Masters Degree in Informatics Engineering

Visualization Techniques for Big Data

Hugo Dinis Pereirinha da Silva Amaro
hamaro@student.dei.uc.pt

Juri:
Bernardete Ribeiro

Luis Filipe Vieira Cordeiro

Date: 2 Setember 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Geographic information enables retailers to make informed decisions. Visualizing geographic and demographic relationships, supports market analysis, site selection, merchandising, distribution, delivery, among others. This thesis presents a visualization system to explore sales records from the major retailer in Portugal, in a geographic context, which integrates administrative and demographic information.

We improve upon previous implementations by combining: First the flexibility and scalability of OpenCL kernels, used to process the original dataset in real visualization time, eliminating the necessity for preprocessing and second the use of modern rendering methodologies, through the OpenGL API, to produce a high detailed and information rich visualization.

Our results leave no doubt to the advantages of parallel processing, even in low end GPUs, and to the flexibility and visual quality attainable when researchers take the extra step of researching and implementing adequate rendering techniques for the programmable graphic pipeline.

Keywords

Anti-aliasing, Big Data, Database, GIS, GLSL, GPGPU, GPU, Heatmap, OpenCL, Shaders, Visualization

Resumo

Informação geográfica permite que as decisões comerciais relativamente ao mercado de retalho sejam feitas de forma informada. A visualização de relações demográficas, apoia a análise de mercado, a escolha de localizações, comercialização, distribuição, entregas, entre outras. Esta Tese apresenta um sistema de visualização de informação para esplorar os registos de vendas do maior retalhista em Portugal, num contexto geográfico, que integra também informação demográfica e administrativa.

Este trabalho apresenta melhorias sobre outras implementações por combinar: Primeiro a flexibilidade e escalabilidade dos Kernels de OpenCL, usados para processar os dados originais em tempo de visualização eliminando a necessidade de um pré-processamento, e segundo, o uso de metodologias de renderização modernas, através da API OpenGL, para produzir uma ferramenta de visualização de elevada qualidade gráfica e rica em informação.

Os nossos resultados não deixam dúvidas sobre as vantagens do processamento paralelo, mesmo em GPUs de baixa gama, nem sobre a flexibilidade e qualidade visual que os investigadores são capazes de alcançar, ao aplicar tempo na pesquisa e implementação de técnicas apropriadas de rendering que façam uso do pipeline programável dos chips gráficos.

Keywords

Anti-aliasing, Base de Dados, Big Data, GIS, GLSL, GPGPU, GPU, Mapa de Calor, OpenCL, Shaders, Visualização de Informação

Index

1. Introduction 1

1.1. Context 1

1.2. Motivation 2

1.3. Objectives 4

1.3.1. Requirements 6

1.4. APIs and Third Party Software 7

1.4.1. Database 7

1.4.2. GPU API 7

1.5. Document Outline 8

2. State of the Art 9

2.1. Scheepens "GPU - Based track visualization of multivariate moving object data" 9

2.2. Buschmann "Hardware-accelerated attribute mapping for interactive visualization of complex 3D trajectories" 11

2.4. Resume and comparison 12

3. Data Sources 15

3.1. Sales Dataset 15

3.2 Complementary data gathered 15

3.2.1. Stores Location Data 16

3.2.2. Clients Postal Codes Location Data 18

3.2.3. Shapes of OpenStreetMaps 20

3.2.4. Demographic Data 20

4. Programming models overview 23

4.1. OpenGL 23

4.2. OpenCL 26

4.2.1. Architecture 26

4.2.2. OpenCL C 29

4.2.3. OpenCL/OpenGL interoperability 30

5. Implementation 31

5.1. Database and Data Stream 33

5.1.1. Database Configuration and Structure 33

5.1.3. File system day exports 36

5.2. Projections 36

5.2.1. Geo-Projection 36

5.2.2. Projection Matrix 37
5.2.3. Un-project 39
5.3. Geographic Heatmap Concept and First Approach 40
5.4. Pixel Heatmap - Graphics API 44
5.5. Pixel Heatmap - OpenCL API 47
5.5.1. Optimization of kernel execution 52
5.6. Coloring Geographic Heatmaps 53
5.6.1. Color and perception 54
5.6.2. Anti-aliasing 56
5.7. Displaying Administrative and Demographic Information 59
5.7.1. Administrative Shapes and Boundaries 59
5.7.2. Mouse Over Location Identification 60
5.7.3. Displaying Demographic Data 61
5.8. Miscellaneous Features 64
5.8.1. Filtering Stores 64
5.8.2. Screen Capture and Video Recording 64
5.8.3. User Interface 65
6. Results 67
6.1. Anti-aliasing and Geographic Heatmap Coloring 67
6.1.1. Performance Considerations 71
6.2. Identification and Confirmation of Events 71
6.2.1. Distant Clients 72
6.2.2. Store Opening 73
6.3. Demographic Visualization Modes 76
6.3.1. StarsShader 77
6.3.2. Choropleth Maps Comparison 79
6.3.3. Performance Considerations 82
6.4. Sales and Sales over Quantity 83
7. Discussion and conclusions 85
7.1. Future Work 85
7.1.2. Application improvements 86
7.1.3. Alternative directions 86
References 89

List of Figures

Figure 1 - Changes in Key GPU Properties over Time (GPU Gems 2 [2])

Figure 2 - Number of research projects matching the keyword GPU over the years.

Figure 3 - Example of Scheepens application renderings. Quoting his legend for this figure, "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)"

Figure 4 - Scheepens process for drawing a segment of a vessel tracks onto the density field FBO (source: Scheepens 2010 Thesis).

Figure 5 - Renderings with different style and mapping configurations, made with Buschmann visualization tool. (source: Buschmann Jan 2014).

Figure 6 - Simplified structure of the application and scripts implemented to match Stores present in our Sales Dataset with the Store data gathered by analysis of the source and JSON data files of Sonae Group's Brands individual Web Sites.

Figure 7 - Screenshot of the additional geo-referencing step interface, created to help match GPS coordinates with Stores.

Figure 8 - Control tables created to keep track of the Geocoding process.

Figure 9 - INE data selection and export interface.

Figure 10 - Diagram of the most important stages in the OpenGL 4.0+ pipeline.

Figure 11 - Simplified rendering pipeline model, with vertex and fragment shader stages.

Figure 12 - Simplified rendering pipeline model, with vertex, geometry and fragment shader stages.

Figure 13 - Simplified Mapping of OpenCL onto a GPU device (source [AMD 22]).

Figure 14 - Example of usage of "dummy" threads in the queueing of OpenCL kernels to optimize execution.

Figure 15 - Minimalist view of the steps relevant to the creation, management and drawing of the information visualization model components. On the left the thread responsible for Database connection, Data Files reading and parsing/streaming the data to Thread2. On the right the thread responsible for uploading new data to video memory, execute operations kernel programs, draw the correct information to pre-buffers (FBOs) and finally draw the information as layers onto the Screen Buffer. Notice that the main cycle presented in Thread2 is our per frame cycle, which we will refer to as Render Cycle also.

Figure 16 - Distribution of records in the database, one Database table per Day.

Figure 17 - Projecting From GPS coordinates to Screen Space

Figure 18 - Model View Projection Matrix transformation to Clipping Space and Clipping Space Projection onto the 2D Screen Space

Figure 19 - Un-projection resultant line segment, still in Projection Space.

Figure 20 - Two different rendering types of geographic heatmaps. On the left discretized units, on the right continuous coloring.

Figure 21 - Description of the Draw Pass used to render the geographic heatmap. A Geometry Shader generates Quad geometry, one Quad per Store location. Quad sizes vary according to the accumulated value for the Store fetched from the Heatmap Texture. The fragment shader colors each Quad according to the 2D Gaussian Function, centered at texture coordinates (0.5, 0.5) and with an amplitude value fetched from the Heatmap Texture.

Figure 22 - On the left our first color mapping Texture for the values calculated during the Quad generation process, on the right our first rendering result.

Figure 23 - HeatTexture structure. Real colors were inverted.

Figure 24 - Abstract representation of how Sales Vertices are positioned above the corresponding Store's pixel, in order to be added to the HeatTexture during the blending process. Blending configuration for the addition is described in Source 3.

Figure 25 - Data structures defined to use with our OpenCL Kernels.

Figure 26 - Structure of the buffers used to upload Sales data to the Video Memory. This buffers will be input arguments of the AddKernel.

Figure 27 - Example of how parallel reduction works in the GetTextureMaxKernel. In this example the Black squares in Step 7 represent the final values returned by the Kernel execution that must be still iterated in the CPU to determine the actual maximum value.

Figure 28 - Aliasing introduced by discretizing the colorspace.

Figure 29 - Application rendering layers.

Figure 30 - Discrete color spaces used to color the Geographic Heatmaps when rendered to the Screen Buffer. The top is used when only one heatmap is rendered and the bottom is used when two heatmaps are rendered at the same time.

Figure 31 - The split-complementary color scheme we picked to color the heatmaps when visualized simultaneous.

Figure 32 - Anti-aliasing of plateaus and a border line delimiting the different ranges.

Figure 33 - Illustration of the Edge interpolation algorithm.

Figure 34 - Shortest Distance from point to line illustration [16].

Figure 35 - Variation of color and the color's alpha value near the interpolated edge in order to render the anti-aliased line.

Figure 36 - Variation of color and the color's alpha value near the interpolated edge in order to render the anti-aliased line. Notice the purple color in the center, result of the blending configuration.

Figure 37 - On the left the Triangulated Country Shape rendered in grey color. On the right, many Civil Parishes (freguesias) on the Lisbon area rendered with different shades of grey denoting their different populations.

Figure 38 - The line generation process implemented through a Geometry Shader. Notice how the line segments caps are shaped to close the corners when connected to another line segment, this is only possible because the Geometry Shader has the information regarding the adjacent segments.

Figure 39 - On the left a plot of the mathematical equation used to color the Quads in the Fragment Shader of the StarsShader Program. On the right the rendering result using that equation.

Figure 40 - Both left and right contain the same exact area representing the same exact values. On the left the GreyShader was used and on the right the ChoroplethShader.

Figure 41 - Aliased line on the left, magnified. On the right an antialiased line obtained with our implemented algorithm, also magnified.

Figure 42 - Comparison between vertical curves. On the left a smooth line rendered by our application, and on the right a smooth line rasterized with Adobe Photoshop.

Figure 43 - Comparison between smooth horizontal curves. On the bottom a smooth line rendered by our application, and on top a smooth line rasterized with Adobe Photoshop.

Figure 44 - Comparison between almost straight curves. On the left a smooth line rendered by our application, and on the right a smooth line rasterized with Adobe Photoshop.

Figure 45 - Comparison between tight curves. On the left a smooth line rendered by our application, and on the right a smooth line rasterized with Adobe Photoshop.

Figure 46 - Client Geographic Heatmap drawn with plateau transparency and anti-aliased lines. The shaded shapes seen through the heatmap are the Civil Parishes colored according to their populations (not normalized).

Figure 47 - Both Stores and Clients Geographic Heatmaps overlaid.

Figure 48 - Examples of screen captures depicting Clients who have to travel far to purchase their products. On top an overview of the Southern area of Portugal. Below is the Geographic area around the City Viseu.

Figure 49 - Picture displaying Sales distribution near the days peak, the day before a new Store opens in the center of the red circle.

Figure 50 - Picture displaying Sales distribution near the days peak, the day a new Store opened (middle left yellow bubble).

Figure 51 - Stores heatmap the day before (top) and the opening day of the new Store (bottom).

Figure 52 - Clients heatmap resultant of filtering out all other Stores except for the new one. On the bottom an overview of the whole south half of the Country, and on top a more local visualization, spanning the country's width.

Figure 53 - Porto City area map with an overlay of the StarsShader result map, with increased contrast, brightness and transparency, over the choropleth Population map colored with shades of grey. The Green, Yellow and Red circles represent Good, Median and Bad correlation points. Only the most relevant were highlighted.

Figure 54 - Lisbon City area map with an overlay of the StarsShader result map, with increased contrast, brightness and transparency, over the choropleth Population map colored with shades of grey. The Green,

Yellow and Red circles represent Good, Median and Bad correlation points. Only the most relevant were highlighted.

Figure 55 - Comparison between greyscale coloring (top) and procedural texture generation methods when representing Population of Civil Parishes in the Lisbon area.

Figure 56 - Low populated areas are indistinguishable from each other using the Procedural Texture Generation method to color the Choropleth map.

Figure 57 - Visual confusion of the Procedural implementation (bottom) when compared with the more appealing greyscale coloring (top).

Figure 58 - Zoomed out comparison of the greyscale and procedural implementations of the Choropleth map.

Figure 59 - Visual representation of the Sale Value over Quantity variable's distribution over the northern part of the Country.

Figure 60 - Visual representation of the Sale Value over Quantity variable's distribution over the southern part of the Country.

List of Tables

Table 1 - Measurement of the impact our geographic heatmap coloring and anti-aliasing algorithms, had on performance. Values are in Frames Per Second, the bigger the better.

Table 2 - Performance results for each of the different demographic visualization modes. Values are in Frames Per Second, the bigger the better.

Acronyms

IPN - Instituto Pedro Nunes

GIS - Geographic Information System

SP - Shader Program

FS - Fragment Shader

VS - Vertex Shader

GS - Geometry Shader

GLSL - OpenGL Shading Language

SPMD - Single Program Multiple Data

CPU - Central Processing Unit

GPU - Graphical Processing Unit

RAM - Random Access Memory

VBO - Vertex Buffer Object

VAO - Vertex Array Object

FBO - Frame Buffer Object

API - Application Programming Interface

MVP - Model View Projection Matrix

FOV - Field Of View

1. Introduction

This document presents the work done and results obtained in the scope of a Master Thesis on the subject Visualization of Big Data. Thesis advisor were Professor Pedro Cruz and Professor Penousal Machado, both from the Department of Informatics Engineering of the University of Coimbra. This work is part of a research project from Instituto Pedro Nunes (IPN) partnering with Sonae SA, in the field of Information Visualization.

This chapter is divided in four chapters. The first describes in more detail the research project in which this Thesis was framed, the second introduces aspects common to data visualization projects that we wish to solve with a different approach and the motivation for that approach, the third chapter describes planning and how that plan evolved over the two semesters, and finally, the fourth chapter describes the structure of this document.

1.1. Context

The work described in this document was framed in a research project involving a partnership between IPN and Sonae SA.

Sonae SA, is the main retailer in Portugal, with an annual revenue of €5.718 billion and over 700 stores. The main objectives of the project are to analyze and explore data sets with sales records form Sonae stores, in order to plan and implement visualization tools to find or highlight patterns and/or relevant information both unknown or as to confirm predictions.

The team responsible for carrying out the project is led by teachers Penousal Machado (Scientific Director) and Professor Pedro Cruz (Project Manager), and consists of three elements, two students of PhD and one of Masters degree in which I am included.

Responsibilities within the team are equivalent, in respect to knowledge of everything that concerns to the project, its status and its objectives. Each team member has to be on pair with each others latest developments, so that everyone can contribute in the discussion of problems and solutions regarding the overall of the project.

In terms of the actual implementation the visualization tools, each member was responsible for the planning and implementation of a different visualization application. Yet, as before, members kept track of each other work closely in order to discuss, criticize, and contribute to the other members implementation, both daily and in the weekly meetings. This way the team was able to develop multiple applications at the same time in a reasonable time frame and at the same time make the best of each one's skills and experience.

This document refers only to the work done by me, and from this, is restricted to the necessary tasks directly related to the visualization tool implemented by myself .

1.2. Motivation

Data visualization is everywhere, particularly in every scientific research area. From Bar charts to complex visualization models, researchers have developed numerous and innovative ways of exploring data sets, highlight patterns, present experiment results, among others. The works of Wong (1994) and Chen (2007) are both extensive and complete in terms of examples, and their description, of visualization models.

Usually, when performing visual exploratory analysis of very large, multivariate data sets, researchers either choose from few of the suitable visualization models, Parallel Coordinates Plot for instance, or, recur to some sort of preprocessing in order to reduce the volume or complexity of the visual information, adapting it to the more common visualization models. These preprocessing operations/transformations occur prior to the data being inputed into the visualization. In other cases, like with Liu (2013) and his innovative solution to compact data into data cubes prior to the visualization, preprocessing is a consequence of the solutions researchers came up with to make their data immediately available on request in the final visualization tool. While there is no dispute to the advantages of preprocessing, in many cases it carries along a high processing time cost and one obvious direct disadvantage, the inability to immediately input and explore new data with the visualization tool, without first running it through all the preprocessing steps.

Until recently, researchers had to have access to computer clusters, or large amounts of time, when to perform massive computation tasks. In the last decade Graphic Processing Units (GPUs) have followed a steep rise in all the critical features that enable for faster operations performed over increasingly larger data sets, as we can see in Figure 1. GPU available memory has also increased significantly, to the point where its quite usual to have personal computers with at least the same amount of GPU memory as of System RAM.

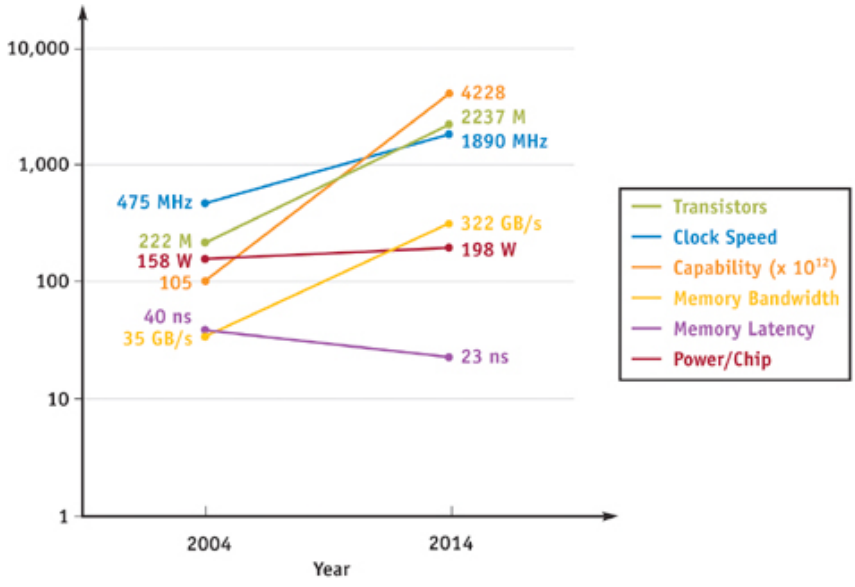

Figure 1 - Changes in Key GPU Properties over Time (source: GPU Gems 2, 2005)

The parallel capabilities of even the low end of currently available Integrated Graphic processing units, Intel Iris 5200 for instance, has at least 40 Compute Units, capable of running up to 7 simultaneous threads each (280 total) and up to a total of 8960 concurrent work-items. More than enough reasons for developers, on any computing demanding implementation, to consider what can or cannot be parallelized. Not long ago, numbers like these were only seen in very large computer clusters. Additionally each manufacturer is constantly changing and evolving the GPUs architectures to boots Single Instructing Multiple Data (SIMD) operations, designing and specialized floating point units (FPU) for different kinds of operations (simultaneous floating point multiplications, integer operations or even transcendental math functions).

These facts have not gone by unnoticed by researchers, in fact, the number of Research projects matching a search for the keyword "GPU", performed on Google Scholar, has seen an almost exponential rise since the year 2000, refer to Figure 2. Owens (2008) shows some good examples of researchers usage of GPUs and also which problems are more suited to be solved using GPUs over CPUs. Yet we feel that there are two main groups of GPU usage by researchers, usually tied to the researchers experience and field of work. The first group is heavily focused on the processing capabilities, making the best of those GPUs capabilities to perform massive computations over very large data sets, collect the results and visualize them using the more simple visualization models (line graphs, bar charts, etc). And the second group that usually have their data preprocessed with the more common CPU applications and scripts, but make use of the GPU to materialize high visual quality, performant and innovative visualization models.

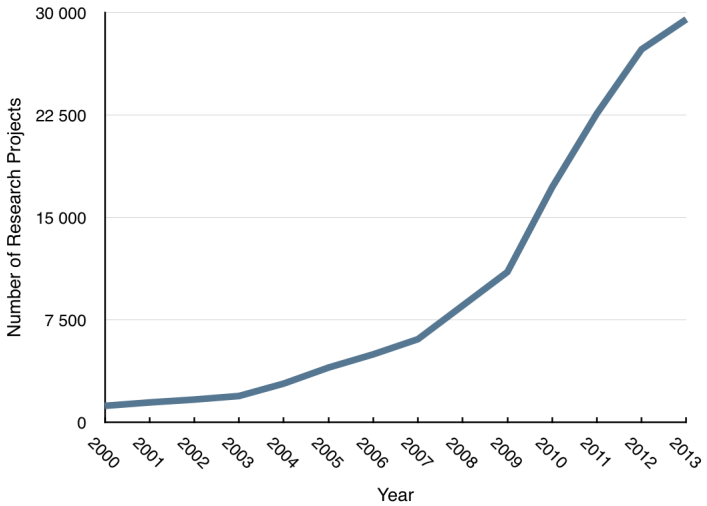

Figure 2 - Number of research projects matching the keyword GPU over the years.

The main focus of this Thesis is to go one step further down this trend, by developing an application that is both capable of using GPU computing to process the data, transforming it accordingly to the chosen visualization model, and at the same time render that visualization model at interactive frame rates and with high quality graphics.

To accomplish this we will explore different techniques and suggested best practices in order to make the most of the Single Program Multiple Data (SPMD) GPU programming model,

as shown in the multiple works presented in GPU Gems 2 (2005) and in the work of Owens (2008); plan and develop the application following a "GPU to GPU" structure which keeps all the relevant application data in video memory and minimizes data transfers to and from the CPU to the minimum possible, as the work of Gregg (2011) shows to be a crucial aspect for performance; and finally, we will explore, adapt existing and/or "design our own" rendering techniques, similar to those employed in video games industry, of which GPU Gems 2 (2005) and GPU Gems 3 (2007) shows multiple examples in a compilation of works from different authors of both of the academic and industry worls, in order to produce a high quality, interactive data visualization, approaching the quality and detail usually only achievable through the use of vector graphics APIs.

1.3. Objectives

The work presented in this document spanned roughly two semesters, during which the objectives we established at start, regarding the visualization model implementation, and the outlined plan for additional models, were forcefully altered and continuously adapted.

Initially we received a Sales data set, containing 2 Years of Sales records from over 700 super and hypermarkets, the data set was accompanied by a group of relational tables with lists of Products, hierarchy between products, Store names and groups, among others. The data was all in comma separated value (CSV) files. The data set was ≈278Gb in size and contained around 2.8 billion sales records.

Additionally, we expected to receive a related Stocks dataset within a time frame that would allow us to implement a visualization model that would include data from both Sales and Stocks.

As such, our plan for the first semester included:

- researching methods to make the dataset accessible both locally and remotely;
- create scripts that would support other researchers analysis on this data;
- and research adequate techniques, plan and implement a data visualization application, using the Sales dataset, that tests the feasibility and value of this Thesis objectives of processing and visualizing data in realtime using the parallel processing features of modern GPUs.

Considering this plan, the objectives we initially set for the first visualization application to implement were the following:

- Study the OpenGL programming model in order to understand how it can be used to perform general purpose programming.
- Search other researcher's similar work and research adequate techniques to perform the required computational and visualization tasks.

- Implement a prototype application using the researched techniques in order to demonstrate the feasibility of the idea and evaluate the applicability of the used techniques.

This would allow us to build the necessary background and implement a set of initial tools that would enable us to quickly import and analyze the Stocks dataset, upon receiving it during the second semester, research of devise an adequate visualization model that would encompass both datasets (Sales and Stocks), and implement that visualization model in an interactive application using the techniques researched and experimented with during the prototype implementation.

Unfortunately the Stocks dataset arrival was continuously postponed until near the end of the second semester, and although it contains an extremely large amount of information, it spans a different time frame from the one of the Sales dataset, removing the possibility of relating each's information over time.

As the Stocks dataset arrival was being postponed we were forced to revise our initial plan, we wanted to keep the implementation process uninterrupted, maintain an open door for the uncertain arrival (at that time) or a related Stocks dataset, and guarantee that either way a complete and rich, both visually and feature wise, information visualization application would be implemented and analyzed by the end of the second semester. We didn't want to put aside the possibility of exploring an additional dataset but we had to devise a plan flexible enough to guarantee that the research project and this Thesis objectives could be met.

Our revised plan would consist in:

- Reimplement our initial scene engine developed for our prototype in order to reflect what we learned from the prototype experiment.

- Move the data processing steps from the OpenGL graphical API to the more flexible, general purpose programming model, of the OpenCL API. Add additional tracking of Client data. Implement interoperability between OpenCL and OpenGL.

- Research, devise and implement rendering techniques that enhance the visualization both in terms of detail and appearance, improving the amount of perceived information relatively to the prototype version. In effect giving the application a more cared and polished aspect, as expected from an application intended to be distributed to users.

- Iteratively research, plan and implement additional features for the application that improve it either in terms of functionality or in terms of additional, more detailed, alternative or related information.

As we mentioned before the Stocks dataset arrived at a time that it wouldn't be feasible to implement a new visualization application and finish it until the end of the semester, and consequently, following the plan of iteratively adding more features, the objectives established for our visualization application also changed, and were added upon during the course of the second semester.

By the end of the second semester, the established objectives and features for our data visualization application were:

- Present a visualization of the Sales data records using a geographical heatmap visualization model relative to both the Stores locations and the Clients residence's Postal Codes locations;
- Use the OpenCL API to implement the data processing steps of the visualization model;
- Keep track and display geographic information, of at least one additional computed variable, besides the accumulated heat values typical of heatmap visualization models;
- Introduce additional Administrative and Demographic data into the visualization;
- Use rendering techniques to enhance the visual experience and blend the different types of information, without compromising the visualization model;
- Provide a user interface that enables the user to perform every action and make use of every feature available within the application;
- Implement reporting mechanisms such as screen capture and video recording;
- Implement functionalities that enable the user to filter sales records of specific Stores;
- Implement a offline data source feature. Allowing the visualization of Sales records data files, previously exported from the database and placed within the application file structure.

1.3.1. Requirements

Our defined requirements that the application must fulfill apply to the hardware of a MacBook Pro Late 2013, or hardware of similar performance. More specifically a 2.4GHz dual-core Intel Core i5 processor, 8GB of 1600MHz DDR3L onboard memory, Intel Iris 5100 integrated GPU and 256Gb SSD drive. In terms of GPU processing power, as GPU hierarchical chart from the famous hardware reviewer Tom's Hardware (2015) shows, this GPU is on the lower end of the spectrum.

For us this represents an opportunity to show how the advantages of using the GPU parallel processing capabilities apply to all modern GPUs[1] and not just the most performant.

Our requirements are the following:

- A loading time below one minute;
- A waiting time before a data visualization starts below 10 seconds;
- An average frame rate of at least 10 frames per second while visualizing data;
- Compatible with at least two operating systems from different developers;

[1] OpenGL 4.1+ compatible

- Compatible with recent GPUs from all three major vendors, ATI, Nvidia and Intel;

1.4. APIs and Third Party Software

Planning the application involves not only the envisions of the applications' features and behavior but also considers the choices of technologies available with which to implement that application, and also the choice of third party software that may shorten the implementation time or support the application in any way.

For our implementation a database was the most efficient solution to feed data remotely to the application. Not just because of the gain in performance when using a database to query data but also due to the large size of the dataset which makes any option where the data is packed and distributed with the application, impractical.

1.4.1. Database

MySQL is a very popular database that has proven it self over the years, it is also one of the best suited to handle large quantities of data and shows great performance in a read intensive, single node, environments, when compared to its competitors, according to the works of Tudorica (2011) and Rabl (2012) which test and compare several of the most used databases. The referenced authors also confirm the suggestions presented in the MySQL documentation regarding database engine choice, pointing to InnoDB as the one that presents better performance when handling vary large tables.

For our database implementation we opted for MySQL due to its almost incomparable maturity in terms of development, 20 years, and for its performance results presented by the Works of Tudorica (2011) and Rabl (2012).

1.4.2. GPU API

Although our application is implemented from scratch, it relies on drivers to manipulate states and exchange data with the GPU. The available APIs, that provide these functionalities, vary with the chosen Operating System, the available hardware and even the Software Development Kit (SDK) used to implement the application (Java SDK, Windows SDK, etc.).

Choosing which API to use was quite straight forward, and we will take you through the same thought process we did. First try to keep in mind the different Graphics/Computing APIs available: DirectX, Cocoa, Metal, OpenGL/OpenCL, Cuda and Mantle. Then, if we start crossing out APIs that have characteristics we don't want, for instance Operating System specific (DirectX, Cocoa and Metal) and Hardware Manufacturer specific (Cuda and Mantle), we are left out with only one option, OpenGL/OpenCL.

So when you think about it, there really wasn't much of a choice. And its quite surprising in a way, that after so many years of GPU and related software development, developers who seek to use a cross platform and cross hardware solution, have to rely on the existence of

the Khronos Group and it's continuing work, revising the standards and improving the OpenGL/OpenCL APIs, keeping them on par with the released hardware features. Fortunately there is no end in sight for the Group, and developing of a successor API for OpenGL, Vulkan, is already underway.

1.5. Document Outline

The remainder of this document is structured as follows:

- Review of similar work from other researchers upon which we hope to improve;
- Description of the data sets we used in our visualizations, both the ones provided by Sonae SA and others gathered by us;
- A brief introduction to the OpenGL and the OpenCL APIs, highlighting the most relevant aspect regarding our work;
- Detail the most relevant implementation steps towards realizing our visualization application;
- Present and review the results of our implemented solutions;
- Discuss and draw conclusions on the achieved results;
- Present ideas for future work;

2. State of the Art

In this chapter we will present, analyze and compare a set of selected works from other researchers that were the most similar we could find to our own work, and from which we wish to improve upon.

2.1. Scheepens "GPU - Based track visualization of multivariate moving object data"

Scheepens (2010) thesis is a great example of a similar work that involves both data processing and transformation to suit a chosen visualization model and implementation of that model while doing all those computations in the same application and keeping all the relevant data in the Video Memory.

Scheepens improved upon Willems (2009) work on "Visualization of Vessel Movements". The data set they used contains information of vessel movement around the Dutch coast, vessels have a set of characteristics and for each there is a list of points containing its movement information, namely Position, Velocity and Time.

The visualization of the vessels tracks over time is done by rasterizing them into density fields as lines with a chosen width (kernel radius), the coloring of each line is done through a devised formula that takes into consideration a gaussian distribution, the previous mentioned kernel radius but also different properties (eg. Time, Velocity), specific to each of the track segments, in such a way that increases the perception on how that same properties varies over the length of the line. For instance a vessel track can appear thinner at its departure location and increase over the length of the trip giving an immediate sense of direction to a human observer. The rasterized tracks are then blended together filling the density field. Refer to Figure 3 for examples of vessel movements rendered by Scheepens application.

