UCLouvain

epl

École polytechnique de Louvain

Automated Forest Inventory Using the iPad Pro LiDAR Scanner

Authors: **Christopher CASTEL, Martin D'HOEDT**
Supervisors: **Olivier BONAVENTURE, Sébastien JODOGNE**
Reader: **Jean GILLAIN**
Academic year 2021–2022
Master [120] in Computer Science

Acknowledgements

First and foremost, we would like to express our deepest gratitude to our two advisors, Olivier Bonaventure and Sébastien Jodogne, for their support, guidance, feedback, and proofreading. Without them, it would not have been possible for us to go this far.

We would also like to thank several members of UCLouvain. We would like to thank Thibaut Thyrion, forest ranger at UCLouvain, for sharing his knowledge and expertise. We are grateful to Aurore François for pointing us in the direction of appropriate data archiving methods. We are also pleased to have Jean Gillain as a reader of our thesis.

Lastly, we are appreciative of the research team of the geography department of UGent, in particular Cornelis Stal, Lars De Sloover and Jeffrey Verbeurgt, who provided us with their TLS point cloud data and general guidance. We are also grateful to Christoph Gollob, researcher at the University of Natural Resources and Life Sciences of Vienna, for sharing his LiDAR iPad data to help us get started.

Contents

1 Introduction **1**
1.1 Contributions and Thesis Overview 2

2 State of the Art **3**
2.1 Airborne Acquisition 3
2.2 Terrestrial Acquisition 4
2.3 Mobile Acquisition 4
2.4 Low Cost Acquisition 5
2.5 Summary 5

3 Forest Inventory Attributes **6**
3.1 Direct Attributes 7
3.2 Derived & Computed Attributes 10
3.3 IPad LiDAR Limitations 11
3.4 Summary 13

4 Material and Methods **14**
4.1 Instrumentation 14
4.2 Operational Context 18
4.3 Summary 19

5 LiDAR Point Clouds Processing **20**
5.1 High-Level Workflow 20
5.2 File Formats 22
5.2.1 ASCII-based Formats 22
5.2.2 Binary Formats 24
5.3 Coordinate Reference Systems 27
5.3.1 CRS Introduction for Mobile Devices 28
5.3.2 Geodetic CRS & GPS 29

5.3.3 Projected CRS & Web-mapping 29
5.3.4 Local CRS 30
5.3.5 Local and Projected CRS Comparison for Tree Measurement . . 30
5.3.6 Summary 32
5.4 Workflow and Algorithms 33
5.4.1 Rotation 33
5.4.2 Reprojection 34
5.4.3 Normalization 35
5.4.4 Resampling 36
5.4.5 Classification 36
5.4.6 DBH Extraction 38
5.4.7 Plot Area Calculation 39
5.4.8 Derived Attributes Calculation 40
5.4.9 Point Cloud Alignment 40
5.5 Summary 42

6 Implementation and Internals 43
6.1 Client-Server Overview 43
6.2 Backend Design and Architecture 44
6.2.1 Architecture Overview 45
6.2.2 Tools & Libraries for Forest Inventory 46
6.2.3 Technological Agnosticity 47
6.2.4 Application Programming Interface 49
6.2.5 Data Storage 50
6.2.6 Components 53
6.2.7 Custom Layer 57
6.2.8 Summary 60
6.3 Frontend Design and Architecture 61
6.3.1 Overview 61
6.3.2 Structure 66
6.3.3 Local Database 69
6.3.4 Concrete Flow Example 70
6.3.5 Improvements 75
6.3.6 Summary 76

7 Final Evaluation and Discussion 77
7.1 DBH Extraction 78

7.2 Performance & Walking Path 79
7.3 Summary 80

8 Conclusions and Future Work **81**
8.1 Final Words 82

Bibliography **83**

Chapter 1

Introduction

Traditionally, forest inventories are performed manually using calipers or tapes to acquire single tree and complete stand information. This is both laborious and time-consuming in practice. Nonetheless, such manual techniques are still commonly used despite the availability of automatic methods. This is due to the high upfront cost of contemporary laser scanner technology, as well as the partial nature of data collection with technologies such as photogrammetry that aims to estimate forest metrics from images alone.

In recent years, many efforts have been made to release low-cost solutions that produce acceptable results. In 2020, Apple released the new iPad/iPhone Pro equipped with a LiDAR (Light Detection and Ranging) sensor. The LiDAR technology can measure the distance to an object with a light pulse, which can be used to create 3D point clouds — i.e. a collection of points along with their x, y, z coordinates in 3D space. Numerous studies [1, 2] have shown that forest inventory using such LiDAR data is generally feasible and achieves accurate and precise measurements.

However, most researchers use the iPad to collect LiDAR data and then process it in a custom workflow that includes a series of tools, programming languages, algorithms, and applications. This is tedious and not accessible to non-programmers. As a result, various studies [2, 3] have determined that an end-to-end user-friendly application needs to be developed and that novel workflows should be examined. This is the starting point of this thesis. Our most significant contribution is the development of a completely automated client-server application based on a workflow for processing iPad-generated point clouds.

1.1 Contributions and Thesis Overview

Chapter 2 gives an overview of the state-of-the-art automatic methods used to conduct a forest inventory.

Chapter 3 defines the different concepts, methods, and attributes used in a forest inventory. It gives an overview of the traditional methods, such as measuring tree diameters with a measuring tape. The chapter also highlights the attributes that can be retrieved via the iPad LiDAR and those that can be derived, as opposed to those that are beyond the reach of the iPad LiDAR.

Chapter 4 documents the instruments and the data used throughout this thesis. It also provides an evaluation of the GPS accuracy of the iPad and the optimal settings to use for the scanner application.

Chapter 5 presents the workflow we designed to extract forest attributes from a point cloud produced by the iPad LiDAR. The chapter provides a comparison of several file formats for storing point clouds. It also evaluates which coordinate systems to use to geolocate trees from a geo-referenced point cloud. Finally, the chapter compares and assesses several algorithms and their optimal parameters.

Chapter 6 focuses on the implementation of the solution. The chapter is divided into two main parts: the backend and the client. The backend section describes how we designed a flexible, maintainable, and testable server that exposes a total of 32 endpoints. It also describes the architecture of the SQL database and the inner workings of a so-called "custom layer" whose purpose is to accommodate different workflows. The second section describes the design of a scalable and testable iOS application with user-friendly interfaces. This section also pays particular attention to the design of a local database with a synchronization mechanism.

Chapter 7 presents and discusses further empirical results of the solution as a whole. These results include a 100% tree detection rate with a root-mean-square error of 3.72 cm for the estimation of tree diameters at the height of 1.50 m. The chapter also shows that traditional inventory time is halved when using the application.

Finally, Chapter 8 concludes the work and outlines potential future improvements.

Chapter 2

State of the Art

Traditionally, forest inventory has been carried out manually using calipers and diameter tapes. There is currently a need to implement a faster, simpler, and automated workflow. This chapter provides a quick summary of the various scanning technologies available to suit such workflows, ranging from the most costly to the most affordable options.

2.1 Airborne Acquisition

Airborne laser scanner (ALS) solutions are usually mounted on a helicopter or an airplane. Thanks to the thorough information the ALS method provides, it is widely recognized as a valuable source of data for forest inventories that allows for the identification of vertical forest structure and, as a result, the estimation of forest inventory attributes such as height, basal area, and volume [4].

Recently, **digital aerial photogrammetry (DAP)** has emerged as a viable alternative to ALS. Not only is it more affordable, but it also provides data on other attributes such as tree species, health and maturity of the forest [5].

The most recent progress made in this area is the **unmanned aerial systems (UAS)** method, which makes use of drones to improve the cost-effectiveness and quality of aerial imaging for forestry applications [6,7]. The major disadvantage of all airborne methods is the canopy closure, which hides part of the forest.

2.2 Terrestrial Acquisition

The **terrestrial laser scanner (TLS)** method uses a static scanner to obtain forest inventory characteristics such as tree canopy volume and basal area, in addition to the more traditional parameters such as diameter at breast height (DBH) [8]. Due to the static nature of the scanner, the occlusion effect is one of the main limitations of this method: some stems, branches, and leaves are hidden by material closer to the scanner. The stands must then be scanned from multiple locations, which increases the cost of this procedure.

2.3 Mobile Acquisition

Mobile laser scanners (MLS) (aka. mobile LiDAR, VLS, vehicle-based laser scanning) are typically installed on all-terrain vehicles such as a quad. They seek to eliminate the occlusion problem that arises with TLS-based approaches as the mobility of the vehicle help cover more angles. MLS systems usually combine a laser scanner with an inertial measurement unit (IMU) and Global Navigation Satellite System (GNSS) [9]. The quality of the final data is then related to the precision and accuracy of the three components, as well as the synchronization of these components. The accuracy of MLS data is typically inferior than that of data acquired via TLS.

Depending on the forest setting, MLS may suffer from poor GNSS signal reception leading to low accuracy. Furthermore, an all-terrain vehicle may not be able to cover the entire region of interest due to the difficulty of accessing denser part of a forest. Finally, the method may have destructive effect on the environment it aims to capture, contrary to other LiDAR data collecting techniques.

The term **personal laser scanning (PLS)** refers to a scanning system wearable by human operators [9]. The concept first appeared as a backpack-style MLS system, with the scanning and positioning equipment mounted on the back of the operator rather than on a vehicle. Being more lightweight and less cumbersome than previously mentioned methods, one of the major advantages of PLS is rapid data acquisition. This method also captures a more equally distributed point cloud, and helps solve the occlusion problem even better the MLS approach.

2.4 Low Cost Acquisition

Nowadays with the ubiquity of smartphones and the wide range of sensors they feature, more affordable solutions are starting to emerge. These solutions were first based on photogrammetry technology using multiple pictures of a stand to extract forest attributes. The error (%RMSE) for the DBH ranges from 5.58% to 16.09% [10]. The main drawbacks of this method are:

- Taking pictures from enough angles is time-consuming;
- The light conditions need to be optimal to obtain the best results;
- High tree densities and occlusions by branches and shrubs can occur;
- The survey cannot be performed by an untrained user.

More recently, with the release of the iPad and iPhone pro, more affordable LiDAR-based solutions have appeared. Thanks to user-friendly interfaces and the popularity of the device, users can quickly operate the device without a lengthy training or acquiring specialist knowledge. The %RMSE for the DBH ranges from 7% to 13.03% [1]. Despite these promising results, to our knowledge, as of August 2021, no end-to-end fully automated workflow exists in the literature.

2.5 Summary

This chapter mentioned state-of-the-art methods for forest inventory along with the various forest attributes that can be collected using them. The solution discussed throughout this document is centered around the iPad-based approach.

Chapter 3

Forest Inventory Attributes

Forests may appear to be immutable and static structures that grow slowly and change little, giving the impression that they are not complicated to monitor or that it is not necessary. However, this is far from being the truth. A forest is a living system in constant evolution that is at the heart of issues such as climate change, which motivates effective forest monitoring.

For a forester, forest measurements and inventories are fundamental to answering these questions and making informed decisions. Whenever possible, forest attributes must be quantified and localized, as one cannot effectively manage something that is not measured. It is also highly important to keep a forest inventory as each decision has its own set of consequences: it is crucial to assess their impacts.

Furthermore, maintaining an inventory of forest attributes is vital in this time of global warming for two main reasons. First, global warming is increasingly wreaking havoc on forests around the world and there is a need to quantify the resulting carbon emissions [11]. Second, forests are under scrutiny for their potential as carbon sinks to offset these rising emissions [12]. Numerous studies are investigating the measurement of forests in order to derive metrics related to carbon emissions [13].

Some of the questions that forest inventory can help to address are as follows [14]:

- How much is the timber and land worth?
- What is its value as a carbon sink?
- Which wildlife species can and should be introduced? [15]

Many important forest attributes can be extracted and computed from LiDAR data. Just a few basic direct measurements at the plot[1] level are required. These baseline attributes can be used to determine almost all other attributes [16].

This section explains the set of valuable variables that can be measured directly at the plot level (Section 3.1) or derived/compiled from plot measurements (Section 3.2). Since the iPad Pro LiDAR is limited in terms of capture angles and capture range, some attributes such as the height of a tree cannot be measured; these attributes will be discussed in the Section 3.3 to provide insight into the limitations of the technology.

3.1 Direct Attributes

This section details the attributes that can directly be extracted from a point cloud generated by the iPad Pro scanner. Some focus is placed on how the device can alleviate manual work by making it easier or more consistent across the plots.

Diameter at Breast Height

Diameter at breast height (DBH; meters) is the most basic dendrometric measurement and the most fundamental: it serves as the basis from which most other plot measurements are derived. It is the diameter of a standing tree at a standardized height which varies from country to country. DBH is usually measured at either 1.3 m (United States), 1.4 m (Australia), or 1.5 m (Belgium) [17]. In general, only trees with DBH above a threshold of 5-10 cm are measured [18]. Forest inventory algorithms and prediction models are mostly focused on this metric due to its importance.

Manual measurement usually involves the use of a tape measure or a caliper. During our experiments, we used a tape as shown in Figure 3.1 for its consistency across repeated measures. There are a variety of methods available, including measuring the tree on its upward slope side and ensuring that the bare ground is used as a reference if there is snow or leaves. These methods need to be consistent across different plots, which can be problematic when multiple humans (some non-experts) are tasked with measuring trees.

[1] A plot is the piece of forest being inventoried. The terms "plot" and "stand" are used interchangeably in this document.

The use of an iPad and specific algorithms allows for consistent measurements. There is also no need to seek the assistance of an expert to educate or check that the procedures are followed. All essential procedures and standards are managed by our application and the algorithms we use. More details can be found in Chapter 6.

Figure 3.1: In-field DBH measurement with a measuring tape

Plot Area

The area (ha) of the plot is an important variable because it allows the calculation of the tree density. When a land such as an olive grove is regulated, the surface area has been precisely measured and is known in advance. This is the case for a maintained plot in a forest; for example, the stands form a circle of 360m^2 in the Bois de Lauzelle.

However, it is possible that the area is not known in advance and must be estimated. If done manually, this estimate can be very rough, for example by measuring only a part of the plot and extrapolating. It is possible to get a reasonable estimate of the surface area with the help of georeferenced point clouds.

Stem Number

The number of stems represents the total number of trees measured in a plot. Foresters use prisms or angle gauges to determine which trees should not be measured based on an estimated diameter.

Our application allows for faster and easier data capture and is able to determine which trees to count and record through diameter estimation. This eliminates the need to manually assess which trees to measure by automating and simplifying most human operations.

Tree Identification & Geolocation

Manual measurements require that trees be uniquely identified (at least at the plot level) to know which measurements correspond to which tree. Measured trees are also marked so that they are not measured twice in the same period. For example, the tree in Figure 3.1 has the unique identifier 15 and is marked by the horizontal white line indicating that it has been measured.

This thesis shows that it is generally possible to identify a tree based on GPS geolocation provided by a low-cost mobile device. This greatly decreases the amount of required manual effort and allows for automatic tracking of a tree.

Stem Diameters

Diameters correspond to any diameter at any height of interest. It is a necessary measurement for estimating the volume of a tree.

In practice, foresters estimate diameter as a function of height using the taper equations[2]. These equations are difficult to apply because they are complex, species-specific, even site-specific and age-specific [19].

The iPad LiDAR can measure diameters up to the maximum height that the sensor can reach. Taper equations are no longer required if the full tree can be captured. If this is not the case and only part of the diameters are known, they remain useful to improve

[2] Taper equations are a set of linear equations that describe the height-diameter relationship of a given species.

the accuracy of the equations. However, due to their complexity and specificity, the application does not attempt to estimate unreachable diameters with taper equations.

Status & Species

The status indicates whether a tree is dead or alive. This thesis does not attempt to make predictions about the status and species of a tree; instead, we let the user enter this information. However, some data, such as height, can be estimated from the species using a species-specific DBH-height regression model [20,21].

