is constructed using the density map generated before by the density combination operator. In the following sections we discuss several possible visualization operators and their uses. For some operators, such as the single target visualization or the aggregation visualization, it is necessary for the visualization operator to have direct access to individual density maps instead of just the result of the density combination operator. For these special cases the required individual density maps are projected into screen space as well—see Figure 30.

5.5.1. Combined Visualization

The simplest possible operator is the combination operator that simply applies the color map to the result of the density combination operator and applies lighting using a normal map derived directly from the same density map. This allows us to view the result of the density combination directly and unaltered. See Figure 29.

5.5.2. Single Target

Willems et al. [1] visualize their density maps by combining two density maps, a density map with a large kernel for the overall area usage and a density map with a smaller kernel for details. The color map is applied only to the density map with the large kernel while the lighting is derived from the combination of both density maps. We allow a similar operator. We use the result of the density combination operator to compute our normal map, but for the color map part we allow the selection of a single density map after projection. The visualization operator applies the color map to this density map and then computes the lighting based on the values of the density map generated by the density combination operator.

In Figure 29 we show a comparison between the normal combined visualization operator and the single target visualization operator on a density map with a kernel radius of one kilometer and a density map with a kernel radius of 300 meter using a weighted addition density operator. On the left we show the combined visualization of the result of the density operator and on the right we show a single target visualization with the color map applied only to the density map with the larger kernel and the lighting applied on the result of the density operator. The contours in the right image are
smoother than the contours in the left image since they are not affected by the smaller and more abrupt height differences caused by the density map with the smaller kernel radius.

5.5.3. Aggregation

An aggregated density map is a combination of multiple density maps based on bins—see section 4.2. If we combine these density maps using the density combination operator and use the combined visualization operator—see section 5.5.1—the distinction between different bins may become unclear or even be lost. This is especially the case if the different bins have similar parameters. As a solution we apply a different color map to each density map. This means that, beside the density map parameters, the bins also get additional visualization parameters such as a color map. We use the bin color codes introduced in Section 4.2.2 to generate color maps that linearly go from grey for low densities to the original color code for high densities by changing the saturation of the color.

After combining these different colored images into a single colored image, the result must still be readable. We propose several different techniques with varying results: color mixing using some opacity transfer function, a maximum operator and a pixel block operator.

**Color Mixing**

When combining density maps we not only want to see the top layer, but also the layers beneath. In this respect, the most straightforward way of combining or blending the colored images is by using translucency. Each bin \( i \) has an assigned weight value \( \alpha_i \) in the range \([0,1]\). For each pixel \( P \) in the resulting colored image \( C' \), we want the color to be the proportional mix of the colored density contributions of all bins that have some non-zero density contribution in pixel \( P \). The density maps are then combined as follows:

\[
C'(P) = \sum_{i=0}^{N-1} \alpha_i C_i(P) \tag{21}
\]

Where \( N \) is the number of bins, \( \sum \alpha_i = 1 \) and \( C_i(P) \) is a colored pixel representing the density value of bin \( i \) in pixel \( P \) using the color map of bin \( i \).
While this method works fine for a small number of bins, i.e. three, the visualization becomes unclear and messy with a larger number of bins. It is possible, however, to use color blending techniques to help to perceptually distinguish between pure colors and mixed colors. As an example see Figure 31, where two visualizations are shown using color mixing. The left image is a combination of a red, blue and purple density map while the right image is a combination of only a red and a blue density map. Notice how the distinction between the purple that is a mix between the red and the blue density map and the purple color of the third density map is unclear. While for this simple example a color map can be found that does not have this mixing problem, i.e. simply red, green and blue, this effect becomes unavoidable with a larger number of density maps.
Maximum
To solve the problem of color ambiguity as demonstrated in the previous section, we introduce another way of blending the colored density maps. We show only the maximum density value. This means that per pixel on the screen we show only the color of the bin that has the highest density at that particular pixel.

In Figure 32 we show a comparison between combining colored density maps using color mixing on the left and combining colored density maps using the max operator on the right. The red density maps are ships in the morning, the blue density map are ships in the afternoon and the purple density map are ships at night. Note how there is no longer any ambiguity in the displayed colors. All three density maps have the same kernel radius and the same density multiplier which allows us to see in which bin the activity is the highest per point in space. We can now see patterns emerging that remain hidden when the color mixing combination technique is used.
This method works particularly well for finding hotspots of activity per time of day or per velocity.

Pixel Blocks
While the color mixing method shows all data, but suffers from ambiguity, the maximum method shows a clear distinction between colors, but does not show all data. To this end we introduce a third method that both shows all data and does not suffer from color ambiguity. We call this method pixel blocks, which is conceptually similar to the attribute blocks of Miller et al. [29].

We divide our screen space into a number of blocks of $s \times s$ pixels. In each of these blocks one or more density maps may overlap. If a single density map overlaps the pixel block we simply show that colored density map. If multiple density maps overlap the pixel block, however, we choose one of the overlapping colored density maps at random. This shows a clear distinction between areas with single values and areas with mixed values. This results in an image as shown in Figure 33, C.
Figure 33. Four density maps colored with red, green blue and purple, each representing a different 4-hour block during a single day, combined using opacity (A), the maximum operator (B) and the pixel block operator with a pixel block size of two pixels (C).
Chapter 6.

Implementation

To implement the parallelization as discussed in Chapter 4, we have chosen to utilize the Graphics Processing Unit (GPU). The GPU is a specialized piece of hardware intended for high performance computer graphics and is available in most modern computer systems. One of the main strengths of the GPU is that it can process many elements in parallel. Due to this structure, the GPU has become a popular instrument for general purpose computing in the last few years. As it turns out, the GPU has more raw processing power than the central processing unit (CPU) when computations can be run many times in parallel. This field of computing using the GPU is called General-purpose computing on graphics processing units (GPGPU). The advent of GPGPU has not gone unnoticed for the major GPU producers. For example, the Nvidia Corporation has developed a general purpose language specifically for their cards called CUDA [19]. More recently, an open standard language has been developed called OpenCL [20], which can run on video cards from multiple vendors.

In this chapter we go into more detail about how we implement the density map computation and visualization techniques described in the previous chapters. We start off with an introduction on the architecture and functionality of modern GPUs which is required to understand some of the steps we have taken in the architecture of our system. If the reader is sufficiently knowledgeable in the field of graphics hardware, section 6.1 can be skipped in its entirety. Next, we explain how we have parallelized the algorithm and why we have made the choices we have made. Following this, we divide the discussion of our implementation into two parts: The density computation part and the visualization part—see Figure 3. In sections 6.3 to 6.5 we describe the density computation and the density combination operator in detail. In sections 6.6 and 6.7 we describe the visualization implementation and the visualization operator in detail.

6.1. Introduction to the GPU

The graphics processing unit processes graphical elements in a pipeline structure. The relevant data, such as raw triangles, is supplied to the hardware after which it is processed by a sequence of dedicated hardware units each designed for a specific purpose. These dedicated hardware units can process many requests in parallel. The first generations of GPUs or graphics cards had what is called a fixed function pipeline. This means the hardware in the pipeline has a single specific purpose and cannot do anything other than that specific purpose. Later generations of graphics cards introduced

i. Pixels, Fragments and Texels

There are three related, yet sufficiently distinct, types of colored points that are often confused with each other: Pixels, Fragments and Texels. A pixel is simply a colored block as seen on the screen. A fragment, however, has additional context besides color and position. A fragment can have attributes such as depth, stencil value, alpha value, window ID, several interpolated attributes (such as color, texture coordinates, etc.) and more. Depending on depth-culling or stencil-culling a fragment may never make it to the screen whereas a pixel can be the combination of multiple fragments. While the term pixel shader is often used to refer to a fragment shader, it is technically incorrect.

Texel is an abbreviation of the term texture element and is a colored block sampled from a texture. This can be a colored block directly from a texture or an interpolated colored block from multiple colored blocks in the texture depending on the location the texel is sampled from.
programmable hardware that could run small special purpose programs called *shaders*. First, cards became available that allowed shaders to process vertices called *vertex shaders*, then cards became available that allowed shaders to process *fragments*—see textbox i—called *fragment shaders* and the latest generations support *geometry shaders*, which are shaders that can process incoming geometry and transform it into different or more complex geometry. Or in other words, a geometry shader is a program intended to generate new *geometry* on the GPU by *emitting* vertices based on some input sequence of vertices.

Initially, each of these types of shaders had their own specific hardware units with specific instruction sets designed to process either vertices or fragments. In the most recent generations of graphics cards, however, a new type of architecture is being used without any distinction between the different types of shaders: There is a single type of processing unit called a *shader model* with a single unified shader instruction set that can process all types of shaders. These shader models are grouped in so-called *Thread Processing Clusters*—see textbox ii. This architecture is called the *Unified Shader Architecture*—see Figure 34.

![Figure 34. A schematic overview of the unified shader architecture of the Nvidia GeForce 200 series [18][25] as used for the implementation of the system described in this thesis. The GPU contains 10 Thread Processing Clusters used for processing (multiple) shader programs of any type. Using a scheduler these threads can be scheduled to a Thread Processing Cluster (TPC) where needed. Three different types of threads can be issued: Vertex threads, Geometry threads and fragment threads.](image-url)
Chapter 6 – Implementation

6.1.1. Unified Shader Architecture

In this section we give a general overview of the architecture of modern graphics hardware, the unified shader architecture—see Figure 34. We focus on the elements that are important for understanding our implementation and avoid irrelevant details.

As input the GPU takes a set of primitives such as triangles including attributes such as textures, texture coordinates and color. The input is processed by a dedicated, fixed function, hardware unit called the Input Assembler—see Figure 35. This unit assembles the triangle data into a format that is easy for

ii. Thread Processing Cluster

A Thread Processing cluster is a hardware unit that contains three streaming multiprocessors (SMs) that can handle up to 1024 concurrent threads each. A single TPC can perform multiple instructions for multiple threads in one clock cycle. Each of these SMs contains 8 single Stream processors (SPs). Using these SPs a single SM can perform a single instruction on multiple threads in one clock cycle. The Instruction Unit creates, manages, schedules and executes the threads. The GeForce 200 series graphics cards have 10 TPCs [18]. The SMs contain specialized Arithmetic Logic Units (ALUs) specialized to handle floating point operations or more extended operations such as sin, cos, exp and rcp. Per clock cycle, a number of these operations can be issued while it takes eight cycles to complete such an operation [25].

Figure 36. A schematic overview of a single Thread Processing Cluster in the GeForce GTX 200 series architecture. Each TPC contains four Streaming Multiprocessors (SMs) and share a single cache. Each SM has an Instruction Unit that manages the threads that run on the Stream Processors (SPs) of the SM. Each SM has 8 SPs which share local memory [18]. This comes to a total of $10 \cdot 3 \cdot 8 = 240$ SPs.
the GPU to understand and process. For each vertex of the primitives, a vertex shader is scheduled on a Thread Processing Cluster (TPC), which is a dedicated hardware unit that can process all kinds of shaders multiple times in parallel. Vertex shaders are mostly used to transform vertices and set up parameters required in the next stages.

The output of the vertex shader program is sent on to the optional geometry stage. In the geometry stage the incoming primitives are processed with, as output, new primitives. A geometry shader has a single primitive as input, but can output zero or more primitives. In the next stage the transformed primitives are sent on to the Triangle Setup, Rasterizer and z-cull. In the triangle setup stage triangles that face away from the camera or are outside of the viewing frame are rejected. The rasterizer is a hardware unit that turns triangles into a set of fragments. This is also where attributes of the vertices are interpolated to attributes of the fragments. The generated fragments are then tested against the depth value recorded in the depth buffer and the fragments are rejected if needed in the z-cull stage. The output is then written to the video memory using the Render Output Units.

Geometry information such as vertices and textures are stored in the video memory on board of the video card. The host then only needs to send one call to start processing a batch of geometry without having to stream the entire geometry from conventional memory to video memory.

6.1.2. Environment and Additional Terminology

In the implementation of this thesis we use the OpenGL graphics library to communicate with the graphics card. We set up parameters and run shaders to perform our calculations. The (intermediate) results of these calculations are written to a Frame Buffer which is connected to some texture using the Frame Buffer Object (FBO) extension of OpenGL. Both the frame buffer and frame buffer objects are two-dimensional arrays or frames in video memory.

We use “C” for Graphics [21] for programming our shaders, which is a C-like language created by the Nvidia Corporation. The Cg language is in structure very similar to C and supports all language constructs supported by C. Cg, however, has additional language constructs and also some vector and matrix types. A more detailed description of these language constructs accompanies the discussion on our shaders in Appendix D. Cg allows a number of binding semantics which allows us to bind input or output parameters to certain specialized hardware registers as shown in Figure 37. The only way to pass on parameters between shaders is through these binding semantics. This means that we can only assign attributes directly to vertices. Cg shaders can also receive parameters directly from the Cg API. These parameters, however, do not vary per vertex but are the same for all vertices.

float shaderFunction( uniform samplerRECT densityMap0 : TEXUNIT0, 
in float2 texCoord : TEXCOORD0 
) : COLOR0

Figure 37. An example of binding semantics (red) in Cg. An input texture (samplerRECT) is received from the TEXUNIT0 binding semantic representing the first texture unit and a two-dimensional vector is received from the TEXCOORD0 binding semantic representing the first texture coordinates. The output of the function is a floating point value that is stored in the first color register using the COLOR0 binding semantic.

6.2. Parallelization

As we discussed in Chapter 4, the density computation can be structured using two fundamentally different approaches. In the first approach the density computation is performed per line segment where, as stated in Section 3.1, a line segment is the movement between two consecutive points. For each line segment, an oriented bounding box is constructed and for each cell Q intersected by the bounding box, the density generated by the line segment in that cell is computed and added to the density of the cell Q. The line segments can be processed independently and in parallel; however, adding the computed density to the cells requires some degree of synchronization. We call this parallelization over the line segments—see Algorithm 1. In the second approach the density in each cell Q is evaluated: The movements of all objects in \( Q \) intersecting the cell are looked up in some data struc-
Streaming from conventional memory

Streaming from video memory

Figure 38. Streaming moving data from the conventional memory versus streaming movement data from video memory. The connection between the host and the GPU, the PCI-express bus, is the main bottleneck. The CPU must first request the data from conventional memory, then stream it through the bottleneck and only then the GPU can start rendering. When the movement data is present in video memory the host only needs to send a render request for the movement data and the movement data is streamed directly from the video memory using a much faster connection.

6.3. Implementation on the GPU

In this section we discuss the implementation of the methods described in Chapter 4 on the GPU. We start by giving a general overview of our implementation and then discuss each part individually.

The density contribution to cell \( Q \) of these object movements are added. All cells can be processed independently and in parallel. We call this parallelization over the cells—see Algorithm 2.

For our system we have chosen the first approach where the density computation is performed per line segment. The second approach, parallelization over the cells, would require some data structure such as a quad-tree or a \( kd \)-tree to quickly find the line segments that intersect the cells, whereas in the first approach, each line segment only needs to be processed once. In the first approach we can simply traverse our line segments once in any order without having to maintain a complicated data structure. This also makes adding and removing line segments from the data structure trivial and, more importantly, we can utilize the regular graphics pipeline.

While a data structure such as discussed above can be constructed and maintained using shaders, it is cumbersome and not as efficient as an implementation in CUDA would be [27]. This means, to implement parallelization over the cells, would require some data structure such as a quad-tree or a \( kd \)-tree to quickly find the line segments that intersect the cells, whereas in the first approach, each line segment only needs to be processed once. In the first approach we can simply traverse our line segments once in any order without having to maintain a complicated data structure. This also makes adding and removing line segments from the data structure trivial and, more importantly, we can utilize the regular graphics pipeline.

