BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

WEB TOOL FOR VIEWING AND MANAGING GEOGRAPHIC AND POINT CLOUD DATA

WEBOVÝ NÁSTROJ PRO PROHLÍŽENÍ A SPRÁVU GEOGRAFICKÝCH A POINT CLOUD DAT

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. FILIP ČIŽMÁR**
AUTOR PRÁCE

SUPERVISOR **Ing. PETR MUSIL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

164642

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Čižmár Filip, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Information Systems and Databases
Title:	**Web tool for viewing and managing geographic and point cloud data**
Category:	Web applications
Academic year:	2024/25

Assignment:

1. Familiarize yourself with the principles of interactive web application development, as well as with libraries for working with geospatial data and rendering image data on the web.
2. Compare existing tools for working with and annotating point cloud data.
3. Design a web-based tool for browsing and annotating point cloud data acquired through mobile mapping.
4. Implement both the frontend and backend of an interactive web application using Vue.js and FastAPI. The tool should support visualization of additional modalities, such as map layers and imagery. Focus on mechanisms that enable efficient labeling of various types of objects within the point cloud.
5. Design and implement a tool to assist with object annotation, running asynchronously in a containerized environment.
6. Design and implement mechanisms for the efficient processing of point cloud data containing a large number of points.
7. Develop and test the tool iteratively with potential users. Focus on the efficiency of their annotation workflows, particularly when labeling traffic signs and vegetation.

Literature:

- potree.org - WebGL point cloud viewer for large datasets.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Musil Petr, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	31.7.2025
Approval date:	29.7.2025

Abstract

The goal of this thesis is to develop a web-based application capable of visualizing point cloud data in their geospatial context, annotating and managing them, and integrating other geospatial modalities for visualization. The thesis presents the necessary definitions to understand and work with point cloud data. Point cloud capture, storage and processing information is presented. Existing tools of similar nature to the application in this thesis are explored. Among those, possible options of implementation are pointed out. The design and implementation details of the application developed as a part of this thesis are explored. Possible design alternatives are mentioned. Finally, the testing process is presented and conclusions are drawn.

Abstrakt

Cieľom tejto diplomovej práce je vyvinúť webovú aplikáciu schopnú vizualizovať point cloud dáta v ich geopriestorovom kontexte, anotovať ich, spravovať a vizualizovať aj ďalšie geopriestorové modality. Práca uvádza definície potrebné na pochopenie a prácu s point cloud dátami. Sú prezentované informácie o zachytávaní, ukladaní a spracovaní takýchto dát. Ďalej sú preskúmané existujúce nástroje podobného charakteru ako aplikácia v tejto práci. Spomedzi nich je poukázané na možné spôsoby implementácie. Práca ďalej opisuje návrh a implementačné detaily aplikácie vytvorenej v rámci tejto práce. Spomenuté sú aj možné alternatívy návrhu. Na záver je predstavený proces testovania a sú vyvodené závery.

Keywords

point cloud, web, application, geographic, geospatial, classification

Kľúčové slová

point cloud, web, aplikácia, geografické dáta, geopriestorové dáta, klasifikácia

Reference

ČIŽMÁR, Filip. *Web tool for viewing and managing geographic and point cloud data.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Petr Musil, Ph.D.

Web tool for viewing and managing geographic and point cloud data

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Musil Petr, Ing., Ph.D.. Language models (LLMs) were used occasionally to check grammar and improve sentence clarity. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Čižmár
July 30, 2025

Contents

List of Figures

Chapter 1

Introduction

Today, in the modern world, the capture and management of geospatial data is becoming increasingly widespread. Point clouds, in particular, are becoming more relevant across a wide range of application areas. Unlike traditional methods that rely on two-dimensional images and photos, point clouds offer a much more accurate and in-depth three-dimensional model of the real world. This shift has transformed industries such as land surveying, geomatics, architecture, engineering, construction, and urban planning, where precise spatial representations are critical. Point clouds enable detailed 3D modeling, precise measurements, and advanced spatial analysis, making them indispensable for tasks ranging from as-built modeling and reverse engineering to virtual reality experiences and infrastructure planning.

The availability, accuracy, density, and size of 3D point clouds have grown exponentially in recent years, driven by advancements in scanning technologies such as LiDAR, photogrammetry, and 3D laser scanning. These developments have pushed the boundaries of what is possible with point cloud data, enabling professionals to capture and process vast amounts of 3D information with unprecedented precision. However, the tools required to work with this data have not kept pace in terms of accessibility and universality.

While there are numerous desktop applications available for processing and visualizing point cloud data—such as CloudCompare [36], Faro SCENE [32], Autodesk ReCap [23], and Pix4D [52] – these tools often require installation on specific systems and may lack the convenience of being accessible from any device. This limitation highlights a significant gap in the market: the scarcity of web-based applications capable of handling the full spectrum of point cloud processing tasks. Although some web tools, such as Potree [58], exist for visualizing point cloud data, they primarily focus on rendering and do not offer additional required features for classification or data management. Additionally, general-purpose GIS platforms like GIS Cloud [37] and Felt [33], while robust for handling vector and raster geospatial data, do not natively support the specialized requirements of point cloud data, such as loading and displaying .laz files or integrating point clouds with map backgrounds for accurate real-world positioning.

This gap underscores the need for a dedicated web application that can fulfill all the required tasks outlined in this thesis: a frontend for displaying and editing point cloud data along with other modalities, a backend for saving and managing the data, the ability to load point clouds from the backend or local files, and a map background to display point clouds in their correct geospatial context. Such an application would not only address the current limitations of existing tools but also provide a convenient solution for professionals and researchers working with point cloud data. It is also proposed that the application be

also able to manage and display other modalities, such as possible points of interest and 360-degree imagery taken at the time of the point cloud acquisition.

The aim of this thesis is to design, describe, analyze, implement, build, and test such an application, contributing to the growing demand for accessible and versatile tools in the field of point cloud processing.

Chapter 2

Point Clouds

A large portion of this thesis's work concerns point cloud data. As such, it is necessary to spend some time explaining what point cloud data are, how they're collected and what can be done with them. This chapter delves specifically into that.

2.1 Definition of a point cloud

Intuitively speaking, or as the name suggests, a point cloud is a collection of points. In short, it's a discrete unordered set of data points in three-dimensional space, the collection of which represents a certain (usually physical, real-life) object or scene, or a part of it. Each point is defined by a set of Cartesian coordinates (X, Y, and Z), along with other possible features, such as the point's color (in RGB format), its itensity, classification, and others... That said, the only necessary feature for a set of points to be considered a point cloud is the coordinate data.

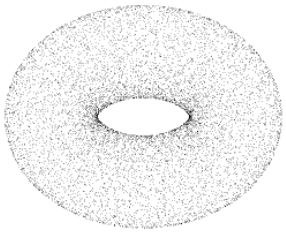


Figure 2.1: An image of a point cloud representation of a torus. Source: [71].

More formally, we can define a point cloud P as a set of n points, where each point p_i is represented by its 3D coordinates [72]:

$$P = \{p_i(x_i, y_i, y_i) : i = 1 \dots n\}$$

2.2 Sources of point cloud data

Even though one could theoretically randomize the positions for a set of points of an arbitrary cardinality and call this collection a point cloud, a point cloud like this, aside from being an intruiging choice, would have no practical uses (aside from maybe performance testing and noise creation for Generative Adversarial Networks [46]).

Point clouds usually serve as a rich source of data for creating accurate 3D models, analyzing shapes and structures, and conducting measurements or assessments of real-world environments. As such, they tend to represent a physical object. Point clouds like this are used in various settings, including architecture, engineering, surveying, environmental monitoring, and computer graphics. However, it's worth noting that point clouds can also be generated in virtual environments or computer simulations to represent digital objects or scenes. In these cases, the point cloud does not correspond to a physical entity, but rather to a virtual or computer-generated object.

Generally, point clouds that represent physical objects are generated using 3D scanners (such as LiDAR) or photogrammetry software [28].

3D scanning

During the process of sensing, the scanner collects with the help of different technologies information about the shape and dimensions of the scanned object and depending on the technology can also record, for example, information about the color of the object [64].

There's many different existing methods of how to process and map real life objects into 3d data. These techniques work with a range of sensor types, including optical, acoustic, laser scanning, radar, thermal, and seismic. Enumerating all of the techniques is not the purpose of this thesis and as such, only a few of them will be mentioned, to gain an overview of how a process like this works [66, 26, 68, 29]:

- Time of Flight – ToF scanners measure the time it takes for a laser beam to travel to the object and back (e.g. the round-trip time). Knowing the speed of travel (the speed of light) and the round-trip time, distance, and subsequently the position of a point is calculated.
- Triangulation – instead of measuring the round-trip time, a support camera is used to detect the laser light and the position of the point is calculated using the angle held by a triangle between the emitter, the laser dot and the camera, as depicted on Figure 2.2
- Structured light – A known pattern (i.e. stripes or grids) are projected onto the surface of an object using an LCD projector or other stable light source. A camera then captures the deformations of the pattern and calculates the distance of each point in its field of view inside the pattern.
- Coordinate Measuring Machines (CMM) – are a type of contact 3D scanners. Contact scanners measure the geometry of an object by physically probing it, subsequentally generating 3D coordinates of points.

3D scanning of scenes and multiple objects is time-consuming, and thus bottlenecked by manual labor. Nevertheless, since the associated processing techniques are under constant improvements, the latter drawback has been substantially alleviated in the last few

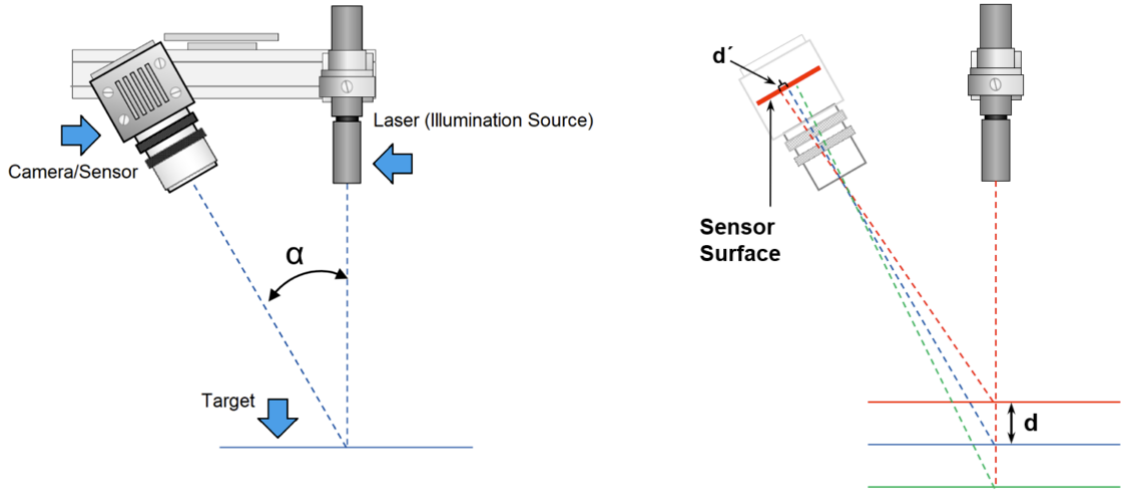


Figure 2.2: Triangulation 3D scanning technique. d refers to the distance between two different points, the angle difference of which is picked up on by the camera due to the resulting displacement d' presented on the sensor surface. Source: [71].

years. It should be noted that usually, when the object of interest has a distinctive surface, photogrammetric techniques are favored [27].

Photogrammetry

While photogrammetry could theoretically be considered a type of 3d scanning, the process used for the generation of a 3d model is quite different.

Photogrammetry, as its name implies, is a three-dimensional coordinate measuring technique that uses photographs as the fundamental medium for metrology or measurement. The fundamental principle used in photogrammetry is triangulation. By taking photographs from at least two different locations, so-called "lines of sight" can be developed from each camera to points on the object. These lines of sight, sometimes called rays owing to their optical nature, are mathematically intersected to produce the three-dimensional coordinates of the points of interest [40].

By identifying the same points in multiple images and taking into account parameters like the camera's position and orientation for each photograph, its focal length, lens distortion, and other variables, it is possible to determine where those points were located in 3D space. When one point is identified in at least two pictures taken from different known locations, imaginary lines from the two camera positions can be drawn in the direction of that point. The intersection point of the lines is then calculated, which results in XYZ coordinates of the targeted point. And with enough points, a model of the scene can be constructed (i.e. a point cloud) [49].

An important fact to note, however, is that photogrammetric processes by themselves do not retain scale of the physical scene. While proportions of an object are retained, not having an accurate scale can hinder usability of photogrammetry in many different contexts, rendering it useless in environments that require precise assessment of the real world scale, such as georeferencing and mapping, GIS (Geographic Information Systems) integration,

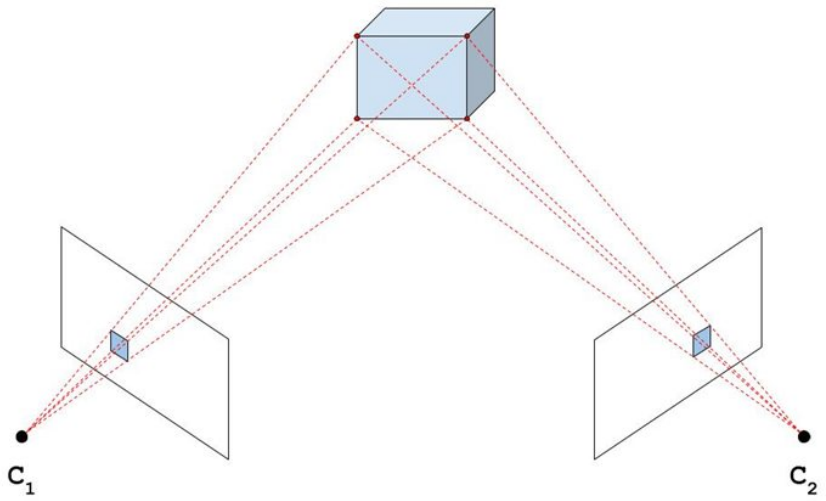


Figure 2.3: Photogrammetry triangulation. Source: [60].

engineering and construction projects, infrastructure asset management, monitoring and scientific research, and others...

To retain size relation to the physical object, markers of known size must be used, the scale of which in relation to the scale of the captured model then serves as the needed measurement to scale the model.

For example:

- Ground Control Points (GCPs) – known objects are identified in or added to the scene. A GPS device or a total station is then used to measure accurate 3D coordinates of these objects. When the coordinates of these objects are known, coordinates for other points of the captured model can be calculated.
- Scale bars – feature a physical object of known dimensions into the scene (i.e. a ruler, but usually specially made scale bar objects are used). Scale bars can then be used to establish an accurate scale during reconstruction.

Due to the nature of these two methods, GCPs are more suited for bigger scenes (i.e. aerial photogrammetry of a residency area) and scale bars for accurately scaling smaller objects (i.e. fossils).

2.3 LiDAR

LiDAR stands for Light Detection and Ranging (or sometimes Laser Imaging, Detection and Ranging). While not exclusively, it is most often used for 3D scanning, belonging to the time of flight category of 3D scanners. As such, the fundamental principle of LiDAR involves the emission of laser pulses from a sensor, typically mounted on an airborne (e.g. Airborne Laser scanning – ALS) or ground-based platform (e.g. Terrestrial Laser Scanning – TLS), towards the target area.

A LiDAR device consists of one or more light emitters (TX) and one or more reflected light detectors (RX) to cover the required field-of-view (FoV). Building components of RX chain have a direct impact on the amount of received data. The receiver chain consists of a photo diode array using avalanche photo diodes or single photon photo diodes followed by trans-impedance amplifiers. Additionally, analog-to-digital converters (ADC) can be incorporated to a LiDAR's receiving path to increase its integration level and eliminate a potential need of external components.

Depending on the sensor's configuration, LiDAR systems generate millions of data points per second, which equals to several GBits/sec of raw data. The amount of received data can be described analytically by the following formula [47]:

$$N_{RX} = \frac{t_{sample}}{t_{ACD}} \cdot N_{ADC} \cdot N_{pixel} \cdot N_{OS} \cdot N_{FR}$$

given t_{sample} – maximum pixel sampling time [s], $t_{ADC} = 1/f_{ADC}$ – ADC sampling period [s], N_{ADC} – ADC resolution [bits/sample], N_{pixel} – FoV frame size, N_{OS} – system oversampling factor, N_{FR} – system frame rate.

While such a large amount of data is not an issue for a process, where a LiDAR captures geospatial information, to be stored and used later for point cloud computing, for applications that require near instant response times (such as automotive automation), this poses a concern. Data compression methods applied during the data capturing phase are necessary for these environments, some of which are explored in [47].

For geospatial data, post-capture compression algorithms are used to store point cloud data in more efficient format, some of which are explored further in the thesis.

A LiDAR works by recording the reflected light. The detector can have different modalities [53]:

- Discrete Return – records individual returns that correspond to the peaks of the waveform curve. A discrete system may record 1-5 returns from each laser pulse. A collection of discrete return LiDAR points is known as a LiDAR point cloud. Returns are recorded when the intensity exceeds a predefined system threshold. Discrete Return LiDAR is most often used for topographic mapping, land cover classification and terrain modeling.

