Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Design and development of a generalized LIDAR point cloud streaming framework over the web

Laureando

Alberto Nale

Relatore

Prof. Massimo Rumor

Anno Accademico 2014/2015

To my grandfather
and Don Valentino

My greetings go to my family and to all those who have supported me during this long journey. Thanks to my mother and father and to my sister and brother-in-law for the trust they placed in me and for the patience whereas they were waiting for this goal.

"C'est le temps que tu
a perdu pour ta rose
qui fait ta rose si importante."

Antoine de Saint-Exupéry, Le Petit Prince

"È il tempo che tu
hai perduto per la tua rosa
che ha fatto la tua rosa così importante."

Antoine de Saint-Exupéry, Il Piccolo Principe

Contents

1 Introduction **3**

2 Objectives and reasons **5**
2.1 Database choice . . . 6
2.2 Tiling . . . 6
2.3 Decimation operation . . . 6
2.4 Three.js . . . 6
2.5 Web Services . . . 7
2.6 Measurement . . . 7

3 Preliminary notions **9**
3.1 LIgth Detection And Ranging (LIDAR) . . . 9
3.1.1 las format . . . 10
3.2 Point cloud . . . 11
3.3 Level of detail (LOD) . . . 11
3.4 No-SQL . . . 12
3.5 LIDAR data managing . . . 16

4 Software and libraries **17**
4.1 PostgreSQL . . . 17
4.1.1 Libpqxx . . . 17
4.2 PostGIS . . . 18
4.3 WebGL . . . 19
4.4 Three.js . . . 21
4.4.1 BufferGeometry . . . 22
4.5 PCL (Point Cloud Library) . . . 22

5 Data Management **25**
5.1 Problem . . . 25
5.2 Point Cloud dataset . . . 26
5.3 Tiling . . . 26
5.3.1 Specific case study analysis . . . 27

5.3.2 MongoDB 27
5.4 Point cloud simplification 28

6 Implementation 31
6.1 System 31
6.2 LOD creation 31
6.2.1 Algorithm Overview 33
6.2.2 Space Optimization 34
6.3 C++ interface code 34
6.4 Visualization 35

7 Tests and Results 37
7.1 Padua LIDAR data 37
7.2 LOD decimation results 38
7.3 Measurement on LIDAR 40
7.3.1 Implementation 41
7.3.2 Points sticking 42
7.4 Measurement results 42

8 Conclusions 45
8.1 Future Improvements 45

List of Figures 47

List of Tables 48

Bibliography 51

Abstract

Modern technologies give us large amount of data we have to store, manage and use. LIDAR data, retrieved by laser systems carried by airplanes, helicopters or cars, require the storage of millions of point information concerning the space we are scanning. A very important requirement when dealing with these data is velocity of retrieval, not a trivial problem for a web application, so it is important to find a fast method to stream LIDAR data. This work illustrates the implementation of a web application which gets LIDAR data from a PostreSQL database and, using an interface written in C++ language, creates n different levels of detail of the point cloud data and saves them into related files. Afterwards data are read from the files and shown on the screen. A first operation implemented on the data displayed is linear point to point measurement.

Chapter 1

Introduction

In recent years, we have witnessed the emergence of a new class of applications that must deal with large volumes of streaming data such as financial data analysis on feeds of stock tickers, sensor-based environmental monitoring, and network traffic monitoring. Traditional *database management systems* (DBMS) which are very good at managing large volumes of stored data, fall short in serving this new class of applications, which require low-latency processing on live data from push-based sources.

Today more than ever, large amounts of data are available on the web to be visualized and consulted. Millions of video can be viewed on Youtube, a lot of people play games online, some TV channels allow to watch their programs in streaming simply connecting to their websites. This large amount of data entails to improve technology to manage them. Big data have to be analyzed to find useful information and maybe reduce their size by removing useless points. This large amount of data has to be cleverly stored, managed and processed. One of the most important problems is the efficient streaming of these data. Data streaming is now used in different kinds of application including mobile apps and web sites. The widespread presence of devices such as smartphones, tablets, smart TVs and media players force the web to evolve to sustain near real-time data transfer to provide streaming services.

Another area of interest where big data are very important is GIS environment. A Geographic Information System (GIS) is a system built to catch, store, manage and represent geographic data. In the geospatial sphere, large amounts of data descend from modern technologies such as LIDAR. LIDAR is a remote sensing technology that measures distance by illuminating a target with a laser and analyzing the reflected light. Hardware is connected to flying machines used to collect geographical information from different places in the world. Data then can be downloaded and are ready to be analyzed and processed. Point clouds collected in that way are regularly used to build

a model of the surface of the scanned area with the intent of studying that portion of reality.

LIDAR data usage covers various sectors, such as gaming, networking, navigation, 3D imaging and so on. LIDAR allows any physical object to be re-created in a computer domain. Whether it is a building, a car or a whole country, LIDAR has the ability to reproduce a number of scenes. Once in the digital realm they can be rendered using a pseudo-color representation of the real-world point or they can be transferred to a 3D software and be re-created with actual photographic textures mapped onto the surfaces. This enables the user to create highly detailed, accurate models in a very short space of time when compared to other modeling methods. In the gaming industry this can be used for a number of purposes. It will allow the quick and precise creation of whole cities.

It can also be used to recreate every undulation in a race track, giving players the most accurate reproduction of their favourite circuit. As regard their real-world application, LIDAR systems make recording the scene of accidents and crimes quick and easy. By using a ground based LIDAR system it is possible to record the scene of a car accident within a few minutes, enabling emergency services to clear the scene and then to reproduce it later on in the digital realm. This reduces traffic jams as well as preserves the evidence before something is compromised. All of the data are recorded with a geographical position that allows them to be used in various software packages for an extra level of accuracy.

Chapter 2

Objectives and reasons

The need of this work was the implementation of a framework capable of transforming raw LIDAR data in order to visualize them in streaming over the web. Nowadays you can download and install different tools able to manage point cloud data.
So why did we want to build a framework to stream data over the web? LIDAR data can be difficult to use for many processes without specific technical software. Streaming them over the web allows to centralize data access decreasing data storage requirements and hardware costs. The only requirement is an available Internet connection. In this way there are no hardware dependencies and, without using any additional plugin, you can directly access and visualize data from your web browser. Another important advantage is the convergence of the storage: there aren't local copies of the data so there is no need to line up local modifies reducing the possibility of data inconsistency. This benefit allows the total control of updates and upgrades operations applying changes directly to the source data. Improvements and fixes are applied once and are available for everyone in every place of the world.
So, in order to solve a complicated problem, break it into different pieces is an efficient strategy that allows to build the solution step by step. The first part of the development was occupied by the analysis of the problem: it was necessary to find out a technique to manage large amount of data. In this phase the problem was decomposed into different small pieces. First of all LIDAR data have to be saved into specific structures which could fast calculate them.
After that the need was to process them to reach the aim of level of detail creation. The next chapters will explain what the problems faced during this specific task were. The last objective was the realization of linear measurement of data drawn on the screen.

Below is an explanation of the steps experienced to build the framework.

2.1 Database choice

LIDAR data contains millions of points that have to be saved into efficient database structures. The database has to be able to manage also geographic information contained into LIDAR data. For this reason we chose the PostgreSQL database with the PostGIS extension: it adds support for geographic objects allowing location queries to be run in SQL. This database implements very useful specific functions (called spatial queries) to rapidly retrieve data.

