



FACULTY OF COMPUTER SCIENCE
DEPARTAMENT OF ELECTRONICS AND SYSTEMS

FINAL YEAR PROJECT
DEGREE IN COMPUTER ENGINEERING
MENTION IN SOFTWARE ENGINEERING

***Real-time management tool for
massive 3D point clouds***

Student: Luis Omar Álvarez Mures
Advisors: Emilio J. Padrón González
Alberto Jaspe Villanueva

A Coruña, September 2014.

Información xeral

Título do proxecto: **“Ferramenta para o traballo interactivo con grandes nubes de puntos 3D”**

Clase de proxecto: Proxecto de desenvolvemento en investigación

Nome do estudante: Luis Omar Álvarez Mures

Nome dos directores: Emilio José Padrón González
Alberto Jaspe Villanueva

Membros do tribunal:

Data de lectura:

Cualificación:

Dedicated to Rebeca and my parents.

Acknowledgements¹

My sincerest thank you for the help provided:

To Mr. Alberto Jaspe Villanueva.

To Mr. Emilio José Padrón González.

To Mr. Juan Ramón Rabuñal Dopico.

¹In alphabetical order

Resumo:

O obxectivo deste proxecto é o deseño e implementación dunha ferramenta multiplataforma para a visualización de nubes de puntos que permita a xestión e manipulación de grandes volumes de datos. Este tipo de *datasets* poden ser obtidos a partir dun escáner LIDAR ou de alternativas de baixo custo como KINECT.

Este tipo de sistemas capturan un gran volume de información, xerando nubes de puntos de gran tamaño con miles de millóns de información xeométrica, color, normais, reflectividade, etc. A consecuencia é un conxunto de información difícil de manexar de xeito interactivo, ao non caber completo na memoria RAM dun ordenador e requirir o traballo desde memoria secundaria (clásicamente un disco duro). Isto complica a creación de aplicacións en tempo real para a xestión e manexo deste conxunto masivo de datos.

A ferramenta software desenvolvida neste proxecto ofrécelle ao usuario final todas as funcionalidades necesarias para o traballo con este tipo de datos, desde un completo visualizador 3D para render de puntos, que inclúe características de visualización avanzada de OpenGL, soporte para o traballo con varias nubes de puntos simultaneamente, múltiples cámaras diferentes coas que traballar, características multiresolución, posibilidade de aplicar transformacións e distintos tipos de pre-proceso sobre os datos, medir distancias, detección de obxectos na nube de puntos, exportación a CAD dos obxectos seleccionados, etc

Para a implementación de todas esas funcionalidades partiuse da biblioteca software PCM, aínda en desenvolvemento e que xurde tamén na Unidersidade da Coruña. Este biblioteca proporciona unha serie de ferramentas de baixo nivel para a xestión de nubes de puntos masivas en hardware de uso común. Neste proxecto non soamente se fai uso das características existentes de PCM, senón que, ademais da propia ferramenta de visualización desenvolvida (bautizada como ToVIEW), tamén se estendeu amplamente a biblioteca PCM para poder ofrecer todas as funcionalidades necesarias no proxecto, ademais de varias melloras de rendemento na mesma.

No proxecto abordouse ademais o aproveitamento das posibilidades para o procesamento paralelo presentes no hardware no que se executa PCM, tanto en canto á CPU como ás capacidades GPGPU da tarxeta gráfica.

Por último, o paquete software resultante foi deseñado para ser facilmente integrable nos fluxos de traballo habituais nos sectores da enxeñería que máis habitualmente empregan escaneos 3D en forma de nubes de puntos, como pode ser, por exemplo, a topografía. Durante a realización do proxecto traballouse con profesionais da topografía habituais a incorporar o uso da tecnoloxía LIDAR no seu traballo, o que facilitou a creación dunha

ferramenta útil e versátil para o usuario final. No aspecto da fácil integración en entornos profesionais tamén axuda o feito de tratarse dun desenvolvemento multiplataforma e *open source*.

Abstract:

The objective of this project is designing and implementing a multi-platform point-based rendering visualizer that allows the management and interactive manipulation of big 3D point clouds. These clouds can be obtained using a LIDAR scanner or cheaper alternatives like KINECT.

These systems can nowadays generate data-sets with huge amounts of point data, position, color, normals, reflectivity, etc. As a consequence, these point clouds will not fit on system RAM and will have to reside in an HDD. Due to this fact, real-time rendering of massive point clouds is a complex task.

The resulting software tool developed in this project, will offer the end user the functionality necessary for working with these types of datasets. Including a complete 3D visualizer, with advanced OpenGL point-rendering capabilities, multiple point cloud support, multiple cameras, multi-resolution capabilities, point cloud transformation and preprocessing, distance measurements, object segmentation, CAD exportation of segmented objects, etc.

To implement all of these features, we will utilize PCM. PCM is a project in development from the University of A Coruña, that provides a series of low level tools for working with point clouds on commodity hardware. This project will not only extend PCM with the aforementioned functionality and a high level software tool for the end user, but will improve the existing code and some performance aspects of PCM.

Furthermore, the provided tool will take advantage of all the parallel computing capabilities of the target platforms. Using multiple cores on CPUs and GPGPU on the graphics cards. The multi-resolution framework will also help when trying to achieve maximum performance when trying to apply any computation.

In conclusion, the resulting software package should be easily integrated and improve substantially the existing engineering workflows, thanks to the fact that the tool will be multi-platform and open source. Since it was also developed with the advice of some surveyors that are working with laser scanners everyday, there are high hopes of truly having created a versatile and useful tool.

Lista de palabras chave:

Motor, render, nubes de puntos, k-d tree, multi-resolución, tiempo real, PCM, ToView, open source, multi-plataforma.

Keywords:

Engine, render, point clouds, k-d tree, multi-resolution, real-time, PCM, ToView, open source, multi-platform.

Hardware e software empregado:

- Intel Core i7s, NVIDIA GTXs e GTs.
- C++, Visual Studio, Git, OSG, PCL, openMP, dxflib, OpenGL, OpenCL, QT, GLEW, GCC.

Contents

1	Introduction	1
1.1	Motivation and context	2
1.2	Objectives	3
1.3	Structure	4
2	Planning and methodology	7
2.1	Agile software development	7
2.1.1	Agile manifesto	7
2.1.2	Extreme programming	9
2.1.3	Application	11
3	Computer Graphics basics	15
3.1	Rendering	15
3.1.1	Rendering process	17
3.1.2	3D Models	18
3.2	Splats and points	19
3.3	3D Euclidean space and transformations	20
3.4	Camera model	22
3.4.1	Camera parameters	23
3.4.2	Camera inner working	25
4	Structure of a point-based visualizer	31

4.1	Analysis	31
4.1.1	Use cases	32
4.2	Design	41
4.2.1	Class diagram	41
5	PCM	45
5.1	Design	45
5.1.1	Dataset conversion and management	45
5.1.2	Spatial data structure	47
5.1.3	Memory hierarchy	48
5.1.4	External interface	48
5.2	I/O	50
5.2.1	Supported point cloud formats	50
5.2.2	BPC	51
5.3	Memory hierarchy	53
5.3.1	L1	54
5.3.2	L2	57
5.3.3	PCM Benchmark	59
5.4	Results	61
6	ToView	75
6.1	Design	75
6.1.1	GUI	76
6.1.2	Render thread	77
6.1.3	Segmentation	78
6.2	Functionality	79
6.2.1	Render modes	80
6.2.2	Camera types	81
6.2.3	Preprocessing	82

6.2.4	Cloud interaction	83
6.2.5	Point picking	84
6.2.6	Distance measurements	85
6.2.7	Primitive segmentation	86
6.2.8	Primitive exportation	88
6.3	Results	89
7	Advanced GPU point rendering	93
7.1	Performance details	93
7.2	Splat rasterization	94
7.2.1	Image-aligned squares	94
7.2.2	Affinely projected point sprites	96
7.2.3	Perspectively correct rasterization	97
7.3	Antialiasing	100
8	Object detection: RANSAC	103
8.1	Original algorithm	103
8.2	Alternative algorithms	105
8.2.1	MSAC and MLESAC	105
8.2.2	PROSAC	106
8.2.3	RRANSAC	107
8.3	Point-primitive distance calculation	108
8.3.1	Point-plane distance	108
8.3.2	Point-cylinder distance	110
8.3.3	Point-sphere distance	111
8.4	Primitive bounding	111
8.4.1	Plane bounding	112
8.4.2	Cylinder bounding	112
8.5	Results	114

9 Preprocessing and filtering	117
9.1 Radii estimation	117
9.2 Statistical outlier removal	119
9.3 Voxel grid filter	120
10 Conclusions and future lines of work	123
10.1 Conclusions	123
10.2 Future lines of work	124

List of Figures

1.1	Modeling types comparison	1
1.2	Examples of LIDAR scanning.	2
2.1	Sketch of the structure used for the story cards.	12
2.2	Gantt chart of the project.	13
3.1	Rendering types comparison: photorealistic vs. non-photorealistic.	16
3.2	The OpenGL pipeline.	18
3.3	Example of a point-based rendering: photo overlaid atop laser scan data from a project held in early 2009 at Kasubi Tombs.	19
3.4	Disc approach: Visual representation of the splat as a disc.	20
3.5	Pinhole camera	22
3.6	Camera orientation	23
3.7	Hither and Yon	24
3.8	Perspective and orthographic camera comparison	24
3.9	Viewing frustum	28
3.10	FPS camera rotation	28
3.11	Orbital camera rotation	29
4.1	Pipeline representing the workflow in ToView.	31
4.2	ToView use case diagrams: (a) User interaction (b) Development interface.	32
4.3	PCM class diagram.	41

4.4	ToShow class diagram.	42
5.1	Conversion and management of datasets class diagram	46
5.2	Data structure class diagram	47
5.3	Memory hierarchy class diagram	49
5.4	External interface class diagram	50
5.5	Chunk structure	52
5.6	Internal tree structure	52
5.7	Memory hierarchy	53
5.8	L1 cache structure	54
5.9	L2 structure	57
5.10	Common nodes close paths	59
5.11	Sample PCMBenchmark output	60
5.12	CPU/GPU radii estimation Castelao	63
5.13	CPU/GPU radii estimation Dorna	63
5.14	Hits versus chunk size versus VBO size in Lourizan	64
5.15	Load time vs. chunk size vs. VBO size in Lourizan	64
5.16	Chunk size vs. hit rate in Lourizan	65
5.17	Chunk size vs. miss rate in Lourizan	65
5.18	Chunk size vs. load time in Lourizan	66
5.19	Hits versus chunk size versus VBO size in CALD	66
5.20	Load time vs. chunk size vs. VBO size in CALD	67
5.21	Chunk size vs. hit rate in CALD	67
5.22	Chunk size vs. miss rate in CALD	68
5.23	Chunk size vs. load time in CALD	68
5.24	Hits versus chunk size versus VBO size in Baron	69
5.25	Load time vs. chunk size vs. VBO size in Baron	69
5.26	Chunk size vs. hit rate in Baron	70

5.27	Chunk size vs. miss rate in Baron	71
5.28	Chunk size vs. load time in Baron	72
5.29	Hits versus chunk size versus VBO size in Castelo	72
5.30	Load time vs. chunk size vs. VBO size in Castelo	73
5.31	Chunk size vs. hit rate in Castelo	73
5.32	Chunk size vs. miss rate in Castelo	74
5.33	Chunk size vs. load time in Castelo	74
6.1	User interface class diagram	76
6.2	Render thread class diagram	77
6.3	Segmentation class diagram	79
6.4	Early ToView screenshot	80
6.5	Render modes	81
6.6	Camera options	82
6.7	Tools menu	83
6.8	Multiple point cloud integration	83
6.9	Cloud interaction	84
6.10	Point picking example	85
6.11	Distance measurement example	86
6.12	Segmentation dialog	87
6.13	Segmented cylinder	88
6.14	Segmented primitives in AutoCAD	89
6.15	FPS obtained in Lourizan	90
6.16	FPS obtained in CALD	91
6.17	FPS obtained in Baron	92
6.18	FPS obtained in Castelo	92
7.1	Image aligned squares defects	96

7.2	Comparison between affinely projected and perspective accurate rasterization	97
7.3	Perspective accurate	98
7.4	Comparison between perspective accurate rasterization and image-aligned squares	99
7.5	Typical super-sampling patterns	100
7.6	MSAA subsample placement for a rotated grid pattern	101
7.7	MSAA and non-MSAA results for a partially covered point	101
7.8	MSAA and non-MSAA result comparison	102
8.1	RANSAC example	104
8.2	Hessian normal form	109
8.3	Cylinder parameters	110
8.4	Sphere parameters	111
8.5	AABB-plane intersection	112
8.6	Ray-plane intersection	113
8.7	AABB-cylinder intersection	113
8.8	Cylinder axis-plane intersection	114
8.9	Plane segmentation test	115
8.10	Cylinder segmentation test	115
8.11	Sphere segmentation test	116
9.1	Estimated radii on a sphere	118
9.2	Statistical outlier filter comparison	119
9.3	Voxel grid filter	120
9.4	Voxel grid filter comparison	121

List of Tables

2.1	Cost estimation.	14
5.1	Datasets used in our tests.	62
8.1	RANSAC testing results	114

Chapter 1

Introduction

Computer Graphics is the discipline that deals with creating, managing, analyzing and visualizing graphics using a computer, and it is closely linked to other disciplines such as Image Processing, Computer Vision or Machine Learning, among others. Advances in computer graphics have had an impact on several media types such as, for example, the movies and video games industry, civil engineering, medicine, etc.

In computer graphics, *rendering* is the process of generating a 2D image to be shown in a display. This process takes usually a virtual scene of 3D objects as input, containing data like geometry, lighting, cameras, etc. Typically, polygonal modeling has been the approach followed for modeling objects in the scene, approximating their surfaces using polygons. Point-based graphics focuses on points as the fundamental representation of the surfaces instead of polygons (see Figure 1.1).

Nowadays, acquisition technologies such as LIDAR (Laser Imaging Detection and Ranging) allow us to obtain high-precision georeferenced 3D scans of the real world in an easy and fast way. These systems use ultraviolet, visible or near infrared light to image objects.

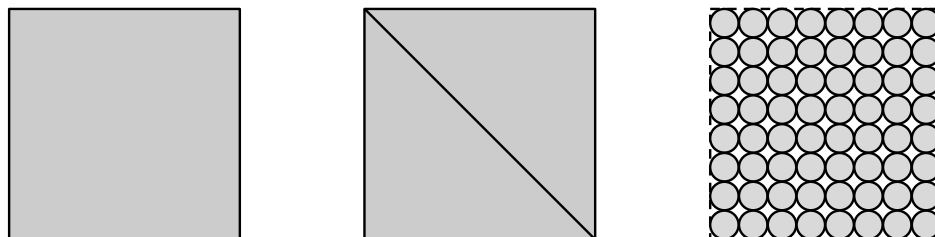


Figure 1.1: From left to right; mathematical representation of a plane, representation using two polygons (triangles) and a point-based representation.

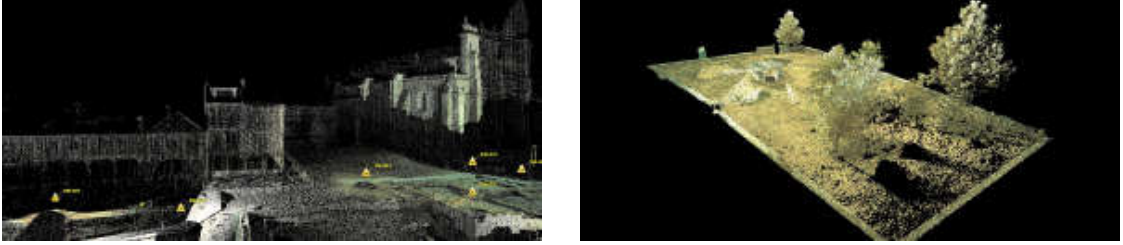


Figure 1.2: Examples of LIDAR scanning.

It can target a wide range of materials and even single molecules. A narrow laser beam can map features with very high precision. The evolution of this kind of technologies is having a tremendous impact in multiple fields in the engineering, and bringing back the camp-work to the office has become a standard practice. This presents clear advantages compared to the classical work-flow in disciplines like topography, that usually need to come back on the ground each time a new measurement is required. A couple of images obtained from a LIDAR scanner are shown in Figure 1.2.

These scanning devices generate data-sets with an enormous number of samples, with 3D coordinates and additional data such as color, reflectivity and so on. These point clouds, as they are usually named, are formed by hundreds or thousands of points, resulting in Giga or even tera-bytes of information. Using point based representations, the polygonal approximation step can be skipped and the massive amount of captured data is directly used. However, managing these huge data-sets, either for applying some kind of data analysis or for visualization, is a computational challenge and an open and state-of-the-art problem. This project deals with this question.

This chapter introduces the motivation behind the project and the objectives we try to achieve. It shows the structure of this document as well.

1.1 Project motivation and context

PCM is a software library developed in the University of A Coruña for the management of huge point datasets obtained by LIDAR in commodity hardware systems. It was started as a MSc Final Project [Jas12] (system structure and basic capabilities) in the Faculty of Computer Science and then continued in a BSc Final Project [Jas13] (improved render and other minor additions).

PCM is based on a multi-resolution structure and includes basic memory hierarchy management, making it possible to transfer the point cloud data between secondary memory (HDD) and video memory in the graphics card. Originally, PCM includes a very basic

visualizer, quite simple and limited but that has the potential to become the seed for a more advanced point cloud software tool that will allow the user to access all of these features in real-time.

Although PCM was designed to deal with arbitrary size datasets, it is far from being a mature software and has some problems managing point clouds in the order of billion points. Because of the massive nature of these datasets, achieving real-time manipulation of them is a complex task. It will require the improvement of PCM, with a complete profiling and performance analysis and a careful refactoring. Having a system that is able to manage such a huge amount of data, will let the user take advantage of the full potential of cutting-edge 3D capture devices. Of course, the library should exploit the full potential of both modern CPUs and GPUs, automatically using parallel processing to reduce the computation time as much as possible. Furthermore, although PCM sketches a framework for applying data analysis on the point clouds, this part was barely crafted at all. Other constraints of the original proposal are also addressed in this work, such as adding read/write operations to allow the modification of the datasets, the possibility of working with multiple point clouds, etc.

In this project, a complete user-oriented software suite for the visualization, management, analysis and manipulation of massive 3D point clouds will be developed from the original PCM proposal. ToView is the name chosen for this software package. It is a multi-platform (GNU/Linux, Windows and MacOS) free software project and makes use of open source software exclusively. For these reasons the user interface is based on the cross-platform application framework QT, that has native implementations and tools for all the aforementioned platforms. The resulting QT application will let the user access all of the functionality including an advanced and complete 3D visualizer.

1.2 Project objectives

The aim of this project is the design and implementation of a point-based multi-platform software tool for the interactive visualization, management, analysis and manipulation of massive 3D point clouds.

The application will offer the user the necessary tools to work effectively with this type of datasets, including a 3D visualizer with advanced point-rendering techniques using OpenGL, that will allow real time interaction with one or multiple point clouds.

From the interface, the user will be able to:

- Select different visualization modes for point-based models: multiple cameras, simple

and advanced visualization, multi-resolution, etc.

- Combine different point clouds with tools that enable rotation, translation and scaling of the datasets.
- Select points in the clouds for working with them.
- Apply operations to a part of the cloud or the complete dataset. From simple operations, such as distance measurement and calculation, to plane or other geometric primitives detection.
- Export the work done to a standard CAD format, so the tool can be integrated in other work-flows.
- Call the pre- and post-processing tools included in the suite: multi-resolution structure computation, filters, etc.

The developed tool will take advantage of the parallel computing possibilities of the platforms. Both using the GPGPU capabilities of the graphics card with OpenCL and the multiple cores of the CPU.

PCM will be used as the foundation for this work. PCM is a project in development that offers several low level tools for working with point clouds of arbitrary size in commodity hardware, from which a multi-resolution structure with different levels of cache is highlighted. This final degree project will not only extend PCM with the mentioned features, offering a high level software suite for the final user, but will also improve the existing source code, paying special attention to performance aspects.

A series of benchmarks will also be implemented that will allow us to obtain automatically and autonomously performance information about PCM and the visualization tool, with graphs and reports. This tool will make finding bottlenecks and performance issues easier.

1.3 Document structure

The rest of the document is organized following this structure:

Chapter 2. Planning and methodology

After this introduction, we start by describing the planning and development methodology used in the project. We chose an agile software development approach.

Chapter 3. Computer Graphics basics

This chapter briefs the very basic concepts about computer graphics needed to understand the rest of the document: How a virtual camera works and the different types of cameras, what 3D transformations are and how a point is represented in 3D.

Chapter 4. Structure of a real-time visualizer for point-based datasets

This chapter describes the structure of the visualizer we propose in this work and presents the class diagram of the software package. Both ToView (the visualization tool) and PCM (the multi-resolution library behind the GUI) are included.

Chapter 5. Point Cloud Manager

This chapter focussed in describing PCM ant detailing all the work invested in the improvement of the original PCM library and tools: code refactoring, bug fixes and new functionalities. It includes the experimental results of the tests included in the benchmarks we have designed to measure the library performance.

Chapter 6. ToView

This chapter depicts the visualizer designed and implemented from scratch in this project to manage and visualize massive point clouds. All the included functionalities are described and experimental results are also included.

Chapter 7. Advanced GPU point rendering

This chapter is devoted to the most advanced visualization methods included in ToView, a bunch of some of the most state-of-the-art point-rendering techniques, and how they are implemented with OpenGL in the GPU using GLSL (OpenGL Shading Language).

Chapter 8. Object detection: RANSAC

This chapter deals with the detection of different shapes and volumes in ToView, describing the different approaches that were studied to implement this functionality in the project. Specifically, an extensive explanation of the RANSAC algorithms used in this project is provided.

Chapter 9. Preprocessing and filtering

Both PCM and ToView need several pre-processing tasks to take advantage of some of the most advanced features. All this pre-processing and the different filtering techniques applied to the datasets are explored in this chapter.

Chapter 10. Conclusions and future lines of work

Finally, this chapter presents the main conclusions obtained with this project and outlines possible extensions to both PCM and ToView.

Chapter 2

Planning and methodology

This chapter explains what software development method was used in the project. We are strong believers in agile software development, above all for research-oriented projects with few main developers like this. In the next sections, some basic concepts of this development philosophy are introduced, as well as how it was applied in our project. Additionally, a Gantt chart with the different steps and periods in the development of the project is presented. Since this is a project that requires a lot of research, it is difficult to plan ahead every step, since we do not know how long we are going to spend on certain elements. Finally, a cost table with the estimated cost for a project of this type is included as well.

2.1 Agile software development

Agile software development [Coc01] is a combination of development methods that use iterative and incremental development, where requirements and solutions mature through collaboration between self-organizing, cross-functional teams. The motto of this method is “embrace change”; that is why it encourages adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and promotes quick and flexible response to change.

2.1.1 Agile manifesto

In February of 2001, several developers met at Snowbird, Utah resort, to debate different lightweight development methods. They published the *Manifesto for Agile Software Development* [Bec01] to define the approach that is now called agile software development.

The conclusions that we can reach from the manifesto's items are described below:

- **Individuals and Interactions:** In agile development self-organization and motivation are really important. Other values promoted by the manifesto are co-location¹ and pair programming².
- **Working software:** Working software will be utilized for more purposes than presenting documents to the client.
- **Customer collaboration:** The software requirements cannot be fully realized from the beginning of the software development cycle, so being in touch with the customer is really important.
- **Responding to change:** Agile development is keen on fast responses to change and continuous development.

More principles are mentioned in the manifesto, some of them are:

- Customer satisfaction by rapid delivery of useful software.
- Welcome changes even late in the development.
- Working software is the principal measure of progress.
- Maintaining a constant pace.
- Cooperation between business people and developers.
- Build projects around motivated individuals.
- Attention to technical excellence.
- Simplicity.

Agile methods break down tasks into small increments with minimal planning and normally long-term planning is not directly involved. *Iterations* are short timeframes that typically last from one to four weeks. A team works in each iteration through a full software development cycle; including planning, requirements analysis, design, coding, etc. This minimizes risk and facilitates adaptation to change. An iteration may not add enough new functionalities to warrant a market release, but the objective is to have an available release at the end of each iteration.

¹The act of placing multiple individuals within a single location.

²Two programmers work together at one workstation.

Team composition does not depend on corporate hierarchies or corporate roles of team members. They normally have the responsibility of completing tasks that deliver the required functionalities that an iteration requires. How to meet an iteration's objectives is decided individually.

The “weight” of the method depends on the type of project, the planning and order of tasks in a generalist project should not be the same as in a research project.

Agile methods encourage face-to-face communication instead of written documents if possible. Most teams work in an open office (the *bullpen*), which makes this type of communication easier.

Each agile team contains a customer representative, that ensures that customer needs and company goals are aligned.

Most agile methods encourage a routine that includes daily face-to-face communication among team members. In a brief session team members tell each other what they achieved the previous day, what they are going to do today and the problems that have appeared.

As agile development emphasizes on working software as the primary measure of progress and has a clear preference in face-to-face communication this results in less written documentation than other methods. This does not mean that documentation should be disregarded, but that less emphasis is made on documentation because it is not needed as much.

2.1.2 Extreme programming

Extreme Programming Explained [Bec99] describes this method as a software-development discipline that lays out how to organize people to produce higher-quality software more productively.

XP tries to reduce the cost of requirement volatility by using short development cycles, rather than a really long one. In this teaching, change is inherent, inescapable and a desirable aspect of development projects, should be welcome with open arms and planned for in advance, instead of trying to cling to certain fixed requirements.

Extreme programming also introduces a number of basic values, principles and practices on top of the agile programming principles.

- **Values**

- *Communication*: To build a software system it is required to communicate the requirements to the developers. This is achieved through documentation.

- *Simplicity*: It is encouraged to start simple and add extra functionality at a later time. The main difference between this method and others is focusing on design and coding that satisfies the needs of today and not tomorrow.
- *Feedback*: From the system, customer and team. This is closely related to communication and simplicity.
- *Courage*: Simplicity, refactoring and getting rid of code when necessary require courage.
- *Respect*: Respect for others and yourself. Always think of the team when making changes and respect the work of everybody.

- **Principles**

- *Feedback*: This principle is most useful when is done frequently and promptly. Minimal delay between actions and feedback is key to learning and making changes.
- *Assuming simplicity*: The fact of treating every problem as if its solution were really “simple”. This doctrine stresses that big changes all at once do not work.
- *Embracing change*: Change should be embraced instead of trying to work against it or avoid it.

- **Practices**

- *Informative workspace*: Build your workspace around your work. Anyone that visits your workspace should have a general idea of how the project is going in 15 minutes.
- *Sustainable pace*: Work as many hours as you can be productive. Overworking today and wasting the next two days is not good.
- *User stories*: Plan using units of work that the client will understand. Having the stories on display will help with having an informative workplace and keeping track of progress.
- *Incremental design*: Invest in system design every day. Try to excel in designing the system so that it meets the day to day requirements.
- *Pair programming*: A technique in which two programmers work together in a single workstation.
- *Continuous integration*: This technique helps to prevent integration problems. It consists on merging all developer working copies with a shared stable version of the software.

- *Ten minute build*: Build the complete system automatically and execute every test in 10 minutes.
- *Test-driven development*: Write automatic tests that fail before changing any code.
- *Whole team*: Not only the developers should communicate between each other, the customer should also be available for questions.
- *Sit together*: Have an open space available for the team. If it is not possible, video-conferences should be used to have more face to face time.
- *Weekly cycle*: Start the week writing automated tests and then during the week implement them.
- *Quarterly cycle*: Plan for releases that will cover themes. Themes are broken down into stories for weekly cycles.

2.1.3 Application

Several of the premises of the agile software development were applied in this project. We have used approximately weekly or biweekly iterations focused on completing simple objectives that would yield working software. The work in each iteration has been constant and diligent, focusing always on completing the task at hand, but not avoiding changes if it was deemed necessary.

Also several XP practices were used in the project, such as user stories (an example is shown in Figure 2.1). Thus, each planned functionality was broken up in multiple stories if it is big enough. Then the effort to implement each story was approximated, since the estimation units are simple enough to provide a rough estimation. This estimation were just meant as guidance and mainly used to keep track of progress. Another practice followed was an informative workspace, the story cards were stuck to the wall so they were easily visible to everybody. They were also organized in groups, so the progress in each functional area was easily tracked.

Incremental design was also used, every day the design was reviewed and adapted to the new requirements. Lastly, a sustainable pace was kept. Because designing and implementing software is a creative process that requires insight, developers have to be rested, ready and motivated. To make the most of the worked time, the premise of coding for two hours straight without checking e-mails or other distractions was used, along with other similar productivity tricks.

The meetings with my two advisors were kept on face-to-face basis when possible. In each meeting we would comment what had been achieved since the last one, laid out what the

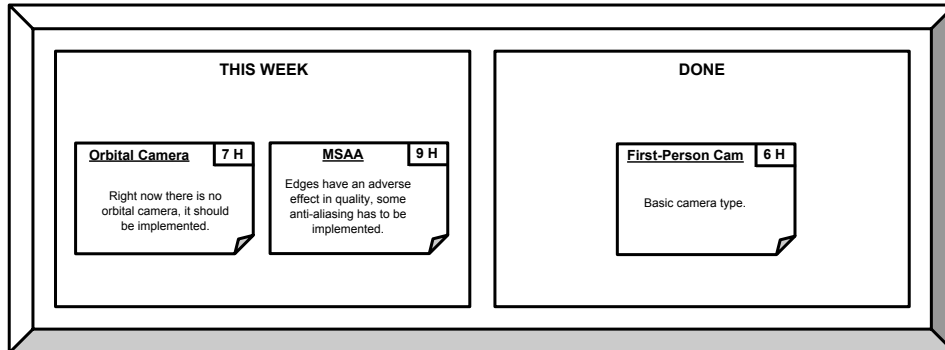


Figure 2.1: Sketch of the structure used for the story cards.

objectives were for the next iteration and talked about the roadblocks found on the task at hand. Whenever communication in person was not possible, e-mails were exchanged regularly if tasks were completed or any other event arised.

To keep the code accessible by everyone in the team, up to date and well organized **Git** [Tor05] version control software was used. Git is free software used to manage source code and keep distributed revision control. Every Git working directory is a repository with complete history and revision tracking capabilities. In this project it was essential to have a system like this in place, because of the research nature of the project. Ideas needed to be tried without wasting time on keeping tabs on the added code, Git makes this really easy and not problematic thanks to history and revision tracking.

Since the project is divided into two sub-projects (PCM and ToView³), it was also necessary to manage the separate repositories in some way. With this in mind, **Redmine** [Lan06] was chosen to achieve our goal. This is a free and open source web-based tool for project management and bug tracking. It has support for multiple projects, a Gantt chart, calendar, support for multiple users, several types of repositories, news, wiki, etc.

As this project is meant to be open source, this tool will allow the coordination of the efforts of future developments well after the end of this project. We believe this project will be a good fit for the “bazaar type” of development that other successful open source software has used. In this model, the code is developed openly for everyone to see. This type of development also encourages the participation of the user in the development process.

Although the research nature of this project made it difficult to have a good estimation in advance of how much time would be spent in each iteration, we provide a Gantt chart that depicts how the project iterations were performed after the fact (see Figure 2.2).

³The visualizer, the main focus of this project.