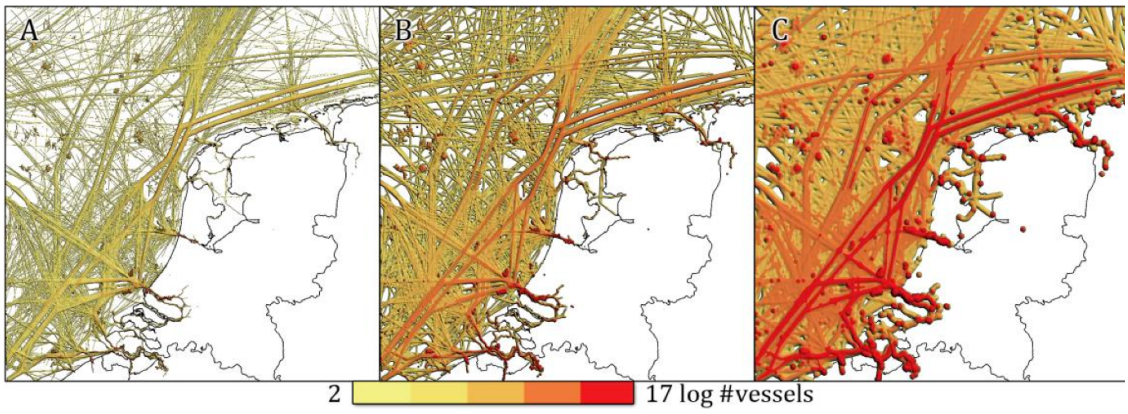

Figure 3 - Example of Scheepens application renderings. Quoting his legend for this figure, "A selection of density maps of vessel movements around the Dutch coast with a cell size of 250 meter and a kernel size of 250 meter (A), 1 kilometer (B) and 3 kilometer (C)"

One of Scheepens main goals was to improve upon Willems work in terms of performance, to do this he implemented the entire density field calculation to the GPU using a Graphics API, namely OpenGL, shaping its mechanisms and features to perform general purpose computations and not a GPGPU based implementation as he names it during the document. Scheepens clever implementation consisted in drawing the track segments individually, setting the vertices up in a Vertex Buffer Object (VBO) mapping data to vertex properties. Those draw calls were issued into a Shader Program consisting of Vertex Shader (VS), Geometry Shader (GS) and Fragment Shader (FS). The VS was used to copy data into the correct vertex properties according to the actual render configurations. The updated vertices were then fed into the GS which played a main role in first filtering vertices to avoid unnecessary calculations, and then generating the geometry of each line segment as a bounding box (rectangle), which was in turn fed forward into the hardware's triangle setup and then rasterizer stages to produce and feed the fragments for the line segment into the FS. The FS was then responsible to calculate the density contribution for each fragment according to a kernel (shader), coloring the segments as described in the last paragraph. The draw output is saved in Frame Buffer Object (FBO) for later usage. Refer to Figure 4 for a visual representation of the density field rasterization process implemented by Scheepens.

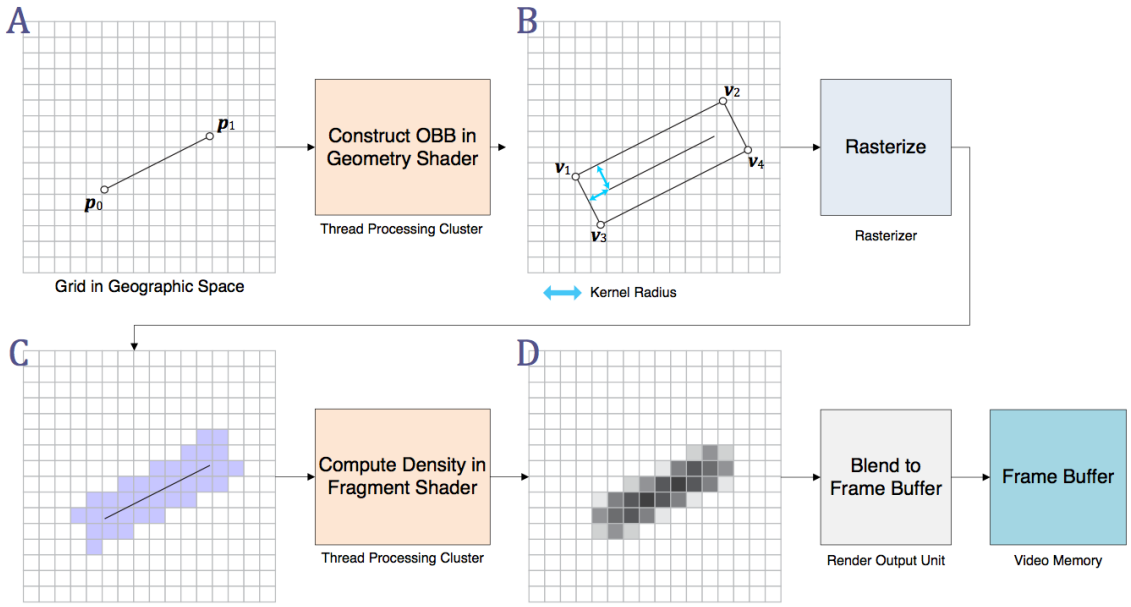

Figure 4 - Scheepens process for drawing a segment of a vessel tracks onto the density field FBO (source: Scheepens 2010 Thesis).

The drawing and calculation of the final density field used in the visualization is quite versatile in terms of configurable options. Multiple filtered density fields with different sets of configurations can be blended into the final one with varying operators, which allows for different levels of detail, highlight of certain features and exploration of different kind of events. The final density map is made of 32 bit floating point values of the density calculated previously. Each pixel relates to a "cell" which represents a patch of real world space of varying configurable sizes.

The final rendering to the Screen Buffer may be derived of one or multiple density fields. To deal with multiple fields Scheepens introduced visualization operators in order to highlight

data from each of those different fields. For each field, and prior to the operators, color is mapped from the density values through a transfer function. Color maps can be exchanged and Scheepens experiments with continuous and discrete ones. As each pixel in the screen hardly ever matches one pixel in the density fields FBOs, due to projection and camera movement, Scheepens relied on cubic interpolation over the hardware bilinear interpolation to improve visual quality and smoothness, avoiding the discontinuities and jagged edges produced with bilinear interpolation.

Scheepens did an impressive job not just in increasing performance over the previous work by a factor of 3000x, but also in being able to validate his visualization tool by producing almost indistinguishable renderings from the ones previously obtained by Willems.

2.2. Buschmann "Hardware-accelerated attribute mapping for interactive visualization of complex 3D trajectories"

Buschmann's work is another great example of the advantages of moving computations from the CPU to the GPU. His work is very similar to Scheepens in which both of them are creating visualization tools to allow a human user to explore trajectories of moving objects over real world space. Also their approaches in terms of implementation contain many similarities, even if their final visualization models are quite different. One of the reasons for this is that Scheepens movement trajectories are projected into a 2D plane and Buschmann's are in 3D space.

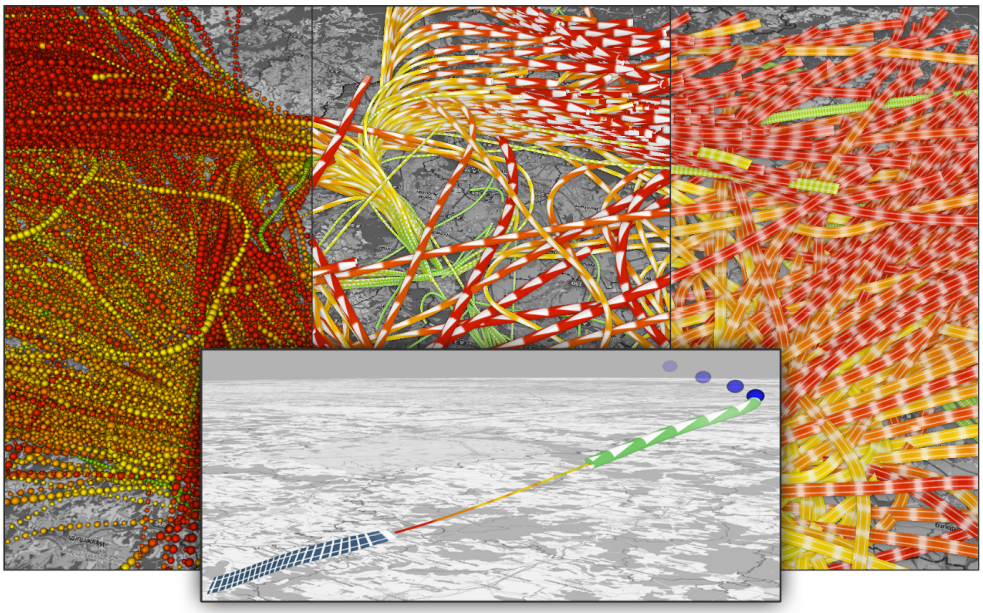

Figure 5 - Renderings with different style and mapping configurations, made with Buschmann visualization tool. (source: Buschmann Jan 2014).

Buschmann's work focus on the visualization of aircraft movement trajectories. Their data set, collected near the vicinity of an airport, contains the trajectories for each aircraft in the form of poly-lines where each poly-line is a sub-set of consecutive points, each containing

several varying and static (relative to each trajectory) attributes including position of the aircraft, velocity, flight ID, etc.

His approach in terms of visualization model consisted in rendering the trajectories in a 3D space either as tubes or as a sequence of spheres, from which the user may choose from. Additionally, Buschmann used both coloring and texturing of the generated geometry as means to represent additional attributes, such as speed varying along the trajectory. The user is able to change which attribute is mapped the color or texture on a per trajectory basis. Refer to Figure 5 for examples of the different styles and attribute mappings. Attributes can be mapped not just to the coloring and a choice of texture, but also to properties both of the texture, like texture stretching and torsion, but also to properties that affect the generated geometry, like for instance radius of tubes and spheres.

To accomplish this, Buschmann approach starts off by moving the data as raw (after preprocessing) to the video memory and then process it and render it using solely a Graphics API. First a Vertex Shader (VS) performs the mapping of attributes to vertex properties, according to the list of configurations structures uploaded to the GPU as a Uniform array. Vertices attributes and rendering configurations are then fed to a Geometry Shader (GS) that is responsible to generate the appropriate geometry, respecting visual configurations like the radius of generated spheres and tubes and correct assignment of texture coordinates. The generated geometry is then rasterized and the generated fragments later processed by a Fragment Shader which is responsible for coloring the geometry, again according to the visual configurations set for that fragment, such as texture fetch, color mapping, etc.

Buschmann's work show interesting usage of vertex attributes to map data, and the solutions he implemented to overcome performance issues give a great insight onto the similar limitations we might face.

2.4. Resume and comparison

Both the works we presented share great similarity with our objectives. Even if our work does not involve moving objects we do have to represent a time series with a geographic component, as both Scheepens and Buschmann did.

We can see by each's description of their implementations that both solutions, in a minimalist view, consisted in moving the computations being performed on the CPU to the GPU. They both had to study the OpenGL programing model and architecture in order to choose the best structures to transport their data onto and through the graphical pipeline, also they had to analyze their algorithms and equations into devising a way to parallelize the necessary computations.

We seek to follow a similar implementation process, studying the best solutions and structures to handle our data, how to implement them and how to process them.

Regarding the results presented by each of them, we see, very common to this research works, the performance comparison between past and current solutions, with tremendous

gains undoubtedly. And this is crucial for our work as well, but not just, we wish to devise a good implementation performance wise but not just to have a "fast" application. We wish to realize an animated visual representation of the data, and whatever performance leftover we might have we will use to enhance the quality and interactivity of the visualization.

Scheepens solutions of discretized color maps and ways explored to blend more than one layer of information, as Buschmann geometry shader usage to generate the geometry of the flight paths, are techniques we intend to explore in our application.

We intend to explore Scheepens and Buschmann ideas during our implementations but not just, we will delve into the world of rendering techniques from numerous authors, of which the referenced compilation GPU Gems 2 (2005) and GPU Gems 3 (2007) provide a great resource, into picking the most adequate to the problems we face.

Regarding Scheepens and Buschmann explored techniques and technologies, we intend to go a bit further, exploring the OpenCL API as a complementary tool for information visualization applications.

3. Data Sources

In this Chapter we first describe the Sales Dataset in more detail, and later we describe complementary data gathered from multiple sources, that we used either to complement the data present in the Sales Dataset or that we used to improve the final visualization model, both in visual quality and in displayed information.

3.1. Sales Dataset

The Sales Dataset, which is the main focus of this Thesis and the dataset we intend to visualize with our visualization model, is composed of 24 moths of sales records from Sonae's chain of retail stores. It has a total of around 2.8 x 10^9 sales records (lines). Each record contains the Date of the purchase, to the second, the Store Id, the Product Id, the Customer Postal Code (7 digits), the Customer Client Card Id, the value of the sale, the purchased Product quantity and the Discount applied to the sale.

Additional information regarding the relationship between Brands and Stores, and Product hierarchy, was provided along side the sales records. This database tables also provided additional description fields for Stores and Products.

Our first analysis of the data revealed that there was a total of 729 named Stores (through the Store description table), but only 465 were referenced in the Sales records, additionally we found 6,579,711 of unique Client Card Ids in the Sales records.

The data was provided to us in the form of several zipped Comma Separated Values files, that when unpacked occupied roughly 278Gb of disk space. More detail about the data format and how we imported it into a database can be found in the Implementation Chapter.

3.2 Complementary data gathered

After our first analysis of the sales data reveled that we would have to gather additional complementary information in order to implement a visualization model that represents the sales data, which is in its essence a time series, and at the same time allow for more abstract and at the same time extremely important in the world of business, questions like "What are the areas covered by the chain of retail stores?", or "Are there meaningful clusters of customers traveling far to make their purchases?", or even "How are sales distributed in relation to the population density?".

In this chapter we describe the additional data we collected, their sources and its relationship to the sales data. First we describe how we gathered GPS coordinates for the Stores locations and for the Customers Postal Codes, then we describe how we used Open Source Open Street Maps to filter all the necessary shapes to allow for geographic visual representations, and finally we briefly describe some additional demographic geo-referenced data we collected.

3.2.1. Stores Location Data

Retrieving Stores geographic coordinates involved an exhaustive search through the multiple Brands Web Sites, scanning their source files and JSON data to collect any data table containing Stores identification, Addresses and GPS coordinates. This set of data downloaded from the Brands Web Sites will be referred to as BWS from this point on, to avoid confusion with our Sales Dataset and facilitate writing.

The search produced multiple data tables of different structures, with similar information. Unfortunately there was no relation between Store Ids on those tables and those of our Data Set and also, Store Brand names weren't a perfect match to those in our dataset. For instance a Brand like "Modelo" is referred to as "MDL" in our Data Set, but in this data, it could be referred to as "Modelo", or as "Continente Modelo" or even "Modelo Cnt". This created a problem for us because Stores to which we couldn't find GPS coordinates would have to be left out of the visualization, that would consequently be an incomplete picture of the actual covered area.

To solve this problem a small Client > Server application was created, using Web Technologies, that allows a user (or group of users) to quickly match our known Stores from the Sales Dataset with the ones in the BWS. Web Technologies, namely HTML, Javascript, JSON (data transfers) and PHP (server side), were chosen, to build all the small support applications that required a User Interface and User interaction, due to the following reasons. On the server side, modern HTTP servers are very easy to setup, particularly for development processes, and also very easy to setup with our already present Database server,. The simple structure of web services allow us to manipulate and query our database with short implementation times. On the client side the versatility, interface oriented and HTTP Request API tools allow us to build simple dynamic interfaces in minutes.

Implementation began by creating a pair of tables in the database to keep track of matched and unmatched Stores, refer to Figure 6 for a visual description of this application structure. Then, an automated script was implemented to initialize the Unmatched Stores table with all the Stores present in our Sales Dataset, while at the same time performing a first matching attempt with the Stores in the BWS tables. This first matching sweep accepted only exact string match between Store names, for certainty, but matched only a few percent of the batch.

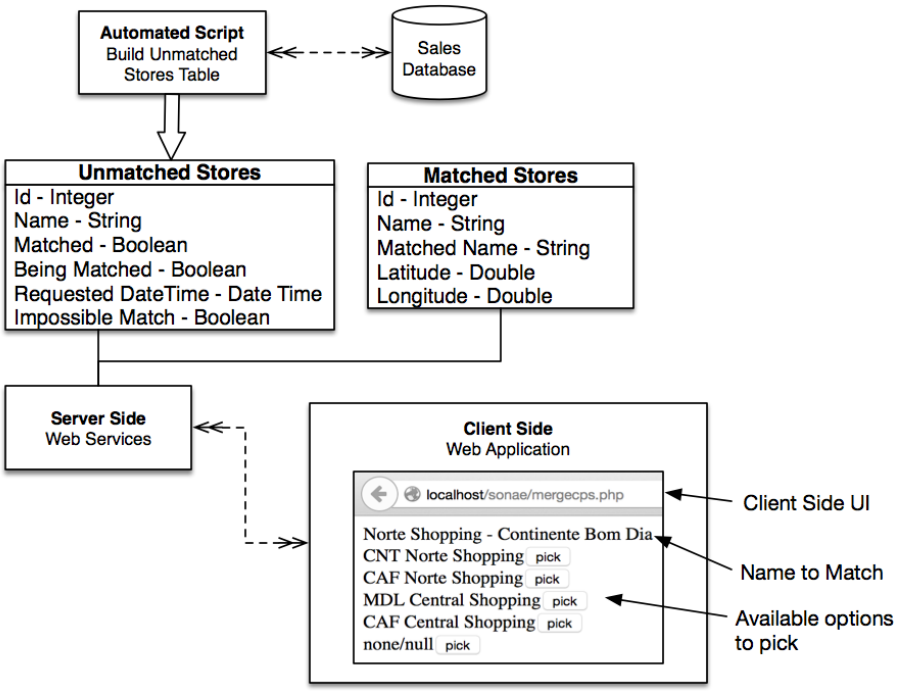

Figure 6 - Simplified structure of the application and scripts implemented to match Stores present in our Sales Dataset with the Store data gathered by analysis of the source and JSON data files of Sonae Group's Brands individual Web Sites.

When implementing the Client User Interface and the corresponding Server side web services, a couple of requirements were taken into consideration. First and most important the implementation time had to be as short as possible and second, the application interface would have to somehow help the user make that necessary correct decision in the smallest amount of time possible. To achieve this, refer to Figure 6 again as it contains a screenshot of the actual user interface, when the user opens the web application (through any web browser) the server picks one of the still unmatched stores, calculates the Levenshtein distance, as in Levenshtein (1966), between the name of the picked store and all the store names in the BWS, picks the five most similar, and then sends them to the Client, ordered by descending similarity. On the client side the Store name to be matched is displayed at the top and then the ordered similar Store names are displayed one by one with a simple "pick" button for each of them. As the user picks the correct name, usually the first, the client send the information regarding the picked name to the Server where, a web service implemented to that effect, saves the match between Store names in the Matched Store database and updates the Unmatched Stores table with the new matching information so that this store is not shown to the user again. At this point the cycle starts over, the Client immediately receives the next store to be matched and the corresponding matching list.

The process described above was successful in matching most of the Store names but unfortunately not all. Some Store names were simply too unique and unrelated with the ones from the BWS tables. To handle these ones, an additional step was introduced using the Google Maps Widget. The application picks GPS coordinates from the set of Stores in the Brands Downloaded Tables, that were until that point unmatched and, by user request moves the Map Widget location between those GPS coordinates, refer to Figure 7 for a screenshot of a matching example using this extra step. The user would then cycle through

the still unmatched Stores names (on the bottom of the interface) and the different Map Widget locations to determine and match the correct Stores and Locations.

This sequence of processes and steps allowed us to match 100% of the Stores referenced in the 2 years of our Sales Dataset with the geolocation data on the BWS tables.

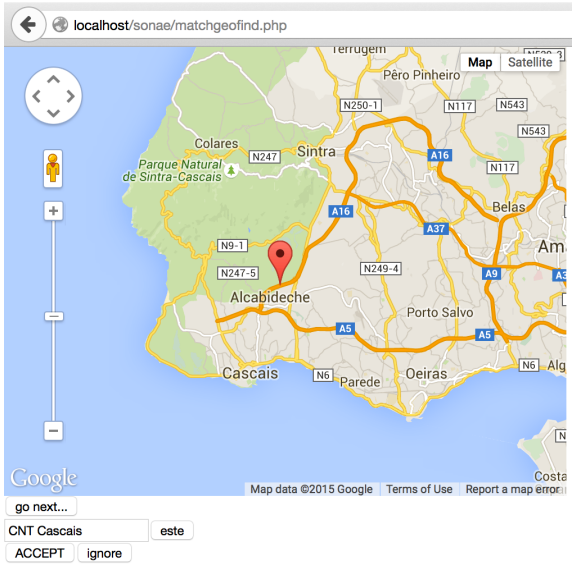

Figure 7 - Screenshot of the additional geo-referencing step interface, created to help match GPS coordinates with Stores.

3.2.2. Clients Postal Codes Location Data

As mentioned in Chapter 3.1, sales records are accompanied by the Postal Code of the Address registered with the Client Card Account of each Customer. This provides accurate information relative to the residence location of who made which purchase and where. To use this information in our visualization, like with the Stores, we needed to convert the Postal Codes to GPS coordinates. The Sales Dataset contains around 180,000 unique Postal Codes, which made it imperative that the task of identifying GPS coordinates for each of them would be automated or in someway automatic.

It is important to mention at this point that the privacy of each client was respected, for a better understanding of how refer to Chapter 5.3, which describes usage for this data.

Portuguese Postal Codes contain 7 numbers, divided into two groups separated by a hyphen, and are extremely accurate, up to the door number. But despite that fact there is no free and efficient web service or API available online to convert large numbers of Postal Codes to GPS coordinates, with that same accuracy.

During our search we found some Web Sites, including the National Postal Service for Portugal, that provided the conversion service through Web Forms, which would be difficult to automate with a script and at the same time would violate the sites EULAs.

The Google Maps Geocoding service, the API we used in our implementation, provides geocoding to GPS coordinates using the 7 digit Postal Code, but fails to identify around 5%

of the Postal Codes. To complement the Postal Code's information we used the full Postal Code Table for Portugal, retrieved from Portugal's National Postal Service Company, to complement the information with City Name and Province Name, increasing Google's identification success rate.

Other advantages favoring the Google API are its mature state of development, is free of charge within a set of limitations and our previous experience with it made the development more swift. As disadvantages, that affected how we modeled the application, the Geocoding service has a daily limit of 2,500 requests per user and also a instantaneous limit of 10 requests per second. If we chose to automate the task using a single machine, running 24/7, it would take around 60 days to complete the task, which was not compatible with our requirements.

To solve this problem, instead of centralizing the task, we created a Web Service to which different users can connect at the same time, using any Web Browser compatible with Javascript. By leaving the Web Service's page open on their browser, users will be allowing a script to execute on their computer that will process the geocoding requests to the Google Maps API from the users computer, until it reaches the 2,500 daily limit for this user, which takes around 5 minutes.

The structure of the implemented application was very similar to the one implemented to match Store names, described in the last chapter. Two control tables were created, one with all the Postal Codes that need corresponding GPS coordinates, and the other to save the complete information for later use, refer to Figure 8 for a more detailed description.

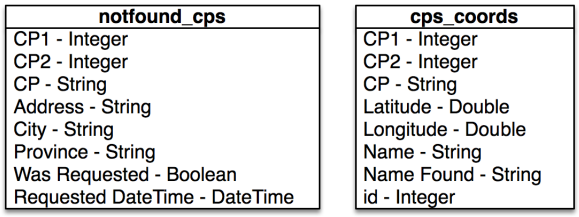

Figure 8 - Control tables created to keep track of the Geocoding process.

The progress of the full task is still managed on the Server side, but the server relies on the connecting clients to perform the geocoding requests. As each Client connects, it sends a request to the Server for the next Postal Code that needs identifying, the Server queries the notfound_cps table for Postal Codes (CP1 and CP2) that don't exist in the cps_coords table, and also that no other Client is currently processing. The Server updates the selected Postal Code's "Requested DateTime" to keep track of timeouts and returns the Postal Code information to the Client. The Client receives the Postal Code, City Name, and Province Name from the Server, assembles the geocoding request using the three fields merged into a String as Query and sends it to the Google Maps API. Upon receiving a response, the Client forwards the relevant information to the Server, where it will be saved in the cps_coords table in case of success.

Upon completing the each geocoding request, the Client process analyses a set of local variables to control the time between the processing of each Postal Code, making sure it

sleeps whenever necessary to respect Google Maps daily and immediate request limits mentioned above.

Using this implementation, five voluntary users were able to help complete the task in approximately 15 days.

3.2.3. Shapes of OpenStreetMaps

Until now we have described how we added additional geographic information to the sales data, but still, using that information in a visualization model without contextualization has little to no meaning.

To provide that contextualization we collected administrative areas boundary data of different levels, Country, Municipality (Concelho in Portuguese) and Civil Parish (Freguesia in Portuguese). We used Open Street Maps (OSM) dowloaded data[2] as the source of that information. Open Street Maps is a non-profit foundation compiling geo-spatial data from multiple sources and providing free access to it, aiming to support research and development.

Several applications exist to access, visualize, filter and export OSM data, we used two in our project. One was GIS Explorer (by BMT Cordah) which provides a GUI and is ideal to visualize the data while at the same time filter by any of the multiple different attributes, it also provides multiple forms of exporting selected data. The second was a command line tool named OSMOSIS, implemented in Java and with an accessible API, which made it ideal access the OSM data from our own Java applications. Both these applications were chosen not only because of their features but also due to being available free of charge.

We used both these applications to export the boundary data into several "poly" files, one per administrative zone, containing the boundary shape (or shapes) as a set of polygons in a GPS coordinate reference.

We implemented Poly file parsing in Java, a GNU licensed file format[3], to enable our applications to use the boundary exported data.

3.2.4. Demographic Data

One of the final features we experimented with and implemented in our visualization model was the ability to overlay additional demographics information with the sales coverage areas (or Heat Spots; or even Heat Zones). Crossing these two types of information might provide invaluable information to the business decision making process, as the work of Pol (1997) , by providing additional information about the social and economic characteristics of both the covered and not covered areas.

---

[2] http://download.geofabrik.de/europe/portugal.html

[3] https://www.cs.cmu.edu/~quake/triangle.poly.html or http://people.sc.fsu.edu/~jburkardt/data/poly/poly.html

We experimented with different styles of visualizing the demographics information, in an attempt to determine which style provided the best aesthetics and at the same time accurate and readable information, when blended with the sales coverage areas. The different styles and their different implementations are described in detail in Chapter 5.7.

In terms of the demographics data, for one of the experiments we build a tailored dataset of points in geo-space, that aggregated building locations extracted from the Open Street Map's data and the GPS coordinates for the Clients Postal Codes, in an attempt that the visual representation of those points would be a relatively accurate representation of the Countries population.

For all the other demographics representations we attempted a more general and extendable approach. Instead of building tailored datasets with specific rendering styles in mind, we chose a credible statistics institute, the National Statistics Institute for Portugal (INE), as a source of accurate data, particularly because their Web Site allows users to browse and export datasets related to several demographics variables, refer to Figure 9 for a screenshot of the Web Site data selection user interface. We then implemented adequate scripts to parse the default format of files exported through the web site, for datasets containing distributions of a single variable (per dataset) over the Municipality or Civil Parish levels of administrative zones. This would enable us to add additional demographic variables easily in future. For the experiments described in this document we used two different demographic datasets from INE, one with the Total Population per Civil Parish and the second the Purchase Power per Municipality.

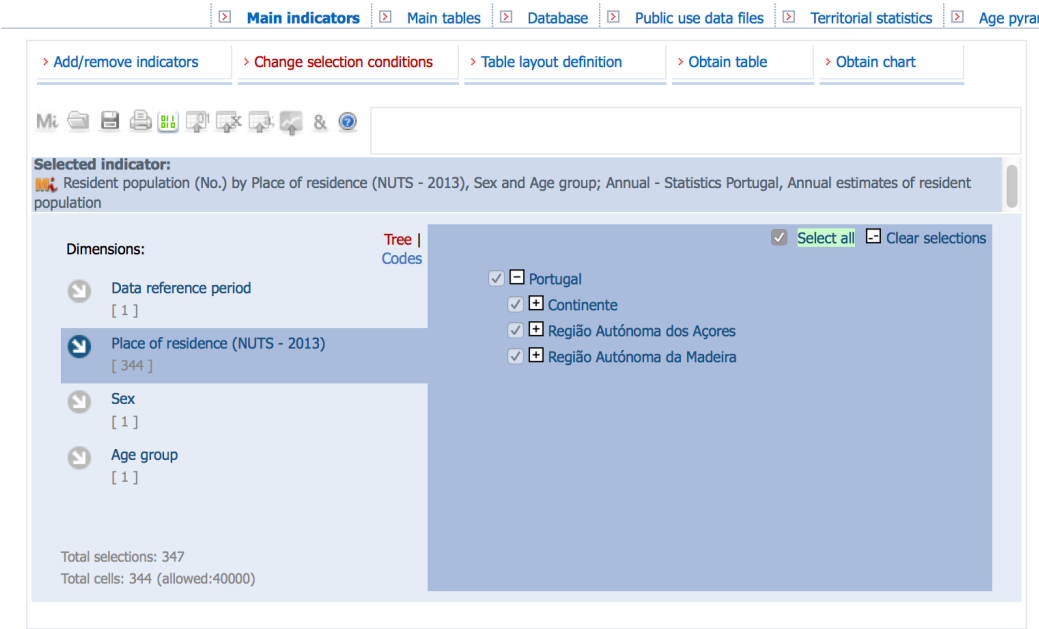

Figure 9 - INE data selection and export interface.

The Total Population variable is self explanatory and for the Purchase Power a detailed description of its formula can be found in an extensive analysis on the subject, INE (2000). We didn't dwell much into to it as the variable for us represents only a relative reference of the differences in purchasing power across the country.

4. Programming models overview

Our implementation focus heavily on small programs that make the best of the parallelism exposed by both the OpenGL and OpenCL APIs. We wont go into every small implementation detail on the Implementation Chapter, for we would risk confusing the readers on what is relevant or not in our implementations when compared to others. So in order to make any implicit introduced development step, during the description of the implementation of features, more explicit, we first would like to make a small introduction to the modern[4] OpenGL and OpenCL programming models. Detailing their configuration steps, creation of data structures, binding of data to GPU registers and finally execution.