One of the disadvantages of using the graphics pipeline approach is a loss of flexibility. Since the line segments are processed completely independently, finding and visualizing relative properties of moving objects, such as distance or orientation compared to other moving objects, is a lot more difficult without a spatial subdivision data structure. Another disadvantage is the floating point precision. On current GPUs 64-bit (double precision) floating point variables are slow [18]; moreover, there is currently no support for 64-bit precision textures. This would not be a problem when using a more general language such as CUDA, since modern GPUs do properly support 64-bit floating point precision [18] in the processing units. We have found, however, that 32-bit floating point precision, although there are some minor precision problems, suits our needs.
We store our track data in video memory using so-called Vertex Buffer Objects as a set of vertices. This allows the system to directly address the data from video memory without having to stream it from physical memory every time the density map is computed—see Figure 38. Each track can still be addressed separately which means any filtering can be performed on the CPU if required. The tracks are stored as a sequence of four-dimensional vertices where the first two coordinates are the geographic position of the point, the third is the time instance of the point and the fourth is its velocity.

We implement Algorithm 1 in parallel using the GPU. We distribute our algorithm over the hardware components discussed in the above sections as given in Algorithm 3.

Vessel density() on the GPU

<table>
<thead>
<tr>
<th>Operation</th>
<th>Host</th>
<th>Geometry Shader</th>
<th>Rasterizer</th>
<th>Fragment Shader</th>
<th>Render Output Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $o \in \text{Objects}$ do in parallel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for $i \in \text{Trajectory Segments}[0, N]$ do in parallel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_0 \leftarrow p_o(t_i); p_1 \leftarrow p_o(t_{i+1})$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Construct OBB around line segment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for $(u, v) \in \text{OBB}$ at distance $r$ to $p_0p_1$ do in parallel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q \leftarrow \text{Center of the cell} (u, v)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C \leftarrow C_{p_0p_1}(q)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D(u, v) \leftarrow D(u, v) + C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 3. The density map computation algorithm given in pseudo-coded and divided over the hardware based on the architecture shown in Figure 34.

We first construct a grid in the form of a texture. This texture represents the subdivision of our geographic space into uniform cells with equal area. We render our movements to this texture with the texture ultimately representing our density map. When we start computing our density map, our application, the host, sends a render request per object $o$ to the video card. We render a polyline consisting of all the points of the object’s movement. For each line segment of the polyline we need a set of pixels within the kernel radius with geographic coordinates to represent our cells. To this end, each line segment is processed by a geometry shader. The geometry shader takes care of constructing and parameterizing an oriented bounding box (OBB) that tightly fits all points at distance $r$ from the line segments—see B in Figure 39. Using the Rasterizer our OBB is then subdivided into a set of fragments—see C in Figure 39. This set of fragments represents the set of cells that have some non-zero density contribution.
density contribution of the moving object $o$. For each of these fragments, a fragment shader is run that computes $C_o(q)$ where $q$ is the coordinates in the center of the area represented by the fragment—see D in Figure 39. A fragment is only rasterized if its center point falls within the OBB. Since we sample only the central point of our cells this is exactly the behavior we want. The resulting density contribution $C_o(q)$ is blended, using addition, into the texture representing our density map using a render output unit.

In the next sections we discuss the above described steps in more detail. We start by describing how we construct our grid and follow by describing the geometry shader that constructs the OBB, the fragment shader that computes the density contribution per cell including more detail on the kernel and the integral, how the shaders come together, how we compute subsets of time or velocity and finally, how we treat the special case of stationary vessels.

### 6.3.1. Constructing the Grid

The division of the geographic space depends on two parameters, the range of the area we wish to divide $[\lambda_0, \lambda_1] \times [\phi_0, \phi_1]$ in longitude $\lambda$ and latitude $\phi$ and the desired size $cs$ of the cells in km. We compute the width and height of our grid in number of cells as follows:

$$
\text{width} = \frac{D_{gc}(\{(\lambda_0, m_\phi), (\lambda_1, m_\phi)\})}{cs}
$$

$$
\text{height} = \frac{D_{gc}(\{(m_\lambda, \phi_0), (m_\lambda, \phi_1)\})}{cs}
$$

Where $D_{gc}$ is the geographic distance function given by equation (45) and $m_\phi$ and $m_\lambda$ are the medians of the latitude and longitude respectively, given as follows:

$$
m_\phi = \phi_0 + \frac{\phi_1 - \phi_0}{2}
$$

$$
m_\lambda = \lambda_0 + \frac{\lambda_1 - \lambda_0}{2}
$$

We have chosen the above procedure such that for relatively small areas, such as the Dutch coast (approximately $400 \times 400$ km), all cells have approximately have an area of $cs^2$; however, due to distortion of the cell size in the north to south direction—since not every cell of equal area will have the same size in latitude $\phi$—we would need a more precise method for larger areas.

### 6.3.2. Constructing the Bounding Box

In the geometry stage we construct an oriented bounding box (OBB) containing at least the cells that receive a non-zero density contribution. Since we have chosen a kernel with finite support and radius $r$, we can construct a bounding box at a minimal distance of $r$ to each point on the line segment. We construct our bounding box $v_0, v_1, v_0, v_1$—see Figure 40—in the index space of the cells using fractional indices. We call this space Grid space. The per vertex input for our geometry shader are the points $p_0$ and $p_1$ in geographic space, the time and velocity of both points $(t_0, v_0)$ and $(t_1, v_1)$. The geometry shader takes an additional number of constant parameters such as the kernel radius $r$ and precomputed constants such as the kernel size $(k_x, k_y)$ in number of cells in both the $x$ and the $y$ direction. Our geometry shader works as shown in Algorithm 4.

In our geometry shader we project our points $p_0$ and $p_1$ to Mercator space using the projection of equation (49). We then compute both the Rhumb length of the line segment using equation (51) and the length of the line segment in Mercator space. Both these lengths are required for compensating the distortion in line length caused by the Mercator projection in equation (11). Recall that we require Mercator space because we have assumed vessels move from point to point with constant bearing and these movements, Rhumb lines, are straight lines in Mercator space and as such easy to parameterize. We then compute the direction $N$ of the line between points $p_0$ and $p_1$ in geographic space to
get the orientation of the OBB. Using the rhumb length computed earlier we now compute the initial 
velocity and the acceleration. The points and are then projected onto grid space such that we can place our oriented bounding box correctly on the grid. We do this using the following equa-
tion:

\[ p'' = \text{Project}(p) = \frac{p - \text{origin}}{DC} \quad (24) \]

Where \( \text{origin} \) are the geographic coordinates of the lower left corner of the grid where the grid coordinates are \((0,0)\) and \(DC\) is the cell size in degrees in both the longitude \(DC_x\) and latitude \(DC_y\), direction derived from the cell size \(cs\) in kilometers. Both \( \text{origin} \) and \( DC \) are additional, pre-
computed parameters to the geometry shader. In the context of shaders, vector division is under-
stood to be element-wise division, i.e. \(A/B\) is equal to \((A_x/B_x, A_y/B_y)\). Using the orientation \(N\) of 
the bounding box set to the length of the kernel radius in \(N''\), we construct four vertices \(v_x1, v_x2, \)
\(v_x3\) and \(v_x4\) in grid space that form our OBB. These vertices are used as positions for projecting 
our OBB to the grid. We now also compute the geographic positions of these four vertices in \(gc1, gc2, \)
\(gc3\) and \(gc4\) respectively. This allows us to use the hardware to assign the appropriate geographic 
positions to the fragments through interpolation. We also want to quickly find the closest position on 
the line segment nearest to any given fragment. To do this we pass a line parameter—see equation 
(2)—to our fragments that runs from \(0\) to \(\|p_0p_1\|\). Since we can only assign values to vertices we 
cannot simple pass on the value \(0\) and \(\|p_0p_1\|\). We must find the shortest distance \(L^+\) of the points 
\(p_0\) and \(p_1\) to the edges such that we can pass on the parameters \(-L^+\) and \(\|p_0p_1\| + L^+\) such that the 
line parameter is correctly interpolated over the dashed line running through points \(p_0\) and \(p_1\) in
Figure 40. Finally, we emit our four vertices with a number of flat attributes, which are attributes that are not interpolated and are the same for every fragment and the geographic position and the line parameter which are interpolated over the fragments.

6.3.3. Computing the Density Contribution

The bounding box, emitted by the geometry shader described in the previous chapter, in the form of a quad $\mathbf{vx}_3, \mathbf{vx}_2, \mathbf{vx}_3, \mathbf{vx}_4$, is divided into fragments and the geographic position and line parameter assigned to the vertices are interpolated over these fragments by the rasterizer—see the emitted parameters in Algorithm 4. A fragment is generated by the rasterizer for each cell that has its center inside the quad. Since we sample the center of the cell, this is exactly the behavior we want. We store the density maps in 32-bit floating point textures where each pixel contains a single 32-bit floating point value—its density—and represents a single cell. We use Frame Buffer Objects to render to these textures.

In the following sections we introduce and discuss the kernel function we use in our implementation, discuss how we compute the integral of equation (5) in Chapter 3 and explain the inner workings of computing the density contribution within the fragment shader.

Kernel Function

We have one important requirement for our kernel function $K(x, r)$ with radius $r$: $K(x, r)$ must be equal to 0 for all $|x| \geq r$ such that the density contribution of a convolved line segment is zero to cells outside of the kernel radius. We based our kernel function on the Gaussian function. However, any other radially symmetric kernel with finite support can be used, such as the cone kernel used by Willems et al. [1].


Geometry Shader

```pseudo
{ Compute the rhumb length } 
$L \gets \mathbf{p}_0 \mathbf{p}_1$
{ Compute the direction vector of $\mathbf{p}_0 \mathbf{p}_1$ in geographic space } 
$N \gets \frac{\mathbf{p}_1 - \mathbf{p}_0}{\|\mathbf{p}_1 - \mathbf{p}_0\|}$
{ Compute initial Velocity and Acceleration using the movement model of Section 3.1 } 
$v \gets \frac{\|L\|}{t_1 - t_0} \frac{v_1 - v_0}{t_1 - t_0}; a \gets \frac{v_1 - v_0}{t_1 - t_0}$
{ Project points to Mercator space } 
$\mathbf{p}_0' \gets M_x(p_0); \mathbf{p}_1' \gets M_x(p_1)$
{ Compute Mercator Length } 
$L_m \gets \|\mathbf{p}_1' - \mathbf{p}_0'\|
{ Project points to Grid space } 
$p_0'' \gets \text{Project}(p_0); p_1'' \gets \text{Project}(p_1)$
{ Construct direction $N'$ of line segment in grid space using $N$ and kernel size $(k_x, k_y)$ in cells } 
$N'' \gets (k_x \cdot N_x, k_y \cdot N_y); \mathbf{B}' \gets (k_x \cdot N_y, k_y \cdot -N_x)$
{ Construct vertices $\mathbf{vx}_1$ in grid space based on $p_0''$, $p_1''$ and $N''$ } 
$\mathbf{vx}_1 \gets p_0'' - N'' - \mathbf{B}''; \mathbf{vx}_2 \gets p_1'' + N'' + \mathbf{B}''; \mathbf{vx}_3 \gets p_1'' + N'' + \mathbf{B}''; \mathbf{vx}_4 \gets p_0'' - N'' + \mathbf{B}''$
{ Construct vertices $\mathbf{gc}_1$ in geographic space using $p_0$, $p_1$ and $N$ } 
$\mathbf{gc}_1 \gets \mathbf{p}_0 - N - \mathbf{B}; \mathbf{gc}_2 \leftarrow \mathbf{p}_1 + N - \mathbf{B}; \mathbf{gc}_3 \leftarrow \mathbf{p}_1 + N + \mathbf{B}; \mathbf{gc}_4 \leftarrow \mathbf{p}_0 - N + \mathbf{B}$
{ Compute line extents in geographic space } 
$L^+ \leftarrow ||N''|| \frac{L}{\|p_1'' - p_0''\|}$
{ Assign flat parameters } 
$f lat \leftarrow (p_0, \mathbf{p}_1, t_0, t_1, v, a, L, L_m)$
{ return $\mathbf{vx}_1, \mathbf{vx}_2, \mathbf{vx}_3$ and $\mathbf{vx}_4$ with non-interpolated (flat) and to be interpolated parameters } 
emit $\mathbf{vx}_1$ with flat and $(\mathbf{gc}_1, -L^+)$ as interpolants 
emit $\mathbf{vx}_2$ with flat and $(\mathbf{gc}_2, L + L^+)$ as interpolants 
emit $\mathbf{vx}_3$ with flat and $(\mathbf{gc}_3, L + L^+)$ as interpolants 
emit $\mathbf{vx}_4$ with flat and $(\mathbf{gc}_4, -L^+)$ as interpolants
```
Since the Gaussian function has infinite support we have to explicitly make the Gaussian finite. This gives us the following kernel function $K$ with radius $r$:

$$K(x, r) = \begin{cases} 
    e^{-4(x/r)^2} - \frac{|x|}{r} e^{-4} & \text{if } |x| < r \\
    0 & \text{otherwise} 
\end{cases} \quad (25)$$

To avoid having to compute the Gaussian kernel a large number of times, we store the function in a normalized lookup table and linearly interpolate in between the values where necessary. Not only does this decrease computation time, it also has the advantage that the kernel function can be easily changed to any other radial, finite kernel function without having to change anything other than the contents of the lookup table.

**Integration Bounds**

Since our kernel has finite support; for any point $q$ and line segment $p_0p_1$ we only need to consider the segment of $p_0p_1$ that falls within the kernel radius with respect to point $q$—see Figure 42. We know that any part of the line segment outside of the kernel radius will have a contribution equal to zero. This allows us to construct tighter integration bounds for the integral of equation (5) by eliminating the part of the line segment that we know will not contribute to the density at point $q$. Let point $q'$ be point $q$ projected onto the line through $p_0p_1$, then we can find the integration bounds by moving point $q'$ over the line by the radius of the kernel projected onto line $p_0p_1$—see Figure 42. We express a point $q''$ on the line $p_0p_1$ as follows:

$$q'' = p_0 + \frac{x}{\|p_1 - p_0\|} (p_1 - p_0) \quad (26)$$

Let $x'$ be the parameter for which equation (26) gives point $q'$. To get the correct interval over the line segment that is at a distance smaller than the kernel radius, we compute the projected radius $r'$ given by the following equation according to the Pythagorean Theorem:

$$r' = \sqrt{r^2 - \|q' - q\|^2} \quad (27)$$

Using the projected radius $r'$ we now get the parametric interval $[x' - r', x' + r']$ on $p_0p_1$ containing exactly the points on $p_0p_1$ with a distance equal to or smaller than $r$ to point $q$. To get the corresponding bounds in time we use the inverse of equation (1) which is given as follows [1]:

$$t(x) = \begin{cases} 
    t_0 + \frac{-v + \sqrt{v^2 + 2ax}}{a} & \text{if } a \neq 0 \\
    t_0 + x/v & \text{if } a = 0 
\end{cases} \quad (28)$$
Since our movement for a line segment \( p_0 p_1 \) is only defined between its start point in time \( t_0 \) and its end point in time \( t_1 \), we clamp the bounds resulting from equation (28) to the range \([t_0, t_1]\). We now get the following convolution integral:

\[
C_0(q) = \int_{\min(t_0, t(x' + r'))}^{\max(t_0, t(x' - r'))} K(||p(t) - q||, r) dt
\]  
(29)

**Integral Approximation**

We numerically approximate the above integral of equation (29) in \( N \) discrete steps using Simpson’s Rule [13] giving us the following formula, where we shorten \( K(||p(t) - q||, r) \) to \( f(t) \) and \( t(x' - r) \) to \( t_0 \):

\[
C_0(q) \approx \sum_{i=0}^{N-1} \frac{\Delta t}{6} \cdot f(t_0 + i \cdot \Delta t) + 4 \cdot f(t_0 + i \cdot \Delta t + \Delta t/2) + f(t_0 + (i + 1) \cdot \Delta t)
\]  
(30)

Where \( \Delta t \) is the discrete time step given by \( \Delta t = (t(x' + r) - t(x' - r))/N \). While Simpson’s rule is more complex in itself than for example the trapezoidal rule, it requires significantly less integration steps to get numerically similar results and as such is more efficient. We have found that a constant number of steps of \( N = 10 \) is sufficient to get an accurate approximation without visible artifacts.