- Full Waveform – the sensor records the entire backscattered waveform for each emitted laser pulse. This means capturing the entire range of reflections within a single laser pulse, offering a more detailed representation of the interaction with surfaces. Results in richer, but more complex to process data. Most commonly used in aerial sensing, especially for forest canopy analysis. Being able to see the entire waveform helps with identifying various vegetation layers and evaluating the structure of the forest.

After capturing the light pulse with a LiDAR, information about the target point is stored, such as the XYZ coordinates, the intensity of the reflected light and the surface normal at each point. It's important to note that the current mass-produced LiDAR devices are not capable of capturing color of the target point. Additional methods or the use of photogrammetry must be applied to capture color of a given point.

Using a lightweight camera system, properly timestamped and boresighted, with a sufficient FOV (at least 120 degrees), a point cloud can be properly colored. The point cloud is processed normally as if no camera was present and then additionally every lidar point

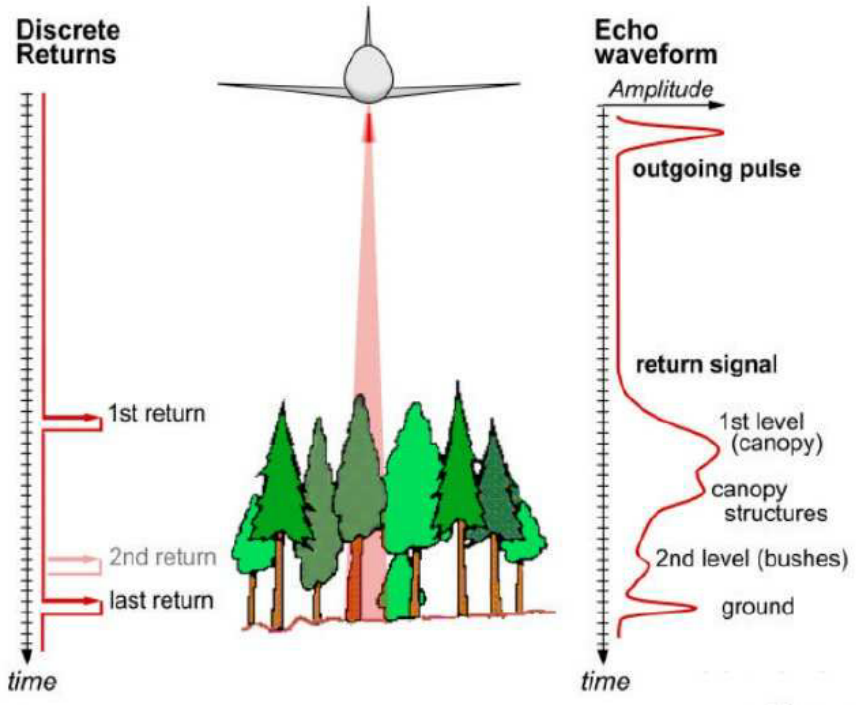


Figure 2.4: The difference between between discrete (left) and full waveform (right) LiDAR systems. Source: [35].

is tagged with an appropriate RGB value from the imagery simultaneously captured. In doing this, there is very little displacement from temporal changes and relief displacement resulting in each point of a point cloud being colored at little expense. Vendors of LiDAR devices are now working on the proper capability to capture color during the scanning process. This reportedly will be available in 2024 [31].

The captured intensity attribute of a point represents the return strength, usually coded into an 8bit integer, meaning the available range is 0-255 (but a 16bit integer or a 0-1 floating point value can also be used). While the intensity value by itself doesn't have much to do with color, coloring each point in a black and white scale of 0-255 gives a suprisingly good looking black and white representation of a point cloud.

Mobile Laser Scanning

Mobile Mapping Systems (MMSs) describe a mobile platform that can be either aerial or terrestrial, in which measurement systems and sensors are integrated for the acquisition of geo-referenced metric data [61]. Essentialy, it is the process of collecting geospatial data from a mobile vehicle.

An MMS is comprised of three main components: optical sensors, navigation/positioning sensors and a control unit [56]. If a LiDAR is added as a component for scanning, this system can be referred to as Mobile Laser Scanning. An MLS combines a moving sensor with position estimation to obtain continuous registration and unlimited viewing angles.

The latest literature divides the various types of mobile terrestrial platforms into four categories: sledge-based, boat-based, wheel-based, and human-based. The phrase "human-

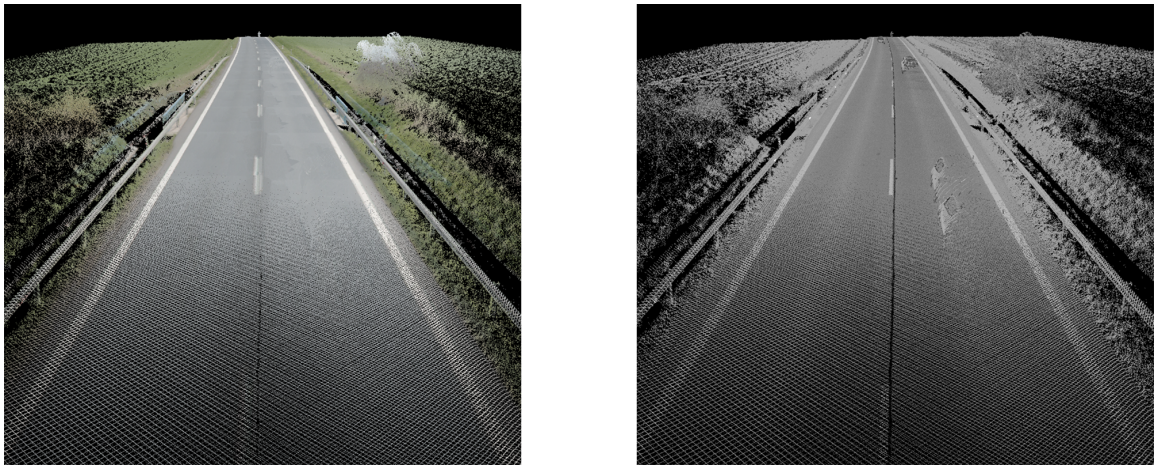


Figure 2.5: A point cloud colored using RGB (on the left) and point intensity (on the right). Image taken from the application developed as a part of this thesis.

based" refers to platforms that are carried by people; these are typically referred to as wearable laser scanners (WLS) or personal/portable laser scanners (PLS). There aren't many distinctions between PLS and WLS; the former typically refers to manually carried systems (like handheld laser scanners), while the latter usually refers to compact devices that the operator can wear, such as backpack scanners. The term "wheel-based" platforms can also be used to describe motorbikes, bikes, trolleys, and vehicles on rails. The latter consist of an unmanned ground vehicle (UGV) and a road vehicle. When discussing mobile airborne devices, the term Airbone Laser Scanner (ALS) is typically used in literature. As far as aerial platforms go, an unmanned platform would be most often referred to as a Unmanned Aerial Vehicle (UAV). Terrestrial Laser Scanning (TLS) used to refer to scanners on a fixed ground-based platform, but these days it includes not only stationary laser units, but also mobile ground-based laser units [61].

A typical MLS system uses a Global Navigation Satellite System (GNSS) receiver and an Inertial Measurement Unit (IMU) to both localize and map itself, meaning that data received from an MLS would not only contain point clouds, but also other information, such as the GPS time of when it was captured, the projected GPS position of the mobile platform, its heading and possibly rotation and others...

Geospatial data

Geospatial data is data about objects, events, or phenomena that have a location on the surface of the earth, including location information (usually coordinates on the earth), attribute information (the characteristics of the object, event, or phenomena concerned), and often also temporal information (the time or life span at which the location and attributes exist) [62].

In essence, data received from an MLS is also geospatial data. Mapping physical objects into digital space while retaining their scale and location is fundamental for creating accurate and reliable representations of the real-world environment for various applications such as urban planning, infrastructure development, and geographic information systems (GIS).

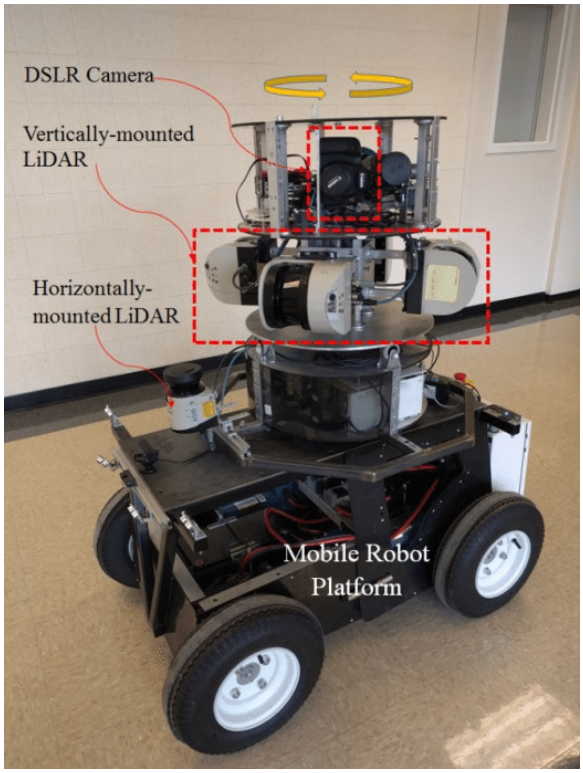


Figure 2.6: Mobile robot platform with a hybrid 3D scanning system. Source: [42].

2.4 Data storage formats

Generally speaking, point cloud data is a straightforward set of XYZ coordinates describing its position, along with optional metadata. The challenges of storing point cloud data come from the sheer volume.

There exists a large amount of different file formats, creating inconveniences as far as inter-operability goes, but the different point cloud store-able file formats can be mostly grouped into ASCII and binary formats. Some formats are capable of both.

- ASCII – store data in ASCII format. Examples are XYZ, OBJ (with some proprietary binary exceptions), PTX (Leica) and ASC.
- Binary – store data directly in binary code. Among the examples FLS (Faro), and LAS can be found.
- Both – PLY, FBX. E57, PCD, etc.

Within an ASCII file, a record is represented by a set of (x, y, z) coordinates. Metadata, such as color and intensity are present in some formats. The biggest benefit of using ASCII formats stems from the higher degree of universal accessibility due to the standardised abstraction. For example, information stored in ASCII formats can easily be opened and read in the most common text editors. While that's not exactly a full-on visualization, it makes working with these formats simpler. This results in ASCII formats usually being recommended for long-term archiving.

Of course, this higher degree of accessibility comes at a cost, and that is the volume of storage required, which also comes with the undesired effect of decreased read speeds.

Binary file formats are more compact and can carry more information. It's possible to include file signatures, software information and metadata for each coordinate, with the added benefit of increased access speeds, partly due to the more compact size, but also because they can be spatially indexed, allowing them to be read in parts, rather than sequentially [63].

This also means that there's a higher degree of diversity and less accessibility present in binary formats.

- OBJ – a simple geometry definition data format that only represents 3D geometry, normals, color, and texture. It is commonly ASCII, however there are some proprietary binary versions of OBJ.
- PLY – supports a relatively simple description of a single object as a list of nominally flat polygons. A variety of properties can be stored, including color and transparency, surface normals, texture coordinates and data confidence values. The format permits one to have different properties for the front and back of a polygon [70].
- PCD – is a file format specifically created for storing point cloud data (as the name suggests, being literally called Point Cloud Data) with the intention to complement existing file formats by supporting some of the extensions that PCL (Point Cloud Library) brings to n-D point cloud processing.
- E57 – is a vendor-neutral file format for point cloud storage. It can also be used to store images and metadata produced by laser scanners and other 3D imaging systems. It is compact and widely used. It also utilises binary code in tandem with ASCII, providing much of the accessibility of ASCII and the speed of binary. E57 can represent normals, colors, and scalar field intensity [63].
- LAS – LASer file format is specifically designed for the interchange and archiving of LiDAR data. It uses 4 types of record. All data values are binary in little-endian format. The record types are as follows (in the order they appear in a LAS file) [45]:
    1. The mandatory single header block – contains a file signature (LASF), basic metadata identifying the project, number, and type of point data records, dimensions of x,y,z range included, and pointers to the major sections of the file.
    2. Optional Variable Length Records (VLRs) – contain variable types of data including projection information, metadata, waveform packet information, and user application data. VLRs are limited to a data payload of 65,535 bytes.
    3. Point Data Records – form the primary content of the file.
    4. Optional Extended Variable Length Records (EVLRs) – allow a larger data payload than VLRs and can be conveniently appended to the end of a LAS file.