We hope that this information, combined with the features implementation description, gives the reader a strong understanding of the work, left out of this document, behind each feature.

4.1. OpenGL

First, and most importantly, OpenGL is a rendering API, and as such it's rendering pipeline was designed and optimized to create a 2D raster representation from geometry data describing a 3D scene. In other words, any sort of GPGPU programming using the OpenGL API requires deep understanding and knowledge about each of the rendering pipeline stages and the available OpenGL data structures in order to devise an implementation that will result in the actual succession of mathematical operations and other transformations that are required.

Over time, the OpenGL API, as others APIs as well[5], evolved into reducing the process, or sequence of commands, one has to issue on the CPU in order to traverse the rendering pipeline with some input data. Nowadays the process is as follows:

- Create shader programs;
- Create and configure buffers;
- Load data into buffers;
- Connect (Bind) data locations with shader variables;
- Render (Draw Call);

When the draw call is issued, depending on the configuration of flags on the GPU registers and the structure of the Shader Program, the data contained in the loaded and binded buffers will traverse a particular path on the GPU pipeline. The result of a draw call has to

[4] Modern OpenGL usually refers to the post OpenGL 3.1 version, where the "old" fixed pipeline functions were removed and developers are forced to use only shaders. Other authors also like to consider the Modern OpenGL as the post 3.2 version, where geometry shaders were introduced.

[5] See the motivation behind the implementation of Mantle, Metal and Vulkan.

be saved somewhere, usually the Screen Buffer, but this also is configurable by the user, enabling more control over the data flow, as we will show later.

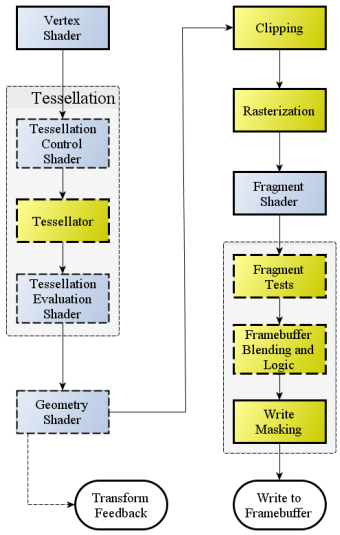

Figure 10 - Diagram of the most important stages in the OpenGL 4.0+ pipeline.

Shader is a computer program which purpose is to produce shading, the production of appropriate levels of color within an image. In computer graphics this process involves several steps, and consequently several types of shaders. And its important to understand the different types, as for each, data is inputed in a particular way (or ways), different native functions are available, the number of times they execute are dependent on different factors and finally each outputs different data. Synthesizing the characteristics of the shader program types we use, we have:

- Vertex Shader (VS): it executes once for each vertex, independently of the primitive type (points, lines, triangles, among several others). It has access to the vertex data structure, which is configurable, and outputs a similar (or not) data structure per vertex (output structure also configurable). In typical rendering scenarios this is where projection of vertices is computed. The vertex shader has limitations regarding data structures and built in functions. Can output directly through Transform Feedback (TF) into a buffer, refer to Figure 10.

- Geometry Shader (GS): it executes once per primitive. This is arguably the most flexible shader type as it enables access to textures and gives the developer the ability of specifying the primitive type of primitive that each execution will have access too. Primitives can be points, lines, lines_adjacency, triangles or triangles_adjacency. Usually is used for generating geometry from the input data. It outputs (also configurable) points, line_strip or triangle_strip. The GS, like the VS, can output data directly through TF.

- Fragment Shader (FS): it executes once per generated fragment. It receives some built in computed variables, related to the fragment, such as the coordinates in windows space, if its front facing and it position relative to the pixels center. Additionally it receives interpolated values, and their derivatives (through built in functions), of the vertex geometry information that generated the fragment. The FS outputs a Color vec4(r,g,b,a),

and, if not overwritten by the shader execution, it also automatically computes and outputs the fragment depth and a sampling mask for multisampling.

A Shader Program (SP) consists in one or more shaders which together form a valid structure in conjunction with the pipeline, defining a path in the pipeline from input to output. In other words, an SP cannot consist in two FSs, as there is no path in the pipeline where data can be inputted directly and, more importantly, no path from one Fragment Shader to another. So Shader Programs can consist of VS>TF, VS>GS>TF, VS>GS>FS, among others, as long as they form a valid configuration.

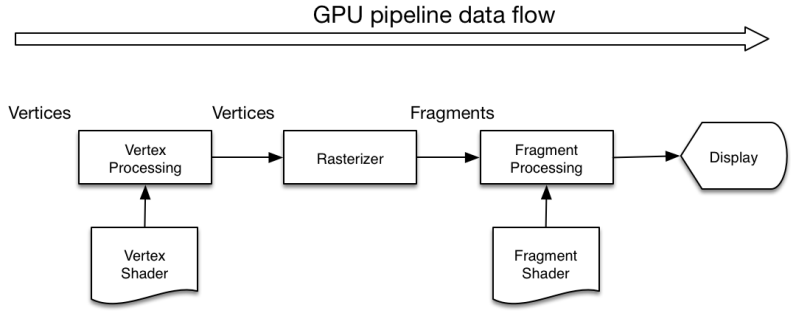

Figure 11 - Simplified rendering pipeline model, with vertex and fragment shader stages.

Figure 11 shows how the most basic pipeline configuration processes vertices data into a 2D raster image. And below in Figure 12 we can see a slightly different pipeline "path", this time including the Geometry shader stage. (shaders)

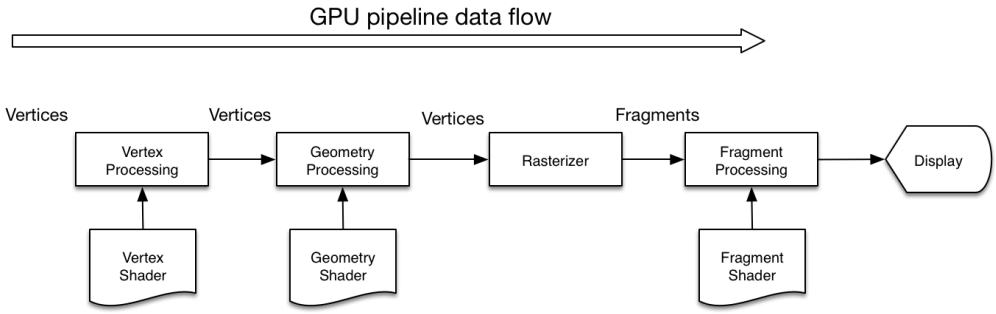

Figure 12 - Simplified rendering pipeline model, with vertex, geometry and fragment shader stages.

Besides the binded buffers of vertices data that inputed to the Vertex Shader, and the Texture objects we mentioned so far, OpenGL provides a special variable qualifier "uniform", named this way because uniform values don't change from one shader execution to the next within the same render call. Uniforms act like shader parameters that are easily manipulated in CPU code between render calls. Uniforms, for instance, help specify parameters defining different shader behaviors without the necessity of a different shader for each behaviors. Textures for instance are Uniforms.

4.2. OpenCL

OpenCL, like OpenGl, OpenAl and others, is an open standard maintained by the non-profit technology consortium Khronos Group. OpenCL is a GPGPU framework, and allows for writing programs that execute across heterogeneous platforms consisting of different types of processors (CPU, GPUs, among others). In this document we will focus only on GPUs.

OpenCL provides a top-level abstraction for low-level hardware routines, that allows developers to make the best of modern GPU hardware architectures, to run massively-parallelized programs, usually name as OpenCL Kernels. A Kernel is a program that is tailored to execute multiple times in parallel to complete a job[6].

4.2.1. Architecture

OpenCL architecture is structured in what we might call levels, and we will give a brief description of each, as to understanding each and its relation to the others, helps understand how to setup execution of Kernels (more on this later), and how that execution is performed in the GPU. The levels are as follows:

- Compute Device (CD): a compute device is for all effects and purposes an individual processing unit, for instance the CPU is a CD and also, the GPU is a CD. A Kernel job can be distributed over several devices.
- Compute Unit (CU) (also called Execution Units (EU)): A CD has one or more CUs and contains one or more (usually more) processing elements. Processing elements in each CU share part of the hardware's memory and computational units.
- Processing Elements (PE): The processing elements are the lowest differentiable processing level, effectively where each kernel execution happens.

Note the vague description for Processing Elements, for these the documentation found on the internet can get a bit misleading sometimes and confusing. Particularly because the specific definition of what a Compute Unit is, varies from vendor to vendor. On Intel for instance, CUs, or EUs are Simultaneous Multi-Threading (SMT) compute processors that drive multiple issue Single Instruction Multiple Data Arithmetic Logic Units pipelined for high throughput floating point and integer compute. In numbers, for the 7.5 generation of processors, each EU has 7 SMT processors, and a pair of SIMD FPUs[7]. Each FPU is capable of SIMD execute up to four 32-bit floating point (or integer) operations, or SIMD execute up to eight 16-bit integer operations. The SIMD-Width, calculated from the number of operations that one FPU will be able to execute simultaneously, for instance 4 32-bit operations equals a SIMD-4, is defined by the compiler and relative to the kernel being

[6] A *job* in OpenCL, refers to the total work to be performed by a kernel when enqueued for execution. The job size is in effect the number of independent work-items that will have to execute in total, to complete the designated task.

[7] Although called FPUs they perform float and integer operations.

executed, will ultimately set the number of kernel instances that can run concurrently. For instance for a SIMD-16 compile of a kernel, it is possible for SIMD-16 x 7 threads = 112 kernel instances to be executing concurrently on a single CU (or EU).

Fortunately for developers, OpenCL API provides abstraction from this vendor specific differences in hardware architectures. OpenCL works with the concept of job dimensions, a one dimensional job of 1000 items means that, to execute the job, there will be 1000 kernel executions. One can relate job dimensions to dimensions in an array, picture a two dimensional array (100x100) for which we need a kernel that at each execution substitutes one value of the array for it power of two. We could issue this job with OpenCL as a 1-D job of size 100x100=10000 or we could issue a 2-D job of size 100x100, the later being the more advisable. The job dimensions will then be used to distribute the individual executions as Work-Items, refer to Figure 13, into Work-Groups. The grouping of individual work-items is constrained by the hardware characteristics but also by each individual kernel's requirements, both in terms of processing and memory. Like we've seen before, a kernel that only performs 16-bit integer operations might be able to execute more times concurrently on certain hardware.

OpenCL provides methods to query each device for their characteristics, for instance the maximum number of compute units or the maximum number of work-items per work-group, and also to perform queries relative to each's kernel compiled code, for instance the kernel group work size and the kernel preferred work-group size multiple. Developers don't really have to work with this information to simply execute a kernel, but they are crucial when it comes to optimizing that execution.

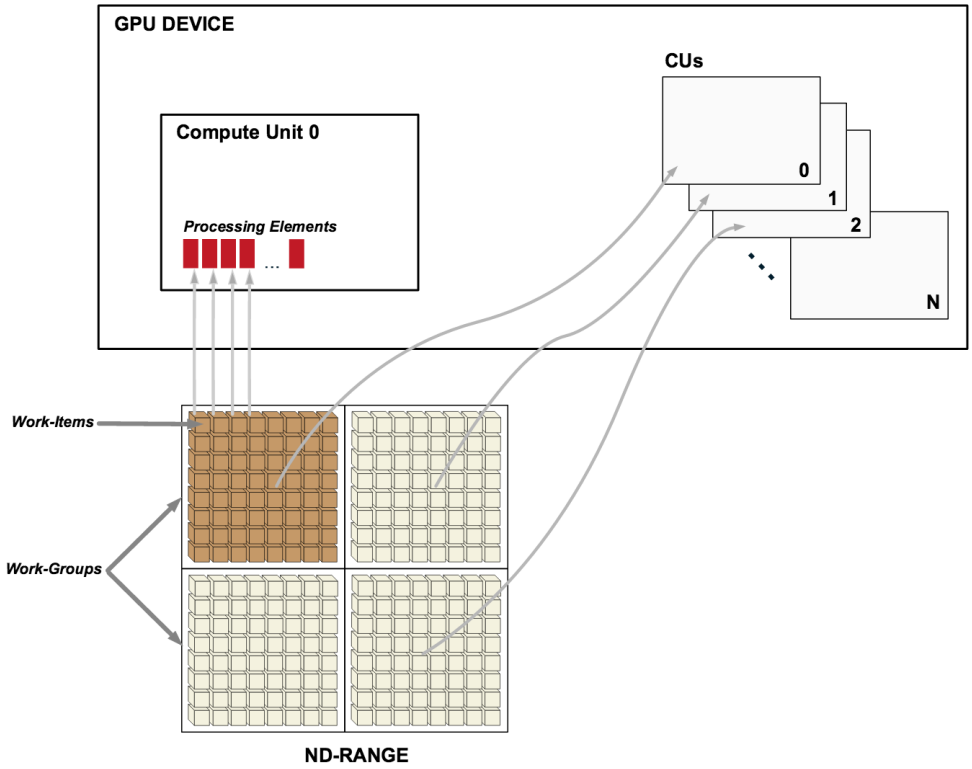

Figure 13 - Simplified Mapping of OpenCL onto a GPU device (source AMD 2013).

Like we see in Figure 13 (bottom), maximum occupancy of a device occurs when every possible work-item is occupied[8], and also when every CU is also occupied. When the developer want's to issue a job using a kernel, it must specify the jobs dimensions and the global size on each of those dimensions, for example purposes lets say a 1D 100 item job (follow the visual description of the example in Figure 14), and then he might specify himself a *local size* (work-group size), of which the *global size* must be a multiple, or he might let the API pick a *local size* for him. And lets consider also that for this kernel, he has a maximum work-group size of 7 and also the CD has 4 compute units available. The best work size, which the API itself would probably pick, would be 5, and 5x20=100, so we will need 20 work-groups of 5 work-items each. As the device as 4 CUs, 20/4=5, means the device will run the same kernel 5 times, using all compute units, each compute unit with 5 of the 7 possible threads busy, ≈72% occupancy. A very bad occupancy rate one might say.

Optimizing the example above could be done by finding the next multiple of 7 that contains the job's global size, which would be 105, and also introducing a condition on the kernel code that confirms its position inside the "original" global size (100), or else it does not do nothing. By setting the *global size* to 105, the API would be "able" to pick 7 as the work-group size, and thus as 105/7=15 a total of 15 work-groups. As in the example there were 4 compute units, 15/4=3.75, meaning a total of 4 executions, 3 of which would have 100% occupancy and the last only 75%. Still much better than the previous solution, and as 25% free, equals a whole compute unit, the developer could issue another kernel at the same time that would execute simultaneously.

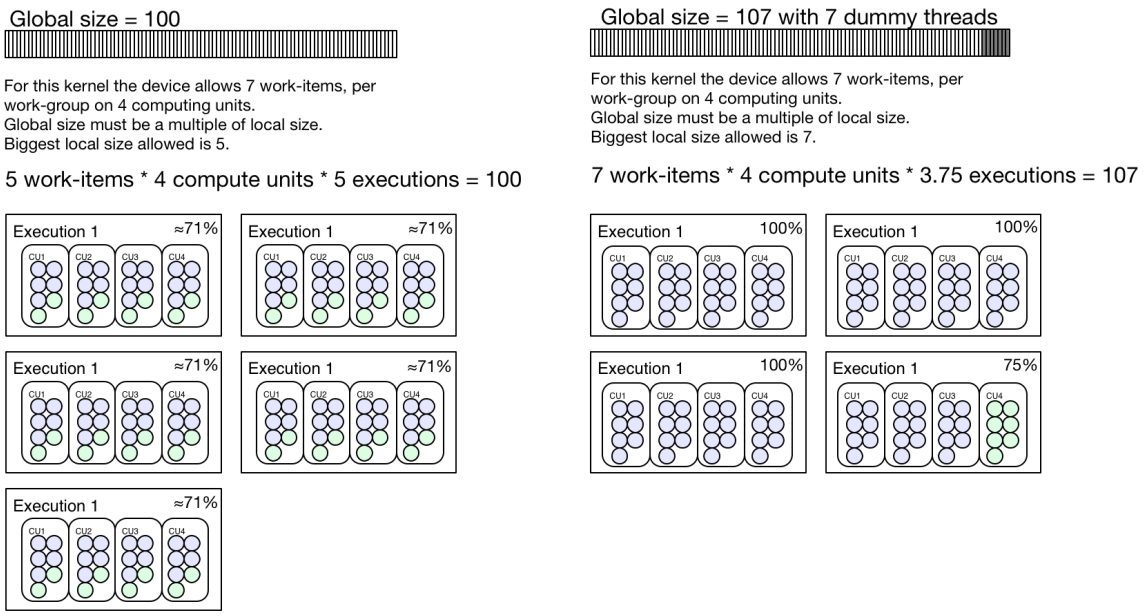

Figure 14 - Example of usage of "dummy" threads in the queueing of OpenCL kernels to optimize execution.

Other optimizations exist, all dependent of the developers knowledge of how the OpenCL framework works and also the capabilities, functionalities and optimizations hardware

[8] Remember that different kernels might allow for different amount of work-items per work-group.

implementations, in general, have. In Chapter 5.5, where we describe our OpenCL implementation we present a different method of optimization for concurrent memory accesses from work-items.

4.2.2. OpenCL C

We mentioned before how the number of work-items per work-group is dependent not just on the device characteristics but also the kernel. The type of operations the kernel executes influence this, but also the local and private memory each kernel execution requires.

Memory hierarchy in OpenCL is divided in four-levels:

- Global memory: shared by all work-items, slowest;
- Read-only memory: smaller, faster, writable by the host CPU but not the work-items;
- Local memory: shared by the work-items of a work-group;
- Per-element private memory (registers).

The programming language used in kernels is called OpenCL C and is based on C99. Memory buffers kernels have access to reside in specific levels of the memory hierarchy, and pointers to those buffer are annotated, in the kernel codes arguments and variable definitions, with the region qualifiers *__global*, *__local*, *__constant*, and *__private*, reflecting the levels described above.

OpenCL C was extended to facilitate the use of parallelism with vector types and operations, synchronization, and functions to work with work-items and work-groups. We will briefly describe the most important ones, which are crucial for work-items and job synchronization.

Unarguably the most important for parallelization of tasks, in our opinion, is the *get_global_id(int dimension)* function, which lets each kernel execution to know its index in the job. An analogy with CPU code is the iterator variable in a *for* loop, that is incremented at every loop and used inside the *for* block to manipulate the correct data. The *dimension* argument in the function refers to the dimension for which we want the global id of this work item.

Work-items can be synchronized by introducing "locations" in the code that every work item has to reach before any of them can go past it. This is done with the *barrier* function which takes as argument one of two constants (CLK_LOCAL_MEM_FENCE, CLK_GLOBAL_MEM_FENCE), which specify if the barrier is local or global. This can useful for instance when work-items, for example, write to a buffer and then want to read from it, making sure every work item has finished writing.

Its important to notice while OpenCL allows for conditional divergence between work-items, it should be avoided when possible, for optimization reasons. Divergence between work-items in GPU hardware is usually resolved by having work-items follow both possible paths, due to the optimizations introduced in thread managers for high throughput

streaming of threads, as suggested by multiple introductory documents to the OpenCL programming model, for example the work of Tompson (2012) provides a good insight on the topic.

As we mentioned before, local memory is shared between work-items, this, depending on the Device available memory per compute unit and the memory each work-item will require, influence the number of work-items that can run concurrently on each compute unit. OpenCL provides built in functions to help manage work-items access to this local memory, namely functions like *get_local_size(int dimension)* and *get_local_id(int dimension)*, the former give the number of work-items in the work-group per dimension, and the later the position of this work-item in the local dimensions.

In Chapter 5.5 we will show examples of these functions usage.

4.2.3. OpenCL/OpenGL interoperability

A crucial feature for our application's performance is the interoperability between OpenCL and OpenGL, by sharing buffer and textures.

Without the ability to create and/or manipulate data in OpenCL and then use it immediately, data would have to be copied to the CPU after the OpenCL operations and then uploaded back to video memory, onto OpenGL buffers. This operation is performed over the PCI Bus and not fast enough to be a per frame operation, not just in speed but in latency.

5. Implementation

This chapter presents detailed descriptions of each of our implementation steps towards the realization of a multi feature data visualization application and its graphical user interface. Bear in mind that some of the implementations steps described were merely experiments, performed for different reasons, either to test the feasibility of an idea or to test if a different technique presents better results.

We attempted to organize this chapter by aggregating implementations regarding the same or similar features, and at the same time keep the order as approximate as possible to the chronological order by which each of the features were implemented.

First, in chapter 5.1 we detail our database structure, data import scripts and any other step taken towards making the Sales Dataset accessible to our application. In chapter 5.2 we address the map projection we used to project all our spatial information in GPS coordinates onto a 2D plane in a 3D space, and then onto the 2D space of the computer screen.

We then enter the actual implementation of the visualization model. Starting with the implemented approach to create the geographic heat zones on chapter 5.3, and then in chapters we 5.4 and 5.5 our two different implementations of the accumulation of values per heatmap item, using the Graphics API and the OpenCL API respectively. Past the computational processes we enter the implementation of the more visual elements, starting with the actual coloring of the geographic heatmap when drawn onto the Screen Buffer in chapter 5.6

In chapter 5.7 we describe the different methods we implemented to display the administrative and demographic information, and finally in chapter 5.8 we briefly describe other additional features implemented.

To help visualize where in the structure of the application each implemented feature fits, we included Figure 15, in the next page, that provides an overview of the most relevant elements implemented, and which are described during this Implementation chapter. We chose a flow diagram to include and give a sense of the actual cycle happening at each frame.

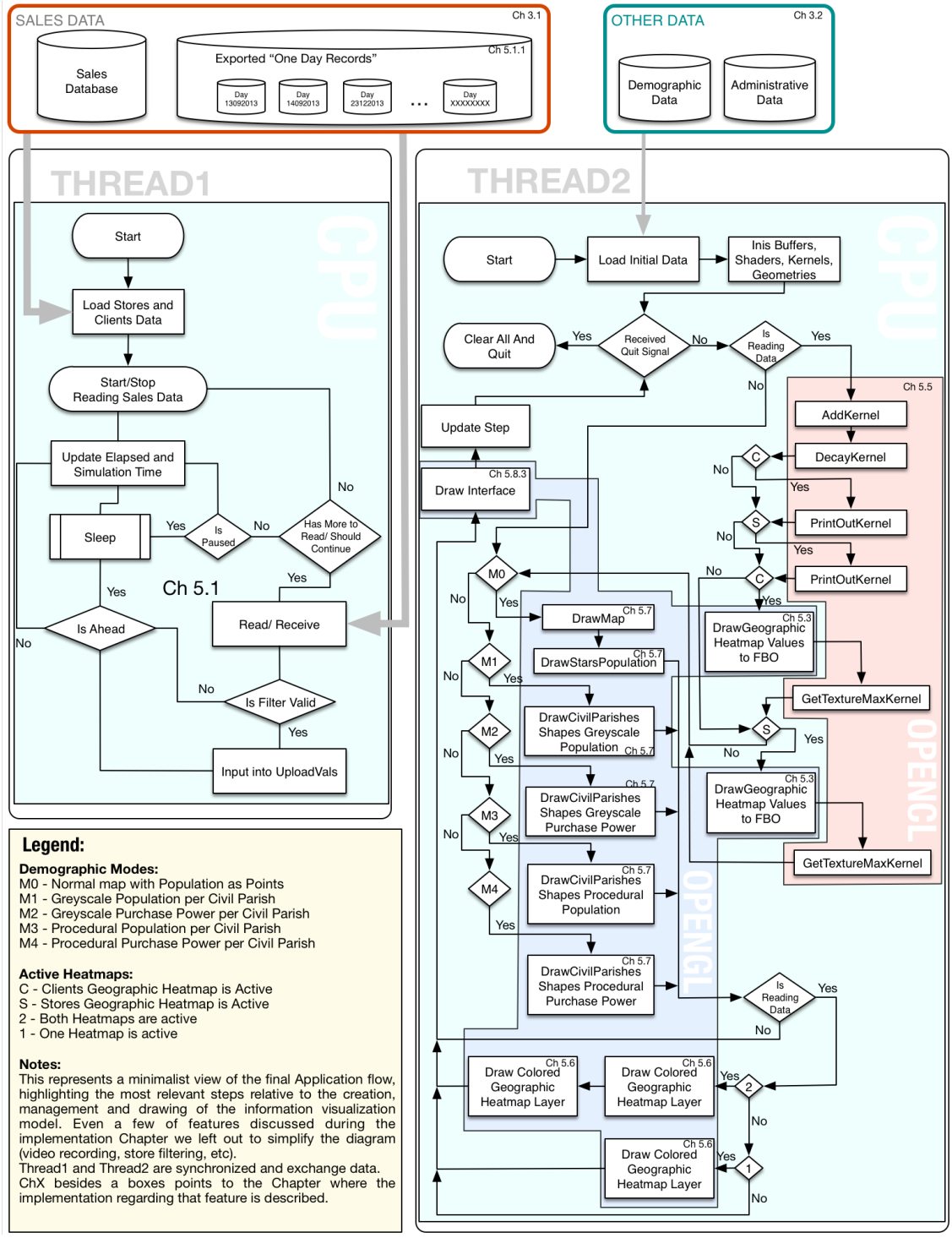

Figure 15 - Minimalist view of the steps relevant to the creation, management and drawing of the information visualization model components. On the left the thread responsible for Database connection, Data Files reading and parsing/streaming the data to Thread2. On the right the thread responsible for uploading new data to video memory, execute operations kernel programs, draw the correct information to pre-buffers (FBOs) and finally draw the information as layers onto the Screen Buffer. Notice that the main cycle presented in Thread2 is our per frame cycle, which we will refer to as Render Cycle also.

5.1. Database and Data Stream

Providing the data to the application at high data transfer rates and with short delaying times is dependent of multiple factors. For instance the data base configuration, its structure, but also the application's implementation it self, as to receive data at high data rates, one has to parse it also at high rates.

5.1.1. Database Configuration and Structure

For our MySQL implementation we opted for innoDB as our database storage engine, our choice was based on Oracle's MySQL documentation, Oracle 2015 (Storage Engines), and on the work of other researchers regarding database comparison as the ones of Tudorica (2011) and Rabl (2012). Based on these documents, not only does innoDB present better performance results, when compared to other engines, on read intensive operations, it also has features that make it a reliable database engine. Such features include Transactions, row level Locking granularity and is full ACID compliant (Atomicity, Consistency, Isolation, Durability), among others.

We also relied on suggestions from the authors mentioned above, and the documents from Oracle's MySQL documentation regarding innoDB configuration, Oracle 2015 (InnoDB Configuration), and innoDB configuration optimization, Oracle 2015 (Optimizing InnoDB),, to balance our cache and read-ahead settings into using the 8Gb of RAM available in our database server, increased the number of maximum threads to make the best of our 8 core database server CPU and set the engine to save one individual table per file. Our first tests indicated these were enough to achieve the data transfer rates we needed while reading data, around 15Mb per second over Local Area Network (LAN).

In order to devise a structure for our database we had to take into consideration the posterior analysis we wanted to perform on it. In other words, due to the number of records present in our data set, we had to devise a structure to hold our data that would not only divide the data into smaller chunks in order to boost query performance but at the same time not divide past a point that would demand for increasingly complex queries in order to perform analysis and comparisons between significant groups (e.g. comparisons with data grouped by Hour).

Grouping the data in time, considering the data regards sales, seemed to make more sense that grouping by any other Variable. Not only would it allow for grouping into smaller chunks than with other Variables, but also would maintain most of the structure of the original data while at the same time still allow for analysis performed with data grouped by other variables. Dividing the data by Day, refer to Figure 16, also allowed to drop the variable Date since its value became implicit for each of the different tables, removing the necessity for its indexation. At the same time, enabling the MySQL option to save one table per file ensured that parallel access to multiple days would not block due to concurrency. This produced tables with around 3.8 million records each.

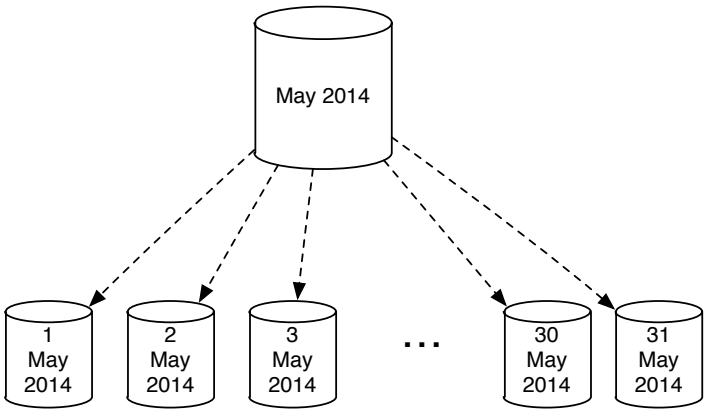

Figure 16 - Distribution of records in the database, one Database table per Day.

The data import step was actually divided into two separate steps for performance purposes, around 6 hours per month instead of over a day. The first step was to perform a direct import of the data in the files without parsing data types, and the second step copied the data from the imported data onto a new table with correct data types, performing the conversion during the copy process. In Sources 1 and 2 below we present the queries used to perform those tasks.

First import table creation query

```
CREATE TABLE `pur201401` (
`TIME_KEY` varchar(30) DEFAULT NULL,
`LOCATION_CD` varchar(5) DEFAULT NULL,
`LOCATION_DSC` varchar(30) DEFAULT NULL,
`SKU` varchar(20) DEFAULT NULL,
`TRANSACTION_HOUR_KEY` varchar(30) DEFAULT NULL,
`POST_CD` varchar(10) DEFAULT NULL,
`CUSTOMER_ACCOUNT_KEY` varchar(30) DEFAULT NULL,
`NET_SLS_AMT_EUR` varchar(10) DEFAULT NULL,
`QTY` varchar(10) DEFAULT NULL,
`PROD_DSCNT_ISSUED_AMT_EUR` varchar(10) DEFAULT NULL
) ENGINE=innodb DEFAULT CHARSET=latin1;
```

Import from file query

```
LOAD DATA INFILE '/home/sonae/datafile.csv' into table pur201401
FIELDS TERMINATED BY ';' ENCLOSED BY ""
LINES TERMINATED BY '\r\n'
IGNORE 1 LINES;
```

Source 1 - Table creation query for the first table where imported data is stored, on top. In the bottom the import query.