**The Fragment Shader**

The density contribution is computed in the fragment shader in Algorithm 5. Recall that the fragment shader receives the following input parameters from the geometry stage: The points \( p_0 \) and \( p_1 \) projected to Mercator space, the initial velocity \( v \), the acceleration \( a \), the time instants \( t_0 \) and \( t_1 \) of the points, the interpolated line parameter \( x \), the Mercator line length \( L_m \), the Rhumb line length \( L \) and the interpolated geographic position of the fragment \( gc \).

We first find the direction of the line segment in Mercator space by computing the normalized direction \( \mathbf{N}_m \) of the line segment projected into Mercator space. We then compute the Mercator correction factor \( m_f \) to project the geographic line parameter into Mercator space—see Figure 10. Using the normal and the correction factor, we find the closest point to the fragment on the line segment and compute the distance \( d \) between these two points. If this distance \( d \) is larger than the kernel radius \( r \) we know there is no point on the line segment that can contribute a non-zero density to our fragment and as such, the resulting density contribution must be zero. If this is the case we discard the fragment, which immediately halts the fragment shader and destroys the thread running the instance of the shader. If the distance \( d \) is smaller than the kernel radius \( r \) we know there can be
some non-zero density contribution from a part of the line segment. We then use equations (27) and
(28) to find tighter integration bounds. Within these bounds we then compute the density contribu-
tion of the line segment to the fragment using Simpson’s rule—see equations (5) and (30).

\[
\text{Algorithm 5. The density contribution computation in the fragment shader shown as Pseudo-code.}
\]

{ Compute the normalized direction in Mercator space }
\[
N_m \leftarrow \frac{P_i - P_0}{|P_i - P_0|}
\]
{ Compute the line parameter correction factor }
\[
mf \leftarrow \frac{L_m}{L}
\]
{ Compute the distance to the line segment using equation (46) }
\[
d \leftarrow D_{bc} (M_p^{-1}(P_0 + mf \cdot \max(0, \min(L, x)) \cdot N_m), gc)
\]
{ Discard fragment if it falls outside of the range of the kernel }
\[
\text{if } d \geq r \text{ then discard}
\]
{ Find integration bounds } \( t'_0 \) and \( t'_1 \) using equations (27) and (28)
\[
r' \leftarrow \sqrt{r^2 - d^2}
\]
\[
t'_0 \leftarrow \max(t_0, t(x - r')); t'_1 \leftarrow \min(t_1, t(x + r'))
\]
{ Compute density in \( N \) steps applying Simpson’s rule to } \( K(\|p(t) - q\|, r) \) \( \text{in the range } \left[ t'_0, t'_1 \right] \}
\[
\text{return density}
\]

6.3.4. Putting it all Together
As mentioned before, we render our density contributions to a texture representing our density map.
The density contributions generated by the fragment shaders are added to the density map using
additive blending. Since we use a floating point texture, we are not confined to a density range of
[0, 1] as is the case with normal textures, but we can use the entire range of the floating point type.

To manage the user interface and the shaders we have implemented an application. In Appendix C
the architecture of this implementation is described.

6.3.5. Computing Subsets of Time or Velocity
In the implementation discussed above, it is trivial to compute a subset of space. A smaller geographic
area can be selected and the graphics hardware takes care of culling bounding boxes that fall outside
of the selected region or even individual fragments that fall outside of the selected region. We also
want to divide our datasets on different quantities for the sliding time window method and for densi-
ty map aggregation. For attributes that remain constant for a vessel such as size or ship type the ob-
jects can be filtered by the CPU. A render request is simply only sent for those objects with the de-
sired attributes. Dividing the dataset on continuous quantities that vary over the movement per ob-
ject is a bit more involved.

Separating Time
If we want to divide our data set over time or want to compute only a certain interval in time it is
possible that object movements fall partially outside of this window and even line segments are pa-
tially outside of this window. The graphics hardware, being optimized for geometry without any no-
tion of time, cannot directly help us here, so we must find another way.

We can discard entire objects using the CPU when the entire movement path of a given object falls
outside of the time window, similarly, we can also discard individual line segments within a geometry
shader. This can be done by not emitting any vertices if both points are outside of the time window. If
one of the points is outside of the time window and the other is inside the time window we know the
cutoff point in time of the time window must be somewhere along the line segment. We can compute
only the part of the movement that is inside the time window by adapting the integration bounds as
follows for a time window \( [t_a, t_b] \) where \( t_a \leq t_b \):
Applying the time weight function $w_t$—see section 4.1—we now approximate the integral by applying Simpson’s rule to $w_t(t) \cdot K(\|p(t) - q\|, w_t(t) \cdot r)$ in the range $[t''_0, t''_1]$. To avoid the kernel radius becoming larger than the bounding box, we require that the time weight function $w(t)$ remains within the range $[0,1]$.

**Separating Velocity**

Similar to selected data segments in time we can discard entire line segments if both the start velocity $v$ and the end velocity $v'$ are either smaller or larger than the selected velocity range. Also, line segments that are completely inside of the range can be rendered entirely. When, however, the line segment is partially inside and partially outside of the selected velocity range, we must, as in the previous section, adjust the integration bounds. Unfortunately it is not as straightforward as directly plugging in time bounds. We first need to find the time bounds associated with the given velocity bounds. From equation we derive the following velocity function for a given $t$:

$$\dot{x}(t) = a(t - t_0) + v$$

(32)

Where $a$ is the acceleration, as given by equation (4) and $v$ the initial velocity as given by equation (3). We know from the movement model—see Section 3.1—that our start velocity is indeed $v$ and using equation (32), we find our end velocity $v'' = \dot{x}(t_1)$. To find a time instance for a given velocity $v''$, we invert equation (32) and get the following function:

$$t(v'') = t_0 + \frac{v'' - v}{a}$$

(33)

If the acceleration $a$ is equal to zero, we know the velocity must be constant over the line segment. In this case we can simple use the tight integration bounds $[t''_0, t''_1]$.

Since we only require velocity segmentation for density map aggregation, we do not require a weight function and can simply adapt the integration bounds as follows for a given velocity range $[v''_0, v''_1]$:

$$t''_0 = \max(t'_0, t(v''_0))$$
$$t''_1 = \min(t'_1, t(v''_1))$$

(34)

---

**Figure 44.** Left: All vessel movements during a single 30 minute period. Right: All vessel movements with a velocity above 30 km/h.

57
In Figure 44 we show an example for separating the data set over time by showing the vessel movements during a single 30 minute period and an example of separation over velocity by showing a density map containing the movements of vessels moving with a velocity over 30 km/h.

### 6.3.6. Stationary Objects

Our movement model does not work properly for objects that are either stationary or barely moving. This manifests itself as degenerate quads and improper line parameterization, which lead to artifacts that are clearly visible and in some cases even dominating in the final visualization. We have found this to be the case in two particular cases; either when the Rhumb length of the line segment becomes too small ($< 1 \cdot 10^{-3}$ km) or when both the acceleration and velocity become too small ($< 1 \cdot 10^{-4}$ m s$^{-1}$ and $< 1 \cdot 10^{-4}$ m s$^{-2}$ respectively).

First we assume that if there is some movement within a certain small margin, we assume the object is stationary and we can model the line segment as a point. We can now approximate the integral of equation (5) using the following equation:

$$K(d, r) \cdot (t_1 - t_0)$$  \hspace{1cm} (35)

Where $t_0$ is the time in the first point, $t_1$ the time in the second point and $d$ is the distance to the center of the point. Since these line segments have a single location in space, we can splat [41] the stationary line segments. We do this by generating a quad exactly the size of the kernel and parameterize it in the range $[-r, +r]$ for both the $x$ and $y$ coordinates such that each fragment can compute its distance to the center of the quad using the Pythagorean theorem: $\sqrt{x^2 + y^2}$. We then find the kernel value for this distance and multiply it by the time between the first and the last point of the line segment.

If a time weight function $W_t$ is used for the convolution within a time window $[T_w - t_ip, t_ip]$, we splat our line segment as follows:

$$K(\sqrt{x^2 + y^2}, r) \cdot \int_{t^*_0}^{t^*_1} W_t(t) \, dt$$  \hspace{1cm} (36)

Where $t^*_0$ is max($t_0, t_ip - T_w$) and $t^*_1$ is min($t^*_1, t_ip$). Instead of multiplying by the length of the line segment in time we multiply by the integral of the part of the time weight function $W_t$ covering the

![Figure 45. The difference between a density map without splatting the stationary points (Left) and a density map with splatting the stationary points (Right). The more clearly visible artifacts on the left image have been highlighted. Note how the left image has sharp edges and elliptical and rectangular shapes.](image)
6.4. Aggregation

As described in section 4.2, we can generate density maps that are an aggregation of multiple smaller density maps, each for a subset of a single dataset, but with varying parameters. Each bin has a filter \( F_i \) and a multiplier and kernel radius assigned by discrete transfer functions. These density maps are then either combined using addition or visualized separately depending on the visualization operator—see section 5.5.

Since we send a render request to the graphics hardware for each moving object separately, we can filter objects based on static properties such as vessel area and vessel type on the host, the CPU. In the case of properties that vary over the course of a moving object’s path, such as time or velocity, we have to apply the filtering partly on the graphics hardware as described in section 6.3.5.

We generate our distribution map by rendering a white polyline to a one-channel texture, for each moving object, where the \( x \) and \( y \) coordinates of the points are the values of the respective selected axes for each data point. We then apply a logarithmic scale to the distribution map and scale the values to the range \([0,1]\).

6.5. Density Combination Operator

The density combination operators discussed in section 4.3 are applied to the targeted density maps in a single fragment shader pass. For efficiency reasons we apply the density combination operator in screen space instead of in geographic space. While this does not change anything conceptually, it does mean that we apply the density combination operator only to the visible part of the density map. This also means that the computation time of the density combination stage does not depend on the size and precision of the density map.

6.6. Visualization

While the density maps are only recomputed whenever a significant parameter changes, the visualization stage is redrawn at a maximum of 20 frames per second. In the visualization stage we apply the color map to the density map, then we apply the visualization operator and finally we apply lighting to

\[ \text{Color Map} \]

\[ \text{Density Map} \]

\[ + \]

\[ \text{Normal Map} \]

\[ \text{Shaded Image} \]

Figure 46. A colored density map and a normal map are combined into a shaded image.
the colored image using the normal map by treating the density map as a height field and illuminating it as such using the Phong shading model [15]—see Figure 46. The normal map is computed as follows for some integer position \((i, j)\):

\[
\text{normal}(i, j) = \left( \frac{\text{density}(i-1,j) - \text{density}(i+1,j)}{2}, \frac{\text{density}(i,j-1) - \text{density}(i,j+1)}{2} \right)
\]

(37)

Where \(\text{density}(i,j)\) is the density at position \((i,j)\). We use the Phong shading model without specular highlights where we use a single white light source positioned at infinity, which means we compute the dot product between its normal and a constant light direction, per fragment. The color of the fragment is then multiplied by the dot product.

### 6.6.1. Color maps
We store our color maps in a linear lookup table in the form of a one-dimensional texture. Where required, we let the graphics hardware take care of the interpolation. The lookup table is the size of the number of samples. Whenever the number of samples is changed, the lookup table is resized.

In the case of aggregation we use a set of one-dimensional color maps. For efficiency reasons we store the color map for each bin in a separate one-dimensional texture instead of in one single two-dimensional texture. The bin colors can either be selected manually or automatically selected from a predefined color map. In the latter case we sample a predefined color map at regular intervals such that the colors of the different bins are sufficiently distinct. We use the following predefined color map:

![Predefined rainbow color map](image)

*Figure 47. A predefined rainbow color map used to automatically select colors for aggregation bins. In the above image colors for three bins are sampled at the black arrows.*

### 6.6.2. Surface Reconstruction
If the area of a cell projects to multiple pixels on the screen, it becomes interesting to interpolate the density values in between the samples, the centers of the cells, to avoid large continuous blocks of identical density on the screen and achieve the appearance of a smooth surface. Standard texture interpolation on the GPU uses bilinear interpolation—see Figure 48—, however, this unfortunately does not accurately describe the nature of the density map, which is, due to the choice of a Gaussian kernel, smooth. Due to discontinuities between surface normals of the consecutive cells—see Figure 48—the transitions between neighboring cells show up as hard edges in the lighting.

Instead, we use a cubic filtering method to reconstruct our density map. This filtering method must adhere to two important requirements. Firstly, the filtering method must preserve the values at the known points and only interpolate in between these points and secondly, the derivative of the filtered density must be continuous and smooth for visually smooth lighting. We have chosen to use an approximation of the bicubic hermite spline using one-dimensional Catmull-Rom splines. We first show the one-dimensional version and then extend it into two-dimensional space.

In the one-dimensional version for two points \(p_i\) and \(p_{i+1}\) and \(t\) in the interval \([0,1]\), where \(t = 0\) is \(p_i\) and \(t = 1\) is \(p_{i+1}\), the cubic hermite spline is given as follows [42]:

\[
p^3(t, p_i, m_i, p_{i+1}, m_{i+1}) = H_0(t)p_i + H_1(t)m_i + H_2(t)p_{i+1} + H_3(t)m_{i+1}
\]

(38)
Where $H_i$ are the four hermite basis functions shown in Figure 50 and $m_i$ and $m_{i+1}$ are the tangents of $p^3$ at point $p_i$ and point $p_{i+1}$ respectively. For Catmull-Rom splines a tangent $m_i$ is given as follows:

$$m_i = \frac{p_{i+1} - p_{i-1}}{2} + \frac{p_i - p_{i-1}}{2}$$

(39)

Note that since our density map is a uniform grid, we can assume each interval between two consecutive points $p_i$ and $p_{i+1}$ to be the unit interval $[0,1]$. Using the basis functions shown in Figure 50, we can see that $p^3(0, p_i, m_i, p_{i+1}, m_{i+1}) = p_i$, satisfying the first requirement. Also, Figure 51 illustrates that the derivatives of the basis functions are continuous and smooth, satisfying the second requirement.

Extending the above spline into two-dimensional space with exact results involves solving a complex linear system with 256 coefficients for each pixel within the square between the points $p_{i,j}$, $p_{i+1,j}$, $p_{i+1,j+1}$ and $p_{i,j+1}$ [28]. This is, however, infeasible for our implementation. Instead, we extend the spline into two-dimensional space by treating the axes separately. Pixels within the square between the points are computed as follows:

$$m_{i,j}^x = \frac{p_{i+1,j} - p_{i-1,j}}{2} + \frac{p_{i,j} - p_{i-1,j}}{2}$$

$$m_{i,j}^y = \frac{p_{i+1,j} - p_{i,j-1}}{2} + \frac{p_{i,j} - p_{i,j-1}}{2}$$

$$p^3(t_x, p^x_{i,k}, m^x_{i,k}, p^x_{i+1,k}, m^x_{i+1,k})$$

$$p_{bicubic}(t_x, t_y) = p^3(t_y, p^y_{i,j}, m^y_{i,j}, p^y_{i+1,j}, m^y_{i+1,j})$$

(40)

Where $t_x$ and $t_y$ are the pixel positions within the interval $[0,1]$ for the $x$ and the $y$ directions respectively.
tively. Another way to approximate the bicubic hermite spline is by using a bicubic convolution algorithm such as suggested by Keys et al. [8]. This, however, gives almost exactly the same result.

Figure 52. A comparison of linear interpolation (left) and approximate bicubic hermite spline interpolation (right).

Figure 53. Our surface reconstruction also helps reduce jagged edges. Linear surface reconstruction (left) and cubic surface reconstruction (right).