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
0.81915 0.32 0 4.2108e+06
0.97192 0.278 0 4.2108e+06
0.944 0.29474 0 4.2108e+06
0.98111 0.24247 0 4.2108e+06
0.93655 0.26143 0 4.2108e+06
...
```

Listing 1: Example contents of a PCD file

Compression

As stated before, point cloud data tends to be very storage-heavy. The large quantity of points required to fully capture a physical scene can easily range in the millions, to the point where sometimes even splitting the scene into multiple point clouds is necessary. Considering the large scale, compression may be required for effective use of storage or transferring data. Just like there's many file formats available, there's also many different approaches to compressing point clouds, with new publications being published at a rapid pace.

We define the following qualities of compression [41]:

- lossy vs. lossless – lossy schemes compress the shape the points represent rather than the exact point coordinates by allowing their positions to change slightly as long as they remain faithful to the underlying surface. They are mainly used in visualization-only applications. Lossless schemes compress point coordinates represented with uniform precision as scaled integers.

- progressive versus non-progressive – progressive techniques compress data in such a way that the decoder can start displaying lower fidelity version of the whole point cloud even before the full data has been decoded. More data is subsequently added with progress of the decompression. Progressive schemes are important for instant-feedback visualization. Non-progressive schemes serve mainly for data transmission and storage.

- streaming versus non-streaming – streaming schemes continously output points as the compression / decompression of data is ongoing, allowing for continous streaming of data, whereas non-streaming schemes require the full dataset to be loaded into

memory (usually inside supporting temporary data structures) before decompression can begin and data is able to be output.

- point-permuting versus order-preserving – in point-permuting schemes, the original ordering of points is not preserved. This is mainly due to the fact that they make use of imposing a clever canonical ordering onto the points that results in small residuals. Order-preserving techniques do not re-order the points.
- sequential versus random-access – random access schemes are able to seek inside the file and only decompress the requested parts (granularity usually being limited to blocks of points).

When speaking about point cloud data, depending on the environment in which it is used, compression can take on a few different connotations. In essence, we're always speaking about reducing the space requirements of data. However, the way that is achieved can differ depending on the use case.

Generally, point clouds are compressed for two different broad-spectrum use cases:

- Compression for real-time transfer and processing
- Compression for data storage

A good compression algorithm can save around ~90% of the original file size. Specifically, LAS files compressed using the LASzip compression method are only 7 to 25 percent of the original file size. The LASzip compression is lossless, non-progressive, streaming, order-preserving, and provides random-access [41].

2.5 Point cloud processing and operations

This section features the typical operations performed on point cloud data [34].

Visualization

Visualization is the most basic operation and usually constitutes a stepping stone for other point cloud operations. Oftentimes, visualization represents the use case as a whole (where simply displaying the data as a 3D model in 3D space is all that's required).

Point selection

Point selection is an important feature for manual classification tasks. We talk about single point selection, where a user manually selects a single point from the point cloud or multiple point selection using one of the possible algorithm for 3d space point selection (for example the LassoNet lasso selection [25]).

Primitive fitting

Fitting geometric primitives to 3D point cloud data bridges a gap between low-level digitized 3D data and highlevel structural information on the underlying 3D shapes. As such, it enables many downstream applications in 3D data processing. Primitive fitting usually incorporates the least square methodology to compute the spatial parameters that define simple geometric figures. Usually, RANSAC-based iterative methods are used, but this

publication [43] proposed a neural network based fitting method (Supervised Primitive Fitting Network (SPFN)).

Segmentation

Point cloud segmentation is the process of dividing a point cloud into (meaningful and homogeneous if binary segmentation) segments based on certain criteria. Segmentation is crucial for understanding and extracting information from 3D data. An example of a segmentation operation would be the separation of points based on intensity values into discrete groups or grouping by the classification value.

Classification

Classification of point cloud data involves assigning semantic labels or categories to individual points within a point cloud. This process is essential for understanding the content of 3D data and is commonly used in applications such as object recognition, scene understanding, and autonomous navigation. The process of classification often involves manual effort by way of point selection and subsequent labeling. The task of classification is also a great candidate for AI-powered automation, for example as presented in [44].

Filtering

Point cloud filtering involves the process of selectively retaining or removing points based on certain criteria. Some other important point cloud operations could be categorized under the filtering process, such as:

- Noise reduction – the process of filtering out unwanted information, e.g. decontaminating the point cloud.
- Outlier detection / removal – points that deviate significantly from the overall characteristics of the point cloud are removed.
- Decimation – also known as downsampling or simplification, decimation is the process of reducing the resolution of a given point cloud, e.g. reducing the amount of points, usually with the intention of reducing computational load.

Transformation

Transformation refers to applying geometric operations to manipulate the position, orientation, and scale of the points within a point cloud, but more broadly it encompasses any modification of the geometric and spatial properties of the data.

- Scaling, translation, rotation – same as any other 3D model
- Cropping – (mostly manual) transformation, which allows for the creation of a point cloud with only the elements that falls within the space of interest.
- Merging – is performed when several point clouds of the same object were collected from different angles or positions each having its own coordinate frame and there is the need to convert all of them into a single spatial coherent point cloud.

Chapter 3

Existing Tools and Technologies

This chapter provides an overview of the current landscape of tools and technologies for working with point cloud data. The discussion is structured in three main parts:

First, we examine standalone tools and applications that are ready to use out of the box for point cloud processing, visualization, and editing. These include both desktop and web-based solutions that are widely used in industry and research. A summary table is presented to outline the main advantages and disadvantages of each application, focusing on their core features, licensing, and suitability for various point cloud tasks.

Next, we take a closer look at selected standalone tools from the table, discussing their capabilities in more detail and analyzing why they may or may not be suitable for the use case presented in this thesis. This section highlights specific limitations or strengths that influence their usability for tasks such as visualization, annotation, integration with map backgrounds, and support for additional modalities, as proposed by the thesis assignment.

Finally, the chapter transitions to an overview of available options for building web-based point cloud applications. This includes a discussion of frameworks and libraries such as Potree, iTowns, Three.js, and others, which provide the foundation for developing custom web applications for point cloud visualization and processing. The strengths and limitations of these technologies are considered in the context of the goals of this thesis.

3.1 Standalone applications

A fair range of products, usable for the purpose of working with point cloud data, is available on the market. Some products take a specialized approach, focusing on just a few aspects of geospatial data visualization and/or management, while others aim to become more of a fully featured tool. The quality of some of these products is undeniable and whether they target a niche part of point cloud rendering or cast a wider net, they usually do their job well.

However, despite their strengths, these tools are generally not suitable for the use case addressed in this thesis. Most existing standalone applications are either desktop-based, requiring installation and limiting accessibility, or web-based with restricted functionality focused mainly on visualization or classification. They often lack integrated support for interactive annotation workflows, efficient handling of very large point clouds, or the ability to combine point cloud data with additional modalities such as map backgrounds and imagery. Furthermore, few of these tools are designed for extensibility or integration with modern web technologies and frameworks, making it difficult to implement custom features such

as asynchronous annotation assistance, efficient object labeling, or seamless collaboration. As the thesis assignment requires the design and implementation of a web-based tool for browsing, annotating, and efficiently processing mobile mapping point clouds—along with support for map layers, imagery, and advanced annotation mechanisms—existing solutions do not fully meet these requirements.

In Table 3.1, a summary of some of the available standalone tools for point cloud visualization, management and/or processing is presented. The table includes both desktop and web-based applications, along with their main features, advantages, and disadvantages.

CloudCompare

CloudCompare is a widely recognized open-source desktop application for point cloud processing, offering a comprehensive suite of tools for visualization, registration, segmentation, and analysis [36]. It supports a variety of file formats, including LAS and LAZ, and can handle large datasets efficiently. Classification is facilitated through plugins such as CANUPO for binary classification and 3DMASC for multi-class classification using Random Forests, enabling users to label points as ground, vegetation, or other classes [4]. Additionally, CloudCompare supports georeferencing, allowing point clouds to be aligned with coordinate systems, which provides some geospatial context [3].

However, CloudCompare falls short of the thesis requirements in several ways. As a desktop application, it requires installation and specific system configurations, limiting accessibility for users who need a solution not reliant on the device. While it supports georeferenced data, there is no native integration with map backgrounds, such as OpenStreetMap or satellite imagery, which is essential for visualizing mobile mapping data in its real-world context. Furthermore, CloudCompare does not offer functionality for displaying panoramic imagery or managing data in a cloud-based backend, both critical for the thesis use case. Its steep learning curve also makes it less user-friendly for non-expert users, further reducing its suitability.

Leica Cyclone

Leica Cyclone is a commercial desktop software suite designed for processing point cloud data, particularly from Leica laser scanners [10]. It provides robust tools for visualization, registration, modeling, and classification, with both automatic and manual methods for assigning class codes to points, such as buildings or vegetation [5]. Cyclone II TOPO, a specialized module, supports the creation of topographic maps from point clouds, indicating some capability for geospatial workflows [11]. The software also integrates panoramic imagery from scanners, which can be used for visualization.

Despite its strengths, Leica Cyclone is not suitable for the thesis use case. Its desktop-based nature requires installation and high-end hardware, restricting accessibility and scalability. While it supports topographic map creation, there is no evidence of direct integration with dynamic map backgrounds like those required for mobile mapping visualization. The lack of cloud-based data management and specific support for panoramic imagery further limits its applicability. Additionally, its high cost and optimization for Leica hardware may pose barriers for users without access to such equipment [10].

FARO Scene

FARO Scene is another commercial desktop application tailored for processing point cloud data from FARO laser scanners [32]. It offers advanced visualization, registration, and panoramic view capabilities, making it a staple in industries like surveying and construction. Scene supports some geospatial workflows, such as aligning point clouds in coordinate systems, and provides tools for basic measurements and annotations. However, its classification capabilities are limited when compared to specialized tools, focusing more on visualization and registration.

FARO Scene does not meet the thesis requirements due to its desktop-based architecture, which necessitates installation and specific system configurations. It lacks native integration with map backgrounds and does not support displaying imagery, both critical for mobile mapping applications. Additionally, there is no cloud-based backend for data management, and its optimization for FARO scanners may limit compatibility with other data sources. The high licensing cost further reduces its accessibility for a broad user base [32].

Potree

Potree is an open-source web-based viewer designed for rendering large point clouds in a browser using WebGL [58]. Potree supports displaying classified point clouds and basic annotations, such as adding labels or measurements, but lacks tools for manual classification, where users select points and assign classes interactively. What Potree excels at is displaying a massive number of points at the same time using an octree to structure the points.

While Potree aligns with the thesis's web-based requirement and supports map background integration, it falls short in other areas. Its primary focus is visualization, not editing or classification, making it unsuitable for the manual annotation of objects like traffic signs or vegetation. Additionally, Potree does not natively support panoramic imagery display or provide a backend for data management and cloud storage. The need to convert point clouds to a specific format using PotreeConverter adds complexity, further limiting its fit for the thesis use case [58].

Pointly

Pointly is a commercial web-based platform specializing in the classification and vectorization of point clouds [55]. It offers both automatic (AI-based) and manual annotation tools, such as 3D bounding boxes and polygon lasso, allowing users to efficiently label points as specific classes, such as roads or trees [54]. Its cloud-based architecture supports data management and storage, aligning with the thesis's backend requirements. It also allows for the displaying of map tiles in 3D space, localizing the point cloud data in their geospatial context. Pointly's user-friendly interface and accessibility without installation make it a strong candidate for web-based applications.

However, while it is the closest and most complete in terms of the goals presented in this thesis and as such represents a very good baseline comparison, Pointly does not fully meet the thesis requirements. There is no evidence of support for displaying imagery or other modalities, both useful for visualizing mobile mapping data in its geospatial context. While it excels in classification, its subscription-based model may introduce cost barriers for some users [55].

Cesium

Cesium is an open-source web-based 3D geospatial viewer that supports point clouds through the 3D Tiles format, offering excellent integration with map backgrounds for geospatial visualization [1]. It can display classified point clouds and apply styling based on attributes like classification, making it suitable for visualizing mobile mapping data [13]. Cesium also supports dynamic annotations using classification volumes, but these are based on geometric criteria rather than manual point selection and labeling [59].

Despite its strengths in visualization and geospatial context, Cesium does not provide tools for manual classification, a critical requirement for the thesis. It also lacks support for displaying 360-degree imagery and does not offer a built-in backend for data management or cloud storage. The preprocessing requirement of converting point clouds to 3D tiles adds additional overhead, which would need to be taken care of outside the confines of the application, reducing its practicality for the thesis use case [1].

Conclusion

The evaluated tools demonstrate significant capabilities in point cloud processing, but none fully satisfy the thesis requirements. Desktop applications like CloudCompare, Leica Cyclone, and FARO Scene offer robust functionality but are limited by their need for installation and lack of web-based accessibility. Web-based tools such as Potree, Pointly, and Cesium provide browser-based access but fall short at various requirements. This analysis underscores the need for a specialized web-based application that integrates these features to support efficient browsing, annotation, and management of mobile mapping point cloud data, as proposed in this thesis.

3.2 Resources for building a new web-based application

With the limitations of existing standalone tools established, the next step is to explore the available frameworks, libraries, and modules that can be used to build a web-based application tailored to the requirements of this thesis. This section introduces and compares key open-source and commercial resources that enable the visualization, interaction, and processing of point cloud data directly in the browser. The focus is on technologies that support integration with map backgrounds, efficient rendering of large datasets, extensibility for annotation workflows and compatibility with modern web development practices, such as Vue.js for the frontend and FastAPI for the backend. The thesis requirements include rendering point clouds with map backgrounds to show them in a geospatial context. Further, the requirements include the support of manual classification by selecting points and assigning classes, providing backend data management with cloud storage and displaying additional modalities such as panoramic imagery taken at the time of point cloud acquisition.

The following subsections provide an overview of relevant technologies such as Three.js, Potree, CesiumJS, Deck.gl, iTowns, and WebGL. The goal is to assess their capabilities, strengths, weaknesses, and other limitations as well as their potential for integration and extension to meet the thesis requirements. Each technology is evaluated based on its ability to handle large point clouds, integrate with map backgrounds and of course, to support manual classification while integrating well with Vue.js and FastAPI.

Three.js

Three.js is a well-established JavaScript library for creating and displaying 3D graphics in a web browser, built on top of WebGL. It provides a comprehensive set of tools for rendering 3D scenes, including support for point clouds through objects like `Points` and `PointsMaterial`. Three.js is highly versatile, allowing for the creation of custom 3D applications with fine-grained control over rendering and interaction [19].

- **Pros**:
  - Highly flexible, offering extensive customization options for 3D rendering, suitable for building complex applications.
  - Large and active community with abundant resources, tutorials, and examples [19].
  - Compatible with Vue.js for frontend development, facilitating integration into modern web applications.
  - Supports direct handling of LAZ files in the browser using libraries like laszip.js, avoiding server-side conversion [8].
  - Enables custom annotation tools through its event system and rendering capabilities, supporting manual classification workflows.
- **Cons**:
  - Not optimized for large point clouds, requiring additional effort to implement level-of-detail (LOD) techniques for performance.
  - Lacks native support for geospatial map backgrounds or coordinate systems, necessitating integration with libraries like Proj4js [15] or Leaflet [9].
  - Requires additional implementation or library integration to support displaying panoramic imagery [12].
- **Reasons for suitability/unsuitability**:
  - *Suitability*: Three.js presents itself as a strong choice for the purpose of developing a web-based client application capable of point cloud handling, largely due to its flexibility and adaptability. Its general-purpose nature allows for simple integration with ecosystems built on Vue.js and FastAPI or other frameworks for frontend and backend requirements. Its suitability for this task is further enhanced by the large community and extensive documentation surrounding it, making it a reliable choice for developing semi-complex 3D features, such as custom annotation tools and integration with other features.
  - *Unsuitability*: Optimizing Three.js for very large point clouds and implementing geospatial features like map backgrounds require significant development effort. The need for additional libraries to support specific features may increase complexity.

Potree

Potree is an open-source JavaScript library specifically designed for rendering large point clouds in a web browser, built on top of Three.js [58]. It provides optimized rendering

techniques, such as octree-based LOD, and supports various file formats, including LAS and LAZ, using laszip.js for decompression.

- **Pros**:
  - Optimized for rendering large point clouds efficiently, capable of handling datasets with billions of points [58].
  - Includes features like measurements, basic annotations. Providing geospatial context is possible in combination with other javascript libraries. An implementation of such a scenario was done for example by the Remote Sensing and Geospatial Analysis Laboratory (University of Minnesota) [65].
  - Open-source, allowing forking and customization to add required features.
- **Cons**:
  - The usage of Potree requires preprocessing point clouds into Potree's optimized format using PotreeConverter, which presents an additional hurdle in managing LAZ files.
  - As a highly specialized library, extending Potree for custom annotation workflows or integrating it with other frameworks like Vue.js poses a bigger challenge than using a general-purpose library like Three.js, for example.
  - Lacks built-in backend data management, requiring possibly difficult integration with other tools for cloud storage and asynchronous processing.
- **Reasons for suitability/unsuitability**:
  - *Suitability*: Potree is highly suitable for point cloud visualization due to its optimized rendering and built-in support for map backgrounds and basic annotations. Its ability to handle LAZ files directly and its open-source nature make it a strong candidate for extension to support manual classification and panoramic or other imagery.
  - *Unsuitability*: Its specialized focus may limit flexibility for custom features compared to Three.js. The potential need for format conversion for optimal performance could be a drawback if direct LAZ handling without preprocessing is prioritized.

CesiumJS

CesiumJS is a JavaScript library designed for 3D geospatial visualization, offering robust support for rendering 3D globes and 2D maps in a web browser [1]. The goal of CesiumJS is not specifically 3D rendering, let alone point cloud rendering. The library is created for the purpose of visualizing geospatial data. However, it does support rendering point clouds through the 3D Tiles format. Similar to Potree, the usage of 3D Tiles instead of points for rendering purposes requires a pre-processing step. Point cloud data needs to be first converted to 3D tiles using Cesium Ion [24].

Cesium Ion is a cloud platform for the management, handling, and efficient streaming of geospatial data. It offers existing collections of 3D content and allows for the curating of custom content. All 3D data is converted into Cesium's implementation of 3D tiles. The point cloud tiler also provides an option to use Google's Draco point-cloud-aware

compression library [39], reducing file size significantly. A difference of up to 3x reduction has been noted when compared the LASzip compression.

- **Pros**:
  - Excels in geospatial visualization with built-in integration of map backgrounds [1].
  - Efficiently handles large datasets via 3D Tiles [13].
  - Compatible with Vue.js, facilitating frontend integration [16].
  - Supports styling point clouds based on attributes like classification, useful for visualization [17].
- **Cons**:
  - Requires conversion of point clouds to 3D Tiles, adding preprocessing overhead [2].
  - Lacks native tools for manual classification, requiring custom development [59].
  - Does not natively support panoramic imagery, necessitating integration with libraries like Pannellum [12].
  - Steeper learning curve and more difficult integration to foreign ecosystems.
- **Reasons for suitability/unsuitability**:
  - *Suitability*: CesiumJS is well-suited for geospatial visualization, with map background integration being available without further development, making it a viable candidate for rendering point clouds in a geospatial context. The 3D Tile format ensures efficient streaming of data and other handling of large datasets, and its compatibility with Vue.js supports frontend development.
  - *Unsuitability*: The requirement for 3D Tiles conversion may conflict with the preference for direct file handling. The lack of native support for manual classification and panoramic or other imagery requires significant custom development, and while CesiumJS offers the use of Cesium Ion, if currently non-existent features were to be requirend, backend integration with FastAPI (or other) would be necessary for data management.

Deck.gl

Deck.gl is a mature, GPU-accelerated framework (originating from Uber's vis.gl suite), able to render points in the order of magnitude of millions smoothly on common hardware. In addition to its `PointCloudLayer` for LiDAR data, deck.gl offers many other layer types (hexagons, grids, paths, etc.) that can be combined into layered visualizations. It supports multiple coordinate systems and view modes (2D map, 3D globe, and custom views) and includes built-in interaction controls (zooming, panning, picking, tooltips, etc.). Deck.gl is open-source with extensive documentation and examples; for example, Uber engineers have used deck.gl together with 3D Tiles to visualize city-scale point clouds with hundreds of millions of points. Its React-friendly design also allows integration into other frameworks (Vue.js, Angular, plain JavaScript) via wrapper components or the "deck.core" API [50].

Vis.gl is the umbrella ecosystem that ties together several interoperable visualization libraries—deck.gl, loaders.gl, luma.gl, and others—under open governance. Its architecture

is deliberately modular: each framework addresses a specific role, and yet they share a consistent API design and data model so that you can mix and match layers, loaders, math utilities, and rendering engines in a seamless way.

For the purpose of loading the data from different file types, `loaders.gl` (another part of the vis.gl ecosystem) is often used in combination with deck.gl. Loaders.gl is a modular JavaScript library, which contains loaders for loading structured data, including geospatial and 3D formats, directly in the browser. For point clouds, it provides support for LAS and LAZ formats via the `LASLoader`. Additionally, loaders.gl supports formats such as PLY, PCD, and 3D Tiles, which broadens its applicability to various point cloud data sources. Loaders.gl also has the option to use web workers, making it possible to parse and stream large datasets without blocking the main thread. The modular design allows dependent applications to include only the loaders they need, keeping a lightweight architecture. When used together, loaders.gl can handle the efficient extraction and preprocessing of point cloud data, while deck.gl takes care of rendering and interaction within the browser. The important thing to note is that loaders.gl are not reliant on deck.gl or other parts of the vis.gl ecosystem. Each tool is usable standalone.

- **Pros**:
  - Supports point cloud rendering with LAS and LAZ formats via loaders.gl [14].
  - Seamlessly integrates with map libraries for geospatial visualization [20].
  - Reasonably compatible with javascript frameworks, mainly React (an example of deck.gl being used in React can be seen in one of deck.gl's demos [6]).
  - Handles large datasets efficiently with WebGL optimization.
  - Has built-in event handling (hover, click, drag) and controllers (orbit, map navigation) to support user interaction.
- **Cons**:
  - Primarily optimized for 2D map visualizations, potentially less efficient for large 3D point clouds compared to specialized libraries [50].
  - Lacks native support for manual classification – annotation tools must be built on top of deck.gl's layer system.
  - Does not support panoramic imagery natively, needing integration of additional libraries.
- **Reasons for suitability/unsuitability**:
  - *Suitability*: Deck.gl excels at geospatial visualization. Its integration with map engines (like Mapbox GL) fits the thesis requirement of displaying point clouds over map backgrounds. The ability to use it within Vue.js applications makes it a decent frontend choice. Its GPU-based performance means it can handle mobile mapping data sets reasonably efficiently.
  - *Unsuitability*: Its general-purpose focus may make it less efficient than specialized point cloud libraries for very large datasets. Custom development is needed for manual classification and panoramic or other imagery. It also does not provide server-side data processing out of the box, so FastAPI (or other backends) must be used to supply and manage the point cloud data.

iTowns

iTowns is a Three.js-based open-source framework for 3D geospatial visualization, developed by IGN and designed to handle various geospatial data types, including point clouds via Potree and Entwine formats [7].

Built with interoperability in mind, iTowns supports multiple data sources through standardized protocols and file formats. It can connect to common web map services (WMS, WMTS, TMS), render elevation data, vector features and point clouds (including LAS/LAZ via PotreeSource, Entwine tiles, COPC, and 3D Tiles) [18]. It also supports oriented imagery through 'OrientedImageLayer', enabling multi-modal visualization alongside map and point cloud layers.

When deploying point clouds, iTowns relies on embedding Potree-rendered content, meaning that instead of using pure iTowns-native rendering, it often encapsulates Potree as a layer. While this enables access to Potree's optimized octree streaming, it introduces potential integration challenges: coordinate reprojection mismatches, bounding-box misalignment, and camera control conflicts have all been reported when combining Potree with iTowns' internal rendering engine. In such cases, it may be technically simpler—and more reliable for pure point cloud use—to use Potree standalone rather than via iTowns, avoiding layering complexities and unintended interoperability issues.

- **Pros**:
  - Designed for geospatial data with smooth map background integration via WMS/WMTS/TMS and elevation layers [18].
  - Native support for point clouds in Potree, Entwine, and COPC formats, and compatibility with 3D Tiles streaming.
  - Supports oriented imagery using 'OrientedImageLayer' for true multi-modal display.
  - Server-aware architecture: can consume Entwine/Greyhound, PostGIS, 3D Tiles services or cloud-hosted buckets—simplifying backend integration.
  - Open-source with contributions from IGN, Oslandia, and academic labs, offering industrial use scenarios and potential for customization.
- **Cons**:
  - Integration complexity: since iTowns uses Potree under the hood, there is risk of misaligned camera controls, projection mismatches, or unreliable placement of point clouds—making it sometimes easier to just use Potree directly.