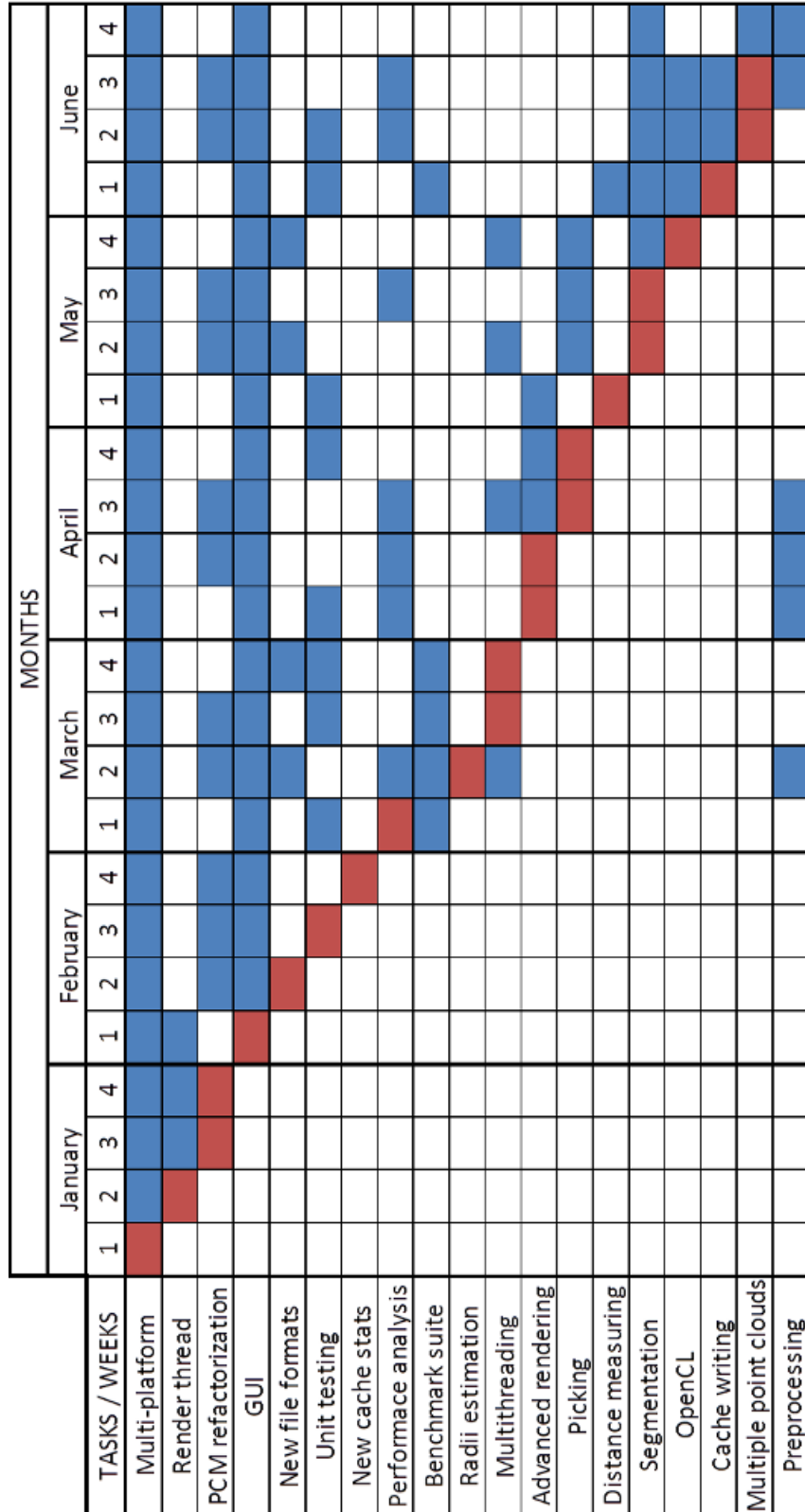


Figure 2.2: Gantt chart of the project.

Resource	Units	Hours	Cost per hour	Total
Software				0 €
Computer	1			1800 €
Analyst-programmer	1	950	16 €	15200 €
				17000 €

Table 2.1: Cost estimation.

We have also estimated the cost of a project like this, as depicted in Table 2.1. It only includes human resources and hardware costs because all the software used was FLOSS.

Chapter 3

Computer Graphics basics

In this chapter, some concepts in 3D rendering are introduced, paying special attention to the main components of a real time point-based rendering visualizer. We first describe the process that generates the frame-buffer. This means that we have to explain the camera model. We will also depict how we represent points in space by several means that will be introduced in this chapter.

Only a brief description of these concepts is presented, to get more familiarized with Computer Graphics the book [FvDFH12] is recommended. For point-based rendering techniques the book [GP07] is an essential source.

3.1 Rendering

Traditionally Computer Graphics has been defined as the computer science discipline that is dedicated to synthesizing images algorithmically with computers. Nowadays we can find even more related topics like hyper realistic photography, animation techniques, virtual reality, etc. To generate images from three-dimensional scenes a process called **render** is used. This set of actions is in charge of modeling objects and their properties, illumination if needed and the camera that will capture everything.

As has been said before, rendering is the computational process of generating an image from a model. From this definition multiple interpretations are possible, from creating a 3D animation film to a bar chart in a spreadsheet, all of them are equally valid. Although the term is normally used when the model we are using is of a spatial nature and more specifically three-dimensional.

Several classifications of rendering techniques could be made [AMHH08], but for this



Figure 3.1: Rendering types comparison: photorealistic vs. non-photorealistic.

introduction we will use two of them. On one side, if we use the type or style of the image we want to achieve, we will have:

- **Non-photorealistic rendering**, uses other types of effects that render the scene with artistic style, intended to look like a painting or drawing.
- **Photorealistic rendering**, tries to be as faithful as possible to reality. This type of render can be subdivided in:
 - **Physically-based rendering**: Tries to compute an authentic simulation of light transport through the virtual scene and its interaction with materials and objects. The precision of this simulation will depend on the mathematical and physical models chosen.
 - **Faked**: Utilizes algorithmic tricks, not trying to pretend an ad-hoc simulation of light, usually to reduce render times.

Figure 3.1 shows a photorealistic render on the left picture and an example of non-photorealistic on the right one.

On the other side, if we compare the interaction capabilities of the rendering application with the user, we can divide rendering techniques in:

- **Offline rendering**: Where the process that generates the image is too slow to respond instantaneously to the user interactions. A time scale of seconds and up to several days can be considered “slow”. This happens for example when generating photograms for a movie.
- **Online rendering, also known as Real-time rendering**: Where processing time would be short enough to respond to user interactions so that the user has the

sensation of continuity. A time scale of milliseconds would be needed to achieve this effect. Normally it is measured in *frames per second* (FPS). A good example of this technique are videogames.

Most of the work included in this document is focused in achieving real-time rendering, even though that means sacrifice a better photorealistic-looking aspect, since ToView is aimed to allow the user to interactively work with huge point cloud datasets.

3.1.1 Rendering process

There are several techniques for generating images from a model. All of them start with a camera position, the geometry of the objects is projected in that direction one way or another, calculating the color of each resulting pixel. The two families of techniques are the ones based on *scanline rendering*, where lists of geometry are traversed with horizontal scan lines. Intermediate calculations determine what object is closer to the camera, how do the lights in the scene affect the objects, etc.

Since we require real-time rendering, we will take advantage of the power of the GPU. The GPU *pipeline* is the process implemented in hardware to create images from the scene information in a highly efficient way. It is comprised of several stages where several lineal algebra operations that account for the position, rotation and scaling of the point of view, objects and lights; to finally determine the color of the pixel drawn to the screen.

A GPU can be used to process graphical data using two APIs¹:

- **DirectX**: Microsoft's proprietary API, only useful in Windows systems.
- **OpenGL**: Standard and multiplatform, this were the main reasons why it was the chosen API for this project.

In modern GPUs, some of the stages of the pipeline can be modified using small programs written in GLSL² called shaders. In Figure 3.2 the simplified pipeline of a modern version of OpenGL can be observed. In it the vertex, geometry and fragment stages are programmable. The *frame-buffer*³ will be the final image that will be shown onscreen.

Although there are some constraints in the form of number of instructions and control structures, shaders started a revolution in real-time rendering. Sometimes even allowing to create images almost as realistic as offline rendering.

¹Application Programming Interface

²GL Shading Language

³The portion of memory (buffer) reserved to maintain temporally an image (frame) awaiting to be sent to the monitor.

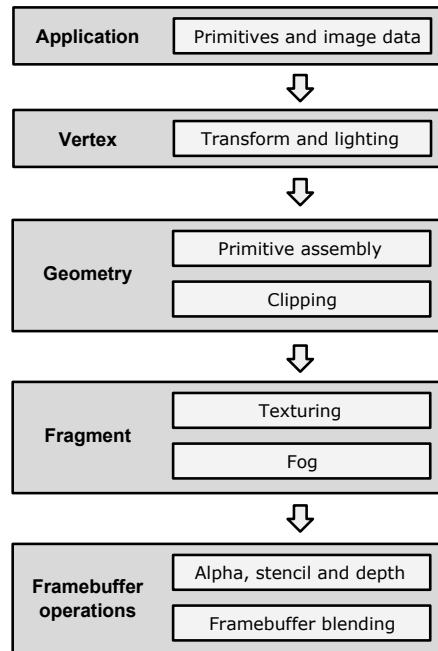


Figure 3.2: The OpenGL pipeline.

3.1.2 3D Models

Modeling describes the process of forming the shape of a virtual object in a computer. There are mainly three different types of 3D models:

- **Polygon-based:** Describe the surface of an object as a set of polygons. Triangles are the most commonly used polygons, since they are flat and trivially convex. Most of the existing graphics hardware is optimized for this primitive.

Although modeling is usually not directly carried out using triangles (a scene is often formed with a mixture of all kind of polygons and parametric surfaces), the final triangles are finally obtained by a process known as tessellation.

- **Voxel-based:** Divides space in a regular 3D grid where the *voxel* is the smallest unit of volume. Each cell is either filled or not and depending on that the pixels are shaded. Memory increases as precision is augmented. This kind of render is often used in Biomedical Informatics.
- **Point-based:** Objects are represented by point samples of their surface, that is why they are usually called *splats*. Each point has a position and some information about the surface that it belongs to. Compared to the traditional triangle-based approach, this primitive needs its own rendering techniques and its own pipeline, since different



Figure 3.3: Example of a point-based rendering: photo overlaid atop laser scan data from a project held in early 2009 at Kasubi Tombs.

challenges must be faced. This is our case study and because of that, this primitive will be further explained in Section 3.2.

3.2 Splats and points

A point cloud is a set of vertices or points in a three-dimensional coordinate system. These vertices are usually positioned in 3D space and have a set of coordinates (x, y, z) . These sets of points normally are representative of the external surface of an object.

Point cloud datasets are usually captured by 3D scanners. These devices capture automatically a large number of points on the surface of an object (see Figure 3.3), and yield a point cloud dataset as a result.

The points from the range data represent a surface, but a point is zero-dimensional; this means that it does not have volume, area, length or any other higher dimensional equivalent. Usually point clouds are not directly usable in most 3D applications because of this reason. That is why we need splats (surface elements), that describe the surface in a small neighborhood.

At this point we need to choose how the surface will be represented by the point. In our project we will represent the splat as a disc.

On the other hand, if the splat is represented as a disc, it can have the following parameters (see Figure 3.4):

Position: x , y and z coordinates in world space.

Normal: A vector that represents the surface normal of the splat. It can be provided or

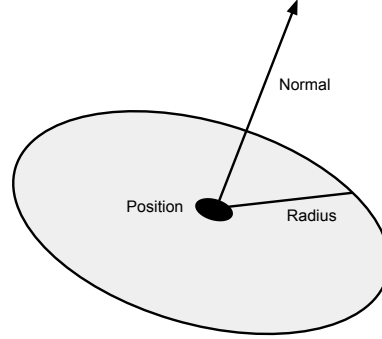


Figure 3.4: Disc approach: Visual representation of the splat as a disc.

the engine can estimate it.

Radius: This will be the radius of the sphere that will represent the splat. The engine will also be capable of calculating the radius automatically.

Color: The splat color in RGB parameters if we are going to use the lighting from the scanned data.

Other properties: Intensity, reflectivity, temperature, etc.

3.3 3D Euclidean space and transformations

In 3D computer graphics we normally work with a three-dimensional space of Euclidean geometry, the term “Euclidean” is used to distinguish between these spaces and the curved spaces of non-Euclidean geometry. The most common types of operations in Euclidean geometry can be represented with a transformation matrix if homogeneous coordinates are used. Because of this a transformation $\mathbf{T}$ can be used for several purposes.

Using basic linear algebra concepts, a 4×4 matrix can be used to express the linear transformation of a point or vector. A transformation will then be represented by the elements of the 4×4 matrix. They can also be used to perform some transformations that are non-linear on an Euclidean space. For this reason transformation matrices are widely used in computer graphics.

Generally a transformation is a mapping from points to points or vectors to vectors [PH10], for example:

$$p' = \mathbf{T}(p) \quad v' = \mathbf{T}(v) \quad (3.1)$$

To transform the points or vectors we just have to perform the appropriate matrix multiplications. This also allows the composition of transformations, we just have to multiply the transformation matrices.

Most commonly used transformations are:

- **Rotation**, will rotate a point or a vector by a given angle either around an arbitrary axis or the x , y or z axis.
- **Scaling**, takes a point or a vector and will scale its x , y and z components by a factor.
- **Translation**, only affects points and will translate coordinates x , y and z a set amount.
- **Model**, this matrix is useful for transforming the model locally. It is comprised of a set of rotation, scaling and translation matrices. When we apply this matrix to all the points, we will have them in world coordinates.
- **View**, initially the camera is placed in the origin of world space. To be able to move around the world this transformation is used. This matrix is also called *look-at*. After this transformation is applied, we will have the points in camera coordinates.
- **Projection**, since a 3D scene has to be projected onto the screen as a 2D image, we need another process that converts the points from camera coordinates to homogeneous coordinates so that they can be projected onscreen.

Usually the composition of the model, view and projection matrices is called *MVP* and is applied to every point that we want to draw.

To illustrate how the transformations are represented by a 4×4 matrix, equation 3.2 shows the generic transformation matrix for a translation:

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.2)$$

This transformation is applied to a point $P = (x, y, z)$ in the following way:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix} \quad (3.3)$$

3.4 Camera model

Almost everyone nowadays has used a camera and knows its basic purpose: you want to record an image of the world (usually pressing a button) and the image is then recorded on a film. One of the simplest cameras in the real world is the *pinhole camera*. Pinhole cameras use a light tight space with a hole at one end. When this hole is uncovered light enters through the hole and reaches a piece of photographic paper on the other end of the box (see Figure 3.5). In this day and age, cameras are more complex than this simple camera model, but this is a good starting point to explain how our simulation works.

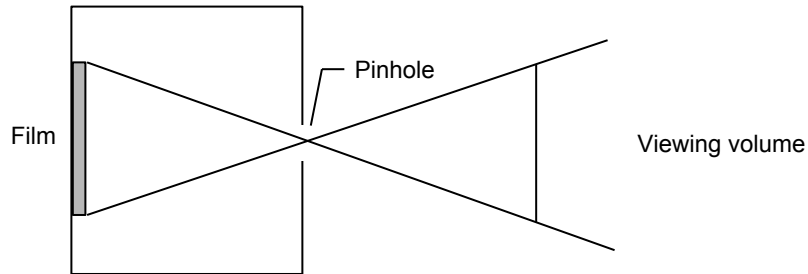


Figure 3.5: Diagram explaining how a pinhole camera works.

The most important function of the camera is defining the part of the scene that will be recorded on the photographic film. Connecting the pinhole to the edges of the film creates a double pyramid that extends into the scene. Objects that are not inside this pyramid will not be imaged on the film. As cameras are now more complex, we will refer to the region that can be imaged as *viewing volume*.

If we were to use the pinhole as the film, the viewing volume would not change. When the film or image is in front of the pinhole, the pinhole is frequently referred to as the *eye*. In our simulated camera where the film is in front of the eye, we will display the amount

of light traveling from the image plane to the eye. Several tests will be performed by OpenGL depending on the type of camera to check which points can be seen and have to be represented in the frame-buffer.

The camera model used in this project supports several settings. First we explain how this settings affect the camera model, after we detail how the camera model interacts with the ray tracing process. Readers can consult [PH10] if they want to delve deeper in the topic.

3.4.1 Camera parameters

The first parameter that the camera model supports is **camera position**. This parameter states where the camera will be positioned in world space coordinates (x, y, z) .

Next supported parameter is the **camera orientation**. This parameter is a point in the world at which the camera is looking at.

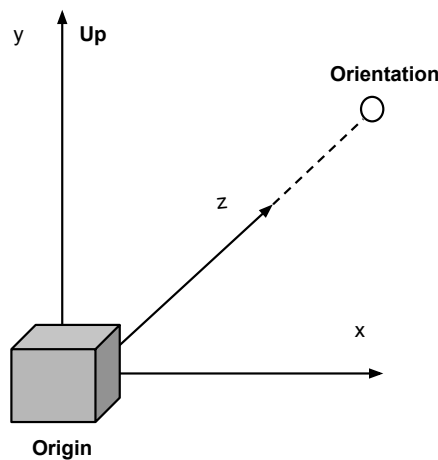


Figure 3.6: Diagram explaining the camera coordinates.

We also need to know how to orient the camera along the viewing direction implied by the first two parameters. The parameter **camera up vector** gives us that orientation.

We can see how this parameters are organized in camera space in Figure 3.6.

Other important parameters in a camera model are the clipping planes. **Hither** dictates where the near clipping plane of the camera is located and **yon** indicates where the far clipping plane is situated as we can see in Figure 3.7. The camera's clipping planes give us the range of space along the z axis that will be visible in images. Objects that are in front of the hither plane or beyond the yon plane will not be visible.

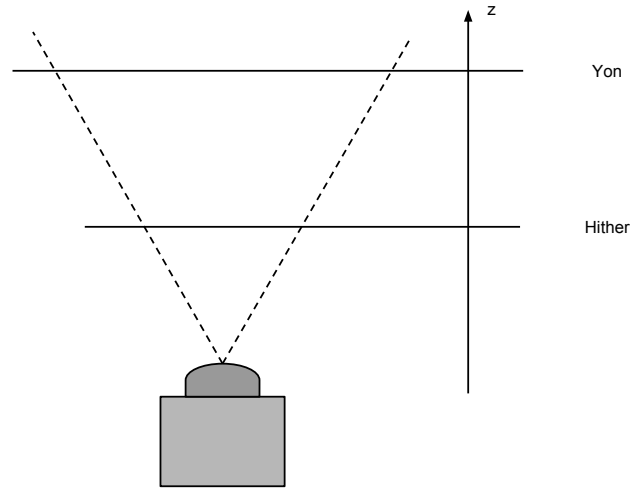


Figure 3.7: Representation of hither and yon planes.

The last parameter that the camera supports is the **field of view**. The angle of view describes the angular extent of the scene captured by the camera horizontally and vertically. The clipping planes and the angles of view define the viewing volume in our model, also known as the *viewing frustum* (see Figure 3.9).

The user also has the option to use a **perspective** or **orthographic** camera depending on what his needs are. An architect for example may prefer an orthographic camera, while a normal user may use a perspective camera that will yield a more natural image as result (see Figure 3.8).

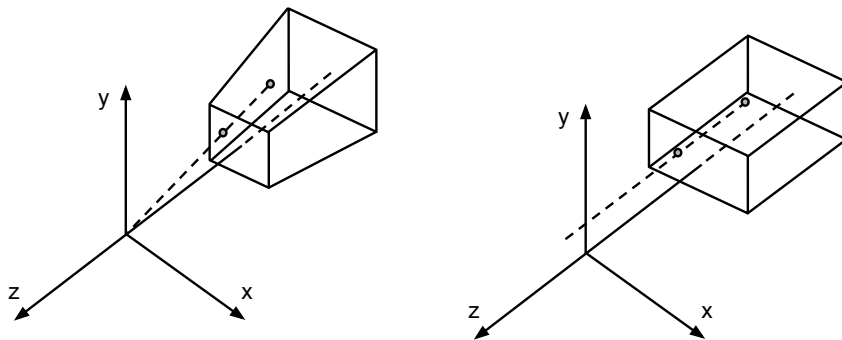


Figure 3.8: Representation of the difference between a perspective (left) and orthographic camera (right).

Orthographic projection is more commonly used in engineering because dimensions are

communicated unambiguously, if there is a one unit length line anywhere in the scene; it will appear the same length everywhere. Also every line that is parallel in the scene will be parallel in reality. With perspective projection, lines of identical real-world length will appear different due to *foreshortening*⁴. This is why it will be difficult to judge relative dimensions when objects are far away.

Another option that is available is using a **FPS**⁵ or an **orbital** camera. On one hand with a first person camera the user can walk freely just like if he was inside the scene, forward, back, left, right, etc. On the other hand with an orbital camera the user sets a center point and then orbits around said point.

3.4.2 Camera inner working

Once we have the camera parameters we need to place the camera in the scene, we will use a *look-at* transformation for this purpose. Using the look-at transformation as view matrix will give us a transformation between world and camera coordinates as mentioned before. The matrix is constructed using the position (p), orientation (o) and up (u) vector.

First, a forward vector f is constructed:

$$f = \frac{p - o}{\|p - o\|} \quad (3.4)$$

Next, the right vector r is constructed:

$$r = f \times u \quad (3.5)$$

After, a new up vector u' is calculated in the camera reference:

$$u' = r \times f \quad (3.6)$$

With these, a rotation matrix can be constructed representing a reorientation into the newly created orthonormal basis:

⁴Objects becoming smaller on the image plane as they get further away.

⁵First-Person Camera

$$R = \begin{bmatrix} r_x & u'_x & f_x & 0 \\ r_y & u'_y & f_y & 0 \\ r_z & u'_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.7)$$

Finally, to transform objects into the camera's frame; not only everything has to be oriented correctly, it also has to be translated from the origin to the position of the camera. This is achieved by concatenating the rotation matrix R with the corresponding translation matrix:

$$V = \begin{bmatrix} r_x & u'_x & f_x & -p_x \\ r_y & u'_y & f_y & -p_y \\ r_z & u'_z & f_z & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.8)$$

Equation 3.8 is the resulting look-at matrix V that will be used to transform the points from world space to camera space.

The next step will depend on the type of camera that the user will choose, either orthogonal or perspective. In light of this a projection matrix will be chosen reflecting the choice.

If a perspective camera is chosen, we have to take into account the foreshortening effect we mentioned before. This projection does not preserve distances or angles, and parallel lines no longer remain parallel. x and y are not left unchanged, they are modified according to the corresponding z value, that dictates how close to the camera the point is (see Figure 3.8). To achieve this effect, a perspective projection matrix will be used. A commonly used matrix is the *frustum matrix*. In it the homogeneous space takes the shape of a truncated pyramid. The parameters used to construct this matrix are left (l), right (r), top (t), bottom (b), near plane (n) and far plane (f) of the frustum.

t is obtained in the following way using the camera parameters field of view (fov) and hither (n):

$$t = n \cdot \tan(fov/2) \quad (3.9)$$

Next, r is obtained using the width (w) and height (h) of the window:

$$r = t \cdot w/h \quad (3.10)$$

Since the frustum is symmetric, l and b will be the same as r and t but with the opposite sign. Once we have all of the parameters we can construct the perspective or orthographic matrices:

$$P = \begin{bmatrix} \frac{2 \cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2 \cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2 \cdot n \cdot f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (3.11)$$

In contrast, if a orthographic camera is required, a somewhat simpler orthographic projection matrix can be used. Points will be projected to the near viewing plane. This type of transformation does not give the effect of foreshortening, it keeps lines parallel and preserves relative distance between objects. This transformation leaves x and y coordinates unchanged, but maps z values at the hither plane to 0 and z values at the yon plane to 1 (see Figure 3.8). The parameters used to construct this matrix are also left (l), right (r), top (t), bottom (b), near plane (n) and far plane (f) of the frustum:

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.12)$$

One of these two matrices (3.11 or 3.12) will represent the projection matrix P .

The composition of these matrices will yield the MVP matrix [SWH13]. This is the mathematical expression used for transforming each point:

$$\begin{aligned} v' &= P \cdot V \cdot M \cdot v \\ &= MVP \cdot v \end{aligned} \quad (3.13)$$

Once we have the MVP matrix, we can perform *frustum culling*. The visualizer selects the only the points that fall inside the *viewing frustum* (see Figure 3.9). This is a essential process to achieve the necessary performance to be able to render the point clouds in real-time. The viewing frustum is the region of space that will appear in the images generated by the visualizer. This process of eliminating objects that are outside the viewing frustum is called *viewing frustum culling*.

Once we have the adequate coordinates we just have to map them to screen space depend-

ing on the resolution of the render. If two points are mapped to the same pixel we have to see which one is closer in the z axis, because that will be the point that the pixel is going to represent.

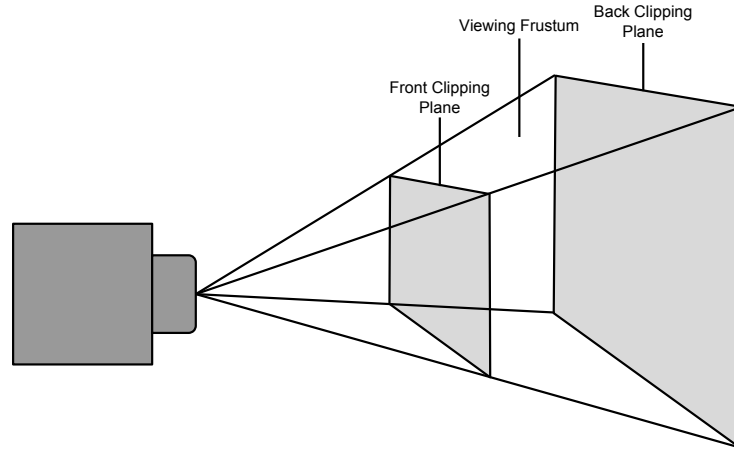


Figure 3.9: Diagram showing the viewing frustum.

These computations are costly when the scenes are big and to simplify the culling process we need an acceleration structure. These structures reject chunks of the scene that are not inside the frustum. In our project we have chosen to use a multi-resolution *k-d tree*.

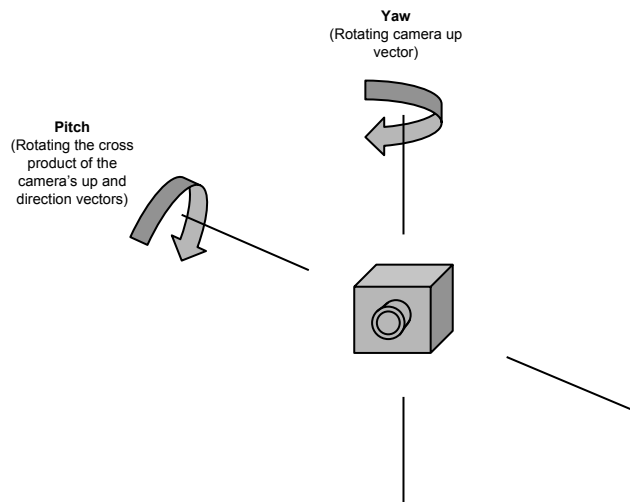


Figure 3.10: Diagram showing the rotation in a first-person camera.

This is the course of action followed to create one frame, but since the visualizer is interactive, we have to repeat this process several times per second. Furthermore, we have to

take into account how the user wants to move. If the FPS camera option is chosen, we start in a certain position and then we will be able to move forward, backward, left and right just as we would be able to do if we were walking around the scene in reality. The user will also be able to look around with complete freedom (see Figure 3.10). *Yaw* allows the user to look left and right and *pitch* up and down.

When the orbital camera is used, instead of freely walking around, we orbit around a center point (see Figure 3.11). The rotation in this instance is represented by a number of rotation degrees (α and β) around the x and y axes of a coordinate system centered in the interest point. Instead of moving the camera, we move the point of interest, that will be followed by the camera.

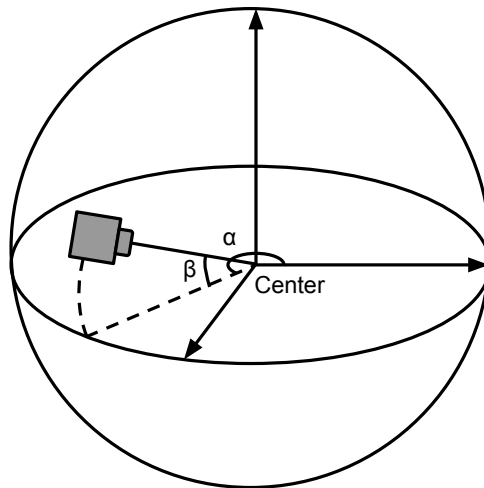


Figure 3.11: Diagram showing the rotation in a orbital camera.

Chapter 4

Structure of a point-based visualizer

In this chapter, the structure of PCM and ToView is shown. Thus, a high level description of the pipeline is done, which serves as a summary of the analysis phase for this project. The different stages of the pipeline are thoroughly described in the following chapters. Finally, a class diagram depicts the main design aspects of ToView and the updated version of PCM.

4.1 Analysis

Figure 4.1 depicts the usage pipeline of ToView. To use the system, a series of steps have to be followed. First, the raw LiDAR data is usually processed using software from the manufacturer (unifying scans, calibration, etc.). Once the resulting point cloud from the aforementioned software is available, it will be converted to our point cloud format.

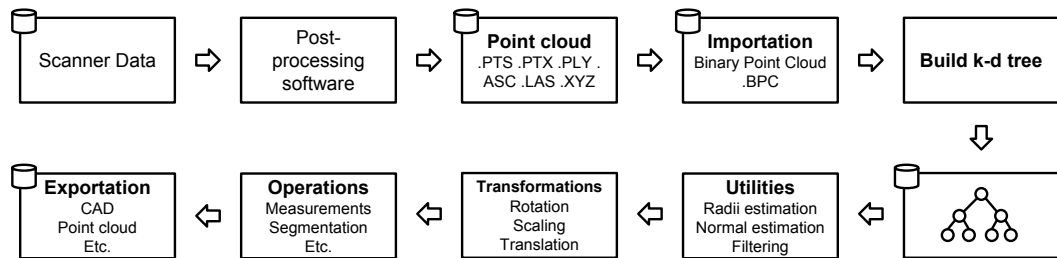


Figure 4.1: Pipeline representing the workflow in ToView.

Next, the k-d tree for accelerating calculations and rendering is created. When the k-d is created and stored in permanent memory, it can be used to estimate radii, normals, apply filters, etc. It can also be used to visualize the point clouds in an efficient way, since we have massive datasets and visualizing them in a naive way is not possible.

In addition, multiple point clouds can be merged using the visualizer. The user will be able to apply transformations (rotation, scaling and translation) to the individual clouds. This allows the user to for example mix a point cloud that represents a statue with another of its destination, and see in advance how it will look.

ToView can also be used to measure distances in point clouds that were registered with this purpose in mind. Lastly, the visualizer can be used to segment parametric primitives (planes, cylinders and spheres) and export them to a CAD compatible format.

4.1.1 Use cases

This section describes the two main actors in ToView and PCM: a normal user and a developer. The normal user will use the interface to interact with ToView to visualize and manage point cloud datasets, whereas the developer will use the library to run its own algorithms on the point clouds. The diagrams in Figure 4.2 depict the use cases for both actors. These uses cases are detailed in the following tables.

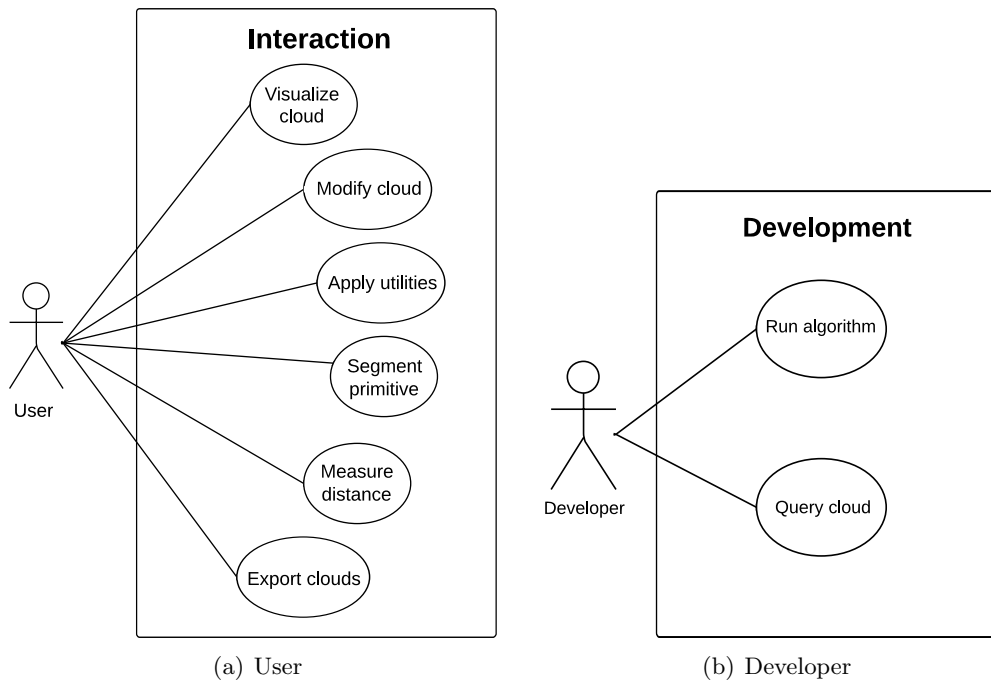


Figure 4.2: ToView use case diagrams: (a) User interaction (b) Development interface.