6.7. Visualization Operator

In Section 5.5 we discussed several visualization operators such as a combined visualization of multiple density maps, the result of the density combination operator, or a single target visualization where the color map is applied to one density map and the shading to the result of the density combination operator. We use specialized shaders to apply our visualization operators. In the next section we discuss the aggregation operator and its combination operators in more detail.

6.7.1. Aggregation

The aggregation visualization operator, as discussed in section 5.5.3, is slightly more complex than the visualization operators we have dealt with before. For each bin, a colored image is generated. This
means that the density maps for each bin have to be projected into screen space individually before they can be combined. The normal map is then generated from the result of the density combination operator and, in a final pass, the colored images and the normal map are combined into an aggregated, shaded visualization. How the individual fragments of each density map are combined per position depends on the chosen combination operator which can be either Color Mixing, Maximum or pixel blocks.

**Color Mixing**
For the color mixing operator we use a transfer function similar to the kernel radius and kernel weight transfer functions of the density computation stage for our color weights. Using an editable bar chart the color weights \(a_i\) can be changed for each bin \(i\). The color maps are applied to the density map for each bin and the colors are combined using weighted addition in an auxiliary four channel buffer as follows:

\[
\sum_{i=0}^{N-1} (a_i r_i, a_i g_i, a_i b_i, a_i) \tag{41}
\]

Where \(r_i, g_i\) and \(b_i\) are the color channels red, green and blue respectively. In a second pass the color channels are normalized by dividing them by the summed weight accumulated in the alpha channel of the auxiliary buffer. The summed weights of some pixel may be equal to zero when there is some density contribution from one or more bins, but the weights of these bins are set to zero. In this case we treat the pixel as if there is no density contribution. Pixels without any density contribution are ignored.

**Maximum**
Which bin has the maximum density value varies per pixel on the screen. This means we need to sort the fragments such that we can easily find the maximum. Fortunately, we can use the graphics hardware to achieve this. We simply need to turn on the depth testing and give each fragment a depth based on its density value as follows for some position \(p\):

\[
depth(p) = \begin{cases} 
1 & \text{if } \text{density}_i(p) \leq 0 \\
1/\text{density}_i(p) & \text{otherwise}
\end{cases} \tag{42}
\]
Where $\text{density}_i(p)$ is the density of bin $i$ at some position $p$. Now the depth test on the graphics card ensures that the fragment with the highest density ends up in front. Using an opacity of 1 we ensure that it is also the only visible fragment. This also means that the order in which we blend the colored density maps is irrelevant.

**Pixel Blocks**

The aggregation combination function defining pixel blocks, as discussed in section 5.5.3, is implemented as follows. We blend each colored density map of the bins in a separate pass. While blending these colored images we also keep track of the number of layers per pixel that have been previously blended. We store the number of layers in a map that we update after each pass. We do this by rendering the density map of the bin again and, for every pixel, add a 1 if the density is non-zero and a 0 otherwise. For each layer we now render a pixel if the following equation evaluates to true:

$$\text{rand} \mod \text{layers} = 0$$  \hspace{1cm} (43)

Where $\text{rand}$ is a randomly selected value we retrieve from a pre-generated random texture and $\text{layers}$ is the number of layers previously rendered to that pixel. This means that each colored density map gets an increasing number of randomized holes depending on the number of layers below.

---

**Pixel Block Fragment Code**

```plaintext
layerMap ← 0
for each bin $i$ do
   for each pixel $p$ do
      if $\text{rand}([p_x/s], [p_y/s]) \mod (1 + \text{layerMap}(p)) = 0$ then render colored density
   for each pixel $p$ do
      if $\text{density}_i(p) > 0$ then
         layerMap(p) ← layerMap(p) + 1
```

*Algorithm 6. The general procedure used to apply the pixel block operator. Where $\text{layerMap}$ is a two-dimensional map with an entry for each pixel $p$, $\text{rand}$ is a two-dimensional texture containing random values, $s$ is the size of the pixel blocks and $\text{density}_i$ is the density map for bin $i$.*

Where $\text{layerMap}$ is a two-dimensional map with an entry for each screen pixel $p$, $\text{rand}$ is a two-dimensional texture containing random values, $s$ is the size of the pixel blocks and $\text{density}_i$ is the density map for bin $i$. We divide the pixel coordinates by $s$ and floor the result when retrieving a random value such that the random value is identical for each pixel in a single pixel block of size $s$. Note that the number of layers may vary per pixel within a single pixel block and therefore the color may change within the pixel block when it is only partially overlapped.
Chapter 7.

Results

In the previous chapters we have discussed a method for visualizing moving object data in the form of a density map by convolving the movement tracks onto a density map. We can isolate subsets of our data set based on time, velocity or any other attribute. These subsets can be shown in a single visualization using different operators for combining the maps and visualizing the result.

In this chapter we analyze the computation speed of our implementation by comparing it to the computation speed of the previous work of Willems et al. [1] and comparing the computational speed of different techniques and parameters within our method. We then proceed by showing and discussing a number of use cases for our system to show some practical uses for our methods as well as showing that our methods can be applied to different kinds of data sets as well.

7.1. Comparison

In this section we compare the computation speed of our method to the method of Willems et al. [1]. For our comparison we use the one day density map over a 130.48 by 102.34 kilometer area around the harbor of Rotterdam. We compute a density map with a kernel radius of one kilometer and a density map with a kernel radius of 100 meter combined using weighted addition. Both density maps are computed with a cell size of 40 meter leading to a total of 8,552,178 cells. The method of Willems et al. [1] uses a cone kernel while we use a kernel based on the Gaussian function.

Computing this density map using the system of Willems et al. [1] takes approximately three hours while computing this density map using our method takes 0.98 seconds. This means our method is approximately a factor 3600 faster.

7.2. Benchmarks

We show a number of computation times for different sets of parameters in this section. These timings are produced using the following procedure: For every trial we compute all density maps 50 times. The timing is then the total time of all passes divided by 50. The size of the density maps depends on their respective parameters, but we always render our visualization to an image of 900 by 900 pixels. These tests are run on an Intel Core i7 CPU at 2.8 GHz with an Nvidia GeForce 285 with 1GB of video memory.

We use two vessel data sets, one data set containing data over a single day with 1390 objects and a total of 306,521 data points and a second data set containing data over seven days with 11,802 objects and a total of 425,591 data points. Both data sets are of an approximate 400 by 400 kilometer area of the Dutch coast.
7.2.1. Kernel Radius

First we investigate the effect of the kernel radius on the computation time. We keep our cell size fixed at 250 meters and set our area to the range of the entire data set. With an area of 429.85 by 402.99 kilometer, this means we compute 2,772,640 cells for each trial. We show some density maps of our test area with varying kernel radii in Figure 56 and the results of our tests in Table 1 below.

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Kernel Radius</th>
<th>Average time per pass in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>250 m</td>
<td>0.13</td>
</tr>
<tr>
<td>2</td>
<td>500 m</td>
<td>0.23</td>
</tr>
<tr>
<td>3</td>
<td>1 km</td>
<td>0.53</td>
</tr>
<tr>
<td>4</td>
<td>2 km</td>
<td>1.37</td>
</tr>
<tr>
<td>5</td>
<td>3 km</td>
<td>2.03</td>
</tr>
<tr>
<td>6</td>
<td>4 km</td>
<td>2.37</td>
</tr>
</tbody>
</table>

Table 1. Density map render times for varying kernel radii with a fixed cell size of 250 meters.

As we can see in Table 1, the computation time for a pass roughly doubles when the kernel radius doubles. When the radius of the kernel doubles, the oriented bounding boxes we construct around our line segments become twice as wide and 2r longer where r is the kernel radius. Since we have a large number of long line segments, the number of fragments added by the bounding boxes becoming 2r longer can be neglected compared to the number of fragments added by the oriented bounding boxes becoming twice as wide.

Figure 55. A plot of the computation times of Table 1.

Figure 56. A selection of density maps of vessel movements around the Dutch coast with a cell size of 250 meter and a kernel size of 250 meter (A), 1 kilometer (B) and 3 kilometer (C).
7.2.2. Cell Size

We now investigate the effect of varying the cell size. Since we are bound to a maximum number of cells we can process due to texture size limitations, we look at a smaller area. This allows us to make the cell size very small relative to the kernel radius while maintaining a realistic kernel radius. We keep our kernel radius constant at two kilometer and plot density maps of a 125.04 by 102.34 kilometer area around the harbor of Rotterdam. We show our test area for a select number of cell sizes in Figure 59.

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Cell Size</th>
<th># of cells</th>
<th>Average time per pass in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000 m</td>
<td>13802</td>
<td>0.13</td>
</tr>
<tr>
<td>2</td>
<td>500 m</td>
<td>54940</td>
<td>0.23</td>
</tr>
<tr>
<td>3</td>
<td>250 m</td>
<td>219350</td>
<td>0.48</td>
</tr>
<tr>
<td>4</td>
<td>200 m</td>
<td>342528</td>
<td>0.56</td>
</tr>
<tr>
<td>5</td>
<td>100 m</td>
<td>1369088</td>
<td>0.77</td>
</tr>
<tr>
<td>6</td>
<td>50 m</td>
<td>5473678</td>
<td>1.88</td>
</tr>
<tr>
<td></td>
<td>25 m</td>
<td>21890618</td>
<td>5.93</td>
</tr>
</tbody>
</table>

Table 2. Density map render times for varying cell sizes with a fixed kernel radius of two kilometer.

We expect an inverse relationship between the cell size and the computation time, which is corroborated by the data in Table 2. When we plot a graph of our results—see Figure 57—we can fit the following function for cell size \(x\) to our results:

\[
\frac{1}{cx^2} + \xi
\]  

(44)

Where \(c\) is the computation time per cell and \(\xi\) is some constant overhead caused by the bounding box setup, visualization and general GPU overhead. If we compare a plot of the above function to a plot of the computation times accumulated in Table 2, we see a striking similarity between both functions—see Figure 58—for a \(c\) of \(~2.5 \cdot 10^{-4}\) seconds and a \(\xi\) of \(~0.2\) seconds.

![Figure 57. The computation times of the varying cell sizes of Table 2 plotted in a graph.](image)

![Figure 58. The function \(\frac{1}{cx^2} + \xi\) plotted in a graph with \(c\) equal to \(~2.5 \cdot 10^{-4}\) seconds and \(\xi\) equal to \(~0.2\) seconds.](image)

7.2.3. Comparing Data Sets

As mentioned before, we have two vessel data sets. The seven day data set has been significantly more simplified than the one day data set. This means there is less detail in the movements, but the travelled distance of the moving objects should be approximately the same.

Since the seven day data set spans an amount of time that is seven times larger than the one day data set, we expect the computation time for the seven day data set is approximately seven times as long as the number of fragments computed for the single day data set using the same settings. We test this hypothesis using several different settings of which the results are shown in Table 3.
Table 3. A comparison of the computation time for the one day and seven day data sets for varying cell sizes and kernel radius including the proportion between the computation time of both data sets.

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Cell Size</th>
<th>Kernel Radius</th>
<th>Average time per pass in seconds:</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>One day / Seven days</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000 m</td>
<td>1000 m</td>
<td>0.16 / 0.35</td>
<td>2.19</td>
</tr>
<tr>
<td>2</td>
<td>1000 m</td>
<td>2000 m</td>
<td>0.28 / 0.51</td>
<td>1.82</td>
</tr>
<tr>
<td>3</td>
<td>500 m</td>
<td>500 m</td>
<td>0.15 / 0.33</td>
<td>2.20</td>
</tr>
<tr>
<td>4</td>
<td>500 m</td>
<td>1000 m</td>
<td>0.25 / 0.48</td>
<td>1.92</td>
</tr>
<tr>
<td>5</td>
<td>500 m</td>
<td>2000 m</td>
<td>0.58 / 0.92</td>
<td>1.59</td>
</tr>
<tr>
<td>6</td>
<td>250 m</td>
<td>250 m</td>
<td>0.13 / 0.33</td>
<td>2.54</td>
</tr>
<tr>
<td>7</td>
<td>250 m</td>
<td>500 m</td>
<td>0.21 / 0.46</td>
<td>2.19</td>
</tr>
<tr>
<td>8</td>
<td>250 m</td>
<td>1000 m</td>
<td>0.50 / 0.83</td>
<td>1.66</td>
</tr>
<tr>
<td>9</td>
<td>250 m</td>
<td>2000 m</td>
<td>1.30 / 1.84</td>
<td>1.41</td>
</tr>
<tr>
<td>10</td>
<td>100 m</td>
<td>100 m</td>
<td>0.13 / 0.38</td>
<td>2.92</td>
</tr>
<tr>
<td>11</td>
<td>100 m</td>
<td>250 m</td>
<td>0.24 / 0.60</td>
<td>2.50</td>
</tr>
<tr>
<td>12</td>
<td>100 m</td>
<td>500 m</td>
<td>0.59 / 1.10</td>
<td>1.86</td>
</tr>
<tr>
<td>13</td>
<td>100 m</td>
<td>1000 m</td>
<td>1.46 / 2.31</td>
<td>1.51</td>
</tr>
<tr>
<td>14</td>
<td>100 m</td>
<td>2000 m</td>
<td>2.29 / 4.12</td>
<td>1.80</td>
</tr>
</tbody>
</table>

As it turns out, the difference in computation is not close to a factor of seven, but instead nearer to the ratio of the number of points, which is ~1.39. This suggests the actual number of line segments is of more importance than the total length of the movements in the data set. Since connecting line segments have overlapping regions where contributions for both line segments are computed, a movement with more line segments will require more fragments than the same movement with only a single line segment—see Figure 61.
We have now seen that the computation time does not only depend on the kernel radius and cell size, but also on the granularity of the line segments.

### 7.2.4. The cost of geographic space

In this section we compare our standard method in geographic space to a specially altered version that computes our density contributions in Euclidian space by first projecting all data into Mercator space and treating the resulting projected points as Euclidian coordinates. This means we do not use expensive operations such as computing the geographic distance or map projections in our shaders. We test a number of different settings using Euclidian space and compare the resulting computation times to those recorded earlier using geographic space. We use the seven day data set for these trials.

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Cell Size</th>
<th>Kernel Radius</th>
<th>Average time per pass in seconds:</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Geographic / Euclidian</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000 m</td>
<td>1000 m</td>
<td>0.35 / 0.23</td>
<td>1.52</td>
</tr>
<tr>
<td>2</td>
<td>1000 m</td>
<td>2000 m</td>
<td>0.51 / 0.30</td>
<td>1.70</td>
</tr>
<tr>
<td>3</td>
<td>500 m</td>
<td>500 m</td>
<td>0.33 / 0.23</td>
<td>1.43</td>
</tr>
<tr>
<td>4</td>
<td>500 m</td>
<td>1000 m</td>
<td>0.48 / 0.20</td>
<td>2.40</td>
</tr>
<tr>
<td>5</td>
<td>500 m</td>
<td>2000 m</td>
<td>0.92 / 0.48</td>
<td>1.92</td>
</tr>
<tr>
<td>6</td>
<td>250 m</td>
<td>250 m</td>
<td>0.33 / 0.25</td>
<td>1.32</td>
</tr>
<tr>
<td>7</td>
<td>250 m</td>
<td>500 m</td>
<td>0.46 / 0.30</td>
<td>1.53</td>
</tr>
</tbody>
</table>

---

Figure 60. The one day data set (left) and the seven day data set (right) rendered with a kernel radius of one kilometer and a cell size of 250 meter.