3.2 Derived & Computed Attributes

This section covers the additional attributes that we derive from the direct attributes. They are equally significant since they provide more straightforward decision-making indicators. The derivation techniques must be specified because they can differ for the same attributes, i.e. the same attribute name can refer to two distinct metrics or may be expressed in another unit system.

Basal Area

Basal area (BA) is one of the most important and common terms in forestry because it provides a lot of information. It indicates the density of the forest but is also related to the growth and volume of the plots.

However, the term "basal area" can be interpreted in two different ways[3]:

- The cross-sectional area of a single tree at breast height, expressed in square meters. The basal area of a tree is estimated from its DBH using the circle formula:

$$Basal\ area\ of\ a\ single\ tree\ (m^2) = \pi * (DBH/2)^2 \tag{3.1}$$

[3] Both interpretations are included in the application.

- The sum of cross-sectional area at breast height of all trees in a plot expressed per unit area.

$$Basal\ area\ of\ a\ plot\ (m^2/ha) = \frac{\sum(\pi * (DBH/2)^2)}{area} \tag{3.2}$$

Although the second interpretation is more relevant, the application handles both. Note that the DBH is supposed to be expressed in meters and the area in hectares in these formulas.

Density, Mean Distance, Mean DBH

The density as the number of stems per hectare, the average distance between trees, and the average DBH are not as informative as the basal area, but they nevertheless provide practical insights.

Some efforts to compute them were found in the literature [22–24], therefore these attributes are also included in the application.

Other Attributes

The application does not provide all the possible attributes derivable from the DBH. For example, a basal area calculated only on the basis of trees with a certain DBH is not directly possible. However, the application is linked to a database (Section 6.2.5) on which it is possible to execute such queries and calculate some specific information.

3.3 IPad LiDAR Limitations

This section details some of the attributes that, while important, cannot be computed directly or indirectly from the iPad LiDAR point cloud. However, this is not a limitation of the iPad per se. It is possible to estimate some of these attributes using other components such as images, but to do so would require the development of a specialized scanner. As this thesis focuses on the LiDAR component, these attributes are not included in the application.

Height

Tree height (m) is the distance from the base of the tree to its top. It is a basic measure that can be used to calculate other attributes such as biomass, mean height, arithmetic mean height weighted by basal area, etc.

According to our experiments, the iPad LiDAR can measure the height up to 3 meters and 5 meters under ideal conditions.

One way to go around the LiDAR limitation is to use the iPad as an inclinometer or hypsometer. These trigonometry-based instruments determine the height of the tree as a function of its distance from the device, and the angles formed by the device and the two extremities (base and top) of the tree. Assuming that the top of the tree is visible, this would be feasible with a combination of LiDAR (distance to the tree) and device tilt (angles) information.

Volume

Volume, usually timber volume or merchantable volume, is the stem volume per hectare (m^3/ha). Since diameters cannot be measured above a certain height and taper equations are too specific to be used, it is not yet possible to estimate volume.

Above-Ground Biomass

Above-Ground Biomass (AGB; Mg/ha) of a tree is its total vegetation mass (stem, leaves, branches) above ground. It is measured using species-specific allometric equations [25] — e.g. the relationship between stem mass and leaf mass for a given species. These equations are known to be inaccurate [16].

Crown measurements

Tree crown[4] size is directly related to the growth of the tree and understorey vegetation. It is also used to determine a more precise biomass of the tree. It requires the use of aerial measuring tools such as ALS (Airbone Laser Scanning) [26] or drones [27].

[4] The crown is the branches and foliage that grow out from the trunk of the tree.

3.4 Summary

We have seen that some attributes, although interesting, cannot be extracted or calculated because of technical limitations. Table 6.5 provides a summary of the different attributes, their units, and whether or not they are available in our application.

Attribute	Type	Available	Unit	Note
DBH	Direct	**Yes**	Meter	Measured at 1.5 m.
Plot area	Direct	**Yes**	Square meter	-
Stem number	Direct	**Yes**	-	-
Geolocalisation	Direct	**Yes**	EPSG3812 EPSG3857[5]	-
Diameters	Direct	*Partially*	Meter	Up to a height of 2.5 m on average.
Status & species	Direct	No	-	The user can provide these information using the description field.
Tree basal area	Derived	**Yes**	Sqare meter	Measured at 1.5 m.
Plot basal area	Derived	**Yes**	Square meter per hectare	Measured at 1.5 m.
Density	Derived	**Yes**	Number of trees per hectare	-
Mean distance	Derived	**Yes**	Meter	-
Mean DBH	Derived	**Yes**	Meter	Measured at 1.5 m.
Height	Both	No	Meter	Requires a custom scanner or precise taper equations.
Volume	Both	No	Cubic meter per hectare	Requires a full scan of the tree or precise taper equations.
Above-ground biomass	Both	No	Megagram per hectare	Requires a full scan of the tree or precise allometric equations.
Crown	Direct	No	-	Requires aerial captures.

Table 3.1: Summary of attributes (available or not) in the application

[5] Broadly speaking, those are the coordinate systems used; see Section 5.3.

Chapter 4

Material and Methods

The previous chapter covered the direct and indirect attributes that can be extracted or computed from a cloud point generated by the iPad LiDAR.

This chapter focuses on the methodology used to capture such point clouds for a given stand. It first introduces in Section 4.1 the tools we used to conduct our data collection, as well as the inherent constraints of these tools. The operational context of Section 4.2 then describes under which conditions these captures were made.

4.1 Instrumentation

As discussed in the previous chapters, our approach focuses on the iDevices that feature a LiDAR sensor. We used an iPad Pro, but the solution would work on an iPhone as well. The device also includes a GPS module, which is used to generate georeferenced point clouds. This section discusses the limitations encountered with these two components as well as with the scanning application used to capture the point clouds.

Scanning Application

Instead of developing our homemade scanner application, which would have been too time-consuming, we decided to use an application called "3D Scanner App" [28], which has already been mentioned several times in the literature [1, 2].

The application offers many different settings, as shown in Table 4.1.

Parameter	Values	Description
Max Depth	0.5 to 5m	Discards sensor data after a certain distance.
Resolution	5 to 20mm	The lower the value the higher the resolution.
Confidence	low, medium, high	Filters the data output by the sensor. Choosing HIGH, for instance, only keeps the best quality data (with the best confidence) but reduces the amount of data available.
Masking	none, object, person	Isolates significant visible objects while masking out persons or less noticeable objects.

Table 4.1: Description of the settings of the 3D Scanner App application

To find the best combination of parameters, we decided to focus our attention on a smaller tree, assuming that it would scale up with bigger specimens. The tree radius was first measured manually using a diameter tape at a height of 1.5m. The application was then used to scan the tree using different combinations of parameters.

Method	Confidence	Resolution (mm)	Radius (m)	Error (cm)
0 (Reference)	-	-	0.0557	0
1	High	5	0,114287	5,8587
2	High	10	0,1053964	4,96964
3	High	15	0,1148198	5,91198
4	High	20	0,1115848	5,58848
5	Medium	5	0,1076879	5,19879
6	Medium	10	0,1104854	5,47854
7	Medium	15	0,09177609	3,607609
8	Medium	20	0,1018625	4,61625
9	Low	5	0,1065483	5,08483
10	Low	10	0,08699632	3,129632
11	**Low**	**15**	**0,078125**	**2,2425**
12	Low	20	0,09504316	3,934316

Table 4.2: Comparison of the settings of the 3D Scanner App application for forest inventory

Table 4.2 shows that scanning with a `Resolution` and `Confidence` parameters set to `15mm` and `Low` respectively yielded the best results. These optimal values were also suggested by a previous evaluation of the 3D Scanner App for forest inventory [1].

GPS

Apple claims that the "GPS accuracy depends on the number of visible GPS satellites. Locating all visible satellites can take several minutes, with accuracy gradually increasing over time" [29]. The `Core Location` documentation also mentions that "the framework gathers data using all available components on the device, including the Wi-Fi, GPS, Bluetooth, magnetometer, barometer, and cellular hardware." [30]. These different factors make it quite difficult to determine the true accuracy of the geographical position of the device.

We assessed the accuracy of the GPS using two methods. The first focused on its relative accuracy by scanning the same tree multiple times as can be seen in Figure 4.1. Assuming a random distribution, we notice a drift of around 2m on average, which is consistent with the error range of 1 to 3 meters observed in the literature [1].

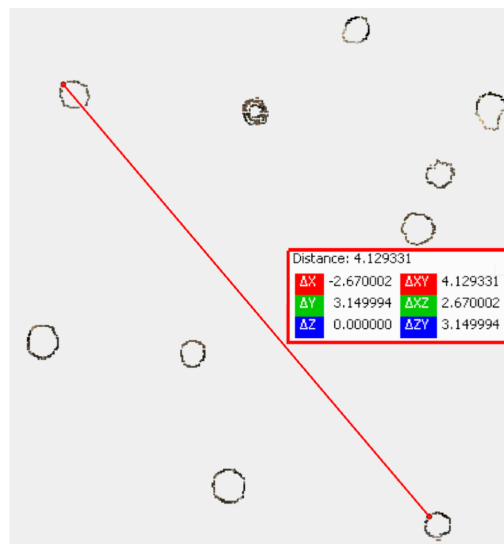


Figure 4.1: GPS drift highlight using multiple georeferenced scans of the same tree

The second method focused on the absolute accuracy of the GPS module. We first scanned a georeferenced point cloud of a known geodetic marker (see Figure 4.2) whose location is provided by the Belgian platform G-DOC [31]. Using Cloud Compare we selected the center point of the scan and extracted the geographic coordinates reported by the iPad. This value was then compared with the coordinates corresponding to the geodetic marker.

Our results show that the geodetic marker and the corresponding georeferenced point cloud are separated by 3.077 meters.

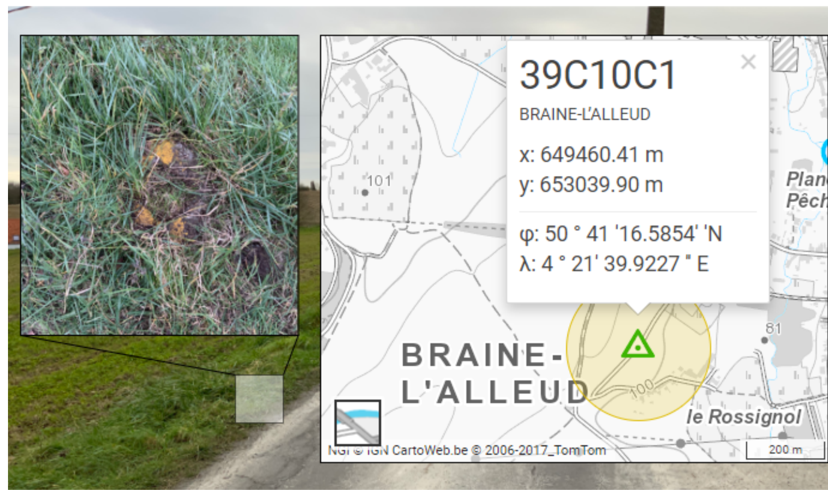


Figure 4.2: Geodetic marker

Coordinates (EPSG 3812)	X (meters)	Y (meters)
NGI Geodetic marker	649460.31	653039.86
iPad georeferenced point cloud	649462.21	653042.28

LiDAR

The sensor is capable of scanning objects up to 5 meters [32]. The accuracy of the LiDAR has not been made public by Apple. We did notice that the application would yield poor results when scanning smaller trees (radius <7.5 cm). This can be partially explained by the relatively low density of points the LiDAR projects as shown in Figure 4.3 using an infrared camera.

Figure 4.3: IR camera displaying the point density of the iPad LiDAR sensor [33]

4.2 Operational Context

The iPad was primarily used to capture tree stands located in the "Bois de Lauzelle", a 20-hectare nature reserve in the town of Ottignies-Louvain-la-Neuve, in Walloon Brabant.

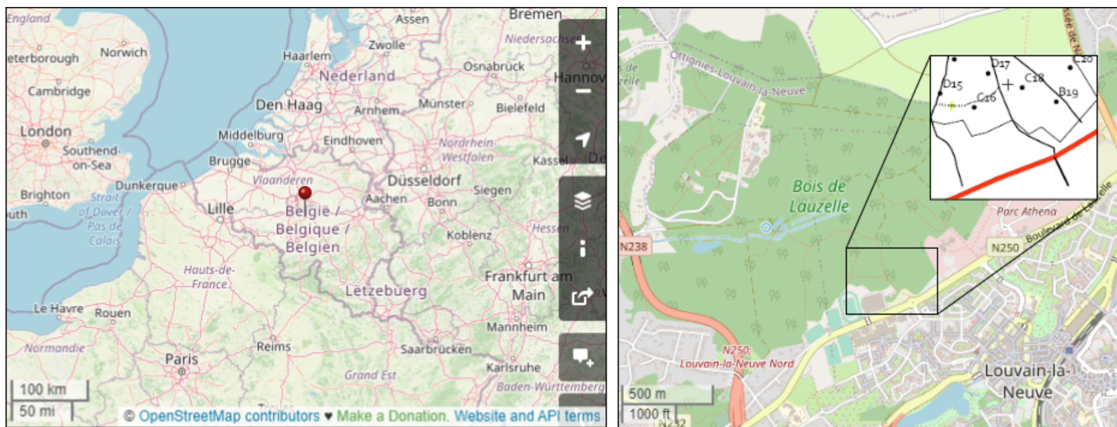


Figure 4.4: Location of the forest "Bois de Lauzelle"

This forest is divided into 149 survey plots. We focused our experimentations around the plot C16 (see Figure 4.4). A stake inserted into the ground is used to define the center of a plot, all the trees that are found within a 10 meters around this stake are part of the plot. For practical reasons, we decided not to follow the plot structure and to capture trees in a straight line (Figure 4.5).

The captures were made between mid-January and mid-April 2022. The trees (mostly beech) did not bear many leaves during this period, as can be seen in Figure 4.5

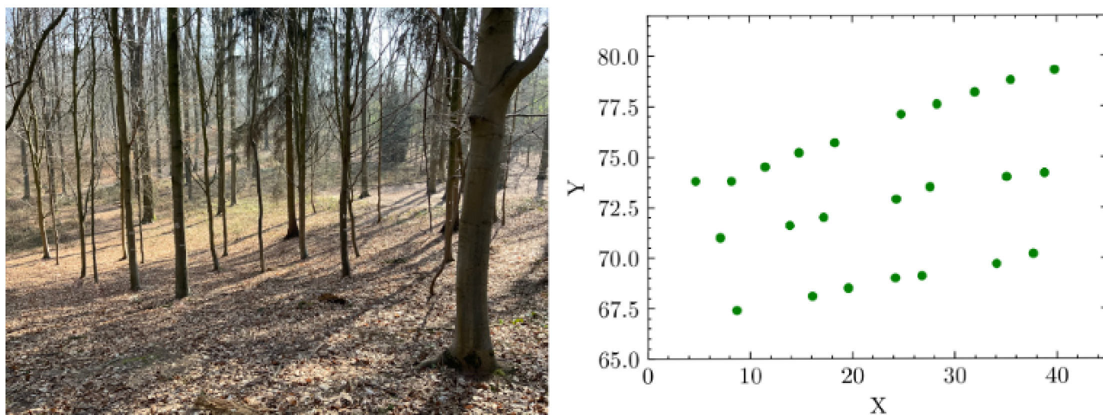


Figure 4.5: Scanned area and the relative locations of its trees (meters)

Using 3D Scanner App, we scanned trees, row by row, walking around each tree individually. We found that halting the scan or directing the LiDAR towards the ground helped avoiding the introduction of artifacts while walking from one tree to the next.