Use Case 1	Visualize cloud
-------------------	------------------------

<i>Scope:</i>	System-wide
---------------	-------------

<i>Primary Actor:</i>	User
-----------------------	------

<i>Preconditions:</i>	
-----------------------	--

<i>Postconditions:</i>	
------------------------	--

<i>Main Success Scenario:</i>	
	1. User starts ToView 2. Converts dataset to BPC format 3. Builds k-d tree 4. Opens k-d tree folder 5. Moves around the scene 6. Closes the visualizer

<i>Extensions:</i>	
3.a Error building k-d tree:	1. System shows failure message 2. User returns to step 2
4.a No k-d tree present in folder:	1. System shows failure message 2. User returns to step 3

Use Case 2 Modify cloud

Scope: System-wide

Primary Actor: User

Preconditions: Having a k-d tree

Postconditions:

Main Success Scenario:

1. User starts ToView
2. Opens k-d tree folder
3. Chooses operation to perform
4. The system applies the operation

The steps 3-4 are repeated for each operation desired

5. Closes the visualizer
-

Extensions:

- 2.a No k-d tree present in folder:
 1. System shows failure message
 2. User returns to step 1
-

Use Case 3 Apply utilities

Scope: System-wide

Primary Actor: User

Preconditions: Having a k-d tree

Postconditions:

Main Success Scenario:

1. User starts ToView
2. Opens k-d tree folder
3. Chooses filter, radii or normal estimation
4. The system applies the utility

The steps 3-4 are repeated for each utility

5. The system or User saves the changes
 6. User closes the visualizer
-

Extensions:

- 2.a No k-d tree present in folder:
 1. System shows failure message
 2. User returns to step 1
-

Use Case 4 Segment primitive

Scope: System-wide

Primary Actor: User

Preconditions: Having a k-d tree

Postconditions:

Main Success Scenario:

1. User starts ToView
2. Opens k-d tree folder
3. Selects the primitive segmentation settings
4. Selects a point in the dataset
5. System samples neighborhood
6. System segments primitive

The steps 3-6 are repeated for each primitive

7. User closes the visualizer
-

Extensions:

- 2.a No k-d tree present in folder:
 1. System shows failure message
 2. User returns to step 1
 - 4.a No points selected:
 1. System shows failure message
 2. User returns to step 3
-

Use Case 5 Measure distance

Scope: System-wide

Primary Actor: User

Preconditions: Having a k-d tree

Postconditions:

Main Success Scenario:

1. User starts ToView
2. Opens k-d tree folder
3. Selects to measure distance
4. Selects a point in the dataset
5. Selects another point
6. System outputs distance

The steps 3-6 are repeated for each measurement

7. User closes the visualizer
-

Extensions:

2.a No k-d tree present in folder:

1. System shows failure message
2. User returns to step 1

4-5.a No points selected:

1. System shows failure message
 2. User returns to step 3
-

Use Case 6 Export data

Scope: System-wide

Primary Actor: User

Preconditions: Successful segmentation made

Postconditions:

Main Success Scenario:

1. User selects export type
 2. Inputs the output file name
 3. System exports the corresponding data
-

Extensions:

2.a Invalid file name:

1. System shows failure message
 2. User returns to step 1
-

Use Case 7 Run algorithm

Scope: System-wide

Primary Actor: Developer

Preconditions:

Postconditions:

Main Success Scenario:

1. Developer selects algorithm
 2. Uses PCM's external interface to implement algorithm
 3. Selects precision desired
 4. Converts cloud to .BPC
 5. Creates k-d tree
 6. The system executes the algorithm
 7. The system displays or saves result
-

Extensions:

- 5.a Error building k-d tree:
1. System shows failure message
 2. User returns to step 2
-

Use Case 8 Query cloud

Scope: System-wide

Primary Actor: Developer

Preconditions: Having a k-d tree

Postconditions:

Main Success Scenario:

1. Developer selects k-d tree
 2. Uses PCM's external interface to make a query
 3. The system obtains the requested parts of the cloud
 4. Developer uses the provided information for a certain purpose
-

Extensions:

- 1.a No k-d tree present in folder:
 1. System shows failure message
 2. User returns to step 1
-

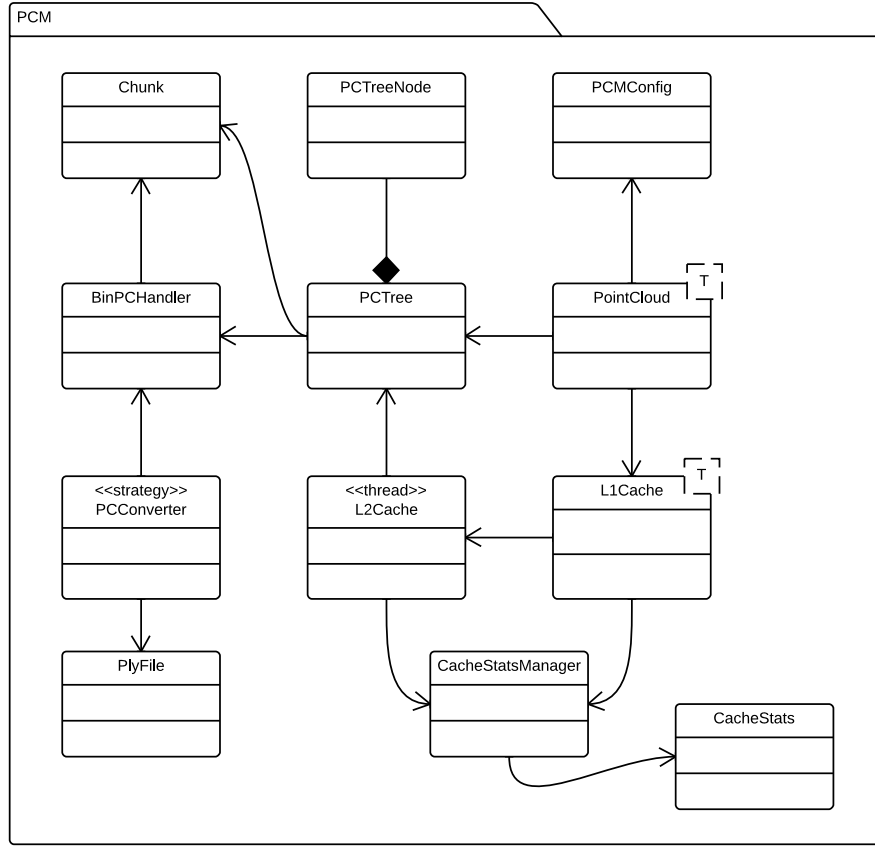


Figure 4.3: PCM class diagram.

4.2 Design

Both PCM and ToView have been designed following an object oriented approach, and having in mind C++ as the development language [Pra12]. Design patterns have been used as much as possible. The rest of this section briefly depicts the high-level design in the form of class diagrams. A more specific description is included in Chapters 5 and 6.

4.2.1 Class diagram

The design of the **PCM** library is depicted in the class diagram of Figure 4.3). In this design, **Chunk** represents the minimum amount of information that will travel across the different levels of memory, that is why its size in bytes will be closely controlled. **BinPCHandler** is the intermediate compacted format (BPC) in the format conversion process and the spatial structure construction, its objective is representing an arbitrary length array of data that resides in permanent storage. **PCConverter** is the class in charge of offering a single interface to convert any type of point cloud file.

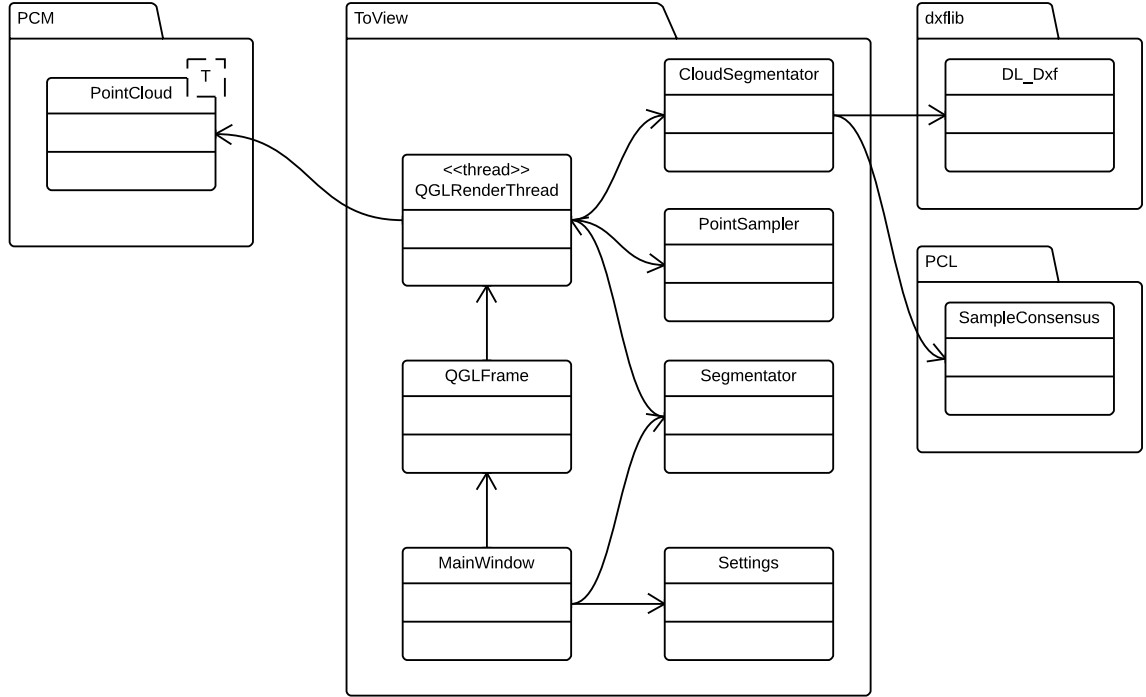


Figure 4.4: ToView class diagram.

The spatial structure is represented by **PCTree**. This class encapsulates two main functionalities, the construction of the chunk database and the multi-resolution tree management and its nodes. **PCTreeNode** contains information about the nodes that make up the spatial structure.

Next, the classes associated with memory management will be briefly described. **L2Cache** is the class responsible of the exchange of data between HDD and RAM. **L1Cache** is in charge of transferring data between RAM and VRAM. Both classes use **CacheStatsManager** to keep track of several statistics, like hit rate, miss rate, etc.

PointCloud represents a complete dataset and is the external interface of the library. It will allow the user to request information about the dataset, to load a region, apply a certain operation, etc. This class is accompanied by **PCMConfig** that stores several information about the characteristics of the point cloud.

On the other hand we have the visualizer **ToView**, whose design is shown in the class diagram of Figure 4.4. It is mainly composed by the following classes:

MainWindow is the class that represents the main window of the visualizer. It contains the navigation menus and the widget in which the rendering will be done. The **Settings** class is where all the information related to the configuration of ToView will be stored and provides the interface to modify it. **Segmentator** is in charge of providing the interface

for the user to be able to control and use the segmentation feature of the visualizer and modify its settings.

QGLFrame is responsible for creating the OpenGL context. It is also in charge of processing mouse and keyboard events. The rendering thread **QGLRenderThread** will also be launched from this class. As the name implies, this is the class that renders the point clouds.

PointSampler will allow the user to sample points from a point cloud to be able to measure distances or segment primitives later. **CloudSegmentator** is the class responsible for segmenting and exporting the supported primitives.

Chapter 5

Point Cloud Manager

PCM is a set of software tools and libraries that allow the user to manage massive point clouds. It comprises a new point cloud format, a multi-resolution spatial structure, a cache system and an external interface.

5.1 Design

PCM is also developed following an object oriented approach and using the C++ language. The compilers used are the same as in the visualizer. The technologies chosen are also the same with an addition. For GPGPU programming we have chosen OpenCL, because of its inter-operation capabilities with OpenGL. This will allow us to process directly data that already resides in the GPU, without having to do expensive memory transfers. Because of the open nature of OpenCL, the software will be able to run on any graphics card vendor (AMD, NVIDIA or Intel) and the SO that the user prefers (Windows, MacOS or Linux).

5.1.1 Dataset conversion and management

In the diagram of Figure 5.1, the following classes are the most relevant:

- **Chunk:** Minimum unit of data that will be managed by the different cache levels. Its size has to be strictly controlled because of this (no extra information just the basic information necessary to use it). Only has methods to read/write from/to disk.
- **BinPCHandler:** This class represents the binary format used in the format conversion process and the spatial structure construction. Its objective is to emulate a STL vector, but that resides in disk instead of system memory. This is necessary

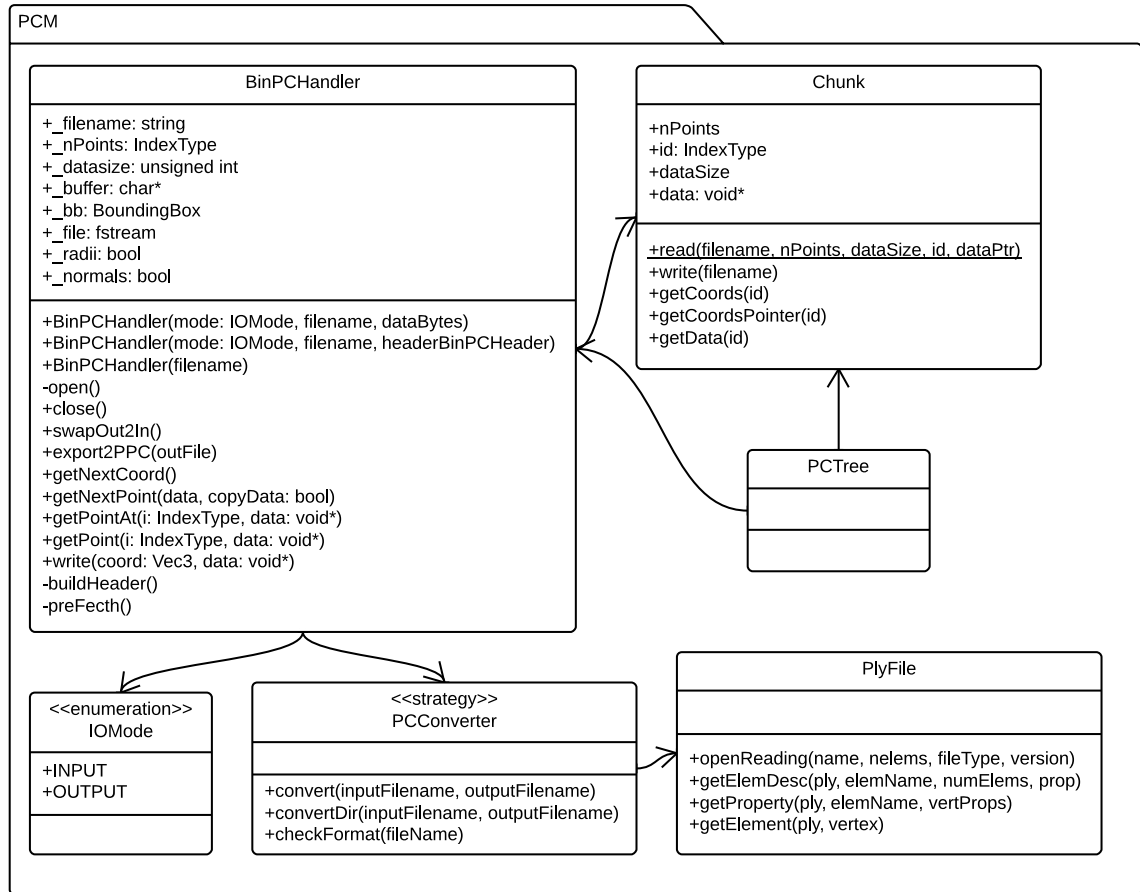


Figure 5.1: Class diagram of the dataset conversion and management classes.

because during the construction, operations like the calculation of the median will use the whole cloud that will not fit in RAM. With this class, the cloud can be used like if it would fit in primary memory. To improve the performance, it will *pre-fetch* and *buffer* data, after a certain number of requests. It will also allow writing and reading. This class has great performance when accessing data sequentially, but it is not as efficient for random access.

- **PCConverter:** This will be the class that will provide a single interface to convert any type of point cloud to BPC. The “strategy” pattern was used to create this class. It first checks the format of the input point cloud and then delegates the responsibility to the adequate method. It can convert a single file or a set of files stored in a folder.

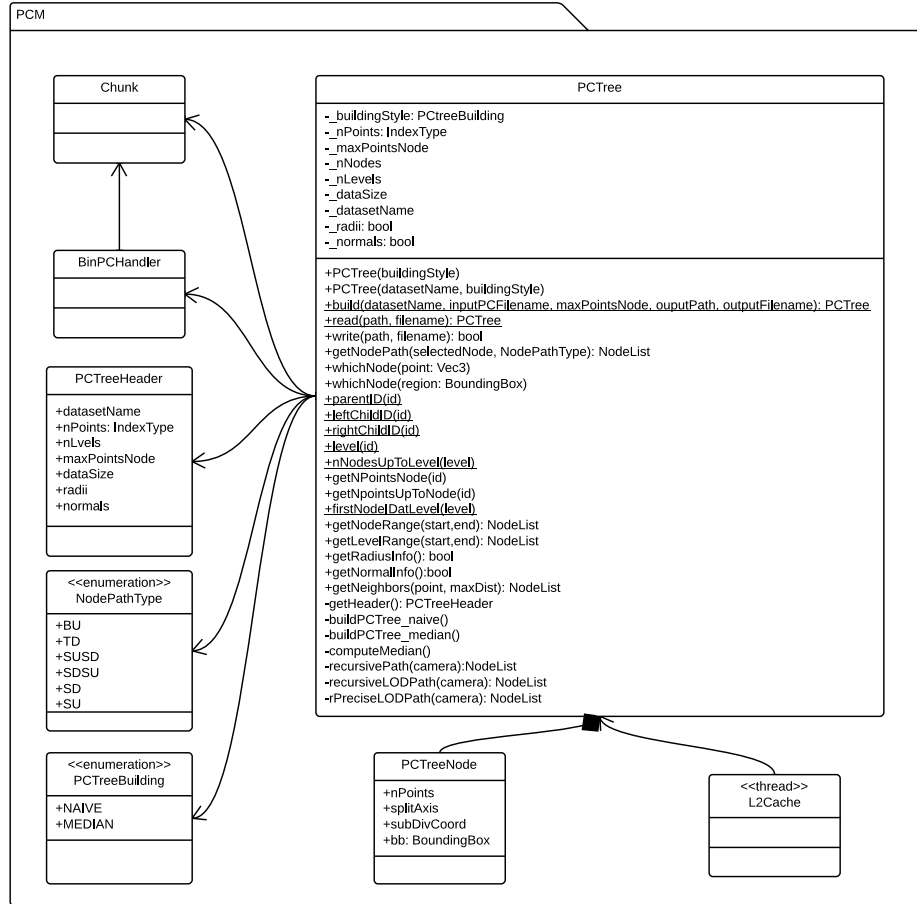


Figure 5.2: Class diagram of the spatial data structure.

5.1.2 Spatial data structure

PCM uses a spatial data structure (a kd-tree) as core of the multi-resolution system. The relevant classes are shown in the class diagram of Figure 5.2. The most important class is **PCTree**. This class encapsulates two of the main functionalities of the multi-resolution system:

- The construction of the chunk DB and the spatial structure itself. This is done as a pre-process, before using the library. This can be achieved using *PCGen*¹ or using the visualizer.
- Managing the multi-resolution tree, obtaining *NodePaths*² and performing tree traversals.

¹A provided command line utility.

²List of nodes to reach a certain area from the tree root.

As an implementation detail, it is relevant to remark that the tree nodes, represented by the class **PCTreeNode**, are all kept in a lineal vector for quick access ($O(1)$). There are also some functions valid for any binary tree related to the traversal of the tree; getting the child nodes of a parent, level of a node, etc. There are also functions to read/write the tree from/to disk. Finally, there are also functions to build priority node lists for the visualizer.

5.1.3 Memory hierarchy

The most significant classes in charge of the management of the memory hierarchy are shown in the shown in the class diagram of Figure 5.3:

- **L1Cache:** Responsible for loading/storing data in VRAM/RAM and the conversion of chunks in one or multiple VBOs. The class uses a variadic template to let the user choose the type of cache container (*_cache*). The *update()* method will take care of making the requests to the L2 and the VBO conversion using *load()* and *store()*.
- **L2Cache:** Is the class responsible for loading/storing data in RAM/HDD, it inherits from **Thread** so that it runs on its own thread. As we will mention in subsection 5.3.2, a Map (*_cache*) keeps references to every chunk that is loaded in the *_data* attribute. The request list is kept in the attribute *_priorityList*. Both structures are protected by *Mutex* and a custom *Semaphore*. The *run()* method contains the main loop.
- **CacheStatsManager:** Both of the previous classes contain an instance of this class. It is in charge of the monitorization of the caches, to get statistics of the cache status. Hits, misses, forced misses, load times, etc.

5.1.4 External interface

In Figure 5.4 the most important class is **PointCloud**, this class represents a complete dataset. This class will allow to load or save a dataset, request information about it, request to load an area or access to the spatial structure so that the user can make queries.

This class is used extensively by the visualizer to simplify the render process. It also provides several point cloud operations, like an statistical outlier removal filter, a voxel filter, radii estimation, etc.

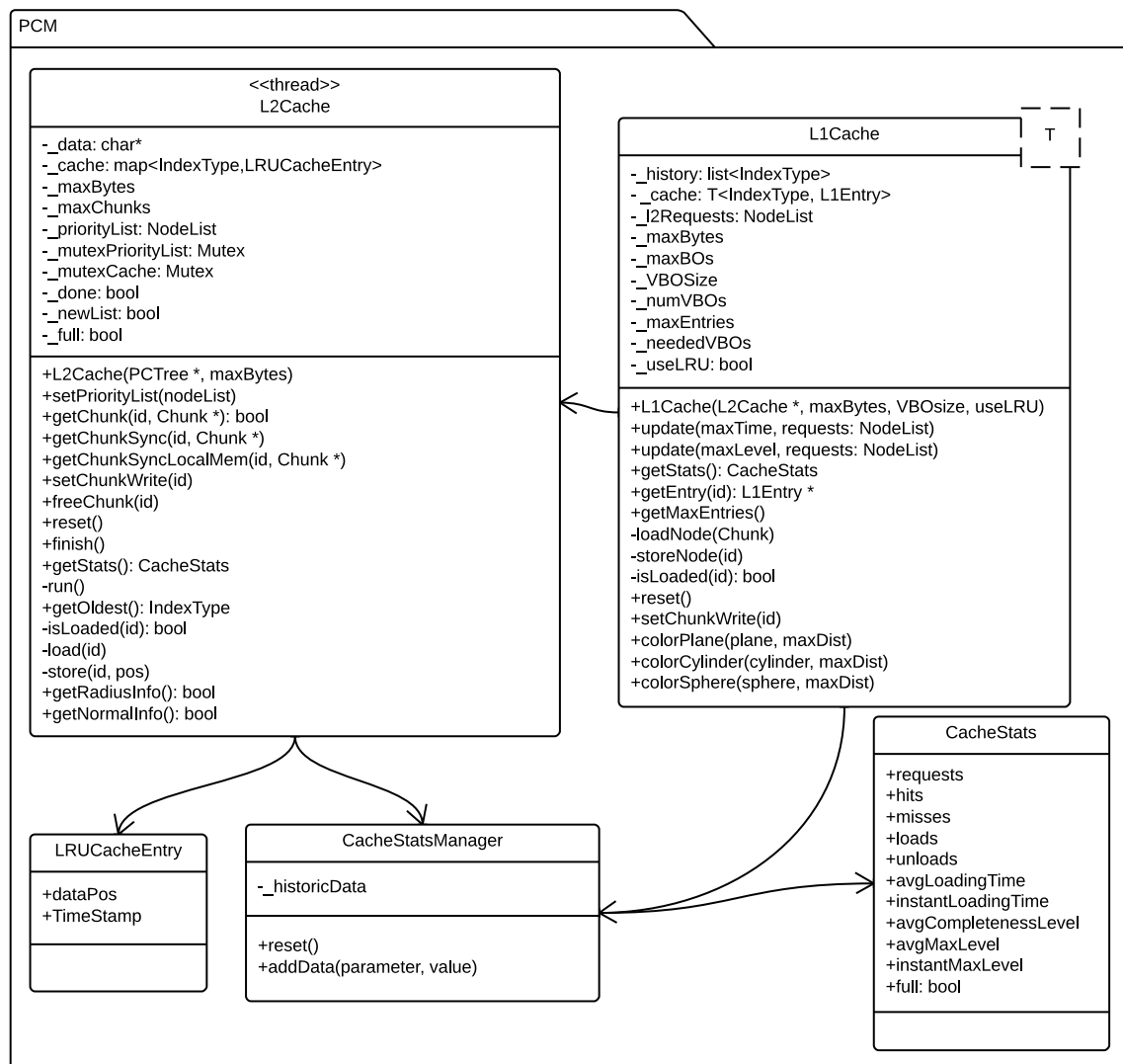


Figure 5.3: Class diagram of the memory hierarchy.

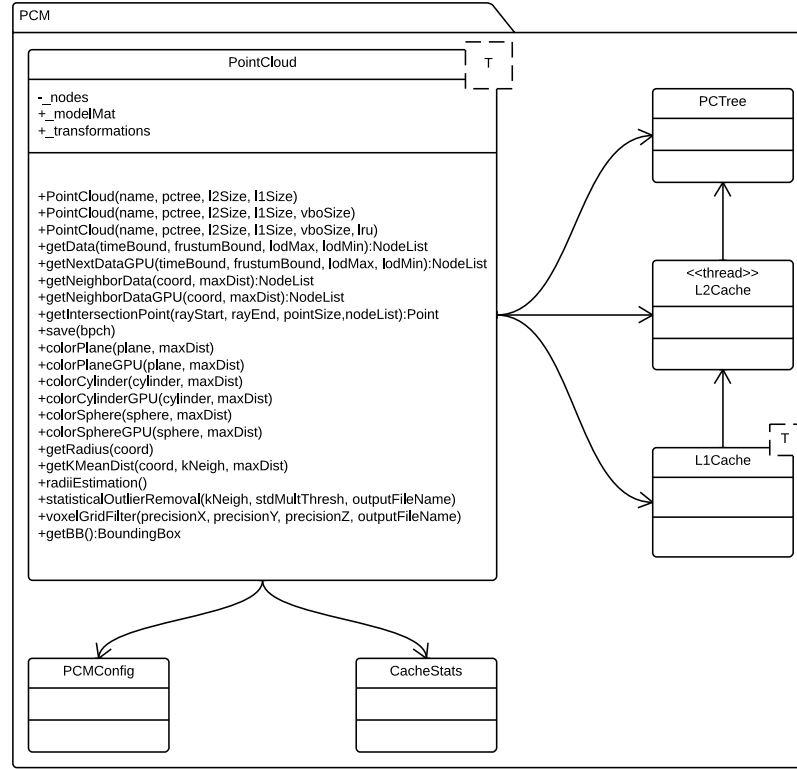


Figure 5.4: Class diagram of the external interface.

5.2 Input and output

This project will support a certain number of common point cloud data input formats. Furthermore, it can also output point cloud or geometric data in certain formats. In this section we will explain all the details related to these two subjects.

5.2.1 Supported point cloud formats

Several new point formats have been added to the capabilities of the point cloud converter utility. PCM will identify automatically the point cloud format and convert it to our own format transparently for the user.

The supported point cloud input formats are:

- **PTS:** Common ASCII point data format. The first line has the number of points to read, and the rest of the lines all the point data: $(x, y, z) + reflectivity + (R, G, B)$.
- **PTX:** ASCII laser scanning format. The first lines have information about the scan and the number of points scanned. The rest of the lines contain all the point data:

$(x, y, z) + reflectivity + (R, G, B)$.

- **ASC:** Common ASCII point data format in which every line contains information about each point in the cloud: $(x, y, z) + reflectivity + (R, G, B)$.
- **LAS:** Binary laser scanning format, it contains all sorts of information about the scan and the point data that can include: $(x, y, z) + reflectivity + (R, G, B)$.
- **PLY:** PLY is a computer file format known as the Polygon File Format or the Stanford Triangle Format. It can be binary or ASCII, and the project supports both. It contains the following information: $(x, y, z) + (R, G, B) + (n_x, n_y, n_z)$.
- **PCD:** The point cloud format of one of the most used point cloud libraries, **PCL** [Gar13]. As the preceding format, it can be binary or ASCII. Each point can have: $(x, y, z) + (R, G, B)$.
- **BPC:** Our own binary point cloud data format. It stores information about the cloud and individual points. Each point can have: $(x, y, z) + (R, G, B) + radius + (n_x, n_y, n_z) + etc$.

Because of the size that this files can reach, they normally are split in several smaller files that range from 500 MB to 1,5 GB. Because of this reason, the ability to read multiple point cloud files in a folder automatically will also be added.

5.2.2 Binary Point Cloud

The point cloud format created for this project (BPC) will now be detailed. It is a binary format that is comprised of a **header** and the **point cloud data**.

The header will contain the following information:

- **Number of points:** The number of points contained in the file.
- **Data size:** Size of the data that accompanies the point (normals, radii, etc.).
- **Bounding box:** Information of the bounding box that contains the point cloud.
- **Radii:** Indicates if the dataset contains point radii.
- **Normals:** Specifies if the point normals are available.

The rest of the file will contain the point cloud data. The data will be stored contiguously point after point as can be seen in Figure 5.5 to minimize the file size.

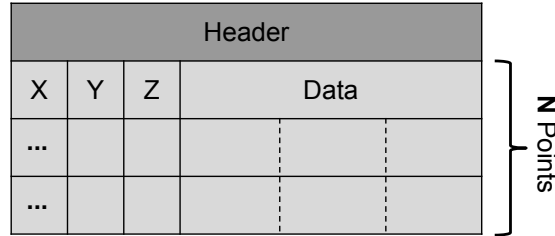


Figure 5.5: Chunk structure.

A BPC file is treated like a single chunk that contains the whole cloud, so all of this information is also valid for chunks. A chunk will be the minimum amount of information that the memory hierarchy will deal with. When rendering, the whole cloud will not be stored in a single chunk, since this would not be efficient. Instead, the points in the nodes of the multi-resolution structure will each be stored in a chunk, and the cloud will be an aggregate of all the chunks (see Figure 5.6).

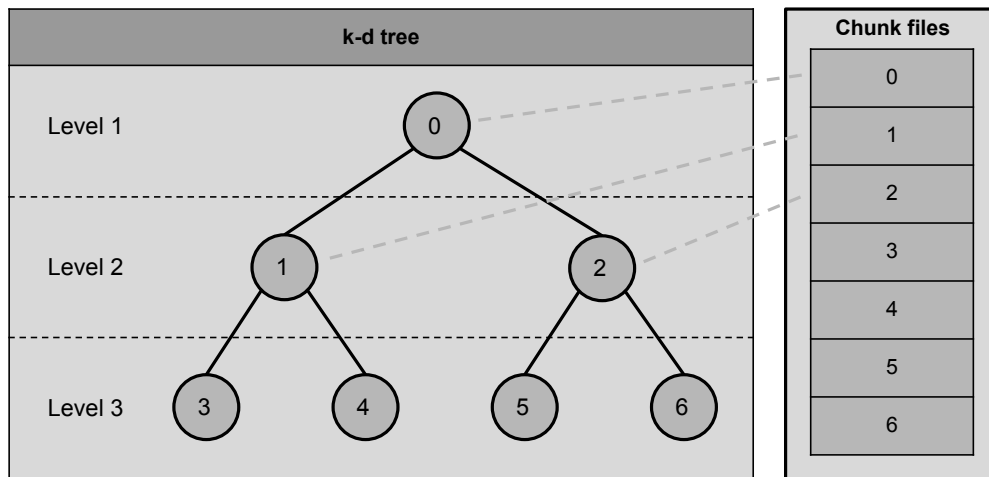


Figure 5.6: Internal tree structure. Each node is stored in a chunk file.

The only big difference between a BPC file and a chunk, is that because the size of a BPC file will be much greater. Therefore it will not be possible to load the complete file at once, and we will rely on an interface class, [BinPCHandler](#). This interface works in a similar manner to an array in system memory, but accessing permanent memory in an efficient way (this class will be further explained in the subsection 5.1.1).

5.3 Memory hierarchy

One of the principal concepts in PCM is the memory hierarchy management, that is made up of two levels of cache. When it is known that the data that is going to be used is related in some way and the different access types, software caches can yield substantial improvements in the performance of the system. In this case, we work mainly with geometric information obtained from real environments and the cache hierarchy will allow us to exploit the spatial coherence in the data.

In this project, a cache system with two levels has been implemented. It will take care of the data transfer needs between memory levels (HDD - RAM - VRAM). This system will take into account the spatial relation between the data, and will have an architecture that will not only allow to store datasets in a HDD but anything with a filesystem (e.g. NFS from a network).

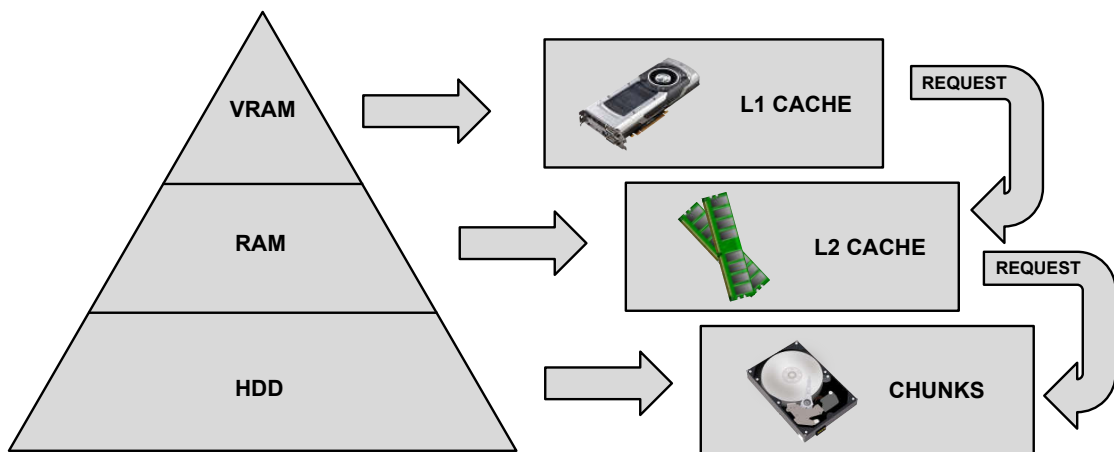


Figure 5.7: Memory hierarchy of PCM.