Correct data types table creation query

```
CREATE TABLE `dia20140427` (
  `TIME_KEY` date DEFAULT NULL,
  `LOCATION_CD` int(11) DEFAULT NULL,
  `SKU` varchar(20) DEFAULT NULL,
  `TRANSACTION_HOUR_KEY` time DEFAULT NULL,
  `POST_CD` varchar(10) DEFAULT NULL,
  `CUSTOMER_ACCOUNT_KEY` varchar(30) DEFAULT NULL,
  `NET_SLS_AMT_EUR` float DEFAULT '0',
  `QTY` float DEFAULT '0',
  `PROD_DSCNT_ISSUED_AMT_EUR` float DEFAULT '0',
  KEY `location_cd_dia20140427` (`LOCATION_CD`),
  KEY `location_cd_hora_dia20140427`
(`LOCATION_CD`,`TRANSACTION_HOUR_KEY`),
  KEY `hora_compra_dia20140427` (`TRANSACTION_HOUR_KEY`),
  KEY `sku_compra_dia20140427` (`SKU`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Copy and parse data types query

```
INSERT INTO parsed201401 (TIME_KEY, LOCATION_CD, LOCATION_DSC,
SKU, TRANSACTION_HOUR_KEY, POST_CD, CUSTOMER_ACCOUNT_KEY,
NET_SLS_AMT_EUR, QTY, PROD_DSCNT_ISSUED_AMT_EUR)
SELECT TIME_KEY, LOCATION_CD, LOCATION_DSC, SKU,
TRANSACTION_HOUR_KEY, POST_CD, CUSTOMER_ACCOUNT_KEY,
CAST(REPLACE(NET_SLS_AMT_EUR, ',', '.') AS DECIMAL(20,4)),
CAST(REPLACE(QTY, ',', '.') AS DECIMAL(20,4)),
CAST(REPLACE(PROD_DSCNT_ISSUED_AMT_EUR, ',', '.') AS DECIMAL(20,4))
FROM pur201401;
```

Source 2 - Table creation query for the table where the copied data with correct data types will be saved, on top. In the bottom the query that copies the data and casts correct data types.

5.1.2. Data Stream Parsing

In the application side, we use an independent thread to query, receive and parse the data. We chose to implement a separate thread to minimize the impact in the visualization's performance, of the process of receiving the data and saving it into CPU memory.

Both threads are synchronized and exchange data. The visualization thread continuously provides and switch a buffer to the parsing thread on which it saves the received information. The visualization thread is then responsible for uploading the values to the GPU memory where they will be processed.

The Data Stream Parsing thread is also responsible for gathering data about clients and stores from the database and supply it to the visualization thread.

The passage of time, with which the parsing thread throttles the stream, is supplied by the visualization thread for consistency purposes.

5.1.3. File system day exports

To eliminate the mandatory network access to the database, source of the visualized data, we devised a simple way of using a file reader stream as a simulated database stream, allowing us to export data files (CSV format) relative to days, pack them along side the application in a Folder, inside the application file structure, specific for that purpose.

Using the SSD of our test case hardware, the file solution is performant enough, even considering the additional resources used, that the network solution.

5.2. Projections

Has we have seen so far, all our spatial data comes from real word locations in geographical space, more exactly GPS coordinates. GPS coordinates map points on the surface of a sphere through two angles, latitude and longitude. To create a visualization using this geographical information we have to transform the data points, first onto a 2D plane in a 3D space using a Map Projection, and finally onto the 2D pixel space of the screen, as Figure 17 depicts.

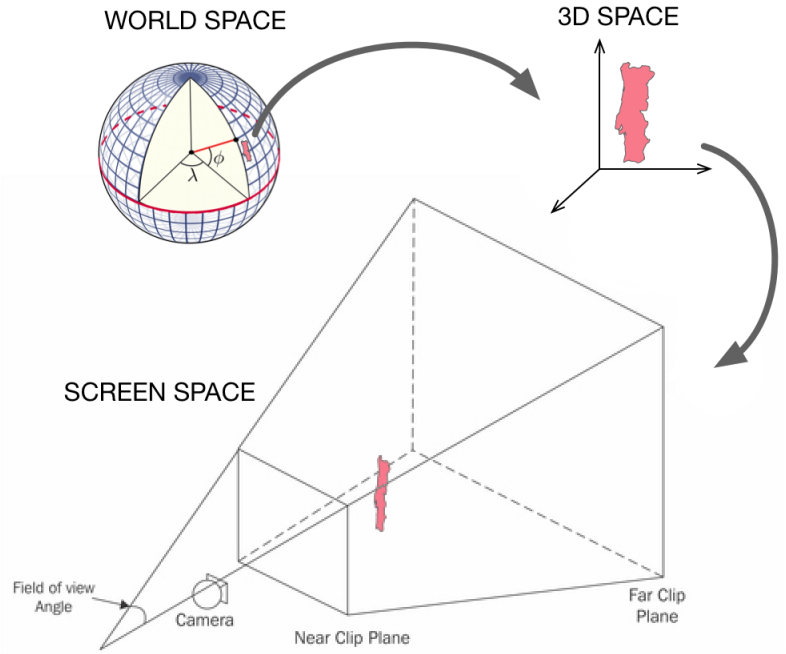

Figure 17 - Projecting From GPS coordinates to Screen Space

5.2.1. Geo-Projection

There are several different map projections to choose from, too many to reference here. No map projection is ideal, each map projection preserves a different property but consequently others are distorted. Map projections fall in one of the different categories, either preserving direction, shape, area, distance or shortest route.

Considering that our final use for the shapes and points is a Data Visualization application, and one that will also perform blending of circular areas on top of the geographic information, we opted for the Mercator Projection, as seen in Feeman (2002, chapter 9), as it preserves angles and shapes. Unfortunately the Mercator Projection does not preserve areas, areas get distorted the most the furthest away from the equator. This effect is most noticed in maps showing huge areas or even the entire world. When used to visualize a small area like Portugal, the relative distortion is almost negligible.

The Mercator projection if given by Equation 1 and 2, where λ is the longitude and ø the latitude. λo is an arbitrary longitude where to center the x origin.

$$x = \lambda - \lambda o \tag{1}$$

$$y = ln\left(tan(\phi) + \frac{1}{cos(\phi)} \right) \tag{2}$$

5.2.2. Projection Matrix

Computer screens are flat, 2D surfaces. In OpenGL, the actual 3D space rendered to the screen, ranges in all 3 axis (x, y, z) from -1 to 1, in other words, a cube of 2 unites in size centered at the origin (Clipping Space). Everything that falls outside this cube is discarded (clipped). Everything that falls inside the cube is projected onto a 2D rectangle equally ranging from -1 to 1 in both x and y axis, which represents the screen space (more accurately the Viewport Space).

Yet, usually application's 3D space does not fall in the range -1 to 1, and many times the application Point Of View (POV) is not fixed, like with a real world camera it can move and rotate. Even objects in this 3D space can rotate or move independently of each other, defining its own Space where its vertices reside.

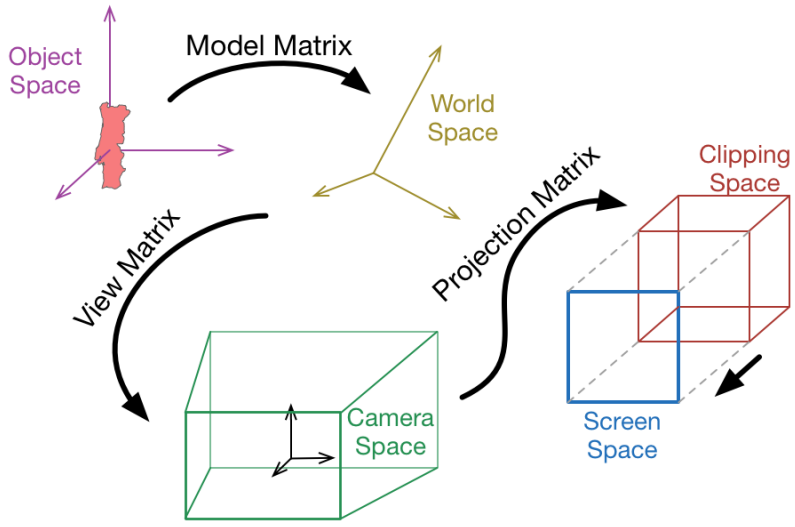

Figure 18 - Model View Projection Matrix transformation to Clipping Space and Clipping Space Projection onto the 2D Screen Space

This series of projections, from Model Space to World Space, then to Camera Space and finally to the Clipping Space, is usually called Model View Projection (MVP), refer to Figure 18 for a visual overview. Projection onto each of these Spaces is done by multiplying vertex coordinates (x,y,z,1) with a [4x4] matrix that defines the scaling, rotation and translation transformation that project the vertex onto that Space.

In our implementation all points that originated from geographical data share an identical Model Matrix. Identical and not "the same" because our implementation allows for manipulation of an individual Mesh's Model Matrix (Mesh as in a group of vertices all hierarchical descendants of the same object). In our implementation Model Matrices were used for Scaling and Translating purposes.

Regarding the View Matrix, as our application is essentially a 2D Visualization the virtual camera never performs rotations or tilts, instead we used the View Matrix to pan and zoom the visualization, essentially Translation.

Last, the Projection Matrix, which defines the shape and size of the actual viewable space, is where we one might consider we chose oddly. Being essentially a 2D Visualization there is no reason to opt for other than an orthogonal projection, yet, considering there was no advantage over a perspective projection, regarding our application, we opted to leave a door open, building our implementation around a perspective projection, allowing for any future feature that might take advantage from it. We used a perspective projection while rendering all objects except for the ones composing the User Interface, for which we used an orthogonal projection. An orthogonal projection while designing the User Interface allows for precise positioning of elements relative to the Viewport Dimensions.

The detailed description of how to calculate each of these projection matrices is given by Ho (2015). Equations 3 and 4 are the equations we used in our implementation, where n is the near plane distance, f the far plane distance and r, l, t, and b the rightmost , leftmost, topmost and bottommost coordinates respectively.

$$\text{Perspective Matrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (3)$$

$$\text{Orthogonal Matrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

Example implementations can be found all over the internet, as these equations have been in use since development of 3D computer graphics started. The most significant source of such and other very useful implementations, as we will see later, is the GLUT source code, as seen in GluProject (2015), used as support library back when OpenGL fixed pipeline was still in use.

5.2.3. Un-project

Un-projection[9], as suggested by the name, is the act of performing projection from the 2D Screen (our projecting target surface when rendering) to the 3D world (which we projected from). This is useful to convert a specific coordinate in Screen Space, the mouse position for instance, to World Space, allowing for mouse input interaction with the rendered objects. Still un-projection is just one of the steps as we will see in Chapter 5.8.3 where the User Interface implementation is described.

The un-project step it self is not a direct point to world coordinates direct transformation either. Screen Space is defined in 2D, consequently excluding any depth information necessary for the complete un-projection. To circumvent this issue we use the knowledge of both the near and far planes from the Clipping Space as two known depths, giving us two points that when un-projected define a line segment that transverses the 3D World space, refer to Figure 19. This segment is usually called a Ray, and can be used to test for geometry interceptions, more on this later in Chapter 5.8.3.

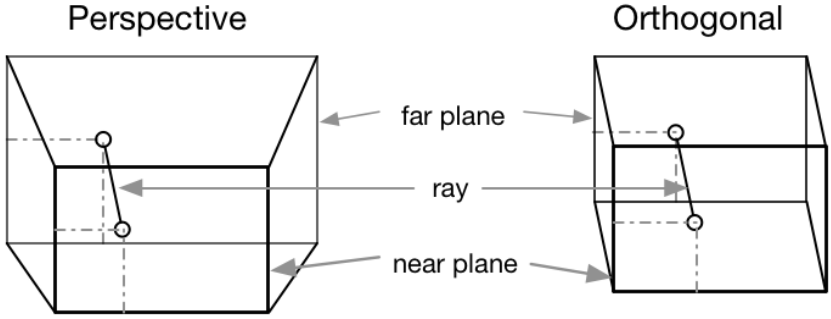

Figure 19 - Un-projection resultant line segment, still in Projection Space.

[9] Un-projection, to differentiate from the inverse of a projection, a term used in the Glut implementation.

To un-project each of the points we first set the z coordinate of each of the points to the corresponding z values of the near and far clipping planes, -1 to the near plane and 1 to the far plane. Then we calculate the inverse of the MVP Matrix (iMVP) and transform both points using the iMVP. Finally we multiply each of the x, y and z components of both points by the multiplicative inverse of the each's w component.

5.3. Geographic Heatmap Concept and First Approach

A Heatmap, or Pixel Heatmap to differentiate from other types of heatmaps, is a graphical representation of data where values of individual items are represented as color, in a one or two dimensional space usually discretized by item. This concept in computer graphics is equivalent to the representation of individual values with the color of each pixel of a 2D texture.

In a Geographic Heatmap, the 2D space of the texture is mapped to real world space, usually in one of two ways. The first is by discretizing the space, defining a minimum unit (squared usually) and paint the geographic space, square by square, according to the values inside each square. Almost like overlaying a pixel heatmap on top of the geographic space, refer to Figure 20 left. And the second consists in coloring the map continuously in world space, using the registered values as centroids of heat zones that spread outwards according to a specific mathematical function, usually Gaussian, refer to Figure 20 right.

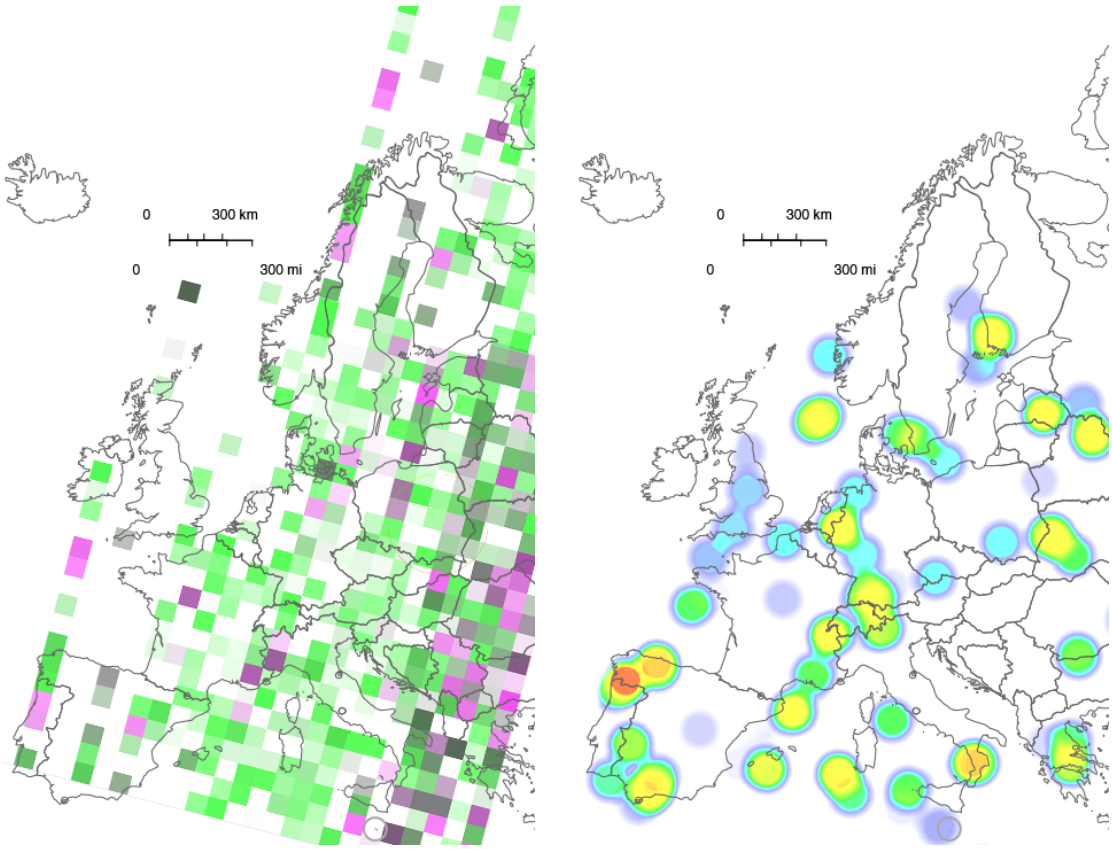

Figure 20 - Two different rendering types of geographic heatmaps. On the left discretized units, on the right continuous coloring.

Considering that our Sales data does not relate to the geographic space itself but to individual Stores, it would made no sense do discretize the space and represent different accumulated sales through squared areas on the map, as each squared zone value would have to still be interpolated from the Store sales and locations. Instead, in this first approach, we chose the Stores as our discretized abstract units, calculating the heat value and its decay per Store from the Sales happening in that Store. In other words we used the Store locations as our Heat Centroids and distribute the Heat Value of each Store continuously, in the geographic space around the Store using a 2D Gaussian function. The Heat Value was used both for the Amplitude and Spread of the Gaussian distribution. See Equation 5 for the 2D Gaussian function, where A is the amplitude, σ the spread of the blob in the x and y dimensions and notice its centered at coordinates (0.5,0.5).

The reasoning for this approach is that, in terms of coloring this conforms with the usual approach, differentiating zones of greater accumulation through the use of color, and in terms of the size of the affected heat area, we emulate the idea that usually Stores with greater sales values service bigger areas, and consequently present a visual approximation of the serviced areas.

$$z = A \cdot exp\left(-\left(\frac{(x-0.5)^2}{2\cdot\sigma^2} + \frac{(y-0.5)^2}{2\cdot\sigma^2} \right) \right) \tag{5}$$

If we were to implement the rendering of the geographic heatmap by going through each pixel of the area being visualized we would have to, not only perform calculations for pixels outside the influence area of any Centroid (empty pixels), but also, for each pixel we would need to calculate distances to each of the centroids to determine its contribution to that location.

Instead, our implementation takes advantage of the fact that Store locations are distinct from each other. By defining a bottom threshold for the 2D Gaussian function beyond which its values are ignored (equal zero), we can calculate the size of the area each Centroid will affect. With that information we can map triangles to cover those areas and limit the rasterization to the surface of those triangles, therefore minimizing the number of processed pixels. Regarding the dispersion, σ in Equation 5, we chose a value of 0.1443, to scale the dispersion of values to the Texture Coordinates Space. See Equation 6 for the adapted 2D Gaussian function.

$$\begin{cases} z = \exp(-(24*x^2 + 24*y^2)) & ,\exp(-(24*x^2 + 24*y^2)) \geq 0.01 \\ z = 0 & ,\exp(-(24*x^2 + 24*y^2)) < 0.01 \end{cases} \tag{6}$$

To implement this using the OpenGL API we created a three shader pass with a Vertex Shader (VS) to perform vertex projection transformations, a Geometry Shader (GS) to generate the triangles that cover the heat areas and a Fragment Shader (FS) to color those triangles. Refer to Figure 21 for an overview of the full process. From this point forward we will refer to this Shader Program (VS + GS + FS) GeoGauss Program.

To achieve this we first stored our Store location information using a Vertex Array Object (VAO), where each vertex structure is composed of a Position followed by TextureCoordinates. The Position is the Store location coordinates after the Mercator Projection and the TextureCoordinates are used to map each Store to a unique (one per Store) position inside a Texture. The Heat value of each Store is stored in the color information of that Texture(s), not the vertex structure. On chapter 5.4 we provide additional justification for this choice of vertex structure. All this information stored using the VAO is constant and once uploaded to the Video Memory once, it is never uploaded again. Also the vertex elements are Indexed, Draw calls are made referencing the index buffer and as GL_POINTS, to conform with our implemented GS input output settings. The indexing allows us to filter stores during the draw calls without changing the VAO information, by defining and using different index buffers.

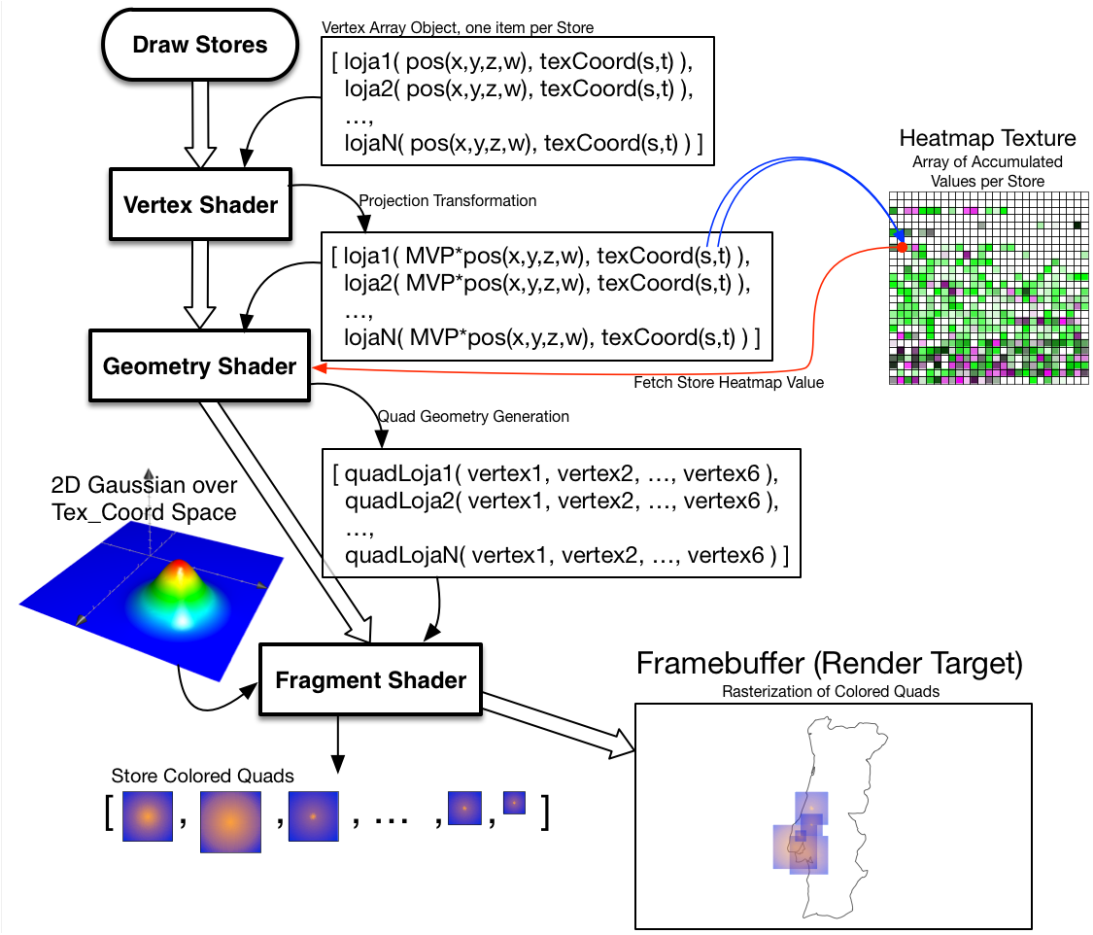

Figure 21 - Description of the Draw Pass used to render the geographic heatmap. A Geometry Shader generates Quad geometry, one Quad per Store location. Quad sizes vary according to the accumulated value for the Store fetched from the Heatmap Texture. The fragment shader colors each Quad according to the 2D Gaussian Function, centered at texture coordinates (0.5, 0.5) and with an amplitude value fetched from the Heatmap Texture.

After the vertices projection transformations in the VS stage, as described in chapter 5.2.2, they are outputted to the GS stage. The GS executes once for each vertex, first the Heat value is fetched from the Heat Texture using the vertex Texture Coordinates, and used to

determine the size of the Quad (2 triangles forming a square) that must be generated, according to Equation 7, where value is the *value* for this Store and *maxValue* the maximum value between Stores. The GS generates and outputs 2 triangles per Store, but only if this 2 triangles are not completely outside the current FOV, to avoid unnecessary processing. The outputted triangle primitives will then enter the Rasterization stage where the rasterization process generates fragments to be processed, per-fragment, in the FS stage.

$$quadwidth = 2 \cdot \left(0.01 \cdot \frac{\log(value)}{\log(\max Value)} + 0.02\right) \tag{7}$$

The vertices that compose the triangles were given positions by the GS, but also TextureCoordiantes. The FS is then responsible to calculate the 2D gaussian value for each fragment using Equation 6, with the TextureCoordinates(s,t) as (x,y), assigning the result value to the red component of the fragment color.

Setup GL_BLEND

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
glBlendEquation(GL_FUNC_ADD);
```

$$\begin{bmatrix} (sR*1) + (dR*1) \\ (sG*1) + (dG*1) \\ (sB*1) + (dB*1) \\ (sA*1) + (dA*1) \end{bmatrix} = \begin{bmatrix} rR \\ rG \\ rB \\ rA \end{bmatrix}$$

Source 3 - On top, the GL_BLEND configuration for geographic heatmap drawn, blending the quads with each other by adding values. On the bottom the resulting color blend equation.

The Quads generated by the GS are drawn one after the other, many times overlapping each other. We want the value distributions of each Quad to add to each other and to do this easily we use the GL_BLEND functionality of OpenGL with the necessary settings presented in Source 3.

The target of this render pass is a Framebuffer Object (FBO), effectively a Texture, that will later be drawn to the Screenbuffer overlaying the Map Layer (more on that later). This allows us to quickly switch the FS that maps the calculated values to real colors on the Screen. Our first implemented FS for this job used the normalized values to pick a color from a color map Texture, refer to Figure 22 left, on the right the first rendering result we obtained.

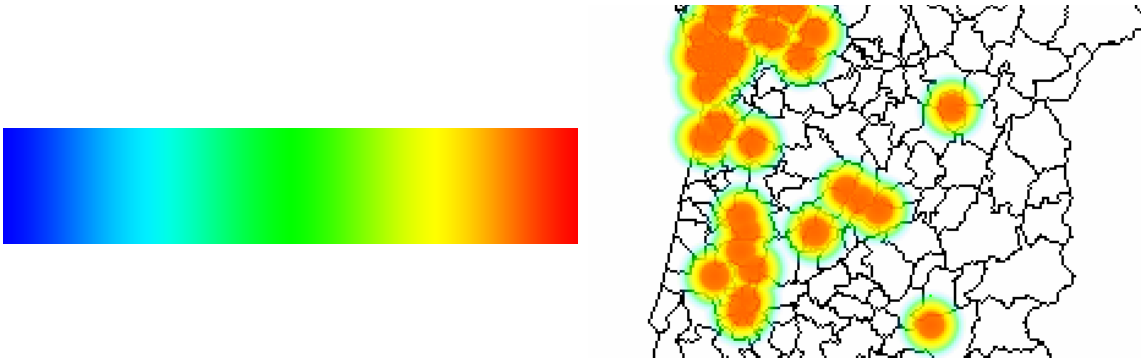

Figure 22 - On the left our first color mapping Texture for the values calculated during the Quad generation process, on the right our first rendering result.

5.4. Pixel Heatmap - Graphics API

In the previous chapter we mentioned the Heatmap Texture a couple of times, without going into much detail about what it is, how it is calculated and why we used it. We chose to separate its description into more than one chapter because of how relevant this feature is to the whole visualization model and because we implemented its calculation using two quite different methods, which could cause some confusion.

We could have added additional attributes to the vertex structure to store the Heatmap accumulation values, but manipulating that information and being able to loop it over and over through the Graphics Pipeline while minimizing data transfers between the CPU and GPU would require additional calculations on both parts. On the GPU end we could use the Transform Feedback feature to perform Vertex Buffer Object (VBO) to VBO operations, still, as the resulting VBO would be used in the next Draw cycle and therefore effectively replacing the first one, every Store would have to be processed each time to avoid "losing" Stores between cycles, but also no more than one time, as having the same Store two times with different accumulation values would make no sense. That would require additional calculations to be performed on the CPU side as well, recreating and scanning the Store array between each Render Cycle.

Textures on the other hand are quite versatile. They are accessible in both the Geometry and Fragment shader stages, their inner datatype can be chosen to fit the intended purpose and they can also be bounded as Render Targets, through the Framebuffer Object (FBO) functionality.

So, this provides an easy way to save the accumulated values when the texture is the render target and use them as input during the GeoGauss Program. The Texture has to be big enough to fit the Stores, we use the square root of the number of stores to determine the appropriate minimum number of cols and rows of pixels needed in the Texture. For instance for 702 stores, the square root is approximately 26.5, the Texture size would be 27x26. Refer to Figure 23 for a visual description using the stores heatmap texture as example.

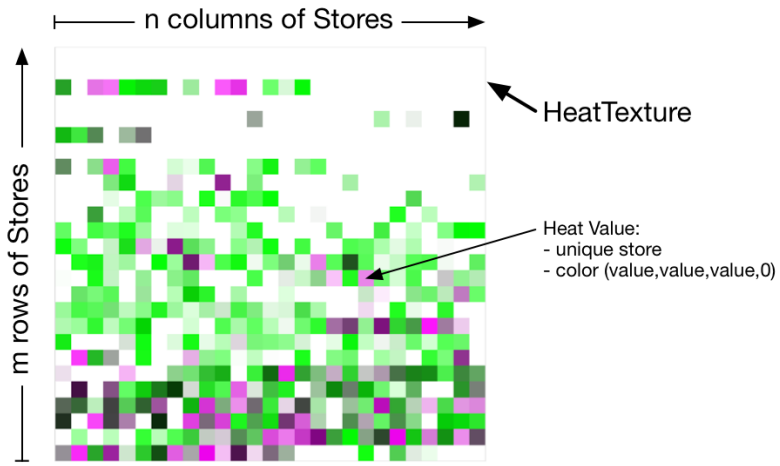

Figure 23 - HeatTexture structure. Real colors were inverted.

Now we are just left with the "how to" manipulate the values inside the texture, add to them as new value arrive from the database and perform a continuous small decay of the all the value over time. To achieve this we first, when the application loads, fill in an Array of custom tailored Vertices, one for each Store, with their positions mapped onto the Texture Space, in a way that each Store lays on top of one unique Texture pixel. When the CPU receives Sales values from the database stream, it Clones the Vertex corresponding to the Store where the Sale occurred, sets the value of the Sale in the red component of the Vertex color, and then adds that Vertex information to a Vertex Buffer Object (VBO). There are two VBOs for this process, used as a double buffer, whenever a new render cycle starts, or the VBO is full (never happened), VBOs are switched. If several Sales are present for the same Store, there will be one Vertex per Sale for that Store, which will not cause a problem and the addition is shifted to the GPU as we will see. When the VBOs are switched, the "full" VBO is uploaded to Video Memory and is CPU memory cleared when finished.

To render, and effectively add, each of these Vertices, that represent values that correspond to Sales in Stores, we implemented a simple Shader Program, that we will call HeatmapAdd Program, constituted of a Vertex Shader that passes Vertices directly forward in the graphics pipeline, without any transformation, and a Fragment Shader that simply outputs the Vertex Color. Figure 24 presents an abstract representation of the blending process onto the heatmap texture. The mechanisms and properties that are responsible for the correct addition of the Values are external to the Shaders themselves, one is the positioning of the vertices "above" the specific pixel of the corresponding Store, in the center of the pixel to be more exact, also the transparency of the vertices allowing for blending calculations, drawing the Sales vertices as GL_POINTS 1 pixel in size, and choosing the correct GL_BLEND equation and function. The blending configuration used to add the values is the same present in Source 3.