2.2 Tiling

The first trials underline the difficulty of working with large amounts of data. Thereby the most efficiently and useful technique to adopt is to subdivide the point cloud into small pieces and apply the level of detail creation to them. In this way the number of data you are working with is greatly downsized allowing to reduce computing capacity and memory requirement. This subdivision was parametrized by the number of tiles: the database was partitioned into a fixed number of parts indexing each tile in order to easy organize it later on the screen.

2.3 Decimation operation

A different thing are algorithms to decimate point cloud data. At this stage we focused on the analysis of different software, libraries and applications that could be useful to this aim. Important aspects to consider were: license, resource requirements and accuracy of the used algorithm.

2.4 Three.js

Three.js is the most used library for 3D processing. So we obviously chose it to implement the virtual environment. Three.js provides different structures specifically developed to have high performance with 3D points. In particular, the THREE.PointCloud structure uses the BufferGeometry: structure with better performances compared with its own Geometry. This is a very efficient class for geometry management as it saves all data in buffers.

2.5 Web Services

Framework construction involved also the implementation of different services providing data to write on the screen. Two PHP services were built: one to pass data to the scene and one to make the measurement operation. In the first one service reads the points from the LOD files and passes them to the Three.js environment; in the second one the data are searched into the database.

2.6 Measurement

Measurement was the following phase: one of the big problems to solve was catching points on the screen. THREE.RayCaster allows you to easily throw rays from mouse pointer to the scene and retrieve the caught point. To simplify the usability of the software we decided to help the measurement operation implementing a point sticking algorithm. In this way when the mouse is moving into the scene, the nearest point to the ray threw is found and cursor is hooked into. Another aspect to consider was measure precision: point drawn on the screen could not be into the database because of the centroid approximation of the LOD generator algorithm. In this way thinking measurement has to depend only on the real point into the database.

Chapter 3

Preliminary notions

Below is a simple description of the concepts used in this dissertation to better understand the terminology used to explain this work. A deep analysis of these concepts lies outside this work, but more information can be easily retrieved on the web.

3.1 LIgth Detection And Ranging (LIDAR)

LIDAR (also written LIDAR or LIDAR) is a remote sensing technology that measures distance by illuminating a target with a laser and analyzing the reflected light. Although widely considered an acronym for LIght Detection And Ranging, the term LIDAR was actually created as a portmanteau of "light" and "radar." LIDAR is popularly used as a technology to make high-resolution maps, with applications in geomatics, archeology, geography, geology, geomorphology, seismology, forestry, remote sensing, atmospheric physics, airborne laser swath mapping (ALSM), laser altimetry, and contour mapping.

LIDAR, which stands for Light Detection and Ranging, is a remote sensing method that uses light in the form of a pulsed laser to measure ranges (variable distances) to the Earth. These light pulses - combined with other data recorded by the airborne system - generate precise, three-dimensional information about the shape of the Earth and its surface characteristics. A LIDAR instrument principally consists of a laser, a scanner, and a specialized GPS receiver. Airplanes and helicopters are the most commonly used platforms for acquiring LIDAR data over broad areas. Two types of LIDAR are topographic and bathymetric. Topographic LIDAR typically uses a near-infrared laser to map the land, while bathymetric LIDAR uses water-penetrating green light to also measure seafloor and riverbed elevations. LI-

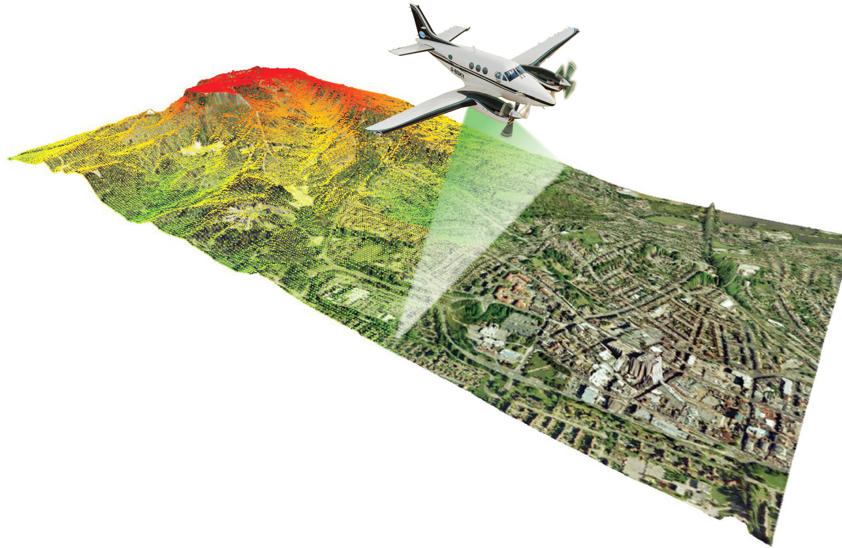

Figure 3.1: Example of LIDAR scanning operation

DAR systems allow scientists and mapping professionals to examine both natural and manmade environments with accuracy, precision, and flexibility. NOAA scientists are using LIDAR to produce more accurate shoreline maps, make digital elevation models to use in geographic information systems, to assist in emergency response operations, and in many other applications.

LIDAR data come from laser measurement systems put on plane and allow to obtain high precision topographic measurements. Raw data consist in a point cloud irregularly disposed on the area. After the elaboration of this points you obtain the DTM (Digital Terrain Model) and the DSM (Digital Surface Model): the former is obtained from punctual data "last pulse", corresponding to the 3D representation of the physical surface of the ground; the latter is obtained from punctual data "first pulse", corresponding to the 3D representation of all reflecting surfaces (ground, vegetation, buildings, etc...).

Aerial LIDAR has been used for over a decade to acquire highly reliable and accurate measurements of the Earth's surface. In the past few years, terrestrial LIDAR systems were produced by a small number of manufacturers.

3.1.1 las format

LIDAR data are collected from LIDAR devices and organized into a special format called las (LASer format). The LAS file format is a public file format

for the interchange of 3-dimensional point cloud data among data users. This binary file format is an alternative to proprietary systems or a generic ASCII file interchange system used by many companies. Las data are obtained by processing the device data with a software which combines GPS, IMU, and laser pulse range data to produce X, Y, and Z point data. The intention of the data format is to provide an open format that allows different LIDAR hardware and software tools to output data in a common format [1].

3.2 Point cloud

A point cloud is a set of data points in some coordinate system. In a three-dimensional coordinate system, these points are usually defined by X, Y, and Z coordinates, and are often intended to represent the external surface of an object. Point clouds may be created by 3D scanners. These devices measure in an automatic way a large number of points on the surface of an object, and often output a point cloud as a data file. The point cloud represents the set of points the device has measured.

Below you can see a data set produced by a "simulated" LIDAR camera that was flown over the MOUT model in a sweeping pattern. The data files contain xyz information, rgba color data, and the index of the camera frame in which the point was imaged. The "simulated" LIDAR data was acquired by passing rays through the cameras center horizontal scanline and the intersection of the closest visible polygon was recorded. Also, the color information from that polygon's texture was recorded.