Between these three levels of memory will circulate the minimum unit of information, the chunk. A chunk is a “container” of a data subset, in our case points and data related to them.

The three levels are:

- **Video memory or VRAM:** Volatile memory and less capacity than the next level, but even faster. Only the GPU will use this type of memory. The L1 cache will be created in this memory level.
- **Primary memory or RAM:** Volatile memory and low capacity compared to the

next level, but fast. The CPU will use this type of memory. The L2 cache will reside in this cache level.

- **Secondary memory or HDD:** Persistent memory and high density, but slower than the other two levels. In this level of memory will reside the point database.

First, in the lowest level, reside the binary files of each chunk. In the next level, RAM memory, a subset of chunks will be stored. Finally in VRAM, the chunks are adapted to a video memory compatible format, so that the GPU can work with them.

These concepts will be further explained in the next subsections.

5.3.1 L1 cache

The first level cache (L1) is in charge of managing the storing and loading of data between primary memory (RAM) and video memory (VRAM). This cache level has been completely reworked, and also has been given the ability to not only load but also store information in primary memory. Even having a similar structure to the L2 cache, because of the characteristics of GPU processes, its operation is synchronous.

Its biggest responsibility, apart from the corresponding memory hierarchy management, will be the conversion of chunks of data to *VBOs*³ in VRAM and viceversa. A chunk may have to be broken up into several VBOs depending on its size, so that optimal sizes of VBOs can be maintained for the GPU.

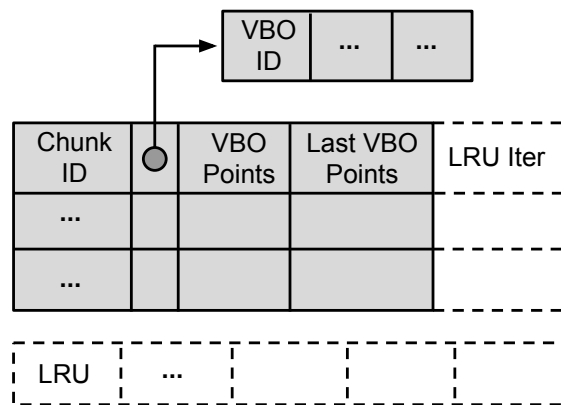


Figure 5.8: L1 cache structure.

The following list will explain each of the fields that make up the cache structure (see Figure 5.8):

³ *Vertex Buffer Object*: Structure that allows to store arrays of data in video memory.

- **Chunk ID:** Id of the chunk stored in L1 cache.
- **VBO ID:** IDs of the VBOs used to store points in the L1 cache entry.
- **VBO Points:** Number of points stored in all but the last VBO in the cache entry.
- **Last VBO Points:** Number of points in the last VBO in the cache entry.
- **LRU Iter:** Iterator to cache entry in LRU access record for fast updating of LRU.
- **LRU:** Least Recently Used chunk.
- **Write:** Flag that marks chunk for storing.

Also two replacement policies were implemented:

- **Naive:** Erase any data block that is not necessary looping over the complete cache one time. This method does not take into account when was the last time that a block was last used, if it is not necessary it will be erased.
- **LRU** ⁴: This algorithm, will replace the elements that have been used least recently if the cache is full. If not full it will keep inserting nodes until it is.

To store the list of nodes currently in the cache two STL ⁵ data structures can be used:

- **Map:** An associative container with key-value pairs with ordered unique keys. Complexity of searches and element access is logarithmic in size: $O(\log n)$.
- **HashMap:** It also contains key-value pairs with unique keys but without an order, element access is done using a hash function. Complexity of searches and element access is constant in the average case: $O(1)$ and linear in size in the worst case: $O(n)$.

Either of these two structures will meet our objective of making cache searches fast, as this is a crucial function that will be frequently used. Theoretically the use of one or the other depends on the number of nodes that the cache will contain. A Map will be faster with less elements, while the HashMap will be faster with big quantities of nodes.

Another implemented feature are types of cache requests, there are several available:

⁴*Least Recently Used*

⁵*Standard Template Library*

- **Restricted by time:** In order for the render process to be interactive and that no freezes occur, we will limit loading of VBOs by time. This type of petition will be limited by the time that the user desires (e.g. 3 ms). The cache will load as much nodes as possible in the given time and if the maximum time is reached it will stop loading nodes, even if there are still nodes missing.
- **Restricted by level:** This type of request is usually used in GPGPU calculations, where no interactivity is needed. The user will specify a level of precision and the cache will load as many nodes as necessary for that level of detail.

This cache not only can read and load nodes from L2, it also has the ability to store changes made in video memory (L1) in RAM (L2). The user can choose if he wants the changes made in GPU (e.g. an OpenCL kernel) to be persistent or not.

When the node is evicted from the cache, its changes will be written to the L2 cache prior to its deletion. In the Listing 5.1 we not only show how to write changes to other cache levels, but also how to use PCM with its new **PointCloud** interface to loop over the complete cloud out of core and perform some custom operation.

This process is completely transparent for the user, that will only have to use this example code:

```
PointCloud cloud (...);

while (cloud.getNextDataGPU(list)) {
    for (auto& node: list) {
        auto * VBOdata = cloud.getL1Entry(node);
        if (VBOdata) {
            makeChangesGPU(node);
            cloud.setChunkWriteL1(node);
        }
    }
}
```

Listing 5.1: How to write data to L1 chunk.

5.3.2 L2 cache

The second level cache L2, will be in charge of transferring data (chunks) from disk to system memory (RAM). The replacement policy used in this cache is LRU [Tai09] as this is the most adequate policy for these types of systems.

To better use the resources available nowadays (multi-core CPUs), this cache will run on its own thread. Also since the massive datasets that we will deal with require a lot of computational power for even the simplest operations, this cache will be prepared to deal with simultaneous requests from different work threads. All of this is achieved with careful use of *mutex*⁶ and *semaphores*⁷.

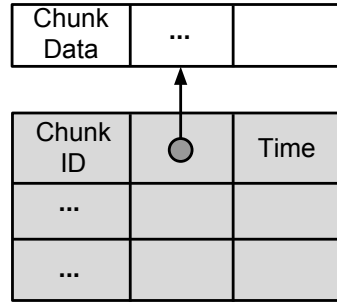


Figure 5.9: L2 cache structure.

The following list will explain each of the cache structure fields (see Figure 5.9):

- **Chunk ID:** Id of the chunk stored in L2 cache.
- **Position:** Position of the chunk in the structure that contains all of them.
- **Time:** Time in which the node was last accessed.

Next, the responsibilities of the L2 cache are:

- Load petition lists from disk that are requested by the upper cache level (L1).
- Keep a “repository” of recently loaded chunks, that because of the nature of the spatial structure, will be utilized again with a high probability.
- Ask the spatial structure which files correspond to the requested nodes.

⁶A synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.

⁷A variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a parallel programming or a multi user environment.

- Keep memory coherence of the system, with a Map that matches node IDs with their corresponding chunk in memory.
- If the user desires it, write changes to disk.

The main loop can be summed up in the following actions. First, it waits until a new petition arrives, when it does, it loads the requested nodes in order. If during the loading of the list a new one appears, the old one is discarded and the process begins again.

This cache can work in one of two ways:

- **Synchronous:** A node is requested and loaded immediately.
- **Asynchronous:** A list of nodes is requested and they are loaded in order as soon as possible.

This cache is capable of not only reading data from the HDD, but also writing modified data from RAM to disk. This is achieved in a similar fashion to the L1. A complete loop over the whole cloud to perform an operation using the CPU and then writing it to disk, can be seen in Listing 5.2.

```
PointCloud cloud (...);

while (cloud.getNextData(list)) {
    for (auto& node: list) {
        Chunk chunk;
        cloud.getChunkL2(node, chunk);
        makeChanges(chunk);
        cloud.setChunkWriteL2(node);
        cloud.freeChunkL2(node);
    }
}
```

Listing 5.2: How to perform an operation using the L2 and store the changes.

The cache hit rate will be really high, because of the topology of the spatial structure. If an algorithm needs to reach the maximum level of detail in different zones, because the levels are additive, it will have to add up all the information contained in the node path

from the root to the leaf. But in regions of space that are close, that means they will share a lot of nodes (see Figure 5.10).

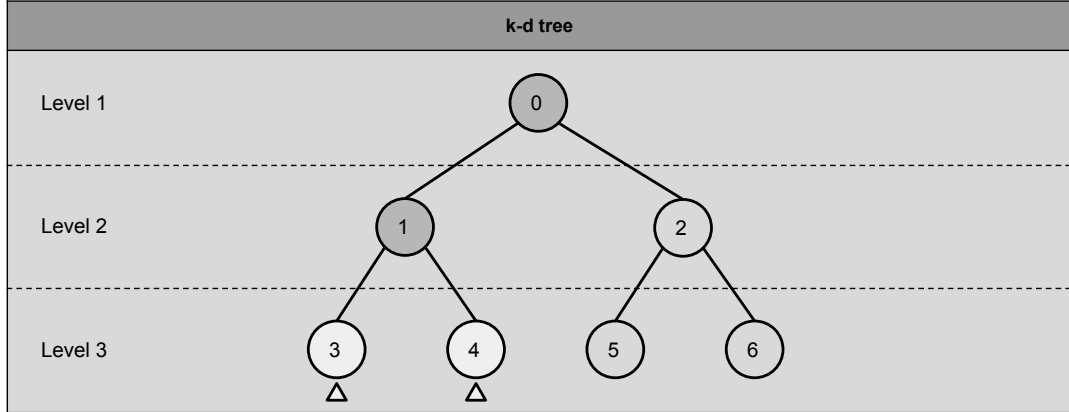


Figure 5.10: Two node paths spatially close (3 and 4), in order to reach the maximum level of detail (level 2) share nodes 0 and 1.

5.3.3 PCM Benchmark

Since we wanted to have a quick and convenient way to test PCM, we created this utility in *Python* to achieve this purpose. PCMBenchmark is a Python benchmark suite that automatically tests PCM and outputs the corresponding metrics for the user to see or use.

In order for PCMBenchmark to work it uses a certain directory structure:

- **bin:** With **debug** and **release** sub-directories that contain the corresponding PCM executables (user creation required).
- **dataset:** Contains the test dataset (user creation required).
- **data:** Automatically created directory that contains testing data. This is raw data just in case the user will want to process the data on its own.
- **trees:** Automatically created directory that contains automatically generated k-d trees.

Once the user has the directory structure created, its just a matter of copying and pasting to test PCM in different machines. The process will be completely transparent for the user that will only need to check the resulting graphs (see Figure 5.11).

The outputs that the benchmark suite creates are:

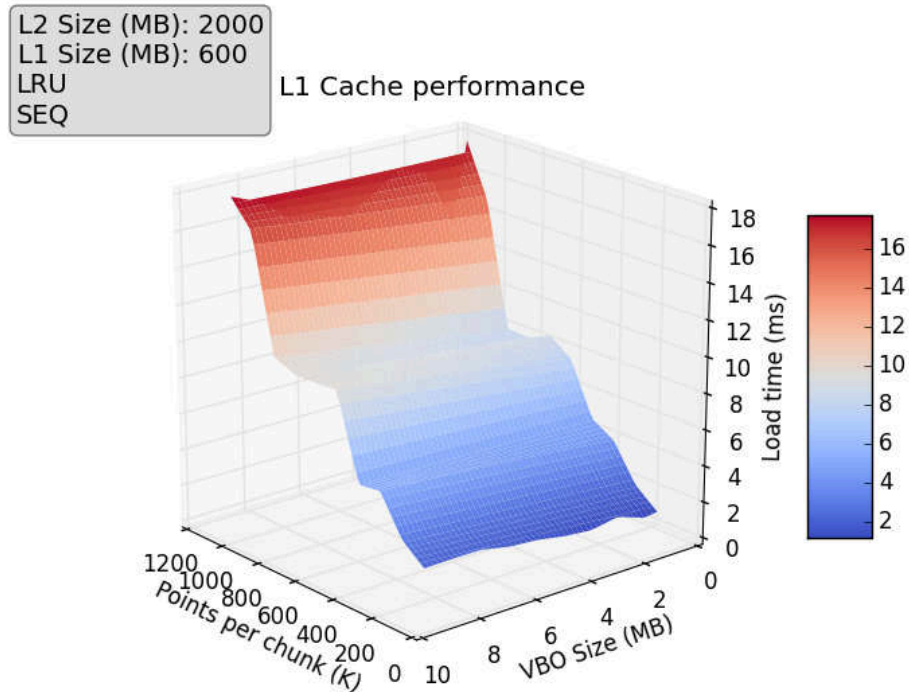


Figure 5.11: A sample output graph created by the benchmark suite.

- **Graphs:** Available in the root directory as a series of .png files. This is the most user friendly output of the tool, a series of graphs that show the metrics in a graphical sense.
- **System information:** Stored in a text file called *sys_info.txt* that contains common system info.
- **Raw data:** Automatically generated data to create the graphs. Suitable for further processing by an advanced user.

The only system requirements are:

- **Python 3.3**
- **Matplotlib**
- **NumPy**

With this tool the testing in multi-platform environments is greatly simplified.

5.4 Results

Two computers with different hardware were used for testing. The first, **Computer 1** has the following specs:

- Intel Core i7-3770 CPU (4 cores, 8 threads)
- NVIDIA GT 640 graphics card
- 16 GB of 1600 MHz DDR3 RAM
- WDC WD10EZRX HDD

The second, **Computer 2** has the following specifications:

- Intel Core i7-4930K CPU (6 cores, 12 threads)
- NVIDIA GTX 780Ti graphics card
- 16 GB of 2133 MHz DDR3 RAM
- WD Caviar Black HDD

The operating system used to run the tests was Windows 7 x64, with the VC++ v110 compiler. The libraries used were:

- OSG 3.0.1
- Freeglut 2.8.1
- GLEW 1.10.0
- PCL 1.7.1

The datasets that will be used for testing are described in Table 5.1.

In the first test (see Figure 5.12), we will estimate the radii using the CPU (using all threads available) and GPU. We can observe that as the chunk size decreases, the estimation is faster. This is because there are less intersection tests to be performed, since a lot of nodes can be discarded as they will not belong to the neighborhood of each point.

It is also interesting to highlight the fact that the GPU is slower than the CPU, this happens due to the setup of the OpenCL kernel. The transferring of information from RAM

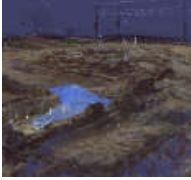
	Dataset	Points	Features
	Castelao	$\sim 500K$	Color, Normals
	Dorna	$\sim 3M$	Color
	Oficina	$\sim 30M$	Color
	Lourizan	$\sim 90M$	Color
	CALD	$\sim 200M$	Color
	Baron	$\sim 500M$	Color
	Castelo	$\sim 1000M$	Color

Table 5.1: Datasets used in our tests.

to VRAM and the kernel setup calls outweigh the benefits of the GPGPU implementation. This means, that the computation of the radii in GPU is faster, but when dealing with a relatively cheap operation like the former, the setup of the kernel takes too much time compared to the kernel execution time and makes the GPU lose against the CPU.

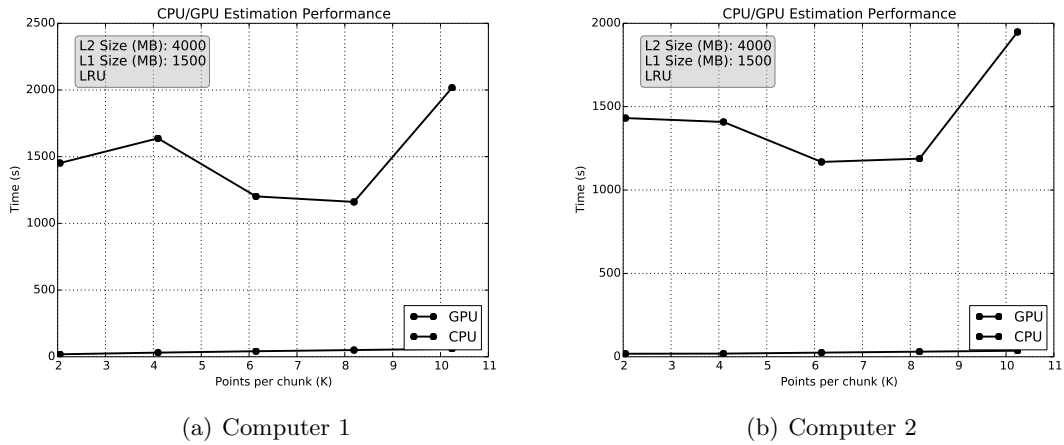


Figure 5.12: Graphs showing the estimation times in the Castelao dataset.

The next test also compares the radii estimation times in the two test machines, but this time with the Dorna dataset (see Figure 5.13).

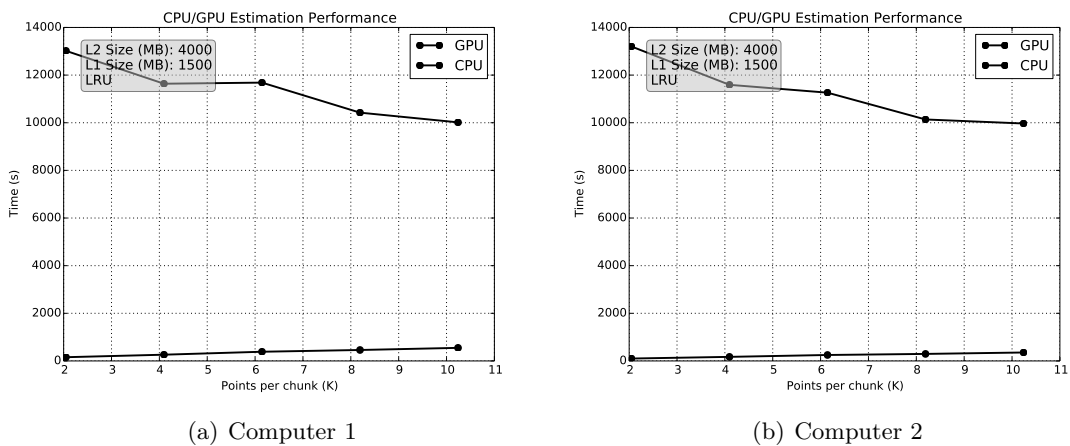


Figure 5.13: Graphs showing the estimation times in the Dorna dataset.

In addition to this compute tests, cache performance will be tested on the most demanding datasets available. The first dataset that will be tested is Lourizan. In Figure 5.14, we can observe how L1 cache hits evolve depending on chunk size or VBO size.

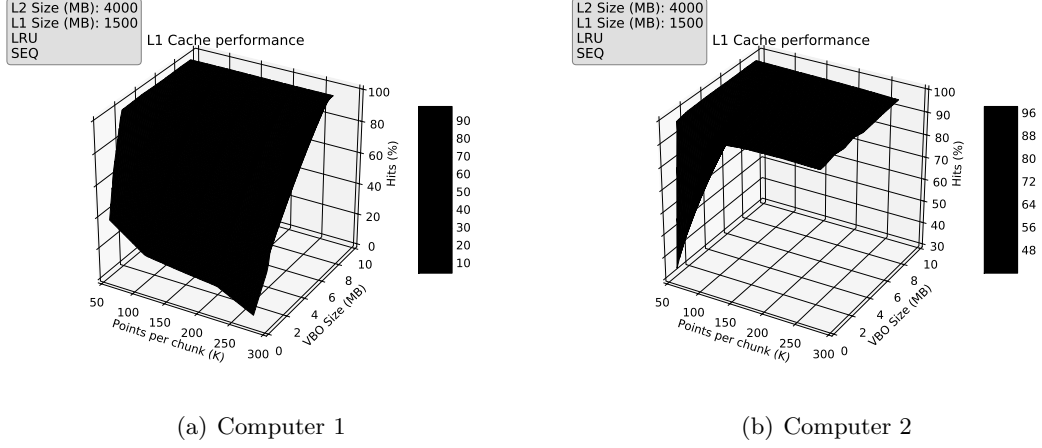


Figure 5.14: Graphs showing how hit rate on L1 behaves depending on chunk size and VBO size in Lourizan.

We can see that the cache behaves better when using big VBO sizes, but this is not the whole picture. To have a real idea of the optimal cache parameters, we need to take a look at Figure 5.15. This test compares load times against chunk sizes and VBO sizes.

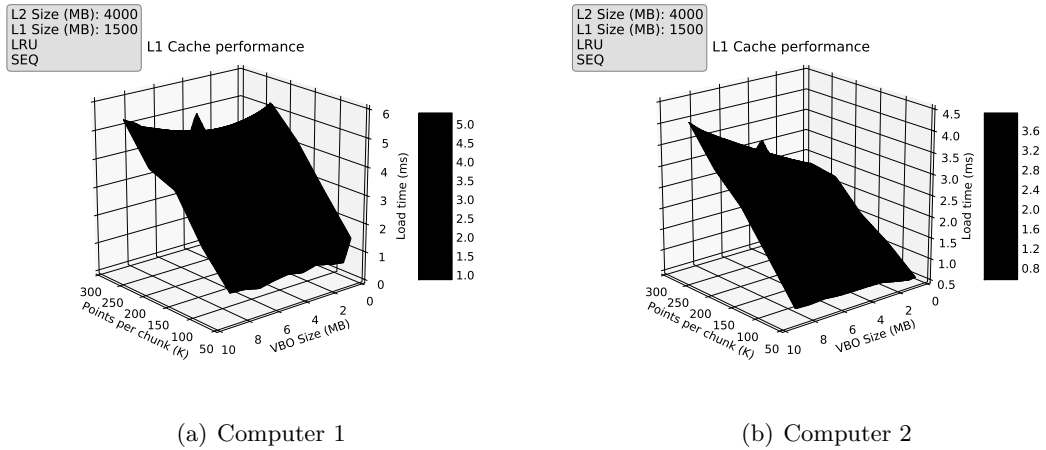


Figure 5.15: Graphs showing how load times on L1 evolve depending on chunk size and VBO size in Lourizan.

What can be deduced from this test, is that loading bigger chunk sizes in GPU memory takes more time in general, this was an expected result. What is most interesting about this second test, is the fact that we can see there is a sweet spot between 1-4 MB when loading VBOs. The loading time increases as we augment the VBO size, when trying to load the same amount of information. This is why we cannot just use the biggest VBO size available to increase hit rate.

Summing up, the optimal parameters are using an intermediate chunk size, and a VBO size between 1-4 MB. Besides testing the L1 cache, we have also tested the l2 cache. In this case, the first test performed compares the hit rate depending on the chunk size chosen (see Figure 5.16).

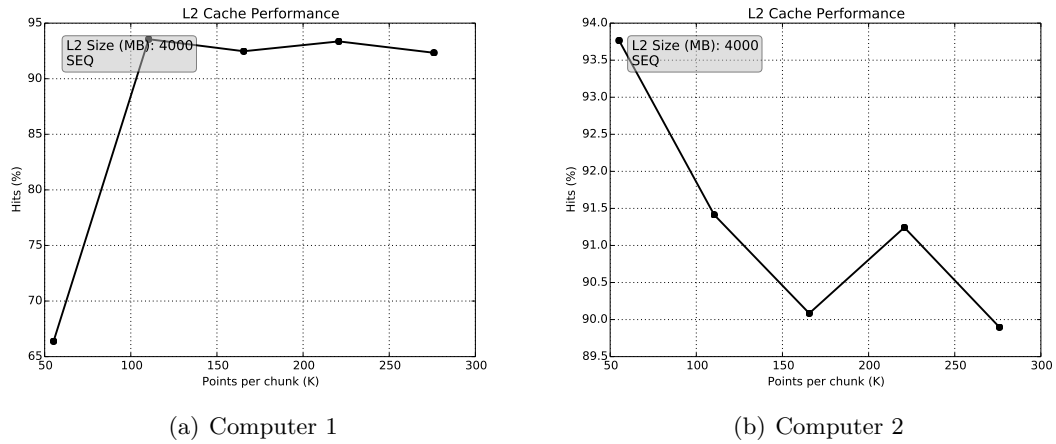


Figure 5.16: Graphs showing how hit rate on L2 evolves depending on chunk size in Lourizan.

On the aforementioned test, that for some chunk sizes the faster machine had worse results came as a surprise. As we will see in Figure 5.17, since the cache is never full using this configuration, this may be one of the reasons the weaker machine was faster on some cases. But taking all tests into account, L1 and L2, the faster computer still comes out on top.

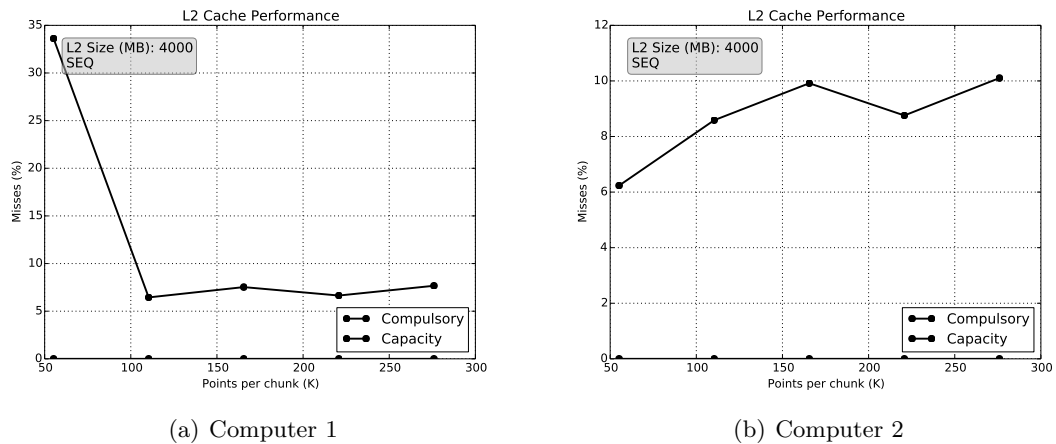


Figure 5.17: Graphs showing how the miss rate on L2 evolves depending on chunk size in Lourizan.

The last test performed, compares L2 loading time depending on chunk size (see Figure 5.18). As we can see in the graph, the optimal chunk size depends on the machine. But it can be said with some confidence, that a size of $\sim 100K$ points is a safe bet when choosing the chunk size.

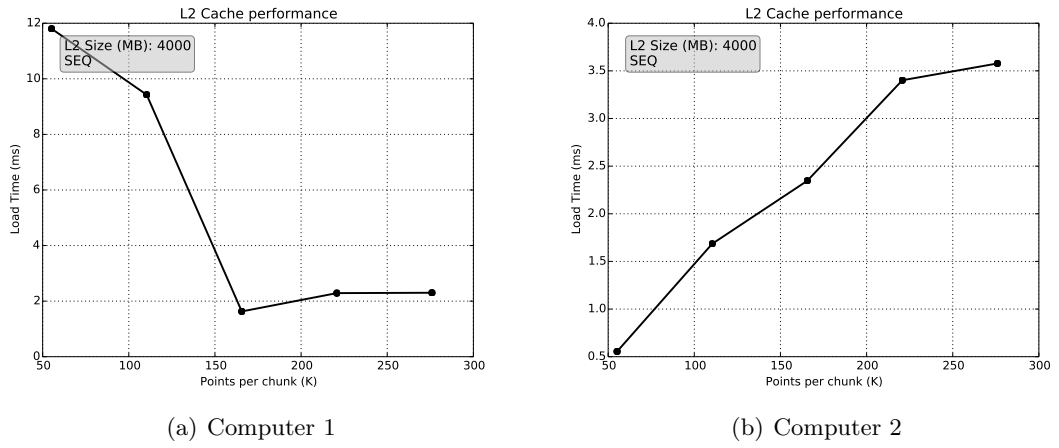


Figure 5.18: Graphs showing how the load times on L2 vary depending on chunk size in Lourizan.

The next dataset tested was CALD, in which the same tests were performed. Therefore, the first test performed was Figure 5.19, in which we can see how L1 cache hits evolve depending on chunk size and VBO size.

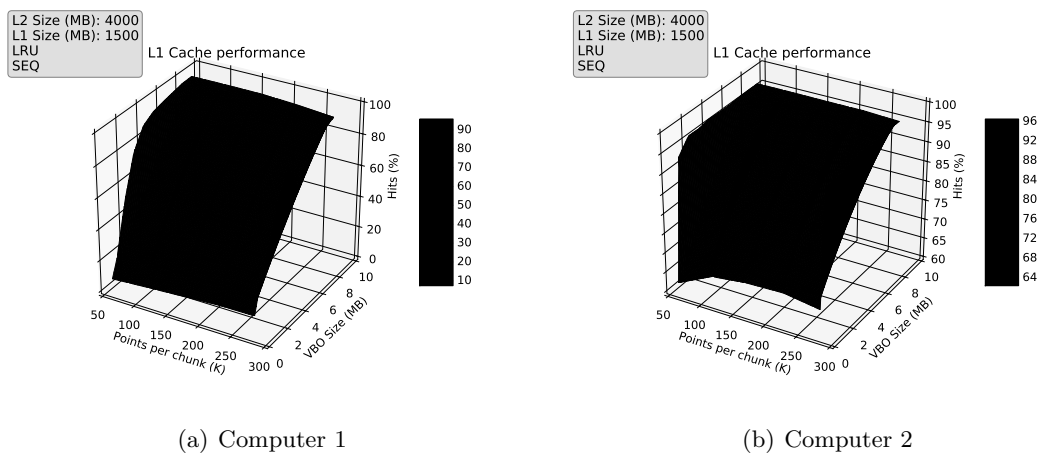


Figure 5.19: Graphs showing how hit rate on L1 behaves depending on chunk size and VBO size in CALD.

Next, the test that compares load times against chunk sizes and VBO sizes (see Figure 5.20). The tests performed in L1 corroborate the results of the tests performed with

Lourizan. The only difference that can be seen, is the fact that in Lourizan the faster computer was able to load every chunk requested with a big array of configurations (see Figure 5.14 and Figure 5.19). With this dataset the graph resembles more the graph of the slower machine.

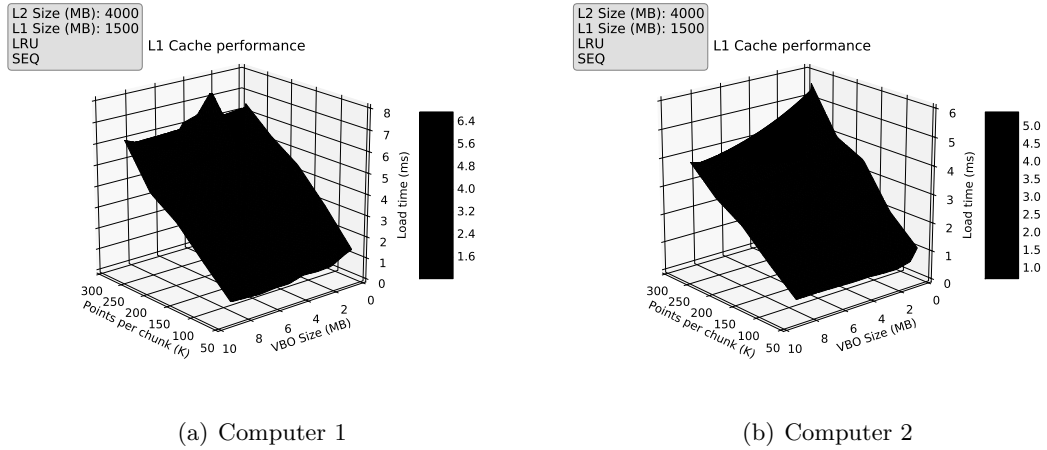


Figure 5.20: Graphs showing how load times on L1 evolve depending on chunk size and VBO size in CALD.

The next test performed compares the hit and miss rate depending on the chunk size chosen in L2 (see Figure 5.21 and Figure 5.22).

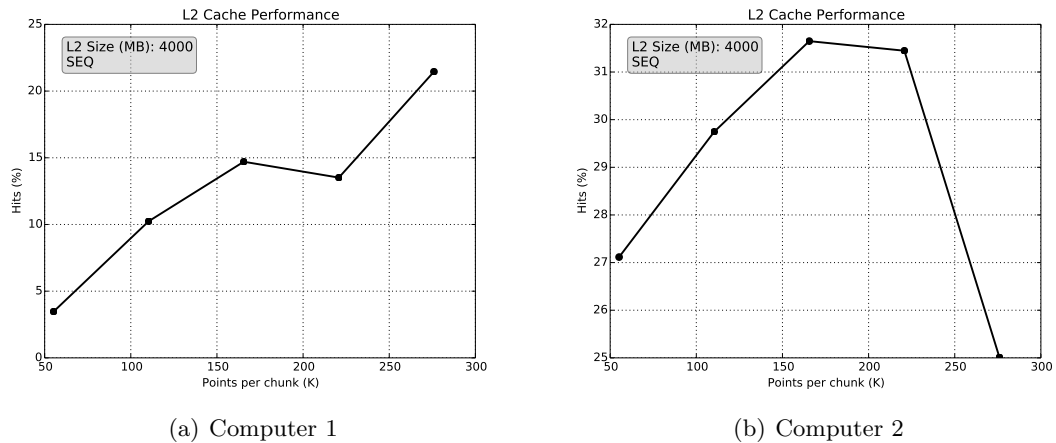


Figure 5.21: Graphs showing how hit rate on L2 evolves depending on chunk size in CALD.