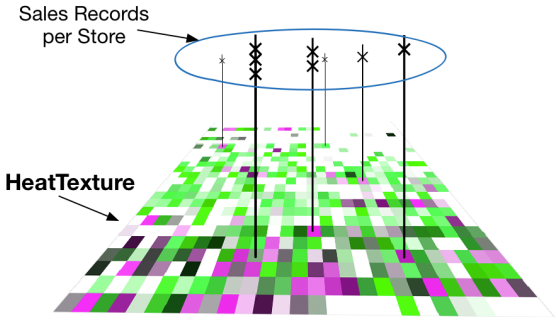

Figure 24 - Abstract representation of how Sales Vertices are positioned above the corresponding Store's pixel, in order to be added to the HeatTexture during the blending process. Blending configuration for the addition is described in Source 3.

Another majorly important step in the heatmap process is the Decay step, where instead of adding to the Texture we will subtract a value, decaying the current values in each store linearly in time. We achieve this by performing a draw pass, where we draw a single colored transparent Quad (2 Triangles) covering the whole Texture, using the exact shader program as on the Add step but with different GL_BLEND configurations, see Source 4 for the detailed equations, that will in effect configure the blending process to subtract a value specified as the color of the Quad to the value of the destination Texture.

Setup GL_BLEND

```
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_ONE, GL_ONE, GL_ZERO, GL_ONE);
glBlendEquationSeparate(GL_FUNC_REVERSE_SUBTRACT, GL_FUNC_ADD);
```

$$\begin{bmatrix} (dR*1) - (sR*1) \\ (dG*1) - (sG*1) \\ (dB*1) - (sB*1) \\ (sA*0) + (dA*1) \end{bmatrix} = \begin{bmatrix} rR \\ rG \\ rB \\ rA \end{bmatrix}$$

Source 4 - On top, the GL_BLEND configuration for Decay step on the Heatmap Texture calculation. On the bottom the resulting color blend equations, notice alpha will remain untouched and the destination components (dR,dG,dB) will be subtracted by a value specified by us in the source color components (sR,dG,dB).

These two successive steps run in a cycle, starting with the upload to the Video Memory of values retrieved form the database stream relative to Sales in Stores, executing the Add step that add the newly uploaded values to the HeatTexture, execute the Decay step to linearly reduce the accumulated values in order to provide a heatmap relative to a time span (45 minutes), at this point the texture is ready to be used in the render of the Geographic Heatmap described in chapter 5.3. When all other rendering and update processes of the application render cycle are done this process starts again.

5.5. Pixel Heatmap - OpenCL API

The implementation presented in the last chapter, although functional and performant, presented some limitations and disadvantages which lead us to research and experiment with different methodologies.

One disadvantage that affects performance is related to the necessity of uploading vertex positions and color information, eight floating point values in total, for each and every Sale, and while this could be improved by customizing the Vertex structure, we would always need to upload more floating points than Sales, introducing a quite large overhead.

In terms of limitations, relying on the blending process to perform our mathematical operations, limits the amount of information we are able to retrieve in real time. Having more general purpose framework, allows for easy implementation of more complex calculations opening up the possibility of calculating average values, standard deviations, etc.

After studying the OpenCL API, introduced in chapter 4.2, realizing its flexibility and how it would allow us to break free of the limitations presented by the previous implementation, we decided to include additional features for this implementation to demonstrate exactly that. First we added a second heatmap (Texture) for the Clients' Postal Codes, and we introduced an additional value being calculated over time, along side the heat value, consisting in the Total Value of Sales over the Total Quantity. Introducing an additional, bigger (as we will show) heatmap shows how the reduction of the overhead and OpenCL memory optimization features allow us to increase the information being displayed, and adding the Sales/Quantity value not only provides insightful information about Value paid per Quantity across the Country, but at the same time demonstrate how the flexibility of OpenCL allow us to perform this and other calculations, if necessary, in real time with a small performance impact.

When implementing the Heatmap using OpenCL, we didn't want to touch the Geographic Heatmap Rendering (Ch. 5.3), as Textures were still the most adequate data structure to communicate the Heat Values to the rendering processes done using the Graphics API. Fortunately OpenGL/OpenCL APIs provide methods to share data structures between them, allowing us to handle all the operations related to the Heat Values with OpenCL Kernels and render the HeatTexture also using OpenCL, making the switching between implementations seamless.

To accommodate the Heat Values, for Stores and now for Client Postal Codes also, we created two arrays of 32 bit floating point values, with three values per Item, refer to Figure 25. The first value is directly equivalent to the valued saved in each pixel on the previous implementation (Ch. 5.4), decaying over time, and the second and third values contain the Total Paid Amount and Total Quantity of Products respectively. These buffers will never be manipulated by the CPU directly.

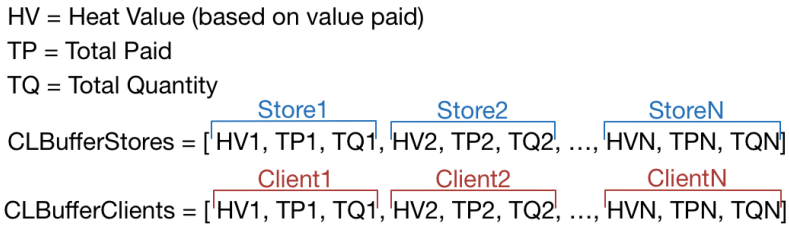

Figure 25 - Data structures defined to use with our OpenCL Kernels.

We defined two pairs of additional buffers, each pair consists of a integer buffer and a float buffer. The integer buffer keeps indexes for Store and Postal Code locations (indexes) in the CLBufferStores and the CLBufferClients respectively. The float buffer contains the values of "Sales Values" and Quantities directly related to the indexes in the indexes buffer. As we can see in Figure 26 we only need to upload four values per Sale.

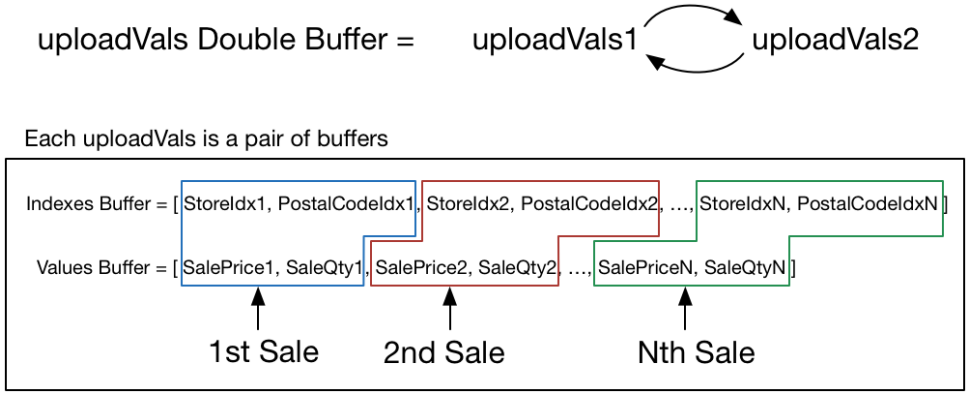

Figure 26 - Structure of the buffers used to upload Sales data to the Video Memory. These buffers will be input arguments of the AddKernel.

These buffers are created with the flag CL_MEM_ALLOC_HOST_PTR, so that the memory allocation of the buffer (CPU memory) is done by OpenCL, which ensures the correct alignment of the memory blocks that provide the best performance with OpenCL, and at the same time, this alignment enables real memory sharing on computers where GPU memory is shared with CPU, or best possible data transfer rate over PCI bus on computers with discrete GPUs. Manipulation of these buffers must be done with care, following *lock* and *release* rules similar to those used for shared variables in multithread environments.

We defined several Kernels to manipulate these buffers, performing the necessary calculations, namely a AddKernel, DecayKernel, PrintOutKernel and finally the GetTextureMaxKernel. Their names might point to their purpose but its important to understand how and when we use them.

Like in the previous implementation, all kernels execute at least once every render cycle, in other words, each Frame. As a new render cycle starts the UploadVals buffer pairs are exchanged (double buffering), the pair that was receiving input values before is unmapped (released/ synchronized to Video Memory) and the other one is mapped to take its place.

First in the Kernel execution sequence is the AddKernel, as we can see by following the source code in Source 5, the AddKernel performs three additions for both the

CLBufferClients and the CLBufferStores buffers, adding the Sale price value in the first two and the quantity in the third. The Sale price is added twice, once to keep track of the Total Amount Paid and the other is the usual heatmap value subject to linear decay over time.

AddKernel

```
__kernel void add(__global float* inValues, __global int* inIndexes,__global float* CLBufferClients,__global
float* CLBufferStores, const int size_in)
{

    int global_index = get_global_id(0)*2;
    int a = 0;
    if(global_index<size_in*2 && a<1){
         //addition of values per Postal Code
        atomicFloatAdd(&(CLBufferClients[inIndexes[global_index]*3]), inValues[global_index]);
         atomicFloatAdd(&(CLBufferClients[inIndexes[global_index]*3+1]), inValues[global_index]);
         atomicFloatAdd(&(CLBufferClients[inIndexes[global_index]*3+2]), inValues[global_index+1]);

         //addition of values per Store
         atomicFloatAdd(&(CLBufferStores[inIndexes[global_index+1]*3]), inValues[global_index]);
         atomicFloatAdd(&(CLBufferStores[inIndexes[global_index+1]*3+1]), inValues[global_index+]);
         atomicFloatAdd(&(CLBufferStores[inIndexes[global_index+1]*3+2]), inValues[global_index+1]);
         a++;
         global_index++;
    }
}
```

Source 5 - The AddKernel source code. The Kernel takes as arguments, in order, the UploadVals float buffer, the UploadVals integer buffer, the per Postal Code heatmap values, the per Store heatmap values and finally the number of Sales being added in this execution.

Next the Decay kernel is executed, once for both the CLBufferClients and the CLBufferStores buffers. This Kernel's simple task is to subtract a fraction, relative to the elapsed time, of the first of every three values. See Source 6 for the source code of this Kernel.

DecayKernel

```
__kernel void decay(global float* currentvals, const float decayamount, const int length)
{
    int global_index = get_global_id(0)*4*3;
    int a = 0;
    while (global_index < length && a<4) {
        float currentValue = currentvals[global_index];
        float newValue = currentValue-(currentValue*0.3333*decayamount);
        currentvals[global_index] = newValue <0 ? 0 : newValue;
        a++;
        global_index += 3;
    }

}
```

Source 6 - The DecayKernel source code. The Kernel takes as arguments, in order, the buffer to be subject to the "decay", the time passed since last execution (deltaTime) and the size of the buffer.

As before, we feed the GeoGauss Program (Ch. 5.3) Textures containing values, mapped in pixels color components, to be represented in the Geographic Heatmap. Previously we had one Texture for the Sales Heatmap, now we have an additional one for the Client Postal Code locations. The size of both Textures is calculated as described in Chapter 5.4. Each of the Textures is generated through the execution of PrintOutKernel, but only if needed, if

the corresponding heatmap visualization is enabled in the User Interface (UI), more about that in Chapter 5.8.3 regarding UI options.

PrintOutKernel

```
__kernel void printout(__write_only image2d_t bmp, global float* currentvals, const int maxval, const int
cols, const int qual)
{
    int idg = get_global_id(0)*8;
    int x=0; int y=0; int a = 0;

    while (idg < maxval && a<8) {
        y = idg/cols;
        x = idg%cols;

        float valor = currentvals[idg*3]; //usual heatmap value
        if(qual==1){ //change value to display
            if(currentvals[idg*3+2]>0){
                valor = currentvals[idg*3+1]/currentvals[idg*3+2]; //calculate Sales/Quantity
            }else{
                valor=0;
            }
        }

        float4 val = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
        if(valor>0){
            float red = valor;
            float blue = valor;
            val = (float4)(red, 0.0f, blue, min(red,blue));

        }
        int2 coords = (int2)(x,y);

        write_imagef(bmp, coords, val);
        a++;
        idg += 1;
      }
}
```

Source 7 - The PrintOutKernel source code. The Kernel takes as arguments, in order, the Texture where to save the values, the floating point buffer from where to read the values, the size of the floating point buffer, the width of the Texture in pixels and finally the choice of value to be saved in the Texture.

The PrintOutKernel, see Source 7, sets Texture colors in batches of 8 pixels per Work Item, allowing for a more effective use of the GPUs caching optimization features. It uses argument *cols* to determine the width of the Texture to be printed and *maxval* to keep inside the Value buffer where it is reading values from. As we intend to display not just the usual heatmap value but also the Sales/Quantity, we use an integer argument (*qual*), to decide which value to "print". The actual division, of the Sales Total over the Total Quantity, is done in "realtime" each time the Texture is generated, as we can see also in Source 7.

The Heatmap Textures generated at this step are configured as to have full 32 bit floating point values for each of the color components, and as such values saved in them are neither normalized nor clamped in any way. The same goes for the Geographic Heatmap Textures generated after execution of the GeoGauss Program (Ch. 5.3), a 2D gaussian distribution is performed without normalization or clamping of values. For that reason, after the execution(s) of the PrintOutKernel, we immediately execute the GeoGauss Program, using the generated Heatmap Textures as input. When the GeoGauss Program execution(s) finishes we perform a search for the maximum value on the Geographic Heatmap Texture(s), with which we will perform normalization later (see Chapter 5.6), using the GetTextureMaxKernel.

GetTextureMaxKernel

```
__kernel void gettexturemax(__read_only image2d_t image1, __local float* scratch, __const int localSearchSpaceSize, __const
int texx, __const int texy, __global float* result) {

  const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

  int global_index_x = get_global_id(0)*localSearchSpaceSize;
  int global_index_y = get_global_id(1)*localSearchSpaceSize;
  float4 pixel;
  float accumulator = 0;
  // Loop sequentially over chunks of input vector
  for (int xi=0; xi<localSearchSpaceSize; xi++){
    for (int yi=0; yi<localSearchSpaceSize; yi++){
      if(global_index_x<texx && global_index_y<texy){
        pixel = read_imagef(image1, sampler, (int2)(global_index_x,global_index_y));
        accumulator = (accumulator > pixel.x) ? accumulator : pixel.x;
      }
      global_index_y++;
    }
    global_index_x++;
  }

  // Perform parallel reduction
  int local_index_x = get_local_id(0);
  int local_index_y = get_local_id(1);
  scratch[local_index_y*get_local_size(0) + local_index_x] = accumulator;
  barrier(CLK_LOCAL_MEM_FENCE);
  int offsety = get_local_size(1) / 2;
  for(int offsetx = get_local_size(0) / 2; offsetx > 0; offsetx = offsetx / 2,offsety = offsety / 2) {
    if (local_index_x < offsetx && local_index_y < offsety) {
      float other1 = scratch[local_index_y*get_local_size(0) + local_index_x + offsetx];
      float other2 = scratch[(local_index_y + offsety)*get_local_size(0) + local_index_x];
      float other3 = scratch[(local_index_y + offsety)*get_local_size(0) + local_index_x + offsetx];
      float mine = scratch[local_index_y*get_local_size(0) + local_index_x];
      float max = (mine > other1) ? mine : other1;
      max = (max > other2) ? max : other2;
      max = (max > other3) ? max : other3;
      scratch[local_index_y*get_local_size(0) + local_index_x] = max;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (local_index_x == 0 && local_index_y == 0) {
    result[get_group_id(1)*get_num_groups(0)+get_group_id(0)] = scratch[0];
  }
}
```

Source 8 - The GetTextureMaxKernel source code. The Kernel takes as arguments, in order, the Texture from which we want to retrieve the maximum value, a temporary local memory buffer shared among Work-items, the initial search space dimension of each Work Item, the width of the Texture, the height of the Texture and finally the floating point buffer where to save the results.

The GetTextureMaxKernel, see Source 8, uses a simple parallel reduction approach to make the most of the GPU parallelization capabilities when searching the Texture Space for its maximum value. Bryan Catanzaro's white paper, Catanzaro (2010) describes the implementation in great detail and provides comparative results with other methodologies, showing the performance advantages of parallel reduction.

First, each work item searches it's assigned local space, a square of *localSearchSpaceSize* pixels in side, and saves the maximum value found to local memory (*scratch*), Step 1 and 2 in Figure 27. Then, in Step 3, the local memory space is divided by 2 rounding up on both dimensions giving us the offset vector (offsetX, offsetY). Then, only Work-items which indexes (IndexX, IndexY) fall in the NW quadrant, see Step 4, perform an additional search, comparing their own value (see dots in figure) with values in memory position derived from adding their Index vector with permutations of the Offset vector (the arrows leaving the dots). During Step 4 the maximum values found by each of the Work-items are saved in each's position. Next, Steps 5 and 6 are in fact a repetition of Steps 3 and 4, first the previous Offset vector is divided by two again (rounded up), delimiting the new search space and which Work-items do the search.

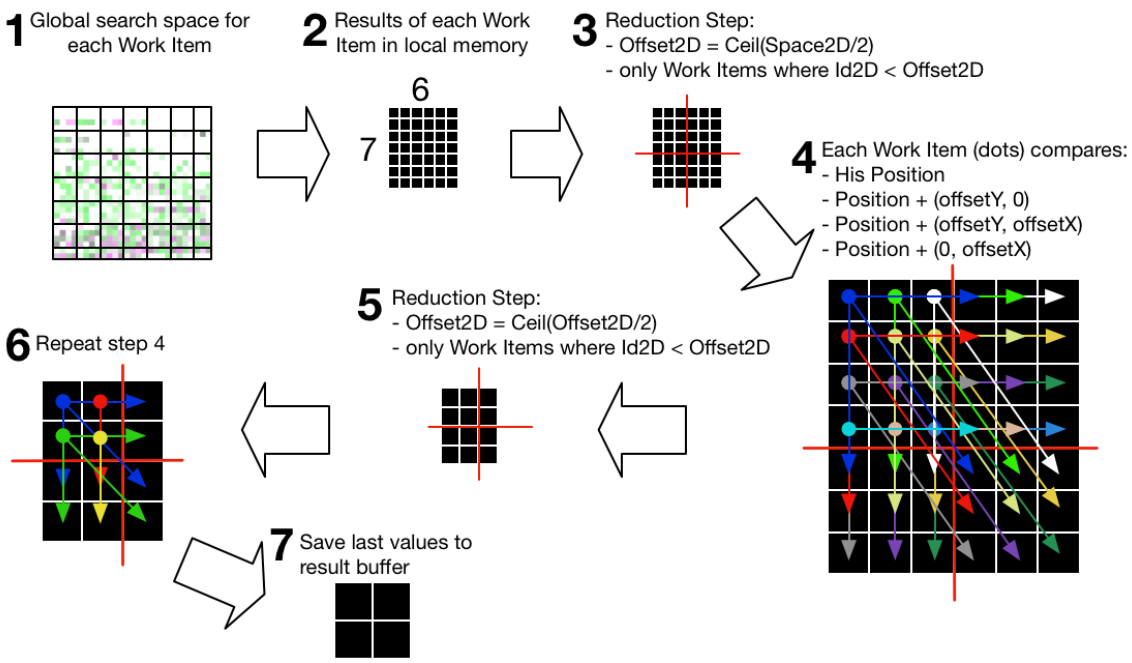

Figure 27 - Example of how parallel reduction works in the GetTextureMaxKernel. In this example the Black squares in Step 7 represent the final values returned by the Kernel execution that must be still iterated in the CPU to determine the actual maximum value.

This method of performing a maximum search is very effective, as Catanzaro (2010) work shows, yet, like he explain in his work, at each reduction step the SIMD occupancy for the next search is reduced by half, to a point where it becomes more effective, performance wise, to iterate through the final values using the CPU. The example in Figure 27 is just that, a figurative example of the algorithm working. In the real implementation we adjusted by hand the best number of reductions and consequently the size of the Results buffer to be iterated by the CPU, in order to achieve the best balance. Also, these adjustments had to be done independently for the Stores Heatmap Texture and for the Postal Codes Heatmap Texture, due to their differences in size, 27x26 for Stores and 346x345 for Postal Codes.

The calculated maximum value for each of Geographic Heatmap Textures will be used to perform normalization of each's values during the mapping of values to colors when rendering the textures to the actual Screen Buffer, as we will detail next.

5.5.1. Optimization of kernel execution

Regarding the optimization of kernel execution, we followed 2 different approaches, one for the AddKernel and another for all the reminding kernels.

Kernels like the DecayKernel, the PrintOutKernel and the GetTextureMaxKernel, all act over a space that is static in size over time, and for which we know the size from start. As such we are able to use a process similar to the one exemplified in chapter 4.2 to determine the best *global* and *local sizes* when enqueueing each of the kernels, as to make the best of the GPU processing capabilities. All of our kernels can only run after the previously has finished its work, as they operate on the same buffers, and as such, the process of determining the values was to query the API for the maximum permitted work group size

for each of the kernels and use the next multiple of that value, greater than the real global work dimensions, as our *global size* for the job and the maximum permitted work group size as our *local size*. Additionally, we fine tuned each kernel's number of operations performed per execution, controlled by variables inside each's kernel code.

Regarding the AddKernel, we also used the process described above to set the *global size*, yet, due to the nature of the data being processed, the number of items in each upload varies and, more importantly, sales registered on the same store and with the same client are usually in succession, as each of the Client's products are registered one after the other. Also, the number of products per sale (and in succession) varies constantly. In terms of the kernel operation, if two work items have sale records regarding the same store and the same Client, one will have to wait for the other to perform its operation in order to gain access to the memory block being used, this escalates when increasingly more work items need to access the same memory block.

To mitigate this problem we experimented with different numbers of operations per work-item (1, 3, 5, 10, 15), forcing the work-items to iterate over consecutive buffer elements. We hoped with this process to find a value more adequate than the others, yet our initial results showed an unquestionable increase when moving from 3 to 5, on a factor of 10x, but no increase what so ever, neither any advantage of either of the values 5, 10 and 15 over each other. The results we obtained greatly vary over the duration of the visualization, in a way unrelated to the number of uploaded sales (real global size), and varies also with data of different days, we believe the performance variance is related to the number of consecutive sales with same store and client, yet we did not explore this further as the performance obtained while using a value of 5 or above is enough to meet our needs.

We calculated time spent on kernel execution as the total time over the number of items processed. The results obtained by values 5 and above varied from 859 nano seconds per item (std=260) to a peak of 159 nano seconds per item (std=27). The peak attained varied a few nano seconds for each of the 3 best configurations. Theses averages were calculated from the time values obtained through the OpenCL API profiling tools, in batches of 200 consecutive uploads at different times of the day (animation time) and for data regarding different days. We collected the values at pre-programmed times in order to measure the execution over the exact same value sequences.

5.6. Coloring Geographic Heatmaps

So far, we only made a quick reference in Chapter 5.3 on how color mapping of the values from the Geographic Heatmap Texture to the Screen Buffer is done. At that point we had implemented linear color mapping from the normalized value of the Geographic Heatmap to a color mapping texture, presented in Figure 22 (left).

5.6.1. Color and perception

Linear color mapping is vastly used in heatmap visualizations, and with the right choice of colors its able to highlight the different values effectively, as the work of Ware (2012) describes. Yet, on animated "pixel" heatmaps, where items are discretized and consequently so are the boundaries of each "zone", this effectiveness is greater, when comparing with the results on animated heatmaps over continuous spaces (geographic heatmaps and similar) where fast variations create chaos and make the perceptibility of "growing" and "shrinking" areas more difficult. Researchers sometimes chose to discretize the color space, like we saw with Scheepens (2010), creating plateaus of values, similar to the plateaus between isolines.

We implemented a similar coloring methodology for multiple reasons. First, even if coloring by range reduces the precision in terms of small differences between values, it improves the visual identification of similar zones and their evolution with time. Second, as we have two Geographic Heatmaps that can be viewed one at a time or both at the same time, overlaid on top of each other. Blending the linear colored heatmaps would introduce a new level of visual confusion. Third, and lastly, we also implemented methods for overlaying demographic information, Chapters 5.7 and 5.8, with one or both heatmaps at the same time. Controlling the number and what colors are overlaid is the only way for us to choose correct colors for the demographic layer in order to facilitate the reading of information.

Unfortunately, when coloring fragment by fragment, as we must when rendering using the GPU pipeline, discretizing the color space introduces aliasing, refer to Figure 28, that even if it doesn't affect the visualization precision it does affect aesthetics, as Scheepens also shows.

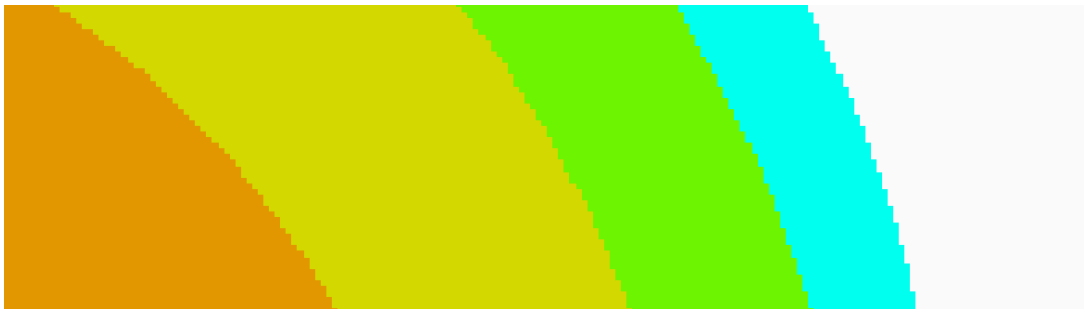

Figure 28 - Aliasing introduced by discretizing the colorspace.

We implemented a Shader Program, named GeoHeatmapColoring Program, composed of a Vertex and Fragment Shader, to render the values from the Geographic Heatmaps to the Screen Buffer by performing the correct color mapping. As we can see in Figure 29, our application renders information in layers onto the Screen Buffer, the GeoHeatmapColoring Program is used when rendering the two top layers.

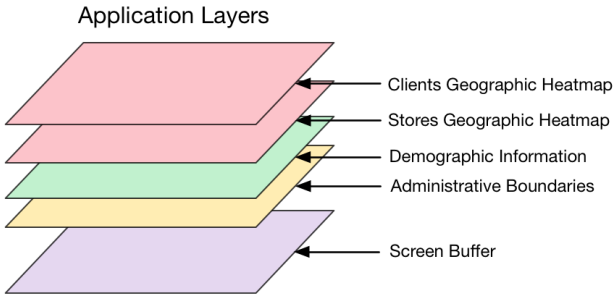

Figure 29 - Application rendering layers.

As we have two visualization modes for the geographic heatmaps, to show one or two heatmaps simultaneous, we implemented two different colors schemes also, one for each mode, refer to Figure 30. We used Uniform variables to instruct the Fragment Shader (FS) which color scheme to use.

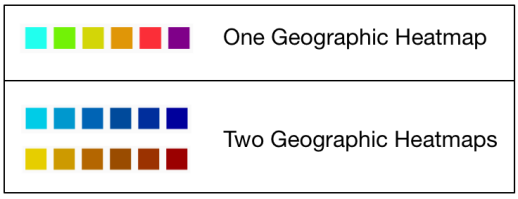

Figure 30 - Discrete color spaces used to color the Geographic Heatmaps when rendered to the Screen Buffer. The top is used when only one heatmap is rendered and the bottom is used when two heatmaps are rendered at the same time.

The colors we chose for each of the heatmaps, when displaying both heatmaps simultaneously, were not picked randomly. We used a free Web application Paletton[10] to construct a split-complementary color scheme, which is a variation of the complementary color scheme, with three neutral pastel colors, refer to Figure 31 We then picked two of the most complementary colors (on opposite sides of the chromatic diagram), for Stores and Clients, Yellow and Blue respectively, and implemented a way to create the third.

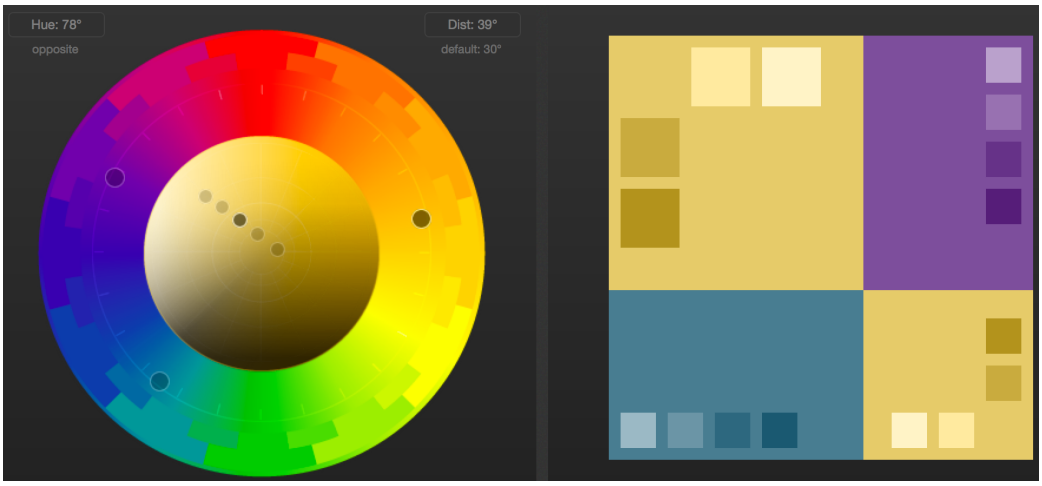

Figure 31 - The split-complementary color scheme we picked to color the heatmaps when visualized simultaneous.

[10] http://paletton.com

To create the third color of the split-complementary color scheme, we implemented a tailored GL_BLEND configuration, see Source 9, that produces the mathematical function that having the picked Blue and Yellow colors as Source and Destination, returns the correct third color in the color scheme. Additionally it maintains the same color alpha value to integrate seamlessly with the other visualization elements. Refer to Figure 36 for the results produced by this configuration.

Setup GL_BLEND

```
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_ONE_MINUS_SRC_COLOR, GL_ONE_MINUS_DST_COLOR, GL_ONE, GL_ONE);
glBlendEquationSeparate(GL_MAX, GL_MAX);
```

$$\begin{bmatrix} \max(sR, dR) \\ \max(sG, dG) \\ \max(sB, dB) \\ \max(sA, dA) \end{bmatrix} = \begin{bmatrix} rR \\ rG \\ rB \\ rA \end{bmatrix}$$

Source 9 - GL_BLEND configuration used when rendering both Geographic Heatmaps at the same time.