3.3 Level of detail (LOD)

In computer graphics, accounting for level of detail involves decreasing the complexity of a 3D object representation as it moves away from the viewer or according to other metrics such as object importance, viewpoint-relative speed or position. Level of detail techniques increase the efficiency of rendering by decreasing the workload on graphics pipeline stages, usually vertex transformations. The reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or fast moving. Level of detail is a widely used technique for virtual reality to control the quality of 3D models' display in real time. It automatically changes display/non-display of 3D objects, as well as display of the objects' geometry

[1] ASPRS LAS specifications are available to:http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html

Figure 3.2: Point cloud visualization example

and texture images with appropriate data amount and resolutions, according to the distances between the viewing positions and viewed objects, and the viewing angles.

3.4 No-SQL

A NoSQL or Not Only SQL database provides a mechanism for storage and retrieval of data that is modeled in ways other than the tabular relations used in relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability. The data structure (e.g. key-value, graph, or document) differs from the RDBMS, and therefore some operations are faster in NoSQL and some in RDBMS. There are differences though, and the particular suitability of a given NoSQL DB depends on the problem it must solve.

NoSQL databases are increasingly used in big data and real-time web applications. NoSQL systems are also called "Not only SQL" to emphasize that they may also support SQL-like query languages. Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability and partition tolerance. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages, the lack of standardized interfaces, and huge investments in existing SQL. Most NoSQL stores lack true ACID transactions, although a few recent systems, such as FairCom c-treeACE, Google Spanner and FoundationDB, have made them central to their designs.

The are four significant categories of NoSQL:

- *Key-values Stores*: These ones use a hash table where there is a unique key and a pointer to a particular item of data. The Key/value model is the simplest and easiest one to implement but it is inefficient when you are only interested in querying or updating part of a value, among other disadvantages. Performance is enhanced to a great degree because of the cache mechanisms that accompany the mappings. To read a value you need to know both the key and the bucket because the real key is a hash (Bucket+ Key). There is no complexity around the Key Value Store database model as it can be implemented in a breeze. Not an ideal method if you are only looking for just updating part of a value or querying the database. When we try and reflect back on the CAP theorem [2], it becomes quite clear that key value stores are great around the Availability and Partition aspects but definitely lack in Consistency. A Key/value type database seems helpful in some cases, but it has some weaknesses as well. One is that the model will not provide any kind of traditional database capabilities (such as atomicity of transactions, or consistency when multiple transactions are executed simultaneously). These capabilities must be provided by the application itself. Secondly, as the volume of data increases, maintaining unique values as keys may become more difficult; addressing this issue requires the introduction of some complexity in generating character strings that will remain unique among an extremely large set of keys.

- *Column Family Stores*: These are useful to store and process very large amount of data distributed over many machines. There are still keys but they point to multiple columns. The columns are arranged by column family. One key difference between a key-value store and a document store is that the latter embeds attribute metadata associated with stored content, which essentially provides a way to query the data based on the contents.

- *Document Databases*: These were inspired by Lotus Notes and are similar to key-value stores. The model is basically versioned documents

[2] CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- *Consistency*: all nodes see the same data at the same time;
- *Availability*: a guarantee that every request receives a response about whether it was successful or failed;
- *Partition tolerance*: the system continues to operate despite arbitrary message loss or failure of part of the system.

that are collections of other key-value collections. The semi-structured documents are stored in formats like JSON. Document databases are essentially the next level of Key/value, allowing nested values associated with each key. Document databases support querying more efficiently.

- *Graph Databases*: Instead of tables of rows and columns and the rigid structure of SQL, a flexible graph model is used which, again, can scale across multiple machines. NoSQL databases do not provide a high-level declarative query language like SQL to avoid overtime in processing. Rather, querying these databases is data-model specific. Many of the NoSQL platforms allow for RESTful interfaces to the data, while other offer query APIs.

Table 3.1 shows a quick comparison of the most important features of NoSQL DBs. Generally, the best places to use NoSQL technology are where the data model is simple; where flexibility is more important than strict control over defined data structures; where high performance is a must; where strict data consistency is not required; and where it is easy to map complex values to known keys. Another important thing to consider is the scalability: the system has to be easy improved and integrated with new data and new future hypothetical features. For our purpose first two categories are both interesting. Our system will have to sustain repeatedly queries to fetch data during the navigation.

	Key-values Stores	Column Family Stores	Document Databases	Graph Databases
Typical applications	Content caching (Focus on scaling to huge amounts of data, designed to handle massive load), logging, etc.	Distributed file systems	Web applications (Similar to Key-Value stores, but the DB knows what the Value is)	Social networking, Recommendations (Focus on modeling the structure of data - interconnectivity)
Data model	collection of Key-Value pairs	Columns -> Column families	Collections of Key-Value collections	"Property Graph" - Nodes
Strengths	Fast lookups	Fast lookups, good distributed storage of data	Tolerant of incomplete data	Graph algorithms e.g. shortest path, connectedness, n degree relationships, etc.
Weaknesses	Stored data has no schema	Very low-level API	Query performance, no standard query syntax	Has to traverse the entire graph to achieve a definitive answer. Not easy to cluster.
Examples	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB	Cassandra, HBase, Riak	CouchDB, MongoDb	Neo4J, InfoGrid, Infinite Graph

Table 3.1: No-SQL compare table

3.5 LIDAR data managing

Traditional DBMS are useful to store data which do not change in time. When you are working with LIDAR data this approach is may be not too efficient because of queries performance but it can be improved using specific structures. What kind of structure do we have to use?

- *Quadtrees* Quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees work best for data that are mostly two dimensional like map-rendering in navigation systems. In this case it's faster than octrees because it adapts better to the geometry and keeps the node-structures small.

- *Octrees* An octree is a tree data structure in which each internal node has exactly eight children. Each node in an octree subdivides the space it represents into eight octants. It benefits if the data are three dimensional. It also works very well if your geometric entities are clusterd in 3D space. The benefit of Oc- and Quadtrees is that you can stop generating trees anytime you wish. If you want to render graphics using a graphic accelerator it allows you to just generate trees on an object level and send each object in a single draw-call to the graphics API. This performs much better than sending individual triangles (something you have to do if you use BSP-Trees to the full extent).

- *BSP-Trees* are a special case really. They work very well in 2D and 3D, but generating good BSP-Trees is an art form on its own. BSP-Trees have the drawback that you may have to split your geometry into smaller pieces. This can increase the overall polygon-count of your data-set. They are nice for rendering, but they are much better for collision detection and ray-tracing. A nice property of the BSP-trees is that they decompose a polygon-soup into a structure that can be perfectly rendered back to front (and vice versa) from any camera position without doing an actual sort. The order from each viewpoint is part of the data-structure and done during BSP-Tree compilation.

The most used structure in 3D processing are otcrees. Octrees are obtained by dividing recursively by 2 the grid you are considering. Starting from the initial cube, by dividing recursively by 2 you obtain the quadtree bringing from one to eight cubes. Iterating this operation you obtain octree. These subdivisions are made considering data inserted into the initial cube. The subdivision involves only cubes containing points. In particular, if the sub-cube is totally empty or totally filled this operation is stopped. This expedient allows to save memory space and reduce useless operations.

Chapter 4

Software and libraries

This chapter focuses the attention on the software and libraries involved in the research activity to evaluate their performance and features.