These results mimic the last results obtained when using the Lourizan dataset. Although, these are not the best datasets to test the L2 cache, since they are not big enough to fill the cache. The two remaining datasets will be true stress tests for the L2 cache when

using this configuration.

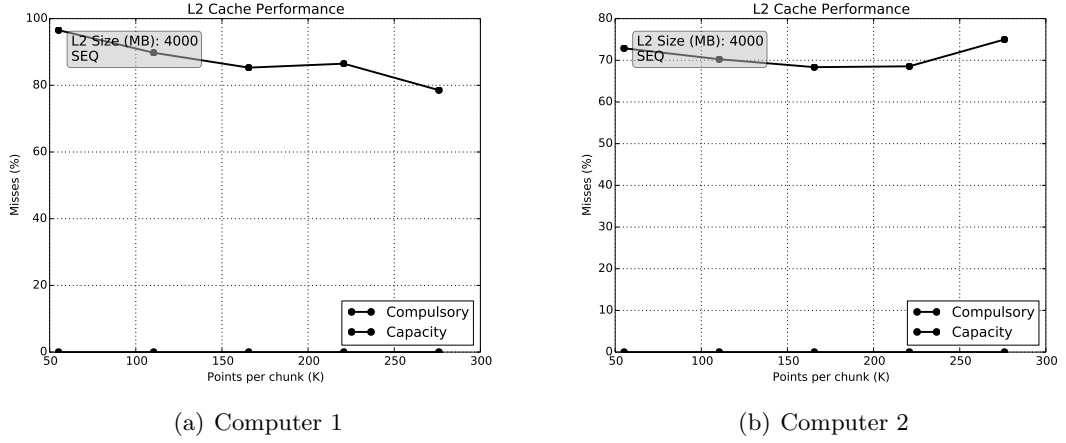


Figure 5.22: Graphs showing how the miss rate on L2 evolves depending on chunk size in CALD.

The last test compares L2 loading time depending on chunk size (see Figure 5.23). As we can see in the graph, the stronger machine is faster for every chunk size. The only chunk size in which the weaker machine can compete is the biggest one tested.

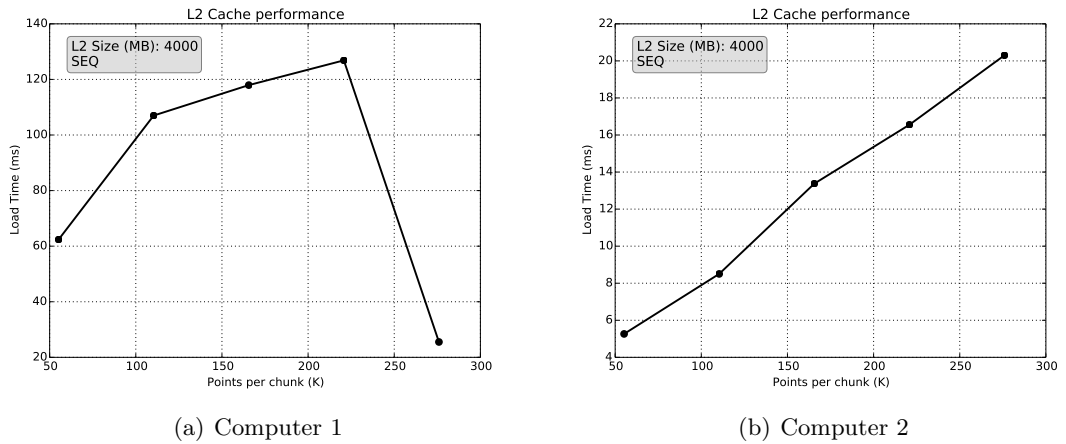


Figure 5.23: Graphs showing how the load times on L2 vary depending on chunk size in CALD.

The next dataset tested was Baron, in which the same battery of tests were performed. As a consequence, the first test performed was Figure 5.24, in which we can see how L1 cache hits evolve depending on chunk size and VBO size.

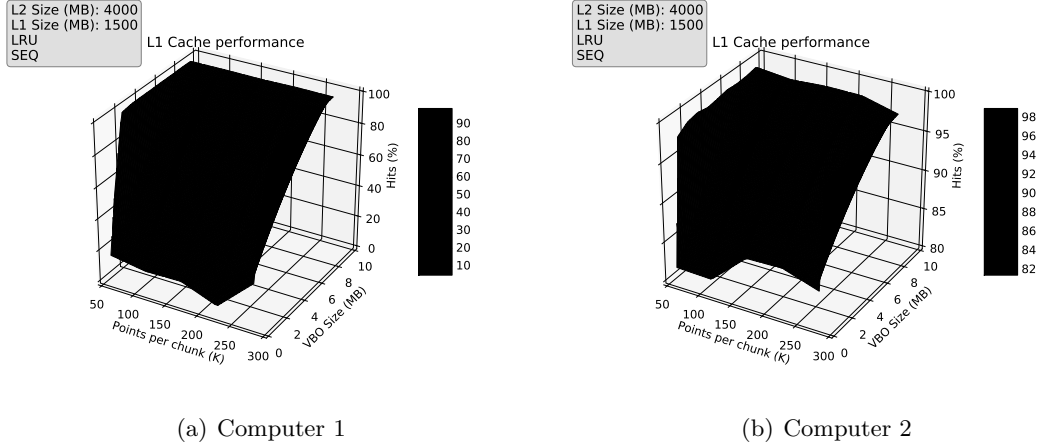


Figure 5.24: Graphs showing how hit rate on L1 behaves depending on chunk size and VBO size in Baron.

Next, the test that compares load times against chunk sizes and VBO sizes (see Figure 5.25). The tests performed in L1 corroborate the results of the tests performed with the Lourizan and CALD datasets. No further conclusions have been reached with this new dataset.

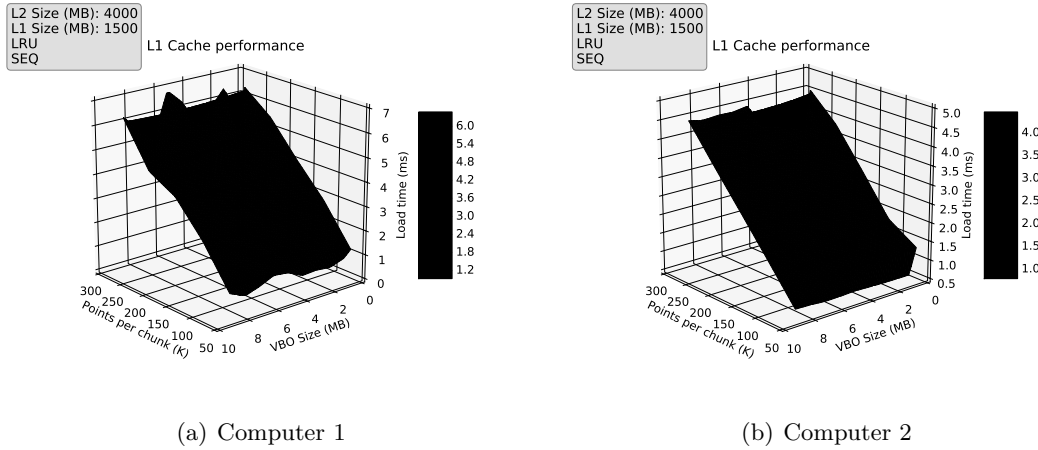


Figure 5.25: Graphs showing how load times on L1 evolve depending on chunk size and VBO size in Baron.

The next test performed compares the hit and miss rate of the L2 cache depending on the chunk size chosen (see Figure 5.26 and Figure 5.27).

These results mimic the results obtained when using the Lourizan and CALD datasets. The stronger machine clearly has a big advantage, that when dealing with bigger datasets

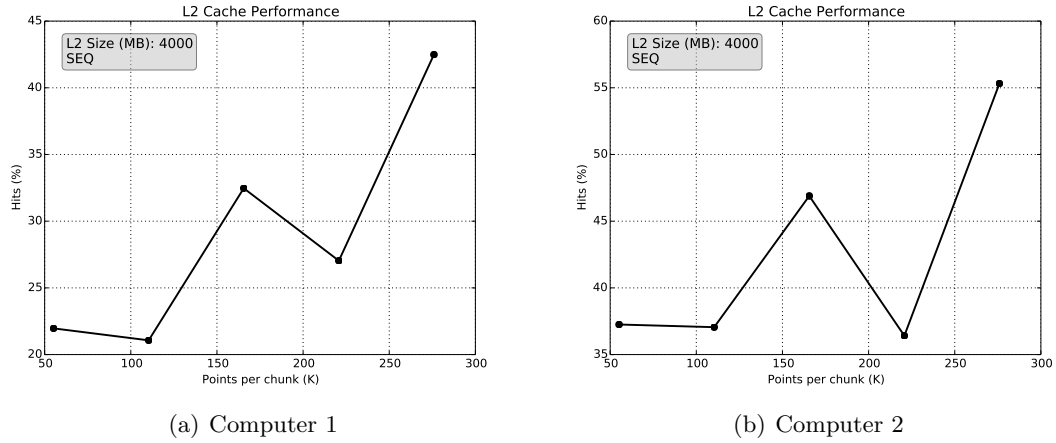


Figure 5.26: Graphs showing how hit rate on L2 evolves depending on chunk size in Baron.

increases even further. But this is the first dataset where 4 GB of memory is not enough to store the complete dataset. This is the reason why we start to see a lot of capacity misses, that eventually surpass the compulsory misses. This is the first big test for our L2 cache, that even when full keeps an acceptable hit rate.

The last test compares L2 loading time depending on chunk size (see Figure 5.28). As we can see in the graph, the stronger machine is faster for every chunk size. The weaker machine is not able to compete anymore when dealing with the Baron dataset.

To sum up, the gap between Computer 1 and Computer 2 gets bigger as the dataset sizes increase. This behavior was expected since the tree gets bigger and cache performance becomes more crucial. The bigger 12 MB cache in the CPU also should help the faster machine achieve better results.

The last dataset tested was Castelo, in which once more, the same battery of tests were performed. Consequently, the first test performed was Figure 5.29, in which we can see how L1 cache hits change depending on chunk size and VBO size.

Next, the test that compares load times against chunk sizes and VBO sizes (see Figure 5.30). The tests performed in L1 corroborate the results of the tests performed with the Lourizan, CALD and Baron datasets. No further conclusions have been reached by testing this new dataset.

The next test performed compares the hit and miss rate of the L2 cache depending on the chunk size chosen (see Figure 5.31 and Figure 5.32).

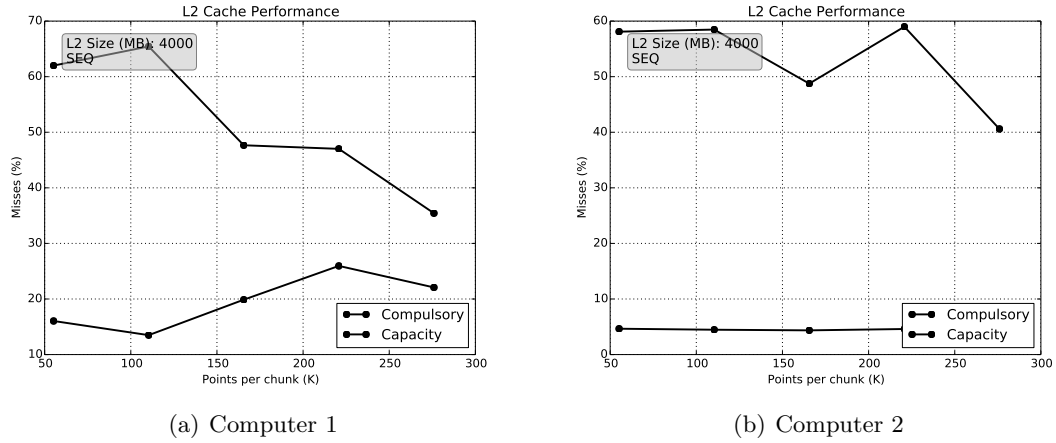


Figure 5.27: Graphs showing how the miss rate on L2 evolves depending on chunk size in Baron.

These results gain confirm the results obtained when using the Lourizan, CALD and Baron datasets. The stronger machine clearly has a big advantage, as this is the most demanding dataset tested in this round of testing. The advantage is so big, that even in the first sample capacity misses are greater than compulsory misses. As a consequence, we do not see the graphed miss lines cross at any point.

The last test compares L2 loading time depending on chunk size (see Figure 5.33). As we can see in the graph, Computer 2 is faster for every chunk size beating Computer 1 by a big margin. The weaker machine was not able to compete in the Baron dataset and with the Castelao dataset is even worse.

In conclusion, PCM scales well when dealing with massive point clouds. Of course, when running on machines with weaker hardware, there will be a dip in performance. But, our objective was satisfied, since PCM was able to work with massive datasets on low and mid-range machines. This fact will be better seen on section 6.3, since a Linux laptop was used for the tests performed.

All of the operations that PCM is able to perform are out of core, this means that all of the operations run in the tests are applicable to clouds of arbitrary sizes. This is specially valuable when dealing with the extremely dense clouds that scanners provide nowadays. The biggest cloud that has been tested in PCM has been an artificially generated $\sim 12KM$ point cloud. This cloud is orders of magnitude bigger than the clouds that point cloud software is able to deal with nowadays. This makes PCM an unique piece of software in its class, that is prepared even for future datasets that are even more demanding.

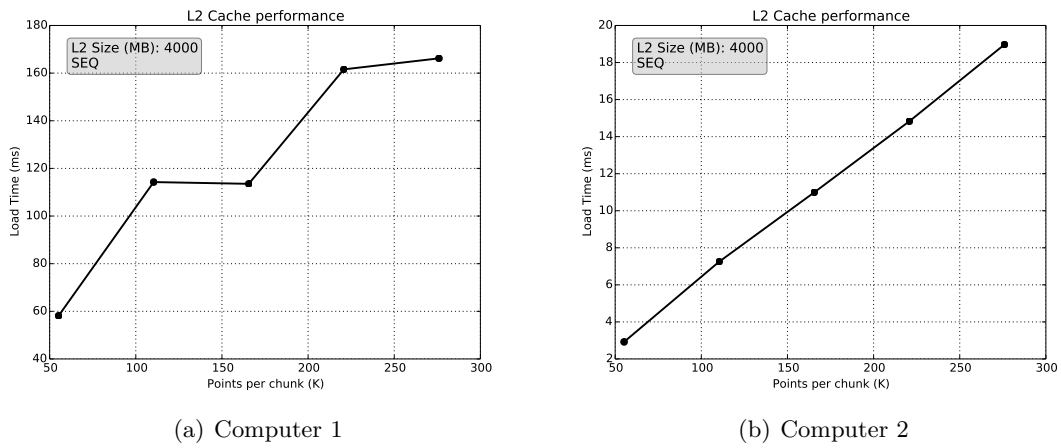


Figure 5.28: Graphs showing how the load times on L2 vary depending on chunk size in Baron.

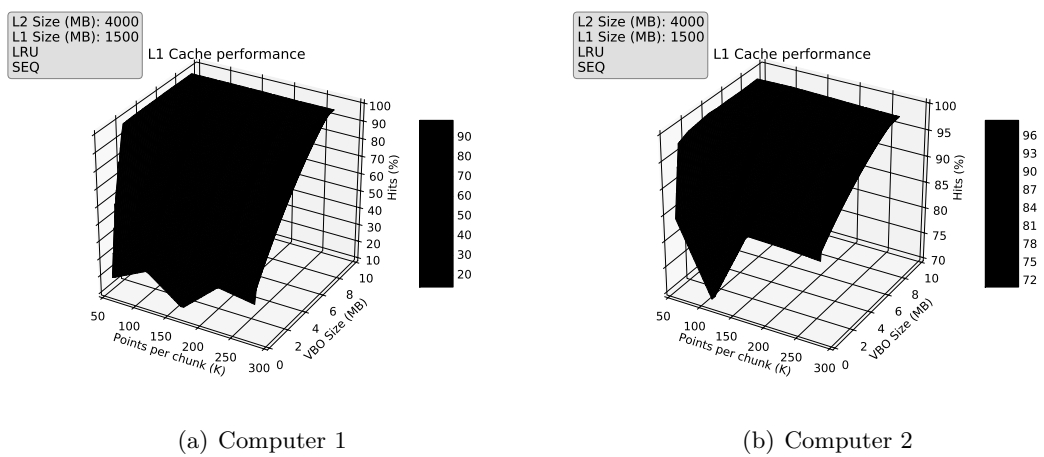
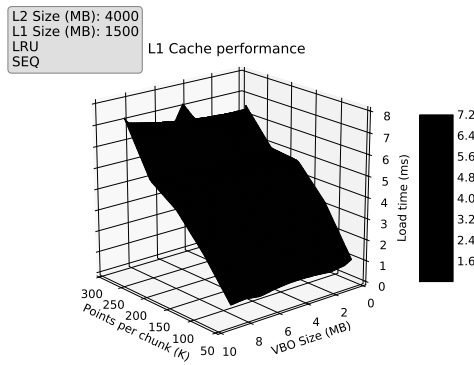
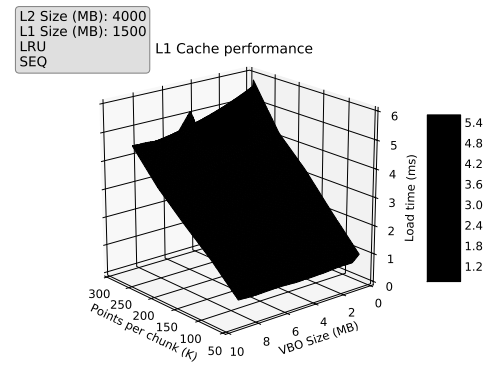


Figure 5.29: Graphs showing how hit rate on L1 behaves depending on chunk size and VBO size in Castelo.

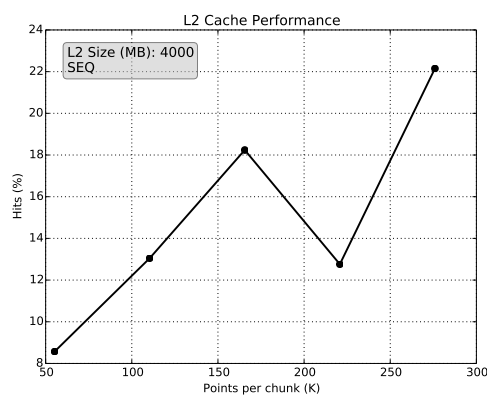


(a) Computer 1

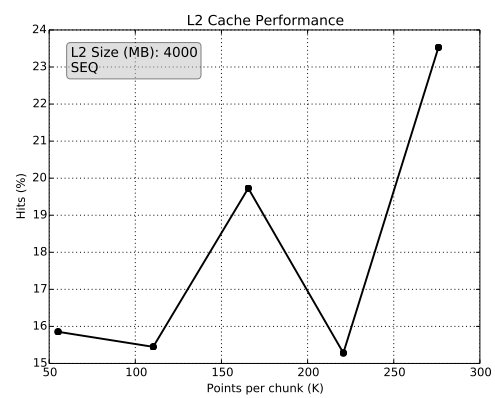


(b) Computer 2

Figure 5.30: Graphs showing how load times on L1 evolve depending on chunk size and VBO size in Castelo.



(a) Computer 1



(b) Computer 2

Figure 5.31: Graphs showing how hit rate on L2 evolves depending on chunk size in Castelo.

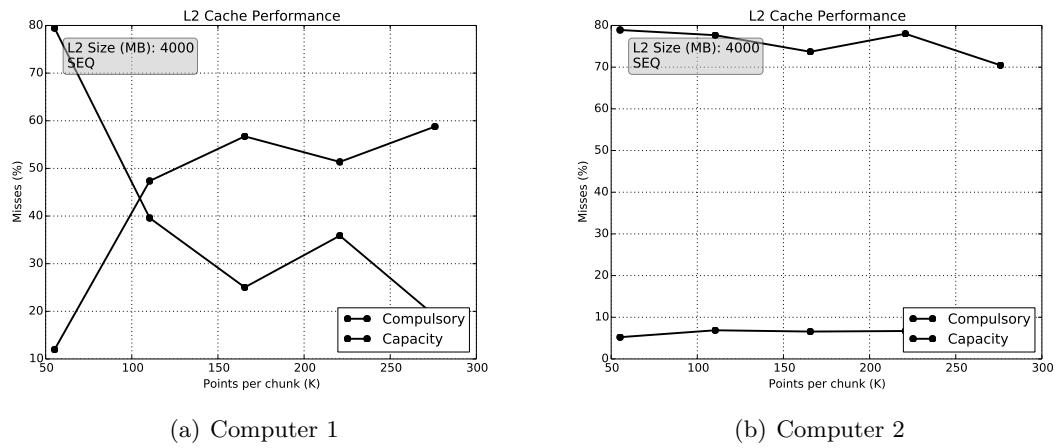


Figure 5.32: Graphs showing how the miss rate on L2 evolves depending on chunk size in Castelo.

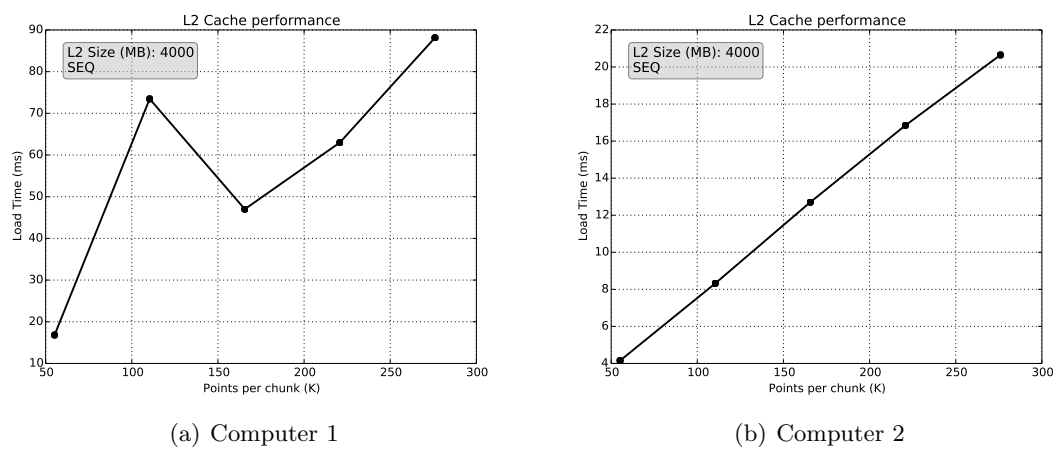


Figure 5.33: Graphs showing how the load times on L2 vary depending on chunk size in Castelo.

Chapter 6

ToView

ToView is how we named the visualizer that uses PCM under the hood, it was also implemented using C++, OpenGL for visualization purposes and QT [NCE95] for the user interface. All of these choices will be explained in the rest of the chapter.

6.1 Design

The visualizer has been implemented using **OpenGL 4.3** and **QT 5.3**. These two technologies were chosen so that the compatibility with multiple platforms was kept. Either OpenGL or QT are compatible with Windows, Linux and MacOS X.

On one hand, OpenGL is an API that allows an application to access and control the graphical system of the machine in which it is being executed. It could be a workstation with a high performance graphics card or a common desktop computer, a video-games console, etc.

On the other hand, QT is a multi-platform graphical application creation framework. QT is open-source software that has been used and improved since it was created. It fits perfectly with our project since is C++ based.

QT is used in more than 70 industries, by leading companies in their markets. QT is intuitive, has a suite for the design of interfaces called *QT Designer* and plugins for the major development IDEs like Visual Studio.

The compilers used as mentioned before are: Visual C++ in Windows systems, GCC in Linux and Clang in MacOS X.

6.1.1 Graphical User Interface

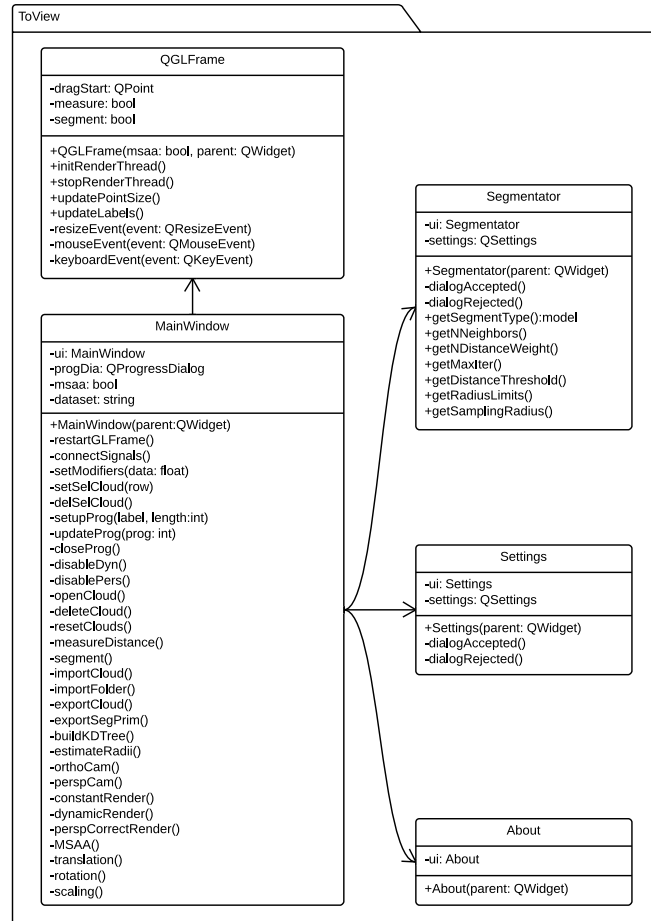


Figure 6.1: Class diagram of the user interface of ToView.

In the Figure 6.1, one of the main classes is **MainWindow**. This class gives the user access to all the functionality of the visualizer, it will serve as the name implies, as the main window in the interface. This class will centralize all the functionality and access to secondary dialogs, as well as communicate with the OpenGL context and render thread.

The OpenGL context is represented in the interface by the class **QGLFrame**, that is in charge of creating the corresponding OpenGL context, render thread and displaying each frame rendered. This class will also handle all the keyboard and mouse events.

The **Segmentator** class, is a sub-dialog that will let the user input the parameters for

the segmentation of primitives in the clouds. All of the choices are saved using the **Qsettings** class. This QT class simplifies the process of managing settings in an application, eliminating the need for custom configuration files.

Furthermore, **Settings** is another sub-dialog that allows the user to configure several visualizer parameters. This dialog also relies on **QSettings** to store the settings permanently.

Finally, **About** is just another sub-dialog that shows information about the software (version, authors, etc.).

6.1.2 Render thread

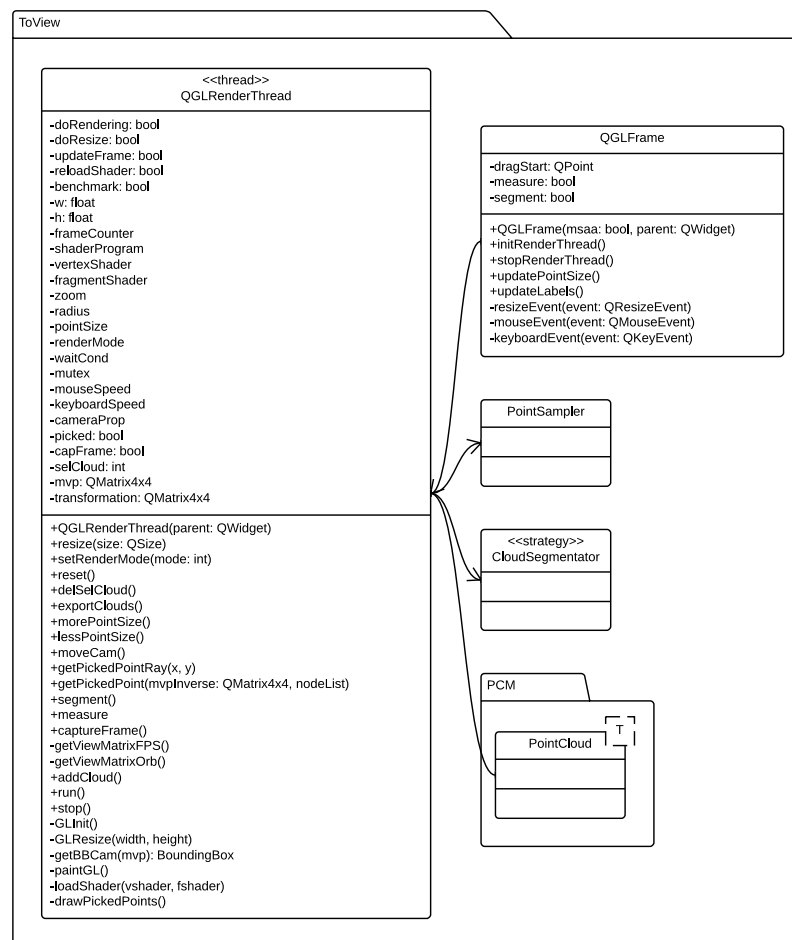


Figure 6.2: Class diagram of the render thread and its related classes.

As we can observe in Figure 6.2, **QGLRenderThread** is an extremely complex class, that as its name hints, represents a render thread. This class will be in charge of initializing OpenGL, loading shaders, rendering, calculating the camera parameters, etc.

The thread is created by **QGLFrame**, next the OpenGL context will be transferred to that thread so that the rendering can begin. Rendering is done in a different thread than the GUI, so that the latter will still be responsive even in the most demanding cases.

Because of this, the communication between the render thread and the GUI will make use of QT *signals* and *slots*¹. With these tools, the communication between the different threads will be robust and simpler. Since the render thread could also be used from multiple threads, we will also use a *mutex* to secure concurrent accesses.

To access the cloud information in PCM, the external interface **PointCloud** is used. **PointCloud** will provide the necessary information to render the desired clouds; like node lists, point data, memory hierarchy management, etc.

6.1.3 Segmentation

In Figure 6.3, the classes related to the segmentation process are detailed. One of the most significant classes is **PointSampler**, this will be the class that samples the corresponding point cloud and then converts the data so that it is compatible with PCL. It also is capable of picking individual points for distance measurements, and storing them in a similar fashion to a STL vector for quick access.

Next, the most important class will be briefly explained. **CloudSegmentator** uses the information obtained by **PointSampler** to segment three types of primitives, *planes*, *cylinders* and *spheres*. But this is not all the functionality that this class provides, it is also capable of exporting these primitives to an AutoCAD compatible format.

To achieve these objectives, this class uses the **RSampleConsensus** class for primitive fitting and **DL_Dxf** for *DXF*² exporting. Since there is not a good way to export cylinders and spheres in the DXF format, the segmentator will also support exporting these primitives as *SCR*³ files.

Since some of the primitives need normals for the segmentation process, **CloudSegmentator** also includes the capability to estimate them. The pattern *Strategy* was used for

¹Language tool introduced in QT for communication between objects, which allows the implementation of the Observer pattern. The concept is that GUI widgets can send signals containing event information, which can be received by other objects using special functions known as slots.

²*Drawing Exchange Format* is a CAD data file format developed by Autodesk for enabling data interoperability between AutoCAD and other programs.

³*Script Files* are simply a list of AutoCAD commands.

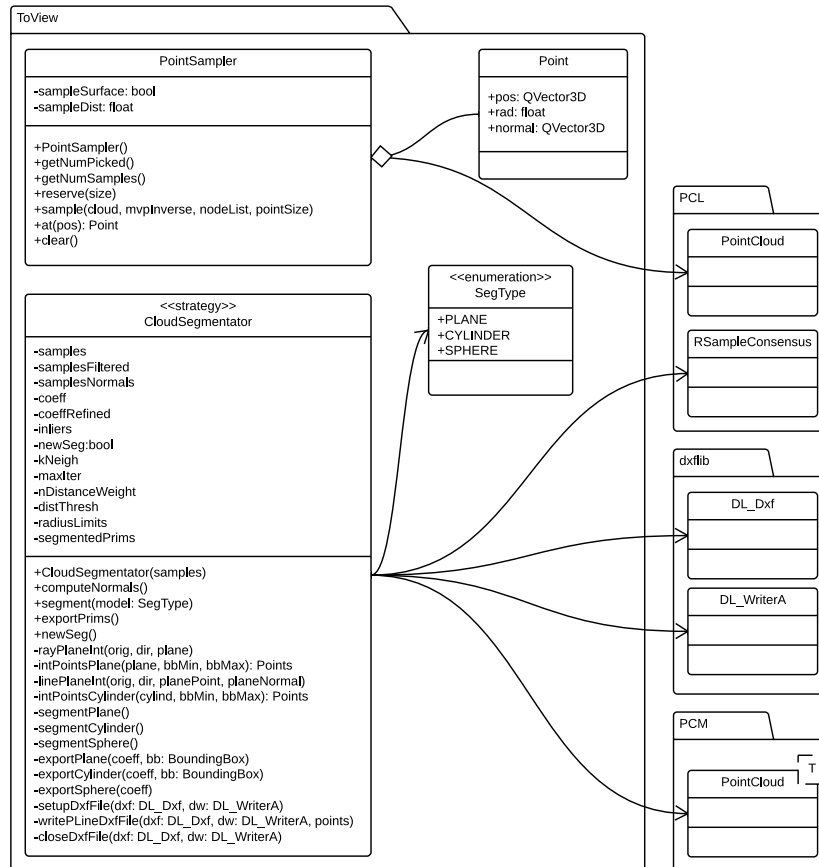


Figure 6.3: Class diagram of the classes related to segmentation.

the design of this class so that adding primitives in the future will be easy.