The FS we implemented does more than just picking colors for fragments depending on which range the heatmap values fall in, it introduces transparency into the layers enabling visibility of the information on bottom layers (Figure 29), and solves the aliasing introduced by the discretized color space as we will see next.

5.6.2. Anti-aliasing

To solve the aliasing problem we used a combination of techniques to devise a method that scans the surrounding space of each fragment in order to determine its proximity to the range's "edge", and mix colors of different "plateaus" accordingly. Additionally, as we could calculate fragment distance to edges, we could also draw a line on the limits of plateaus, allowing us to increase the inner plateau transparency without impacting the quick identification of heat zones, refer to Figure 32 for an actual render using the described approach.

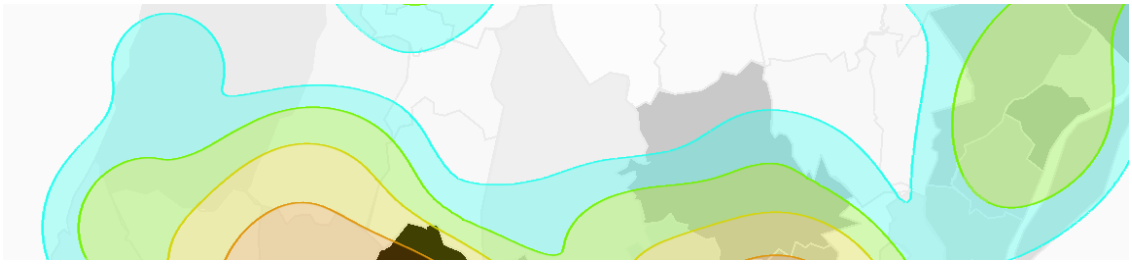

Figure 32 - Anti-aliasing of plateaus and a border line delimiting the different ranges.

To achieve this, that Figure 32 depicts, for each fragment we first determine the partial derivative of the geographic heatmap texture space in x and y of the Screen space, using

GLSL functions dFdx and dFdy. The resulting vector, lets us calculate the TextureCoordinates on the geographic heatmap texture space of the surrounding pixels. We scan a maximum of 12 pixel neighbors, both at 2px and 1px (screen space) in distance, refer to Figure 33 for an illustration of the process we are describing. First we scan the 4 furthest away pixels, 2px away from the current fragment in both x and y dimensions. If this first 4 scanned pixels are inside the same range of values as the current fragment, then we stop our scanning of surrounding pixels and color this fragment as completely inside the range of values (plateau). If instead any of the 4 scanned neighbors falls on a different range it means this fragment is near an edge, and as such we move on to scan the remaining 8 neighbors, stopping if at one point we have found already 2 points in a different range and 2 points in the same range of values as this fragment. If when we finish scanning the neighbor pixels we have found these 2 points inside the range and 2 points outside the range, we use them in pairs (Pair1(ouside1, inside1), Pair2(outside2, inside2)) to interpolate points where the line segments intersect the threshold edge, ending up with 2 different points that lay on the edge. When we only have one point outside of the range, we use that one in conjunction with one inside the range to interpolate a point on the edge, as before, and use 4 of the neighbors inside the range to perform extrapolations and average these extrapolations in order to determine a second point on the edge.

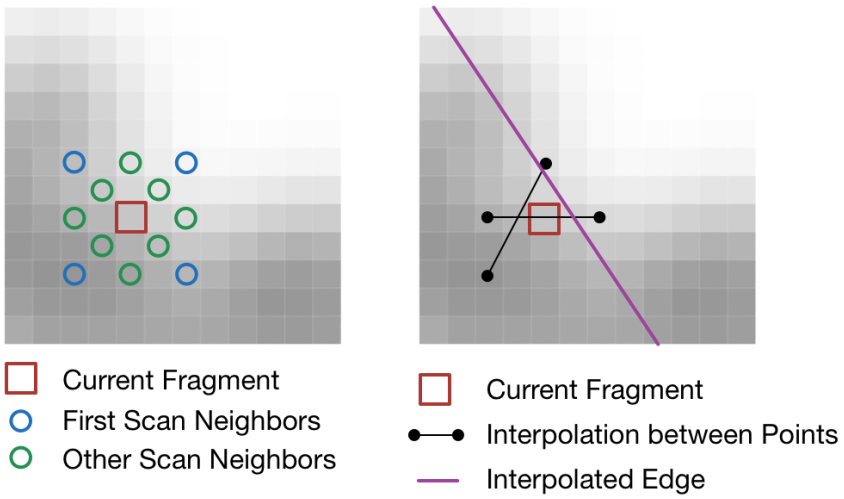

Figure 33 - Illustration of the Edge interpolation algorithm.

We then consider the edge as a line segment defined by the 2 calculated Points, and use that information to write the edge line equation in the vector form **x = a + tn**, where *x* is a 2D vector giving the two coordinate values of any arbitrary point on the line, *n* is a 2D unit vector in the direction of the line, *a* is a 2D vector giving two coordinate dimensions of a particular point on the line, and t is a scalar, as the work of Sunday (2001) shows. With the line in vector form we are able to calculate the Shortest Distance from the current fragment to the edge Line using Equation 8, where p is a 2D vector representing the Point position and the reminding variables are the same as in the description of the vector form above. In Figure 34 we present a small illustration of how the Equation works very helpful in understanding the formulation (source Wikipedia).

$$dist(x = a + t \cdot n, p) = \|(a\text{-}p)\text{-}dot(a\text{-}p, n) \cdot n\| \quad (8)$$

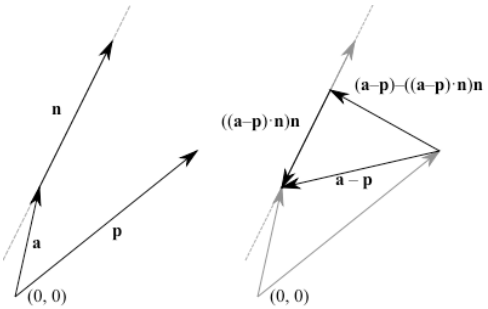

Figure 34 - Shortest Distance from point to line illustration (source Sunday (2001)).

Having determined if the current fragment is or not near an edge, and if it is, the distance to that edge, and also knowing that the distance is relative to screen space allows us to interpolate the correct color and the correct color's alpha, according to the plots in Figure 35, and according to the set Color scheme (Figure 30). We use GLSL functions *mix* and *smothstep* to perform this interpolation.

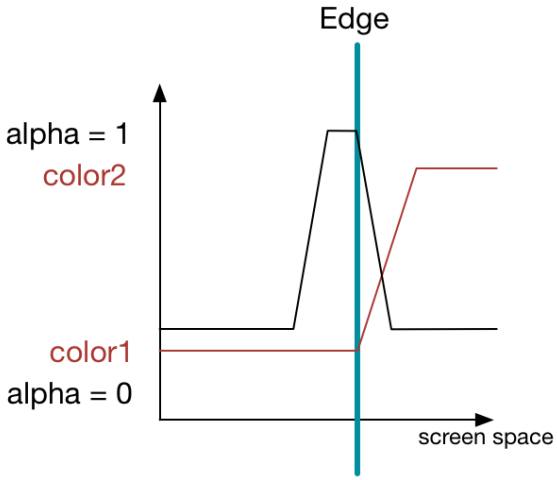

Figure 35 - Variation of color and the color's alpha value near the interpolated edge in order to render the anti-aliased line.

As the interpolated edge is recalculated for each fragment being colored, the assumption of the edge as a straight line does not reflect onto the final rendering, as we can see in Figure 36, where smooth curved lines appear.

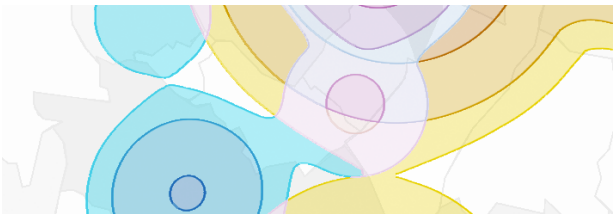

Figure 36 - Variation of color and the color's alpha value near the interpolated edge in order to render the anti-aliased line. Notice the purple color in the center, result of the blending configuration.

5.7. Displaying Administrative and Demographic Information

In Chapters 3.2.3 and 3.2.4 we talked about how important it is to contextualize the Geographic Heatmaps information within the actual locations being overlaid, and also described the data we collected of both Administrative boundaries and Demographics to achieve that. Here we briefly describe how we display that information to the User and enable him to identify Administrative areas.

5.7.1. Administrative Shapes and Boundaries

First, regarding the Administrative Boundaries, from the whole country to the civil parishes (freguesias), we use the collected contour points for each area to create polygons that we are able to Triangulate using the Triangulation feature of the OpenGL API that uses the Graphics Tessellator to perform the shape triangulation. With the shapes triangulated we are able to draw any of the Administrative areas with any Shader Program to either color or texture those areas, refer to Figure 37 for an example.

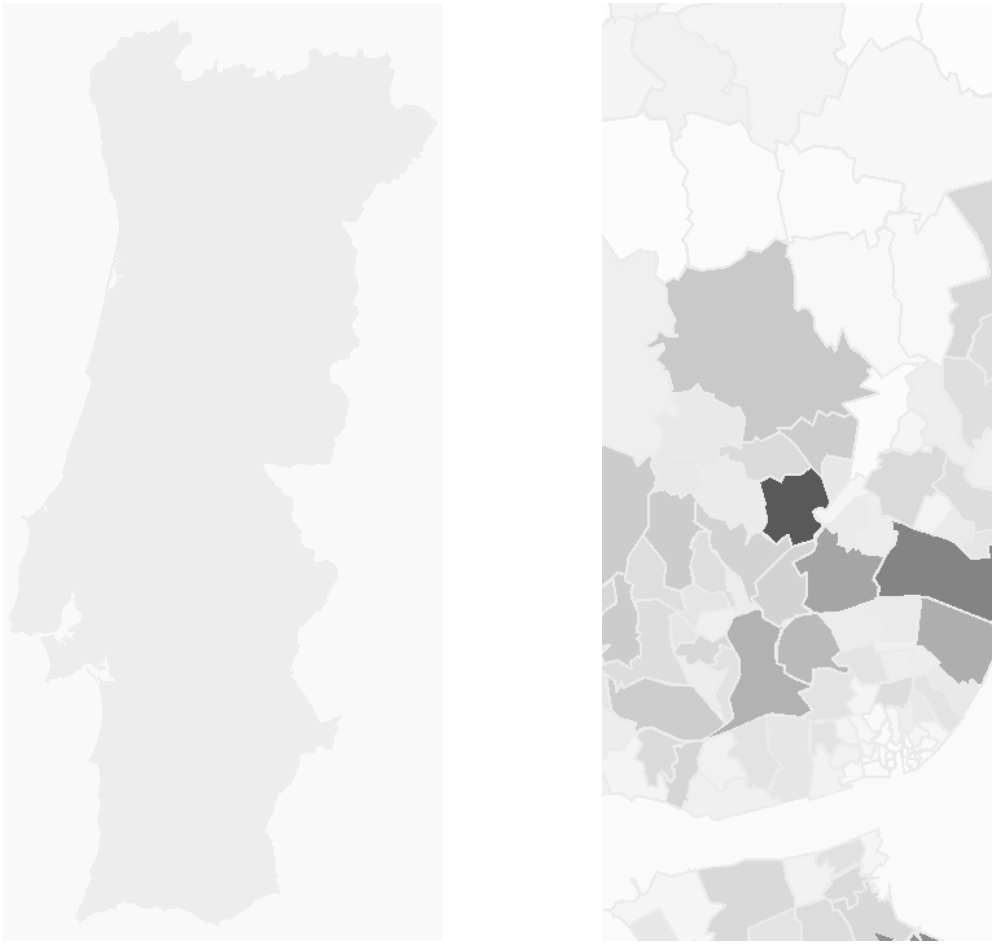

Figure 37 - On the left the Triangulated Country Shape rendered in grey color. On the right, many Civil Parishes (freguesias) on the Lisbon area rendered with different shades of grey denoting their different populations.

Additionally, still regarding the Civil Parishes shapes, we implemented a line drawing Shader Program, that we will call LineShader Program, composed of a Vertex, Geometry and

Fragment Shaders, that we use to render lines of specified color and thickness. We use this Shader Program to draw Administrative areas outlines, both to differentiate different areas but also to highlight areas on Mouse Over. The Vertices composing each contour are drawn with GL_LINE_STRIP_ADJACENCY draw mode, so that for each line segment to be generated by the Geometry Shader it receives 4 Vertices, the one that precedes the start of the line segment, the one that determines the start of the line segment, the one that determines the end of the line segment and the one comes after the end of the line segment. This is a usual approach to the generation of lines using the GS, providing the necessary information on each execution of the GS, to calculate the angles between each end of the of the line segments with other adjacent line segments and with that information create a seamless connection between segments on either ends. Refer to Figure 38 for a rough illustration of the process.

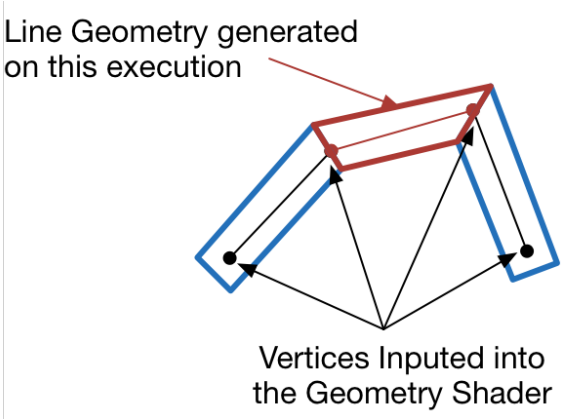

Figure 38 - The line generation process implemented through a Geometry Shader. Notice how the line segments caps are shaped to close the corners when connected to another line segment, this is only possible because the Geometry Shader has the information regarding the adjacent segments.

5.7.2. Mouse Over Location Identification

We described in Chapter 5.2.3 how we perform un-projection from mouse coordinates in screen space to a Ray in World Space. We then use the calculated Ray do determine if the Mouse Position hovers over any significant object in the Scene, in other words we test the Ray for collisions with the geometry being displayed (rendered) to the Screen.

To perform collision detection with every Triangle being displayed is extremely expensive computational wise, as we render hundreds of thousands of triangles at each Render cycle. That sort of collision detection would have to be performed in the GPU, and while several techniques exist for that approach, all of them involve considerable computational resources of both the GPU to perform the collision detection and the CPU while reading results from the GPU.

Another approach, which we used and is also widely used on performance critical applications (Video Games), consists in calculating, at loading time, Bounding Boxes (6 rectangles) for 3D objects and Bounding Rectangles facing the camera for 2D shapes, and calculate the collision detection of the Ray against the Object's delimiting rectangles.

Our implementation only contains 2D shapes, and as such each Object has a corresponding Bounding Rectangle, additionally every group of Objects, the Civil Parishes Shapes for instance, have the same z value (height). So we first calculate the collision point of the Ray with the plane where the shapes reside, obtaining a 2D coordinate, and then test the Bounding Rectangles to determine whether they contain the Collision Point or not. Additionally, to avoid having to test every single Bounding Rectangle (hundreds in the case of Civil Parishes) for Collision, we implemented a Quadtree of Bounding Rectangles to subdivide the World Space into smaller chunks at each Quadtree sub-level. The Quadtree, when queried with a Position, will test only the Bounding Rectangles contained in chunks of space which contain that given Position. With this approach we are able to perform Collision detection entirely on the CPU with residual impact on the global performance.

The Mouse Over functionality is used both for the interaction with the User Interface and the identification of Civil Parishes when hovering over the Country's map, displaying both the Civil Parish Name and the highlight of its boundaries for easier identification.

In our implementation we opted to only display the locations Name on Mouse Over to avoid obstructing possibly important information on the heatmap(s).

5.7.3. Displaying Demographic Data

In Chapter 5.7.1 we described how we triangulate the administrative zones' shapes so we can render their geometry with any applicable Shader Program. (Applicable because it would make no sense to render these triangles, properly ordered to render as GL_TRIANGLES, with the LineShader Program also described in 5.7.1.) The geometry information for each Civil Parish is kept in same Video Memory buffer, but we keep pointers to each of the shapes position in that buffer so we can both draw them individually or all in one call. This approach, regarding Video Memory buffers, was used whenever there were many related objects which together, total thousands of Vertices, the Civil Parishes Contours is another example.

To display the Demographic data using the Civil Parishes and the Country geometry we implemented 3 different Shaders Programs, one for the Country's geometry, named StarsShader, that while with some inaccuracy was an interesting visual and aesthetic experiment, and two for the Civil Parishes geometry, that consists in our approach for a more generic model able to represent any Demographic variable. The difference between these two lies in the coloring method, a comparison experiment, and we named them GreyShader and ChoroplethShader.

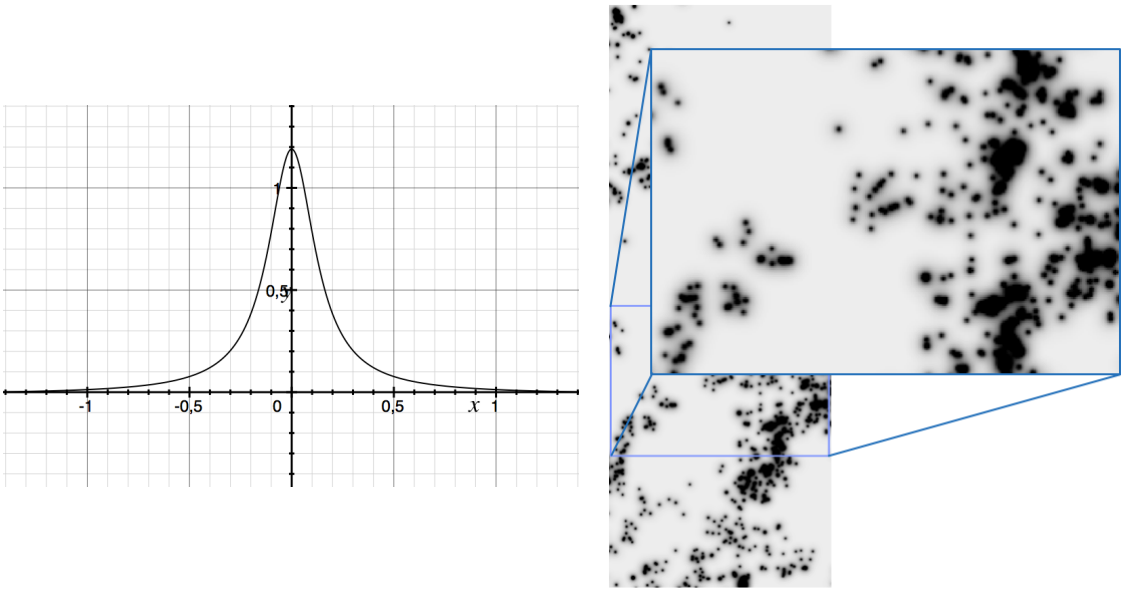

Figure 39 - On the left a plot of the mathematical equation used to color the Quads in the Fragment Shader of the StarsShader Program. On the right the rendering result using that equation.

The StarsShader, uses the data relative to Building locations extracted from OpenStreetMaps and the Postal Code locations, described in 3.2.4, as representative approximation of the Country's Population and draws them with a slight distortion and halo effect similar to the visual effect produced when looking at a star in the night sky. Experiments of rendering locations as dots produced aesthetic un-appealing results. To achieve this the StarShader is similar to the GeoGauss Program in that the Vertices are drawn as GL_POINTS and a geometry shader generates two triangles, composing a Quad, that are rasterized and then colored in the Fragment Shader (FS). The FS in the StarsShader uses the distance to the center of the TextureCoordinates Space of the Quad to calculate the appropriate color and transparency for each fragment according to the equation plotted in Figure 39 (left). On the right, also in Figure 39, we present a zoomed in screenshot of the visual effect produced. Notice how particularly the halo is noticeable in both levels of zoom.

The GreyShader and the ChoroplethShader are both used to represent the same information by coloring each Civil Parish (freguesia) geometry differently, according to each Civil Parish "score" in a determined Demographic variable at a time. In our experiments we used both Population per Civil Parish and Population's Purchase Power per Civil Parish. The difference between the two Shader Programs is that one, GreyShader, colors the Civil Parish geometry with shades of grey, and the other, ChoroplethShader, uses a procedural texture generation algorithm to paint the geometry with orthogonal lines with varying distances as described in the work of Tobler (1973) on choropleth maps. With smaller distances the generated textures appear darker and, with larger distances, brighter.

The GreyShader implementation is straight forward, we use Uniforms (also in the ChoroplethShader) to configure the shader to render the correct darker or brighter tone. The VS transforms the vertices as usual using the MVP and the FS colors the fragments with the correct shade of grey. The only particularity of the implementation is the mapping of the normalized values to represent (0 to 1) not linearly in the white to black range, but

logarithmic to better fit the human perception of the difference between of shades which if logarithmic also, see Tobler (2973). Refer to Figure 40 (left) for an example render using this shader.

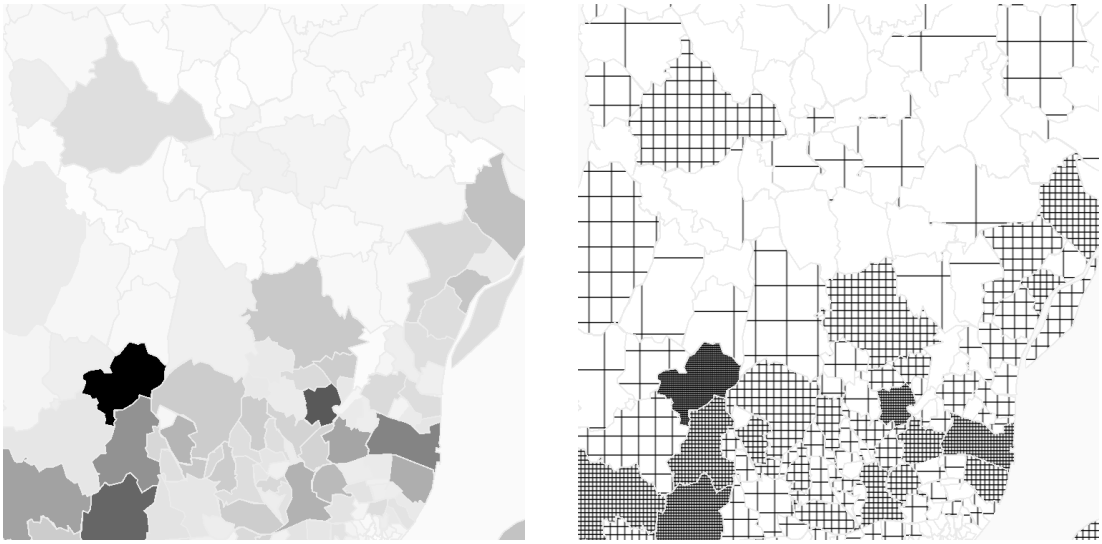

Figure 40 - Both left and right contain the same exact area representing the same exact values. On the left the GreyShader was used and on the right the ChoroplethShader.

The procedural generation of the orthogonal lines in the ChoroplethShader FS uses a similar implementation to those of procedural generation of anti-aliased brick patterns, see Ebert (2003), without the offset every odd row and with bricks having a square shape. We use the mortar size to reduce the distance between lines, and consequently produce a darker texture as needed. Figure 40 right shows an example of the effect achieved using this texturing procedure.

In Figure 40 we can also notice the usage of the line generation method described in 5.7.1. The thickness of these lines varies when the User zooms the visualization in and out, in order to maintain the visibility of Civil Parishes Boundaries without obstructing the color of texturing applied. The drawing of Civil Parishes Boundaries also serves the double purpose of covering the visible seams introduced by the optimization of the shapes done outside of the application.

Another important implementation step taken to improve the application's performance, derived from the fact that the Administrative and Demographic information is static in time, was to, instead of drawing all the Boundaries and Administrative information directly to the Screen Buffer every frame, we do a pre-render into a Texture as FBO, that is as large as the Screen Space, and then draw this Texture onto the Screen Buffer. We still have to draw the Texture onto the Screen Buffer at every frame, but the pre-render into the Texture, which actually needs to render all the described geometries, is only rendered whenever the Camera is panned or the Zoom factor changes, drastically improving performance.

5.8. Miscellaneous Features

In these Miscellaneous features we include not all, but the most relevant "small" features which help make the whole visualization application more complete. By referencing them as small of miscellaneous we by no means intend to diminish their importance.

5.8.1. Filtering Stores

Giving the User the ability to filter Sales records from specific Stores in or out of the visualization is a critical feature to support the analysis of the data. We further detail and provide a practical example showing the importance of this feature in Chapter 6.2.2, regarding Results.

To implement this feature we use an Hashmap that gets populated as the User selects different Stores using the User Interface. The Hashmap starts empty, instead of full, which is equivalent to the SHOW_ALL state. The Hashmap contents are checked in the CPU, and the actual filtering occurs as records are read from the data stream, Sale records are only accepted and read if either the Hashmap is empty or the Store Id is contained in the Hashmap.

The reasoning behind this approach is that when the User wants to analyze a group of Stores, it usually consists of a small group or even only one Store, keeping the Hashmap with few items and consequently fast search times. The User is still allowed to filter in and out as many Stores as he needs, and this won't have a great impact on the search times due to the number os Stores, but we sought to improve the overall performance wherever we can.

5.8.2. Screen Capture and Video Recording

Other important features in any application which's purpose is to facilitate exploratory data analysis through information visualization, is the ability to save the commonly called Screen Shots, by capturing the contents of the application's window and write them to disk as Images, and also, the ability to record Movie Clips, capturing heatmap animation segments to be shared, presented or further analyzed without the necessity of rerunning the application.

To implement these features we use the OpenGL API function *glReadPixels* to copy the contents of the Screen Buffer in Video Memory to a floating point buffer in CPU Memory. We then decode the color components for each pixel and write them to a PNG file on disk. Performing this process significantly impacts performance, this is particularly noticeable when recording a movie clip where we perform the memory copy every frame.

With the purpose of minimizing the performance impact and produce a good quality and high frame rate video we made alterations to how time is tracked. Usually animations and animation elapsed time is calculated using the time elapsed since last frame, usually called deltaTime. For instance, by multiplying the deltaTime by the animations minutes per second rate we know how much time the animation must advance at every frame. When the User

starts Video Recording, we change to another usual method of handling elapsed time in similar applications, a fixed deltaTime, more precisely the application assumes at every frame that the elapsed time since the last frame was 1/30 of a second.

The disadvantage of using a fixed deltaTime, in our application, is that as the application doesn't run at constant 30 frames per second, the animation will be slower than usual as the User views it in real time when the performance goes below the 30 fps, and faster than usual when performance is above the 30 fps. And while we can solve the speeding up of the animation by switching from fixed deltaTime to the real deltaTime whenever the application is "ahead" and skipping saving to disk more frames than the specific frame every 1/30 of a second.

The main advantages of using the fixed deltaTime are the reproducibility of the same animations in any computer giving the same dataset, and the assurance of a completely fluid, 30 fps video with the correct minutes per second rate of animation.

The images used in this document were created using these features. Additionally every video we recorded to present alongside this document were also created using these features, but here with a small remark. In our recorded videos the User interface is visible, as we want to highlight the application it self and not the data being visualized. In the final version of the application, to be used as an analysis tool, the interface's menu bar and its children are never visible, as we render them to the Screen Buffer only after its contents have been copied to CPU memory.

5.8.3. User Interface

Noticeable through out Chapter 6, in the screen shots that display the complete application window, is the menu bar at the top. We chose to implement a simple menu bar inside the application window in an effort to avoid, as much as possible, the introduction of obstacles that for being Operating System (OS) specific, difficult the portability of the application between different platforms and operating systems.

Our implementation of the menu bar behaves much the same way as a usual menu bar, with mouse over interaction. And although it doesn't provide the same flexibility as the Mac OS or Windows OS menu bars, it enables us to easily create and link action listeners for clicking and toggling events.

6. Results

In this Chapter we present the results obtained with the implemented features. Every performance results presented in this Chapter were obtained using a MacBook Pro Later 2013. We chose this hardware as our test base for being in the lower range of the currently available GPUs. It's most relevant specs are the 2.4GHz dual-core Intel Core i5 processor, 8GB of 1600MHz DDR3L onboard memory, Intel Iris 5100 integrated GPU and 256Gb SSD drive.

Images presented in this chapter were obtained with our screen capture feature implemented within the application. We save the image data to the disk in PNG format with the best quality possible, yet, when visualizing the images produced, we feel that they still don't transmit the actual visual quality obtained when visualizing the actual application running and being displayed on a high pixel density screen, or retina.

We start off by analyzing our implemented solution for the aliasing problem introduced during the coloring of the Geographic Heatmap. Later we test the implemented features as a whole, by using them in performing a small exploratory analysis and also visualize the impact of a specific event, the opening of a new Store. Finally we show the results obtained by switching to our second tracked variable, which displays the Total Value of Sales over the Total Quantity of Products, described in Chapter 5.5.

6.1. Anti-aliasing and Geographic Heatmap Coloring

In Chapter 5.6 we described how we color the heatmaps using a discrete color map, and how we implemented an anti-aliasing technique in order to remove the aliasing introduced by that color mapping approach.

In Figure 41, we present two magnified images of similar lines, so we can identify individual pixels, and we can clearly observe the difference between an aliased and an anti-aliased line. In the anti-aliased line notice how the color of the line performs a harder transition when the line is straighter and a softer transition when the line starts to curve, as to produce a visually smoother line.

Figure 41 - Aliased line on the left, magnified. On the right an antialiased line obtained with our implemented algorithm, also magnified.

To help validate our anti-aliasing solution, we used Adobe Photoshop to draw anti-aliased lines, similar in shape to a group of examples we picked for comparison. To use Photoshop to draw the anti-aliased lines, we first constructed a path using the Pen tool, and then we Stroke the path (raster) using a smooth 3px wide Brush.

Figures 42, 43, 44 and 45 present the result obtained, putting side by side curves rendered by our application (colored backgrounds) and smooth lines rasterized by Photoshop. We present the images magnified so we can compare the how the blending of colors is performed per pixel, as its distance to the "middle" of the line increases.

As we can observe by the results our algorithm performed as expected, creating similar color blending to smooth the line as the ones from Photoshop. In our opinion, the pairs in Figures 42, 43 and 44 are particularly similar and is only on the pair in Figure 45, the bottom part of the line segment, that we find some discrepancy.

Figure 42 - Comparison between vertical curves. On the left a smooth line rendered by our application, and on the right a smooth line rasterized with Adobe Photoshop.