4.1 PostgreSQL

"PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, among others, and exceptional documentation."

Figure 4.1: PostgreSQL logo

4.1.1 Libpqxx

Libpqxx is the official C++ client API for PostgreSQL. The source code for libpqxx is available under the BSD license. This library allows you to connect

to the PostgreSQL db and retrieve data by querying it. With this library you have full database control so you can read, write and modify it. This library works on top of the C-level API library, libpq. Coding with libpqxx revolves around transactions. Transactions are a central concept in database management systems, but they are widely under-appreciated among application developers. In libpqxx, they are fundamental. With conventional database APIs, you issue commands and queries to a database session or connection, and optionally create the occasional transaction. In libpqxx you start with a connection, but you do all your SQL work in transactions that you open in your connection. You commit each transaction when it is complete; if you do not, all changes made inside the transaction get rolled back. There are several types of transactions with various "quality of service" properties; if you do not really want to use transactions at all, one of the available transaction types is called nontransaction. This transaction type provides basic non-transactional behaviour. Every command or query returns a result. Your query fetches its result data immediately when you execute it, and stores it in the result. Result objects can be kept around for as long as they are needed, completely separate from the connections and transactions that originated them. You can access the rows in a result using standard iterators, or more like an array using numerical indexes. Inside each row you can access the fields by standard iterators, numerical indexes, or using column names.

4.2 PostGIS

PostGIS is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL. PostGIS is released under the GNU General Public License (GPLv2). PostGIS is an extension to the PostgreSQL object-relational database system which allows GIS (Geographic Information Systems) objects to be stored in the database.

Figure 4.2: Postgis logo

A spatial database is a database optimized to store and query data that represents objects defined in a geometric space. Most spatial databases allow representing simple geometric objects such as points, lines and polygons. Some spatial databases handle more complex structures such as 3D objects,

topological coverages, linear networks, and TINs. While typical databases are designed to manage various numeric and character types of data, additional functionality needs to be added for databases to process spatial data types efficiently. These are typically called geometry or feature.

A spatial database provides a complete set of functions for analyzing geometric components, determining spatial relationships, and manipulating geometries. PostGIS turns the PostgreSQL Database Management System into a spatial database by adding support for three features: spatial types, indexes, and functions. PostgreSQL is a powerful, object-relational database management system (ORDBMS).

4.3 WebGL

"OpenGL is an application programming interface - "API" for short - which is merely a software library for accessing features in graphics hardware. Version 4.3 of the OpenGL library contains over 500 distinct commands that you use to specify the objects, images, and operations needed to produce interactive three-dimensional computer graphics applications. OpenGL is designed as a streamlined, hardware-independent interface that can be implemented on many different types of graphics hardware systems, or entirely in software (if no graphics hardware is present in the system) independent of a computer's operating or windowing system. As such, OpenGL doesn't include functions for performing windowing tasks or processing user input; instead, your application will need to use the facilities provided by the windowing system where the application will execute. Similarly, OpenGL doesn't provide any functionality for describing models of three-dimensional objects, or operations for reading image files (like JPEG files, for example). Instead, you must construct your three-dimensional objects from a small set of geometric primitives (points, lines, triangles, and patches)."

Figure 4.3: WebGL logo

"The following list briefly describes the major operations that an OpenGL application would perform to render an image.

- Specify the data for constructing shapes from OpenGL's geometric primitives.

- Execute various shaders to perform calculations on the input primitives to determine their position, color, and other rendering attributes.

- Convert the mathematical description of the input primitives into their fragments associated with locations on the screen. This process is called rasterization.

- Finally, execute a fragment shader for each of the fragments generated by rasterization, which will determine the fragment's final color and position.

- Possibly perform additional per-fragment operations such as determining if the object that the fragment was generated from is visible, or blending the fragment's color with the current color in that screen location." [1]

"WebGL is a royalty-free, cross-platform API that brings OpenGL ES 2.0 to the web as a 3D drawing context within HTML, exposed as low-level Document Object Model inter faces. It uses the OpenGL shading language, GLSL ES, and can be cleanly combined with other web content that is layered on top or underneath the 3D content. It is ideally suited for dynamic 3D web applications in the JavaScript programming language, and will be fully integrated in leading web browsers." [2]
With WebGL, you get hardware-accelerated 3D graphics inside the browser. You can create 3D games or other advanced 3D graphics applications, and at the same time have all the benefits that a web application has. In addition to these benefits, WebGL also has the following attractive characteristics:

- WebGL is an open standard that everyone can implement or use without paying royalties to anyone.

- WebGL takes advantage of the graphics hardware to accelerate the rendering, which means it is really fast.

- WebGL runs natively in the browsers that support it; no plug-in is needed.

- Since WebGL is based on OpenGL ES 2.0, it is quite easy to learn for many developers who have previous experience with this API, or even for developers who have used desktop OpenGLwith shaders.

The WebGL standard also offers a great way for students and others to learn and experiment with 3D graphics. There is no need to download and set up a toolchain like you have to do for most other 3D APIs. To create

your WebGL application, you only need a text editor to write your code, and to view your creation you can just load your files into a web browser with WebGL support.

4.4 Three.js

Three.js is a cross-browser JavaScript library/API used to create and display animated 3D computer graphics on a Web browser. Three.js scripts may be used with HTML5 canvas element, SVG or WebGL. The source code is hosted in a repository on GitHub. Three.js allows the creation of GPU-accelerated 3D animations using the JavaScript language as part of a website without relying on proprietary browser plugins.

Figure 4.4: Three.js logo

Three.js is one of the most used libraries for 3D programming. Three basic objects used in this library are:

- Camera: this object provides the user's point of view;
- Mesh: point matrix used to draw pixel on the screen;
- Scene: space in the virtual 3D world where you create geometries.

Three.js provides different kind of cameras:

- OrtographicCamera: starting with an orthogonal projection this camera makes an assonometric vision;
- PerspectiveCamera: this camera uses a perspective projection.
- CubeCamera: this camera is used for rendering cube maps. It renders scene into an axis-aligned cube.

Three.js provides, also, to mangage point cloud data using its Pointcloud object: a geometry where each point spatial coordinate and rgba information can be pushed into the geometry to fill it. The library indexes points using a Buffer Geometry to efficiently retrieve them. Three.js provides also Rays: you can catch a point on the screen (usually where the user clicked a mouse button) by throwing a ray from the camera view to a specific direction. Using the intersectObjects method you obtain all the objects crossed by the ray.

4.4.1 BufferGeometry

When you are working with a lot of data the performance is very important. For this reason you have to consider the BufferGeometry object. This is a very efficient class for geometry manage because it saves all data in buffers. It reduces memory costs and cpu cycles.

In WebGL element indices can be only 16-bit, so in one draw call you can address just up to 65,536 unique vertices. You can have as many triangles as you want, but they all must use just these vertices.

In BufferGeometry this indexing problem for large number of vertices is handled in a different way: there is just one huge index buffer and several huge per-attribute buffers shared by all chunks and metadata about how are chunks organized is stored in offsets array. Draw call for each chunk is sending count indices starting from start offset (from the beginning of index buffer) and these indices are interpreted as being themselves offset by index number of vertices in corresponding attribute buffers (thus you can address vertices with indices larger than 65,535). When you create buffers, you need to take care that in each chunk only up to 65,536 vertices will be addressed.