6.2 Functionality

The need for a robust tool to complement PCM was quickly noticed when using PCM's basic visualizer. This tool was enough for testing but left a lot to be desired, even when multiple clouds were not rendered onscreen and no advanced operations were performed on them.

Moreover, the rendering code was written with the help of an open source scene graph called OSG⁴. This was also an inconvenient because of the massive nature of the clouds

⁴Open Scene Graph

that this software is designed to deal with. Heavy optimization would have to be performed and complete access to OpenGL rendering code was needed.

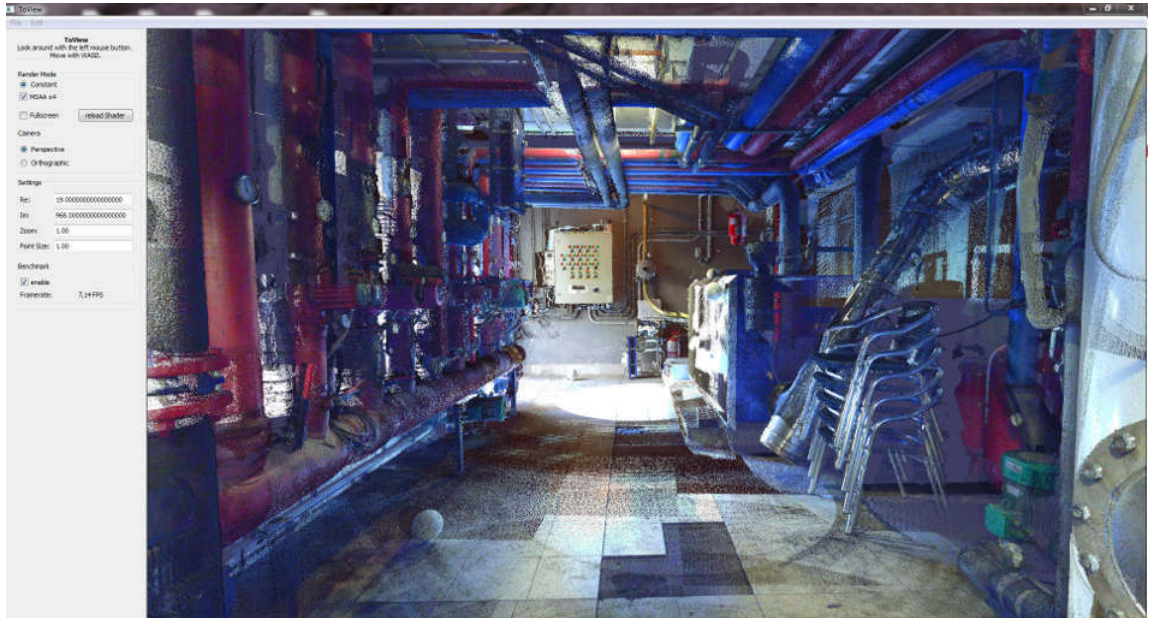


Figure 6.4: Screenshot showing an early version of ToView.

Furthermore, almost all of PCM's functionality was only accessible through multiple command line tools. When trying to integrate this software in engineering workflows, this was a big limitation.

If PCM was to be adopted by non-technical users, a GUI and more interactivity were essential. This is where ToView came to the rescue (see Figure 6.4), with multiple cloud support, a GUI, advanced rendering, distance measuring capabilities, object detection, etc. All the functionality that ToView brings to the table will be exposed in the following subsections.

6.2.1 Render modes

In light of the need for high quality point based rendering, but also trying to keep versatility, three rendering modes are supported in ToView:

- **Constant:** Most basic type of point-based rendering, it does not require point normals or radii.
- **Dynamic:** Almost the same as the previous mode, but requires point radii to reduce clipping and increase rendering quality and performance.

- **Perspective accurate:** Most advanced type of rendering, requires point normals and radii. This rendering mode yields high quality renders but at a greater computational cost.

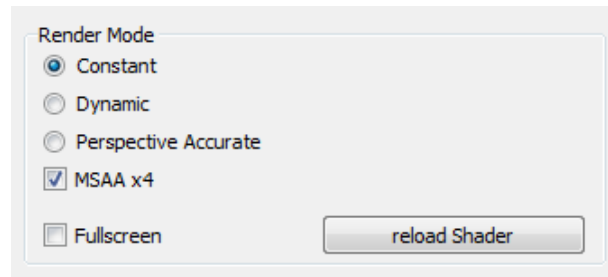


Figure 6.5: Screenshot showing all of the available render modes in ToView’s interface.

As we can see in Figure 6.5, more parameters like anti-aliasing or full-screen rendering are also exposed through the GUI. Anti-aliasing will play a huge role in high quality rendering as it increases image quality dramatically.

All of these rendering modes and a complementary technique that was not exposed to the user in ToView, but was also implemented for testing, will be explained in depth in chapter 7.

6.2.2 Camera types

Since camera only one camera type was present in the old visualizer, two types of camera paradigms were implemented:

- **First-person camera:** Type of camera that imitates how movement is implemented in video games. This camera is ideal for virtual tours of the scenes, or working with really big scenes with multiple objects.
- **Orbital:** This camera model emulates cameras in modeling software (Blender, AutoCAD, etc.). This camera type is optimal for working with single objects and an orthographic projection.

Furthermore, because of the engineering nature of our software, a perspective camera may not be the optimal choice. This is the reason why two types of projection are available for the user:

- **Perspective:** Projection model that reflects how the human eye perceives scenes, objects in the distance appear smaller than objects close by. This projection type is useful when moving through a city or interior datasets.

- **Ortographic:** This projection type ignores the aforementioned effect so that to-scale modeling can occur. This projection model is useful when working with single objects or when we need to see parallel lines as parallel in the final image.

All of these camera types are controlled using the keyboard and mouse, trying to make it as easy as possible for the end user. The math and implementation details behind these cameras are explained in section 3.4. The options are exposed to the user through a convenient menu in the interface and do not require restarting the application to be applied (see Figure 6.6).

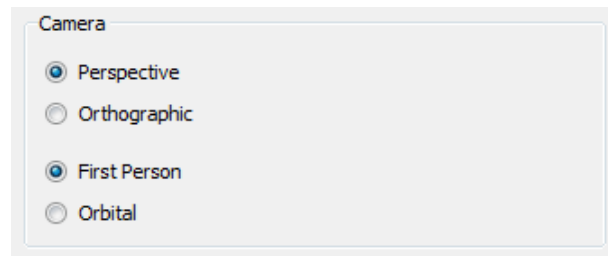


Figure 6.6: Screenshot showing all of the available camera options in ToView's interface.

6.2.3 Preprocessing

As mentioned before, scanners sometimes output clouds that may not be optimal for the task at hand. As a consequence, some preprocessing can be needed. ToView offers its users several operations:

- **Voxel grid filter:** Downsampling filter, useful to thin dense point clouds or keep the cloud density uniform.
- **Statistical outlier removal:** Noise removal filter, it can be used to remove outliers or noise present in the clouds.
- **Radii estimation:** Necessary process if the end user wants to use the two advanced rendering modes. It estimates the value of the point radii.
- **K-d tree building:** Essential process for the visualization of the point clouds. Without a spatial acceleration structure, the clouds would be too big to render or perform any operation.

The preprocessing filters are further explained in chapter 9. All of these point cloud operations are exposed to the user through the tools menu in the interface(see Figure 6.7).

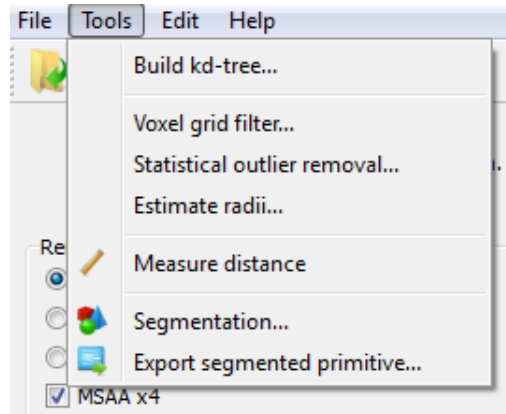


Figure 6.7: Screenshot showing all of the available point cloud tools in ToView's interface.

6.2.4 Cloud interaction

One of the main drawbacks of having a simple visualizer without a GUI was the inability to integrate multiple point clouds, and perform modifications on the clouds (see Figure 6.8).

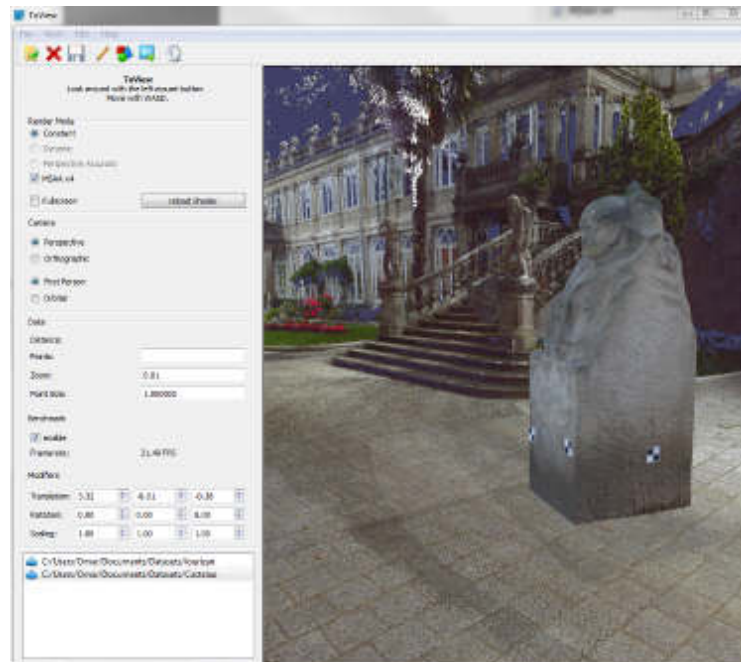


Figure 6.8: Two clouds (a park and a statue) integrated using ToView.

ToView offers multiple possibilities on this front:

- **Point cloud transformation:** Each cloud can be transformed to the user's liking.

- **Translation:** The cloud can be moved through space.
- **Rotation:** The cloud can be rotated.
- **Scaling:** The cloud can be transformed to be bigger or smaller.
- **Multiple point cloud integration:** The user can transform multiple clouds at the same time. This allows the integration of multiple point clouds, correcting positioning errors, wrong coordinate systems, etc.
- **Cloud selector:** Select a cloud to transform, eliminate or perform any operation in an easy way through the interface.

The aforementioned features are exposed through the interface in a panel in the interface (see Figure 6.9). All of these features work in real-time, allowing the user to see the changes made reflected instantly. Clouds can also be added or removed in real-time according to the user's needs.

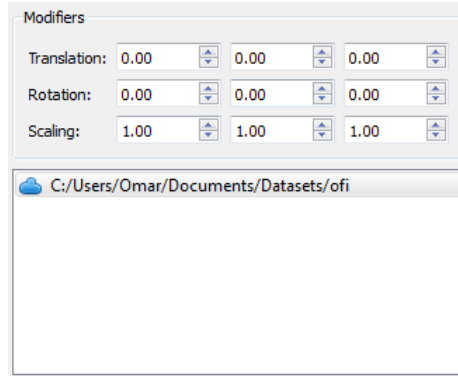


Figure 6.9: Screenshot showing the cloud interaction menu in ToView's interface.

6.2.5 Point picking

In order to be able to measure distances or segment primitives interactively in the clouds, the visualizer has to be capable of finding the correct point that the user desires in the clouds. This action is called point picking, and is performed in real-time by the visualizer.

To achieve this, the first step is getting the position the user has clicked in the window. This is done using QT's API and returns two values, the x and y coordinates.

Secondly, the coordinates are transformed so that we have the coordinates of a ray corresponding to that pixel in NDC. The ray origin can be obtained with the following equation:

$$\mathbf{r}_{\text{start}} = ((x/w - 0.5) * 2, (y/h - 0.5) * 2, 0) \quad (6.1)$$

Being w the width of the window and h the height. The ray end can be obtained with the equation that follows:

$$\mathbf{r}_{\text{end}} = ((x/w - 0.5) * 2, (y/h - 0.5) * 2, 1) \quad (6.2)$$

Once we have the ray in NDC space, we need to transform it to world space. The operation needed to get the ray in world space is just a matrix multiplication:

$$\mathbf{r}_{\text{world}} = MVP^{-1} \cdot \mathbf{r}_{\text{ndc}} \quad (6.3)$$

Finally, after the ray in world coordinates is obtained, it is just a matter of obtaining the points that intersect with the ray and then return the closest to the viewpoint.

To use this functionality, the user will just have to right-click the mouse, and the visualizer will take care of all of the steps internally (see Figure 6.10).

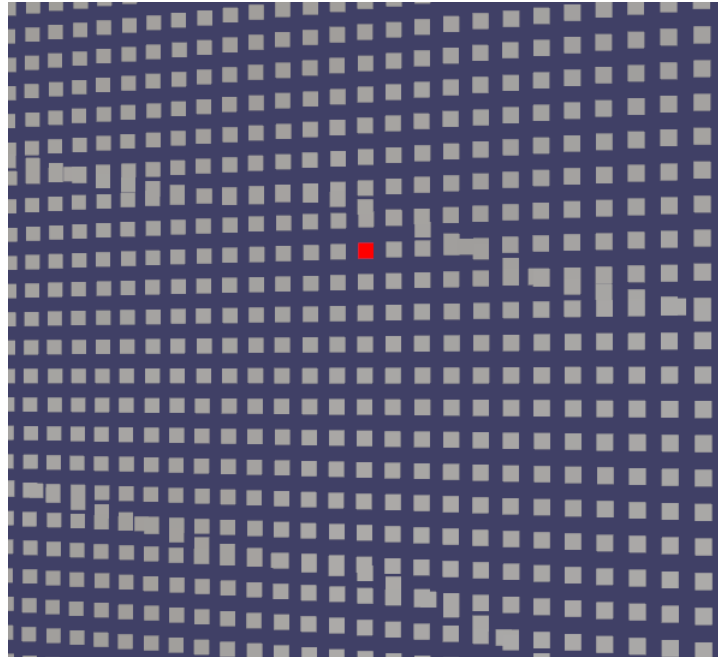


Figure 6.10: Screenshot showing the picked point in a cloud in red.

6.2.6 Distance measurements

A typical operation in a point cloud for engineers is measuring distances. The surveyor nowadays, does not have to go back to the field each time a new measurement is needed. With point clouds, the laser scanner will register data of every surrounding surface and

all of the information will be stored in the point cloud for later use.

Therefore, once we have the ability to select two or more points, we will have the possibility of obtaining measurements. The accuracy of the measurements, will depend on the precision of the scanner (the scanner that we are using has millimeter precision). Having the points $\mathbf{p}_1$ and $\mathbf{p}_2$, the distance is obtained with the following equation:

$$D = \|\mathbf{p}_1 - \mathbf{p}_2\| \quad (6.4)$$

The user just needs to click the distance measurement option in the tools menu (see Figure 6.7) and then select two points in the cloud. The measurement will be performed in real-time and the result displayed in the interface (see Figure 6.11).

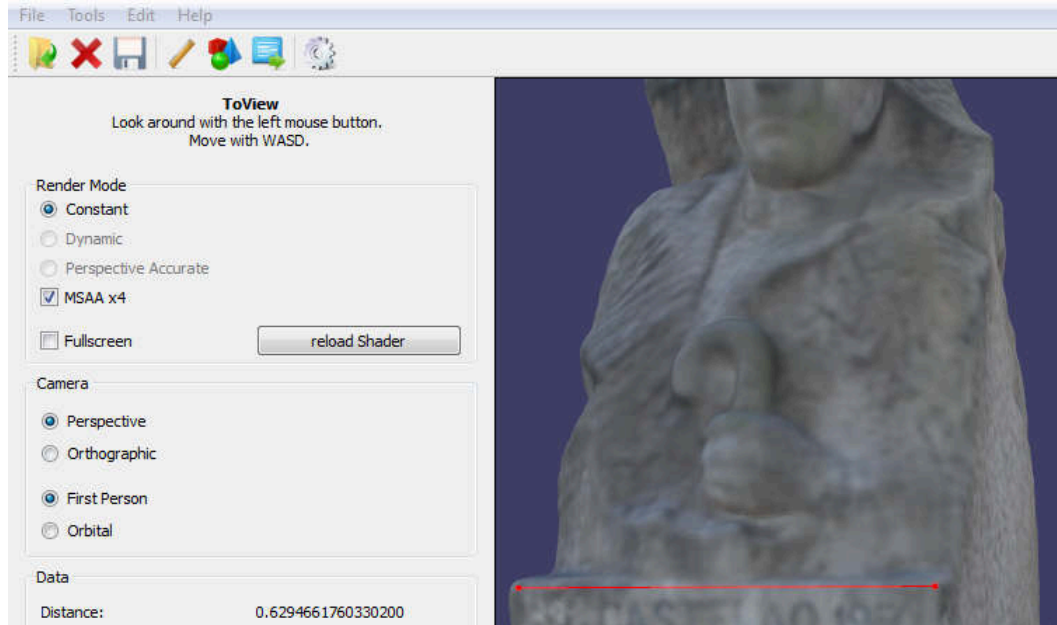


Figure 6.11: Screenshot showing a distance measured in a point cloud.

6.2.7 Primitive segmentation

Another feature widely requested by civil engineers, is the estimation of parametric primitives from the point cloud. To better explain the usefulness of this feature, the reader may think of a pipe. If we wanted to select each point to extract the cloud representing the pipe or create an CAD representation of the object, how would we go about it? This is not a trivial problem, and even less when dealing with massive point clouds.

To solve this problem, in ToView we have designed a semi-automatic way of segmenting

objects. The steps involved in this process are:

1. The user selects a point in the cloud.
2. The visualizer samples the surrounding area.
3. Estimates the necessary features.
4. Uses RANSAC to estimate the chosen primitive.
5. Displays the result to the user.

Due to the limited time available to implement an ambitious project as this one, PCL was integrated with ToView to make use of the implemented RANSAC classes available in PCL. Thanks to this we have been able to test advanced RANSAC implementations and employ more time investigating and making the visualizer more robust.

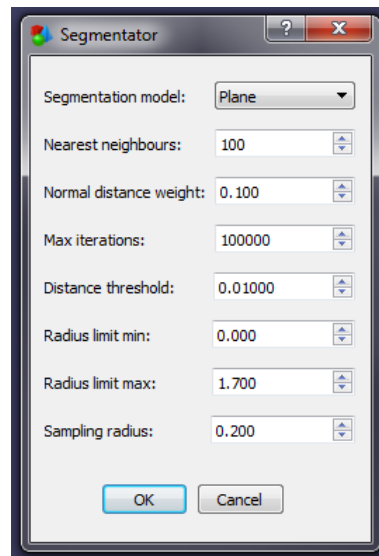


Figure 6.12: Screenshot showing the segmentation options available.

PCL is a widely used point cloud library, with lots of features not only limited to RANSAC implementations. In the future it will be interesting to integrate PCL further, for normal estimation, advanced segmentation, etc.

ToView is capable of estimating planes, cylinders and spheres. The process also allows the user to further refine the primitives selecting more points after the initial estimation (see Figure 6.13).

This estimation is also performed in real-time and is easily accessible from the GUI. When the user clicks on the option to segment primitive (see Figure 6.7), a new dialog will appear

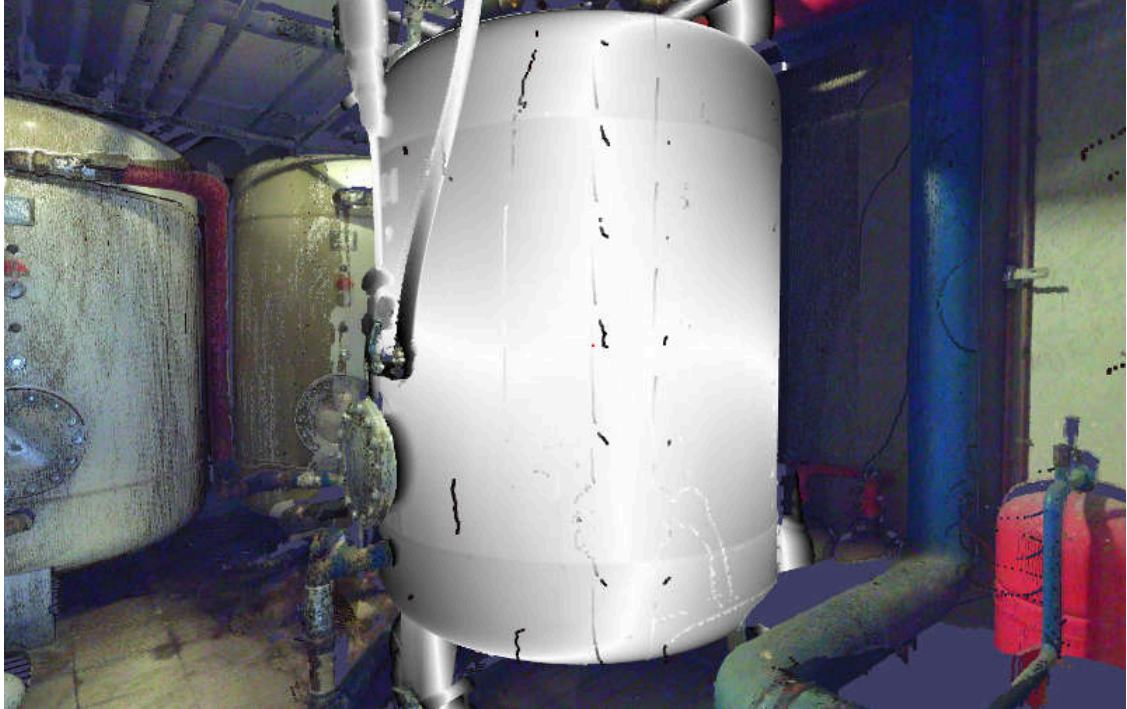


Figure 6.13: Screenshot showing a segmented cylinder.

that will let the user fine tune the segmentation options (see Figure 6.12). The process is really intuitive for the user, as it is only needed to select a point of the object that needs to be segmented.

All of the theory behind primitive segmentation will be explained in depth in chapter 8.

6.2.8 Primitive exportation

In order to use the estimated parametric primitives, we will have to export them in a format that other software can read. Therefore, *DXF*, a standard CAD exchange format was chosen for planes and *SCR* an AutoCAD scripting format for more complex primitives like cylinders or spheres.

To create the exported DXF file, we have used *dxflib* [Mus12], an open source library for the manipulation of DXF files.

In order to highlight and export these primitives, some previous steps are needed. Namely, calculating the point-primitive distance and bounding the primitives. These two steps are extensively explained in chapter 8.

ToView supports the estimation of any number of primitives and the exportation in one

step of all of them. The user just has to select the option export segmented primitive in the tools menu (see Figure 6.7).

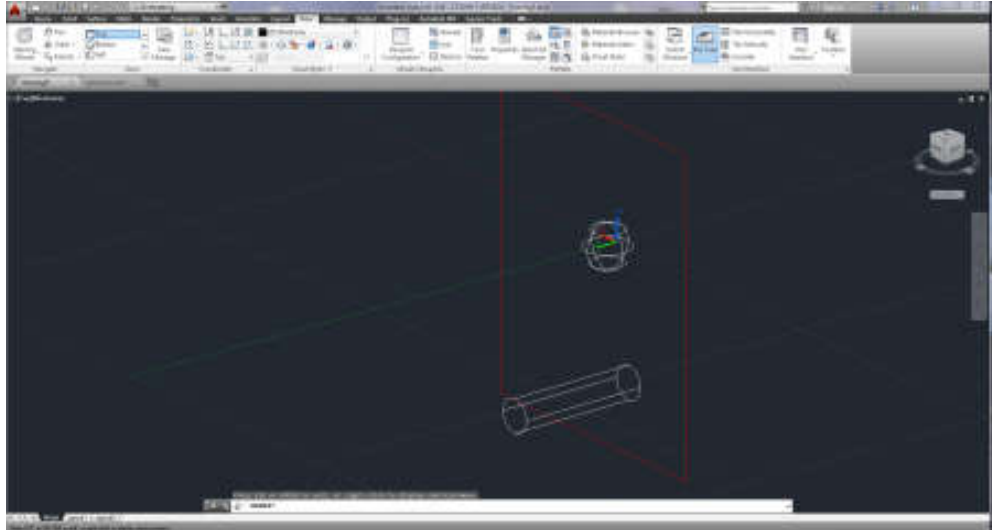


Figure 6.14: A segmented plane, cylinder and sphere opened in autocad.

This feature has the potential to substantially reduce the time needed to create CAD models of rooms, pipes, etc. (see Figure 6.14). It also has the potential to clean clouds faster, the surveyor just needs to click on a point to select the complete object (in contrast to manually selecting all the points), and then choose what to do with it.

6.3 Results

Again, two different computers were used for testing. The first, **Computer 3** has the following specs:

- Intel Core i7-2600K CPU (4 cores, 8 threads)
- NVIDIA GTX 680 graphics card (2560x1440)
- 8 GB of 1600 MHz DDR3 RAM
- WD Caviar Black HDD

The second, **Laptop 1** has the following specifications:

- Intel Core i7-3632QM CPU (4 cores, 8 threads)

- NVIDIA GT 720M graphics card (1366x768)
- 16 GB of 1333 MHz DDR3L RAM
- SSHD

The operating systems used to run the tests were:

- Computer 3: Windows 7 x64 with the VC++ v110 compiler
- Laptop 1: Ubuntu 14.04 x64 with the GCC 4.8 compiler

The libraries used were:

- QT 5.3
- OSG 3.0.1
- GLEW 1.10.0
- PCL 1.7.1

The datasets that will be used for testing are described in Table 5.1. The L1 cache size used in Computer 1 was 1500 MB, while in Laptop 1 was 512 MB. In addition, the L2 cache size used in Computer 1 was 4 GB and in Laptop 1 was 1 GB. The cache settings used are those recommended in section 5.4

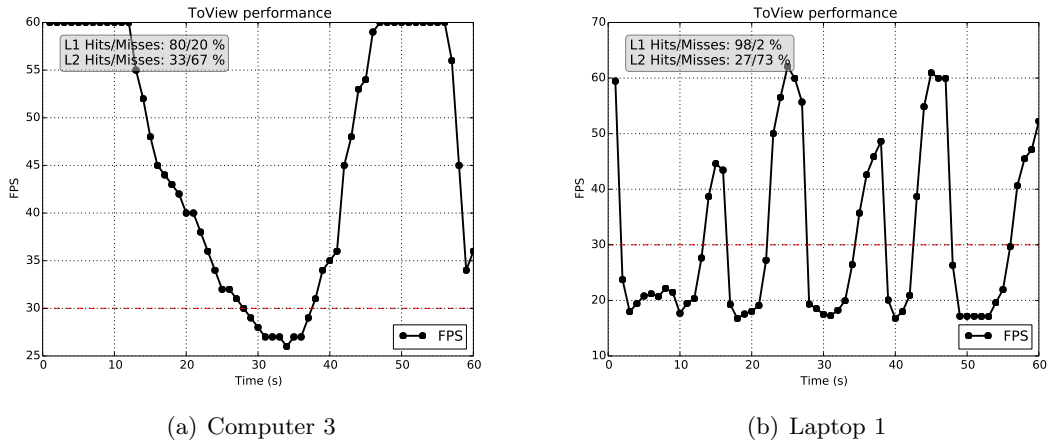


Figure 6.15: Graphs showing the FPS that ToView achieves in the Lourizan dataset.

Since the smaller datasets tested (Castelao, Dorna, etc.) obtained more or less 60 FPS in both computers, we will not present graphs with the results. The first truly taxing dataset tested was lourizan as can be seen on Figure 6.15. In the graph we can see that both computers keep a good level of interactivity when rendering the Lourizan dataset.

Computer 3 will allow a better representation of the scene, since the cache sizes are bigger and more points will be rendered onscreen. Furthermore, Computer 3 does not dip below 30 FPS in much occasions when rendering this dataset.

The next dataset tested was CALD, as can be seen on Figure 6.16. This dataset is harder on both Computer 3 and Laptop 1, as we can see in the graphs. A lot more dips below 30 FPS on both computers can be seen as soon as the caches start filling up.

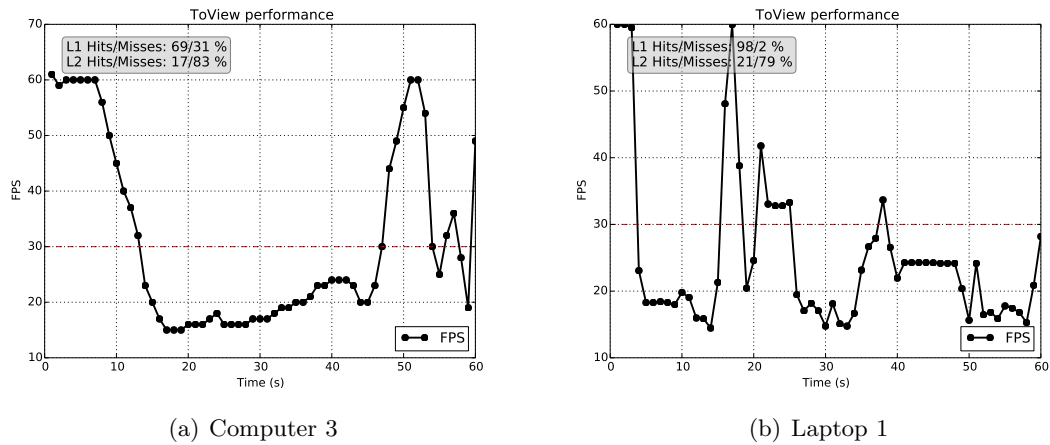


Figure 6.16: Graphs showing the FPS that ToView achieves in the CALD dataset.

Although performance starts to suffer on this massive datasets, we still can interact without much trouble with the point clouds even in Laptop 1. Furthermore, the Baron dataset was also tested on both machines, the results can be seen on Figure 6.17.

These results are truly impressive, where the true potential of PCM and ToView shines. When starting to deal with massively dense point clouds we can see that both computers have dipped below 30 FPS, but still keep interactivity. Even more impressive is that when using the laptop the sensation was smooth even with low FPS, achieving this with other software is complicated to say the least.

Finally, the last dataset used when testing ToView was Castelo. This is the biggest dataset at our disposal, a truly massive scene. The results of this test can be seen in Figure 6.18.

In this last test Laptop 1 truly suffers, but still keeps some impressive numbers taking into account that Castelo is a $\sim 1KM$ point cloud. Computer 3 also fares pretty good on this

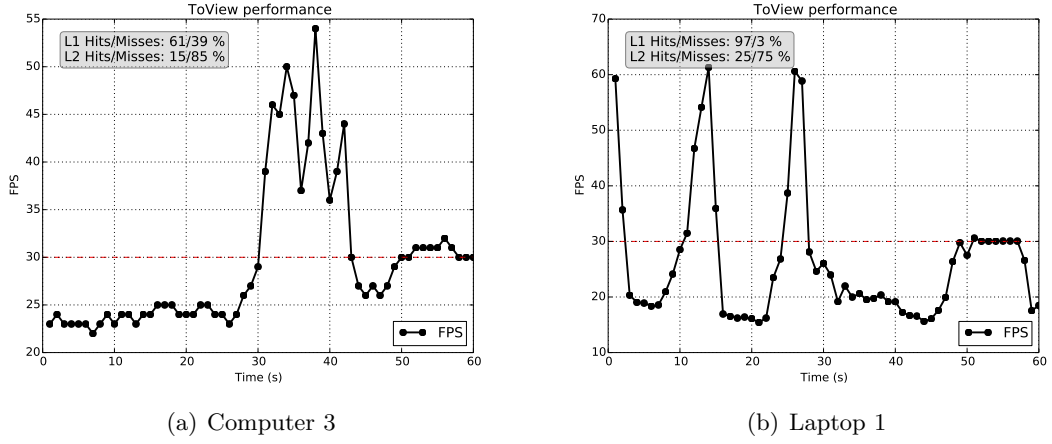


Figure 6.17: Graphs showing the FPS that ToView achieves in the Baron dataset.