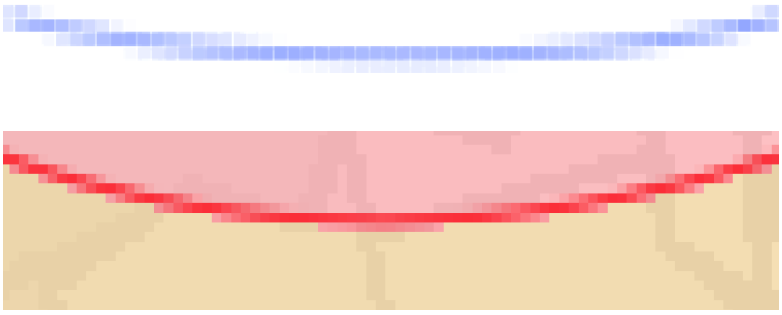

Figure 43 - Comparison between smooth horizontal curves. On the bottom a smooth line rendered by our application, and on top a smooth line rasterized with Adobe Photoshop.

The lines are continuous and form closed shapes. Any of the small discrepancies, like the one in Figure 45, are too small to be noticeable and are minimized by the surrounding pixels correct blending and by constant changing of the shapes with time. We consider them small discrepancies when considering that the lines rendered by our application are the result of a combination of the per pixel blending of colors, which is based the pixel's distance to an also per pixel estimated straight line segment.

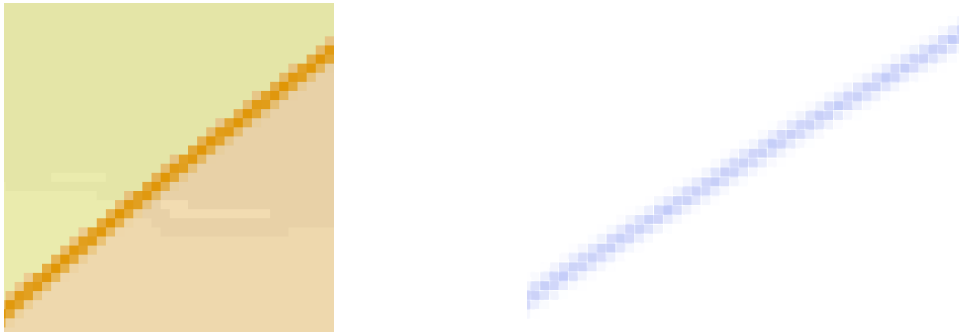

Figure 44 - Comparison between almost straight curves. On the left a smooth line rendered by our application, and on the right a smooth line rasterized with Adobe Photoshop.

Figure 45 - Comparison between tight curves. On the left a smooth line rendered by our application, and on the right a smooth line rasterized with Adobe Photoshop.

In Figure 46 we present an unaltered screen capture performed with our application where we can see the transparent plateaus and the smooth anti-aliased lines resultant of our heatmap coloring shader. In this example we have enabled only the Clients Geographic Heatmap, thus the presented colors for the plateaus. Notice how the lines help define the plateaus and the geographic areas they span over. Also, particularly in the bigger red area, notice how we can relate the higher value areas of the heatmap with the high population in the Civil Parishes "underneath", and how the curved red area almost seems to define a contour around those Civil Parishes. The identification of these relations is possible only due to the balance of the color transparency to achieve the correct blending, in conjunction with the plateaus borderline with helps the recognition of shapes.

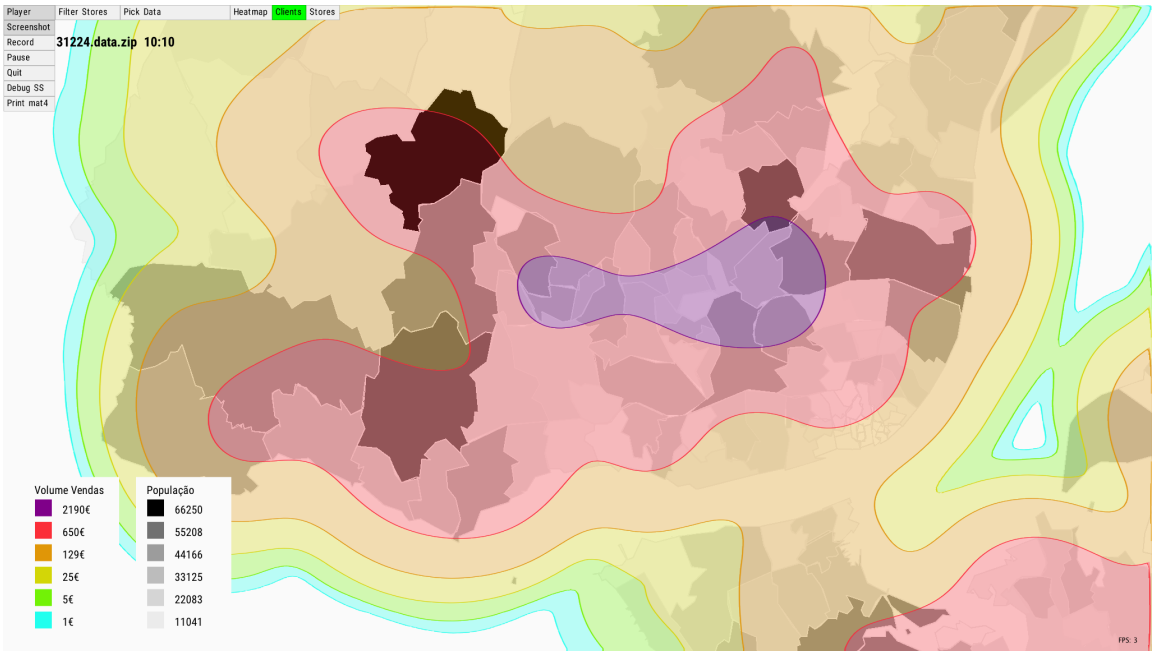

Figure 46 - Client Geographic Heatmap drawn with plateau transparency and anti-aliased lines. The shaded shapes seen through the heatmap are the Civil Parishes colored according to their populations (not normalized).

Concerning the visualization of both heatmaps simultaneously, in Chapter 5.6.1 we described how we picked a split-complementary color scheme, handpicked, and devised a GL_BLEND configuration that when the two colors, chosen to color the Clients and Stores heatmaps, would overlap, the blending process would return the third color from that same color scheme. By observing Figure 47, and Figure 31 from Chapter 5.6.1 to freshen the memory, we see that the GL_BLEND configuration worked as expected. Also, the strong visual contrast typical of split-complementary color schemes is also noticeable. We can precisely distinguish the Client heatmap, the Stores heatmap and the overlapping areas.

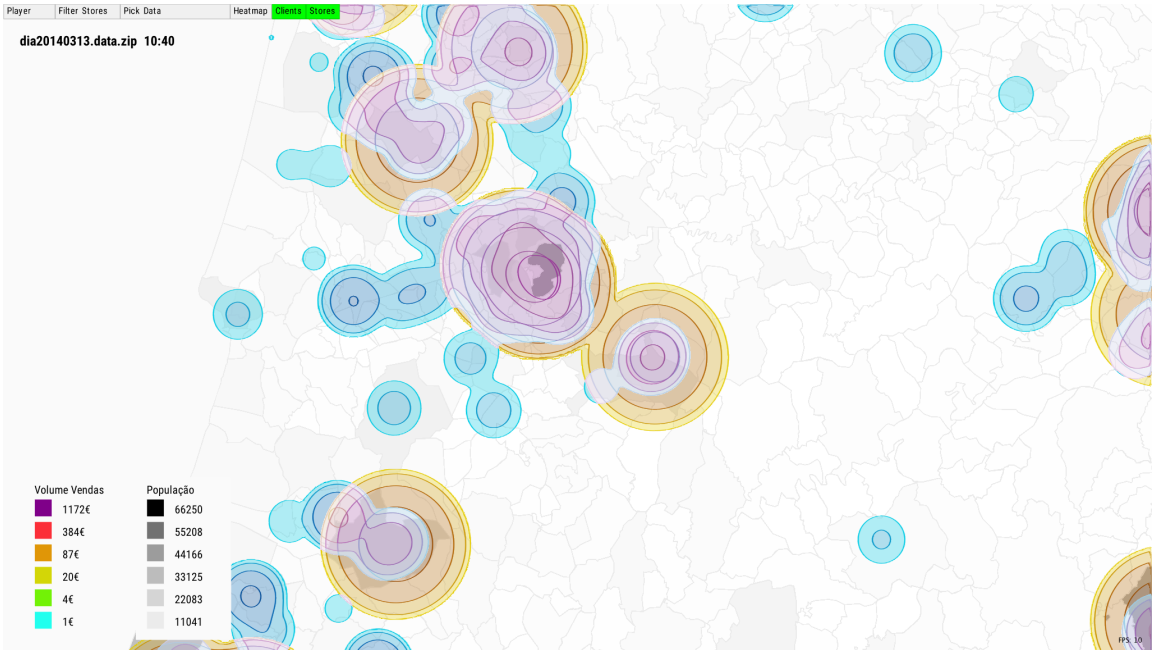

Figure 47 - Both Stores and Clients Geographic Heatmaps overlaid.

6.1.1. Performance Considerations

Another important aspect we have to take into consideration regarding this rendering approach is its impact on performance. To test this we ran visualizations with fixed deltaTime in our update and of the same set of Sales records, this way guaranteeing the reproducibility of exactly the same animation. During each of the visualizations we changed the Shader performing the coloring of the geographic heatmaps, and measured the time spent per frame during a span of 2000 frames and calculated the average number of frames per second.

As we can see in Table 1, with the Just Scan method being the one where there is no anti-aliasing or even discretization of the color map, the performance impact on the average frame rate, with our solution, is of approximately 1 frame per second.

	Just Scan		**Anti-aliasing + Plateau Contours**	
	Average	**STD**	**Average**	**STD**
One Heatmap	18.162033	3.608996	17.277062	3.733841
Two Heatmaps	13.079525	2.782339	12.869658	2.887267

Table 1 - Measurement of the impact our geographic heatmap coloring and anti-aliasing algorithms, had on performance. Values are in Frames Per Second, the bigger the better.

6.2. Identification and Confirmation of Events

Although performing the actual visual exploration of the dataset is not the focus of this Thesis, as we are developing an application that Users will use to perform that data exploration, it is important to assure that the implemented features that we expect will help in that task, effectively do so.

To test this we performed visual exploration of a handful of days, using the our application, with the objective of identifying previously unknown relevant information and also to visualize the effects of a known event.

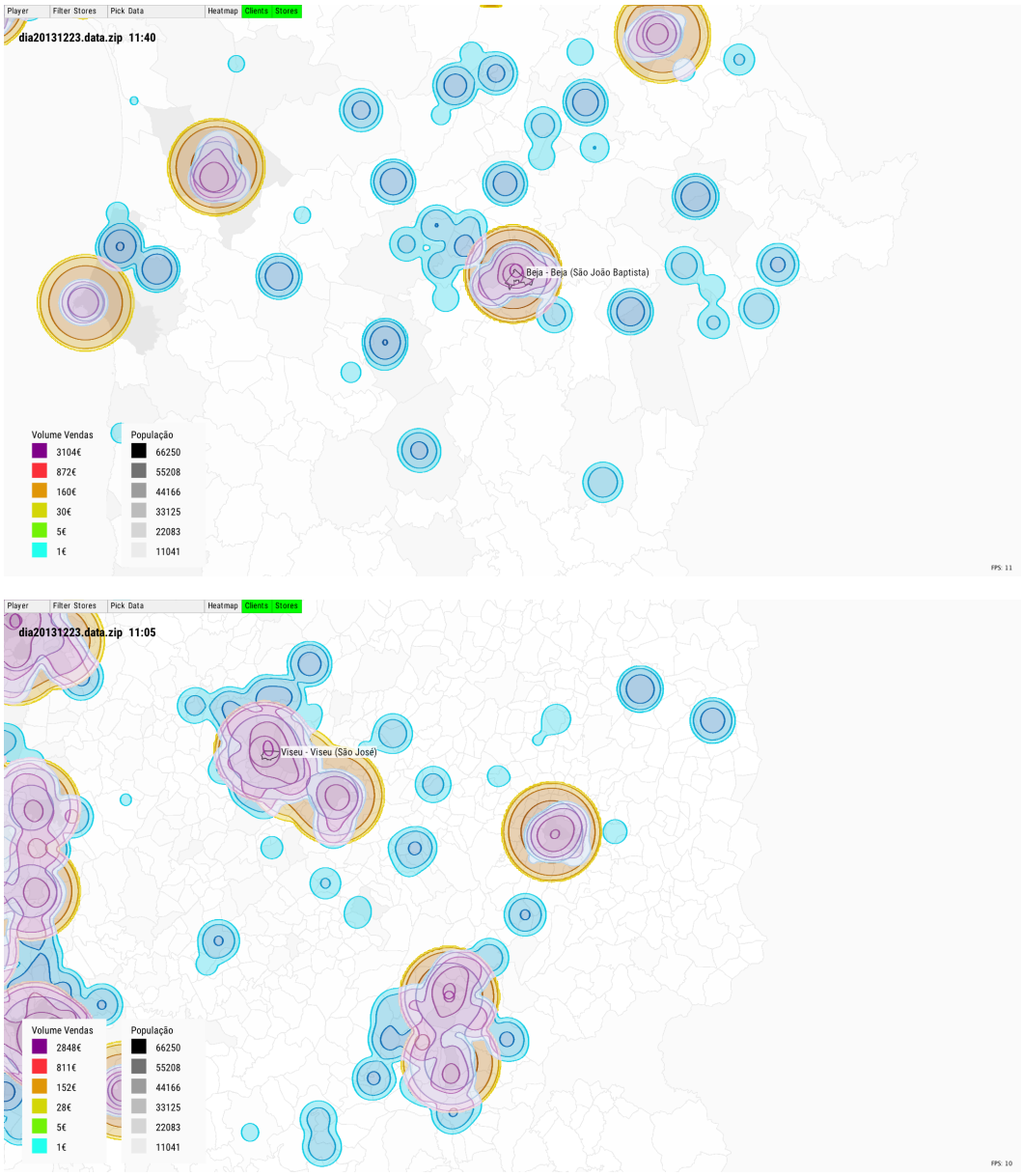

Figure 48 - Examples of screen captures depicting Clients who have to travel far to purchase their products. On top an overview of the Southern area of Portugal. Below is the Geographic area around the City Viseu.

6.2.1. Distant Clients

During our exploratory analysis, even if short, we noticed a particular phenomena that occurs every day in the least populated areas of the Country. We called this phenomena the Distant Clients, as it consists in population of isolated areas that by having no other option or by choice, travel considerable distances to purchase their products.

By visualizing both Stores and Clients heatmaps at the same time, refer to Figure 48, we can clearly identify patches of Clients (in blue) without a Store nearby (yellow). Particularly in the top image we can see how the Store "bubble" near the center (City Beja), overlaps patches of the Client Heatmap (in purple) that are of sizes similar the to the ones found in the isolated areas.

Having the ability to view both heatmaps at the same time and have the areas where they overlap highlighted made the phenomena "stand out", supporting its identification.

6.2.2. Store Opening

As mentioned before, we also tested our application by using it to visualize the effects of a known event, more precisely we chose the before and after the opening of a new Store.

In Figure 49 we can see that this southern least populated area of the country has few Stores, yet we can see that are Clients still make the effort of traveling to purchase in the available Stores. Particularly interesting also is that if we look closely at the population shades "below" the heatmap for the whole area, we notice that the zone where the new Store is opening is the darkest (more population) of all the zones that don't have a Store already. Pointing us to one of the probable reasons of why that location was chosen. And showing how more similar locations can be identified by using the same process.

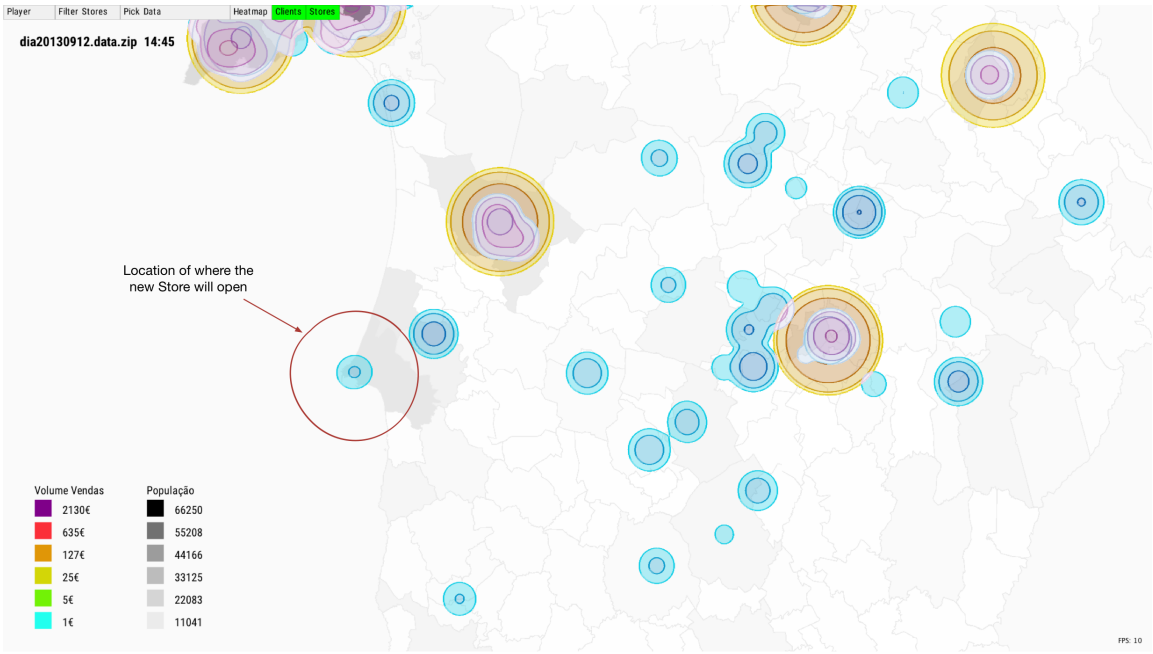

Figure 49 - Picture displaying Sales distribution near the days peak, the day before a new Store opens in the center of the red circle.

When the Store opens, Figure 50, we can notice the appearance of additional client patches, near the new Store's location, indicating that Clients might have shifted from stores of other competing Brands to this new Store.

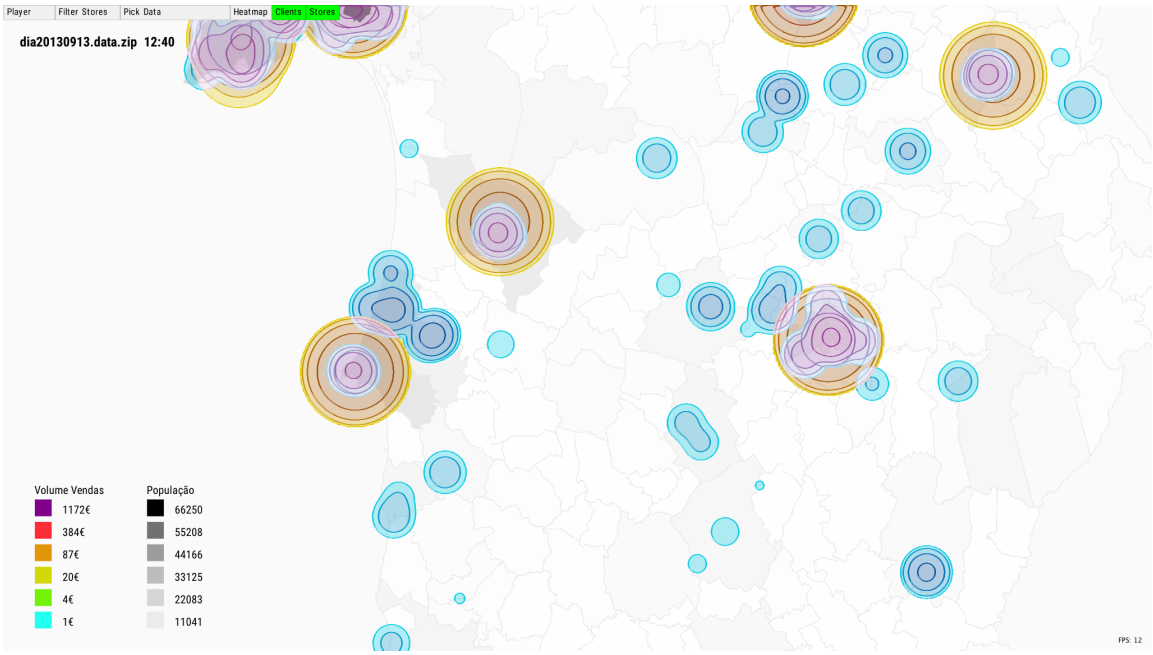

Figure 50 - Picture displaying Sales distribution near the days peak, the day a new Store opened (middle left yellow bubble).

Using just an overview of all the Sales is not enough to visualize the impact of the new Store on the opening day. Isolating heatmaps and filtering Stores helps to view the real affected areas.

In Figure 51 for instance, by visualizing only the Stores heatmap both the day before and the day of the opening of the new Store, we can clearly view how the new Store affected the balance between Stores in the area. The new Store takes a significant cut of the Sales at that moment in time, lowering the overall sales in those stores. Notice that although it seems that the lower image presents higher values, the value for each color as we can see in the legend are quite different. If we carefully analyze the legend we notice that values in both days are of the same magnitude approximately.

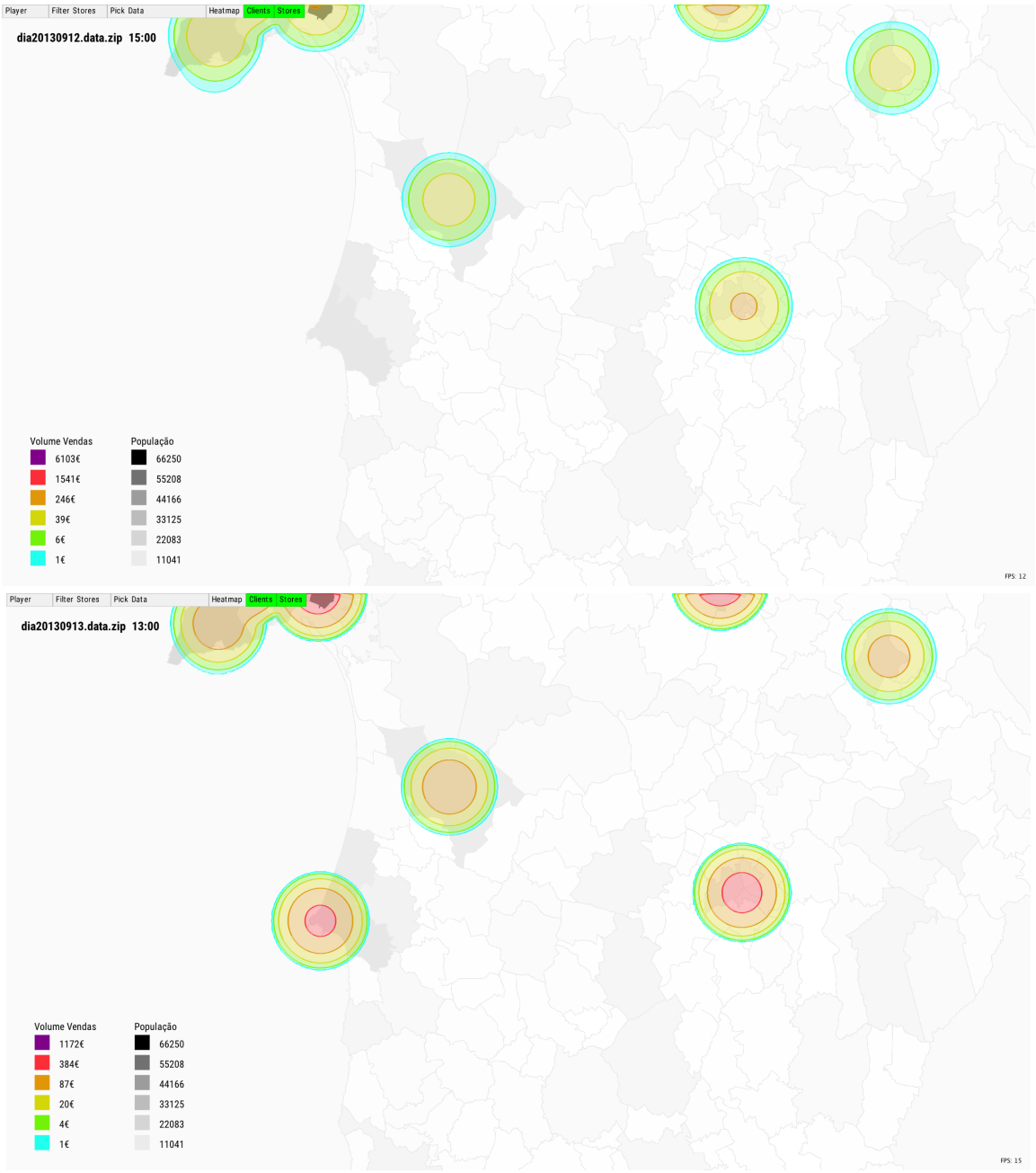

Figure 51 - Stores heatmap the day before (top) and the opening day of the new Store (bottom).

This can be confirmed additionally by observing both images in Figure 52, both the more local (zoomed in) area around the Store (on the left) and the overview of the southern half of the country. We used our filtering feature to visualize only information relative to Sales occurring in the new Store, and by displaying only the Clients heatmap we can identify exactly where the people buying at the Store live. And as we can observe, most of the Sales are to Clients who reside nearby, which makes sense as the Store is intended to serve that local area. Yet we can also identify Clients from more distant location who where at or traveled to that location on that day.

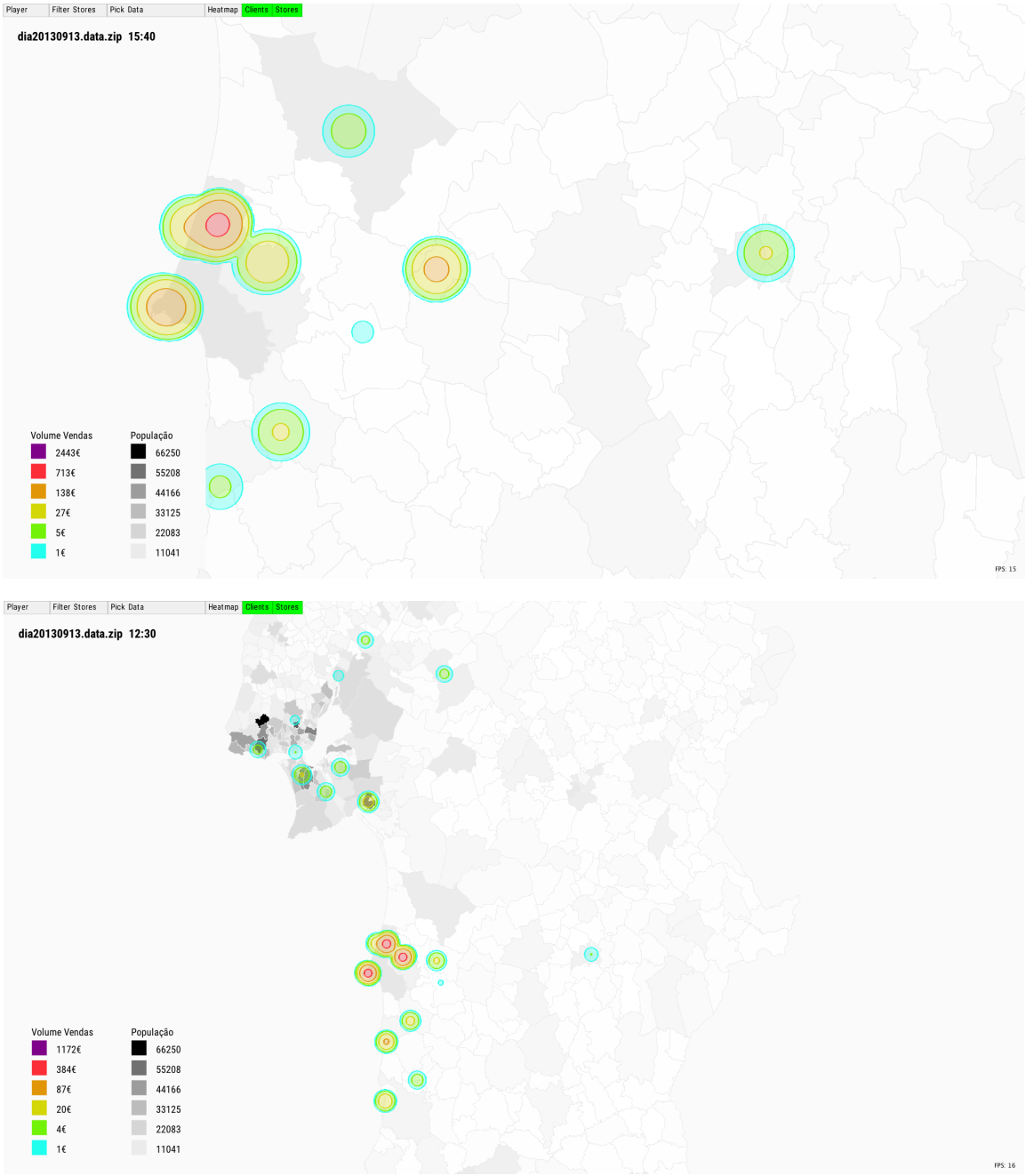

Figure 52 - Clients heatmap resultant of filtering out all other Stores except for the new one. On the bottom an overview of the whole south half of the Country, and on top a more local visualization, spanning the country's width.

6.3. Demographic Visualization Modes

As described in Chapter 5.7.3 we implemented more than one way of displaying the Demographic information. One of those implementations, StarsShader, is more of an aesthetic experiment with the purpose of approximating the distribution of the real population over the Country. And the other implementation, is a more generic and accurate approach of displaying any demographic variable as choropleth maps either greyscale or procedural textures.

We will first analyze the StarsShader results in terms of its accuracy to represent the most significant higher density populated areas, and also how aesthetics match our expectations. Later we compare the results of both methods used for coloring the choropleth maps, and finally analyze the performance impact of each of the Demographic representations.

6.3.1. StarsShader

The extra step of rendering the points as Quads so we can make them appear "fuzzy", also randomly varying the generated Quads sizes, produces a more visual appealing image than simply drawing them as black dots (disks). Yet, as visual appealing as it might be, it is crucial to be used as a reliable tool, that the visual artifacts produced represent a good approximation of the population for each geographic area, particularly in cities and towns.

We already knew, from initial experiments, that major cities are identifiable, as they tend to have the larges concentration of building and also more Postal Codes per Square Km. To analyze areas outside the major cites we used the choropleth maps generated for the Demographic variable Population, which we know to have accurate data, and overlaid it over the StarsShader results to see if we find correlation between the Points in it and the shades of grey of the choropleth map.