Buffer Geometry has best performance than Geometry thanks to these features, especially when data doesn't change in time.

4.5 PCL (Point Cloud Library)

Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing. PCL is released under the terms of the BSD license, and thus free for commercial and research use. We are financially supported by a consortium of commercial companies, with our own non-profit organization, Open Perception. We would also like to thank individual donors and contributors that have been helping the project [1].

Figure 4.5: PCL logo

As you can read from the official website, PCL library is a useful open source tool that allows you to efficiently manage point cloud data. This library contains different tools to read point cloud data from file with different formats, such as PCL, PLY, OBJ, as well as the PCD format: extension created to complement other point cloud file formats. PCL library provides tools to build kd-trees or octrees to simplify searches but also tools to filter

[1]PCL is available at: http://pointclouds.org/

data. The pcl_filters library contains outliers and noise removal mechanisms for 3D point cloud data filtering applications.

Chapter 5

Data Management

One of the most difficult problem to solve working with point cloud is how to cleverly manage them. Point cloud is a large dataset with millions of points collected by modern LIDAR systems. After the acquisition it is necessary to store them into spatial dbms using clever technique that allows you to retrieve them with elevate performance. In this work we won't be limited to one kind of db structure but we aim at developing a transparent solution. LIDAR real-time streaming is a very heavy operation so the analysis is the most important phase.

When we are playing, or looking at a virtual reality, every second millions of points have to be drawn on screen. This process can't be slow or the movement in the view won't be fluid, threatening a correct navigation into the space. So the aim of this work was to find an applicable solution to allow us to stream LIDAR data in real-time.

In real world when we are looking at a scene we notice that closer objects have higher resolutions than farther ones. In this way on the screen the thinking is the same: the point density is higher if points are closer and lower if points are farther. To obtain an accurate representation of the data collected by LIDAR we have to reproduce this effect by reducing the density of the points we draw on the screen: close-up to us we will see objects with higher number of points meanwhile objects in the background have a lower level of detail, so a lower number of points.

5.1 Problem

Point cloud data contain millions of points collected by LIDAR devices so they have to be organized to fast retrieve them. Nowadays, despite newer technology, organizing data is a considerable operation and it is important

to analyze data to find out the best solution to this problem/inconvenience. The most useful approach known at present day uses an efficient data structure called octree. After data are inserted into a similar structure, they are easily retrieved and the search operation is faster. Next problem now is the simplification of the point cloud to build the different levels of detail.

5.2 Point Cloud dataset

Point cloud data sets are easily retrieved on line. Point cloud are large data sets occupying many GB of memory. The data set used for the test was produced by a 'simulated' LIDAR camera that was flown over the MOUT model in a sweeping pattern. The data files contain xyz information, rgba color data, and the index of the camera frame in which the point was imaged.

The 'simulated' LIDAR data were acquired by passing rays through the camera center horizontal scanline and the intersection of the closest visible polygon was recorded. Also, the color information from that polygon's texture was recorded.

The considered dataset contains about 640,000 points and occupies ∼45MB of memory. It represents a simulated rural scene with few buildings and some vegetation. [1]

5.3 Tiling

When data you are using contain millions of points you have to subdivide them into small sets to be able to manage them into modern systems. Every subset is called *tile*.

This technique allows you to manage a little subset of data and process it with higher performance. In this case tiling allows us to have the total control of every portion of the space so we know every time which tile we have to draw on the screen.

Tiling is an operation you have to consider: depending on your goal you can make different tiling subdivision to efficiently reach your purpose. Data also have an important role in tiling: data analysis is important to have as many information as possible to take decisions to organize them into partitions. Working with point cloud may be useful to limit the max number of points per tile because data density could be too heavy for render operation. Moreover you optimize the quantity of data required allowing to

[1] Different datasets are available at: http://www.math.tamu.edu/ hielsber/USC/ChangeDetection/MOUT.htm

remove worthless data for the current view and reduce time to retrieve data. Tiling also allows to spatial organize points collecting information about their position to make easier later operations on them. Knowing where a point is located can be essential for some kinds of operations, such as measurements, saving time to obtain output data.

5.3.1 Specific case study analysis

In this way, a possible solution is the usage of a No-SQL database. No-SQL databases are built to have great performance on data which do not change in time. However, each problem has specific requirements so the evaluation of different kinds of No-SQL database is a very delicate phase. Important aspects to consider are then:

- *License Type*
- *Concurrency Control*
- *Transactions*
- *Data storage*
- *Characteristics*: Consistency, High Availability, Partition Tolerance, Persistence.

Transactions and concurrency control are useful to maintain consistent data, helping to avoid inconvenients that permanently damage data. This work pays particular attention to few important features: license and data storage. Data do not change in time, so the thing to consider was the memory requirement limit because we were working on cloud and we didn't have large amount of available resources with reasonable costs. Our maximum memory size available was 2GB. Taking advantage to queries performance and database structure, document No-SQL database take suits for us, so we tested MongoDB to know the effectively usability of this tool.

5.3.2 MongoDB

In that way we looked for information about on the web and, after comparing No-SQL db, we chose MongoDB for our try phase. MongoDB is a BSD No-SQL db that uses a document-oriented storage, has a full index support and high performance when you have to frequently read data. We configured our system with a PostgreSQL db where we load a db with about 640000 points. After that we downloaded and installed MongoDB [2] on our client.

[2]MongoDB is available at:http://www.mongodb.org/

The next phase was the interrogation of the PostgreSQL db to retrieve data. Data were organized in tiles. After retrieved points contained in several tiles we tried to load them into MongoDB database. This operation required few minutes. Later we tried to retrieve data from MongoDB db: performances were double than the PostgreSQL ones. On the average time to retrieve data was double in the RDBMS. If you want to retrieve about 300000 points you have to wait ten seconds with MongoDB, but more than fifteen with PostgreSQL queries. After that we noticed that the amount of memory used was too big: our db occupied about half a GB of RAM. DB we will use will be much bigger than the one used in our test example so the memory requirement will probably be out of our possibility.

Figure 5.1: MongoDB logo

This wasn't an adoptable solution to our problem so we decided to try another method: extract tiles from PostrgreSQL db and process them to create different files, one for every level of detail. Spatial databases provide spatial queries to efficiently retrieve data: these instructions differ by non-spatial queries because they allow the use of geometry data types such as points, lines and polygons and consider the spatial relationship between these geometries.

5.4 Point cloud simplification

There are different algorithms to simplify point cloud data. Below we analyze three different approaches to find the best one/most suitable to solve our problem:

- *Order and remove*: This algorithm consists in ordering points and removing the t'th element. Taking advantage of points organization, you can use a simple removal operation of the t'th element. However, this approach doesn't guarantees to be without loss of relevance;

- *Build octree with maximum number for branch*: This approach builds an octree with number of points under a branch less or equal than a preset value. This approach is useful when data have to be query in the future;

- *Clustering*: This solution puts data into a large virtual cube divided then into sub-cubes to better identify points location. After that you

proceed to cluster data and analyze them to reduce the number. The important thing is that the reduction has to consider the arrangement of the points to maintain the relevance: points can't be random deleted but you have to analyze their position to remove points without loss of significance.