The application paints surfaces that have already been captured using virtual reality (Figure 4.6). This allows the user to avoid scanning the same tree several times, which might result in artefacts due to overlaps between previously recorded and newly recorded data.

Figure 4.6: In-field plot survey using 3D Scanner App

4.3 Summary

This chapter focused on the tools and methodologies used to capture point clouds using an iPad featuring a LiDAR sensor and GPS module. Those experiments showed great potential in providing an accurate enough solution for the survey of forest stands. The actual accuracy and feasibility of our proposed workflow are thoroughly discussed throughout the upcoming chapters.

Chapter 5

LiDAR Point Clouds Processing

In Chapter 3 we described the importance of the many attributes that can be extracted from a point cloud generated by the iPad LiDAR. This chapter explains the methodology and workflow we have used to extract the tree and plot attributes.

The workflow and its components outlined in Section 5.1 serve as the guideline of this chapter. Most of the intricacies behind the workflow — especially with regard to the choice of algorithms and their parameters — will be covered in Section 5.4

However, before diving into the algorithms, it is necessary to understand LiDAR point cloud nomenclature and operations. We therefore strongly advise reading the terminology in Chapter 1 before proceeding with this chapter.

To make the most informed choices possible, it is also essential to understand how the data is organized (Section 5.2). In particular, we will see that the points are georeferenced, which entails a variety of technical considerations in terms of geospatial standards, data, and coordinate systems. All of this is detailed in Section 5.3.

5.1 High-Level Workflow

The proposed workflow seeks to extract forest attributes from a point cloud captured by the iPad LiDAR. The series of steps used to retrieve these properties is depicted in Figure 5.1. Note that this workflow works equally well for plots with several trees and plots with only one tree.

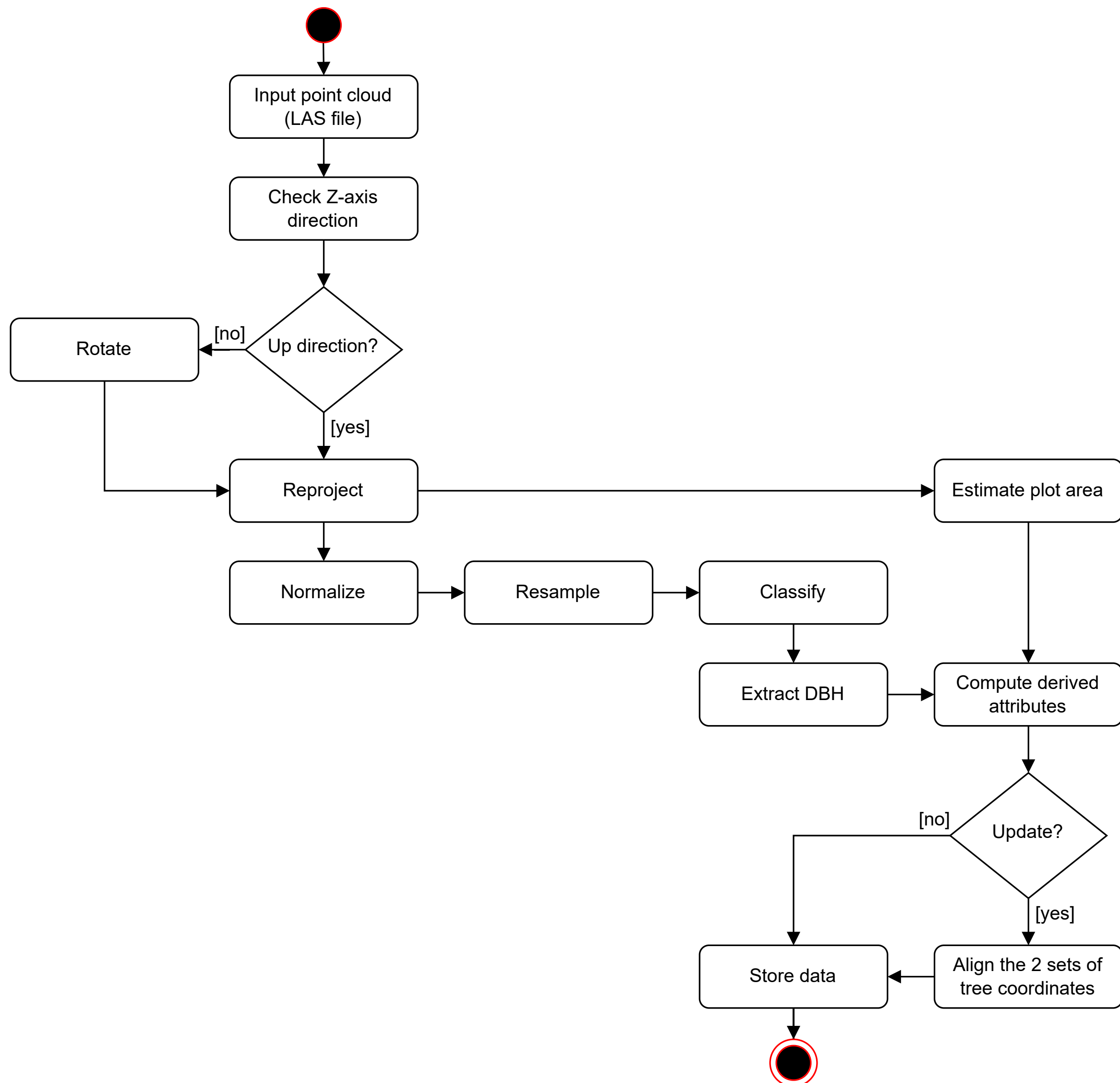


Figure 5.1: Activity Diagram for the whole data extraction workflow

Prior to data extraction, a lengthy pre-processing phase is required, which includes point cloud rotation (if necessary), reprojection and normalization, a tree extraction process, and so on. Following the extraction of direct attributes (such as DBH, area, geoposition, etc.), derived attributes (e.g. basal area, mean DBH, etc.) are computed.

As the system is intended to provide a forest monitoring, a user can upload a point cloud for two reasons. First, to create a new plot. Second, to update an existing plot with more recent data. In the second case, because the exact position of a tree is not precise (see Section 4.1), it is necessary to align the previous trees with the new information in order to keep a tree-by-tree inventory.

5.2 File Formats

There are many different 3D graphics file formats that LiDAR scanners can generate. Different scanners produce different formats, and different tools only accept a subset of these formats. For example, the iPad application we use to acquire data (3D Scanner App) can export to 8 formats, and some 3D processing software such as CloudCompare can handle up to 30 different formats.

This chapter compares some of the most common open point cloud formats. As we will see, some formats are better suited to our needs. Proprietary formats are not discussed as they are not a sound long-term approach and are not freely available [34].

5.2.1 ASCII-based Formats

The point clouds can be stored in plain text and encoded in an ASCII-based format. This type of format is usually based on a table where each row corresponds to the properties of a point or another geometric element. These properties include at least the corresponding spatial coordinates (x, y, z).

The main disadvantages of such ASCII-based formats are as follows:

- They are slower to read due to the necessary translation from a human-readable format to a binary format.
- They contain few metadata, unlike binary formats that can have additional metadata for each point.
- They are not space-efficient. For instance, the use of a format converter showed that the same point cloud is 6.23 MB in an uncompressed binary format and 10.3 MB in a simple 7-column ASCII format.

The ability to be easily edited and read by humans is frequently reported as one of their main advantages, although this is misleading because many tools can help perform these tasks on binary formats.

Their primary benefit is that they are the most future-proof as they save point clouds as a basic set of coordinates, ensuring that they can be used for years to come.

The following subsections describe the most common ASCII formats. Note that conversion from one format to another is possible but may result in loss of information.

XYZ, TXT, and ASC File Formats

XYZ, TXT, and ASC are very simple non-standardized ASCII-based formats. Because the units and specifications are not standardized, these formats are not advised for data transfer. However, they are recommended for data storage, provided that the exact format specification used is documented.

These formats are often similar to ad-hoc formats that developers design for their own needs. However, these ad-hoc formats rarely gain traction, and the lack of formal documentation typically results in misuse.

OBJ File Format

Although commercial binary variants of the OBJ format exist, the most common and free versions are in ASCII. The format is used to store normals, vertex positions, textures, colors, and other aspects of geometric objects. Companion files for rendering textures (MTL and JPG) are frequently included.

PLY File Format

PLY (Polygon File Format) files [35] represent 3D objects as a set of polygons in a mesh. The polygons are described as a collection of elements (typically vertices and faces), each containing a list of properties. The coordinates are the most fundamental attributes, but the files can also include other properties such as colors, surface normals, transparency, and textures.

This format is the most commonly used ASCII format in the LiDAR industry, but it is not as popular as the binary formats detailed in the following section due to the ASCII-induced drawbacks.

5.2.2 Binary Formats

None of the ASCII formats listed above were suitable for our needs. In addition to the general disadvantages discussed above, there are additional LiDAR-specific drawbacks that prevent these formats from being used:

- Some LiDAR-specific metadata could be lost due to the inability to include it.
- There is no way to set coordinate system and unit standards.

Binary formats usually carry more information while being more compact and faster to interpret. They can be converted into an ASCII file — with some information loss — for long-term storage.

The subsections that follow review some of the most prevalent binary formats used in the industry — the E57 and LAS formats. The standards for both formats are openly available, allowing for better interoperability, especially through the development of third-party verification tools.

E57 File Format

The E57 file format is a flexible, compact, and vendor-neutral format for storing point clouds [34]. The format is documented in the ASTM E2087 standard.

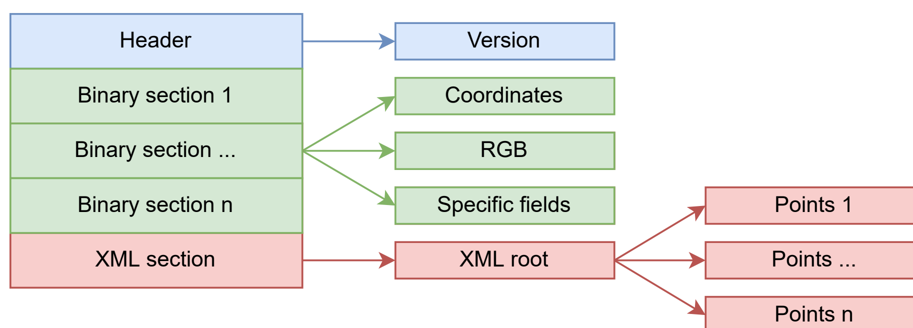


Figure 5.2: E57 simplified file format structure

It is a format that mixes ASCII and binary, where the points and images that constitute the bulk of the data are encoded in binary for efficiency. The rest, such as the metadata and headers, are encoded in ASCII for readability.

As shown in Figure 5.2, an E57 file is composed of three parts: the header, the binary sections, and the XML section.

These parts are organized as follows:

- The header contains the locations of the XML section and some information such as the version number.
- The binary sections contain the data, for example information about a point such as its coordinates, colors, etc. These attributes are flexible, and the format supports the most popular ones. If some fields are missing, the standard allows for extensions.
- The XML section (ASCII) is organized as a tree structure that contains the different elements that make up the point cloud. This section references the binary section corresponding to each element. The root element contains file-level information not listed in the header, such as the necessary information to encode the data in a standard coordinate system.

The format also allows for the storage of many point clouds in the same file, each with its own coordinate system and rotation vector, allowing for consistent alignment. Although the format is very promising, there are some flaws. The files cannot be compressed, which makes the data very large. In addition, the location of the sensor at any time is missing, and some tools do not support the format.

LAS File Format

The LAS (LASer) file format is the de facto standard in the airborne and terrestrial sensing industries. The format is specified in the ASPRS standard [36].

The overview in Figure 5.3 shows that the structure is composed of 3 parts:

- (Optional) The header contains file-level information and some generic data (version[1], bounds, number of points, file signature[2], ...).
- (Optional) The variable-length records contain user-defined data such as the projection information and the format of the point data records.

[1] There are several versions of the format, but they are backward compatible. The iPad 3D scanner app uses version 1.2.

[2] The signature is helpful to detect when the same point cloud twice is uploaded to avoid extra processing.

- (Mandatory) The point data records contain general information about each point, such as its coordinates, RGB values, and classification (ground, building, vegetation, noise, ...). It also contains information specific to airborne LiDAR, such as the number of returns for a pulse.

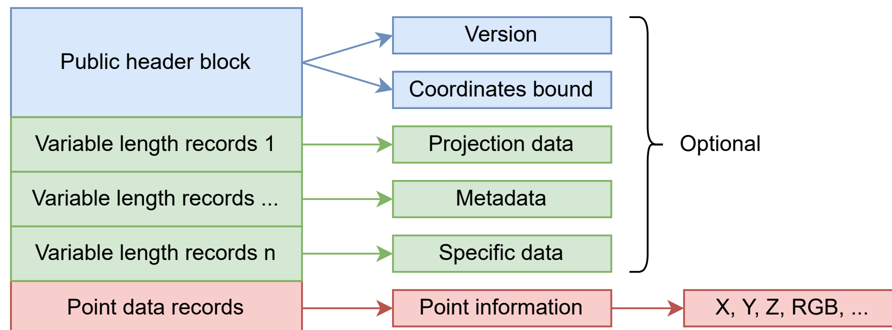


Figure 5.3: LAS 1.2 simplified file format structure

Unlike the E57 format, the LAS format is not flexible as it does not feature an extension mechanism. However, the available fields are sufficient for our purpose since they are designed to store airborne LiDAR data, which is a superset of terrestrial LiDAR data. The file can be used for terrestrial applications like the one developed in this thesis by ignoring the superfluous airborne-related information.

These are the reason we picked this format:

- LAS allows the projection information to be embedded into the header, making it possible to use the most appropriate projections without misinterpreting the point locations.
- It has a simple, well-documented standard and structure.
- It is tailored to the needs of the LiDAR community. All the necessary fields are already available.
- There are lossless compression techniques that allow the files to be compressed down to 7-25% of their original size [37].
- The LAS file format is well supported by LiDAR software and tools.

The primary drawback of this format is that it is limited to a single coordinate system. The implication of this limitation will be discussed in the next section.

5.3 Coordinate Reference Systems

In Chapter 3.1, we introduced the necessity to uniquely identify a tree based on its location. On the surface, it appears to be a simple matter of using the GPS of the device (see Chapter 4) to generate georeferenced point clouds and display the extracted location of the trees on a web-map.

However, it is not as straightforward as it appears because the process involves a variety of ways to assign coordinates to real-world locations — i.e. the coordinate reference systems (CRS).

There are three components[3] subject to coordinate systems in the context of the application developed in this thesis:

- The GPS of the device that outputs the locations of the receiver in terms of coordinates (Section 4.1).
- The survey data in the form of a LAS file that contains the coordinates of each point and associated spatial information (Section 5.2.2).
- The web-map that shows the position of the surveyed trees in a visual and user-friendly manner (Section 6.3).

It turns out that the GPS and the web-map coordinate reference systems are fundamentally different, and neither is well suited for extracting tree attributes.

This section discusses the technical issues and solutions arising from the use of a georeferenced point cloud produced by a mobile device. This is an interesting challenge because, to our knowledge, there is no end-to-end mobile solution in the forestry literature that supports georeferenced point clouds.

A brief introduction to coordinate reference systems will be given in Section 5.3.1. As it represents such a broad topic, the introduction will be centered on the requirements of a mobile device surveying application.

Then, we will discuss the technical requirements and decisions behind the use of the principal coordinate reference systems in the sections that follows. Finally, we will discuss which system is the most appropriate for measuring distances.

[3]When the word "component" appears in this section, it will refer to one of the three components listed unless otherwise noted.

5.3.1 CRS Introduction for Mobile Devices

A coordinate reference system (also known as a spatial reference system) is a coordinate-based system used to non-ambiguously reference a point on the surface of the Earth. As shown in Figure 5.4, the essence of coordinate reference systems is made up of three main elements.