We performed this analysis for the whole Country but only present here the most relevant results from the Porto and Lisbon areas.

First, in Figure 53, we can observer that in the Porto metropolitan area, even if the correlation from point density and the greyscale is not perfect, is a close approximation of the areas where Population is most concentrated. Also, notice the three smaller green circles on top, that highlight areas with different shades of grey denoting different population and also a similar variation in the number of points and their density. On the negative side, we can see that all the red circles denote areas where the number and density of Points is too great when compared with the shades of grey of the same areas, and also, within the yellow circles we can observe that the shade of grey is darker than in the surrounding areas but still, the number and density of Points seem exaggerated for those area's Populations.

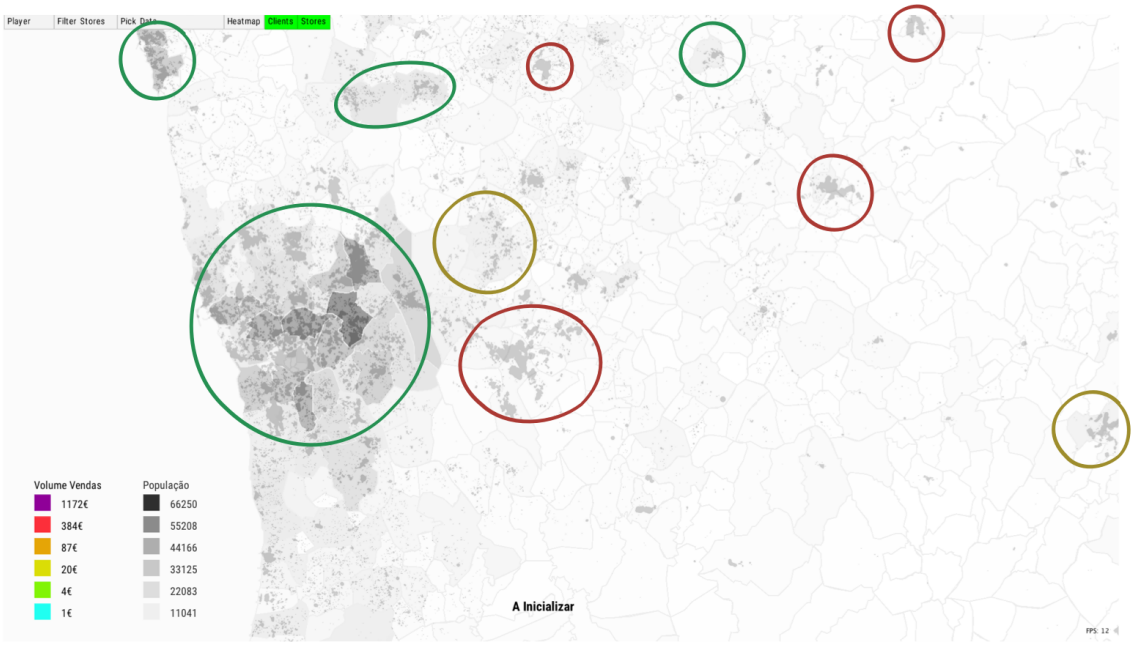

Figure 53 - Porto City area map with an overlay of the StarsShader result map, with increased contrast, brightness and transparency, over the choropleth Population map colored with shades of grey. The Green, Yellow and Red circles represent Good, Median and Bad correlation points. Only the most relevant were highlighted.

In Figure 54 we present the same type of analysis but this time for the Lisbon area. The three leftmost green circles, as in Porto area, represent the greater metropolitan areas, and as such present good results, as we already expected. But, also on the positive side, the areas within the reminding green circles presents small accumulation of dots in number and sizes that seem to correctly relate with the corresponding shades of grey. One could even ask if those points are not more accurate representations of where within the Civil Parishes (freguesias) those people actually live, against the choropleth representation that shades the whole Civil Parish shape with the same shade of grey.

Unfortunately, still in Figure 54, we can find not only the same types of uncorrelated distributions of points over areas that in reality have small populations, see the Top two yellow circles and the two smaller red circles, but also an additional type, where the shade of grey appears to indicate more population than the one the points within it convene, see the larger red circle.

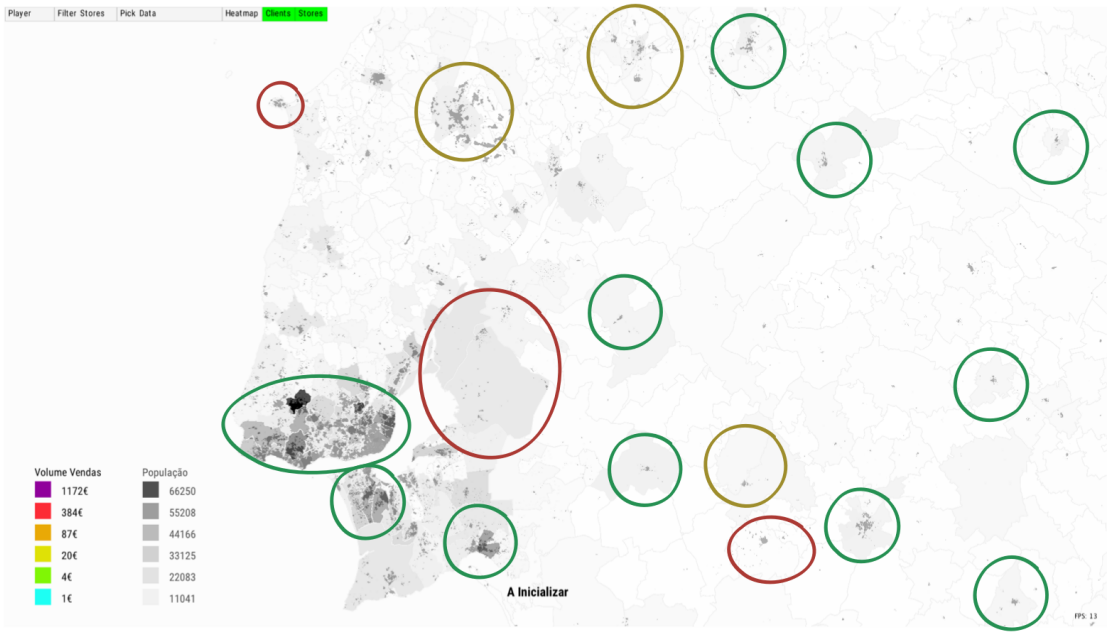

Figure 54 - Lisbon City area map with an overlay of the StarsShader result map, with increased contrast, brightness and transparency, over the choropleth Population map colored with shades of grey. The Green, Yellow and Red circles represent Good, Median and Bad correlation points. Only the most relevant were highlighted.

6.3.2. Choropleth Maps Comparison

Even if Choropleth Maps created with orthogonal lines varying in distance is something back from the time of Plot Printers, due to the procedural texture generation performance attainable now-a-days, we decided to implement it alongside the more usual greyscale coloring methods to perform comparisons.

In Figure 55, representing Population of Civil Parishes in the Lisbon area, we can see that both methods produce almost identical results, in terms of perceived information. If we look closely at the more populated areas with numerous small Civil Parishes (in geographic space), in the top map, we notice that we can identify multiple different shades of grey, and if we analyze the exact same Civil Parishes in the Procedural version we notice that the variation of distance between lines from civil parish to civil parish is also differentiable. Yet, if we look closely at the right side of both maps, particularly the brighter areas, we will notice in the greyscale implementation that some Civil Parishes have different Populations than others, yet, we can see that in the Procedural version they are indistinguishable.

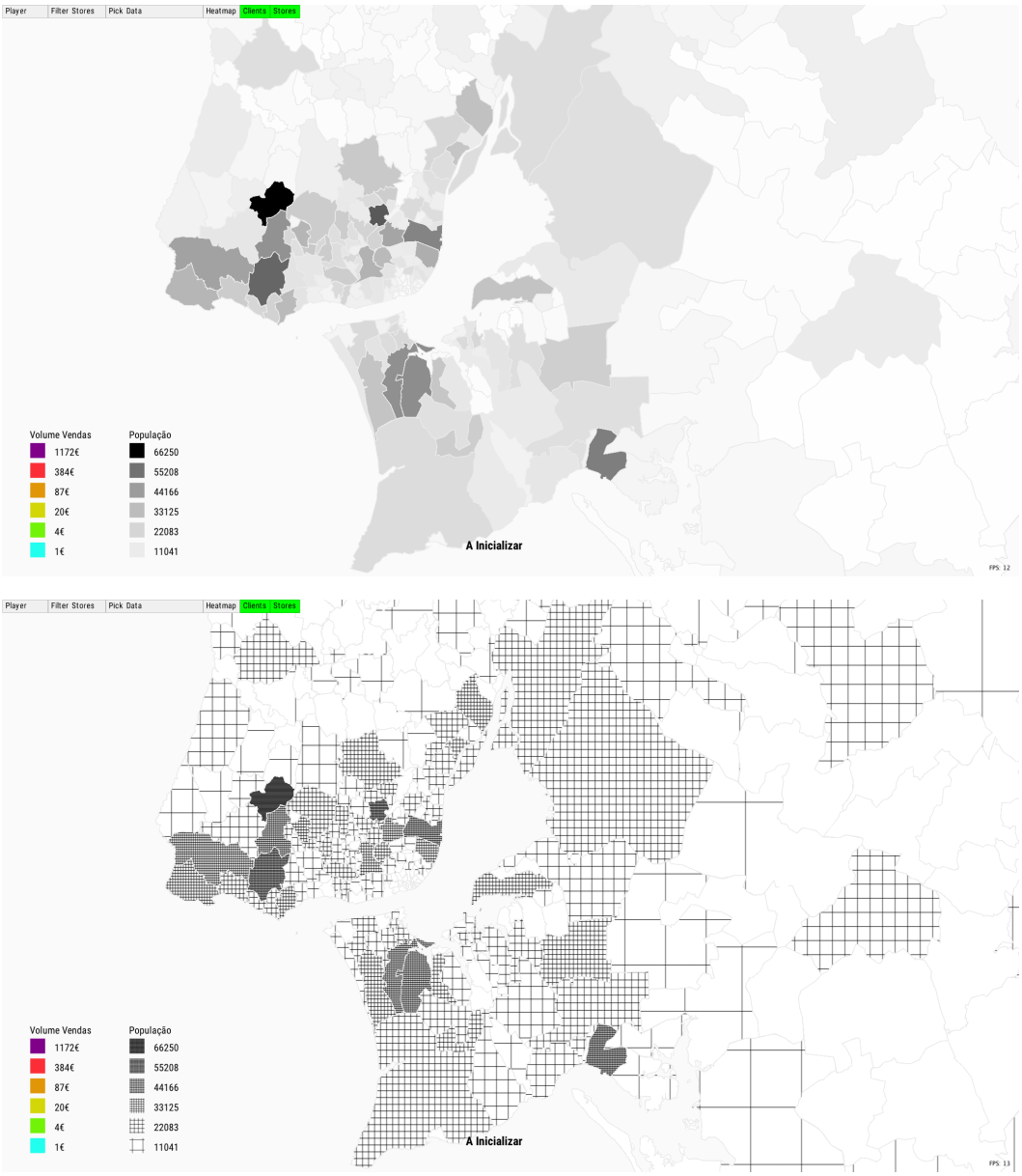

Figure 55 - Comparison between greyscale coloring (top) and procedural texture generation methods when representing Population of Civil Parishes in the Lisbon area.

This problem in differentiating areas with low values of the variable being represented is more noticeable in the least populated areas, refer to Figure 56. This disadvantage, or problem, can be an advantage if what we want is to highlight and differentiate only higher values. Yet, as we will see, this is not the only problem affecting this implementation.

In Figure 57, representing Purchasing Power Score of the Population per Civil Parish, which in contrast to the Population per Civil Parish, have very little variation from area to area, we can see that the Procedural Texturing approach creates a confusing surface, due to the high number of lines, with multiple different offsets and all with distances between lines in similar ranges of the scale.

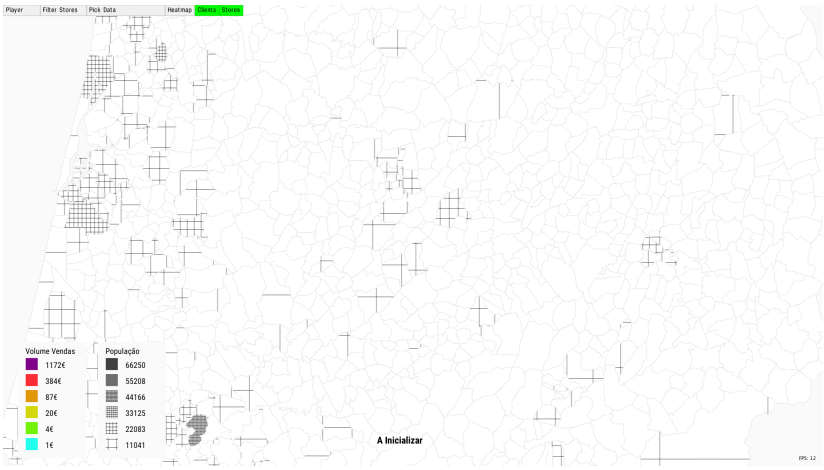

Figure 56 - Low populated areas are indistinguishable from each other using the Procedural Texture Generation method to color the Choropleth map.

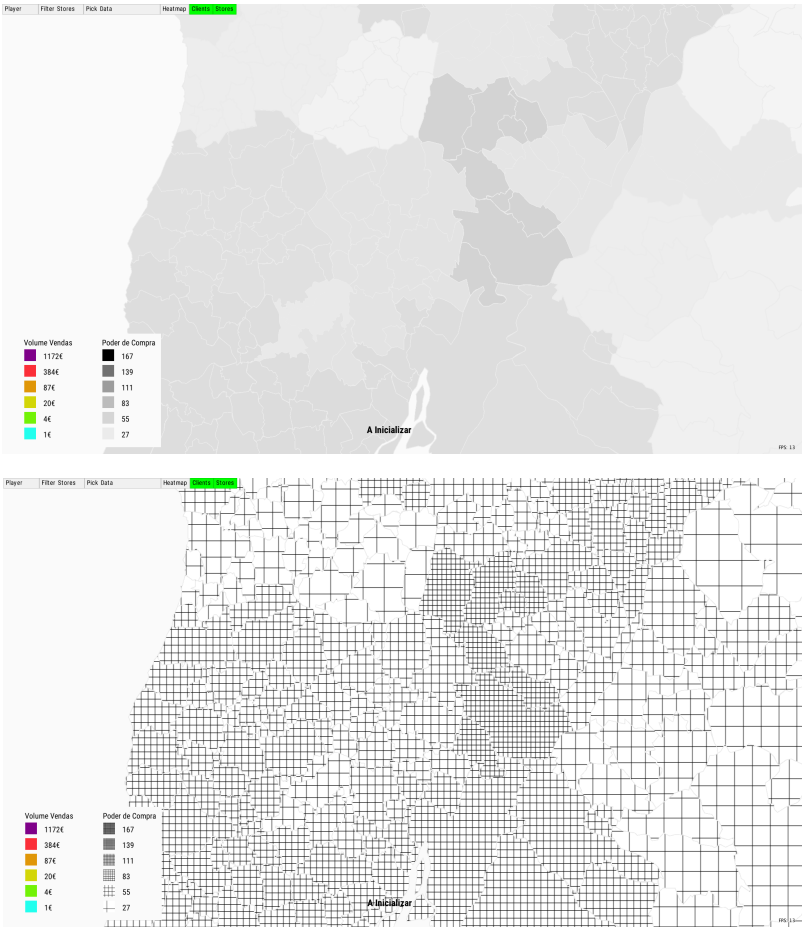

Figure 57 - Visual confusion of the Procedural implementation (bottom) when compared with the more appealing greyscale coloring (top).

Not mentioned so far, but as we mentioned in the implementation of the GreyShader, the greyscale we used to map the values (0-1) was logarithmic, as to give the Human User the most correct relation between tones and values, possible. The work of Tobler (1973), which motivated us to implement the Procedural version follows the same principle, and we can

confirm this by analyzing Figures 55 and 57, and observe how the perceived darkness or brightness of the same areas in both implementations appears the same.

This effect is best observed in the zoomed out Country view presented in Figure 58, where by zooming out and consequently pulling the procedural generated lines closer, we get almost identical darkness and brightness tone areas.

Figure 58 - Zoomed out comparison of the greyscale and procedural implementations of the Choropleth map.

Finally, as we can observe in many of the application's screen captured images shown so far, whenever the images depict any of the geographic heatmaps with the population per civil parish represented in shades of grey underneath, it is possible to differentiate the populations through the transparency of the heatmaps.

6.3.3. Performance Considerations

To compare the performance of the different Demographic implementation we devise a test in wish the geographic heatmaps were disabled and no data was being inputed, as to minimize the influence from other application components. Like before, each of the values presented was calculated from measured times spent per frame during a span of 2000 frames.

Regarding the performance of each of the Demographic representation implementations, we can see by the results presented in Table 2, that the differences between them are almost negligible, and not perceptible by the User.

	Stars		Greyscale		Procedural	
	Average	**STD**	**Average**	**STD**	**Average**	**STD**
zoomed out	30.951275	4.360317	31.176904	3.590133	31.254867	3.617087
zoomed in	12.673956	0.976230	14.223831	1.490249	14.048597	1.539867

Table 2 - Performance results for each of the different demographic visualization modes. Values are in Frames Per Second, the bigger the better.

6.4. Sales and Sales over Quantity

When the User selects the Sales over Quantity variable, to replace the default heatmap visualization, the GL_BLEND function used to blend values of different clients and stores is GL_MAX instead of the GL_ADD. We expected this to create a very dim value representation when compared with the normal geographical heatmap, yet we expect that due to normalization we are still able to differentiate zones where clients pay more, in general, per unit of bought products.

Although we didn't use this feature to perform any type of analysis, we present a couple of screen captures of the obtained results, Figures 59 and 60. Figure 59 shows an overview of the northern part of the Country, and Figure 60 of the southern part of the Country.

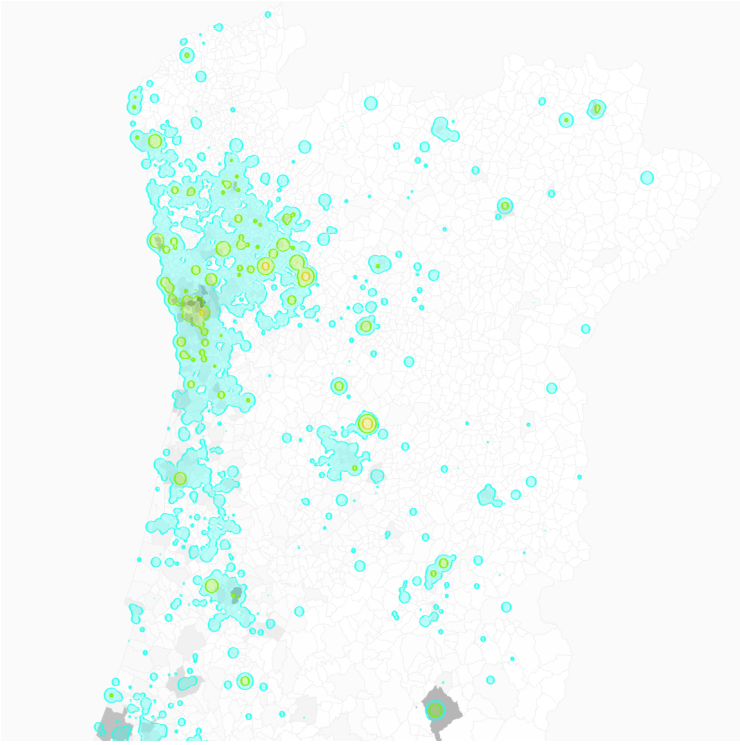

Figure 59 - Visual representation of the Sale Value over Quantity variable's distribution over the northern part of the Country.

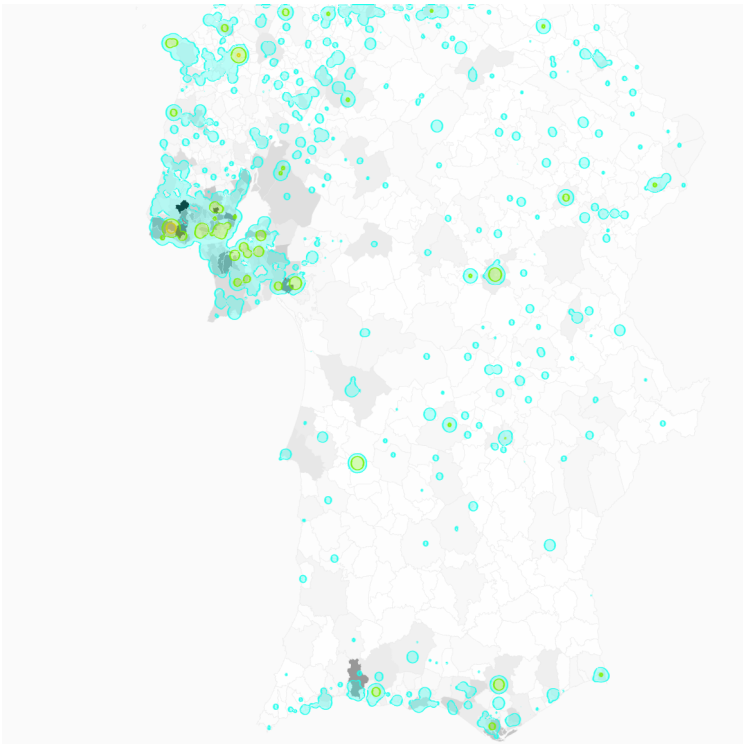

Figure 60 - Visual representation of the Sale Value over Quantity variable's distribution over the southern part of the Country.

7. Discussion and conclusions

In this Thesis we presented different methods of how researchers can make use of the available processing power in any nowadays laptop and desktop computers. By using the OpenGL graphics pipeline to perform simple operations, or the more general purpose programing model of OpenCL, that allows for high parallelization of programs and high programming flexibility. Additionally we showed how using OpenCL/OpenGL interoperability removes the necessity for additional data transfers from the GPU (OpenCL) to the CPU and back again from the CPU to the GPU (OpenGL), for rendering purposes.

Choosing a lower end GPU as our test platform, the results we achieved with it, show how this parallel processing power is widely spread across todays hardware, from tablets to desktop computers.

Switching to the OpenCL API to perform the computing steps, allowed to add additional information without impacting performance. Not only we were able to additionally track values for Clients, more numerous than the Stores (709 to ≈180000), but we were also able to track an additional variable, marginally more complex, the Sales Values or the Quantity of Products for both Stores and Clients. Hoping that this demonstrates the referenced flexibility of OpenCL.

We also implemented an anti-aliasing algorithm, that results show it's working as expected[11], with a negligible impact on performance. The purpose of the algorithm was to enhance the visual quality of the representation and also improve the recognition of shapes. With it, we attempted to demonstrate how researchers can use modern rendering APIs, OpenGL in our case, to produce pixel perfect representation of elements, comparable to those produced by more complex software development kits aimed at vector art representations.

Finally, we showed public and private, free of charge, services that enable anyone to gather large and diverse geographical information, and then demonstrated methods of displaying it using OpenGL, and integrate it with the geographical heatmap visualization model.

Regarding our implementation as a software product, we have fulfilled the defined objectives and, as we seen in the results obtained, we were also able to meet the requirements set, described in Chapter 1.3.1.

7.1. Future Work

During our implementation process, there were, at multiple points, too many different paths of implementation and optimization for us to explore them all. Researchers present multiple and diverse interesting techniques that we could had experimented with.

[11] In fact the results exceeded our expectations, as we expected more artifact or broken lines.

Additionally, the work we done as a whole, and for some of the problems we encountered, we formulated possible experiments that could constitute independent research projects on their own.

7.1.2. Application improvements

Optimization is a very time demanding and complex process. The different components where to perform optimization and the multiple ways there are to optimize them, make it a "never ending" process. Developers have to balance and determine if further optimization is worth the spending of resources. We followed a similar approach but still, we think a little more time spent in optimization could further improve the Users experience, particularly in regards the rendering of the Geographic Heatmaps FBOs, that is currently where the application spends most of it's processing time. We still experienced with a technique called Instancing, replacing the geometry generation, but was left out as it did not improve the results.

Further future work could also be done in introducing additional features to the application. As an exploratory tool there are several additional features that could support that task, to name a few:

- additional mouse over information, for instance isolating Stores heatmaps, view Stores detailed information and others;
- additional visualization tools (charts and others) to complement the visual information of the heatmap;
- more tracked statistical variables that provide different analysis, for instance a moving average or static full day average views;
- high dpi printing feature, which was only partially implemented using a technique derived from Multisampling;
- mechanisms that give the User more control over the animation flow;
- natively introduce additional, and relevant administrative variables;

Concerning the more overall objectives of the Research Project integrated, we also think it would be of great interest, the exploration and implementation of visualization models that encompass both Sales and Stocks records. Unfortunately we need additional data, overlapping in time we the data we currently have, to make that possible.

7.1.3. Alternative directions

During the optimization of OpenCL kernels, particularly the AddKernel, the procedure we followed in order to seek the best configuration, suggested us an alternative research project that could yield interesting results.

The proposition consists in the exploration of evolutionary solutions to the problem of optimizing OpenCL kernels executing continuously, or over undetermined amounts of time, while processing undetermined quantities of highly variable data.

We suggest, that the evolved solutions be the ones who determine the appropriate dimensions and sizes chosen when enqueueing the kernel execution, but also allow the solutions to manipulate kernel properties and control variables that influence the kernel execution, for instance the number of operations performed by each work item.

Additional we suggest that the evolutionary process should be used, **not** to evolve an "individual" that will be later used in "live" execution of the kernel, but instead, we suggest that the evolutionary process should run continuously along side the kernel execution, in order to promote the continuously evolution and search for the best solutions at any giving time and giving the varying kernel execution conditions.

References

[1] Nguyen, H. (2007). Gpu gems 3. Addison-Wesley Professional.

[2] Pharr, M., & Fernando, R. (2005). Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley Professional.

[3] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. Proceedings of the IEEE, 96(5), 879-899.

[4] Gregg, C., & Hazelwood, K. (2011, April). Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on (pp. 134-144). IEEE.

[5] Levenshtein, V. I. (1966, February). Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady (Vol. 10, No. 8, pp. 707-710).

[6] Pol, L. G., & Thomas, R. K. (1997). Demography for business decision making. Greenwood Publishing Group. (Preface and Chapter 1)

[7] INE, D. G. D. C., & de Estudos Regionais, G. (2000). Estudo sobre o poder de compra concelhio. Nucleo de Estudos Regionais da Direccao Regional do Centro, Coimbra.

[8] Feeman, T. G. (2002). Portraits of the earth: A mathematician looks at maps (Vol. 18). American Mathematical Soc..

[9] Willems, N., Van De Wetering, H., & Van Wijk, J. J. (2009, June). Visualization of vessel movements. In Computer Graphics Forum (Vol. 28, No. 3, pp. 959-966). Blackwell Publishing Ltd.

[10] Scheepens, R. J. (2010). GPU-Based Track Visualization of Multivariate Moving Object Data (Doctoral dissertation, Master's thesis, Eindhoven University of Technology, Dept. of Computer Science and Engineering, Visualization Group, Eindhoven, The Netherlands, 2010. http://www. win. tue. nl/visnet/wiki/doku. php).

[11] Buschmann, S., Trapp, M., Lühne, P., & Döllner, J. (2014). Hardware-accelerated attribute mapping for interactive visualization of complex 3D trajectories. In Proc. of the 5th International Conference on Information Visualization Theory and Applications (IVAPP 2014) (pp. 355-363).

[12] Maciejewski, R., Rudolph, S., Hafen, R., Abusalah, A. M., Yakout, M., Ouzzani, M., ... & Ebert, D. S. (2010). A visual analytics approach to understanding spatiotemporal hotspots. Visualization and Computer Graphics, IEEE Transactions on, 16(2), 205-220.

[13] Ho Ahn, S. (n.d.). OpenGL Projection Matrix. Retrieved August 19, 2015, from http://www.songho.ca/opengl/gl_projectionmatrix.html

[14] GluProject and gluUnProject code. (n.d.). Retrieved August 20, 2015, from https://www.opengl.org/wiki/GluProject_and_gluUnProject_code

[15] Catanzaro, B. (2010). Opencl optimization case study: Simple reductions. White Paper.

[16] Sunday, D. (2001). About Lines and Distance of a Point to a Line (2D & 3D).

[17] Tobler, Waldo R. "Choropleth maps without class intervals?." Geographical Analysis 5.3 (1973): 262-265.

[18] Ebert, D. S. (2003). Texturing & modeling: a procedural approach. Morgan Kaufmann.

[19] Tudorica, B. G., & Bucur, C. (2011, June). A comparison between several NoSQL databases with comments and notes. In Roedunet International Conference (RoEduNet), 2011 10th (pp. 1-5). IEEE.

[20] Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H. A., & Mankovskii, S. (2012). Solving big data challenges for enterprise application performance management. Proceedings of the VLDB Endowment, 5(12), 1724-1735.

[21] Tompson, J., & Schlachter, K. (2012). An introduction to the opencl programming model. Person Education.

[22] AMD Accelerated Parallel Processing OpenCL Programming Guide. (2013, November 1). Retrieved August 29, 2015, from http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

[23] Angel, E., & Shreiner, D. (2011, August). Introduction to modern openGL programming. In ACM SIGGRAPH 2011 Courses (p. 7). ACM.

24] Optimizing InnoDB Configuration Variables. Retrieved August 30, 2015, from https://dev.mysql.com/doc/refman/5.7/en/optimizing-innodb-configuration-variables.html

[25] InnoDB Configuration. Retrieved August 30, 2015, from https://dev.mysql.com/doc/refman/5.7/en/innodb-configuration.html

[26] Alternative Storage Engines. Retrieved August 30, 2015, from https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html

[27] Graphics Card Performance Hierarchy Chart - Best Graphics Cards For The Money: June 2015. (2015, June 22). Retrieved August 30, 2015, from http://www.tomshardware.co.uk/gaming-graphics-card-review,review-32899-7.html

[28] Wong, P. C., & Bergeron, R. D. (1994, May). 30 Years of Multidimensional Multivariate Visualization. In Scientific Visualization (pp. 3-33).

[29] Chen, C. H., Härdle, W. K., & Unwin, A. (2007). Handbook of data visualization. Springer.

[30] Liu, Z., Jiang, B., & Heer, J. (2013, June). imMens: Real−time Visual Querying of Big Data. In Computer Graphics Forum (Vol. 32, No. 3pt4, pp. 421-430). Blackwell Publishing Ltd.

[31] Ware, C. (2012). Information visualization: perception for design. Elsevier.