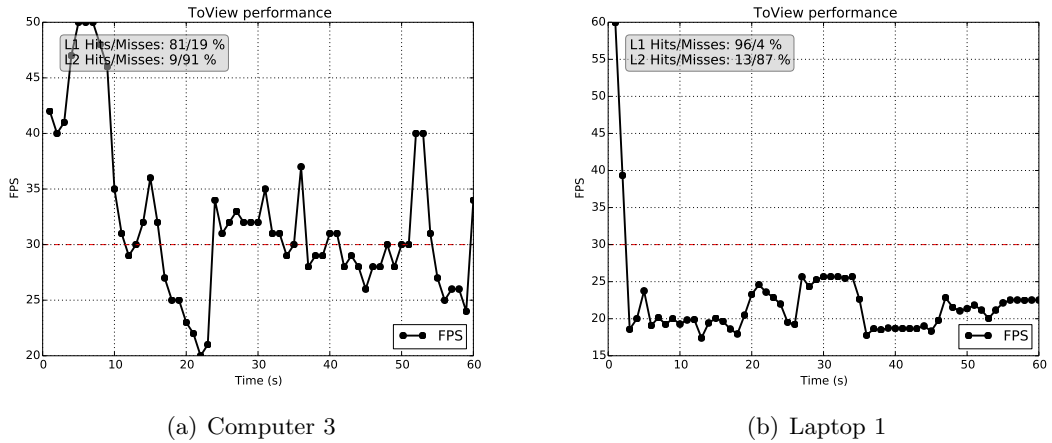


Figure 6.18: Graphs showing the FPS that ToView achieves in the Castelo dataset.

dataset, frames are always around 30 FPS.

Summing up, running ToView in almost any kind of computer is possible. Taking a look at these results we can see that ToView can be run perfectly in a 650 € laptop, if we wanted higher rendering quality, we would have to invest on a more powerful machine, but ToView and PCM can adapt to low resource situations perfectly.

ToView thanks to QT can also run natively on Linux, Windows, etc. This makes ToView a tool with a lot of versatility, that should be able to be adapted to any engineering workflow.

Chapter 7

Advanced GPU point rendering

The first approaches to point rendering like [ZPvBG01] were CPU based, this was clearly not the right approach for real-time point rendering. To increase performance, the massively parallel computing power of GPUs should be taken advantage of. In a similar fashion to how triangle meshes are rendered nowadays, we will offload rendering to the graphics hardware freeing the CPU to perform other tasks. This will provide much higher performance than a CPU based solution.

This would seem an easy task at first sight, but due to missing point splat support in graphics hardware; exploiting the GPU's hardware acceleration for point-based rendering is not an straightforward task. Fortunately, thanks to the introduction of fragment and vertex shaders, we will be able to use the hardware for this job using custom shaders.

7.1 Performance details

One of the reasons why point-based rendering is used, is the huge complexity of today's datasets (more than 1KM points), that can be acquired by LiDAR scanners. When these models are rendered, a triangle may only cover a few or not even one pixel. Since the triangle rasterization setup in this cases does not pay off, rendering massive triangle meshes is not efficient.

In these cases, is where points seem to be the more suited rendering primitive. But, when rendering massive point clouds, we should use the following guidelines:

- **Interleaved vertex arrays:** To minimize the number of calls to the OpenGL API, the point data (positions $\mathbf{p}$, colors $\mathbf{c}$, radii $\mathbf{r}$, normals $\mathbf{n}$, etc.) has to be stored in one large array. Thanks to this, we can render the points with just one call to

glDrawArrays(). For better memory coherence, the data should also be stored in an interleaved manner (i.e. $[\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{c}_1, \mathbf{c}_2, \dots]$).

- **Vertex Buffer Objects:** In each frame, all the point data has to be transferred to the GPU. This would quickly become a bottleneck in massive datasets. Since sometimes this data will be static for several frames, it will be stored in high performance video memory for efficient GPU access. VBOs are the standard way of achieving our objective since OpenGL version 1.5 (2003).
- **Optimum VBO size:** The VBO size will be kept in the optimum range recommended by NVIDIA (1-4 MB). If chunks are larger than that, they will be split in several VBOs.
- **Data formats:** In order to reduce the GPU memory consumed, and decrease transfer costs, the point data will be stored in a compact format. Position and normals will be stored using single-precision floating-point numbers, colors will be stored using just 4 bytes (one per RGBA channel).
- **Shading language:** Low-level shading languages offer a small performance advantage, but since nowadays it is marginal, GLSL will be used for shader programming. High-level languages like GLSL are more convenient to use and way less problematic when adding features or reviewing the code.

All of these optimization techniques are valid for all the point rendering approaches that will be explained next.

7.2 Splat rasterization

This is the first step in splat rendering, it determines the pixels that a point will cover. Since there are no splat primitives, splats have to be created from other rendering primitives, like points or triangles. All of the following methods were implemented and tested in modern OpenGL, the code can be reviewed in the repository of the project (<http://tovias.citic.udc.es/>).

7.2.1 Image-aligned squares

For the highest performance, splats should be rendered using only one OpenGL point for each splat, using:

```
glDrawArrays(GL_POINTS, 0, VBOSize);
```

Listing 7.1: Draw call to draw a set of splats.

An optional output of a vertex shader is the point size, that when set to s , creates a image-space square of $s \times s$ pixels centered in the vertex position. Instead of squares, image aligned circles can be rendered by enabling point smoothing and MSAA:

```
glEnable(GL_POINT_SMOOTH);
```

Listing 7.2: OpenGL flag needed for point smoothing.

The screen-space size of the projected OpenGL point has to be adjusted in the vertex shader, so that holes do not appear when moving closer to the object. To allow us to modify the point size dynamically in the vertex shader and specify our coordinate system origin, we will use the following code:

```
glEnable(GL_PROGRAM_POINT_SIZE);
glPointParameteri(GL_POINT_SPRITE_COORD_ORIGIN, GL_LOWER_LEFT);
```

Listing 7.3: Flags needed for changing the point size in a vertex shader.

The adjusted point size is approximated by calculating the foreshortening of the point radii r , depending on the point position $\mathbf{p}$ in camera coordinates:

$$s = 1.5 \cdot r \cdot \frac{n}{p_z} \cdot \frac{h}{t - b} \quad (7.1)$$

Where n , t and b are the near/top/bottom parameters of the camera's view frustum and h the height of the viewport in pixels. The advantage of this rasterization method, is that because of its simplicity, it is the fastest of the three studied methods. It is also the one that requires the least amount of information, since it only needs the splat position, color and optionally radius.

However, this method is not optimal for high quality rendering. It yields bad results specially near the object's contour (see Figure 7.1), where the poor approximation of the splat shape is specially noticeable. Furthermore, since all fragments created with this technique have the same depth value, this prevents the correct blending of overlapping splats.

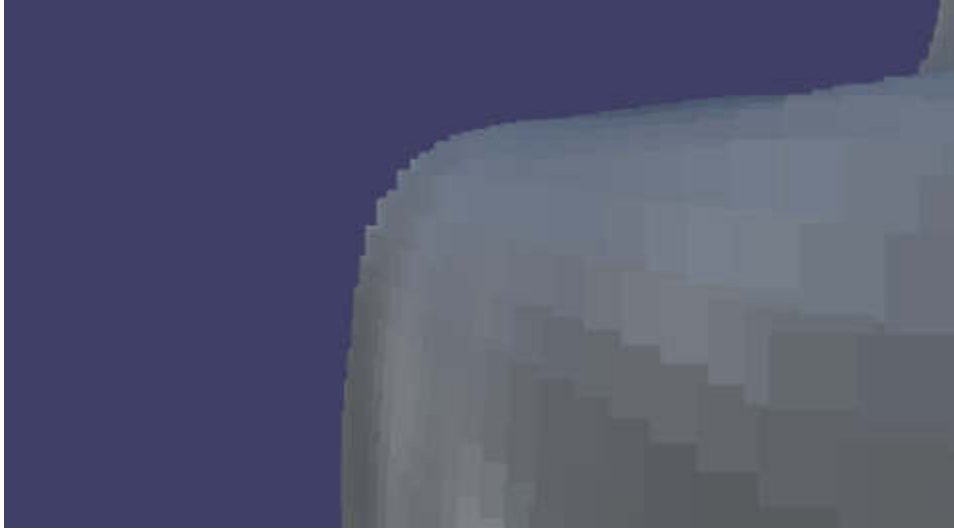


Figure 7.1: Render image showing the defects that image-aligned squares present in object contours.

7.2.2 Affinely projected point sprites

An improved approximation to the true splat shape is the one in [BK03], this technique was called circular object-space splats. It starts by adjusting the point size in the vertex shader as in the previous method, but for each of the squares, the fragment shader will determine whether or not the fragment corresponds to the inside or outside of the splat.

Fragments outside the splat will be discarded using the *discard* GLSL command. This will result in circular splat rasterization. For this technique to work we will have to activate the following feature in OpenGL:

```
glEnable(GL_POINT_SPRITE);
```

Listing 7.4: Flag needed to discard fragments in fragment shader.

Next, we need to parameterize the screen-space square in the range $[-r, r]^2$, being r the splat radius. For each of its fragments $(x, y) \in [-r, r]^2$, a depth offset δz from the splat center $\mathbf{p}$ can be computed as a linear function that depends on the camera-space normal vector $\mathbf{n} = (n_x, n_y, n_z)$:

$$\delta z = -\frac{n_x}{n_z} \cdot x - \frac{n_y}{n_z} \cdot y \quad (7.2)$$

The depth offset can after this step be used to compute the 3D distance to the splat center.

This distance can then be used to check whether a the fragment corresponds to the inside of the splat or not with:

$$\|(x, y, \delta z)\| \leq r \quad (7.3)$$

Since one of the problems that we found in the image-aligned squares was the lack of a depth value for the splat, the depth offset δz should also be used to correct the fragment's depth value. Starting from the camera space depth value $z' = p_z + \delta z$, the frustum and viewport transformations allow us to calculate the window-space depth buffer entry, from the far plane f and the near plane n :

$$\text{zbuffer}(x, y) = \frac{1}{z'} \cdot \frac{fn}{f - n} \cdot \frac{f}{f - n} \quad (7.4)$$

This approach yields much better results in comparison to the previous one, this is specially apparent in object contours. But it does not come without its share of issues. Since the depth offset is just an approximation due to the fact that in the Equation 7.2 a parallel projection is assumed. Thus neglecting the angle between the viewing ray and the splat normal. This creates artifacts when viewing the splat from extreme angles (see Figure 7.2).

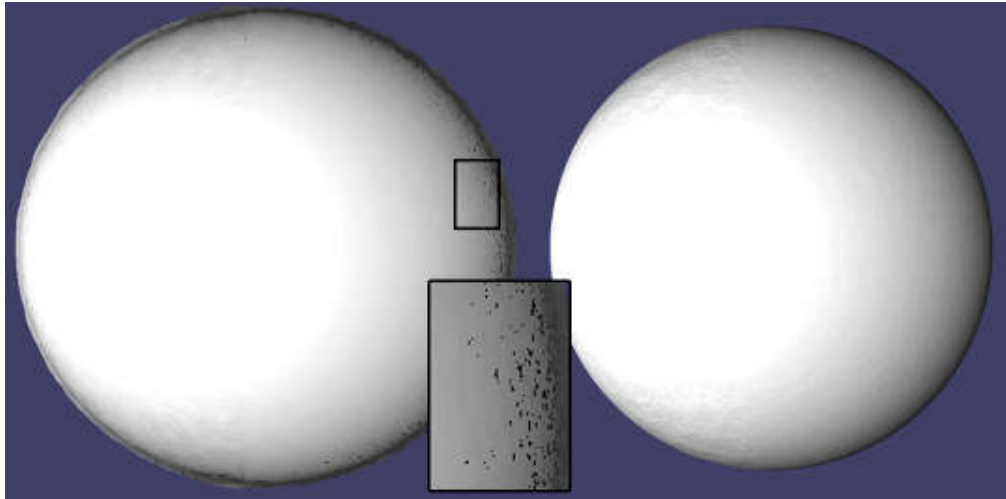


Figure 7.2: Local affine approximations can cause holes for extreme viewing angles (**left**). This problem is taken care of by using perspective correct rasterization (**right**).

7.2.3 Perspectively correct rasterization

The last approximation was able to correctly transform the splat center, but failed to do the same with the contour, this caused holes to appear in the rendered image. This is why,

to create a perspective correct approximation, the technique in [ZRB⁺04] was created. In this method, the outer contour of the splat was correctly transformed, but had projective errors in the splat center. This approach had a high computational cost, that made it very costly.

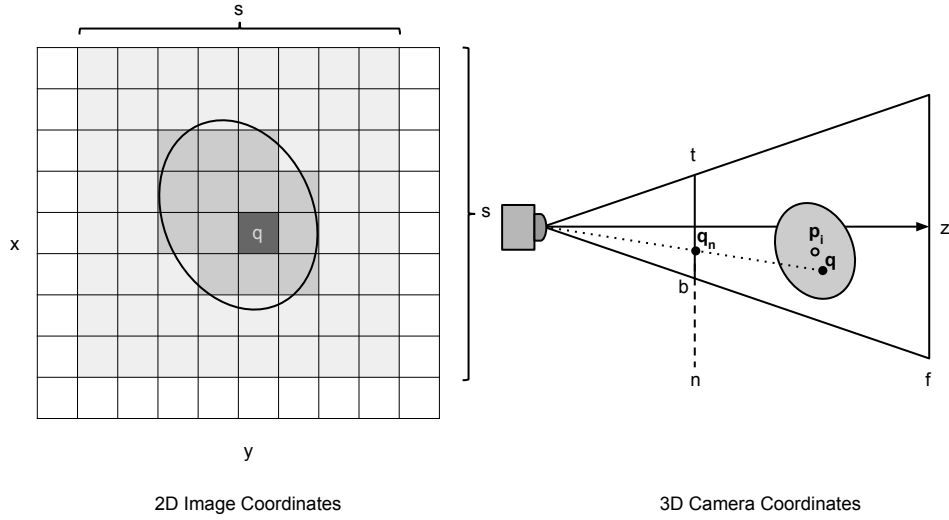


Figure 7.3: Projection of the splat and raycasting visual explanation.

A more efficient while being perspective correct technique was presented in [BSK04]. The main concept in this method is to determine the 3D point corresponding to a 2D fragment based on local raycasting. This method can also handle elliptical splats, although we will concentrate our efforts in circular splats since this is our case study.

A splat s_i is represented by a center $\mathbf{p}_i$, a radius r_i and its normal $\mathbf{n}_i$. We start by rendering the splats again with *GL_POINTS* and the pixel size given by the splat radius. Next, for each fragment (x, y) the *exact* corresponding 3D point $\mathbf{q}$ is computed using local raycasting (see Figure 7.3). With this point $\mathbf{q}$, the distance to the center of the splat is computed, if the fragment lies outside of the splat it is discarded with the *discard* command. The Equation 7.5 dictates if the fragment is outside or inside the splat:

$$\|\mathbf{q} - \mathbf{p}_i\| \leq r \quad (7.5)$$

To compute $\mathbf{q}$ we will first need to invert the window-to-viewport transformation, mapping the fragment (x, y) to a 3D point $\mathbf{q}_n$ on the near plane:

$$\mathbf{q}_n = \begin{pmatrix} x \cdot \frac{r-l}{w} - \frac{r-l}{2} \\ y \cdot \frac{t-b}{h} - \frac{t-b}{2} \\ -n \end{pmatrix} \quad (7.6)$$

Being $b/t/l/r$ the bottom/top/left/right corners of the frustum and w/h the width/height of the viewport. Casting a ray from the origin through the point $\mathbf{q}_n$ and intersecting with the splats supporting plane will yield the necessary point $\mathbf{q}$. First we will calculate the distance at which the ray intersects the plane:

$$t = \frac{\mathbf{p}_i \cdot \mathbf{n}_i}{\mathbf{q}_n \cdot \mathbf{n}_i} \quad (7.7)$$

Once the distance t is obtained, the point $\mathbf{q}$ is easily obtained with the following equation:

$$\mathbf{q} = t\mathbf{q}_n \quad (7.8)$$

After the point $\mathbf{q}$ is known, we can apply Equation 7.5 to test whether the fragment falls inside the splat or not. Of course, we will not be done until we adjust the new fragment depth value using q_z as z' in Equation 7.4. Since the 3D point $\mathbf{q}$ is the *exact* preimage of the projective transform, this means that this method is perspectively correct for each pixel.

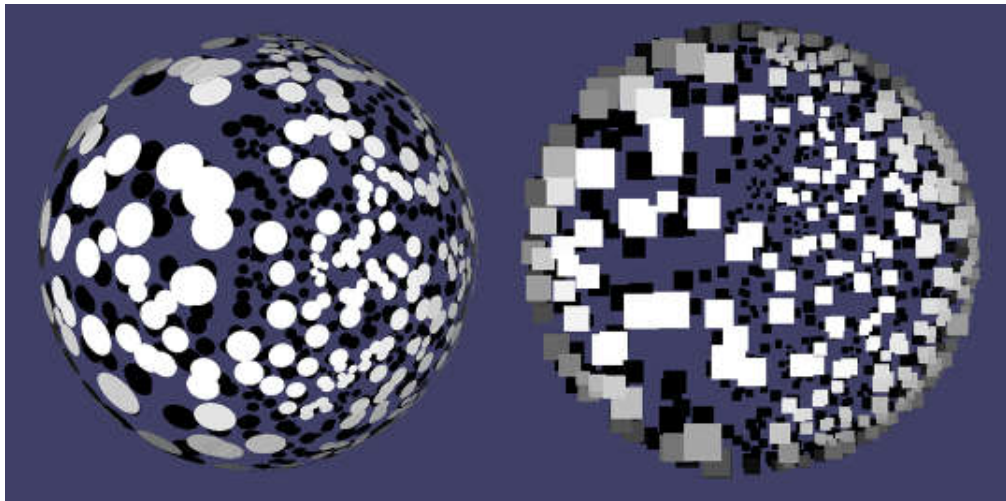


Figure 7.4: Perspectively correct rasterization represents the splat with a lot more fidelity than the initial technique, specially in the object contour (**left**). Image-aligned squares centered in the splat position (**right**).

7.3 Antialiasing

The chosen method to remove anti-aliasing artifacts in the visualizer was **MSAA** [AMHH08]. The term usually refers to a special case of super-sampling. If the reader wants more information about super-sampling, feel free to check [Álvarez Mures12], SSAA was the anti-aliasing method used in that point-based rendering engine. That type of anti-aliasing works well in offline rendering, but for real-time rendering it is not a good option because of its computational cost.

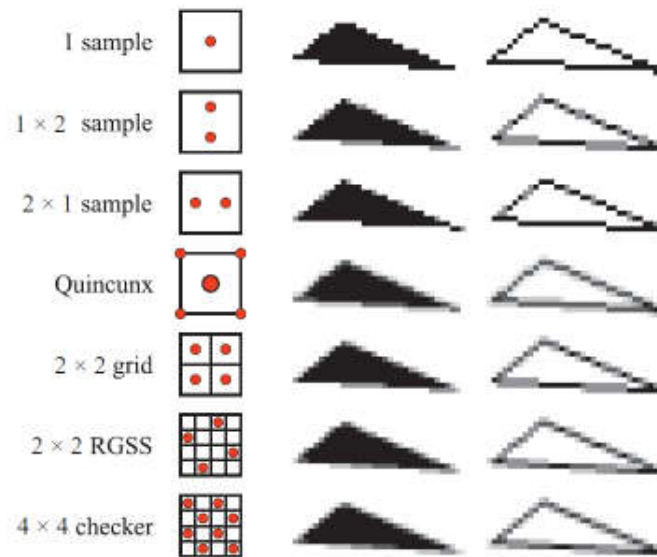


Figure 7.5: A series of typical super-sampling patterns applied to a triangle. Aliasing is reduced even when the resolution is not increased. Source: [AMHH08].

One way to reduce aliasing artifacts is through *oversampling*. Oversampling is the process of sampling a signal at a rate that is higher than the desired output. In the case of images this results in a reduction in the aliasing artifacts. When this is applied to graphics or 2D images we call it SSAA. We can apply lots of different super-sampling patterns and each one will yield a different result (see Figure 7.5).

In order to keep most of the benefits of SSAA without the performance penalty. Seeing that aliasing only occurs in the borders of geometric primitives, we will exploit this fact in our advantage. MSAA works in a similar manner to SSAA, the coverage and occlusion tests are performed at a higher sampling rate (from x2 to x16). For coverage, we have N samples within a pixel, where N is the multi-sample rate. A typical MSAA sample pattern can be observed in Figure 7.6.

With the sample points the primitive is tested for coverage building a *coverage mask*. For occlusion testing, the primitive depth is interpolated at each sub-sample. Where MSAA

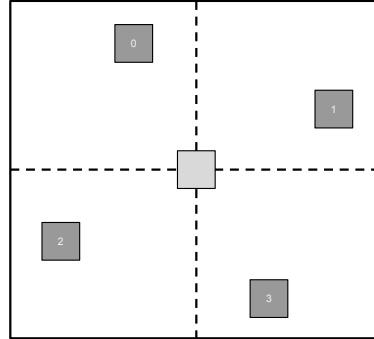


Figure 7.6: Subsample placement for a MSAA x4 rotated grid pattern.

starts differing from SSAA is in the pixel shader. It is not executed for each sub-sample, but once for each pixel with a coverage mask that is not zero. This means that the shader cost is not as big as with SSAA. When a pixel shader outputs a value, the value will be written only if both the coverage and depth tests are passed (see Figure 7.7).

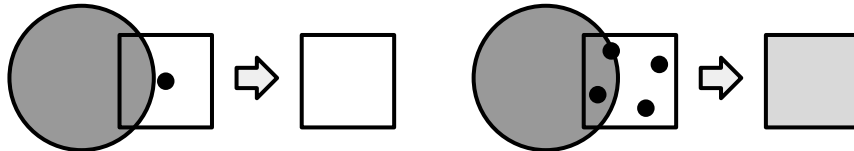


Figure 7.7: Non-MSAA rendering does not support partially covered primitives and in this case would not color the pixel (**left**). MSAA rendering on the other hand would detect that some subsamples are covered and would average an output color from all the subsamples (**right**).

Once we have the oversampled signal, we have to resample it to the target resolution before it can be displayed. In MSAA this process is called *resolving* the render target. In the early days of MSAA, the resolve process was carried out in the GPU's hardware, using a one pixel wide box filter. Which in essence is the same as averaging all the subsamples of a given pixel.

As programmable shaders arrived, we now have the possibility to perform the MSAA resolve in a custom shader. In this project this step was not deemed a priority and we

have relied on the OpenGL API to perform it. But, these are not the only advantages, bandwidth usage is also improved when using MSAA versus SSAA. Since the pixel shader is only executed once per pixel, as a result the same value is often written to all the subsamples of the render target. GPUs can take advantage of this factor sending not only the pixel value, but the number of samples that have to be written, which in term is a form of loss-less compression.

Summing up, the use of MSAA has given us an increase in image quality without a big impact in performance. This increase in rendering quality is specially noticeable in dynamic cases, but can also be observed in static images (see Figure 7.8).

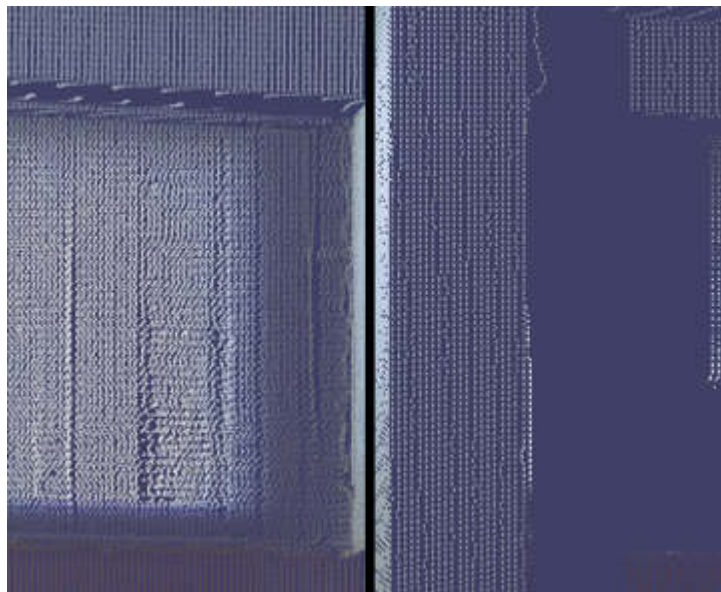


Figure 7.8: MSAA rendering obtains smoother surfaces without holes and less edge aliasing (**left**). Non-MSAA rendering results in holes and non-smooth edge gradients (**right**).

Chapter 8

Object detection: RANSAC

RANSAC is an abbreviation for “RANdom SAmple Consensus”. This is an iterative method to estimate parameters of a mathematical model from a set of observed points which can contain outliers. This is a non-deterministic algorithm, since we are not able to guarantee that it will produce a correct result. The probability of reaching a reasonable result will improve as the number of iterations increases. The algorithm was first published in [FB81].

The basic assumption is that the data contains *inliers*¹, though it may contain noise and *outliers*². Outliers can come from errors or imprecise measurements, extreme values of noise, etc. RANSAC assumes that given a set of inliers, there should exist a procedure that can estimate the unknown parameters of a model that will fit this data (see Figure 8.1).

8.1 Original algorithm

Unlike other estimation techniques, RANSAC generates candidate solutions using the minimum number of points required to fit the model parameters. RANSAC instead of using as much of the data as possible and then eliminate the outliers, uses the smallest set possible and then tries to enlarge this subset with coherent points. The basic algorithm can be summed up in the following steps:

1. Select randomly the minimum subset of points to determine the model parameters.
2. Solve for the parameters of the model (i.e. with a least squares method).

¹Data whose distribution can be estimated by some model.

²Data that does not fit the model.

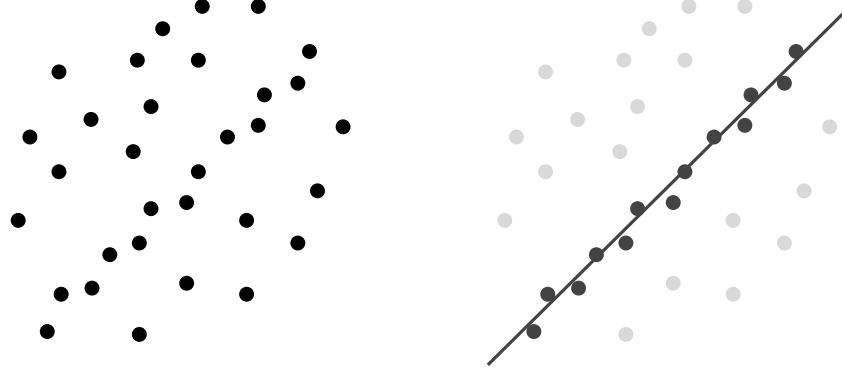


Figure 8.1: A set of points with many inliers in which a line is to be fitted (**left**). Resulting fitted line using RANSAC, in which outliers have no influence in the result (**right**).

3. Test how many points from the complete dataset fit with a predefined tolerance ϵ .
4. If the fraction of inliers over the total number of points is sufficiently good, then the model is re-estimated (since it was only estimated using the initial inliers).
5. Finally, the model is evaluated by estimating the error of the inliers relative to the model.

This procedure is repeated a certain number of times, each time producing either a new model which is either rejected because too few points are classified as inliers or a better model together with its error measure. In the latter case, we keep the refined model if its error is lower than the last saved model.

The algorithm will stop when it reaches N iterations or when the model reaches a desired error measure. N can be calculated to ensure that the probability p (usually 0.99) of one of the subsets of random samples will not contain an outlier. Being u the probability that any selected point is an inlier and $v = 1 - u$ the probability of observing an outlier. N iterations of the minimum number of points m are required:

$$1 - p = (1 - u^m)^N \quad (8.1)$$

After solving for N we will obtain the necessary number of iterations:

$$N = \frac{\log(1 - p)}{\log(1 - (1 - v)^m)} \quad (8.2)$$

Since our objective is estimating primitives in real-time, we will let the user choose this parameter so that computational time is not too high.

An advantage of RANSAC is the robust estimation of the model parameters (it can estimate the parameters even when a significant number of outliers is present). But, a disadvantage of RANSAC is that there is no upper bound on the time that it would take to compute the correct model parameters. Another disadvantage is that it can only estimate one model at a time, if two or more instances of the model exist, RANSAC will fail to find either of them.

8.2 Alternative algorithms

Since 1981 RANSAC has become one of the most widely used tools in computer vision. Since then, there have been several variations of the algorithm meant to improve its robustness, speed and accuracy. Some of the improved algorithms will be briefly described in this section.

8.2.1 MSAC and MLESAC

In the original algorithm, if the threshold ϵ is too high, the robust estimate can be very poor. This happens because RANSAC finds the minimum of a cost function:

$$C = \sum_i p(e_i^2) \quad (8.3)$$

Where $p(e^2)$ is:

$$p(e^2) = \begin{cases} 0 & e^2 < \epsilon^2 \\ \text{constant} & e^2 \geq \epsilon^2 \end{cases} \quad (8.4)$$

As we can see in the equations, inliers will score nothing and each outlier will score a constant penalty. As ϵ^2 is higher, more solutions with the same value C will appear, leading to worse estimations. In [TZ97] it is shown that this can be avoided changing C for:

$$C_2 = \sum_i p_2(e_i^2) \quad (8.5)$$

Where $p_2(e^2)$ is:

$$p_2(e^2) = \begin{cases} e^2 & e^2 < \epsilon^2 \\ \epsilon^2 & e^2 \geq \epsilon^2 \end{cases} \quad (8.6)$$

This is known as a re-descending M-estimator. Outliers are still given a constant penalty, but now inliers are scored depending on how well they fit the data. The implementation of this method is called MSAC³.

The definition of a maximum likelihood error, will allow to further improve the results of this new technique. This improved technique is called MLESAC⁴ [TZ99]. The improved algorithm steps are:

1. Get the minimum number of samples that will satisfy the model criteria.
2. Search for inliers.
3. Calculate distances to the model.
4. Use Expectation-Maximization to find out the right value for the penalty.
5. If the fraction of inliers over the total number of points is sufficiently good, then the model is re-estimated (since it was only estimated using the initial inliers).
6. Finally, the model is evaluated by estimating the error of the inliers relative to the model.

It can be seen that MSAC and MLESAC outperform the other two estimators, providing a 5 – 10% improvement according to [TZ99].

8.2.2 PROSAC

PROSAC⁵ is a new robust method, that exploits the linear ordering in a set of candidates using a similarity function for this purpose [CM05]. On one hand, RANSAC normally treats all correspondences as equals and drawing uniform random samples from the dataset. On the other hand, PROSAC samples are drawn from progressively bigger sets of the best ranked candidates.

Usually, RANSAC will generate N candidates. The set of ε candidates, will contain a now unknown number of inliers I . The RANSAC loop is terminated when the probability of

³M-Estimator Sample Consensus

⁴Maximum Likelihood Sample Consensus

⁵Progressive Sample Consensus

finding a better solution falls under a predetermined threshold. The average number of samples drawn is proportional to $(N/I)^m$. This method will be less expensive computationally than the original RANSAC, by using the linear order in ε . The set of candidates is obtained evaluating a similarity function $q(\cdot)$. After this step, the candidates are ordered according to the quality function. The algorithm will evaluate N candidates like RANSAC, but it will do it in a different order, the samples that are of higher quality (less contaminated) will be examined first. The function chosen $q(\cdot)$ can be the Euclidean distance or any other similarity metric.

The algorithm steps are in this case:

1. Choose the hypothesis generation set.
2. Generate hypothesis using random sampling (drawn from the subset of high quality data).
3. Search for inliers using the estimated model.
4. Model verification.
5. If the probability of finding a better solution falls below a threshold terminate.

According to [CM05], PROSAC is more than a hundred times faster than RANSAC in non-trivial cases.

8.2.3 RRANSAC

Typically, RANSAC will spend a lot of computational resources evaluating a large number of erroneous candidate models. These erroneous models will only fit a small subset of the complete dataset. This fact is exploited in [CM02] to create RRANSAC and significantly increase the speed of the original algorithm.

The main concept in this technique, is that the efficiency of the algorithm will be improved substantially when the hypothesis evaluation step is randomized. What this means, is that because most of the hypothesis are influenced by outliers, testing a small number of points d from the total N ($d \ll N$) will suffice to discard the bad solution.

This idea is implemented in a two-step evaluation procedure. First, a statistical test is performed on d randomly selected data points. Second, the evaluation of all the points in the dataset is only performed if the pre-test is passed. The preverification test is called the $T_{d,d}$ test, and is passed only when d data points out of d randomly selected are consistent with the hypothesis.