Chapter 6

Implementation

Concerning this, we decided to use PCL library to decimate tiles with the intent of create different levels of detail of the given point data. It was necessary to implement an interface to read data from the database and save them into files.

6.1 System

The system implemented is made of three different parts: PostgreSQL db, a service to retrieve data at a specific level of detail and point cloud visualizer. Db is used to store point cloud data. In the pre-processing phase the interface written in C++ queries the db to build LOD files. When the web page is charged, it calls a web service that retrieves data regarding the selected level of detail.

6.2 LOD creation

After a first test operation we were successful in reading data from a point cloud file (containing the same data as write above) and decimate them obtaining three levels of detail.
PCL library uses an algorithm that subdivides data in a voxel grid. A voxel represents a value on a regular grid in three-dimensional space. In each voxel, all the points present will be approximated with their centroid. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately. The algorithm has custom input parameter that allows you to set the octree leaf size: in that way you can calibrate the decimation operation to obtain the level of detail you want.

This method was a good start but file read and write operations were inefficient and worthless. So we decided to create an interface to connect to PostgreSQL DB to retrieve data to directly feed PCL library, bypassing useless read operation from file. The interface was built in C++ language. To connect to the database we had to use specific PostgreSQL library named libpqxx[1]. To track queries we decided to save them into files: we created a simple file containing all the queries we will use to ask the db. This algorithm considers an easy pre-process phase where file containing queries is created. After that, by launching this sequence of instructions, queries are read from the file and db is interrogated to retrieve the data of the tile. So, by passing data to the voxel grid function, a voxel grid is built and, considering the leaf size, data is downsampled and filter making LODs files. Then we obtained one different file for each level of detail for every tile we made.

The algorithm pseudo code follows:

```
Result: Create LOD files from PostgreSQL db
connect to the db;
while there is a new tile do
    read current query;
    query PostgreSQL db to retrieve data;
    for i ← 0 to nLODs do
        create voxel grid;
        set raw data as input cloud;
        set octree leaf size;
        get cloud_filtered data;
        write cloud_filtered data to the corresponding LOD file
    end
end
```

Algorithm 1: Create LOD files from PostgreSQL db

Leaf size is an input parameter you can set before running this algorithm. Leaf size sets the dimension of the leaves: value used to subdivide the point cloud. After setting the leaf size, the algorithm calculates the number of subdivision to make and uses it to compute centroids.

[1] Libpqxx is available at: http://pqxx.org/development/libpqxx/wiki/DownloadPage

6.2.1 Algorithm Overview

Looking closely we see the PCL library algorithm works as follows:

```
Result: Filtering point cloud
calculate point cloud bounding box;
validate leaf size parameter;
for cp in point cloud do
    calculate leaf index idx;
    index_vector ← (idx, cp);
end
sort index_vector by idx;
count distinct indexes from index_vector;
prepare output vector;
for cp in point cloud do
    save point into centroids vector;
    i = cp + 1;
    while i<index_ vector(size) and index_ vector[i] equal to
    index_ vector[cp] do
        save point into temporary vector;
    end
    centroid += temporary;
end
centroid /= (i - cp);
for idx > cp do
    copy centroid into output vector;
end
```

Algorithm 2: Filter point cloud

Basically, the thinking behind this algorithm is quite simple: every point belonging to a leaf will be substituted with the centroid of that leaf calculated with formula below:

$$\frac{\sum pts\ into\ the\ leaf\ coordinates}{number\ of\ pts\ into\ the\ leaf} \tag{6.1}$$

In this way the greater the leaf size parameter is, the greater will be the number of points contained into the leaf and, consequently, greater will be the reduction of this leaf. By wisely choosing the leaf size you can obtain different levels of detail: the greater the leaf size is, the smaller the level of detail will be.

Analyzing the pseudo code, in particular, we see that in the first step the leaf index (*idx*) is calculated for every point in the point cloud and saved into

index_vector. After that, index_vector is ordered to count distinct indexes. Now count is used to prepare output vector. Later centroids are calculated. For each point in the leaf:

- save it into centroids vector;
- save other points of the same leaf into temporary vector;

The next operation adds the temporary vector to the centroid vector. After that the average is calculated normalizing that value by dividing for the number of points in that leaf. The last procedure copies centroids to the output data to give back.

There aren't fixed values for this parameter so you have to try different sizes to tune it to obtain the best results. Every point cloud is different from each other so tuning leaf size is part of your application setup.

The first implementation of this algorithm worked only with point cloud data with x, y, z coordinates. LIDAR data may contain color information. So, to preserve point's information, it was necessary to build a customized class containing x, y, z coordinates and rgba values. After this little change, data were correctly written into files without loss of useful information.

6.2.2 Space Optimization

After the first aim was achieved, we moved our attention to the space optimization. Up to now, we had saved files in a human-readable text format. To reduce the amount of memory occupied by files we decided to save them as binary data. The interface created before was then modified to save data into a specific format. This operation allowed us to gain about 50% of the space occupied going from about ∼100MB to ∼50MB. This change obviously involved also service modifications. These were then modified to read the new data file type and to unpack binary data to retrieve point information.

6.3 C++ interface code

This interface has leaf sizes as input parameters. Leaf size is a float value representing leaf dimension in meters. The C++ interface code written to create levels of detail files follows:

```
...
/* Read data from db */

for (int lod = 0; lod < n_lods; lod++) {
  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
```

```
  sor.setInputCloud(cloud);
  sor.setLeafSize(lods[lod], lods[lod], lods[lod]);
  sor.filter(*cloud_filtered_blob);

  fromPCLPointCloud2(*cloud_filtered_blob, *cloud_filtered);

  /* Write data into files */
}
...
```

Listing 6.1: LODs creation interface

6.4 Visualization

After obtaining LODs files, we built a simple scene where adding point cloud data. Using Three.js, point cloud data was drawn on the screen. To cleverly manage it, original point cloud data were divided to create a THREE.Pointcloud object for every tile set. In this way it was easier to work with points. A simple interface allows user to choose the level of detail and keep points drawn on the screen. Another button enables edit mode to allow measurement operation. Simple camera movements (like zoom and pan) were implemented.

Chapter 7

Tests and Results

Below is an explanation of the dataset used in this work and the results obtained.

7.1 Padua LIDAR data

The next step was the usage of Padua LIDAR. Entire database occupies about 7GB and contains 21 million points (estimated). To import dataset was necessary to implement another service able to read data from original .las files and insert it into the database. To easy read data each .las file was converted into human-readable text file using libLAS: libLAS is a C/C++ library for reading and writing LAS LIDAR format files. libLAS provides also *las2txt* function to convert .las data to txt without loss of information. After that, using libpq, a script, written in C++ language, was developed to directly connect to the DB to repeatedly insert data read from the current file. The first implementation was very inefficient because of PostgreSQL parameter. Modifying shared_buffer parameter in postgresql.conf file and setting it to 25% of pc's RAM size performance augmented reducing time to insert data.

Shared_buffer sets the amount of memory the database server uses for shared memory buffers. The default is typically 128 megabytes (128MB), but might be less if your kernel settings do not support it (as determined during initdb). This setting must be at least 128 kilobytes. However, settings significantly higher than the minimum are usually needed for good performance. This parameter can only be set at server start.