  - Overhead for point cloud focused workflows: features iTowns provides (multi-modal layers, map protocols, interaction tools) may be beyond what the thesis requires, adding unnecessary complexity.
  - Community and docs smaller than larger stacks (e.g. CesiumJS, Three.js), meaning custom extendability may be constrained.
  - No built-in annotation UI: you still need to build classification, labeling, selection, or editing workflows yourself.
  - Imagery capabilities, while present, are less mature than dedicated panoramic-viewing libraries; deep zoom or 360° support may require additional effort.

- **Reasons for suitability/unsuitability**:
  - *Suitability*: iTowns aligns with thesis goals when multi-modal display is required alongside backend-served point cloud and map data. Its data-source architecture handles streaming from Entwine/Greyhound or cloud buckets natively—facilitating server-side integration with minimal ETL.
  - *Unsuitability*: For workflows centered on point cloud annotation and dataset handling, the Potree-based rendering in iTowns may introduce integration friction. In such cases, using Potree alone may be more straightforward, avoiding the added abstraction layer and potential reprojection and camera-control issues.

WebGL

WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES2.0, exposed through the HTML5 Canvas element as Document Object Model interfaces. It exposes a JavaScript API for rendering interactive 2D and 3D graphics within compatible web browsers without plug-ins [21].

OpenGL ES ("Embedded Systems") 2.0 is an adaptation of the standard OpenGL API, designed specifically for devices with more limited computing power, such as mobile phones or tablets. WebGL is designed to use, and be used in conjunction with standard web technology; thus, while the 3D component of a web page is drawn with the WebGL API via Javascript, the page itself is built with standard HTML [30].

WebGL serves as the foundation for libraries like Three.js and Potree, providing direct access to GPU rendering capabilities. While this makes using WebGL as the base for point cloud rendering a very strong choice, implementing base structure and optimizations like those provided in existing libraries based on WebGL is impractical in the scope of this thesis.

- **Pros**:
  - Offers low-level control over GPU rendering, enabling highly optimized and customized rendering.
  - Provides maximum flexibility to implement any 3D rendering technique, including those not supported by higher-level libraries.
- **Cons**:
  - Requires in-depth knowledge of computer graphics and WebGL specifics, making development complex and time-consuming.
  - Building a full application from scratch is extremely time-intensive, far beyond the scope of a thesis project.
  - Lacks higher-level abstractions for common tasks like loading models, handling user input, or managing scenes.
- **Reasons for suitability/unsuitability**:
  - *Suitability*: WebGL offers the most possibilities for customization and optimization, as it is the underlying API for all other 3D libraries. It provides direct control over rendering, which could be beneficial for highly specialized applications.

– *Unsuitability*: For a thesis project, using WebGL directly is not feasible due to its complexity and the extensive development effort required. Libraries like Three.js, which abstract WebGL, are more practical for building a functional application within the project's constraints.

Technology	**Pros**	**Cons**
Three.js	Highly flexible, large community, Vue.js compatible, direct LAZ support via laszip.js, custom annotation capabilities	Not optimized for large point clouds, no native geospatial support, requires additional libraries for panoramic imagery
Potree	Optimized for large point clouds, supports LAZ files, map background integration, open-source	Requires format conversion for optimal performance, limited customization, no built-in backend
CesiumJS	Excellent geospatial visualization, supports point clouds via 3D Tiles, map background integration, Vue.js compatible	Requires 3D Tiles conversion, no native manual classification or panoramic imagery support, steeper learning curve
Deck.gl	Supports LAS/LAZ, map library integration, Vue.js compatible, efficient for large datasets	Optimized for 2D maps, no native classification or panoramic imagery, no built-in backend
iTowns	Geospatial focus, supports point clouds via Potree/Entwine, open-source	Smaller community, limited annotation support, unclear panoramic imagery support
WebGL	Low-level control, maximum flexibility for custom rendering	High complexity, time-intensive development, no built-in features

Table 3.2: Comparison of web-based technologies for point cloud applications

This section has provided an overview of the most relevant available web-based technologies for building a point cloud application. The idea was to highlight their key features, advantages, and limitations. The suitability of each technology for the thesis project is further discussed in the subsequent chapters, where the design and implementation approaches are outlined. A general overview can be seen in Table 3.2.

Tool Name	Type	License	Pros	Cons
Autodesk ReCap	Desktop	Commercial	Integrated with Autodesk ecosystem, good for AEC	Subscription-based, requires installation
Bentley ContextCapture	Desktop	Commercial	Advanced photogrammetry, good for large projects	Costly, requires installation
CloudCompare	Desktop	Open-source	Free, powerful, large community, supports many formats	Requires installation, steep learning curve
FARO Scene	Desktop	Commercial	Industry-standard, robust features, good support, panoramic visualization	Costly, requires installation, optimized for FARO scanners
Leica Cyclone	Desktop	Commercial	Industry-standard, robust features, good support	Costly, requires installation
LP360	Desktop	Commercial	Specialized for LiDAR, good for mobile data	Costly, desktop-based
Meshlab	Desktop	Open-source	Free, good for mesh and point cloud editing	Less powerful, limited features
TerraScan	Desktop	Commercial	Advanced classification, used in forestry	High cost, requires installation
Trimble RealWorks	Desktop	Commercial	Professional-grade, supports Trimble hardware	High cost, requires installation
Cesium	Web	Open-source	Powerful 3D geospatial viewer, supports point clouds	Requires 3D Tiles format, steeper learning curve
GeoSignum Pointer	Web	Commercial	Web-based, automated classification	Limited to classification, subscription-based
PointCab Share	Web	Free	Free, web-based, no installation, basic interactions	Limited to viewing, tied to PointCab Origins
Pointly	Web	Commercial	Web-based, no installation, good for classification	Subscription-based, limited to specific tasks
Potree	Web	Open-source	Free, good for visualization, web-based	Limited to visualization, no editing

Table 3.1: Overview of existing standalone applications for point cloud rendering and/or processing.

Chapter 4

Design Rationale and Implementation Overview

Following the comparative overview of available visualization and processing tools in the previous chapter, this section introduces the architecture and core design decisions the application developed as part of this thesis. The selected technology stack consists of **Vue.js** for the frontend framework, **Three.js** for the purposes of 3D rendering together with user interaction, and **FastAPI** as the backend service framework. The choice was not made arbitratily and several thoughts were taken into account. The choice reflects both practical development considerations and the specific requirements outlined earlier – notably, interactive point cloud visualization, annotation capabilities, and integration with supplementary data.

One of the central decisions in designing the frontend was the use of **Three.js** for rendering 3D content. Compared to more specialized tools like Potree or CesiumJS, Three.js offers a general-purpose rendering framework that can be shaped to fit the exact needs of the application. While tools like Potree are extremely efficient when it comes to rendering very large point clouds, they are often rigid in terms of customization and require data preprocessing steps that introduce additional complexity. In contrast, Three.js provides full control over the rendering pipeline, camera behavior, and user interaction logic, which is essential for building a responsive and extensible annotation workflow. The tradeoff, of course, is that features like level-of-detail rendering or geospatial context must be implemented manually or via auxiliary libraries. However, this tradeoff was considered acceptable in exchange for greater flexibility. So, to reiterate and sum up the choice – while using other libraries would require building additional features on top of a pre-existing, sometimes rigid, implementation, Three.js integrates more naturally as a component within a broader application architecture, rather than serving as a fixed, standalone core around which everything else must be structured.

The choice of **Vue.js** as the frontend framework was influenced primarily by familiarity with the framework, rather than by its integration with the specific visualization tools discussed in the previous chapter. As observed, many modern WebGL-based visualization libraries, for example those from the vis.gl ecosystem like Deck.gl, are designed with React in mind and provide React components out of the box. Others often offer companion packages or wrappers that adapt the core library for use within the React ecosystem, or they integrate more smoothly through established patterns – something that is frequently reflected in the structure of official or community-driven examples. While this means that

integrating certain libraries with Vue.js can require additional effort, the core functionality of Three.js remains independent of any frontend framework. While not the most popular choice for geospatial applications, Vue's flexibility and lightweight syntax made it a practical option in this case.

The backend is built using **FastAPI**, a modern Python framework well suited to lightweight web services. Given the nature of the thesis, which revolves around interactive point cloud rendering, the majority of the computational and architectural complexity lies on the frontend. Most of the processing, rendering, and user interaction is handled directly in the browser. Consequently, the backend is not required to be particularly elaborate and can remain relatively lightweight, primarily serving as a means of cloud-based storage and data delivery, with flexibility to accommodate future extensions if needed. Given this limited but essential role, FastAPI was selected for its performance, straightforward syntax, and excellent support for modern Python tooling such as type hints and asynchronous endpoints.

Compared to more traditional frameworks like Django or Flask (or other ecosystems / languages entirely), FastAPI offers a cleaner interface for building RESTful APIs quickly, along with built-in automatic documentation and validation through OpenAPI. Since most of the computational work takes place on the frontend – including rendering, data parsing, and interaction handling – the backend only needs to be efficient and easy to extend. Should more complex features be required in the future (e.g., user management, background processing, or data analytics), FastAPI leaves sufficient room for expansion without requiring a full rewrite.

As with any technology stack decision, when selecting a database, it is important to keep in mind the role it will fulfill. In this case, the backend primarily handles input data consisting of point clouds, which are stored as files. Therefore, the core requirement is not complex relational querying but rather a straightforward mechanism to store these files, organize them effectively, and maintain metadata about this organization. To remain consistent with the lightweight and flexible nature of the backend, the database solution that best aligns with the application's needs is **MongoDB**. In the context of the backend's role, the rigidity constraints typical of traditional SQL databases are not required. Instead, a NoSQL database offers the necessary flexibility to accommodate the data structures required for managing information about point clouds and other modalities. MongoDB's document-oriented model facilitates seamless storage of hierarchical and diverse metadata without imposing stringent schema definitions. Moreover, its native JSON-like format integrates naturally with modern web technologies and pairs efficiently with FastAPI's architecture. This combination ensures that the backend remains both performant and extensible. This will allow the application to support future growth while avoiding unnecessary complexity.

As with any technology stack decision, when selecting a database, it is important to keep in mind the purpose it will serve. The input data the backend is going to be dealing with, largely, is composed of point clouds. Since point clouds are saved in files, we only really need a way to save these files, structure them and keep track of this structure.

In summary, the chosen stack reflects a balance between technical feasibility, project scope, and prior development experience. While not every tool integrates seamlessly with Vue.js, the combination of Three.js and FastAPI alongside Vue provides a solid foundation for building a modular, responsive, and maintainable application tailored to the goals of this thesis.

4.1 Functional requirements and quality attributes

In this section, the functional and non-functional objectives of the application implemented as part of this thesis are outlined. First, specific requirements for the implementation are defined, taking into account the intended use cases and scenarios such as visualizing, inspecting, and organizing large point cloud datasets in a browser environment. This is followed by a set of quality attributes – core concepts being followed during the implementation. These describe system-wide expectations such as responsiveness, scalability, and robustness rather than individual features.

Functional requirements

- **Interactive loading and viewing of point clouds**
  The application must support loading and rendering of point cloud data (e.g., LAS / LAZ files) through both browser-based and backend-assisted workflows. It must be capable of parsing and visualizing attributes such as spatial coordinates, color, intensity, and classification, enabling interactive inspection within the browser.

- **Flexible rendering approaches**
  The system should provide a baseline rendering mode for visualizing point data, as well as the ability to scale toward more advanced rendering techniques. Support for spatial acceleration structures (e.g., BVH or octrees) is essential to enable efficient selection, interaction, and performance at larger scales.

- **Geospatial context and mapping**
  The application must be capable of rendering point clouds in a geospatial context by overlaying them on map tiles. Geospatial context should be preserved as close to reality as possible. In cases where input data is provided in a local coordinate reference system (e.g., EPSG:5514), the system must be able to perform coordinate transformations to global systems such as WGS 84 (EPSG:4326). If necessary, other measures should be taken to ensure the preservation of correct locality of given data.

- **User interaction and annotation**
  Users must be able to interact with the point cloud through mechanisms such as point selection, possibly others, for its main purpose of point classification. The application should provide tools to manually annotate or classify selected points to support data enrichment, inspection, or editing workflows.

- **File and dataset organization**
  The system must provide a way to organize and access point cloud datasets, including support for browsing collections of files and loading multiple point clouds in parallel. It should expose interfaces for file listing, selection, and efficient data streaming.

- **Supplementary media integration**
  The application should support linking of additional media such as panoramic imagery to specific spatial locations within the point cloud or the point cloud itself. The user should be allowed to navigate between 3D data and contextual reference material in a seamless manner.

Quality attributes

- **Performance**
  The system should maintain a responsive and fluid user experience, minimizing user interface latency and rendering delays during point cloud navigation and interaction, particularly as dataset sizes increase. Backend endpoints should support asynchronous execution where applicable to avoid UI stalling.

- **Usability**
  Prioritize an intuitive and straightforward user interface that facilitates efficient workflows with minimal user effort. Provide clear and immediate visual feedback to support understanding of interactions and system state.

- **Scalability**
  Both frontend and backend components should be capable of scaling to accommodate larger datasets and increasingly complex user workflows. This includes the use of spatial acceleration structures on the frontend and modular, scalable APIs on the backend.

- **Portability**
  The application should ensure compatibility across modern desktop browsers and operating systems by making use of being deployable in standard cloud or containerized environments.

- **Annotation efficiency**
  Facilitate rapid and accurate user annotation processes by designing interactive tools and assistance mechanisms that aim to minimize effort or cognitive load. Incorporate approaches that allow for asynchronous and scalable annotation workflows.

4.2 System architecture

The application architecture is composed of three core services that are containerized using Docker and orchestrated with Docker Compose. Each service runs in its own isolated container. The main goal of using a docker environment is to promote reproducibility and ease of deployment across different environments. The services include:

- **Frontend**: A Vue.js single-page application, built and served from the `frontend` directory. It is accessible at `http://localhost:8080` (container port `8080:8080`) and provides the user interface for browsing, uploading, visualizing, and classifying point cloud data. The frontend communicates with the backend exclusively via HTTP API calls.

- **Backend**: A Python FastAPI application, located in the `backend` directory and started with `uvicorn`. It listens on `http://localhost:8000` (and uses the container port `8000:8000`) and exposes a REST-like API for point cloud management, file operations, folder and panorama organization, and serves as the main interface between the frontend and the database. Uploaded files are stored in a dedicated `uploaded_files` directory, which is mounted as a Docker volume for persistence.

- **Database**: A MongoDB 5.0 instance running in its own container (`mongo`). The database is accessible internally on port `27017` and mapped to `27018` on the host (`27018:27017`) for development and debugging.

All configuration, including environment variables, is managed via `.env` files and Docker Compose. The backend and frontend source code is mounted into their respective containers.

4.3 Frontend design and implementation

The frontend of the application is implemented as a single-page application (SPA) using Vue.js. The project is organized into modules, components, and utility scripts, aiming for a modular architecture. Core UI elements such as the file browser, sidebar, and point cloud controls are implemented as Vue components, while rendering logic and data processing are encapsulated in dedicated JavaScript modules.

A central feature of the frontend is its integration with Three.js. Three.js is used to render point clouds, map backgrounds, and interactive overlays within a WebGL context, enabling GPU-accelerated visualization of millions of points. The rendering logic is encapsulated in dedicated modules such as `PointCloudRenderer` and `PointCloud`, which handle the creation, updating, and display of 3D objects.

For the purpose of making API calls, domain-specific service modules are provided. The frontend communicates with the backend exclusively via these modules.

Point Cloud Rendering with Three.js

Three.js is used as the primary rendering engine for visualizing point clouds. The rendering pipeline is built around the `THREE.Points` class and `BufferGeometry`, allowing efficient GPU-based rendering of millions of points. Each point cloud is represented as a set of buffer attributes (position, color, intensity, etc.), and dynamic color modes (RGB, heightmap, intensity) are supported for interactive exploration. The rendering logic is encapsulated in the `PointCloudRenderer` and `PointCloud` classes, which manage the Three.js scene, camera, controls, and per-point attributes.

Map Background and Coordinate Transformation

To provide spatial context, the application renders a 3D map background beneath the point clouds. Map tiles are fetched from OpenStreetMap assembled into the Three.js scene as textured planes. Since point cloud coordinates are typically provided in a local or national reference system (e.g., EPSG:5514), the `proj4` library is used to transform these coordinates into WGS84 (EPSG:4326) for correct alignment with the map. The map plane is centered and positioned based on the average coordinates of the first loaded point cloud, ensuring that both the map and point cloud data are spatially synchronized. Normalization of point cloud coordinates (centering around the origin) is performed to maximize rendering precision and avoid floating-point artifacts (explained further in following sections).

Scene Management and Statistics

The `SceneManager` module handles maintaining the Three.js scene graph, tracking of loaded point clouds and computing scene statistics (e.g. min, max, average coordinates). The

statistics are then used for camera positioning, normalization, and UI feedback. Additionally, methods for adding, removing, and updating point clouds are provided and used in maintaining the consistency and interactivity.

File Browser and Data Grouping

The frontend features a custom file browser component that allows users to navigate, group, and select point cloud datasets and folders. It provides a hierarchical view of available data and allows for multi-selection and batch operations. Additionally, data grouping is implemented to organize point clouds under logical folders, improving usability for larger projects. The file browser component interacts with the backend via API calls to fetch metadata, initiate downloads, and manage folder structures. Upon selection and loading, point cloud files are streamed from the backend and processed for visualization using the same pipeline as direct client-only uploads.

4.4 Point cloud data workflow

While there are some other modalities to be considered, persisting point clouds fundamentally comes down to writing and storing a large volume of spatial point coordinates. There is, technically, no universally accepted standard for doing this. Consequently, the ecosystem surrounding point cloud storage is quite varied. It should however be noted that the primary reason for this diversity lies in the methods by which point data is acquired. Individual hardware vendors often design their own proprietary file structures to store the point streams generated by their devices. A general overview of commonly used file formats is provided in section 2.4. While this is by no means an exhaustive catalog, it includes the most prevalent formats in which point cloud data is typically encountered.

The data utilized in the development of the application component of this thesis consists entirely of real-world data captured through mobile mapping, the output of which was in the LAS format (specifically compressed using LASzip). For this reason, the application's primary focus is on LAS files. At present, the application supports only .las and .laz files. However, extending support to other formats is straightforward, should the need arise.

As mentioned in previous chapters, the vis.gl suite includes a set of data loaders known as loaders.gl, which facilitate the handling of various data formats. Loaders for tabular, geospatial, and 3D data are all relatively well-supported, with clearly defined and standardized output formats [51]. Since most point cloud data loaders produce similarly structured output, adapting the application to handle a different file format generally only involves substituting the appropriate loader based on the input type. Additionally, should it be required, loaders.gl offers a seamless way to create and integrate custom loaders into the existing ecosystem. Nonetheless, for now, the focus is on LAS data.

This section aims to explain how LAS data is read, parsed, stored, and ultimately used within the application developed as a part of this thesis.

A closer look at the LAS file format

All information about the internal structure of LAS files is based on the current latest (as of writing this thesis) LAS specification published by ASPRS, specifically the LAS Specification 1.4 - R15 [22], last revised in July 2019.

The LASer format is a binary file specification for the storage and exchange of LiDAR point cloud data. The current version, as noted earlier, is LAS 1.4 (revision 15), though earlier versions (1.0–1.3) share the same general structure with relatively minor differences. It consists of a public header block, any number of (optional) Variable Length Records (VLRs), the Point Data Records, and any number of (optional) Extended Variable Length Records (EVLRs). All data are in little-endian format.

- **Public Header Block** – contains generic data such as point numbers and point data bounds.
- **Variable Length Records (VLRs)** – contain variable types of data including projection information, metadata, waveform packet information, and user application data. They are limited to a data payload of 65,535 bytes.
- **Point Data Records** – houses the actual point coordinate data.
- **Extended Variable Length Records (EVLRs)** – (introduced in LAS 1.4) – allow a higher payload than VLRs and have the advantage that they can be appended to the end of a LAS file. This allows, for example, adding projection information to a LAS file without having to rewrite the entire file.

Section	**Purpose**
Public Header Block	Metadata including version, point count, scale factors, offsets, etc.
Variable Length Records (VLRs)	Optional metadata such as projection, CRS, or extra dimensions
Point Data Records	The actual LiDAR point records stored in binary format
Extended Variable Length Records (EVLRs)	Used to store large metadata blocks after point data

Table 4.1: Overview of the LASer file format sections.

Table 4.1 shows the file sections contained in a LAS 1.4 file format and Table 4.2 shows the contents of the Public Header Block. Not all fields are shown. Some fields are either reserved or version-specific (e.g. legacy counts). Fields can be required or optional, all fields not containing data must be zero filled.

Table 4.3 shows the contents of a single Point Data Record. It's important to note that there are a total of 11 possible formats for the Point Data Record, ranging from Format 0 to Format 10, each differing slightly in some of the specific fields they describe.

Formats 1–5 extend Format 0 by adding GPS time, RGB color, and/or waveform packet fields; formats 6–10 use a 30-byte core and support extended classification bits, higher-precision scan angles, and mandatory GPS time etc.

Point data records store integer values $X_{\text{record}}, Y_{\text{record}}, Z_{\text{record}}$ as 32-bit signed integers. The scale and offset values are read from the public header block. The real-world coordinates are computed as:

$$X_{\text{coordinate}} = (X_{\text{record}} \cdot X_{\text{scale}}) + X_{\text{offset}}$$
$$Y_{\text{coordinate}} = (Y_{\text{record}} \cdot Y_{\text{scale}}) + Y_{\text{offset}}$$
$$Z_{\text{coordinate}} = (Z_{\text{record}} \cdot Z_{\text{scale}}) + Z_{\text{offset}}$$

Field	Format	Size	Description
File Signature	char[4]	4 bytes	Always “LASF”
... 20 bytes ...			
Version Major	unsigned char	1 byte	Major version (e.g. 1)
Version Minor	unsigned char	1 byte	Minor version (e.g. 4)
System Identifier	char[32]	32 bytes	Scanner/vendor identifier
... 38 bytes ...			
Offset to Point Data	unsigned long	4 bytes	Byte offset to point records
Number of VLRs	unsigned long	4 bytes	Count of VLRs preceding point data
Point Data Record Format	unsigned char	1 byte	Format code (0–10)
Point Data Record Length	unsigned short	2 bytes	Bytes per point record
... 24 bytes ...			
X Scale Factor	double	8 bytes	Scale for X coordinate
Y Scale Factor	double	8 bytes	Scale for Y coordinate
Z Scale Factor	double	8 bytes	Scale for Z coordinate
X Offset	double	8 bytes	Offset for X origin
Y Offset	double	8 bytes	Offset for Y origin
Z Offset	double	8 bytes	Offset for Z origin
Max X	double	8 bytes	Maximum unscaled X value
Min X	double	8 bytes	Minimum unscaled X value
Max Y	double	8 bytes	Maximum unscaled Y value
Min Y	double	8 bytes	Minimum unscaled Y value
Max Z	double	8 bytes	Maximum unscaled Z value
Min Z	double	8 bytes	Minimum unscaled Z value
... 20 bytes ...			
Number of Point Records	unsigned long long	8 bytes	Total point count (64-bit)
Number of Points by Return	unsigned long long[15]	120 bytes	Counts per return (up to 15)

Table 4.2: Selected fields from the Public Header Block. Omitted fields were replaced with their byte sum. The data was adapted from Table 3 of the LAS 1.4 specification [22].

Before the precise coordinates are computed, the values must first be read. The point coordinates would be extracted from a LAS file as follows:

1. Read public header to determine record format, length, scale, and offset.
2. Seek to the “Offset to Point Data” location.
3. For each point record (length given in header):
   - Read X, Y, Z as 32-bit signed ints.
   - Apply the above math to obtain real coordinates.
   - Optionally parse other fields depending on the record format (e.g. classification, intensity, GPS time).

The overview included in this section shows how LAS stores point clouds in binary form and how applications, such as the one developed as a part of this thesis, can reliably parse and convert LAS (and compressed .laz) coordinates to real world values. Note that although earlier versions of LAS (1.0–1.3) may omit EVLRs or use 32-bit counts, the core ideas of header+scale/offset + integer coordinates remain consistent across versions.

Field	Format	Size	Description
X	long	4 bytes	X coordinate of the point, stored as an integer
Y	long	4 bytes	Y coordinate of the point, stored as an integer
Z	long	4 bytes	Z coordinate of the point, stored as an integer
Intensity	unsigned short	2 bytes	Return pulse strength
Return Number	3 bits (bits 0–2)	3 bits	Position of this return in the pulse (first, second, etc.)
Number of Returns (Given Pulse)	3 bits (bits 3–5)	3 bits	Total number of returns for the given pulse
Scan Direction Flag	1 bit (bit 6)	1 bit	Indicates scanning direction (forward or backward)
Edge of Flight Line	1 bit (bit 7)	1 bit	Indicates if the point is at the edge of a flight line
Classification	unsigned char	1 byte	Encodes the type of object the point represents
Scan Angle Rank	signed char	1 byte	Angle between the laser and the nadir (range: -90 to +90)
User Data	unsigned char	1 byte	Optional field for user-defined information
Point Source ID	unsigned short	2 bytes	ID of the originating sensor or flight line

Table 4.3: Point Data Record Format 0 (adapted from Table 7 of the LAS 1.4 specification [22]).

Loading LAS format point cloud data into Three.js buffers

Upon loading a LAS file in the browser, parsing produces typed arrays for each point attribute. Following is an itemized list of the point data, where N is the number of points included in the point cloud:

- `POSITION`: `Float64Array`, length $3N$ (X, Y, Z per point)
- `COLOR_0`: `Uint8Array`, length $4N$ (RGBA)
- `intensity`: `Uint16Array`, length N
- `classification`: `Uint8Array`, length N

Additional metadata such as bounding box coordinates, point count, scale, and offset are also retrieved. The use of 64-bit floats preserves original precision before rendering.

An example of loader output data can be seen in Listing 2. Loader-specific data is included. The example does not contain all data that the loader returns, only the relevant parts. The most important, however, is the `attributes` attribute. The pre-computed bounding box and coordinate extremas can also be used to save some processing time.

Conversion into Three.js Buffers

Three.js uses `BufferGeometry` to store point attributes efficiently. However, WebGL shaders only support 32-bit single-precision floats (`Float32`). Thus, when `Float64Array` position