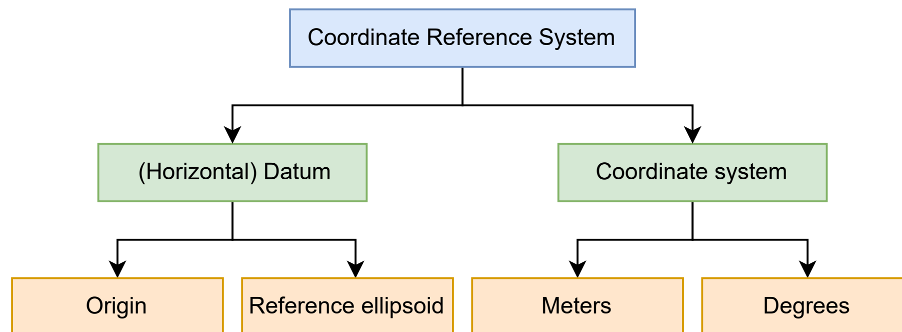


Figure 5.4: Geometric parameters of the coordinate reference systems

Each piece, or geodetic parameter, is defined as follows:

- The coordinate system[4] is an abstract mathematical grid that defines the locations of points in space on an XY(Z) axis.
- The units used to define the coordinate system.
- The horizontal datum[5] is a reference frame for matching a model of the shape of Earth to the coordinates. Note that two different datums will result in two different pairs of coordinates for the same location.

These geodetic parameters can be used to design a variety of coordinate reference systems. However, there are only a few categories, and only three of them are significant for the components mentioned in the introduction section.

The following subsections discuss the type of coordinate system to use for each application component. We will review the benefits and limitations of each category in the context of a tree survey application.

[4] The terms "coordinate reference system" and "coordinate system" are occasionally used interchangeably in the geodesic field.

[5] There is also the vertical datum which takes into account the elevation. Horizontal and vertical datums are usually included in the same standard.

5.3.2 Geodetic CRS & GPS

Geodetic (or geographic) coordinate reference systems model the Earth as a spheroid or an ellipsoid. Their angular latitude and longitude coordinates are formed by the equator and the meridian of longitude zero (i.e. the prime meridian).

The Global Geodetic System (WGS; or `EPSG:4326`[6]) is the most well-known example of a geodetic coordinate reference system. It was popularized by the advent of GPS and is still widely used nowadays. As the LiDAR scanner uses the GPS module of the device, the produced georeferenced point cloud uses this coordinate system.

Geographic coordinate systems are helpful for locating places on the planet. However, longitudes and latitudes are angles measured in degrees of an ellipsoid. Calculating distances using trigonometric relationships is possible. However, this is not advised as the spacing between the parallels (lines of latitude) is not uniform.

As a result, georeferenced point clouds produced by the iPad cannot be used as is to measure forest attributes. The solution is to reproject the point cloud into a suitable coordinate system before any data processing. The process of reprojecting a point cloud consists in changing the coordinates from one reference system to another.

5.3.3 Projected CRS & Web-mapping

Projected systems model the Earth as a flat surface and use the Cartesian coordinates (x and y) to locate a point on Earth. The flat surface is created by a map projection, which tries to (imperfectly) flatten a spherical[7] model of the planet.

The projection must be carefully chosen for the region of interest to minimize distortions. The more local the projection, the more accurate the distance measurements at that location will be. For example, the Web Mercator projection (`EPSG:3857`) used by most[8] web-maps uses meters as its unit. However, this unit is misleading because the meter becomes more squashed the further north a place is located. On the other hand, the unit is the actual meter at the equator, which means that this projection can be used to calculate distances in that area.

[6] The EPSG codes are standard codes for coordinate systems, datums, spheroids, units, and other geodetic parameters.
[7] Projected CRS are based on a geodetic CRS, hence the spherical model.
[8] Apple Map, Google Map, and Open Street Map all use this CRS.

In our application, we use `EPSG:3857` for our web-map[9] as it is the de facto standard. However, because of the deformations, this coordinate reference system cannot be used for measuring distances in Belgium. Instead, there are Belgium-specific systems such as the Belgian Lambert 2008 (`EPSG:3812`) that are worth discussing.

5.3.4 Local CRS

Local coordinate reference systems are based on a flat-Earth approximation of a small and specific area. When the iPad produces a non-georeferenced point cloud, the output coordinate reference system is local to the device. Because there is no distortion, a non-georeferenced point cloud should be used for attribute retrieval.

However, there are a few issues with this approach. First, the LAS file format does not allow for the inclusion of two systems at the same time: one has to choose between a georeferenced cloud point or a non-georeferenced cloud point. Second, in the iPad case, the local system is not associated with any known system standard: it is not possible to reproject the local coordinates.

The first solution is to use both a non-georeferenced (local) point cloud and a georeferenced (`EPSG:4326`) point cloud. The two point clouds would then be aligned using a point cloud alignment algorithm such as the iterative closest point algorithm [38]. This solution is the best for accurate measurements. However, it forces the user to provide two point clouds for the same plot, which would be inconvenient and costly.

The other solution is to use a georeferenced coordinate reference system instead. The following subsection evaluates some of the projected coordinate systems against the local coordinate system.

5.3.5 Local and Projected CRS Comparison for Tree Measurement

In Section 5.3.4, we established local coordinate reference systems as the optimum systems for measuring distances. However, this approach is not chosen as it would require the user to upload two point clouds. In Section 5.3.3, we noted that projected

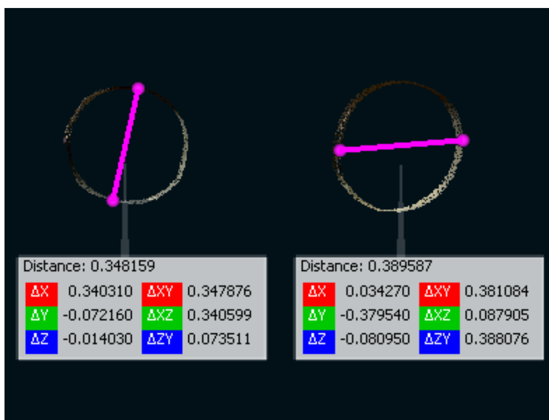
[9] Web-maps follow a special convention whereby EPSG:3857 coordinates are used (the map is flat), but EPSG:4326 coordinates are shown as if the map was a sphere.

systems might be a good solution for measuring distances, provided the chosen system is suited for the specific surveyed region.

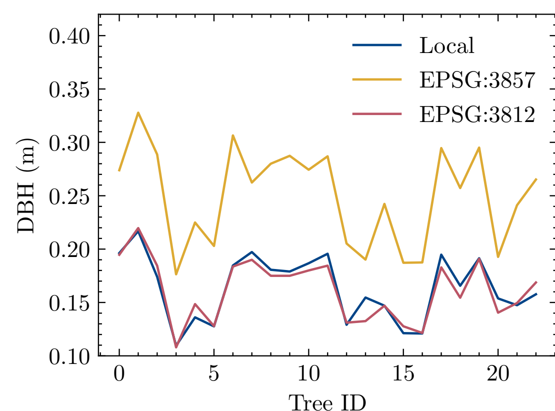
In this section, we use the DBH as the point of comparison between the projected systems and the default local system used in non-georeferenced point clouds.

The procedure we used is as follows:

1. Produce a non-georeferenced point cloud (local system) of a 23-tree plot (see Section 4.2);
2. Produce a georeferenced point cloud (geodetic system) of the same plot;
3. Reproject the georeferenced point cloud into the projected coordinate systems to be evaluated (i.e. `EPSG:3812` and `EPSG:3857`);
4. Manually estimate the DBHs from the local and projected systems using CloudCompare (point cloud processing software) and its point picking tool to avoid any algorithmic influence as shown in Figure 5.5 (a).



(a) DBH estimations with point picking tool



(b) CRS comparison DBH-wise

Figure 5.5: Comparison of the EPSG:3812 and EPSG:3857 projected coordinate reference systems against the default local reference system in Belgium

As the experiments are done in Belgium, we picked a Belgian-specific projected coordinate system known as Belgian Lambert 2008 (`EPSG:3812`). The second selected system is the Pseudo-Mercator (`EPSG:3857`) used for the web-map.

As expected, Figure 5.5 (b) shows that `EPSG:3857` results in higher values due to the distortions. The `EPSG:3812` system, on the other hand, is remarkably similar to the local system. The slight variation observed is most likely due to the fact that the point

picking tool is prone to small human errors; the algorithms discussed in the following section do not have this problem.

It is important to note that since the deformations are location-dependent, these results are only reproducible and valid in Belgium. However, we can conclude that projected systems are a good compromise to the local system, provided that the appropriate projection is chosen.

The chosen solution is to reproject the cloud point into `EPSG:3812` before extracting the data. The application has a custom layer allowing third-party developers to extend the application with another projection more appropriate to the part of the world where their survey is taking place. We encourage these developers to ensure that the used projection provides meaningful data.

5.3.6 Summary

This section tackled the interesting question of the choice of coordinate systems in the context of a mobile survey application. The coordinate reference systems taken into account in the application design are summarized in Table 5.1.

EPSG Code	Type	Component	Note
4326	Geodetic	GPS - georeferenced point cloud (LAS file)	CRS provided by default by the scanner for the georeferenced point clouds.
3857	Projected (cylindrical)	Web-map	Used to geolocate the trees. Requires a reprojection from EPSG:4326.
-	Local	Non-georeferenced point cloud	Not used.
3812	Projected (conic)	Reprojection georeferenced point cloud (LAS file)	Used for extracting tree attributes. Requires a reprojection from EPSG:4326. This CRS is Belgium-specific.

Table 5.1: Summary of the coordinate reference systems used in the application

5.4 Workflow and Algorithms

The preceding sections laid the groundwork for the processing of georeferenced point clouds generated by the iPad LiDAR by introducing the data format and coordinate reference systems:

- We understand the limitations of the LAS file format and that there is a need to implement certain parts of the process to work around some of the related issues.
- We can assume that the measurements extracted by the algorithms are in SI units, which allows the comparison with data captured in the field.

Workflows used in related studies [39–41] usually comes with some form of manual component. For example, a study may use specific software to calculate the distance between two selected points or rotate the point clouds. To completely prevent the influence of approximation methods or defects in point clouds, workflows are sometimes entirely manual [42].

In this section, we discuss the proposed (and implemented) workflow described in the overview (Section 5.1). This workflow aims to provide a fully automated end-to-end process to extract tree attributes from iPad LiDAR point clouds.

The following subsections describe the components of the workflow and the algorithmic choices that were made.

5.4.1 Rotation

Although most point cloud processing tools expect the z-axis to be in the up direction, scanning applications (e.g. 3D Scanner App) may have the y-axis pointing up instead. To fix this issue, the point cloud must be rotated by 90 degrees around its x-axis.

Transformations such as rotation, scaling, and translation can be parameterized by a 4x4[10] matrix called the transformation matrix. The point cloud is then transformed by left-multiplying each point coordinate by the matrix.

[10] Such matrices are based on homogeneous coordinates, which means 3-vectors (x, y, z) are represented as 4-vectors (x, y, z, 1) in 3D space.

A clockwise rotation of θ radians about its x-axis can then be expressed as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Transformation\ matrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.1}$$

5.4.2 Reprojection

One of the takeaways of Section 5.3 is that the point clouds must be reprojected into other coordinate systems to be usable. For instance, the formula to derive projected x and y coordinates from ellipsoidal longitude λ and latitude ϕ — e.g. from `EPSG:4326` to `EPSG:3857` — is as follows:

$$\begin{aligned} x &= Fx + a(\lambda - \lambda_0) \\ y &= Fy + a \ln\left[\tan(\pi/4 + \phi/2)\right] \end{aligned} \tag{5.2}$$

where:

λ = longitude (radians)
ϕ = latitude (radians)
Fx = false easting[11] (radians)
Fy = false northing[12] (radians)
λ_0 = longitude of natural origin (radians)
a = equator radius (meters)

The formulas required to perform coordinate transformations and conversions are fully documented under the EPSG Dataset [43]. The PROJ software [44] and its Python wrapper, which we used, implement most conversion formulas.

In the case of the forward `EPSG:3857` reprojection, all variables are equal to 0 except the longitude, the latitude, and the equator radius (a = 6378137 meters).

No loss of precision has been observed after reprojecting several times a point cloud.

[11]The origin is generally given false coordinates to avoid negative coordinates.
[12]See preceding footnote.

5.4.3 Normalization

The difference in elevation on a large plot can be significant. In order to treat the trees the same regardless of their location, it is necessary to normalize the point cloud to remove the elevation. Furthermore, some of the following algorithms require the point cloud to be normalized. The process is shown in Figure 5.6.

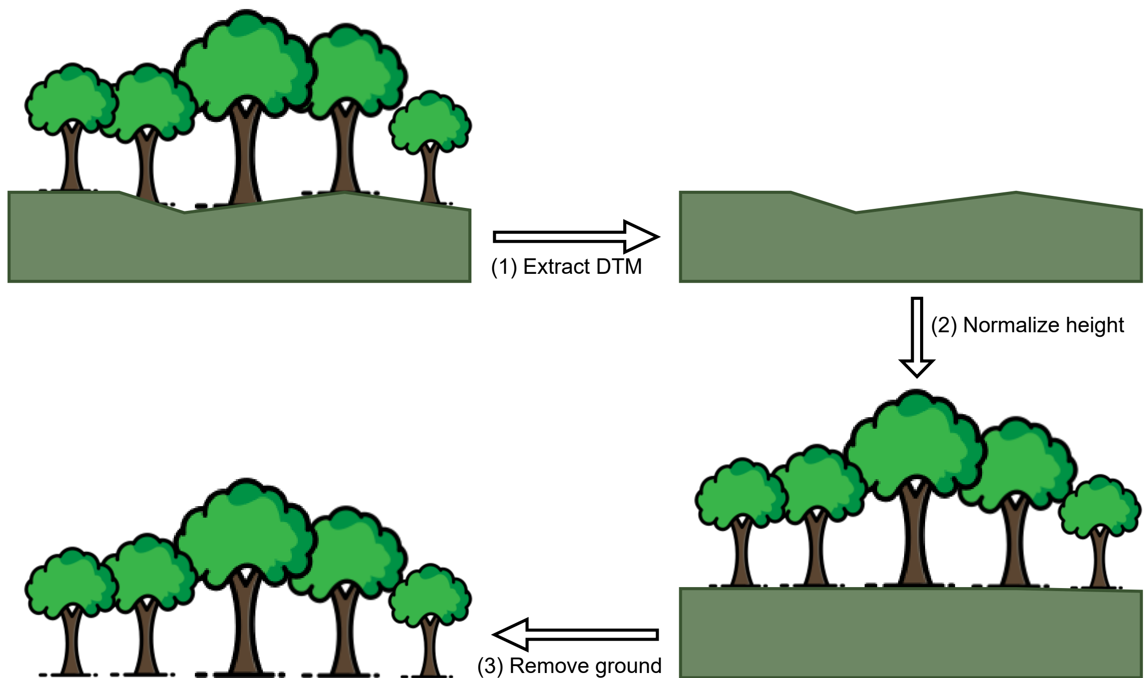


Figure 5.6: Point cloud normalization

The three steps are as follows:

1. The ground points are classified and extrapolated to create a digital terrain model[13] (DTM) using the cloth simulation filtering algorithm [45]. This algorithm simulates a piece of a rigid cloth falling onto the inverted point cloud. The final shape of the cloth is the DTM.