Another improvement directly derives from PostgreSQL db: instead of using recurring insert operation there is a specific command COPY [3] used to import data from csv files. Taking advantage of this, data were rapidly

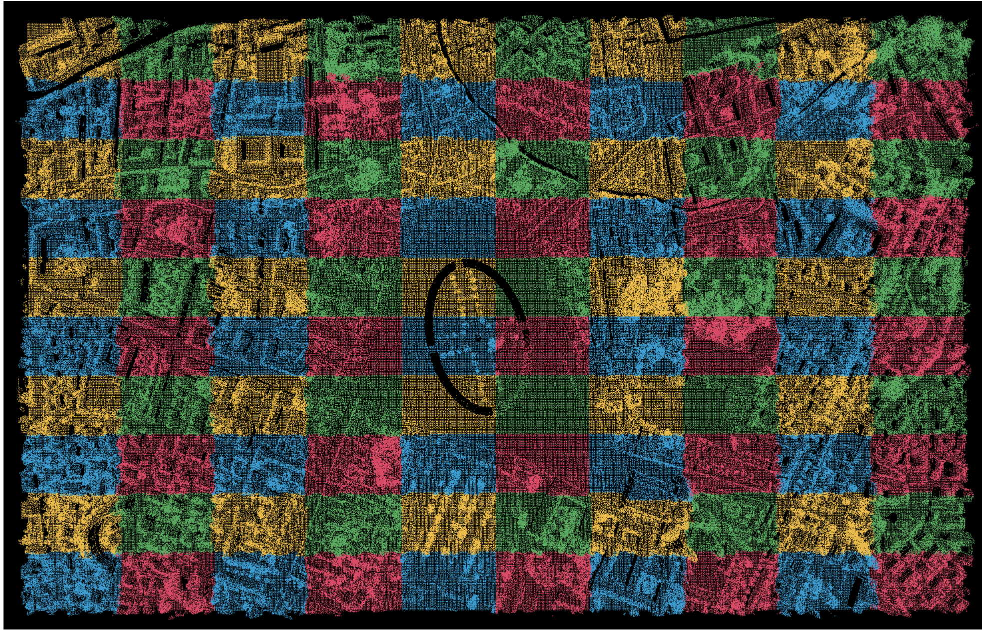

Figure 7.1: Tiles

inserted into the database. COPY rapidity is given by the fact that PostgreSQL commit is executed after all data are populated on a database. This improvement allows to significantly reduce time to insert data.

7.2 LOD decimation results

Working with the entire dataset is too heavy because of the big number of data to manage. We decided, then, to process a small part of the database. The chosen subset contained about two millions of points and it was easily reduced.

The first operation was the subdivision in hundred tiles: as you can see in Figure 7.1 the entire data is divided into blocks and everyone will be reduced to create levels of detail. Purely for demonstration purposes, every tile is colored to emphasize the subdivision.

Figure 7.2 shows the maximum level of detail created by the framework. This step reduces the points number of 33% keeping the relevance. Going on in this way we created the other levels of detail and then organized them to the same scene to compare them.

Figure 7.3 accentuate levels of detail: the number of points drawn is

Figure 7.2: Subset with the higest LOD

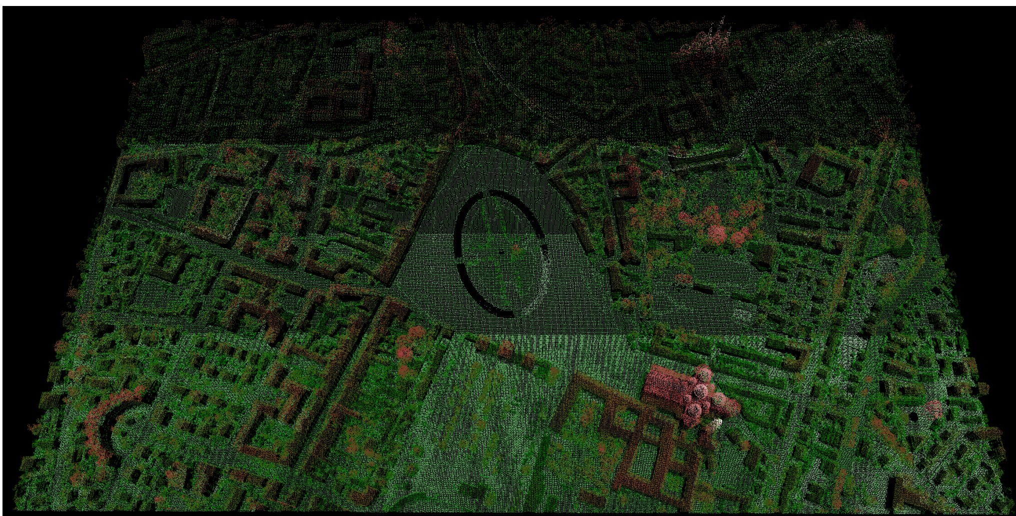

Figure 7.3: Level of details

gradually reduced going far from the point of view: nearest points are more detailed instead of farther ones. In this picture you can see four levels of detail: leaf sizes are set to 2.0mt, 1.5mt, 1.0mt and 0.5mt going from the closest to the farther level. Table 7.2 shows the percentage of reduction. In this case four files for each tile were created making a total of four hundred files subdivided into four folders. The total space occupied on disk is 85,5MB.

	Leaf_size (mt)	Number of Points	Reduction (%)
Database	-	2,072,935	0
LOD3	2,0	1,388,381	33,02
LOD2	1,5	839,715	59,49
LOD1	1,0	554,405	73,26
LOD0	0,5	165,867	92,00

Table 7.1: Percentage of point cloud reduction

In this work, point reduction is intentionally accentuated to underline levels of detail creation. The first level indeed, reduces point cloud of ∼33% going from two millions to about 1,4 millions of points. Going ahead in this way, we reached the very high reduction (∼92%) of the last level of detail obtaining about 165K points. All this passages achieved the goal of conserving points relevance.

7.3 Measurement on LIDAR

One of the most useful things you can do with LIDAR data, after efficiently drawing them on the screen, is calculating the linear distance from one point to another. This is a difficult operation to do because of the complexity of the point cloud. One of the most important problems to solve is catching the exact point the user clicked over. The point density gradually decreases looking to the horizon of the view, so points are far from each other then catching them is more difficult. To amend to this inconvenience, it would be interesting make sure that point clicked is effectively a point drawn on the screen. To do this you have to automatically clasp points while mouse is moving. Another important thing to consider is that, reducing the level of detail, point drawn on the screen can be different compared with points in the database because of the approximation deriving by the creation of levels of detail. In this case, so, you have to compare points with those in the database to obtain the best measure possible.

7.3.1 Implementation

A first implementation of linear measurement between two point uses rays to locate the effective point the user has clicked on querying PostgreSQL database to retrieve the real point which is as near as possible to the virtual point drawn on the screen.

Pseudo code follows:

```
Result: Take point to point measurement
throw a ray in the direction of point clicked;
if point caught then
 | use this point as caught point;
else
 | project into the scene the coordinates of the point clicked;
 | set a buffer to locate the point near the ray;
 | query the DB to find the set of points near to that point and
 | search, if exists, a point nearest to the ray;
 | if exists then
 |  | take it as point clicked;
 | else
 |  | use the point found before;
 | end
end
```

Algorithm 3: Take point to point measurement

A first implementation of the measurement operation used ray picking: every time user click on the screen, a ray is thrown to the scene. Using intersectObjects method, belonging to RayCast object, you can find every object ray passed through. This procedure, however, doesn't pick only point cloud points but allows to locate also points inside its bounding box. This solution has also an intrinsic problem: point caught throwing the ray couldn't be in the original database because of the density reduction. You may pick a point which is a voxel centroid and, for this reason, isn't in the database as it was calculated in the previous reduction phase.