Figure 61. A movement with its bounding box shown as a single line segment (above) and multiple line segments (below). The darkened areas are where neighboring bounding boxes overlap.
Chapter 7 – Results

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Cell Size</th>
<th>Kernel Radius</th>
<th>Average time per pass in seconds:</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Geographic / Euclidian</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>250 m</td>
<td>1000 m</td>
<td>0.83 / 0.48</td>
<td>1.73</td>
</tr>
<tr>
<td>9</td>
<td>250 m</td>
<td>2000 m</td>
<td>1.84 / 0.95</td>
<td>1.95</td>
</tr>
<tr>
<td>10</td>
<td>100 m</td>
<td>100 m</td>
<td>0.38 / 0.29</td>
<td>1.31</td>
</tr>
<tr>
<td>11</td>
<td>100 m</td>
<td>250 m</td>
<td>0.60 / 0.41</td>
<td>1.46</td>
</tr>
<tr>
<td>12</td>
<td>100 m</td>
<td>500 m</td>
<td>1.10 / 0.68</td>
<td>1.62</td>
</tr>
<tr>
<td>13</td>
<td>100 m</td>
<td>1000 m</td>
<td>2.31 / 1.30</td>
<td>1.78</td>
</tr>
<tr>
<td>14</td>
<td>100 m</td>
<td>2000 m</td>
<td>4.12 / 2.34</td>
<td>1.76</td>
</tr>
</tbody>
</table>

Table 4. A comparison of the computation time between the standard method in geographic space and an altered method in Euclidian space including the proportion between the computation time of both methods.

The results for these trials are shown in Table 4. We can see a clear increase in speed when we compute our density maps in Euclidian space. With an average increase of a factor 2, we can conclude that both the geographic distance function and the map projection functions require a significant amount of computation time.

In Figure 62 we show the difference between a density map computed in geographic space and a density map computed in Euclidian space. While the difference is not easy to see, we can see upon careful inspection that the density map computed in Euclidian space is slightly elongated at the top, which is consistent with the areal distortion of the Mercator projection—see Appendix A.2.

7.2.5. Integral Approximation Precision
The number of iterations in our numeric approximation of the integral of equation (30)—see Section 6.3.3—may also have a large impact on the computation speed of our method. In this section we test how big of an impact the number of iterations has by running several trials with different settings for a varying number of steps. For the above tests and all images generated for this thesis, a constant number of 10 iterations was used to approximate the integral. As before, we use the seven day data set for the trials of which we show the results in Table 5.

Figure 62. A density map of the seven day data set with a kernel size of 500 meter and a cell size of 250 meter computed in geographic space (left) and computed in Euclidian space (right). The rightmost density map is slightly elongated at the top which is consistent with the distortion caused by the Mercator projection—see Appendix A.2.
<table>
<thead>
<tr>
<th>Trial #</th>
<th>Cell Size</th>
<th>Kernel Radius</th>
<th>Average time per pass in seconds for # of iterations:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>500 m</td>
<td>500 m</td>
<td>0.21</td>
</tr>
<tr>
<td>2</td>
<td>500 m</td>
<td>1000 m</td>
<td>0.23</td>
</tr>
<tr>
<td>3</td>
<td>500 m</td>
<td>2000 m</td>
<td>0.33</td>
</tr>
<tr>
<td>4</td>
<td>250 m</td>
<td>500 m</td>
<td>0.22</td>
</tr>
<tr>
<td>5</td>
<td>250 m</td>
<td>1000 m</td>
<td>0.30</td>
</tr>
<tr>
<td>6</td>
<td>250 m</td>
<td>2000 m</td>
<td>0.54</td>
</tr>
<tr>
<td>7</td>
<td>100 m</td>
<td>500 m</td>
<td>0.37</td>
</tr>
<tr>
<td>8</td>
<td>100 m</td>
<td>1000 m</td>
<td>0.65</td>
</tr>
<tr>
<td>9</td>
<td>100 m</td>
<td>2000 m</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Table 5. A comparison of the computation time between varying numbers of iterations of the numeric approximation of the integral using different cell sizes and kernel radii.

As can be seen in Table 5, the number of iterations used for the numeric approximation of our integral has a significant impact on the computation time. In Figure 63 we show a number of density map images with a kernel radius of 500 meter and a cell size of 40 meter for the number of iterations tested in Table 5. Using a single iteration, we can observe a significant loss in quality, which is especially visible in the sparse movement data at the bottom of the image. When we double the number of iterations we can still see similar artifacts in the same area, only to a lesser extent. We also notice that the spatial density distribution has changed. This is because an approximation using two iterations is closer to the actual integral than an approximation using a single iteration. We see the spatial density distribution changes yet again when we switch to five iterations. Though they are barely visible, there are still some artifacts left using five iterations. The last of the visible errors disappear at 10 iterations, which is why we have chosen this number of iterations as our standard. The image generated using 20 iterations is visually identical to the image generated using 10 iterations. Note that reducing the amount of iterations also reduces the amount of times the geographic distance is computed and the amount of times the inverse Mercator projection is applied—see Section 3.3.

Figure 63. A comparison of the image quality for different numbers of iterations. The images of one and two iterations show clear artifacts and a different spatial density distribution in the encircled areas as can be seen due to the contours of the colors. The image of five iterations looks similar to the image of 10 and 20 iterations, but artifacts can still occur. The images of 10 and 20 iterations look identical.
7.2.6. Visualization
The majority of the computation time comprises of generating the density map. The visualization steps take a negligible amount of time to process. Since our visualization techniques work in screen space, the only factors of influence on the computation time of the visualization stage are the size of the screen and the number of density maps processed.

7.2.7. Conclusion
In the previous sections we have shown what the main factors of influence on the computation time are, namely: The size of the cells, the radius of the kernel and the numeric precision. This means there is a clear trade-off between the quality of the image and the render time of the density computation. The visualization stage takes a negligible amount of time to process and remains mostly constant.

7.3. Use cases
In the following sections we show some of the possible practical uses of our method in a number of hypothetical use cases. First we examine the anchor zones of our moving vessel data by separating velocities using the aggregation techniques described earlier. In the second use case we delve into anomaly detection using the anomaly detection operator of Section 4.3.2. And finally, we investigate how our methods can be applied to another data set such as pedestrian movement data set, created to aid urban planning.

7.3.1. Analyzing Anchor Zones
For this use case we use a data set containing movements near the Dutch coast over seven consecutive days. We analyze a number of hot spots around the harbor of Rotterdam and try to find an explanation for each of these hot spots.

Near the Dutch coast there are a number of designated anchorage areas where vessels are allowed to lie at anchor. These areas are mostly near the mouth of a harbor or near busy shipping lanes. We want to find these areas using our system. To do this, we first generate a density map of the entire area—see Figure 65. We can immediately see a number of hot spots, however, we cannot clearly distinguish between hot spots caused by ships lying at anchor and hot spots caused by a large amount of vessel activity in the area. For example; the harbor of Rotterdam is marked by box G. On the open sea, near the mouth of the harbor we see a hot spot that continues all the way inland. We cannot be certain which part of this hot spot is caused by ships that are lying at anchor and which part is caused by ships that are moving into or out of the harbor. We will need to separate these different types of hot spot. This means we need a better way to visualize our potential anchor zones.

We use aggregation to split our dataset into stationary vessels and moving vessels by creating two bins as shown in Figure 64. We give our stationary bin a red color and the rest a blue color. We then use the aggregation visualization operator with the maximum combiner such that our hot spots are on top. We now get the density map shown in Figure 66. We can now see that our hot spots are not only more clearly visible, but also a lot more accurate. We see the hot spot around the harbor of Rotterdam has been split into a blue and a red hot spot. Since we are only interested in slow moving or stationary vessels, we are only interested in the red hotspots. Also, we see some new red hot spots that were not clearly visible before in Figure 65: K, L and M. Hot spots E and F seem to be slightly larger than we initially thought and can be viewed as a single hot spot area.
Figure 65. A combination of two density maps, one with a kernel size of three kilometers and the other with a kernel size of 300 meters. The color map is applied only to the density map with the larger kernel. We can see several hotspots of potentially stationary vessels marked with the letters A through J.

Figure 66. A combination of two density maps: One aggregated density map using the two bins shown in Figure 64 and a normal density map with a small (300m) kernel size. As visualization operator we use the aggregation visualization operator with as target the aggregated density map and as combiner the maximum operator.
If we compare our map to the vessel traffic chart of the north sea of the Netherlands Hydrographic Office [43], we can see that we have found all designated anchorage areas in the visible data set: Area B, C, E, H, I, J and L. Note how we were only able to positively identify area L as a hot spot using the visualization of Figure 66.

We know that area G is the harbor of Rotterdam so that leaves us to investigate the other regions. We take a closer look at the unidentified regions or hot spot regions close to anchorage areas and look at the type of ships we find there. We start with region A. To get an idea of what may be causing the hot spots, we set our bin axes over velocity and vessel type. We create a bin per vessel type for low velocities as shown in Figure 67 and use aggregation visualization as our visualization operator. Since we now potentially have a large number of colored images overlap, we use the pixel block combiner—see Section 5.5.3—so we can see all density contributions. In region A in Figure 68, we can see brown and blue hot spots, which means, according to our bins, hotspot A is caused by cargo vessels and tankers. If we reduce the kernel radius of the aggregation, we can see that the hot spots coincide exactly with ocean platforms. These vessels are most likely resupplying the ocean platforms or collecting gas or oil. In region M we see a single yellow hotspot, which means this hotspot is caused by a fishing vessel remaining stationary for a longer period of time in what seems to be a busy area.

In region E we can see stationary vessels of all types, especially near the harbor of Scheveningen (at the bottom of the map). Outside of the harbor we see a number of sailing vessels and pleasure crafts, one or more anchored cargo ships and some special crafts including a law enforcement vessel and a law enforcement vessel.
search and rescue vessel. The brown hot spot at the top left of the image is caused by a tanker. This area turns out to be a designated anchorage area near the harbor of Scheveningen. The pink hot spot in the same area is caused by a vessel marked as 'Other Type'. Further inspection using the vessels’ call-sign and various marine databases, however, shows this vessel is a tanker as well.

In area J we can see two distinct hot spots caused by stationary special crafts and sailing or pleasure crafts. The right hot spot is concentrated in and around a designated anchorage area and a small harbor of the town of Hellevoetsluis. The left hot spot coincides with the location of the Haringvlietdam, which is a part of the Delta Works and contains a lock—see Figure 69. This explains why there is a hotspot of stationary vessels around the Haringvlietdam. In the last area, area K, we see two hot spots. The hot spot in the lower left corner of the area is caused by a special craft, or more specifically, a dredger. The strange looking pattern in the right of the area is caused by a law enforcement vessel that periodically makes a stop along its path.

We have now determined that the hot spots we have found are designated anchorage areas or are caused by harbors, locks or behavior consistent with the vessel type.

### 7.3.2. Anomaly Detection

In this section we investigate how we can use our system to find anomalous behavior in our data sets. We define anomalous behavior as movement patterns that have little or nothing in common with other movement patterns. These other movement patterns are, in our case, in the context of some historical density map.

In this use case we look at the seven day data set where we try and find anomalies in the seventh day using the first six days as our historical context. The first thing we can try is generating a density map of the historical context using a large kernel radius and a low weight and a density map of the seventh day using a small kernel radius and a high weight. Using weighted addition, this gives us the image in Figure 70. On this map we can clearly see the hot spots formed by the harbor of Rotterdam and various anchorage areas, but we cannot clearly see any anomalies except for two vessels in the right side of the image, which may be anomalous since we cannot see any other activity in the area. However, we also cannot see whether or not there is any historic activity that coincides exactly with the movements we can see.
We now turn to the anomaly detection operator defined in Section 4.3.2. Since the time span of our historical data set is six times larger than the time span of the seventh day we expect there to be approximately six times as much movement. We now set $r$ to six and, using a green to red color map, get the image of Figure 71. The first thing we notice is that among the various anchoring areas and the harbor of Rotterdam we see several red peaks. This means that there were ships anchored at those exact spots for a longer period of time during the seventh day than during the first six days. We also see a number of areas with a higher density, which means some activity has taken place in those areas that is anomalous compared to the historical context, the previous six days.

We investigate each anomaly we have found separately. In the first area, area A, we see two paths showing up as anomalies. Closer investigation reveals that these paths are caused by two cargo vessels moving from the harbor of Rotterdam to the harbor of Ijmuiden using, in the context of the historical map, an unusual route.

In area B we can see a fairly strong anomaly. This anomaly turns out to be caused by a research and survey vessel, following a periodic path back and forth in area B—see Figure 72. In area D we see another large anomaly. Closer investigation into the area reveals that this anomaly is caused by a single ship of the coastguard. Additionally, this is the same vessel causing the anomalous track in area E.

Figure 70. Weighted addition of a density map of the first six days of data with a kernel size of one kilometer and a weight of 0.01 and the seventh day with a kernel size of 500 meter and a weight of one. We cannot clearly see any anomalies except for the movements in the encircled area. However, we cannot determine whether these movements actually are anomalies or just seem anomalous because they obscure historic area usage.
In area C we see a strong anomaly in a busy area at the mouth of the harbor of Rotterdam. This would suggest there is some extra activity in area C compared to the historical context. To investigate what is happening in this area we use our time window density map with an interest point $t_p$, moving over the seventh day and a time window size of half an hour. We now notice a number of vessels moving around in the hotspot in area C. The vessels occasionally leave the area individually, but they always seem to return shortly. Closer investigation reveals that these vessels are all pilot vessels. A pilot vessel is a small boat carrying so-called pilots. These pilots are mariners that guide large vessels such as tankers through dangerous or congested areas such as the harbor of Rotterdam. The pilot vessels take care of picking up or dropping off pilots from large vessels that are leaving or entering the harbor respectively. This increased activity of pilot vessels may suggest the presence of more large vessels. To investigate this theory, we generate density maps of the first six days and of the seventh day of vessels with an area larger than 5000 m$^2$ using our binning system. We then apply the anomaly detection operator using a $\tau$ of 6. This reveals some possible hotspots in

**Figure 71.** An anomaly density map of one day of movement data compared to a historical map of six days of movement data.

**Figure 72.** A density map containing only the movements of the research vessel during the seventh day.
our area—see Figure 74. There are, however, two types of anomalies that our anomaly detection operator can reveal: A difference in spatial distribution and a difference in activity. The first type of anomaly is caused by vessels taking (slightly) different routes compared to the historical data, such as in Areas A and E. The second type is caused by an increase in activity in a certain area compared to the historical data, such as in Areas B and D. To find out whether the hotspots in Figure 74 are caused by large vessels taking slightly different routes or by an actual increase in the number of large vessels we use the absolute difference operator. As can be seen in the resulting image in Figure 75, there are no clear peaks and the density is fairly uniform in our area. This means the anomalies in Figure 74 are caused by a difference in spatial distribution, which in turn means there must be some other reason why the pilot vessels show more activity in region C during the seventh day compared to the first six days. The waters around the harbor of Rotterdam could have been more dangerous during the seventh day due to low tide or worse weather conditions compared to the first six days.

We have shown in this section how to use a combination of the anomaly detection operator, the absolute difference operator and the sliding time window method to find and identify possible anomalies in a sparse dataset, using some historic reference data set. While, in the above, we have shown our anomaly detection operator on static data, it can also be applied to real-time data so an operator can immediately recognize an anomaly in real-time. In Figure 73 we show an example of such an application using the sliding time window. We can see one of the cargo vessels that we have established that contributes to the anomaly seen in area A in Figure 71 as a moving anomaly. Since the density of the historical map is zero in the area, the vessel will show up as an anomaly until it enters the harbor mouth of Ijmuiden where there is a large amount of historic movement.

### 7.3.3. Urban Planning

For our third use case we use a totally different type of data set. We investigate how our method can contribute to urban planning by looking at a data set of moving pedestrians in the city center of Delft.
GPS tracking devices are distributed to pedestrians at a number of set locations, access points to the city center such as train or bus stations and parking facilities, and their movements are tracked by the devices [44]. The movement behavior of pedestrians in city centers is used to help improve city management and urban planning by finding which streets are preferred by pedestrians and where congestion points occur.

In Figure 76 we show our method applied to the pedestrian data set overlaid on an open street map [45] of the city center of Delft. We can clearly see where the busy areas are and we see several hotspots. The majority of the activity is centered around areas such as markets, streets with a lot of shops, squares, restaurants and tourist attractions. Unfortunately, pedestrian movements are not as well behaved as vessel movements. Where vessels maintain a fairly constant velocity and bearing, pedestrian movement is fairly chaotic. Additionally, due to floating point precision problems, our map projection functions are unstable in such a small area. To remedy this problem we first project our data into Mercator space once, using the higher precision of the CPU and then compute our densities.
in Mercator space. Since the city center of Delft is a relatively small area, this method is precise enough.