```
attributes:
    COLOR_0: {value: Uint8Array(22448652), size: 4}
    POSITION: {value: Float64Array(16836489), size: 3}
    classification: {value: Uint8Array(5612163), size: 1}
    intensity: {value: Uint16Array(5612163), size: 1}
header:
    boundingBox:
        0: [-562732.513, -1169758.8360000001, 242.37]
        1: [-562621.731, -1169652.252, 250.141]
    vertexCount: 5612163
loader: "las"
loaderData:
    isCompressed: undefined
    maxs: [-562621.731, -1169652.252, 250.141]
    mins: [-562732.513, -1169758.8360000001, 242.37]
    offset: [-0, -0, -0]
    pointsCount: 5612163
    pointsFormatId: 3
    pointsOffset: 333
    pointsStructSize: 34
    scale: [0.001, 0.001, 0.001]
    totalRead: 5612163
    totalToRead: 5612163
    versionAsString: undefined
mode: 0
schema:
    metadata:
        0: {"las_pointsOffset" => "333"}
        1: {"las_pointsFormatId" => "3"}
        2: {"las_pointsStructSize" => "34"}
        3: {"las_pointsCount" => "5612163"}
        4: {"las_scale" => "[0.001,0.001,0.001]"}
        5: {"las_offset" => "[0,0,0]"}
        6: {"las_maxs" => "[-562621.731,-1169652.252,250.141]"}
        7: {"las_mins" => "[-562732.513,-1169758.8360000001,242.37]"}
        8: {"las_totalToRead" => "5612163"}
        9: {"las_pointsFortotalReadmatId" => "5612163"}
topology: "point-list"
```

Listing 2: Example of intermediary data obtained through parsing an input file via LASLoader.

data is transferred to the GPU via a `Float32BufferAttribute`, the data is implicitly down-cast. Large coordinate magnitudes lead to quantization errors because, while IEEE-754 single-precision floats always use the same number of bits for the mantissa, their fixed relative precision means that the distance between representable values increases with magnitude—resulting in reduced positional accuracy for large values [67].

IEEE-754 floating point numbers offer higher relative precision near zero, but as absolute magnitude increases, the spacing between representable values also increases. When coordinates are in the order of millions, slight differences may round off entirely, causing visual artifacts, such as points aligning into lines or jittering, when rendered as 32-bit floats.

To mitigate precision loss, point coordinates are translated so that the centroid of the point cloud becomes the origin $(0, 0, 0)$. Formally, if the original positions are $P_i \in \mathbb{R}^3$, and the centroid is

$$\bar{P} = \frac{1}{N} \sum_{i=1}^{N} P_i,$$

then normalized positions are

$$P_i' = P_i - \bar{P}.$$

This re-centring ensures most values remain near zero, where 32-bit floats have maximum precision. As WebGL represents these in `Float32BufferAttribute`, quantization artifacts are significantly reduced. The map or background layer is then translated conversely to maintain correct global positioning.

Rendering Pipeline

The data flow can be summarized as:

1. LAS file $\xrightarrow{\text{parsing}}$ typed arrays (Float64, Uint8, Uint16)
2. Typed arrays $\xrightarrow{\text{normalization}}$ BufferAttribute objects (Float32, Uint8, Uint16)
3. Buffer attributes $\xrightarrow{\text{Three.js}}$ rendered data

The data flow is additionally depicted in Figure 4.1. It's important to note that, while intensity is a 2 byte long unsigned integer, it is only used in the Three.js color attribute, which expects an unsigned integer buffer of a single byte size. Additional transformations are required to get a color buffer from the intensity buffer, which are explained further in the next subsection.

Dynamic Coloring of Point Clouds in Three.js

To support flexible visualization, the application allows users to switch between different color modes for point clouds, such as RGB color, heightmap, and intensity. This is implemented in the `PointCloud` class, which manages the Three.js geometry and its associated color buffers.

When the color mode is changed, the application updates the `color` attribute of the point cloud's geometry using the appropriate buffer. The process is as follows:

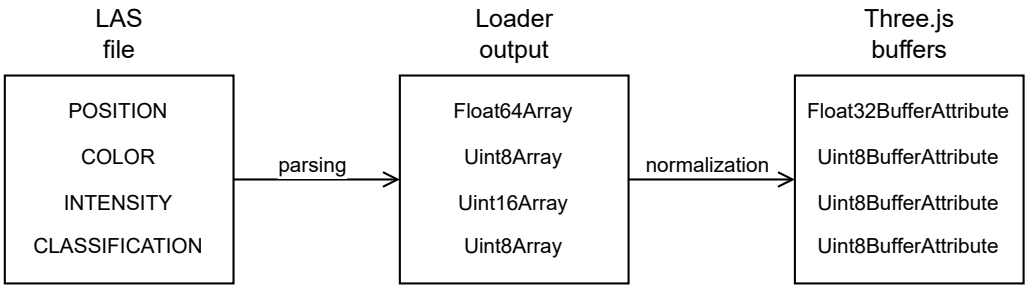


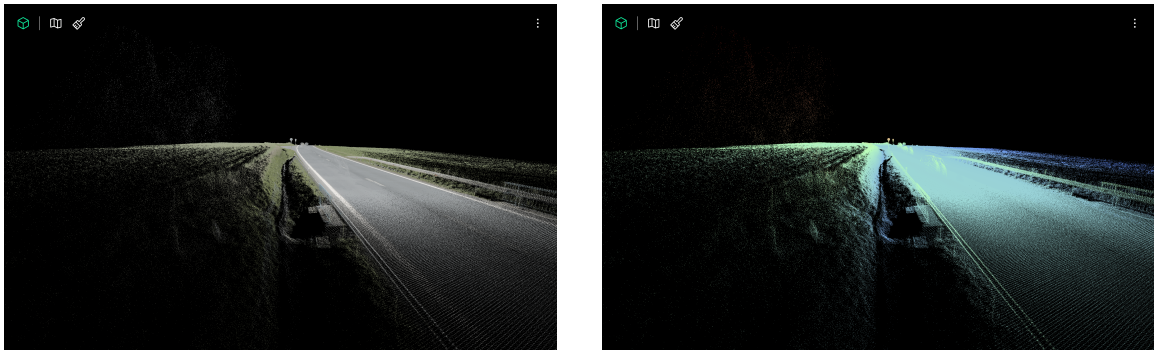
Figure 4.1: Data flow from LAS file to rendering. For the position attribute, for example, it would be: LAS → Float64 typed arrays → centroid normalization → Float32BufferAttributes → Three.js rendering.

- **Color Mode (RGB):** The original color buffer, typically loaded from the LAS file and stored as a `Uint8BufferAttribute`, is assigned directly to the geometry. This displays the point cloud in its true colors.
- **Heightmap Mode:** The application computes a color buffer based on the normalized height (Z value) of each point. The `getHeightmapColorBuffer` helper function generates a new `Uint8Array` where the color of each point is mapped according to its height, often using a gradient. This buffer is then wrapped in a `THREE.Uint8BufferAttribute` and set as the geometry's color attribute.
- **Intensity Mode:** The intensity values, originally stored as a `Uint16Array`, are normalized to the 0–255 range and mapped to grayscale colors using internal helper `getIntensityColorBuffer` function. The resulting `Uint8Array` is used to create a new color attribute, allowing users to visualize the relative intensity of each point.
- **Classification Mode:** Each point is colored by its LAS classification code (e.g., ground, vegetation, building) using a predefined lookup table `ClassificationColors`, which also contains the labeling. The helper `getClassificationColorBuffer` maps every code to an `RGBA` quadruplet in a new `Uint8Array` (alpha set to 255). This buffer is wrapped in a three.js buffer attribute and assigned to the geometry's `color` attribute. Unknown codes fall back to the "Unclassified" color. An example classification lookup table can be seen in Listing 3

This dynamic assignment is handled in the `setColorMode` method, which switches the color buffer based on the selected mode. By updating the geometry's color attribute, Three.js automatically re-renders the point cloud with the new coloring scheme, providing immediate visual feedback.

4.5 Point classification

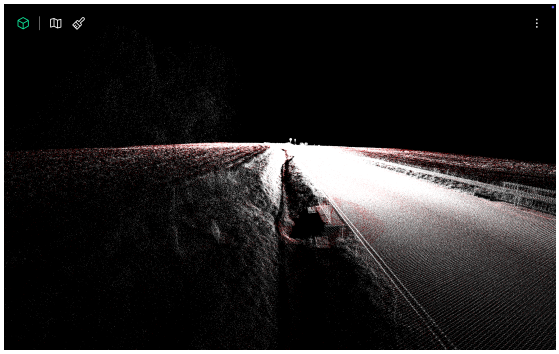
Classification is a key feature of the developed application and a requirement given by the thesis assignment. Classifying point cloud data is meant to allow users to assign semantic labels (classes) to specific points in the point cloud dataset.



Color

Heightmap

Intensity



Classification

Figure 4.2: A point cloud scene applied with different color modes.

To achieve classification, we first need to allow the user to select points. Since point clouds are rendered using `THREE.Points` rather than individual `Object3D` instances, standard selection tools such as the built-in `SelectionBox` from Three.js are insufficient. The default implementation of the Three.js selection box computes a frustum from two screen-space corners and returns only those `Object3D` instances whose bounding boxes intersect the frustum. However, the `SelectionBox` does not allow direct access to the underlying frustum planes and cannot be used for per-point classification.

To overcome this limitation, a custom wrapper around `SelectionBox` was implemented. This wrapper captures and exposes the internal frustum used during selection, allowing precise control over which points fall within the selection volume. Once the frustum is available, the next challenge is to efficiently test a large number of points (in the range of millions) against it.

To accelerate this selection process, the application builds a uniform spatial grid over each point cloud upon loading. The uniform grid is a simple spatial partitioning data structure that divides the point cloud's bounding volume into equally-sized 3D cells. Each cell contains a list of point indices falling within its bounds. During selection, only those cells whose bounding boxes intersect the frustum are considered. This drastically reduces the number of point-frustum intersection tests, especially in large datasets where most points lie outside the selection area.

An alternative approach using Bounding Volume Hierarchies (BVH) was also explored. BVH structures provide logarithmic-time spatial queries and are effective for raycasting or nearest neighbor lookups. That said, since Three.js point clouds use a flat vertex buffer without a triangle topology, as is further explained in following sections, implementing