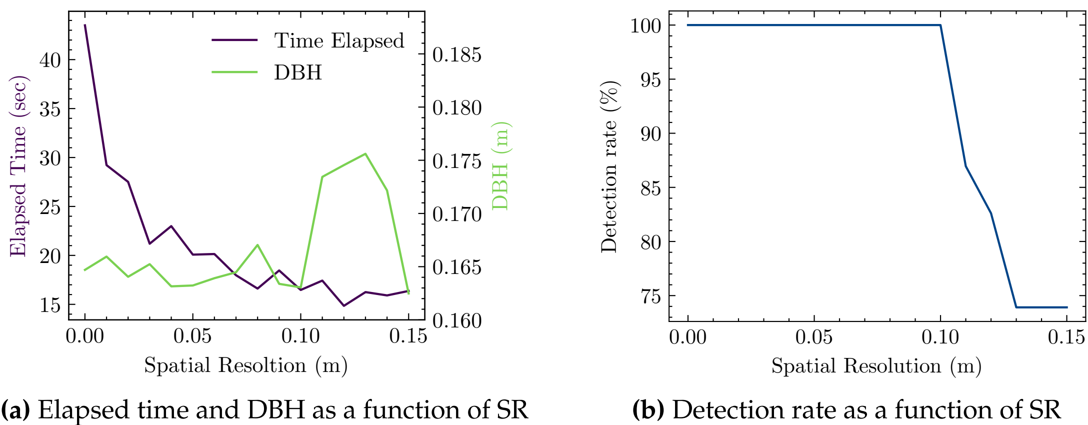
2. From the resulting DTM, we remove the elevation by setting the ground at $z = 0$ everywhere. The point cloud is normalized from this step onward.

3. The ground is removed by deleting the points with classification = 2 in the LAS file to reduce the file size and potential noise. Points that are left below the ground as artifacts of the previous methods are also filtered out.

[13]Note that the DTM is generated by external libraries.

5.4.4 Resampling

It is recommended to resample the point clouds to achieve a homogeneous point density [46]. This is especially true for iPad point clouds, as they have very heterogeneous point densities. A voxel-based sampling algorithm is applied to homogenize and thin the point clouds. The algorithm creates a voxel grid and selects one point per voxel. The voxel is the 3D equivalent of what a pixel is in a 2D raster image. Note that we define the term "pixel" as a point with x and y coordinates. The main parameter of the algorithm is the side length of a voxel — i.e. the spatial resolution.



(a) Elapsed time and DBH as a function of SR **(b)** Detection rate as a function of SR

Figure 5.7: Sampling with voxelization; spatial resolution comparison

As shown in Figure 5.7 (a), the spatial resolution has little to no impact (SD = 0.0012m) on the DBH measurement up to 0.10. After that value, the DBH does not longer remain steady and the detection rate drops. The elapsed time for the entire procedure decreases as the spatial resolution increases; nevertheless, due to the small gain after 0.08 and the possibility of missing some trees, 0.08 was chosen as the spatial resolution.

5.4.5 Classification

As of the previous step, the point cloud is fully normalized, but the tree parameters cannot yet be extracted. The point clouds are mostly composed of several trees because whole plots are usually scanned, which means that it is necessary to extract the individual trees beforehand. In addition, there may be noise due to the scanner or tree foliage, whereas algorithms that fit circles on a tree to extract its diameter work best on bare trunks.

The Hough Transform [47] is usually used to extract and identify the position of a particular shape or some of its features via a voting procedure. To extract the trees (read: several pre-selections of points), we use a circle version of the Hough transform that searches for imperfect circles with unknown radii in a layered point cloud.

The algorithm is summarized by the following steps:

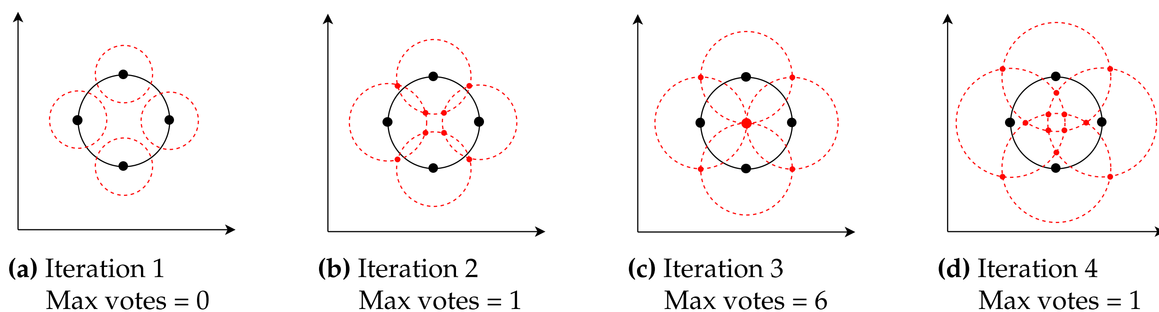
1. The point cloud is divided into several 2D layers every 0.25 meters over a chosen height interval — between 0.50 meters and 1.75 meters as the range of the LiDAR on the iPad is limited. Points outside this interval are ignored.

2. The pixels with a density[14] above a certain number (10%) are selected as the center of a parameter circle (red circles in Figure 5.8).

3. Each parameter circle is iteratively tested with a given radius increased by the pixel size at each step.

4. An accumulator counts the number of intersections (= votes) per pixel.

5. Only the pixels with at least a certain number of votes are pre-selected as possible centers of a tree at the height of the current layer.



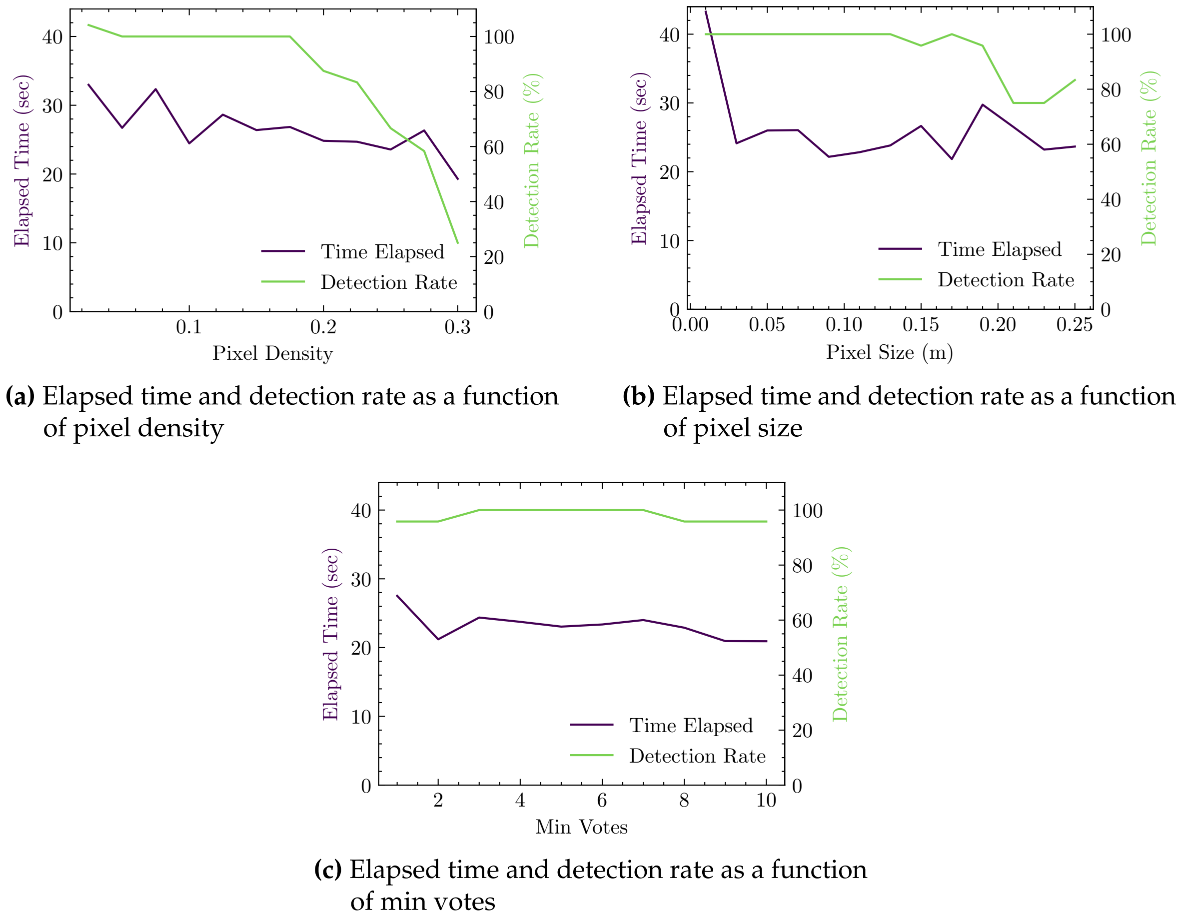
(a) Iteration 1 Max votes = 0 **(b)** Iteration 2 Max votes = 1 **(c)** Iteration 3 Max votes = 6 **(d)** Iteration 4 Max votes = 1

Figure 5.8: Four iterations of the Circle Hough Transform with increasing radius size. The true center is found at the third iteration.

Good parameterization is necessary for a high detection rate. False positives can occur when parameters are too flexible (low density), while conservative choices (high density, high pixel size) can result in omitted trees. The best parameter selection is determined by the forest type and scanner as well.

[14] The density of a pixel is the ratio between its number of points and the number of points of the most dense pixel.

After a trial-and-error phase (see Figure 5.9), we achieved a 100% detection rate with the following settings: pixel density = 0.1%, pixel size = 0.03, and min votes = 4. If false positives occur, we let the user remove them via the API or the mobile application.



(a) Elapsed time and detection rate as a function of pixel density

(b) Elapsed time and detection rate as a function of pixel size

(c) Elapsed time and detection rate as a function of min votes

Figure 5.9: Trial-and-error phase to find the right parameters for the Hough Transform

The individual regions are then extracted by cropping the point cloud by one meter around each center. A more clever way would be to partition the point cloud using a Voronoi diagram [48], but this would add unnecessary overhead in our situation as the trees in the surveyed plots are not too close to one another.

5.4.6 DBH Extraction

Based on the selected points by the Hough Transform for each extracted region, a circle-fitting variant of the RANSAC algorithm is applied. The algorithm seeks to fit a model (a circle) onto a set of points. The points that belong to the circle are the

"inliers" and those that do not are the "outliers". The algorithm iterates numerous times to find samples of points (at least 3 points) that fall within a certain proportion of inliers. Numerous studies have had good results using this method to extract diameters [39, 49].

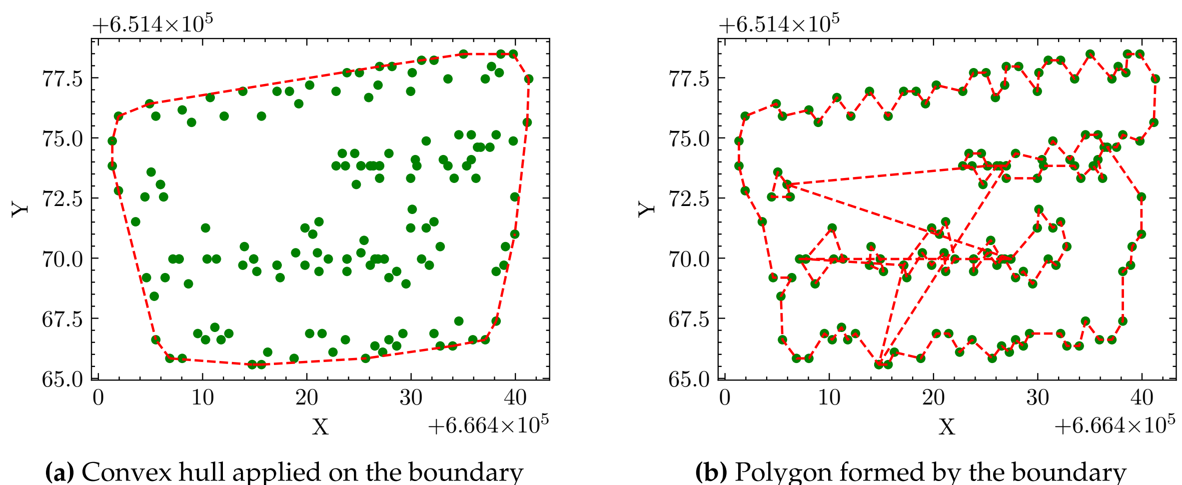
We selected the minimum fitting circle to exclude any remaining noise. The diameter of this circle is used to estimate the DBH of the tree. We chose 10 iterations because more resulted in longer processing times with no significant improvement in measurements. The results will not be presented in this section but in Chapter 7.

Other studies [47] observed that the cylinder-fitting variant provided a slightly better DBH estimate (but resulted in a slight decrease in detection rate), but this was not observed with our dataset, most likely due to smaller stem heights.

5.4.7 Plot Area Calculation

The naive method is to calculate the surface area of the plot based on the shape formed by the four most extreme coordinates. However, this is not a good solution as it would underestimate the area of the omnipresent round-shaped plots.

Instead, we use an approach based on the calculation of the area of the convex hull. This is the smallest convex polygon (i.e. each pair of points has a line segment inside the polygon) that contains each point. The convex hull is computed from the points on the boundary of the point cloud as shown in Figure 5.10 (a).



(a) Convex hull applied on the boundary **(b)** Polygon formed by the boundary

Figure 5.10: Convex hull and boundary polygon areas comparison

Another solution is to directly compute the area of the polygon formed by the boundary as shown in Figure 5.10 (b). However, this approach underestimates the area as the holes were not counted, contrary to the convex hull approach. Holes occur when the plot is not densely packed or when the ground is not constantly scanned. The main default of the convex hull is that it overestimates concave-shaped plots. A concave-oriented approach (concave hull, alpha shape) was tested but did not provide satisfactory results as it requires a visual inspection to determine the best fit.

5.4.8 Derived Attributes Calculation

The derived attributes (e.g. basal area) are computed using the methods and formulas described in Section 3.2. The preceding steps provide all necessary measurements.

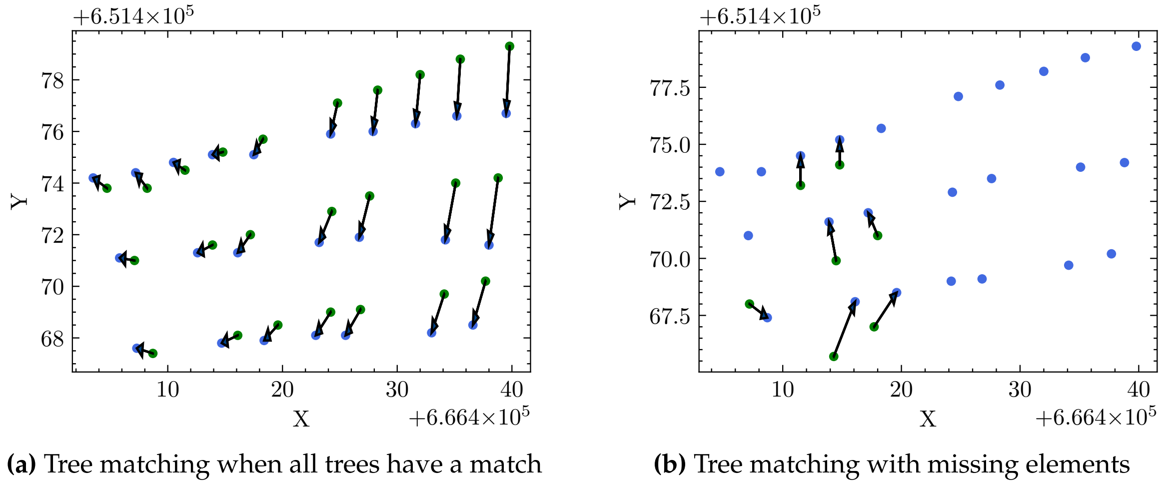
5.4.9 Point Cloud Alignment

One of the purposes of the application is to keep a forest inventory. Users can update a plot by uploading a new point cloud of the same area; the information (trees DBH, basal area, etc.) is updated consequently. However, as discussed in Section 4.1, the GPS has an error of about 2 meters[15]. To update the right trees, it is therefore necessary to solve the problem of matching a set of tree positions to another that is slightly rotated, translated, and with potentially missing elements.