The pedestrians in the data set have supplied several attributes such as purpose of visit and gender. We use our binning and aggregation method to investigate the distribution of the different purposes over time and over area. We bin our data set into four bins, one for each different purpose: Shopping, Tourism, Leisure or Other. For each of the four days in the data set, we compute an aggregated density map using the aggregation visualization operator and the maximum combiner. This gives us the density maps shown in Figure 77. The first thing we notice is that the shopping bin appears to be the dominant bin for each of the four days. Also, we notice that there is a significant, yet unsurprising, increase in pedestrian activity during the weekend, both in the respect of actual density and the spatial distribution of the movements.

![Figure 77](image_url)

*Figure 77. An aggregated density map of pedestrian movement in the city center of Delft for each day in the data set. The data set has been separated into four bins for each purpose and aggregated using our visualization aggregation operator and maximum combiner. We observe that the shopping bin contributes to the density significantly more than the other bins. Also, we see that pedestrian activity is higher during the weekend.*
Chapter 7 – Results

We focus our attention on an area of special interest containing two museums, ‘Het Prinsenhof’ and ‘Volkenkundig Museum Nusantara’, a historic church called ‘Oude Kerk’ and some café/diners. We separate our data set again, based on the purpose attribute, and aggregate the resulting density maps using the aggregation visualization operator. We show the result in Figure 78 with a close up with the pixel block combiner and a close up with the maximum combiner. This area deviates from the rest of the density map in that there is no clear dominant bin, as can be seen in the image with the maximum combiner. This is due to an increase of activity in the leisure and tourism bin which can be explained due to the presence of the museums, the historic church and the café/diners.

We have shown that our method is not only applicable to moving vessel data, but also to other moving object data sets, where we can also separate and aggregate subsets of the data based on some attributes, similar to what we have done with the vessel data earlier.

Figure 78. A zoom in on an area containing some museums (the left building), a historic church (the right building) and some café/diners. We show both a density map with the pixel block combiner and a density map with the maximum combiner.
Chapter 8.

Discussion & Conclusion

In this thesis we have shown how we significantly reduce the computation speed of the density computation method proposed by Willems et al. [1]. We have also shown how to use this increase in speed to achieve real-time interaction. This allowed us to introduce new visualization techniques and operations on the density maps such as separation, aggregation and animation. And, most importantly, we can change the parameters of these methods and see the results in real-time. We have discussed a geographic information system that can generate multiple density maps with varying parameters such as kernel radius and cell size and combine these using a generic density map combination operator. We have shown a weighted addition operator to show the accumulated density of the input density maps, an anomaly detection operator to help find spatial and temporal anomalies and an absolute difference operator to reveal the difference between two density maps. The result serves as input for the visualization operator, which is a generic operator that turns a density map or a set of density maps into a colored and shaded image. We have shown two general methods for visualizing our density maps. Either we treat the result of the density combination operator as a single density map and visualize it as such or we, in the case of aggregation, apply a color map to each density map individually and combine these images using an additional combination operator. This combination operator takes care of combining the colored image into a final color image that can be shaded by the visualization operator. We have shown three ways to combine these colored images: Through color mixing using the weighted average of the colors, through a maximum operator that only shows the color of the density map with the maximum density per pixel or through the pixel block operator which reveals all layers of data without the color ambiguity inherent to the color mixing combiner.

We have sacrificed some precision for speed in some of the functionality, such as the geographic distance function, the cell subdivision of the data region and the numeric approximation of the kernel. For smaller areas, however, the subdivision of the data region in cells has proven to be sufficient. And similarly, for relatively small kernels with a radius near 2 kilometers, both the geographic distance approximation and the numeric approximation of the kernel are sufficient.

We have shown how our techniques can be used in practice through several hypothetical use cases. One of the problems of a density map is its ambiguity; however, with our use cases we have shown how to make sense of the information in the density maps and effectively reduce the ambiguity using our methods. We have also shown that our method is not exclusive to moving vessel data but can also be used for other data sets such as pedestrian movements. In fact, our method can be applied to any moving object data set in which the moving objects can be expressed with tracks using some movement model.

8.1. Future Work

One of the goals of the follow-up of this project is to find and identify practical issues related to vessel movements and our density maps and implement new techniques to solve these issues. In the following sections we discuss some of these issues and possible extensions for our work that may be implemented in the future.

8.1.1. Small Multiples

For aggregation using the visualization aggregation operator some data may become lost when combining the layers, even using the pixel block operator. To fix this problem we introduce a small mul-
multiples operator that shows each layer as an independent shaded height-map on the screen. The screen space could be divided into a number of equally sized blocks, each of which contains a fully visualized image of one of the bins. This not only gives a visualization without any loss of, but may also help when generating or understanding images using the other combiners. For example, it may give a user some idea which weights to use for the color mixing operator, based on the spatial distribution of the density maps in the small multiples visualization.

### 8.1.2. Directional Information

There are several secondary attributes that can be derived from the movement data. One such derived attribute is the direction of a moving object per line segment. We can use this directional information in several ways. We can somehow generate a density map based on direction, either by taking the average direction of all moving objects and returning the angle of the direction relative to some start vector or we can bin our data set on direction and use the density to compare the spatial distribution of directions.

Our anomaly detection operator as defined in Section 4.3.2 can show spatial and temporal anomalies and even anomalies in attributes such as velocity, but it cannot show anomalies in direction such as a vessel that is moving in the opposite direction in a shipping lane. This does not show up in our anomaly detection operator, but it is a potentially dangerous situation. We can visualize such anomalies by generating a historical direction map and comparing it to some direction map of another sparse data set using the anomaly operator. Note that intersections between shipping lanes are also potentially marked as anomalies.

### 8.1.3. Projection Precision

We have currently chosen to approximate the subdivision of our data region into cells—see Section 6.3.1, however, while this is sufficient for a 400 by 400 kilometer region such as our moving vessel data sets, it is no longer sufficient if we wish to explore data sets that cover a larger region of space. If we want to have a precise spatial subdivision, we require a number of one-dimensional lookup tables that contain the cell’s height (latitude) in degrees and the kernel radius in the latitude direction in number of cells, per cell. We have a problem here though, whenever the data region changes, the cell size changes, the kernel radius changes or even if the data region moves over latitude, we need to recompute some or all of these lookup tables, which may be a relatively expensive operation.

### 8.1.4. Kernel

For our implementation we have used a Gaussian kernel and numerically approximated it—see Section 6.3.3. Another approach would be to find a kernel that can be analytically solved in less time than the Gaussian kernel can be approximated.

Furthermore, we can scale the kernel function such that the volume of a convolved point is equal to one. Currently, we do not do so, however, we may want to introduce this factor, against a small performance hit, to normalize the vessel contribution.

### 8.1.5. Density Maps as Parameter

We currently use two or more density maps as input for the density computation operator. An additional option could be to use density maps as parameters for generating a new density map. One could think of varying the kernel radius locally in a new density map depending on the density of the input density map. This allows the generation of a density map where there is more detail in busy areas due to a small kernel radius while there is less detail due to a larger kernel radius in calmer areas.
References


References


Appendix A

Geographic Space

In this appendix we explain some of the terms and concepts related to geographic space needed to understand our system and apply it to real-world data.

A.1. Geographic Space

Positions on the earth’s surface are expressed by the geographic coordinates longitude $\lambda$ and latitude $\phi$—see Figure 79. Longitude is an angle in degrees in the West to East. A meridian is a line with a constant longitude angle on the earth. The longitude at some meridian is the angle in the West to East direction between the meridian and the so-called prime meridian [2] (Chapter 1)—see Figure 79. The prime meridian—the meridian with a constant longitude of 0°—runs through the Royal Observatory in Greenwich, England. Longitude ranges from $-180^\circ$ to $180^\circ$. The latitude is the angle in degrees in the North to South direction to the equator—see Figure 79. The equator is the line that divides the earth into two hemispheres, the Northern hemisphere and the Southern hemisphere. A Circle of Latitude is the set of all positions on the earth with the same latitude, for example, the equator is the set of all points with a latitude of 0°. Latitude ranges from $-90^\circ$ to $90^\circ$. While geographic coordinates are generally given in degrees, most computations require geographic coordinates in radians. Therefore, in the following, we assume the geographic coordinates in degrees are converted to radial coordinates when used as input for an equation.

The earth is not a perfect sphere, but has a slightly flattened elliptical shape. This means the radius of the earth at some geographic point depends on its latitude. Also, to make things worse, the surface of the earth is not smooth since it has mountain ranges and ocean trenches. A common standard to model the earth in a simplified form is the World Geodetic System (WSG’84). In this model the radius of the earth from the equator to the poles ranges from ~6378.14 km to ~6356.75 km, respectively [14]. Since our datasets cover a relatively small area we can make the assumption that the earth is a perfect sphere with a single radius. We take the Radius of Sphere of Equal Area $R_1$ [14], also known as the Authalic Radius which is the radius of the earth modeled as a perfect sphere with the same area as the elliptical earth. This significantly simplifies geographic calculations. The authalic radius is approximately 6371.01 km. Since we use a single radius for our model of the earth, we refer to the radius as $R$ in this thesis.

Shortest Distance

A great circle is a circle intersecting the earth whose center coincides with the center of the earth—see Figure 79. Each distinct pair of points on the earth’s surface has a unique great circle intersecting both points. The shortest distance between two points on the surface of the earth, called the geographic distance, is the shortest path on its great circle [2] (Chapter 4). This line is called a geodesic.
The shortest distance between two points on a sphere is also called the great circle distance or geodesic distance. The function to compute the great circle distance \( D_{gc} \) is given as follows for the geographic positions \( p_0 = (\lambda_1, \phi_1) \) and \( p_1 = (\lambda_2, \phi_2) \) in radians [2] (Chapter 4):

\[
D_{gc}(p_0, p_1) = R \cdot \cos^{-1}(\cos \phi_1 \cdot \cos \phi_2 \cdot \cos(\lambda_1 - \lambda_2) + \sin \phi_1 \cdot \sin \phi_2)
\]  
(45)

Where \( R \) is the radius of the earth. This function, however, suffers from severe rounding errors for small distances when computed on a system with finite precision. Therefore, we introduce an approximate geographic distance function suitable for small distances. This function is not only less sensitive to rounding errors, it is also faster to compute. It is, however, an approximation and as such may suffer a small error [46]:

\[
D_{gc}(p_0, p_1) \approx R \cdot \sqrt{(\phi_2 - \phi_1)^2 + (\lambda_2 - \lambda_1)^2 \cos^2 \phi_m}
\]  
(46)

Where \( \phi_m \) is the mean latitude \((\phi_1 + \phi_2)/2 \) of the two points. For our purposes the approximation of equation (46) is accurate enough and in the following if we refer to the geographic distance function we refer to the approximation unless stated otherwise. See Appendix B for more details about the accuracy of equation (46).

### A.2. Map Projection

To map the surface of the earth onto a flat surface a map projection function is required. A map projection is a function that projects the surface of a spherical object, usually the Earth, onto a flat surface—a plane. There are three important requirements an ideal map projection has to satisfy [2], (chapter 5): Geodesics have to be mapped to straight lines on the plane, the angle between any two arbitrary lines on the earth’s surface is unchanged by projection to the plane and, lastly, the area of any arbitrary region on the earth is the same when projected to the plane.

Unfortunately, there is no such thing as an ideal map projection. There is no map projection that shows all geodesics as straight lines, preserves the area of all regions and preserves all angles [2], (chapter 6). In fact, no spherical object can be projected onto a flat surface while preserving all above stated properties. This is due to the Gaussian curvature of spheres [2], (chapter 6).

There are two classes of map projections that are important for our model and for which we give a map projection in the following sections. The first class is the class of equal-area map projections that,
Appendix A — Geographic Space

No Ideal Map projections
Take a triangle on a surface of which all sides are the shortest path on the surface between the points. If the sum of all the angles of this triangle is larger than 180° we say the surface has positive Gaussian Curvature and if the sum of all the angles is smaller than 180° we say the surface has negative Gaussian Curvature. This leads to the notion of Gaussian Curvature of a certain point on a surface [2], (chapter 6).

Gauss’s Theorem Egregium [9] states that when a curved surface is projected onto another surface the measure of (Gaussian) curvature in each point of the curved surface remains unchanged. So if a triangle on a curved surface with the sum of its angles larger than 180° is projected onto a plane the sum of its angles in the projection remains larger than 180°. This means that the edges of the triangle—the shortest distance on the curved surface—cannot show up as straight lines on the projection if the area remains the same. If, however, one would make sure that the edges of the triangles are straight lines the total area of the triangle will have become smaller.

This means that if and only if the Gaussian curvatures of both surfaces are identical, the curved surface can be projected onto the other surface with a uniform scaling factor—an identical scaling factor for each point [2], (chapter 6). Since the plane has a Gaussian curvature of zero and the earth has as positive Gaussian curvature along its surface, this means, in practice, that the surface of the earth cannot be projected onto a plane without some distortion.

as the name implies, preserve the area. The second class is the class of conformal map projections that preserve angles.

A method to visualize distortions in a map project are the so-called Tissot’s Indicatrices. These are circles on the surface of the earth. The projected circles are elliptic and show certain properties of the map. The ellipses remain circles in the projection if the map projection preserves angles. All the projected ellipses have the same area as the original circles on the surface of the earth, if the map projection is an equal-area projection. We use these indicatrices to illustrate the properties of the map projections we discuss in the following sections.

Equal-Area Projection
In this thesis we use a cylindrical equal-area map projection called Lambert’s cylindrical equal-area projection [4]—see Figure 80. For a geographic point \((\lambda, \phi)\) in radians, Lambert’s cylindrical equal-area projection is given as follows:

\[
\begin{align*}
  x &= (\lambda - \lambda_0) \cos \phi_0 \\
  y &= \frac{\sin \phi}{\cos \phi_0}
\end{align*}
\] (47)

Where \(\phi_0\) represents the chosen central latitude of the projection and \(\lambda_0\) its central meridian in ra-

---

Figure 80. Lambert’s Cylindrical Equal-area projection of the world. The central longitude and latitude are both set to 0°. The red ellipses are Tissot’s Indicatrices. The indicatrices are not circular, meaning that this projection does not preserve angles. They do, however, have the same area. Image generated using G-Projector [10].
diams. The above projection is not conformal. The corresponding inverse of Lambert’s cylindrical equal-area projection is given as follows:

\[
\lambda = \frac{x}{\cos \phi_0} + \lambda_0 \\
\phi = \sin^{-1}(y \cdot \cos \phi_0)
\] (48)

We have chosen this map projection for its computational simplicity while minimizing visual distortion around the central latitude. Compared to other equal-area map projections, Lambert’s cylindrical equal-area projection shows a fairly intuitive shape of the earth’s surface.

**Conformal Projection**

We also introduce a second map projection \(\mathcal{M}_p\) called the **Mercator Projection** [2], (Chapter 9)—see Figure 81. This projection is a conformal map projection, which does not preserve areas. The Mercator projection is given as follows for longitude \(\lambda\) and latitude \(\phi\) in radians:

\[
x = \lambda - \lambda_0 \\
y = \ln \left( \tan \frac{\phi}{\cos \phi} \right)
\] (49)

Where \(\lambda_0\) is an arbitrary longitude in the center of the map, also in radians. The corresponding inverse of the Mercator projection \(\mathcal{M}_p^{-1}\) is as follows [4]:

![Figure 81. The world projected using the Mercator projection. Note the elongation of the landmass further away from the equator (0° latitude). The central longitude is set at 0°. The red ellipses are Tissot’s Indicatrices. The ellipses maintain their circular form which means that the Mercator projection preserves angles. The area, however, is not preserved as the radius of the circles becomes larger further away from the equator. Note how Greenland appears to have the same size as the whole continent of Africa. Image generated using G-Projector [10].](image-url)
Appendix A — Geographic Space

\[\begin{align*}
\lambda &= x + \lambda_0 \\
\phi &= \tan^{-1}(\sinh y)
\end{align*}\]  
(50)

A.3. Rhumb Lines

We assume our vessels move between points with a constant bearing. To model this behavior we require the concept of Rhumb lines. A Rhumb Line, also known as a loxodrome, is a trajectory on the surface of the earth—or on a projected surface—with a constant bearing intersecting all meridians at the same angle [3]. For example, if a vessel would constantly sail in a North-West direction, its path would be a Rhumb line. A Rhumb line between two points does, in general, not coincide with a great circle, and as such is not the shortest path—see Figure 82.