```
export const ClassificationColors = {
  0: { name: 'Never Classified', color: [200, 200, 200] },
  1: { name: 'Unclassified', color: [255, 255, 255] },
  2: { name: 'Ground', color: [255, 0, 0] },
  3: { name: 'Low Vegetation', color: [0, 255, 0] },
  4: { name: 'Medium Vegetation', color: [0, 200, 0] },
  5: { name: 'High Vegetation', color: [0, 128, 0] },
  6: { name: 'Building', color: [0, 0, 255] },
  7: { name: 'Low Point (Noise)', color: [255, 255, 0] },
  9: { name: 'Water', color: [0, 0, 128] },
  17: { name: 'Bridge Deck', color: [128, 0, 128] },
  18: { name: 'High Noise', color: [255, 0, 255] },
  20: { name: 'Pole', color: [128, 64, 0] },
  21: { name: 'Traffic Sign', color: [255, 128, 0] },
  22: { name: 'Street Light', color: [255, 255, 128] }
};
```

Listing 3: An example classification color and label lookup table.

BVH requires generating triangle indices manually (e.g., as degenerate triangles). This led to significant slowdowns in the general rendering pipeline due to GPU-side inefficiencies, and the idea was ultimately abandoned in favor of the uniform grid approach, which offered better runtime performance and simpler integration. The application remains open to a possible octree or other spatial structure implementation. Structuring the points into an octree-based hierarchy would allow for both level-of-detail generation and point picking optimization.

The classification workflow follows these steps:

1. The user clicks the “Classify” button in the UI menu. This locks the camera controls and enables drawing of a screen-space selection box using mouse drag events.

2. Once the selection box is drawn and the mouse button is released, the application computes the corresponding frustum and tests it against the uniform grid to retrieve all points falling within the selection volume.

3. When the user clicks the “Confirm” button, they are prompted with a modal dialog to choose a classification label (e.g., ground, building, vegetation). Alternatively, a “Cancel” button is available once the selection is drawn, but before the user confirms the selection.

4. After confirming the label, the selected point indices are updated in the classification buffer. If the point cloud is currently rendered in classification mode (i.e., using classification color mapping), the color buffer is also updated to reflect the new class labels in real time.

This interactive classification system allows users to semantically annotate regions of interest in large point clouds with minimal performance overhead.

4.6 Spatial structures and rendering techniques for point clouds

Efficiently rendering large point clouds in the browser requires not only raw GPU throughput but also smart data structures to cull or level the detail of points. A straightforward baseline is to use the built-in point rendering of Three.js: load all points into a `THREE.BufferGeometry` and draw them with a single `THREE.Points` object. However, drawing millions of points without spatial filtering can quickly overwhelm the GPU, so spatial indexing (e.g., octrees) and acceleration structures (e.g. BVH) are also considered for the purposes of optimizing rendering and/or picking. In what follows, the direct Three.js method is first described, then octree and BVH-based approaches are outlined in later sections.

Direct rendering via Three.js's POINTS and BufferGeometry

In the direct approach, the entire point set is uploaded to the GPU in one geometry. As shown in the Three.js documentation, this typically involves creating a `BufferGeometry`, setting its "position" attribute via a `Float32BufferAttribute`, accompanied by the use of `PointsMaterial` to draw each point (often with vertexColors=true for per-point color) For example, the geometry can be built as follows:

```
geometry = new THREE.BufferGeometry();
geometry.setAttribute('position',
    new THREE.Float32BufferAttribute(normalizedPositionsBuffer, 3)
);
geometry.setAttribute('color',
    new THREE.Uint8BufferAttribute(colorBuffer, 4, true)
);
const material = new THREE.PointsMaterial({
    size: 0.5,
    sizeAttenuation: false,
    vertexColors: true
});
const points = new THREE.Points(geometry, material);
scene.add(points);
```

Listing 4: Example of using a `THREE.Points` object

All points are now contained in one `THREE.Points` object, which is rendered with a single draw call. This is efficient compared to creating many separate mesh objects (each would incur its own draw call). In our implementation, we disable size attenuation so that points remain a constant pixel size regardless of camera distance; this aids visibility but means distant points do not shrink naturally. Because this method has no internal spatial culling or LOD, the frame rate depends directly on the total number of points. In practice we observe that average render time increases roughly linearly as point count grows, as is charted on Figure 5.1. Even without any fancy shaders, the GPU must process every point each frame. This baseline method is simple and leverages the GPU's ability to draw sprites

in one call, but it motivates the need for hierarchical techniques when handling very large point clouds.

Octrees for spatial partitioning

An octree is defined as a tree data structure in which each internal node has exactly eight children, where a three dimensional space is created by recursively subdividing it into eight octants. To index a 3D point cloud using octree the 3D boundary is divided into eight octants, which are further subdivided recursively only when they bear point(s) within themselves until the sequence reaches a given threshold value, namely depth. The final subdivision results in eight leaf nodes that store points within their archives [38].

- **Recursive subdivision:** Starting from a root cube of side length L, each subdivision halves the side length (so at depth d, each child cube has side $L/2^d$ and volume $1/8^d$ of the root). This yields up to 8^d cells at depth d. Because each level multiplies child count by 8, the total number of potential nodes grows geometrically. In a full octree of depth d, there can be up to $\sum_{i=0}^{d} 8^i = (8^{d+1} - 1)/7$ nodes (though empty regions are often pruned).
- **Point insertion:** As the algorithm descends, it checks each point's coordinates to decide which of the 8 sub-cubes it falls into. Formally, one can compute a 3-bit index (b_x, b_y, b_z) indicating whether the point's x-, y-, z-coordinate is above the midplane of the current node; this index selects one child octant.
- **Stopping criterion:** Subdivision stops when a desired depth or point-count threshold is reached. Deeper (larger d) trees yield smaller leaf volumes (fewer points per leaf) and thus finer spatial resolution, at the cost of a taller tree. In practice, d is tuned to balance memory and search costs

Octrees neatly encode 3D space and enable spatial searches and LOD queries. For example, to find all points near a given location, one need only traverse a single path of nodes (one child per level) down to the leaf covering that location. This "one-branch" descent is efficient: each level prunes away seven of the eight subregions, so queries run in $O(d)$ time rather than examining all points. In effect, an octree is memory efficient and has high querying speed while keeping a structural simplicity. If a leaf node's boundary contains the query point, the node's point list is immediately retrieved. (In contrast, a flat array would require checking all points.) Octrees also support neighbor or range queries by recursively traversing only branches whose bounding cubes intersect the query region.

However, octrees have trade-offs. A fully static octree always allocates eight child pointers per internal node, which can waste memory when many children are empty. Advanced implementations avoid storing null children or stop subdividing empty regions. Overall, octrees work well when the point cloud is spatially coherent: one can skip large empty volumes quickly, and represent far-away areas with few nodes (low detail) while focusing detail on dense or near regions.

Octrees for point cloud LODs

In rendering massive point clouds, octrees serve as a multiresolution hierarchy. Each node of the octree can store a subset or summary of the points in its region at a certain detail

level. This naturally yields multiple LODs: higher nodes (near the root) have coarse representations, while deeper leaves contain finer detail. Rendering then traverses the octree and selects nodes by view: branches not intersecting the camera frustum are culled entirely, and distant branches are often drawn from higher (coarser) nodes only. Octree-based LOD has been widely used in point-cloud visualization. For example, the Potree point-cloud viewer builds an octree offline to manage billions of points. In Potree's scheme, every node (even internal ones) holds a subset of the original points, so that if a leaf is too fine to load, its parent's points can be used instead. During traversal, Potree maintains lists of visible nodes (to render) and visible-but-unloaded nodes (to fetch). This allows the renderer to stream points on demand. In practice, at any frame Potree traverses down from the root, stopping when either the node's projected size is below a threshold or until reaching max depth; each visited node then renders its stored points as "splats" or simple points [58, 48].

Other LOD and hierarchical schemes

Octrees are one of several multiresolution schemes. Early point-cloud renderers used different hierarchies. For example, `QSplat` builds a binary bounding-sphere tree instead of an octree. In QSplat, each inner node stores a sphere covering its children, and each leaf is a single point sample; traversal stops when the projected sphere is small [57]. Other methods use kd-trees or BSP-trees, splitting space by alternating axes, or even quadtrees (for 2D-projected data). In principle, any hierarchical subdivision (e.g. Octree, kd-tree, or layered point-cloud (LPC) tree) can provide LOD: some store original points only in leaves and compute simple aggregates (averages or bounding volumes) in inner nodes [58]. More basic LOD techniques (without full octrees) include:

- **Uniform downsampling (voxel grid):** Place a 3D grid over the cloud and take one representative per voxel (e.g. the voxel center). This is easy (used in many tools) but rigid: it ignores viewpoint, and choosing the wrong grid scale can produce aliasing or holes in sparse regions.
- **Random or regular skip:** Simply take every N-th point or a random subset. This yields some reduction, but often with uneven density. In fact, Potree's converter found that pure random sampling "introduce[s] holes in some areas and clusters in others" [58], hence its move to Poisson-disk sampling.
- **Distance-based culling:** Sort points by distance and drop far ones beyond a threshold. This is a crude LOD (fewer points far away), but does not respect spatial continuity and often looks noisy unless heavily filtered.
- **Bounding-volume hierarchies:** As noted, using spheres or boxes to summarize large regions can yield view-dependent LOD, but require building a custom tree and careful blending to avoid popping.

In summary, octrees generalize these ideas. They subdivide uniformly in 3D (like a voxel grid hierarchy), but adaptively only where points exist. Unlike fixed-grid decimation, an octree supports hierarchical culling (skip empty space) and progressive refinement. It also avoids some pitfalls of random sampling by preserving spatial locality. Many modern point-cloud tools (e.g. Open3D, PCL) include octree modules for these reasons.

On balance, the octree approach offers powerful culling and LOD but at the price of implementation complexity and preprocessing. Given its complexity and the fact that

octrees have been implemented and studied many times in full by research teams with more resources available, the application developed as a part of this thesis defers a full octree solution for now, but stays open to a possible potree integration or a custom octree implementation in the future.

Bounding Volume Hierarchies (BVH) for point clouds

A Bounding Volume Hierarchy (BVH) is a tree of nested bounding volumes that encloses a set of primitives (points, triangles, etc.). In a BVH each leaf node contains a small subset of objects wrapped in a simple volume (usually an axis-aligned box), and parent nodes enclose their children's volumes. When querying (e.g. shooting a ray), the algorithm tests the ray against a node's volume: if the ray misses the box, all contained primitives are skipped. This spatial pruning cuts down intersection tests dramatically – roughly logarithmic in the number of objects. In practice BVHs use axis-aligned bounding boxes (AABB) because they are cheap to store and quick to intersect [69].

BVHs are widely used in 3D rendering and collision detection. For example, hardware ray-tracing cores (e.g. Nvidia RTX) accelerate BVH traversal and software libraries like `three-mesh-bvh` use BVH to speed up raycasting in three.js. Without acceleration, every ray would naively test all triangles, which is far too slow for large scenes.

A key design choice is how to split nodes. A common heuristic is the Surface Area Heuristic (SAH), which chooses split planes to minimize expected overlap. The three-mesh-bvh library supports different strategies (e.g. midpoint, median or SAH) and finds SAH often yields tighter BVHs at higher build cost. In all cases, the goal is to keep sibling volumes small and non-overlapping so traversal prunes more objects.

Unlike an octree, which splits space into uniform voxels, a BVH partitions objects. Each object appears in exactly one leaf of the BVH (it is not duplicated), whereas an octree cell may straddle many objects. Thus BVHs adapt tightly to object distribution. An octree may waste effort subdividing empty space, while a BVH skips empty regions automatically because no primitives were placed there. Of course, it's important to keep in mind that these two structures are used with different purposes in mind.

Raycasting and point selection with BVH

BVHs excel at ray-geometry queries. In point clouds, we want to pick or select points under the mouse or in a region. One effective trick is to model each point as a "degenerate" triangle: clone the point-cloud geometry into a mesh whose index buffer is triplets of the same vertex (i.e. every "triangle" has zero area and a single vertex). A BVH built over this mesh effectively indexes the point set. When a ray is cast, traversal tests each bounding box: if the ray misses a box, all points inside are skipped. In leaves, each "triangle" yields one point to test.

To select points near a ray, we compute the ray's distance to each point. If O is the ray origin, d the normalized direction, and P a point, the squared distance is:

$$dist^2 = \|P - O\|^2 - ((P - O) \cdot d)^2$$

If this is below $threshold^2$ (a pixel radius), we consider P hit. Keeping track of the smallest hit distance yields the closest point along the ray.

This method is fundamentally equivalent to approximating each point as a small sphere around its position and intersection-testing the ray against those spheres. The BVH structure ensures that only points within nearby spatial partitions are tested in detail.

Beyond single-point selection, BVHs also support region selection. By representing a lasso or rectangular tool as a 2D or 3D volume, the BVH can quickly eliminate boxes completely outside the selection. Only nodes whose bounding volumes intersect the selection need to be further queried, enabling efficient multi-point classification or grouping.

Once points are selected, they can be visually highlighted, grouped into separate Three.js objects, or exported for classification, segmentation, or even server-side analysis. BVH-based selection therefore supports responsive and scalable workflows even when datasets contain millions of points.

BVHs are especially useful for tools that require precision interaction, such as lasso selection or dynamic brush tools. By dramatically pruning the search space, they enable fast, accurate selection without heavy computation.

4.7 Backend design and implementation

As mentioned in previous sections, where motivations for stack selection are explained, the backend is implemented in Python using the FastAPI framework, with MongoDB as the primary data store. It exposes a REST-like API for managing point cloud data, files, folders, and panoramas. The backend is designed for simplicity, scalability, and efficient handling of large geospatial datasets, supporting asynchronous operations and containerized deployment.

While the backend endpoint design does follow REST principles to an extent, it cannot be categorized as a fully-fledged RESTful API. The backend is not meant to expose fully accessible CRUD operations for every available domain. While some CRUD operations are available and they do follow standard REST conventions—such as using GET for retrieval, POST for creation, PUT for updates, and DELETE for removal—these are only selectively implemented where necessary. The API is designed primarily around the application's specific workflows rather than exposing generic, resource-based endpoints for all entities. The backend API is not intended to function as a standalone service. While its implementation is not tightly coupled with the frontend developed as part of this thesis, it is not designed for consumption by external clients or third-party applications. As such, data is sometimes assumed to be valid where an open client exposed API would check for errors.

Architecture and design

The backend follows a modular, service-oriented architecture. Each domain (files, point clouds, folders, panoramas) is separated into routers (API endpoints), services (business logic), and models (Pydantic schemas for validation and serialization). The application is containerized using Docker for reproducibility and ease of deployment, with both the API and the database running in separate containers as isolated services within a shared environment. An overview of the backend architecture is depicted in Figure 4.3.

Exposed endpoints

The main endpoints are grouped by resource:

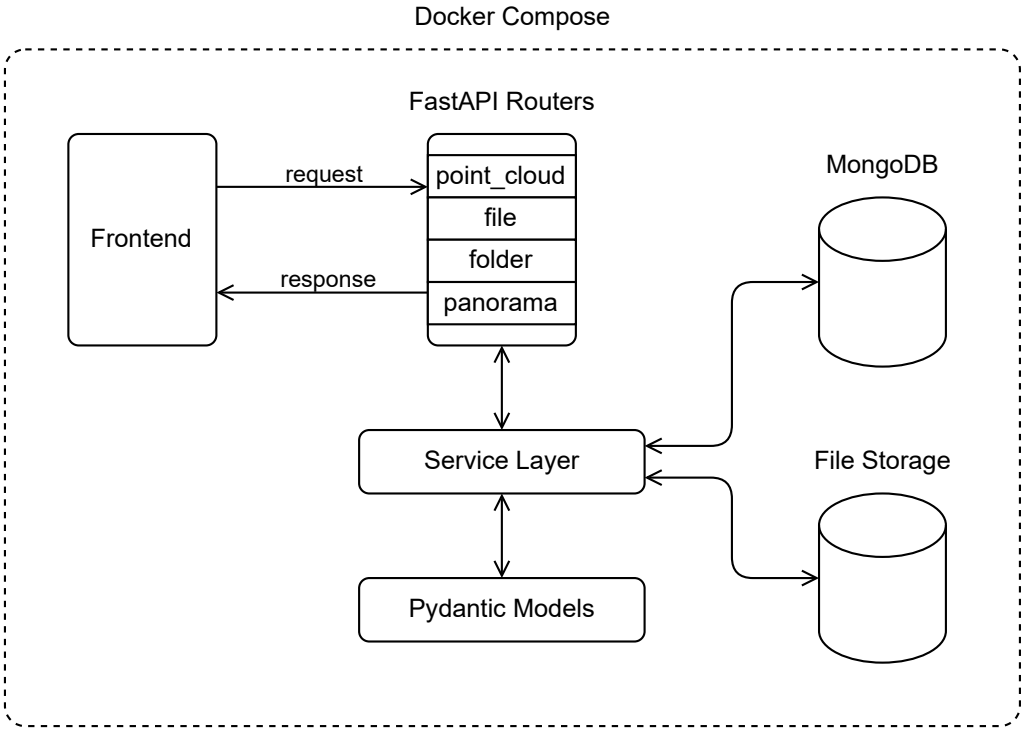


Figure 4.3: An overview of the backend architecture. Note that for diagram clarity, there is a slight inaccuracy present in the diagram. Pydantic models are also used by the routers for request validation and response serialization, but only their use in the service layer is shown.

- **Point Clouds (`/api/point-clouds`)**: CRUD operations, upload, move, and retrieval (including file / other related modalities metadata).
- **Files (`/api/files`)**: Fetch file metadata and stream file data for download.
- **Folders (`/api/folders`)**: CRUD operations, move, and a special `/browse` endpoint for hierarchical navigation.
- **Panoramas (`/api/panoramas`)**: CRUD and file association for panoramic images.

The most important endpoints are those related to point clouds, as they represent the core data for visualization and annotation. The file browser endpoint (`/api/folders/browse`) is also critical, as it provides a hierarchical view of folders and point clouds for the frontend's file browser component.

File browser endpoint

The file browser endpoint is designed to efficiently return the folder hierarchy and point cloud metadata in a single response, minimizing the number of API calls required by the frontend. This endpoint leverages service methods that recursively build the folder tree and aggregate point cloud data, including file metadata for each point cloud.