The naive approach of looking for the nearest matching tree (with a 2.5-meter limit) is prone to inaccuracies because of the slight rotation. Instead, we defined this problem as the assignment problem [50] whose general formulation is to find the minimal-cost assignment of n agents to m tasks given a set indicating the cost of each possible agent-task assignment. The problem is reformulated by making the assignment-cost the distance between two tree locations. Distances over 2.5 meters are replaced by infinity, and trees with all their costs at infinity are removed. The problem is then solved using the Hungarian algorithm, which has an $O(n^3)$ time complexity [51].

We evaluated the Hungarian algorithm in two ways. The algorithm was first evaluated on real data collected in the Bois de Lauzelle. We scanned the plot three times (two full scans and one partial scan) on three different days to ensure that the coordinates changed over the scans.

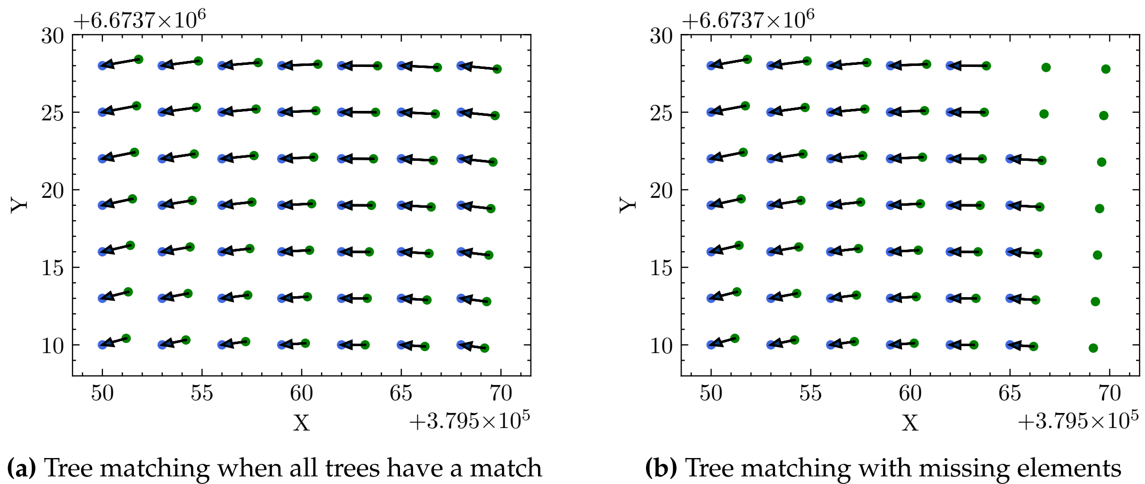
[15] The error is consistent across a single scan, but not across multiple rescans.



(a) Tree matching when all trees have a match **(b)** Tree matching with missing elements

Figure 5.11: Results of the Hungarian algorithm under real conditions in two different scenarios. The green and blue dots represent the old and new positions.

To further evaluate this approach, we generated an artificial set of points spaced 3 meters apart, which we rotated and translated to obtain another set. The data generated is purposely similar to that expected in a tree plantation.



(a) Tree matching when all trees have a match **(b)** Tree matching with missing elements

Figure 5.12: Results of the Hungarian algorithm under near-reality conditions in two different scenarios. The green and blue dots represent the old and new positions.

With both real (Figure 5.11) and artificial (Figure 5.12) data, the results are convincing as all correct matches are found. In cases where some trees are missing, we observed that the algorithm would have gone awry without the 2.5-meter constraint with the artificial data. Other approaches (RANSAC, iterative nearest point, etc.) may yield better results in forests where this constraint is not applicable.

5.5 Summary

This section covered many methods and algorithms, including several parameters that we found most optimal. Table 5.2 summarizes the proposed workflow.

Step	Approach	Parameters
Rotation	Transformation matrix	Rotation = 90° around the x-axis
Reprojection	EPSG-documented formulas	-
Normalization	Cloth Simulation Filtering	-
Resampling	Voxel-based sampling	Spatial resolution = 0.08
Classification	Hough Transform	Height interval = [0.5, 1.75], step = 0.25, pixel density = 0.1, pixel size = 0.03. min votes = 4, max diameter = 1
DBH extraction	Circle-fitting RANSAC	Iteration = 10, expected inliers = 0.8, number points per iteration = 10
Area estimation	Convex Hull	-
Trees alignment	Hungarian Algorithm	Cost matrix = distances between the trees, max distance = 2.5

Table 5.2: Summary of the proposed workflow

The proposed workflow is fully automatic and covers the entire end-to-end process, unlike similar workflows observed in the literature, as discussed in the introductory section. The literature concludes [3,41] that there is a need to design an application that implements such workflow. The implementation of such an application is the subject of Chapter 6. We will see that the workflow is part of a so-called custom layer, which allows the use of other workflows if necessary.

The results are promising so far, which is in line with related studies that concluded that the iPad LiDAR is a viable and low-cost alternative to the expensive TLS approach. In Chapter 7, we will re-examine this workflow from a higher vantage point where the whole workflow is evaluated under field conditions. The chapter will compare the measurements extracted via the application, which implements this workflow, with the measures we collected manually.

Chapter 6

Implementation and Internals

This chapter covers the implementation of the solution, which includes the development of a backend (discussed in Section 6.2) and a frontend (discussed in Section 6.3). These sections address the technical details, design and architectural choices, and reveal some important subtleties behind the development phase. During the implementation, there was a strong emphasis on reducing technical debt; this chapter will equally focus on that aspect.

6.1 Client-Server Overview

The application is divided into a client (frontend) in charge of the presentation and a server (backend) responsible for the data storage and business logic.

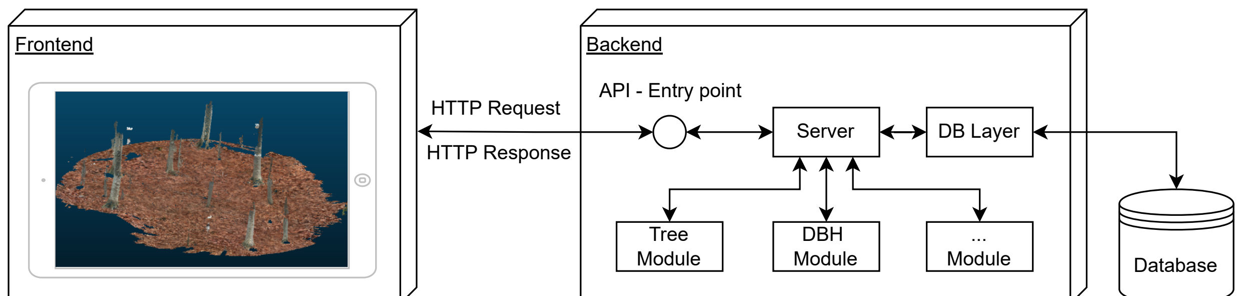


Figure 6.1: A high-level overview of the client-server

As depicted by the Figure 6.1, the client sends HTTP requests to the server's entry point (i.e. its API), the server sends back responses, and the client displays the results

on user-friendly interfaces. A database is used to store the data, for instance the stands that have been scanned alongside their trees.

6.2 Backend Design and Architecture

One might wonder why not process everything directly on the iPad since its performance is more than sufficient and it would avoid going through the network. This choice is a comprise. We opted for this solution for the following reasons:

- It enforces the principle of separation of concerns [52]. Since the server is not tied to the client, it can be written, maintained, extended, and reused without being hindered by the client.
- In the future, the LiDAR technology is likely to expand to other devices such as those running the Android operating system; the server is a cross-platform choice that would remain relevant.
- The server is not only meant to be consumed mobile devices; one may use the server from a web client and send TLS point clouds. In fact, before acquiring the iPad we used TLS point clouds instead.
- The server can be used by several clients simultaneously to survey a region in parallel.
- Going through the network is indeed a bottleneck; however, eliminating unused fields and compressing the file can significantly reduce its size — about 20% of the original size (see Section 5.2.2).

This section discusses the technical intricacies of the server and how it results in a flexible, maintainable, and testable application — or RESTful API to be precise, that is, a web application programming interface that conforms to the REST [53] architectural guidelines and constraints.

An overview of the structure will be presented in Section 6.2.1 which will serve as a guideline for the rest of the backend discussion. Before going further into the technical details, an overview of the tools and technologies used to implement the architecture will be given in Section 6.2.3 and Section 6.2.2.

6.2.1 Architecture Overview

REST is entity-oriented, which means that the fundamental unit of a REST API is the entity, i.e. the resource we can create, get, update, and delete. Each entity corresponds to an API resource path.

The different entities present in the architecture are described in Table 6.2.

Entity	**Description**
Stand	A cluster of trees. Each stand corresponds to a single point cloud of a given zone. A stand can be updated by uploading a new point cloud of the same area.
Tree	A tree extracted from a point cloud. The tree is part of a stand and is unique with respect to its geographical location.
Tree capture	A capture of a tree at a given time. New tree captures are created when a stand is updated.
Stem diameter	A diameter of a tree capture at a specific height. The same capture can have multiple diameters at different heights.
Stand history	An old version of a stand. The stand histories represent the old versions of the stands that have been archived after new ones have been uploaded.

Table 6.1: Description of the backend basic entities

Each entity is defined by its own set of components. These are the `Controller`, `Schema`, `Interface`, `Service`, and `Model` parts depicted in Figure 6.2. The purpose of the components is to ensure that the principle of separation of concerns is followed. For example, the model component interacts with the database while the schema component is responsible for data validation.

Figure 6.2 also contains a `Custom Layer`. One of the goals of the architecture is to be flexible so that different developers can integrate their own workflows and algorithms. To achieve this, we have designed a custom layer that is meant to be easily modified. This layer has been supplied with the algorithms described in Section 5.4 but is designed to be configurable.

Finally, a relational database has been set up to store all the data that can be collected and calculated by the application.

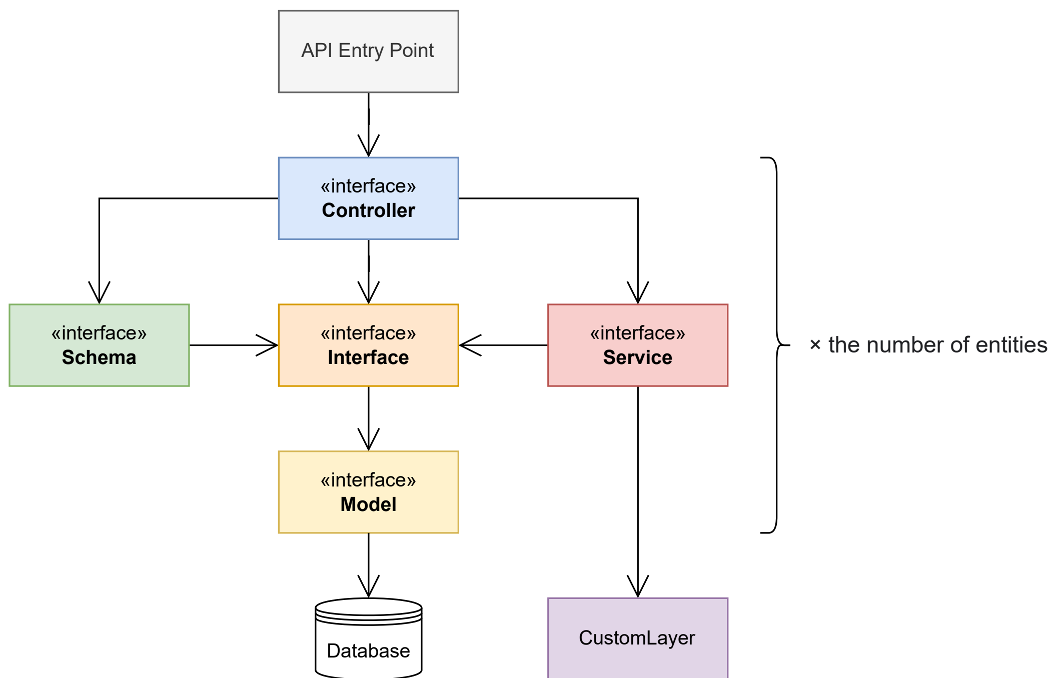


Figure 6.2: Overview diagram of the backend structure

As the implementation details are closely related to the tools and technologies used, these two elements will first be discussed in Section 6.2.2 and Section 6.2.3 respectively.

In the subsections that follow, the elements outlined in this overview, namely the API endpoints (Section 6.2.4), the database (Section 6.2.5), the components (Section 6.2.6), and the custom layer (Section 6.2.7), will be presented in detail in a top-down approach — from the big picture to the technical details.

The color code in Figure 6.2 will be used throughout the following subsections to help highlight which parts of this overview the diagrams correspond to.

6.2.2 Tools & Libraries for Forest Inventory

Reinventing the wheel is a great way to learn how a wheel works, but it is not a good way to build a car. As the main objective of this thesis is to build an application rather than to improve on current algorithms and tools, we have chosen to rely on existing libraries and tools.

That being said, libraries specific to forest inventory, particularly those based on TLS point clouds, are scarce and many are no longer maintained.

Those looking to contribute to the field would almost certainly come upon one of the tools listed in Table 6.2.

Tool	Description	Purpose
PDAL [54]	Generic C++ library for manipulating and transforming point cloud data.	Reprojecting (into another coordinate system) and rotating the point cloud.
LidR [55]	R package for Airborne Laser Scanning (ALS) data analysis.	Ground point classification, point cloud normalization and decimation.
TreeLS [56]	R package for processing TLS point clouds of trees.	Tree detection and stem segmentation.
LASTools [57]	Suite of LiDAR processing tools known for their very high efficiency.	LAS file compression.
CloudCompare [58]	Application for point clouds manipulation and visualization.	The application does not use it per se, however we heavily used it to constantly verify our results and to do manual manipulation of our data.

Table 6.2: Summary of the tools used throughout the development

Note that TreeLS is no longer maintained. The library and its forks do not compile anymore. The solution was to replace `PI` with `M_PI` everywhere in the code. We decided to fork the repository and apply the fix[1].

In addition, standard mathematical Python libraries such as `Numpy`, `Pandas`, and `SciPy` have been used extensively.

6.2.3 Technological Agnosticity

Technology-agnostic architectures are able to adapt to inevitable changes in tools, technologies, and trends. The architecture is designed without any technological considerations in mind. This means that the design can withstand a technology shift, which we believe is a significant step toward building a more maintainable and future-proof system.

[1] https://github.com/Lymero/iPad-LiDAR-Forest-Inventory/tree/dev/libs/TreeLS

However, the actual software uses specific technologies, and developers and maintainers do care about how the system is built. Therefore, while the design is technology-agnostic, the following technologies were used in the development of the backend:

- **Python** as the programming language of choice because of the numerous tools and libraries it offers while also being quite powerful.
- **Flask** as a minimalistic, extensive, and maintainable web framework, allowing developers to build exactly what they want with the help of additional libraries.
- **Marshmallow** for input data validation, deserialization, and the serialization of Python objects to JSON for the HTTP API.
- **SQLAlchemy** as an SQL toolkit and object-relational mapper (i.e. ORM).
- **Flask-RESTX** for building a REST API and for the Swagger documentation.
- **Injector** for dependency injection to reduce the coupling between classes and their dependencies, thus making the code more testable, maintainable, and readable.
- **Pytest** as a testing framework.

Because the goal of this section is not to explain in depth how the architecture works, we will not delve into the details.

A high-level interaction between the technologies is illustrated in Figure 6.3.