To reason about and calculate the length of a Rhumb line we require the Mercator Projection. Since this projection preserves angles, it maps a Rhumb line to a straight line. While there is no distortion in the East to West direction in the Mercator projection—see equation (49)—the projection does introduce distortion in the North to South direction: The projection of a Rhumb line of a certain geographic length and bearing is longer than the projection of a similar Rhumb line close to the equator. To calculate the length \(\|p_0 p_1\|\) of a given rhumb line \(p_0 p_1\) from \(p_0 = (\lambda_1, \phi_1)\) to \(p_1 = (\lambda_2, \phi_2)\) we use the following formulas [3]:

\[\|p_0 p_1\| = R \sqrt{1 + \frac{1}{m^2} |\phi_2 - \phi_1|}\]  
(51)

where \(R\) is the radius of the earth and \(m\) is as follows:

\[m = \frac{\text{Distort}(\phi_2) - \text{Distort}(\phi_1)}{\lambda_2 - \lambda_1}\]  
(52)

and Distort(\(\phi\)) is the distortion function of the Mercator projection for some latitude \(\phi\)—see Figure 83. Since we already assumed the earth has a spherical shape we can use the following distortion function [3]:

![Figure 82. A great circle path (The shortest path) in blue and a Loxodrome (a Rhumb Line) in green, between Amsterdam and Washington.](image-url)
This means the length of a rhumb line between two points does not only depend on the position of the two points relative to each other, but also on the absolute latitude of both points.

\[
\text{Distort}(\phi) = \ln \left( \tan \left( \frac{1}{2} \left( \frac{\pi}{2} + \phi \right) \right) \right)
\] (53)

Figure 83. A plot of the distortion function \( \text{Distort}(\phi) \). It can clearly be seen that the area distortion increases when the latitude increases. Also, \( \text{Distort}(\phi) \) tends to infinity when the latitude nears the poles: 
\[
\lim_{\phi \to 90^\circ} \text{Distort}(\phi) = \infty.
\]
Appendix B

Details

B.1. Great Circle Distance Approximation Accuracy

In this section we compare the accuracy of the approximate great circle distance function of equation (46) to the exact great circle distance function of equation (45). The average length and error of 25 different lines with a set length in degrees, but different orientation and location are shown in Table 6. The actual geographic distance is assumed to be the exact result of equation (45). As can be seen from the table not only the absolute error increases with distance, but also the relative error.

The point at which the approximation of equation (46) becomes too inaccurate heavily depends on the size of the kernel and the chosen cell size. We can conclude, however, that for smaller areas—such as the Dutch coast, which is approximately 400 by 400 kilometers—the approximation of equation (46) more than suffices. For large areas care needs to be taken to select the appropriate function based on the kernel radius and the cell size. Generally, however, for the type of data we consider the approximation above is sufficient for computations within the distance of the kernel radius. For distance computations in larger areas the exact formula of equation (45) becomes more attractive.

<table>
<thead>
<tr>
<th>Average length in degrees</th>
<th>Average length (km)</th>
<th>Average error (km)</th>
<th>Approximate error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1.77044</td>
<td>5.61 \times 10^{-8}</td>
<td>3.17 \times 10^{-6}</td>
</tr>
<tr>
<td>1</td>
<td>17.7042</td>
<td>5.62 \times 10^{-5}</td>
<td>3.17 \times 10^{-4}</td>
</tr>
<tr>
<td>2</td>
<td>35.4071</td>
<td>4.50 \times 10^{-4}</td>
<td>1.27 \times 10^{-3}</td>
</tr>
<tr>
<td>3</td>
<td>53.1073</td>
<td>1.51 \times 10^{-3}</td>
<td>2.86 \times 10^{-3}</td>
</tr>
<tr>
<td>5</td>
<td>88.4941</td>
<td>7.05 \times 10^{-3}</td>
<td>7.96 \times 10^{-3}</td>
</tr>
<tr>
<td>10</td>
<td>176.819</td>
<td>5.69 \times 10^{-2}</td>
<td>3.22 \times 10^{-2}</td>
</tr>
<tr>
<td>20</td>
<td>352.272</td>
<td>0.472</td>
<td>0.134</td>
</tr>
<tr>
<td>30</td>
<td>524.919</td>
<td>1.69</td>
<td>0.32</td>
</tr>
<tr>
<td>40</td>
<td>693.171</td>
<td>4.36</td>
<td>0.63</td>
</tr>
<tr>
<td>50</td>
<td>855.208</td>
<td>9.37</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table 6. The average error in absolute length and relative length shown for selected geographic lengths in degrees.

B.2. Automatic Identification System

The vessel movement we use comes from Automatic Identification Systems which periodically broadcast information on the vessel’s status. Every passenger ship and every other large vessel is required to have an AIS transponder installed [24]. AIS is an alternative to radar tracking which is significantly less reliable [47].

AIS is a self-organized tracking system used by vessel Tracking Systems (VTS) and other vessels to determine the location, heading and speed of other vessels. It uses a Self-Organized Time Division Multiple Access (STDMA) to send and organize broadcasts within time slots such that different messages don’t conflict or collide [23]. This means there is no master system such as a control tower for vessels. Using their AIS Transponders, vessels broadcast their static information such as name, call-sign and size every 6 minutes. Dynamic information such as the vessel’s current position, heading and speed are broadcast on more regular intervals ranging from two seconds to three minutes. The up-
The update rate of the dynamic information is based on the status of the ship, whether it is anchored or moving, the speed of the vessel and whether the vessel is changing course. For example, a vessel with a velocity of 20 knots per hour that is changing course has an update rate of two seconds and a vessel with a velocity of 12 knots per hour with a constant bearing has an update rate of 10 seconds while a vessel with the same speed but changing course has an update rate of about 3 seconds.

The messages sent by the AIS transponders of the vessels can be received by other vessels with an AIS transponder or by a VTS designed to receive and process AIS messages. This allows other vessels and tracking systems to observe position, heading and speed of vessels. The system even allows for a VTS to project virtual buoys onto the sea.

Vessels send the following information using their AIS transponders:

**Static and voyage related Information**

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMSI Number</td>
<td>Maritime Mobile Service Identity, a 9 digit identifier to uniquely identify vessels. This identifier is tied to the flag and other external attributes of the vessel.</td>
</tr>
<tr>
<td>IMO Number</td>
<td>International Maritime Organization identification number, a 7 digit identifier to uniquely identify vessels. The IMO number of a vessel never changes and is never assigned to another vessel.</td>
</tr>
<tr>
<td>Call Sign</td>
<td>A seven character radio call sign. This identifier is assigned to the ship by its country of registry.</td>
</tr>
<tr>
<td>Name</td>
<td>An informal name of the ship.</td>
</tr>
<tr>
<td>Ship Type and Cargo Type</td>
<td>The type of the ship and what type of cargo it can carry.</td>
</tr>
<tr>
<td>Ship Size</td>
<td>The dimensions of the ship such as length and width</td>
</tr>
<tr>
<td>Type of Positioning Device</td>
<td>The type of device used to determine the geographic position of the ship.</td>
</tr>
<tr>
<td>Draught</td>
<td>The distance between the water surface and the bottom of the...</td>
</tr>
</tbody>
</table>

Figure 84. A schematic of an AIS transponder, which has two receivers and one transmitter. It uses a Global Navigation Satellite system (GNSS) to retrieve position and time information and uses the ship’s own sensors to gather data such as velocity. The system can be controlled through an external display and interface. The signals broadcasted by the transmitters can be received by other vessels, vessel tracking systems or a repeater station which propagates the signals it receives.
### Appendix B — Details

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Point for Position</td>
<td>The location relative to the ship from where the geographic position is determined.</td>
</tr>
<tr>
<td>Destination</td>
<td>The destination of the ship’s current voyage.</td>
</tr>
<tr>
<td>ETA</td>
<td>The estimated arrival time for the ship’s current voyage.</td>
</tr>
</tbody>
</table>

*Table 7. List of static and voyage related information broadcast by the AIS transponder every 6 minutes [23].

#### Dynamic Information

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMSI Number</td>
<td>Maritime Mobile Service Identity, a 9 digit identifier to uniquely identify vessels. This identifier is tied to the flag and other external attributes of the vessel.</td>
</tr>
<tr>
<td>Time Stamp</td>
<td>The standard universal time of the message.</td>
</tr>
<tr>
<td>Position</td>
<td>The geographic position in longitude ( \phi ) and latitude ( \varphi ) of the ship.</td>
</tr>
<tr>
<td>Speed over Ground</td>
<td>The scalar speed over ground of the ship.</td>
</tr>
<tr>
<td>Position Accuracy</td>
<td>States whether the accuracy of the positioning is high or low.</td>
</tr>
<tr>
<td>Navigational Status</td>
<td>The navigational status of the ship such as at anchor, under way using engine or engaged in fishing.</td>
</tr>
<tr>
<td>Rate of Turn</td>
<td>The rate of turn of the vessel in degrees per minute.</td>
</tr>
<tr>
<td>Course over Ground (COG)</td>
<td>The course of the ship relative to true North.</td>
</tr>
<tr>
<td>True Heading</td>
<td>The heading of the ship as given by its gyrocompass. This may not give the same heading as COG when the ship is turning.</td>
</tr>
</tbody>
</table>

*Table 8. Dynamic Information broadcast by the AIS transponder every 2 seconds to 3 minutes [23].

The static information—Table 7—of the ship is programmed into the AIS transponder and generally does not change. The voyage related information such as destination and ETA is programmed into the transponder at the start of the voyage and generally only changes per voyage. The dynamic information—Table 8—of the ship is measured by the vessel’s own instruments. The geographic position is usually determined using a Global Navigation Satellite System (GNSS) such as the Global Position System (GPS) which can also determine the standard universal time. Other attributes are determined using the ship’s own instruments.
Appendix C

System Architecture

Figure 85. An architectural overview of our geographic information system.
In Figure 85 we show the architectural overview of the geographic information system application surrounding our density maps. The application has been written in C++ using Microsoft Visual Studio 2008 express edition. We utilize the Microsoft .NET framework using C++/CLI. To control the graphics card we use the OpenGL API [20] and for using our shaders we use the Nvidia Cg toolkit and API.

In the following sections we give a brief description of the components of our application as shown above in Figure 85.

**CObjectProperties**
A moving object is represented by a CObjectProperties object containing attributes of the moving object such as ship type as well as bounding boxes in space, time and velocity. The attributes in the attribute list of the CObjectProperties object contain either a continuous value or a discrete index of some enumerations generated by the movement loader. The individual points of the tracks are stored in the video memory of the GPU.

**CMovementContainer**
The CMovementContainer class manages and maintains the movement data in local memory as well as in video memory. It also takes care of the render calls for the moving object data and filters these objects when needed using the object properties of each object.

**CDensityRenderer**
The CDensityRenderer class is an object associated to a single density map. It takes care of generating a single density map or generating multiple density maps and aggregating these. Each density renderer contains a set of CBaseBin objects which represent the bins. This object can be one of three types: CGeneralBin: A bin that does not filter and is used for non-aggregated density maps, CTimeBin: A bin that filters only on time and is used for the sliding time window mode and CGenericAxesBin: A bin that can filter on two generic axes.

This object also maintains and stores the framebuffers related to the density map associated with the density renderer. It also takes care of resizing the framebuffers when the cell size changes. If the density renderer is in the aggregation visualization mode, it can also generate the colored image for the visualization. In other words, the combiner operator of the visualization operator is performed by the density renderer.

**CMainFrameBufferStorage**
All Framebuffer objects related to the screen are stored and maintained by the CMainFrameBufferStorage object.

**MView**
The MView class is the main view class and controls all functionality of the application. The user interface only communicates with this class. The MView class also manages the density renderers, the shader container, the movement container and the main framebuffer storage. The MView class also initiates the density map generation, handles the visualization and handles the density combination operator and the visualization operator.

**CMovementLoader**
The CMovementLoader class is used by the MView object to instantiate and fill a CMovementContainer object.
Appendix D

Shaders

In this appendix we discuss the most important parts of our shaders. First we start with a short introduction to Cg and shaders required to understand the code below. Following this we treat the main density computation shaders.

D.1. Introduction to Cg

The Nvidia shader language, “C for graphics” (Cg), is a C-like language in that it has the same syntax as Ansi C—except for switch statements—including some Cg specific language constructs. One of the more important language constructs unique to Cg are the vector and matrix types. These types are designated by a primitive type with a numeral or two numeral suffixes—see Figure 86. The elements of the vectors can be indexed by the so-called swizzling operator which works by suffixing a dot followed by a sequence of letters: x, y, z or w. For example, \( \mathbf{V}.xy \) returns a two-dimensional vector containing the x and y element of \( \mathbf{V} \)—see Figure 86. Matrices are indexed as normal two-dimensional arrays are in C.

```c
float2 vector; // Two-dimensional floating point vector
float3x3 matrix; // Three by three floating point matrix

float2 newVector = vector.xx; // newVector == ( vector.x, vector.x )
```

Figure 86. A two-dimensional floating point vector variable and a three by three floating point matrix. Using the swizzling operator a new two-dimensional variable is, for both elements, set to the first element of the first vector variable.

There is also a set of types that represent texture surfaces the shader can sample from. We use the sampler1D type for one-dimensional textures, the sampler2D type for two-dimensional textures and the samplerRECT type for non power of two-dimensioned textures. Using texture coordinates, texels can be sampled from these surfaces using specialized sampler functions such as tex1D and tex2D.

A number of specialized vector operations are defined in Cg. These are operations that perform some computation per vector element in parallel. One of these operators is vector division. Take for example two three-dimensional vectors \( \mathbf{V} \) and \( \mathbf{W} \). The operation \( \mathbf{V}/\mathbf{W} \) is then defined as the element-wise division of the vectors as follows: \( \mathbf{V}/\mathbf{W} = (\mathbf{v}.x/\mathbf{w}.x, \mathbf{v}.y/\mathbf{w}.y, \mathbf{v}.z/\mathbf{w}.z) \).

Geometry shaders have a specialized instruction to create or emit vertices called emitVertex. This function can create a vertex with a number of attributes such as position, texture coordinates, etc. These attributes are explicitly bound to hardware registers using binding semantics—see Section 6.1.2.