To satisfy precision commitment it wants to make sure that ray actually hit a point of the data drawn on the screen and then find the nearest real point into the database to make the measurement. This idea needs to consider the maximum level of detail to make measurement much as correct as possible. In that way so, a specific service was built to retrieve the nearest real point into the database. User click on the screen to keep start and end point of the object he wants to measure and then the service is called and the measurement is built on the effective distance of real database's points. To

faster retrieve data from the database, we decided to limit the query searching only points near to the point clicked by the user, in a range of $\pm 0.01\%$. Doing so the number of points retrieved by the query is minimal and the speed is higher. In that way user has a quickly reply to his query and reach measurement faster.

7.3.2 Points sticking

An important problem to solve now is to keep the pointer sticking on real points drawn on the screen. The relevant thing to do is to make sure that measure takes place by two points on the screen shifting the cursor to attach it to a point really drawn. Taking advantage of the subdivision made earlier, it was possible to limit the search operation only to points of the specific tile, reducing then the search domain. Catching a point, the object returned contains also the reference to the related point cloud. So, making a simple linear search operation, it was able to retrieve the point closest to the one clicked. Moving mouse into the scene, you can see that the highlighted point is always a point of the point cloud. This expedient allows to remove all non existent points deriving by projection of the point clicked on the screen to virtual space.

7.4 Measurement results

In the next screenshot (Figure 7.4) you can see the measurement operation compared to the Google Maps system. This example shows a measure made on the Prato della Valle central space. On the left you can see the measurement applied to LIDAR data with the maximum level of detail (yellow line) while on the right the measurement was made using Google Maps (blue dotted line). As you can see, the results are very similar: our system returns a measure that little differs ($\sim$0,4%) from the one obtained by Google Maps. Therefore, the measurement has to be considered reliable

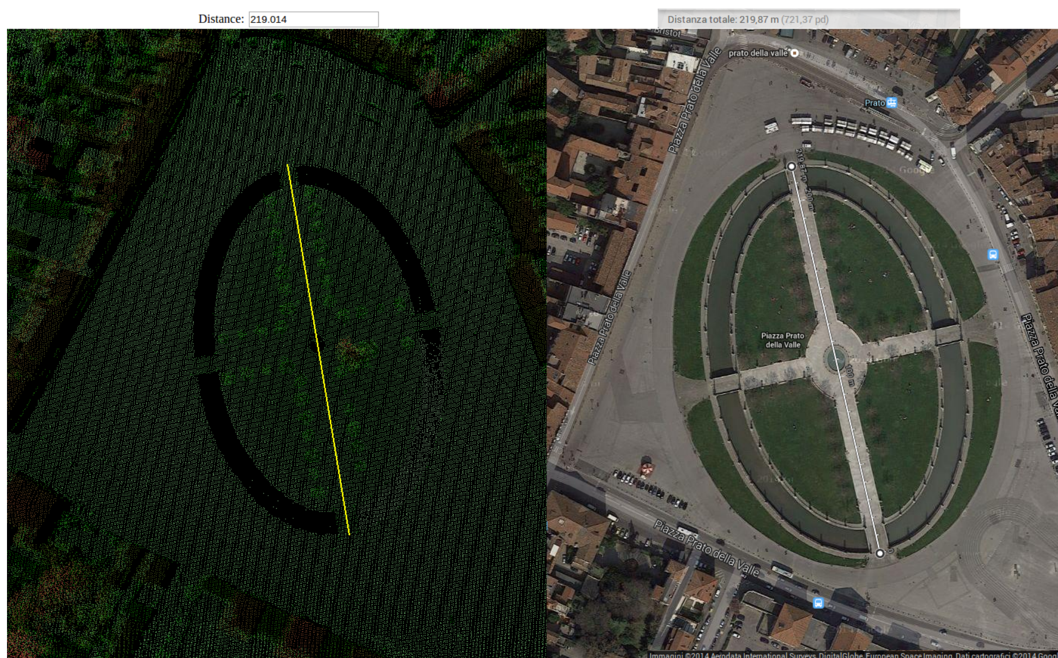

Figure 7.4: Measurement comparison

Chapter 8

Conclusions

This work focuses on the analysis of the LIDAR data distribution on the web. A large amount of data is difficult to manage so it is essential to cleverly use. This first implementation reached the expected goal of different levels of detail creation using an open source library useful to work with point cloud data.

8.1 Future Improvements

Future improvements include algorithm enhancement: the centroids calculation can be improved using a weighted method giving importance to the points depending on the distance from the voxel centroid. In this way the approximation will be more precise and data-related.

By carefully observing the results we saw that decimation removed the Point cloud noise only partially. Therefore, in the future, it could be necessary to apply noise filtering (maybe one of those supplied by the PCL library) to remove worthless data improving the quality of the view.

Another good thing is the application of texture to the scene objects to produce a more realistic representation of the Point cloud: after identifing objects in the scene it would be great to apply texture to buildings, roads, to the vegetation and other objects in the scene to improve the user's experience. In this way new information can be added to the scene helping the user to move into the virtual world. In the specific case study a useful development could be roofs identification.

Measurement is very important in LIDAR data works, therefore, measurement operations must be as much precise as possible. Our results prove that this implementation has a good accuracy comparing with Google Maps. However, during measurement operation, you can bump into a little annoy-

ing problem: due to the perspective view two points may overlap bringing the user to click on the wrong point on the screen. To avoid this inconvenient, you could use section plane passing through points clicked and improve measurement switching to the ortoghonal view. In this way points can not overlap avoiding unwanted point clicking. Measurement could be also applied to roofs, squares and other objects identified to retrieve the area of the geometry.

List of Figures

3.1 Example of LIDAR scanning operation 10
http://lidar-america.com/?p=1
3.2 Point cloud visualization example 12

4.1 PostgreSQL logo 17
http://www.postgresql.org/
4.2 Postgis logo 18
http://postgis.net/
4.3 WebGL logo 19
https://www.khronos.org/webgl/wiki/Main_Page
4.4 Three.js logo 21
http://threejs.org/
4.5 PCL logo 22
http://pointclouds.org/

5.1 MongoDB logo 28
http://www.mongodb.org/

7.1 Tiles 38
7.2 Subset with the higest LOD 39
7.3 Level of details 39
7.4 Measurement comparison 43
Map of Italy, retrieved on Nov 26, 2014 from https://www.google.it/maps

List of Tables

3.1 No-SQL compare table 15

7.1 Percentage of point cloud reduction 40

Listings

6.1 LODs creation interface 34

Bibliography

[1] D. S. G. S. J. K. B. Licea-Kane, *OpenGL® Programming Guide Eighth Edition*. Addison-Wesley, 2013.

[2] T. Parisi, *WebGL: Up and Running*. O'Reilly, 2012.

[3] T. P. G. D. Group, "Postgresql copy command reference."