```
@router.get("/browse", response_model=FileBrowserResponse,
↪  response_model_by_alias=False)
```

Listing 5: Example of using a `THREE.Points` object

An implementation detail worth noting is the use of `response_model_by_alias=False`. This ensures that the `id` field in the response is returned as `id` (not `_id`) to match the parameters the frontend would expect while avoiding additional transformations. However, due to FastAPI's internal handling, the OpenAPI documentation (Swagger, Redoc) may still display the field as `_id`. This trade-off was made for consistency and simplicity on the frontend.

File endpoint and streaming

The file endpoint is primarily used for streaming large point cloud files to the frontend. While file metadata is typically included in point cloud responses, the actual file data is streamed separately to allow for progress tracking and efficient downloads. This separation is crucial for handling large files and providing a responsive user experience.

Database model and entity relationships

MongoDB is used as a NoSQL document store, with collections for `files`, `point_clouds`, `folders`, and `panoramas`. Each collection stores documents with relevant metadata and references to related entities using ObjectIds.

Although MongoDB is a NoSQL database and does not enforce a schema, traditional Entity-Relationship Diagrams (ERDs) can be adapted to represent MongoDB collections and their relationships, including cardinality. Such a diagram of the MongoDB collections used in the application developed as a part of this thesis can be seen in Figure 4.4.

Collections:

- `files`: Stores metadata about uploaded files (filename, size, timestamps).
- `point_clouds`: Stores point cloud metadata, including references to files and folders.
- `folders`: Represents a hierarchical folder structure, supporting nesting via parent-child relationships.
- `panoramas`: Stores panorama metadata and file references.

Data validation and serialization

The backend uses Pydantic models to enable efficient data validation and serialization. This ensures that all data being exchanged between the frontend and the backend adheres to well-defined schemas, reducing how error-prone the application is while improving maintainability. Custom types (such as `PyObjectId`) are used to handle MongoDB ObjectIds transparently.

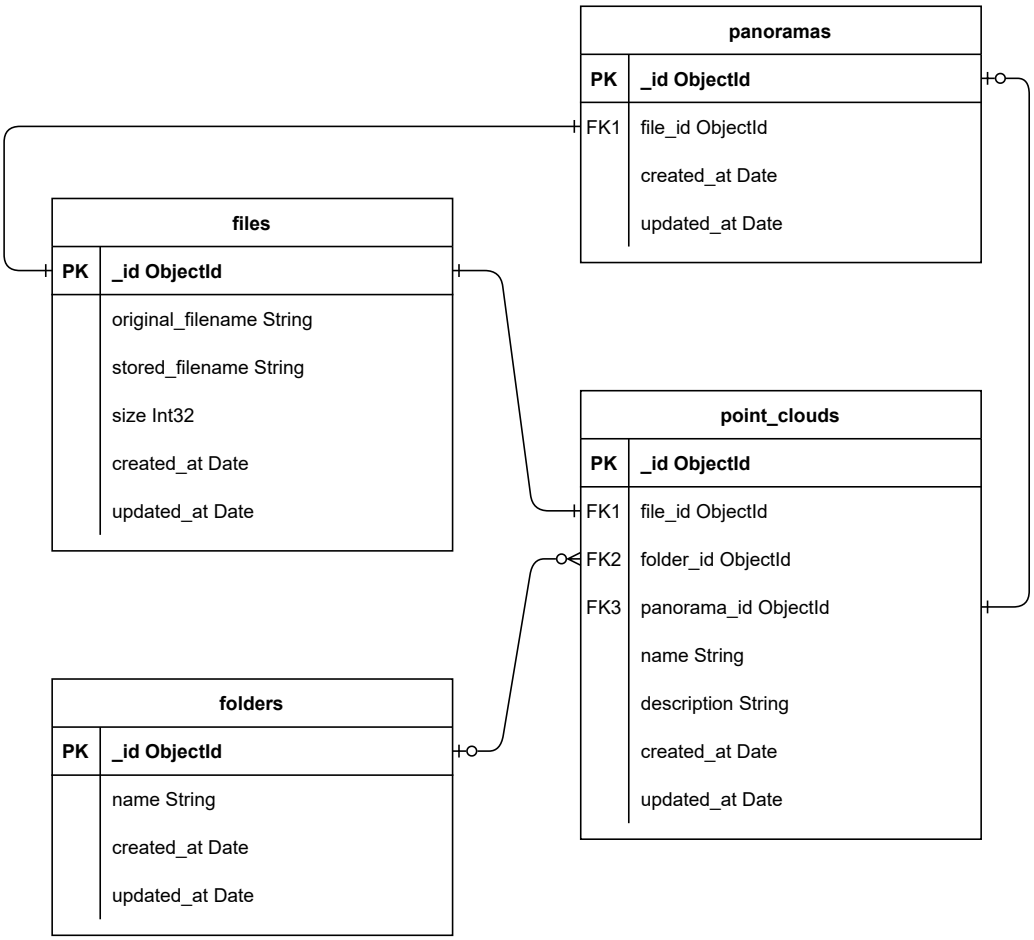


Figure 4.4: Entity-relationship diagram for the MongoDB collections.

Key design decisions

- **Asynchronous operations:** All database and file operations are asynchronous. The application leverages FastAPI and Motor for non-blocking I/O.
- **Service layer:** Business logic is encapsulated in service classes, promoting code reuse and separation of concerns.
- **Hierarchical data:** The folder and point cloud structure supports arbitrary nesting, with recursive service methods for traversal and aggregation.
- **Frontend consistency:** API responses are tailored for frontend convenience, such as returning `id` instead of `_id`.
- **File streaming:** Large files are streamed to the frontend, enabling progress tracking and efficient downloads.
- **Containerization:** Docker is used for deployment, ensuring consistent environments.

The main objective of the backend design is to be simple while remaining robust, as possible as that is. The aim is to provide essential services for the management of point cloud metadata and file data with as little fluff as possible while remaining modular and easily extendable, should the application require it. That is why FastAPI and MongoDB are the right combination.

Chapter 5

Testing and Evaluation

The application was tested through a series of user testing sessions with the aim of assessing the usability, performance, and suitability for real world point cloud annotation tasks. The goal was to obtain structured feedback from representative users, to identify usability issues, and to verify if the features that have been implemented met user expectations.

Alongside qualitative feedback, quantitative performance metrics were collected. The two performance indicators used for evaluation were the average render time per frame and the actual frames per second (FPS) which were plotted against point count. The first graph (Figure 5.1) shows the raw rendering performance in an unconstrained environment which gives information about the GPU's throughput and how render time increases with complexity. The second graph (Figure 5.2) shows FPS under VSync limitations which indicates the point at which the application can no longer maintain interactive frame rates.

Following is a summary of the user testing methodology, key findings and possible improvements based on direct user interaction.

5.1 Objectives

The aim of user testing was to assess the usability, performance, and practicality of the developed web application for annotating point cloud data and their management. The testing was focused on evaluating how well users could navigate the application UI and the point cloud scene, interpret the data and perform given labeling tasks.

The specific objectives of the testing phase were:

- To identify usability issues and bottlenecks in the annotation workflow.
- To evaluate the satisfaction of the users with the interface and visualization tools.
- To collect feedback regarding the effectiveness of features such as selection tools, point annotation workflows, and dataset management via the integrated file browser.
- To gather the suggestions for possible future improvements.

5.2 Methodology

The test was carried out with a small group of five participants with technical background and levels of experience ranging from minimal to advanced. Each user had to complete a set of tasks using the application in its current state. These tasks included loading and

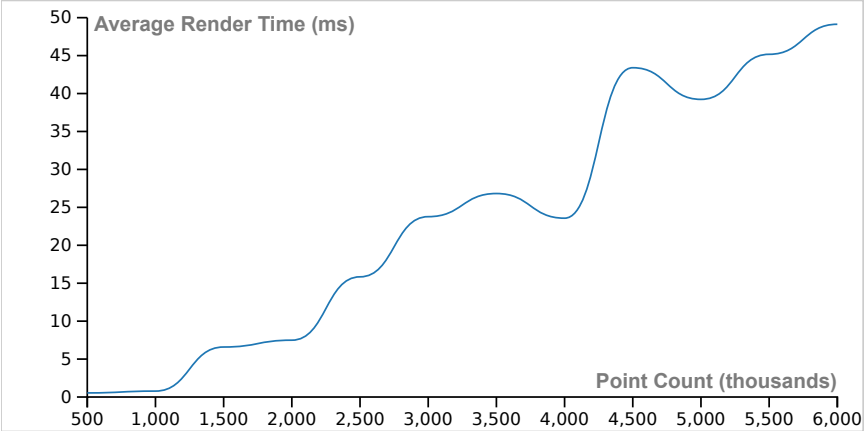


Figure 5.1: Raw rendering performance (unbounded) – render time vs point count

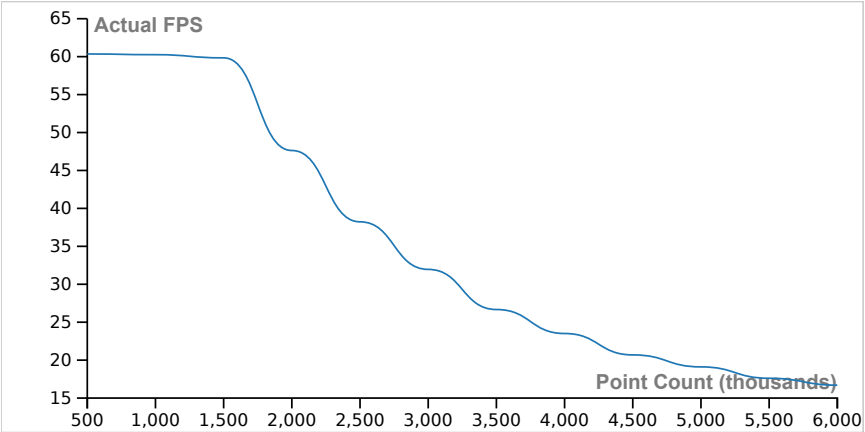


Figure 5.2: User experience performance (VSync limited) – FPS vs point count

exploring point cloud files, annotating specific object categories (for example, traffic signs and vegetation), moving between panoramic imagery and map layer views, and organizing datasets through the file browser interface.

The following testing methods were used:

- **Think-aloud protocol** — Participants were urged to verbalize their thoughts and comment on the interface while interacting with it.
- **Observation** — User behavior was observed to detect points of confusion, hesitation, or inefficient interaction patterns.
- **Post-test questionnaire** — Users completed a brief survey rating various aspects of the system, including ease of use, clarity of controls, and overall satisfaction.

5.3 Results and feedback

Positive Feedback

In general, the users were satisfied with the application interface and its capabilities. The following aspects were particularly appreciated:

- The user interface was clear and easy to understand and the users were able to navigate the scene and access relevant tools with minimal effort.
- Useful integration of supplementary data such as panoramic imagery and map overlays, which improved spatial orientation and annotation accuracy.
- Users reported that switching between the different views was intuitive even for first time users.
- The annotation tools and interaction mechanisms were considered effective and easy to understand by the users.
- Dataset loading and file management through the built-in browser was easy and worked reliably.

Identified Issues

Although the overall usability of the application was satisfactory, some areas that could be improved were identified:

- Some interface elements did not have tooltips or clear labels, leading to uncertainty about the functionality they represent.
- Difficulty selecting some specifically located points was noted.
- The file browser, while functional, could be improved by additional filtering or sorting options, especially when dealing with larger collections of datasets.
- While the performance was generally deemed acceptable for the moderately sized datasets, the users noticed a significant reduction in frame rate when working with very large point clouds, affecting the interaction smoothness.

Suggested Improvements

Based on direct user feedback and observed interaction patterns, the following improvements are recommended:

- Add descriptive tooltips and inline guidance for key buttons and interface components.
- Implement more sophisticated selection tools and allow the users a choice of which to use for the given task.
- The file browser needs to add several key features: sorting options and metadata previews together with recent files filtering and a search feature.
- The performance and responsiveness when working with large point clouds can be improved by implementing a level-of-detail (LOD) system, such as octree-based culling or simplified point skipping.

5.4 Summary

The user testing sessions validated that the application provides a solid foundation for point cloud annotation and dataset management. Users, in general, were able to successfully finish essential tasks without requiring much support and expressed satisfaction with the system's layout.

However, the user feedback also revealed multiple usability issues and areas that require refinement, mainly concerning performance and visual clarity. While not critical, addressing these issues will substantially enhance both new user experience and the efficiency of annotation work. The gathered insights will direct future development of the tool toward becoming a more polished, intuitive and, user-friendly platform.

Chapter 6

Conclusion

The main goal of this thesis was to develop an application capable of rendering, annotating, and managing point cloud data directly in the browser. Although a number of desktop-based applications exist to support similar annotation workflows, at the time this thesis was begun, there was no web-based application available that met all of the outlined requirements. This presented a clear gap in the available tooling and a motivating need for such an application to exist.

Over the course of development and writing, new solutions began to emerge, such as CesiumJS and its accompanying Cesium Ion cloud platform, which, at least partially, addressed the original requirements. However, these applications are developed by dedicated teams of professionals or researchers and naturally have a much broader scope than what can be achieved within the constraints of a thesis project.

Even so, there remains no existing web-based solution that combines all of the following: visualization of point cloud data along with their accompanying 360-degree panoramic imagery captured during scanning, annotation capabilities tied to specific points, spatial contextualization via a map plane rendered in 3D space, and the ability to manage and group datasets on server-side storage. The application developed as part of this thesis attempts to fill that specific niche.

To reach the final result, several key steps were followed. First, the fundamentals of point cloud data were explored from both a theoretical and practical perspective. An overview of existing desktop and browser-based solutions was then provided to help frame the current landscape. The thesis continued by reviewing the tools available for developing a new solution, including their respective strengths and limitations. This was followed by the design and implementation of the application itself, with in-depth discussion of selected technologies, components, and trade-offs, along with notes on viable alternatives.

All major parts of the system were addressed in detail, from the frontend, which carries most of the application's logic and user-facing functionality, to the backend and database, which handle file storage and dataset management. The resulting application was tested through user evaluation, and several conclusions were drawn based on the feedback.

While the application performs well in most practical use cases, it struggles with very large datasets, where rendering performance (in terms of frame rate) begins to degrade. A natural next step for improving this would be the implementation of a level-of-detail (LOD) structure, such as an octree or a simpler hierarchical decimation strategy, which would help maintain interactive performance even at high point counts. It should be noted that a basic decimation strategy was attempted during development to improve performance by reducing the number of rendered points based on framerate. However, the naive approach

led to drastically higher frame times in un-culled scenes and was ultimately abandoned. Nonetheless, this approach could be revisited or more refined techniques like LOD structures could be explored in future work with potentially good results. Though not implemented in this thesis, LOD systems for point cloud data are already well-documented in other research and represent a well-understood path forward.

In conclusion, this thesis presents a functional, browser-based application that addresses a clearly defined gap in the current set of available tools. It serves as both a working solution and a foundation for future development, where further performance improvements, usability refinements, and feature extensions can be gradually added.

Bibliography

[1] *Cesium* [online]. Available at: https://cesium.com/.

[2] *Cesium Point Cloud Generator* [online]. Available at: https://github.com/CesiumGS/cesium-point-cloud-generator.

[3] *CloudCompare - How to Create Contours/DEM/DSM/DTM Using RESEPI* [online]. Available at: https://lidarpayload.com/docs/tutorials-and-supporting-documents/dtm-dsm-cloud-compare/.

[4] *CloudCompare Plugins* [online]. Available at: https://www.cloudcompare.org/doc/wiki/index.php/Plugins.

[5] *Coenradie Uses AI-Driven Point Cloud Classification with Leica Cyclone 3DR* [online]. Available at: https://leica-geosystems.com/products/laser-scanners/software/leica-cyclone/leica-cyclone-3dr/coenradie-uses-cyclone-3dr-ai-classification.

[6] *Deck.gl PointCloudLayer Example* [online]. Available at: https://deck.gl/examples/point-cloud-layer.

[7] *ITowns* [online]. Available at: https://www.itowns-project.org/.

[8] *Laszip.js* [online]. Available at: https://github.com/verma/laszip.js.

[9] *Leaflet* [online]. Available at: https://leafletjs.com/.

[10] *Leica Cyclone* [online]. Available at: https://leica-geosystems.com/products/laser-scanners/software/leica-cyclone.

[11] *Leica Cyclone II TOPO: Map Creation from Laser Scan Data* [online]. Available at: https://www.geoweeknews.com/news/leica-cyclone-ii-topo-map-creation-from-laser-scan-data.

[12] *Pannellum* [online]. Available at: https://pannellum.org/.

[13] *Point Clouds – Cesium* [online]. Available at: https://cesium.com/learn/3d-tiling/ion-tile-point-clouds/.

[14] *PointCloudLayer | Deck.gl* [online]. Available at: https://deck.gl/docs/api-reference/layers/point-cloud-layer.

[15] *Proj4js* [online]. Available at: https://github.com/proj4js/proj4js.

[16] *Rendering Point Cloud in CesiumJS* [online]. Available at: https://spatial-dev.guru/2020/08/25/rendering-point-cloud-in-cesium-js/.

[17] *ScanX Analyzes Point Clouds in the Cloud with Cesium* [online]. Available at: https://cesium.com/blog/2020/scanx-point-clouds/.

[18] *Supported Features* [online]. Available at: https://www.itowns-project.org/doc/supported_features.html.

[19] *Three.js* [online]. Available at: https://threejs.org/.

[20] *Views and Projections | Deck.gl* [online]. Available at: https://deck.gl/docs/developer-guide/views.

[21] *WebGL* [online]. Available at: https://www.khronos.org/webgl/.

[22] ASPRS. *LAS Specification 1.4 - R15* [online]. Available at: https://www.asprs.org/wp-content/uploads/2019/07/LAS_1_4_r15.pdf.

[23] Autodesk, Inc.. *Autodesk ReCap* [online]. Available at: https://www.autodesk.com/products/recap/overview.

[24] Cesium. *Cesium Ion* [online]. Available at: https://cesium.com/platform/cesium-ion/.

[25] Chen, Z., Zeng, W., Yang, Z., Yu, L., Fu, C.-W. et al. LassoNet: Deep lasso-selection of 3D point clouds. *IEEE Transactions on Visualization and Computer Graphics*. IEEE. 2019, vol. 26, no. 1, p. 195–204.

[26] Curless, B. From range scans to 3D models. New York, NY, USA: Association for Computing Machinery. nov 1999, vol. 33, no. 4, p. 38–41. DOI: 10.1145/345370.345399. ISSN 0097-8930. Available at: https://doi.org/10.1145/345370.345399.

[27] Daneshmand, M., Helmi, A., Avots, E., Noroozi, F., Alisinanoglu, F. et al. 3D Scanning: A Comprehensive Survey. *CoRR*. 2018, abs/1801.08863. Available at: http://arxiv.org/abs/1801.08863.

[28] Doggett, S. *What Are Point Clouds, And How Are They Used?* [online]. Available at: https://www.dronegenuity.com/point-clouds/.

[29] Ebrahim, M. A.-B. 3D laser scanners' techniques overview. *Int J Sci Res*. 2015, vol. 4, no. 10, p. 323–331.

[30] Evans, A., Romeo, M., Bahrehmand, A., Agenjo, J. and Blat, J. 3D graphics on the web: A survey. *Computers & Graphics*. 2014, vol. 41, p. 43–61. DOI: https://doi.org/10.1016/j.cag.2014.02.002. ISSN 0097-8493. Available at: https://www.sciencedirect.com/science/article/pii/S0097849314000260.

[31] Fagerman, J. Colorizing Lidar Point Clouds. dec 2023.

[32] FARO Technologies Inc.. *FARO Scene* [online]. Available at: https://www.faro.com/en/Products/Software/SCENE-Software.

[33] Felt. *Felt* [online]. Available at: https://felt.com/.

[34] Fernandez, J., Singhania, A., Caceres, J., Slatton, K., Starek, M. et al. An overview of lidar point cloud processing software. *GEM Center Report No. Rep_2007-12-001, University of Florida*. 2007, vol. 27.

[35] Ferraz, A., Bretar, F., Jacquemoud, S. and Gonçalves, G. The Role of Lidar Systems in Fuel Mapping. august 2009.

[36] Girardeau Montaut, D. *CloudCompare* [online]. Available at: https://www.danielgm.net/cc/.

[37] GIS Cloud. *GIS Cloud* [online]. Available at: https://www.giscloud.com/.

[38] Han, S. Towards Efficient Implementation of an Octree for a Large 3D Point Cloud. *Sensors*. 2018, vol. 18, no. 12. DOI: 10.3390/s18124398. ISSN 1424-8220. Available at: https://www.mdpi.com/1424-8220/18/12/4398.

[39] Hoog, J. de, Ahmed, A. N., Anwar, A., Latré, S. and Hellinckx, P. Quality-Aware Compression of Point Clouds with Google Draco. In: Barolli, L., ed. *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer International Publishing, 2022, p. 227–236.

[40] Horswell, J. Recording. In: Siegel, J. A., Saukko, P. J. and Houck, M. M., ed. *Encyclopedia of Forensic Sciences (Second Edition)*. Second Editionth ed. Academic Press, 2013, p. 368–371. DOI: https://doi.org/10.1016/B978-0-12-382165-2.00207-5. ISBN 978-0-12-382166-9. Available at: https://www.sciencedirect.com/science/article/pii/B9780123821652002075.

[41] Isenburg, M. LASzip: lossless compression of LiDAR data. *Photogrammetric Engineering and Remote Sensing*. february 2013, vol. 79. DOI: 10.14358/PERS.79.2.209.

[42] Kim, P., Chen, J. and Cho, Y. Autonomous Mobile Robot Localization and Mapping for Unknown Construction Environments. In: *Construction Research Congress 2018*. April 2018. DOI: 10.1061/9780784481264.015.

[43] Li, L., Sung, M., Dubrovina, A., Yi, L. and Guibas, L. J. Supervised fitting of geometric primitives to 3d point clouds. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, p. 2652–2660.

[44] Li, R., Li, X., Heng, P.-A. and Fu, C.-W. Pointaugment: an auto-augmentation framework for point cloud classification. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, p. 6378–6387.

[45] LoC. *LAS (LASer) File Format, Version 1.4* [online]. Available at: https://www.loc.gov/preservation/digital/formats/fdd/fdd000418.shtml.

[46] Luo, S. and Hu, W. Diffusion Probabilistic Models for 3D Point Cloud Generation. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2021, p. 2837–2845.

[47] Maksymova, I., Steger, C. and Druml, N. Review of LiDAR Sensor Data Acquisition and Compression for Automotive Applications. *Proceedings*. 2018, vol. 2, no. 13. DOI: 10.3390/proceedings2130852. ISSN 2504-3900. Available at: https://www.mdpi.com/2504-3900/2/13/852.

[48] Martinez Rubi, O., Verhoeven, S., Meersbergen, M. van, Schütz, M., Oosterom, P. et al. Taming the beast: Free and open-source massive point cloud web visualization. In:. November 2015. DOI: 10.13140/RG.2.1.1731.4326/1.

[49] Mubunga, K. *What is photogrammetry* [online]. 5. may 2022. Available at: https://www.artec3d.com/learning-center/what-is-photogrammetry.

[50] OpenJS Foundation. *Deck.gl* [online]. Available at: https://deck.gl/.

[51] OpenJS Foundation. *Loaders.gl Documentation* [online]. Available at: https://loaders.gl/docs.

[52] Pix4D. *Pix4D* [online]. Available at: https://www.pix4d.com/.

[53] Podest, E. *The Fundamentals of LiDAR* [online]. 16. march 2021. Available at: https://appliedsciences.nasa.gov/sites/default/files/2021-03/SIF_LIDAR_Podest_Final.pdf.

[54] Pointly GmbH. *Manual Classification on Point Clouds* [online]. Available at: https://pointly.ai/manual-classification-on-point-clouds/.

[55] Pointly GmbH. *Pointly* [online]. Available at: https://pointly.ai/.

[56] Remondino, F., Toschi, I. and Orlandini, S. Mobyle Mapping Systems: recenti sviluppi e caso applicativo. *GEOmedia*. 2015, vol. 19, no. 4. Available at: https://ojs.mediageo.it/index.php/GEOmedia/article/view/1231.

[57] Rusinkiewicz, S. and Levoy, M. QSplat: A Multiresolution Point Rendering System for Large Meshes. *Proceedings of SIGGRAPH*. october 2001, vol. 2000. DOI: 10.1145/344779.344940.

[58] Schütz, M. *Potree: Rendering Large Point Clouds in Web Browsers*. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2016. Master's thesis. Institute of Computer Graphics and Algorithms, Vienna University of Technology. Available at: https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/.

[59] Shehata, O. *Dynamically Annotating 3D Tiles in CesiumJS* [online]. 5. november 2018. Available at: https://cesium.com/blog/2018/11/05/dynamic-3d-tiles-annotations/.

[60] Sims Waterhouse, D. Camera-based close-range coordinate metrology. july 2020.

[61] Stefano, F. D., Chiappini, S., Gorreja, A., Balestra, M. and Pierdicca, R. Mobile 3D scan LiDAR: a literature review. *Geomatics, Natural Hazards and Risk*. Taylor and Francis. 2021, vol. 12, no. 1, p. 2387–2429. DOI: 10.1080/19475705.2021.1964617.

[62] Stock, K. and Guesgen, H. Chapter 10 - Geospatial Reasoning With Open Data. In: Layton, R. and Watters, P. A., ed. *Automating Open Source Intelligence*. Boston: Syngress, 2016, p. 171–204. DOI: https://doi.org/10.1016/B978-0-12-802916-9.00010-5. ISBN 978-0-12-802916-9. Available at: https://www.sciencedirect.com/science/article/pii/B9780128029169000105.

[63] Thomson, C. Common 3D point cloud file formats and solving interoperability issues. 2018.

[64] Tóth, T. and Živčák, J. A Comparison of the Outputs of 3D Scanners. *Procedia Engineering*. 2014, vol. 69, p. 393–401. DOI: https://doi.org/10.1016/j.proeng.2014.03.004. ISSN 1877-7058. 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013. Available at: https://www.sciencedirect.com/science/article/pii/S1877705814002501.

[65] University of Minnesota RSL. *3D Point Clouds* [online]. Available at: http://rsl02.cfans.umn.edu/mission/.

[66] USA EMS. *Types of 3D Scanners and 3D Scanning Technologies* [online]. Available at: https://www.ems-usa.com/tech-papers/3D%20Scanning%20Technologies%20.pdf.

[67] WebGLFundamentals. *WebGL Precision Issues* [online]. Available at: https://webglfundamentals.org/webgl/lessons/webgl-precision-issues.html.

[68] Wikipedia. *3D scanning* [online]. Available at: https://en.wikipedia.org/wiki/3D_scanning.

[69] Wikipedia. *Bounding volume hierarchy* [online]. Available at: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy.

[70] Wikipedia. *PLY (file format)* [online]. Available at: https://en.wikipedia.org/wiki/PLY_(file_format).

[71] Wikipedia. *Point cloud* [online]. Available at: https://en.wikipedia.org/wiki/Point_cloud.

[72] Zatout, C. *Introduction to Point Cloud Processing* [online]. 13. september 2022. Available at: https://betterprogramming.pub/introduction-to-point-cloud-processing-dbda9b167534.