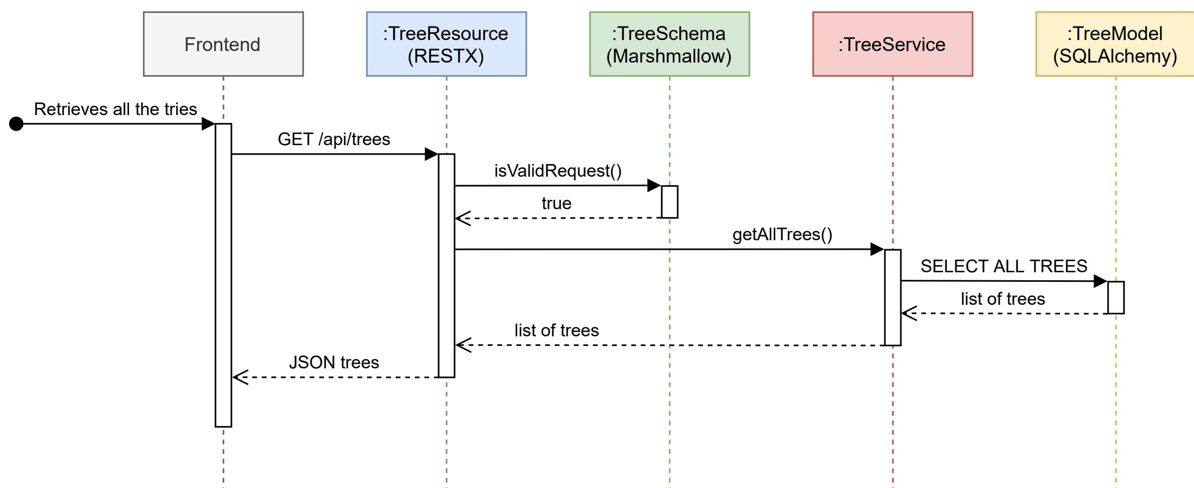


Figure 6.3: High-level sequence diagram of the interaction between the technologies

In broad strokes, requests are routed using RESTX Resource classes; the input data is validated via one or more Marshmallow schemas. The data is processed by a service in conjunction with an SQLAlchemy model. Finally, the output is serialized into JSON via Marshmallow and sent back as an HTTP response to the client.

Swagger is being used to document this process partially. This process will be discussed in more detail later in this document.

6.2.4 Application Programming Interface

The previous sections have provided an overview of the architecture, technologies and tools used for the development of the application. This section provides the end result, i.e. the set of exposed endpoints.

The server exposes a total of 32 API endpoints as documented in Table 6.3. Note that the resource $*$ is a globbing wildcard that refers to all the resources (i.e. stands, trees, tree captures, diameters, and stand histories).

HTTP Method	Routes (prefix: /api)	Behavior
GET	/*	Retrieves all entities of the resource.
POST	/*	Creates a new entity.
GET DELETE PUT	/*/{id}	Retrieves/deletes/updates the given entity.
POST PUT	/stands/pointcloud	Creates a new stand. Underlying entities are created/updated. Requires a geo-referenced point cloud las/laz file.
GET	/stands/{id}/{histories, trees}	Retrieves the list of histories/trees for the given stand.
GET	/trees/{id}/{captures, diameters}	Retrieves the list of captures/diameters for the given tree.
GET	/tree_captures/{id}/diameters	Retrieves the list of diameters for the given capture.

Table 6.3: List of the API endpoints

The response status codes are in accordance with the HTTP specifications[2]. For example, a poorly formatted request will result in a response with the error code 400 (Bad Request), and a request to a non-existent resource will return an error code 404 (Not Found). Other HTTP status codes can be found in Table 6.4, which also contains information about the expected JSON response when no error occurs.

HTTP Methods	Single Resource (e.g. /api/trees/46)	
	Response Status Code	**JSON Response**
GET	200 (OK)	Tree 46 all information
POST	405 (Method Not Allowed)	-
DELETE	200 (OK) if exists, 404 (Not Found) if not	-
PUT	200 (OK) if exists, 404 (Not Found) if not	Updated tree 46 information
	Collection Resource (e.g. /api/trees)	
	Response Status Code	**JSON Response**
GET	200 (OK)	List of all the trees
POST	201 (Created)	New tree information
DELETE	405 (Method Not Allowed)	-
PUT	405 (Method Not Allowed)	-

Table 6.4: API status codes & JSON responses

When a client error (4xx status codes) occurs, the client receives a meaningful error message in addition to the status code. In the event of a server error, the stack trace is logged and an error 500 is sent to the client.

All the endpoints are thoroughly documented via Swagger. It includes an explanation of each endpoint as well as a description of the fields (field names, required or not, what types, and so on).

6.2.5 Data Storage

One of the goals of the system is to provide a way to perform forestry monitoring, allowing users to follow the evolution of a stand and its trees over time. As a result, the content of `POST` requests must be saved in order to be accessed later via `GET` requests.

[2]RFC 7231 - Response Status Codes: https://datatracker.ietf.org/doc/html/rfc7231#section-6

To accomplish this, the three data storage solutions listed in Table 6.5 are responsible for storing the data at different levels of granularity.

Storage	**What data**	**Notes**
Plain .laz files	The compressed point clouds. One file per point cloud.	High disk usage; keep track of the original data.
CSV files	The direct attributes extracted from the point clouds. One file per point cloud.	Low disk usage; contain the most useful information.
PostgreSQL Database	All processed data post-extraction.	Average memory usage; requires some maintenance and configuration; keeps tracks of all the extracted and computed data.

Table 6.5: Data storages used.

Each storage shares the same unique ID (timestamp) for the same data source — i.e. for each point cloud. Consequently, the data extracted from a given point cloud can be retrieved by querying the database.

Database Design

The database tables directly map all entities in the architecture. Some constraints are enforced by the database mechanisms (`NOT NULL`, `Foreign Key`, `Unique`, `Primary Key`), such as the fact that a tree capture should not have two diameters estimated at the same height or a stand should not have two histories at the same date. Other constraints, such as ensuring that the `tree_count` entries in the `stands` table correspond to the correct number of trees in the trees database, are enforced by the server.

The most challenging aspect of the database design was tracking record updates. In other words, how do you keep the previous data for a stand when a new point cloud is uploaded for that same stand?

One of the most effective ways to accomplish this task is to create an identical table (see `stand_histories` in Figure 6.4) whose primary key is the concatenation of the

stand identifier and the capture date. Whenever the `stand` table is updated, a copy of the data is inserted into the `stand_histories` table. We have opted for this solution.

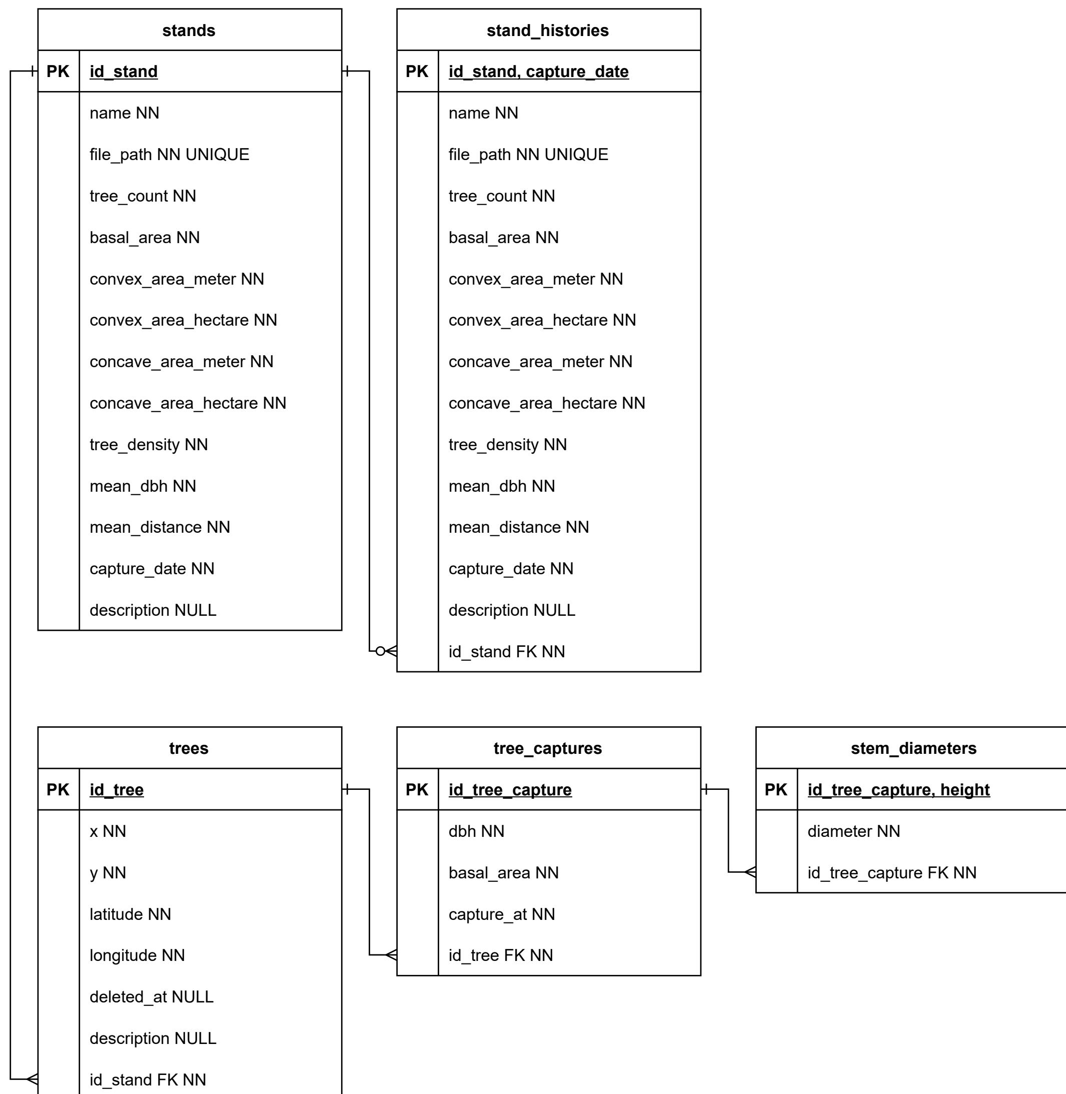


Figure 6.4: Full database diagram; the business-related fields are explained in Chapter 3.

The other option considered was to embed the history directly into the `stand` table. However, given that the bulk of queries are run on the current version of the stands, this solution is inefficient in performance. The table would be `n` times larger, with `n` depending on the update frequency, which would slow down most queries.

When updating a stand, it is possible that some trees are no longer present, for instance they may have been harvested. Instead of deleting these trees and losing the associated data, we use the field `deleted_at` in the table `trees` to indicate such occurrences. If the field is set to `NULL`, it means that the tree is still part of the stand.

6.2.6 Components

The previous subsections focused on the high-level elements, i.e. the elements that the client can interact with directly. The following subsections will dive into the technical details of the structure.

In the overview (Section 6.2.1), we saw that the entity is the basic unit of the structure and that the following components make up an entity in our implementation: controllers, models, interfaces, schemas, and services. The subsections that follow elaborate on the role of each individual component and how they are tested. Code snippets are given when they contain concepts important to our implementation.

Model

The model defines the Python representation of the entity. SQLAlchemy requires that the class inherits from `db.Model` and defines the entity's relationships. The fields map onto tables and fields in the database — see Section 6.2.5 for database specifications.

Testing a model consists of ensuring that it can be instantiated. We have set up the tests to run on a development database, so there is no need to roll back after each test or worry about causing problems with the production database.

The project structure is organized so that there is one folder per entity and each folder contains one file per component that comprises the entity. Unlike the rival structure, which scatters the files about the same entity across separate directories such as "models," "services," and so on. This structure makes it easier to manage a large project or at least makes it scalable in that respect.

Interface

The interface is the blueprint for the model. It defines what the model is (types, fields, and so on) — that is, an abstraction of the SQLAlchemy model. This breaks the dependency over the SQLAlchemy underlying database types, thus achieving loose coupling that increases flexibility (e.g. easier to instantiate a model object) and re-usability (e.g. easier to switch ORMs).

Testing an interface consists of checking that it can be used to instantiate its model.

Schema

As the layer between the clients and the API, the schema checks the responses and requests for well-formedness by the provided definitions. It also manages the uniformization of naming conventions for its corresponding entity. For instance, it handles the translation of the fields between the frontend and the backend as the former uses CamelCase and the latter snake_case.

```
1  class TreeSchema(Schema):
2      class Meta:
3          ordered = True
4      id = fields.Number(attribute="id_tree")
5      latitude = fields.Number(attribute="latitude", required=True)
6      longitude = fields.Number(attribute="longitude", required=True)
7      deletedAt = fields.DateTime(attribute="deleted_at")
8      description = fields.String(attribute="description")
9      idStand = fields.Number(attribute="id_stand", required=True)
```

In this simplified example, three fields are optional; if the required fields are not set or ill-typed, an HTTP error (400 Bad Request) is raised and sent to the client.

Testing a schema is similar to testing an interface with an additional naming convention check.

Service

The service component handles the business logic and data manipulation using models and interfaces. This includes, among other things, point cloud manipulation and CRUD (Create - Read - Update - Delete) database operations.

The following SOA (Service-Oriented Architecture) properties [59] apply to all services:

- They logically represent a repeatable business activity with a specified outcome (e.g. retrieve stored trees, extract stem measurements from a point cloud).
- They are implemented in an opaque manner; they can only be viewed in terms of inputs and outputs, with no knowledge of their inner workings.
- They may have inter-service dependencies.

These properties ensure that the services are kept modular. As they are designed in such a way as not to have knowledge of who uses them, they are also reusable. Additionally, the tests and the documentation are confined to the services themselves because it is not essential (or desirable) to consider the intricacies of the upper layers.

The services have been implemented using the dependency injection design pattern in order to meet these properties — or requirements. The concept of dependency injection is to provide objects with their dependencies rather than having them built internally. This is a great way to achieve loose coupling (no hard-coded dependencies) and separation of concerns. It is especially relevant for testing since dependencies can be mocked or stubbed out.

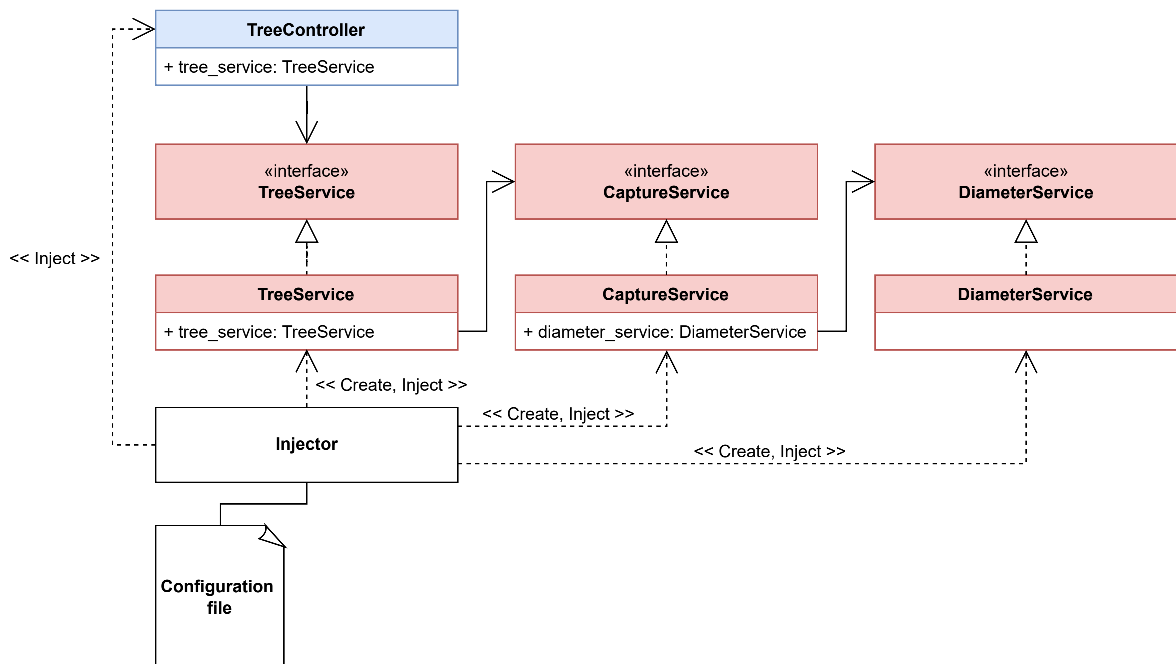


Figure 6.5: Dependency injection class diagram

Figure 6.5 depicts how dependency injection takes place in the application. The `Injector` reads the configuration file (interface-implementation key-value mapping), instantiates the specified implementations, and injects them where needed. Note the number of inter-service dependencies.