D.2. Density Shaders

Our geometry shader that creates and parameterizes the bounding boxes takes a number of attributes in a so-called attribute array as input in the form of a line segment determined by the LINE prefix and outputs a triangle strip determined by the TRIANGLE_OUT prefix. This means the attribute
Appendix D — Shaders

Arrays contain a set of two positions and a set of two parameter vectors. The position is a four-dimensional vector containing the longitude, latitude, time and velocity. The parameter vector is a vector of undefined parameters intended for later use. Our geometry shader is shown below.

```cpp
// ********************
// Geometry Shader
// ---
// Takes a line as input and outputs a bounding box based on the kernel radius iGeokernelRadius

TRIANGLE_OUT LINE void densityGeometry( AttribArray<float4> iPosition : POSITION,
                                      AttribArray<float4> iParameters : TEXCOORD0,
                                      uniform float iGeokernelRadius,
                                      uniform float4 iGeoProjectionParameters,
                                      // (x, y) degrees per cell for longitude (x) and latitude (y)
                                      // (z, w) Size of the kernel in cells
                                      uniform float4 iGeoProjectionParameters2,
                                      // (x, y) Origin of the FBO
                                      // (z, w) Size of the kernel in degrees,
                                      // uniform float2 iGeoMapProjectionCenter
                                      // The central point used for map projection
                                      #if defined(DENS_TYPE_AGGREGATION) || defined(DENS_TYPE_TIME_WINDOW)
                                          , uniform float2 iGeoTimeWindow
                                      #endif
                                      // Time window for the data set; x is the start time, y is the end time.
                                      #ifdef DENS_TYPE_AGGREGATION_VELOCITY
                                          , uniform float2 iGeoVelocityWindow
                                      #endif
                                      // Velocity window for the data set; x is the minimum velocity,
                                      // y is the maximum velocity.
                                      #endif
                                      }

// ********************
// Constructs a quad for a given line segment

inline void constructQuad( float4 a_Point0, // Position information of the first point
                           float4 a_Parameters0,
                           float4 a_Point1, // Position information of the second point
                           float4 a_Parameters1,
                           iGeokernelRadius, iGeoProjectionParameters,
                           // (x, y) degrees per cell for longitude (x) and latitude (y)
                           // (z, w) Size of the kernel in cells
                           iGeoProjectionParameters2,
                           iGeoMapProjectionCenter
                                      // The central point used for map projection
                                      )
{
    // Construct bounding box for line segment iPosition[0] (p0) to iPosition[1] (p1)
    //
    // v1
    // +--------------------------
    // |      | p0                |
    // |      | -----------------|
    // v3
    //
    // We uses two different spaces, one for the computation and one for the visualization
    // The computation space uses the space in which the data is supplied and the
    // visualization space is a transformed space such that the dataset fits "exactly" on
    // the screen.

    // ****
    // Compute values for computation
    // Construct the normal
```
float4 position0 = a_Point0;
float4 position1 = a_Point1;
float kernelSize = iGeokernelRadius;

#ifdef USE_GEOGRAPHIC_SPACE
// Project the positions to mercator projection space
float4 mercPosition0 = float4( mapProjectionMercator( position0.xy, iGeoMapProjectionCenter.x ), position0.zw );
float4 mercPosition1 = float4( mapProjectionMercator( position1.xy, iGeoMapProjectionCenter.x ), position1.zw );

// Compute the length of the line segment in mercator projection space
float mercatorLength = distance( mercPosition0.xy, mercPosition1.xy );
float lineLength = abs( rhumbLength( position0.xy, position1.xy ) );
#else
float4 mercPosition0 = position0;
float4 mercPosition1 = position1;
float lineLength = distance( position0.xy, position1.xy );
#endif

// Normal over the geographic coordinates
float2 normal = normalize( position1.xy - position0.xy );
float2 kernelNormal = float2( normal.x * iGeoProjectionParameters2.z, normal.y * iGeoProjectionParameters2.w );
float2 kernelBiNormal = float2( normal.y * iGeoProjectionParameters2.z, -normal.x * iGeoProjectionParameters2.w );

// Convert km/h to km/s
float v0 = a_Point0.w / 3600.0f;
float v1 = a_Point1.w / 3600.0f;
float veloDiff = { v1 - v0 };
float timeDiff = a_Point1.z - a_Point0.z;
float acceleration = veloDiff / timeDiff;
float velocity = ( lineLength / timeDiff ) - ( veloDiff / 2.0f );

#ifdef USE_GEOGRAPHIC_SPACE
float4 movementParameters = float4( velocity, acceleration, lineLength, mercatorLength );
#else
float4 movementParameters = float4( velocity, acceleration, lineLength, lineLength );
#endif

if( lineLength <= .001f || ( velocity <= .001f && acceleration <= .0001f ) )
{
    constructSplatQuad( a_Point0, a_Parameters0, a_Point1, a_Parameters1, iGeokernelRadius, iGeoProjectionParameters, iGeoProjectionParameters2, iGeoMapProjectionCenter );
    return;
}

// ****
// Compute values for visualization
float2 visPosition0 = ( a_Point0.xy - iGeoProjectionParameters2.xy ) / iGeoProjectionParameters.xy;
float2 visPosition1 = ( a_Point1.xy - iGeoProjectionParameters2.xy ) / iGeoProjectionParameters.xy;
float2 visKernelNormal = normalize( normal );
float2 visKernelNormal = float2( normal.x * iGeoProjectionParameters2.z, normal.y * iGeoProjectionParameters2.w );
float2 visKernelBiNormal = float2( normal.y * iGeoProjectionParameters2.z, -normal.x * iGeoProjectionParameters2.w );

// ****
// Compute the proportion of the extended parts of the line segments of the quad to the
// line in the screen space to get correct proportions in the computation space
float lineExtent = ( lineLength / length( visPosition1.xy - visPosition0.xy ) )
* length( visKernelNormal );
float leftLineParameter = -lineExtent;
float rightLineParameter = lineLength + lineExtent;

// ****
// Emit Vertices for the quad
float2 vertex; // Vertex in world space
float2 projectedVertex; // Vertex in screen space

// Vertex 1 (TopLeft)
vertex = float2( ( position0.x - ( kernelNormal.x ) ) - ( kernelBiNormal.x ) );
projectedVertex.x = ( visPosition0.x - ( visKernelNormal.x ) ) - ( visBiNormal.x );
projectedVertex.y = ( visPosition0.y - ( visKernelNormal.y ) ) - ( visBiNormal.y );
projectedVertex = mul( glstate.matrix.mvp, float4( projectedVertex, .0f, 1.0f ) ).xy;
emitVertex( projectedVertex : POSITION, 
a_Parameters0 : TEXCOORD0, 
mercPosition0.xyz : TEXCOORD1, 
mercPosition1.xyz : TEXCOORD2, ...
Our density fragment shader is as follows:

```c
// Vertex 2 (TopRight)
vertex.x = ( position1.x + ( kernelNormal.x ) ) - ( kernelBiNormal.x );
vertex.y = ( position1.y + ( kernelNormal.y ) ) - ( kernelBiNormal.y );
projectedVertex.x = ( visPosition1.x + ( visKernelNormal.x ) ) - ( visBiNormal.x );
projectedVertex.y = ( visPosition1.y + ( visKernelNormal.y ) ) - ( visBiNormal.y );
emitVertex( projectedVertex, float4( projectedVertex, 0.f, 1.0f ) ).xy;
```

We use pre-compiler constructs such as `#ifdef` to avoid having to duplicate and manage multiple shaders that are almost identical. This allows us to maintain multiple instances of the same shader with minor differences by recompiling the same code with different pre-compiler definitions. For example, our geometry shader behaves slightly different for normal density maps, time filtered densities with minor differences by recompiling the same code with different shader definitions. This allows us to maintain multiple instances of the same shader that are almost identical. This allows us to maintain multiple instances of the same shader with minor differences by recompiling the same code with different pre-compiler definitions.

The function `constructSplat` is a function similar to `constructQuad`, only a square quad is constructed for splatting.

Our density fragment shader is as follows:

```c
// Vertex 2 (TopRight)
vertex.x = ( position1.x + ( kernelNormal.x ) ) - ( kernelBiNormal.x );
vertex.y = ( position1.y + ( kernelNormal.y ) ) - ( kernelBiNormal.y );
projectedVertex.x = ( visPosition1.x + ( visKernelNormal.x ) ) - ( visBiNormal.x );
projectedVertex.y = ( visPosition1.y + ( visKernelNormal.y ) ) - ( visBiNormal.y );
emitVertex( projectedVertex, float4( projectedVertex, 0.f, 1.0f ) ).xy;
```
// Attributes used if the line segment has to be splat
{
    float density : COLOR;
};

// Output structure for the fragment shader
struct pixel
{
    float density : COLOR;
};

// Fragment Shader

// Computes density per cell
pixel densityFragm
tent( IN, uniform sampler1D iKernel : TEXUNIT0, #ifdef DENS_TYPE_TIME_WINDOW
    uniform sampler1D iTimeWeight : TEXUNIT1, #endif
    uniform float iFragkernelRadius, unifo
m float2 iFragMapProjectionCenter // The central point used for map projection
    #if defined(DENS_TYPE_AGGREGATION) || defined(DENS_TYPE_AGGREGATION_VELOCITY)
        uniform float2 iFragmentTimeWindow // Time window for the data set; x is the start time, y is the end time.
    #endif
    #ifdef DENS_TYPE_AGGREGATION_VELOCITY
        uniform float2 iFragmentVelocityWindow // Velocity window for the data set; x is the minimum velocity,
    #endif

    pointPosition = IN.parameterization; // Current location of the cell

    #if defined(DENS_TYPE_AGGREGATION) && defined(DENS_TYPE_AGGREGATION_VELOCITY)
        float density = integrateDensity( IN.point0, IN.point1, IN.movementParameters, IN.parameters, pointPosition, iFragkernelRadius, IN.lineParameter, iFragMapProjectionCenter, IN.splatParameters, iKernel, iFragmentTimeWindow, iFragmentVelocityWindow );
    #elif defined(DENS_TYPE_AGGREGATION)
        float density = integrateDensity( IN.point0, IN.point1, IN.movementParameters, IN.parameters, pointPosition, iFragkernelRadius, IN.lineParameter, iFragMapProjectionCenter, IN.splatParameters, iKernel, iFragmentTimeWindow );
    #elif defined(DENS_TYPE_TIME_WINDOW)
        float density = integrateDensity( IN.point0, IN.point1, IN.movementParameters, IN.parameters, pointPosition, iFragkernelRadius, IN.lineParameter, iFragMapProjectionCenter, IN.splatParameters, iKernel, iTimeWeight, iFragmentTimeWindow );
    #else
        float density = integrateDensity( IN.point0, IN.point1, IN.movementParameters, IN.parameters, pointPosition, iFragkernelRadius, IN.lineParameter, iFragMapProjectionCenter, IN.splatParameters, iKernel );
    #endif

    OUT.density = density;
    return OUT;
};

// ************************
// Computes the density
//
// a_Point0/1
// a_Point0/1 is a three-dimensional point where xy represent the 2D-location in Mercator space and z the time.
// a_Point0 is the first point of the movement and a_Point1 is the second point of the movement (a_Point0.z <= a_Point1.z)

// a_MovementParameters
// a_MovementParameters is a three-dimensional vector where x is the average speed of the movement, y is the acceleration and z is the total length of the movement in km

// a_Parameters
// a_Parameters is a four-dimensional vector containing optional parameters.

// a_PointPosition
// a_PointPosition is a two-dimensional location where the integration takes place

// a_Radius
// a_Radius is the radius of the kernel

// a_LineParam
// a_LineParam is the line parameter on the line closest to the cell in geographic space

// a_KernelLookupTable
// a_KernelLookupTable is a lookup table in the form of a one-dimensional texture containing the kernel function

// a_TimeWeightLookupTable
// a_TimeWeightLookupTable is a lookup table in the form of a one-dimensional texture containing the time weight function. This parameter is only available when the density type is set to time window

// a_TimeWindow
APPENDIX D — SHADERS

/**
 * integrateDensity
 * 
 * @param a_Point0 The start point
 * @param a_Point1 The end point
 * @param a_MovementParameters The movement parameters
 * @param a_Parameters Additional parameters
 * @param a_PointPosition The point position
 * @param a_Radius The radius
 * @param a_LineParam The line parameter
 * @param a_SplatParameters The splat parameters
 * @param a_KernelLookupTable The kernel lookup table
 * @param a_TimeWeightLookupTable The time weight lookup table
 * @param a_TimeWindow The time window
 * @param a_VelocityWindow The velocity window
 * 
 * @returns The integrated density
 */

inline float integrateDensity( float4 a_Point0, float4 a_Point1, float4 a_MovementParameters, float4 a_Parameters, float2 a_PointPosition, float a_Radius, float a_LineParam, float2 a_SplatParameters, sampler1D a_SplatLookupTable, sampler1D a_TimeWeightLookupTable, float2 a_TimeWindow, float2 a_VelocityWindow )
{
    if ( a_SplatParameters.x == 1.0f )
    {
        return splatDensity( a_Point0, a_Point1, a_SplatParameters, a_MovementParameters, a_KernelLookupTable, a_TimeWeightLookupTable, a_TimeWindow, a_VelocityWindow);
    }
    float2 normal = normalize( a_Point1.xy - a_Point0.xy ); // Normal in mercator space
    float mercFactor = a_MovementParameters.w / a_MovementParameters.z;
    if( a_MovementParameters.z == 0.0f )
        mercFactor = 1.0f;
    float distanceToLine = geoDistance( mapProjectionMercatorInverse( a_Point0.xy + ( normal * min( max( mercFactor * a_LineParam, 0.0f ), a_MovementParameters.w ) ), a_PointPosition ) );
    float lineLength = a_MovementParameters.z;
    float startTime = timePositionRelative( max( .0f, a_LineParam - projectedRadius ), a_MovementParameters.x, a_MovementParameters.y, t0, t1 );
    float endTime = timePositionRelative( min( lineLength, a_LineParam + projectedRadius ), a_MovementParameters.x, a_MovementParameters.y, t0, t1 );
    if( abs( a_MovementParameters.y ) > .0f )
    {
        float time0 = velocityTime( a_VelocityWindow.x, a_MovementParameters.y, time0, t0, t1 );
        float time1 = velocityTime( a_VelocityWindow.x, a_MovementParameters.y, endTime, a_TimeWindow[1] - t0 );
        startTime = max( startTime, a_TimeWindow[0] - t0 );
        endTime = min( endTime, a_TimeWindow[1] - t0 );
    }
    return splatDensity( a_Point0, a_Point1, a_SplatParameters, a_MovementParameters, a_KernelLookupTable, a_TimeWindow, a_VelocityWindow,
    a_MovementParameters.x, a_MovementParameters.y, a_SplatParameters, a_KernelLookupTable, a_TimeWindow, a_TimeWeightLookupTable, a_TimeWindow, a_VelocityWindow );
}

104
float time0 = velocityTime(a_VelocityWindow.y, a_MovementParameters.y, a_MovementParameters.x);
float time1 = velocityTime(a_VelocityWindow.y, a_MovementParameters.y, a_MovementParameters.x);
startTime = max(startTime, min(time0, time1));
endTime = min(endTime, max(time0, time1));
#endif

float timeDiff = max(0.0f, endTime - startTime);
float timeStepSize = (timeDiff / (float)NUM_INTERVALS) / 2.0f;
if(timeStepSize <= 0.0f)
    return 0.0f;
float currentTime = startTime;
float density = 0.0f;
float linePos = mercFactor * min(linenLength, linePosition(currentTime, a_MovementParameters.x, a_MovementParameters.y));

#ifdef DENS_TYPE_TIME_WINDOW
float timeDiff = timeWeightFunction(a_TimeWeightLookupTable, a_TimeWindow, currentTime + t0);
#endif

#ifdef USE_GEOGRAPHIC_SPACE
float prevValue = timeWeight * kernel(geoDistance(mapProjectionMercatorInverse(a_Point0.xy + (normal * linePos), a_projectionCenter.x), a_PointPosition), timeWeight * a_Radius, a_KernelLookupTable);
#else
float prevValue = timeWeight * kernel(distance(a_Point0.xy + (normal * linePos), a_PointPosition), timeWeight * a_Radius, a_KernelLookupTable);
#endif

else
#endif

float midValue = kernel(geoDistance(mapProjectionMercatorInverse(a_Point0.xy + (normal * linePos), a_projectionCenter.x), a_PointPosition), a_Radius, a_KernelLookupTable);

float nextValue = kernel(geoDistance(mapProjectionMercatorInverse(a_Point0.xy + (normal * linePos), a_projectionCenter.x), a_PointPosition), a_Radius, a_KernelLookupTable);

#endif

density += (timeStepSize) / 3.0f * (prevValue + 4.0f * midValue + nextValue);
prevValue = nextValue;
};
    return density;
}