The only remaining task is knowing the optimal value of d . The following equation will yield that value:

$$d^* \approx \frac{\ln\left(\frac{\ln \varepsilon(t_M+1)}{N(\ln \delta - \ln \varepsilon)}\right)}{\ln \delta} \quad (8.7)$$

Being ε the fraction of inliers in the dataset, δ the probability that a data point is consistent with a “random” model and t_M is the time needed to compute the parameters of the model from the selected samples.

The algorithm steps in this instance are:

1. Choose the hypothesis generation set.
2. Generate hypothesis using random sampling.
3. Perform the pre-test on d points.
4. Model verification only if test is passed.
5. If the probability of finding a better solution falls below a threshold terminate.

It is interesting to mention that the speed boost of this algorithm can be combined with the more robust estimation of MSAC, resulting in a variation called RMSAC. This method combines the best properties of these two techniques, yielding fast and robust estimations.

8.3 Point-primitive distance calculation

Once the parameters of a model are estimated, a evaluation of the solution has to be performed. In order to do this, the outliers and inliers will be calculated. To know if a point is an outlier or an inlier, the distance from the point to the surface of the primitive will have to be computed. In the following subsections the math needed to measure the distance from planes, cylinders and spheres to a point will be explained.

8.3.1 Point-plane distance

The chosen convention to express the parameters of the plane is the *Hessian normal form*. These parameters will specify an infinite plane, that will have to have its length limited later on, in order for it to be used in the CAD exportation.

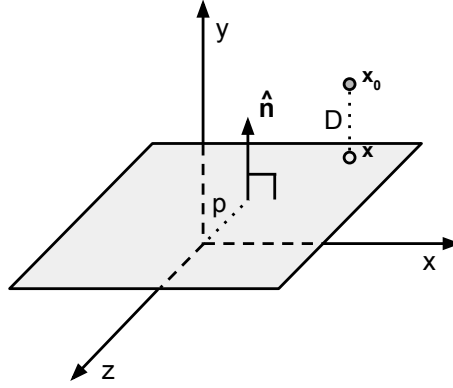


Figure 8.2: Visual explanation of the parameters involved in the distance calculation from a point to a plane in Hessian normal form.

This form can be obtained by calculating several parameters using the general parametric equation of a plane:

$$ax + by + cz + d = 0 \quad (8.8)$$

Thanks to this equation we can define the *unit normal vector* $\hat{\mathbf{n}}$:

$$\hat{\mathbf{n}} = (n_x, n_y, n_z) \quad (8.9)$$

Being n_x , n_y and n_z :

$$\begin{aligned} n_x &= \frac{a}{\sqrt{a^2 + b^2 + c^2}} \\ n_y &= \frac{b}{\sqrt{a^2 + b^2 + c^2}} \\ n_z &= \frac{c}{\sqrt{a^2 + b^2 + c^2}} \end{aligned} \quad (8.10)$$

And the constant p :

$$p = \frac{d}{\sqrt{a^2 + b^2 + c^2}} \quad (8.11)$$

Then the Hessian normal form of a plane is:

$$\hat{\mathbf{n}} \cdot \mathbf{x} = -p \quad (8.12)$$

Once we have the Hessian normal form parameters (see Figure 8.2) and a point $\mathbf{x}_0$, the

point-plane distance D is given by the following equation:

$$D = \hat{\mathbf{n}} \cdot \mathbf{x}_0 + p \quad (8.13)$$

8.3.2 Point-cylinder distance

In the case of the cylinder, the parameters chosen to represent it can be seen in Figure 8.3. These parameters will specify an infinite cylinder, that will have to be bound later in order for it to be useful.

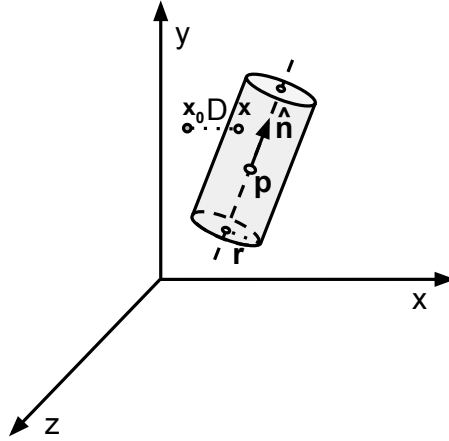


Figure 8.3: Visual representation of the parameters involved in the point-cylinder distance calculation.

The parameter $\hat{\mathbf{n}} = (n_x, n_y, n_z)$ represents the *unit normal vector* of its axis, $\mathbf{p} = (p_x, p_y, p_z)$ a point on its axis and $\mathbf{r}$ its radius. Furthermore, if we define two more unit vectors $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ as a right-handed orthonormal set $\{\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{n}}\}$, then any point in the cylinder $\mathbf{x}$ can be written as:

$$\mathbf{x} = \mathbf{p} + \hat{\mathbf{u}}x + \hat{\mathbf{v}}y + \hat{\mathbf{n}}z \quad (8.14)$$

Once we have every parameter and a point $\mathbf{x}_0$, we can calculate the distance D with the following equation:

$$D = \|\hat{\mathbf{n}} \times (\mathbf{x}_0 - \mathbf{p})\| \quad (8.15)$$

8.3.3 Point-sphere distance

For spheres, the parameters chosen to represent the estimated primitive can be seen in Figure 8.4.

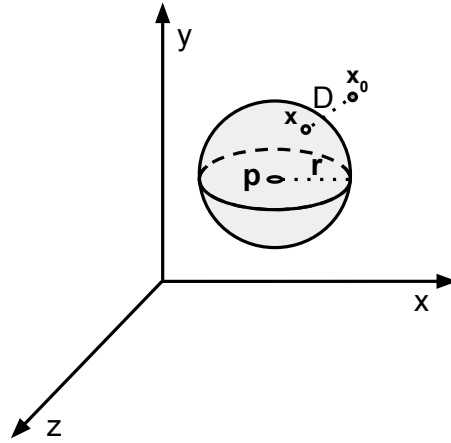


Figure 8.4: Visual explanation of the parameters involved in the point-sphere distance calculation.

The parameter $\mathbf{p} = (p_x, p_y, p_z)$ represents the center of the sphere and $\mathbf{r}$ the radius. The Cartesian equation of a sphere centered in $\mathbf{p}$ is:

$$(x - p_x)^2 + (y - p_y)^2 + (z - p_z)^2 = r^2 \quad (8.16)$$

When we have every parameter and a point $\mathbf{x}_0$, we can calculate the distance D with the next equation:

$$D = \|\mathbf{x}_0 - \mathbf{p}\| - r \quad (8.17)$$

8.4 Primitive bounding

Due to the fact that parametric cylinders and planes are infinite, in order to export them to CAD software we need to bound them. Therefore, we will use the bounding box of the complete dataset to establish these boundaries.

8.4.1 Plane bounding

To bound a plane, we will have to calculate the intersection between the $AABB$ ⁶ of the scene and a plane. This intersection will yield a three to six vertex polylineFigure 8.5.

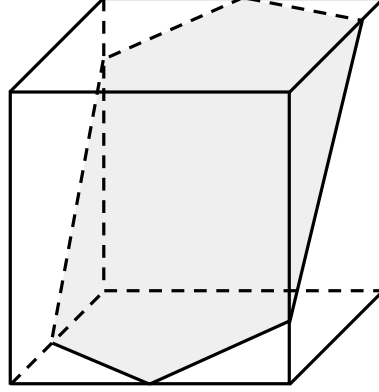


Figure 8.5: Visual representation of a AABB-plane intersection.

In order to obtain the polyline vertices, we will intersect every edge with the plane. This means that a common ray-plane intersection test has to be performed for each edge (see Figure 8.6). The point of intersection in the plane can be written as in Equation 8.12 and in the ray can be written as:

$$\mathbf{x} = \mathbf{x}_0 + t\hat{\mathbf{v}} \quad (8.18)$$

Being $\mathbf{x}_0$ the origin of the ray, $\hat{\mathbf{v}}$ the direction and t :

$$t = \frac{-(\mathbf{x}_0 \cdot \hat{\mathbf{n}} + p)}{\hat{\mathbf{v}} \cdot \hat{\mathbf{n}}} \quad (8.19)$$

Once all the intersection points have been obtained, they are tested to check if they fall inside the AABB. After this we will have the vertices of the bounded plane ready to be exported.

8.4.2 Cylinder bounding

In order to obtain the two points corresponding to the centers of the cylinder bases, we will have to perform a AABB-cylinder intersection test (Figure 8.7). To achieve this, we will carry out six ray-plane intersection tests.

⁶Axis Aligned Bounding Box

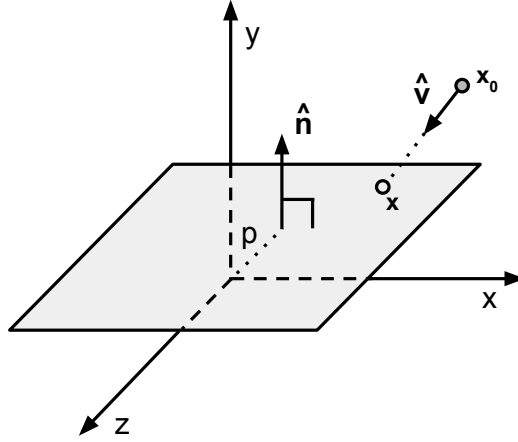


Figure 8.6: Visual representation of a ray-plane intersection.

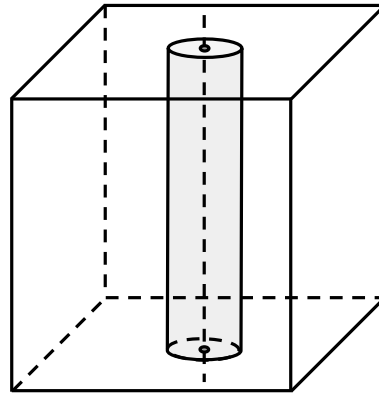


Figure 8.7: Visual representation of a AABB-cylinder intersection.

In this instance, the ray will be the cylinder axis that will be tested against each of the faces of the AABB. This intersection test will be performed in the same manner as in subsection 8.4.1, but the ray direction and position instead of being those of the AABB, will be those of the cylinder axis (see Figure 8.8). The planes against which the ray has to be tested will be the faces of the AABB.

Once we have all the intersection points, we will have more than two intersection points. That is why one extra step is needed to bound the cylinder, the sorting of the points according to the distance to the AABB center $\mathbf{c}$:

$$D = \|\mathbf{x} - \mathbf{c}\| \quad (8.20)$$

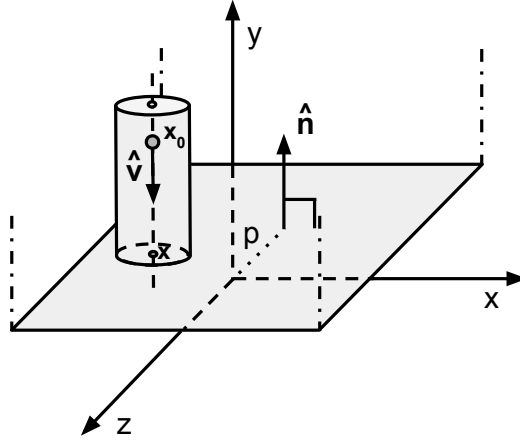


Figure 8.8: Visual representation of a ray-plane intersection with the cylinder axis as ray.

Finally, the two cylinder bases will be the two closest points to the AABB center. This is the last step necessary to get the cylinder ready for exportation.

8.5 Results

To test the different algorithms we will compare the estimation times achieved with them. We will first test a 3038 point sample, from which we want to estimate a plane (see Table 8.1).

Algorithm	Time(ms)
RANSAC	71.886
MSAC	71.8124
MLESAC	106.342
PROSAC	71.641
RRANSAC	71.5368

Table 8.1: Table showing the estimation times achieved with each algorithm.

As we can see, the fastest algorithm is RRANSAC. Although, we have to keep in mind that the most robust algorithms are MSAC and MLESAC after thorough testing. In light of these results, the chosen method to estimate primitives in ToView was MSAC. The following segmentations were performed using the aforementioned algorithm: Figure 8.9, Figure 8.10 and Figure 8.11.

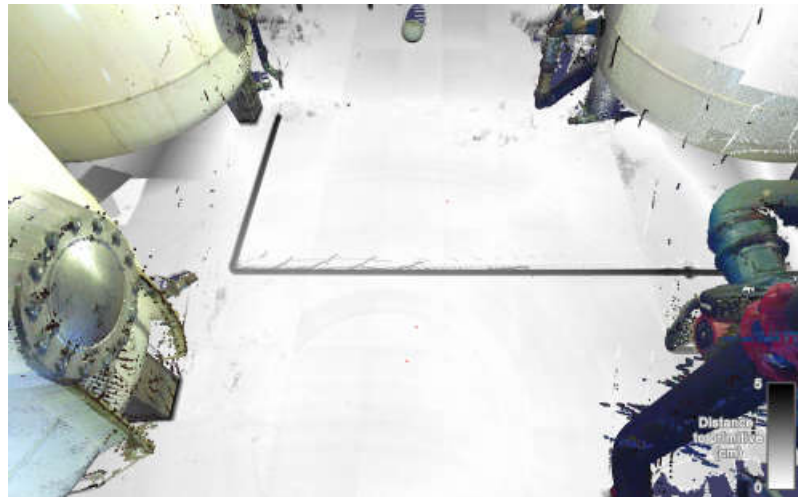


Figure 8.9: Segmented plane using ToView.



Figure 8.10: Segmented cylinder using ToView.

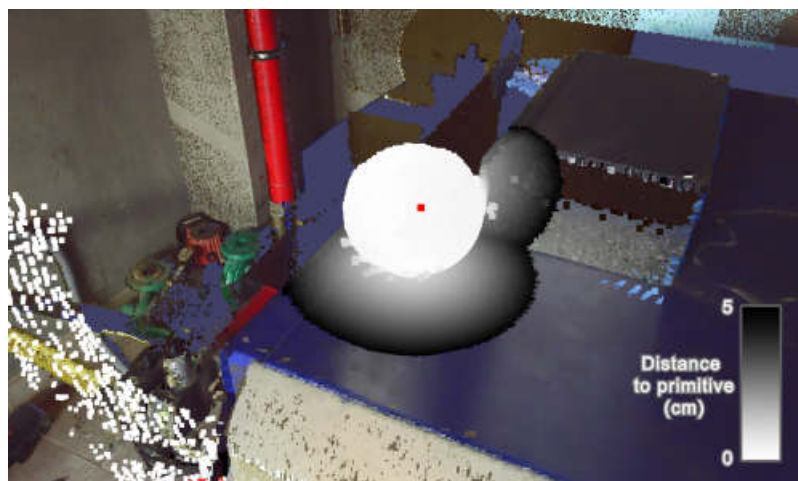


Figure 8.11: Segmented sphere using ToView.

Chapter 9

Preprocessing and filtering

Scanners nowadays yield point clouds with high detail and point count, but are not exempt from noise or other inconvenient artifacts. These scanners typically produce a dense set of points with position, color and a variety of other possible attributes. Artifacts can appear because of several reasons:

- Physical limitations of the sensor or motion artifacts, the latter can occur when scanning animals or humans.
- Multiple reflections and heavy noise.
- Holes that result from occluded parts of the scanned scene.
- Artifacts that appear when unifying multiple scans.

For high quality point-based rendering, some extra attributes may also have to be calculated, like point radii or normals if they are not provided.

These are some of the reasons why the raw point-cloud data needs to be preprocessed before it is ready for use.

9.1 Radii estimation

As we have mentioned before, points lack surface area by definition. That is why apart from needing the point normals, we will need the radius in order to obtain a high quality representation of its surface (see Figure 9.1).

In order to estimate the point radius, the first step will be obtaining its k -nearest neighbors $\mathcal{N}_k(\mathbf{p})$. Since we want a watertight surface, that is a surface bounding a closed solid, or

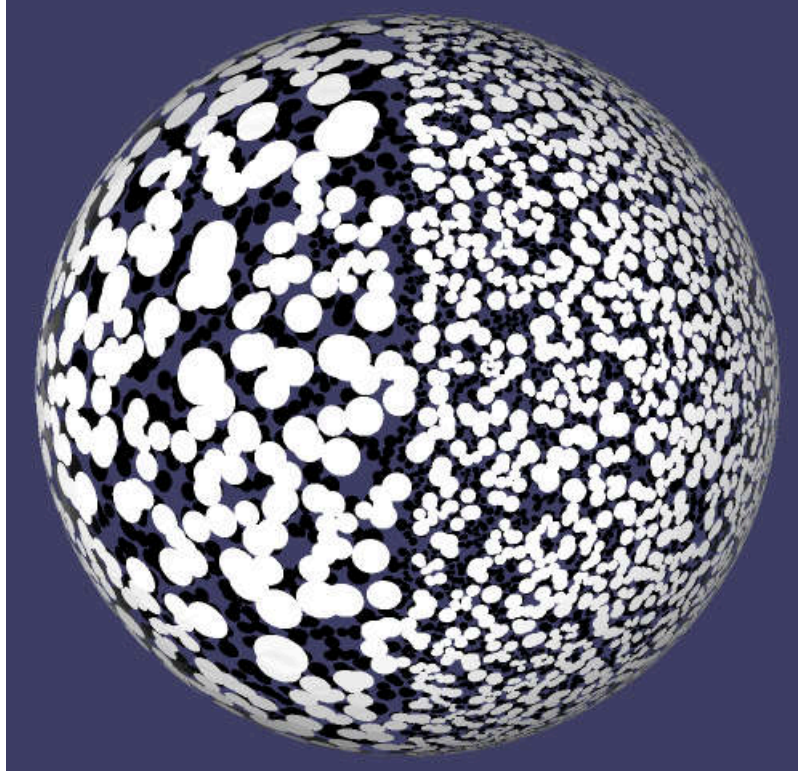


Figure 9.1: In the sphere we can see that when points are further apart (**left**) the resulting radii are bigger than when they are closer together (**right**).

even better a closed manifold, we will need to obtain the point radius with that in mind. If the point normal is available we will use:

$$\mathbf{r} = \max_j \|(\mathbf{p}_j - \mathbf{p}) - \mathbf{n}^T(\mathbf{p}_j - \mathbf{p})\mathbf{n}\|, \forall \mathbf{p}_j \in \mathcal{N}_k(\mathbf{p}) \quad (9.1)$$

Being $\mathbf{p}_j$ the neighbor position and $\mathbf{n}$ the point normal. If we do not have access to the point normal, the following equation can also be used to calculate the radius:

$$\mathbf{r} = \max_j \|\mathbf{p}_j - \mathbf{p}\|, \forall \mathbf{p}_j \in \mathcal{N}_k(\mathbf{p}) \quad (9.2)$$

With these two equations we will be able to obtain a point radius, that as mentioned previously will provide better results when rendering points.

9.2 Statistical outlier removal

As was mentioned in the introduction of this chapter, laser scanners typically generate scans with different point densities or noise. Moreover, measurement errors will lead to sparse outliers that will corrupt the resulting scan even more. These outliers will complicate further calculations, like radii or normal estimations, leading to visual artifacts or point cloud registration failures (see Figure 9.2).



Figure 9.2: Original dataset without filtering applied, ellipses highlight outliers (**left**). Resulting dataset without outliers after filtering is applied (**right**).

Some of this problematic points can be removed by performing a statistical analysis of the point neighborhood, and eliminating those that meet certain criteria. The sparse outlier removal algorithm implemented in this project, is based on the distribution of point-neighbor distances in the dataset.

For this purpose, we will start by computing the mean distance $\bar{D}$ to each point neighbor:

$$\bar{D} = \frac{\sum_j \|\mathbf{p}_j - \mathbf{p}\|}{k}, \forall \mathbf{p}_j \in \mathcal{N}_k(\mathbf{p}) \quad (9.3)$$

Assuming that the resulting distribution will be Gaussian in nature, with a mean and a standard deviation, points that have a mean distance that does not fall in an interval will be removed. This interval is given by the mean and standard deviation of all the global distances.

The following equation is used to calculate the global mean:

$$\mu = \frac{1}{N} \sum_j \bar{D}_j, \forall \bar{D}_j \in \mathcal{D}_N \quad (9.4)$$

To calculate the standard deviation we can use:

$$\sigma = \sqrt{\frac{1}{N} \sum_j (\bar{D}_j - \mu)^2}, \forall \bar{D}_j \in \mathcal{D}_N \quad (9.5)$$

Once we obtain these two values we can for each point check if the mean is abnormally superior to the standard deviation and remove the point if this is the case.

9.3 Voxel grid filter

Another possible issue when dealing with point clouds, is that the resulting cloud could be too dense for our purpose. In these instances, downsampling the cloud will be necessary for optimum results. In order to achieve this, we will use a voxelized grid approach. This filter will create a 3D voxel grid from the cloud data, and then for each voxel all of the points that belong to it will be approximated using its centroid (see Figure 9.3).

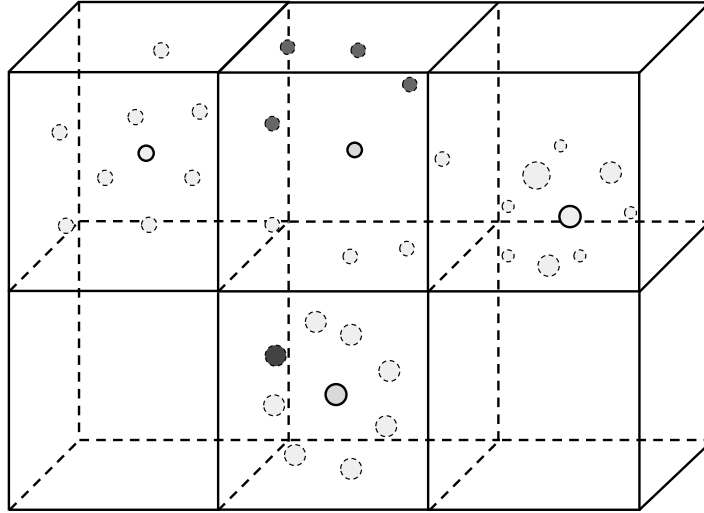


Figure 9.3: Visual representation of the process that the voxel grid filter performs.

Firstly, a 3D voxel grid is a set of 3D boxes that cover a certain amount of space (a voxel can be thought of as a 3D pixel). Once we have isolated the points that belong to a voxel $\mathcal{V}_N$, we can use the following equation to compute the centroid:

$$\mathbf{c} = \frac{1}{N} \sum_j \mathbf{p}_j, \forall \mathbf{p}_j \in \mathcal{V}_N \quad (9.6)$$

Being $\mathbf{p}$ the position of the corresponding point. In Equation 9.6 we can substitute the position for the color, normals or any other attributes; to obtain the rest of the centroid data. The resulting centroid will be the downsampled point corresponding to the voxel in our new decimated cloud (see Figure 9.4).

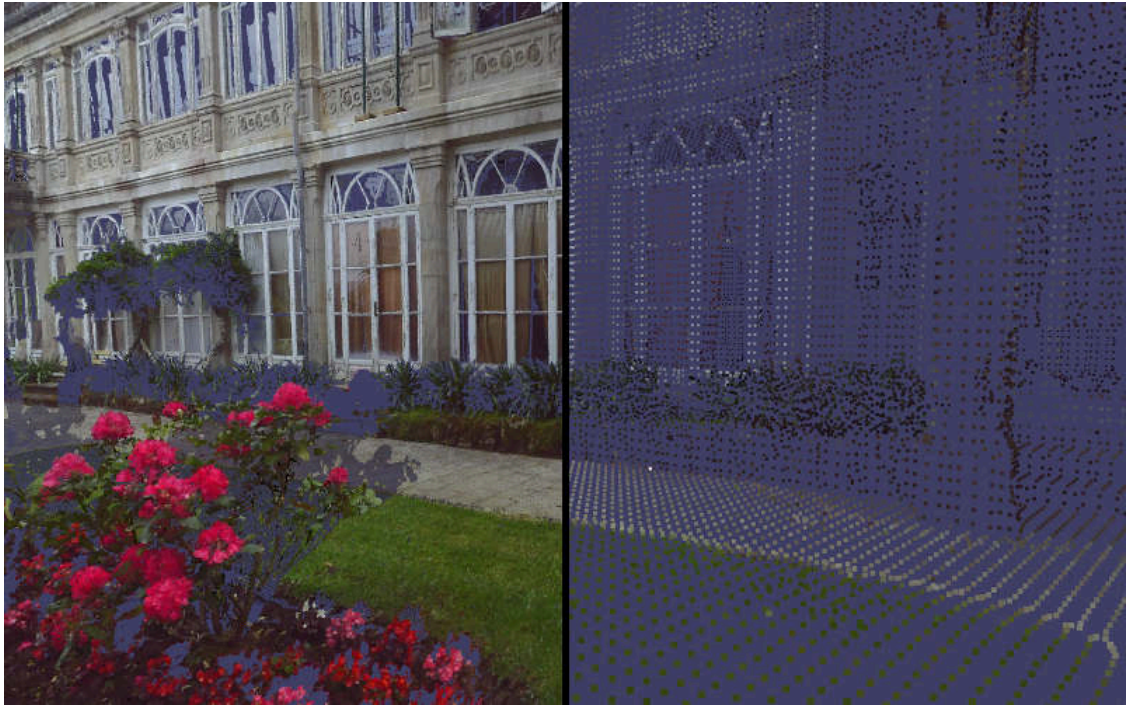


Figure 9.4: Original dataset without filtering applied (**left**). Resulting downsampled dataset after filtering is applied (**right**).

This filter can also be used in cases in which noise is present due to unification of multiple scans. Since the same point can be scanned from multiple scanning positions, this will lead to visual artifacts as the illumination can change from one scan to another. With this filter we can eliminate the excess points, setting the same precision as the laser scanner used to obtain the cloud.

Chapter 10

Conclusions and future lines of work

In this chapter we will briefly take a look at the conclusions reached after finishing this project, and the possible future lines of work that the project can follow.

10.1 Conclusions

The first conclusion reached, has been that all of the objectives of the project were met:

- Implement different visualization modes for point-based models.
- Combine different point clouds and transform them.
- Point picking.
- Apply operations to a part of the cloud or the complete dataset.
- Export the segmented objects to a CAD format.

After finishing and achieving the aforementioned objectives, the other conclusions that have been reached are:

- **The spatial acceleration structure is key:** The interactivity that will be achievable when rendering the different datasets, will strictly depend on the acceleration structure employed. If we decided not to use a k-d tree, we would not be able to achieve interactive rendering times, and the computations like radii estimation would take months.

- **Multi-resolution is not always optimal:** When trying to compute exact values (i.e. point radii without estimation errors), a multi-resolution structure can hinder performance quite a bit, since exact k-neighbor searches take a lot more time.
- **Interactivity with massive clouds:** It is certainly possible to achieve interactivity even with clouds of $1000M$ points. If advanced rendering modes are desired, it will require a long preprocessing time. But once all the point features are estimated, the user will be able to interact in real-time.
- **Too much density is not always good:** Sometimes when combining multiple high density scans, the result may need some preprocessing in order to achieve the best rendering quality. This is due to the fact that a lot of cluttered points in the same area, can cause clipping and undesired visual artifacts.
- **Best rendering quality:** The best rendering quality is achieved using perspective correct rasterization. This type of point rasterization yields the best results when approximating the splat shape on a custom shader.
- **Multi-threading:** The use of multiple threads can improve preprocessing and rendering times significantly on current processors. It can also improve interactivity since the GUI will not depend on the render thread.
- **GPGPU is not always the answer:** As seen in the results, if the workload is not complex enough to compensate for kernel setup time, GPU computation time will be higher than the time it takes using the CPU. This is why when trying to accelerate a workload, a GPU implementation may not always be the best option.

10.2 Future lines of work

After finalizing the work on this project, several ideas for the expansion of the visualizer come to mind:

- **WebGL:** Implementing a version of the visualizer in WebGL could make ToView usable in a greater variety of devices. That could mean that an engineer could use ToView right on the work site on his cellphone.
- **Streaming of point clouds:** In order to be able to utilize the visualizer on mobile platforms or on low-end machines, PCM would need the ability to stream point clouds over a network. This is due to the fact that the massive point clouds that ToView is able to display in real-time, are too big to fit on those devices.

- **Advanced point shading:** As of now, we determine correctly what pixels will a point cover. The next step would be shading this splats so that visual artifacts are minimized. There are several shading techniques that could be used to improve rendering quality, Gouraud shading, Phong shading, etc.
- **Improve k-d tree construction:** The multi-resolution framework right now has no guarantee that points are distributed uniformly across levels. A new k-d tree building technique could ensure that in each level a certain point precision is reached.
- **Improve level of detail algorithm:** The LOD algorithm in the visualizer is quite basic. A new technique taking into account human vision knowledge to improve the selection of tree nodes, would increase rendering quality substantially.
- **Hybrid point-polygon rendering:** Engineers will sometimes want to check how would a certain design impact a real environment. Right now we rely on plugins that obtain clouds relying on the sampling of polygons, but the optimal approach would be being able to directly render polygon models.

Bibliography

- [Álvarez Mures12] Luis Omar Álvarez Mures. Global Illumination for Point-based Rendering. Final Degree Project, BSc in Computer Science. Faculdade de Informática, Universidade da Coruña, Spain, July 2012.
http://kmelot.biblioteca.udc.es/record=b1488553~S1*gag. 100
- [AMHH08] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. ISBN 978-1-56-881424-7. AK Peters, third edition, 2008. 15, 100
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. ISBN 978-0-20-161641-5. Addison-Wesley, first edition, 1999. 9
- [Bec01] Kent Beck. Manifesto for agile software development. *Agile Alliance*, 2001. 7
- [BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering in modern gpus. *Pacific Graphics*, pages 335–343, 2003. 96
- [BSK04] Mario Botsch, Michael Spornat, and Leif Kobbelt. Phong splatting. *Proceedings of Symposium on Point-based Graphics*, pages 25–32, 2004. 98
- [CHK⁺00] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, and Alexander Neundorff. CMake. <http://www.cmake.org>, 2000.
- [CM02] Ondřej Chum and Jiří Matas. Randomized RANSAC with $T_{d,d}$ test. *Image and Vision Computing*, pages 448–457, 2002. 107
- [CM05] Ondřej Chum and Jiří Matas. Matching with PROSAC - Progressive Sample Consensus. *Computer Vision and Pattern Recognition*, 1:220–226, 2005. 106, 107

- [Coc01] Alistair Cockburn. *Agile Software Development*. ISBN 978-0-20-169969-2. Addison-Wesley Professional, first edition, 2001. 7
- [FB81] Martin Fischler and Robert Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Comm. of the ACM* 06, pages 381–395, 1981. 103
- [FvDFH12] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. ISBN 978-0-20-184840-3. Addison-Wesley Professional, third edition, 2012. 15
- [Gar13] Willow Garage. PCL. <http://www.pointclouds.org>, 2013. 51
- [GP07] Markus Gross and Hanspeter Pfister. *Point-based Graphics*. ISBN 978-0-12-370604-1. Morgan Kaufmann Publishers, first edition, 2007. 15
- [Jas12] Alberto Jaspe Villanueva. Point Cloud Manager - A Multiresolution System for Managing Massive Point Cloud Datasets. Master Thesis, MSc in High Performance Computing. Faculdade de Informática, Universidade da Coruña, Spain, September 2012.
<http://albertojaspe.net/academics/degree-dissertations>. 2
- [Jas13] Alberto Jaspe Villanueva. Real-time Multiresolution 3D Visualization System for Huge LIDAR Datasets. Final Degree Project, BSc in Computer Science, Information Technology Curriculum. Faculdade de Informática, Universidade da Coruña, Spain, September 2013.
http://kmelot.biblioteca.udc.es/record=b1515710~S1*gag. 2
- [Lan06] Jean-Philippe Lang. Redmine. <http://www.redmine.org>, 2006. 12
- [Mus12] Andrew Mustun. dxflib. <http://www.ribbonsoft.com/en/what-is-dxflib>, 2012. 88
- [NCE95] Haavard Nord and Eirik Chambe-Eng. QT. <http://qt-project.org>, 1995. 75
- [PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. ISBN 978-0-12-375079-2. Morgan Kaufmann Publishers, second edition, 2010. 20, 23
- [Pra12] Stephen Prata. *C++ Primer Plus*. ISBN 978-0-321-77640-2. Addison-Wesley, sixth edition, 2012. 41

- [SWH13] Graham Sellers, Richard S. Wright, and Nicholas Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. ISBN 978-0-32-190239-4. Addison-Wesley, sixth edition, 2013. 27
- [Tai09] Javier Taibo. Geotextura: Una arquitectura software para la visualización en tiempo real de información, 2009. 57
- [Tor05] Linus Torvalds. Git. <http://git-scm.com>, 2005. 12
- [TZ97] S. Torr and A. Zisserman. Robust parameterization and computation of the trifocal tensor. *Image and Vision Computing*, 15:591–607, 1997. 105
- [TZ99] S. Torr and A. Zisserman. Mlesac: A new robust estimator with application to estimating image geometry. *Computer Vision and Image Understanding*, 78(1):138–156, 1999. 106
- [ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. *SIGGRAPH 2001 Proceedings*, pages 371–378, 2001. 93
- [ZRB⁺04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. *Proceedings of Graphics Interface*, pages 247–254, 2004. 98