Service testing is done by ensuring that the operations perform as intended. These are the most crucial tests to write because services are the core business of the application and are the ones that are modified the most. When testing a service, the dependencies

must be mocked in order to focus on the service itself. This is accomplished by using a test configuration file that instructs the `Injector` to instantiate mock objects.

Controller

The controller component links the routes and the other components to define API entries and process requests. As Swagger is used, the controllers are also responsible for documentation.

The following code snippet illustrates the `POST /api/trees` route:

```
@api.route("/")
class TreeResource(Resource):
    @inject
    def __init__(self, tree_service: TreeService, api):
        super().__init__(api)
        self.tree_service = tree_service

    @accepts(schema=TreeSchema, api=api)
    @responds(schema=TreeSchema)
    def post(self):
        """Creates a new tree."""

        return self.tree_service.create(request.parsed_obj)
```

The route of this resource is defined by `api.route("/")`, which in this case results in `/api/trees/` as the full route. The `Injector` in Figure 6.5 injects the tree services using the `@inject` decorator[3]. The schemas to be used are specified by the `@accepts` and `@responds` decorators; these are also used for Swagger documentation, such as documenting which fields are required.

Testing a controller is done by mocking the service and by simulating a client[4] that calls the API. The mock ensures that its controlled return value matches the API response.

[3] A Python decorator is a function that wraps a function to extend the behavior of the wrapped function, more detailed information is available at https://peps.python.org/pep-0318/

[4] It is easier to use the FlaskClient class Flask provides for the tests.

6.2.7 Custom Layer

The services relay the implementation of the algorithms to the custom layer whose goal is to implement the workflow and algorithms. The workflow we have developed is presented in Section 5.4.

The purposes of the custom (or algorithmic) layer are as follows:

- To enable other developers and researchers to use their own workflows or algorithms for their experiments via our system. For instance, a researcher may want to use the system to evaluate a novel tree detection technique.
- To allow the algorithms (or their parameters) to be changed depending on the forest/environment type.
- To make it possible for the system to adapt to different coordinate systems (see Section 5.3). Users in other parts of the world may prefer to (and should) use coordinate systems more appropriate for their locality.
- The pre-processing algorithms may require some tweaking. For instance, some devices require the point cloud to be rotated, while others do not.
- More importantly, to allow the system to be flexible and enable researchers to find creative approaches to carry out the process in ways we had not considered.

As this layer is designed to be generic and independent from the other parts of the system, users will be able to provide their own custom layer by simply implementing a set of classes. The provided classes will then be integrated into the existing system in a plug-and-play fashion thanks to object-oriented mechanisms that avoid the creation of concrete dependencies between the different layers of the application.

This layer and its mechanisms are designed to be as intuitive and flexible as possible. Various well-known design patterns are implemented to achieve this goal. Default classes have also been implemented to provide a complete workflow that can be used on its own while also serving as an example. We attach great importance to this part because we believe it is essential to make it easy for experts to contribute to the field by providing a simple to use framework.

This subsection presents how the layer work and can other developers integrate their custom algorithms.

How it Works

From an object-oriented point of view, each provided class must implement its corresponding interface — or parent class in Python. The interfaces are defined in such a way as to be used regardless of the operation of the concrete classes. In addition to specifying a contract to comply with, the interfaces allow for dynamic changes to an implementation without changing its type thanks to subtype polymorphism [60]. This is known as the strategy design pattern, which involves defining a family of algorithms, having one algorithm per class, and making their instances interchangeable.

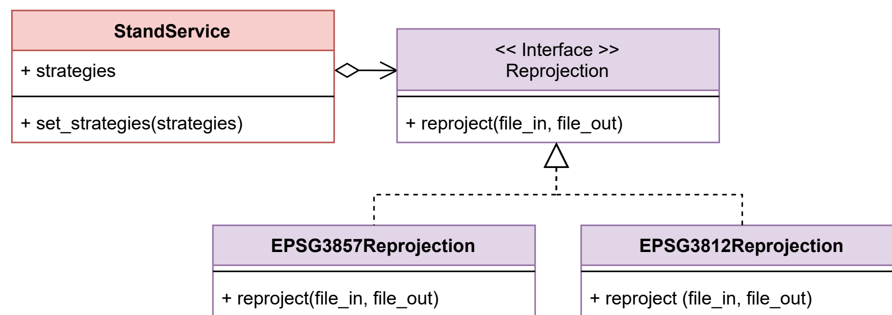


Figure 6.6: Class diagram of the stragy pattern for the reprojection

A class must be able to integrate with the rest of the system once implemented. But how to instantiate the desired class? The easy solution is to change the instantiation logic where required. However, this would be problematic because it would force users to modify parts deep in the code, plus the solution would create hard dependencies.

The chosen solution is to go through a configuration file containing the names of the classes to instantiate. However, this requires the implementation of a mechanism to instantiate a class based on a string. There are a least three mechanisms that can solve this problem.

The first mechanism imagined consists of a mile-long switch looking for the classes to instantiate. However, this is a well-known code smell and it is not elegant.

The second mechanism imagined is based on the use of the flyweight pattern. It consists in pre-loading information and retrieving it when needed, for example to implement a cache. It is most often used to prevent performance problems but can have a broader use as in this case. Concretely, a flyweight in this context would be a class with a dictionary (access and addition in a constant time-complexity) whose keys correspond to a class to be instantiated and values to the instance. The flyweight would

be loaded when the application is launched. By using the names of the classes in the configuration file, it is possible to retrieve the desired classes at runtime dynamically. When the users want to add their own classes, they would have to add the key-value pairs to the flyweight.

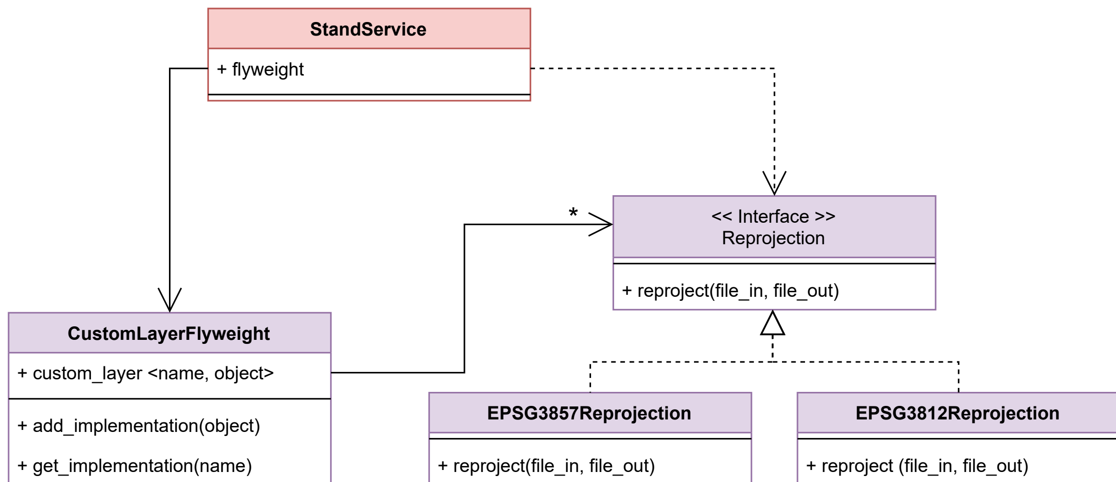


Figure 6.7: Class diagram of the flyweight pattern for the reprojection

The third option is to use the dependency injection approach, which has already been implemented for the services in Section 6.2.6. Because this solution would create no new classes outside of the `Injector`, the only configuration task when adding a custom layer will be to update the entries in the configuration file. This is the chosen solution for its simplicity.

Custom Layer Integration

As an example, suppose we want to implement a new type of reprojection. The example remains valid for other classes of the custom layer. The actions to take are as follows:

1. Make a subclass for the interface/class `Reprojection`.
2. Override `reproject(file_in, file_out)` with the intended behavior.
3. Add the the entry `<Reprojection, class_name_of_your_subclass>` to the file `dependency_injection.py`. If the key `Reprojection` is already defined, edit its value. The subclass will be instantiated as a singleton.
4. (Optional but advised) Add and run the tests for the subclass.

6.2.8 Summary

It is with the idea that tools and developers do not remain the same over the years in mind that we designed the server. We implemented a system that can handle changes in tools and needs, but also changes in people.

To do so, we had three main criteria that helped us guide most of our technical choices: flexibility to accommodate future changes in requirements; maintainability to continually modify the application for corrective or improvement purposes; and testability to allow the application to grow serenely and be modified without fear.

These three criteria have been met by involving well-known techniques, and more specifically in the following ways:

- Flexibility is achieved by implementing a custom layer that allows developers to extend and configure the server for their own needs in terms of tools or algorithms. The custom layer has been implemented with well-known design patterns, allowing the code to be explicit the moment the pattern is named.
- Maintainability is achieved by the overall design and the choice of well-known structural patterns to implement it. The enforcement of the separation of concerns principle and associated documentation make trivial the creation/correction of routes or entities in the application. The implementation of a logging system enables for the detection of errors and subsequent corrective maintenance.
- Testability is achieved by organizing the application around entities and components in such a way that hard dependencies are limited, allowing each part of the code to be tested separately.

These criteria accompanied us throughout the whole development, requiring us to modify the code many times. The entire structure was refactored once to better meet our objectives. Although probably imperfect, we think that the server as it is is a good starting point for the future.

Aside from these criteria, the developed server exposes approximately thirty routes to assist with tree monitoring, comes with a database for data storage, and allows for the development of a client application to assist foresters in their laborious work.

The following section (Section 6.3) discusses the development of the associated frontend application that actively uses this server.

6.3 Frontend Design and Architecture

As described in the overview (Section 6.1), our solution consists of a backend and a frontend. The backend handles the tree measurement computation logic and is used to store the results. The frontend can communicate with the backend using user-friendly interfaces that are used to display these results, but also to modify and add new data.

This chapter will focus on the features offered by the application developed specifically for Apple devices featuring a LiDAR sensor (see Chapter 4.1). A brief overview of all the screens available to the user will be provided as an introduction. We will then proceed to explain the structure, which technical choices were made, and what motivated them. A separate section will also discuss the application's local database, which allows the user to access measurements when no Internet connection is available. This chapter ends with an in-depth analysis of a concrete use-case, the creation of a stand via the upload of a point cloud file, which should shed some light on how the major components of the application all work together.

6.3.1 Overview

Adding a Stand

Upon starting the application for the first time, the user lands on an empty list of stands, as shown in Figure 6.8 **(1)**. They can either upload (**1.b**, **1.c**) their own point cloud files to the server, or fetch (**1.a**) already available stands.

In this example, the user chose to upload 4 point cloud files locally stored on the device. Adding stands in this manner is performed in 2 phases:

1. Uploading the point cloud file to the server;
2. Waiting for the point cloud to be analyzed by the backend in order to retrieve all the relevant computed data for that stand (stand measurements, trees and their captures, tree capture diameters).

The user can follow the first phase of this process thanks to the status `uploading` that is accompanied by a progress indicator (**1.d2**). The user can tap on a file being uploaded to cancel the upload.

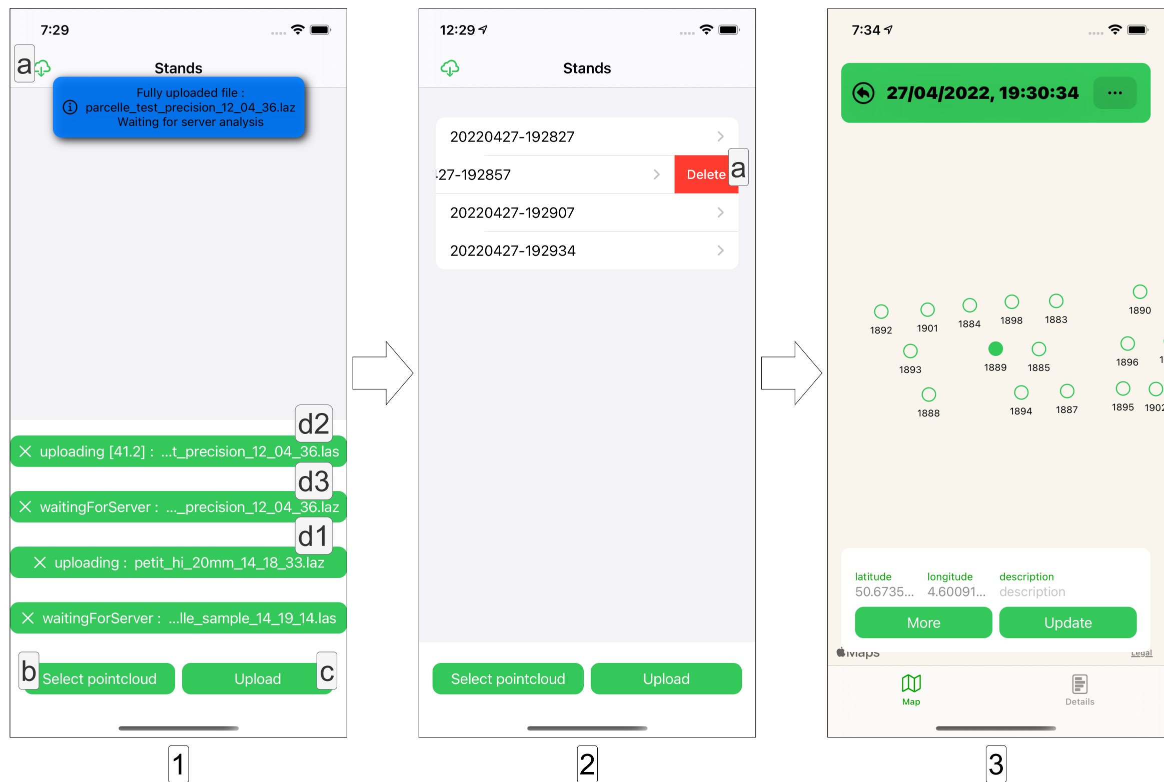


Figure 6.8: Uploading and browsing a newly created stand

Once fully uploaded, the status is updated to `waitingForServer` (**1.d3**), indicating the beginning of the second phase. Note that smaller files only display the `uploading` status without any progress indication as they will quickly be uploaded and processed by the backend.

Finally, once the server has finished processing the point cloud, all the relevant data is fetched from the API, stored locally for offline use, and the stand list is refreshed.

The user can swipe left on a stand to remove it from the list (**2.a**). If the server confirms that the deletion of the remote entity was a success, the local copy (Section 6.3.3) of that stand is also removed from the device.

Information about a Stand and its Trees

The stand list shows all locally available stands. Tapping on one of them displays the stand map screen, as presented in Figure 6.9 **(1)**.

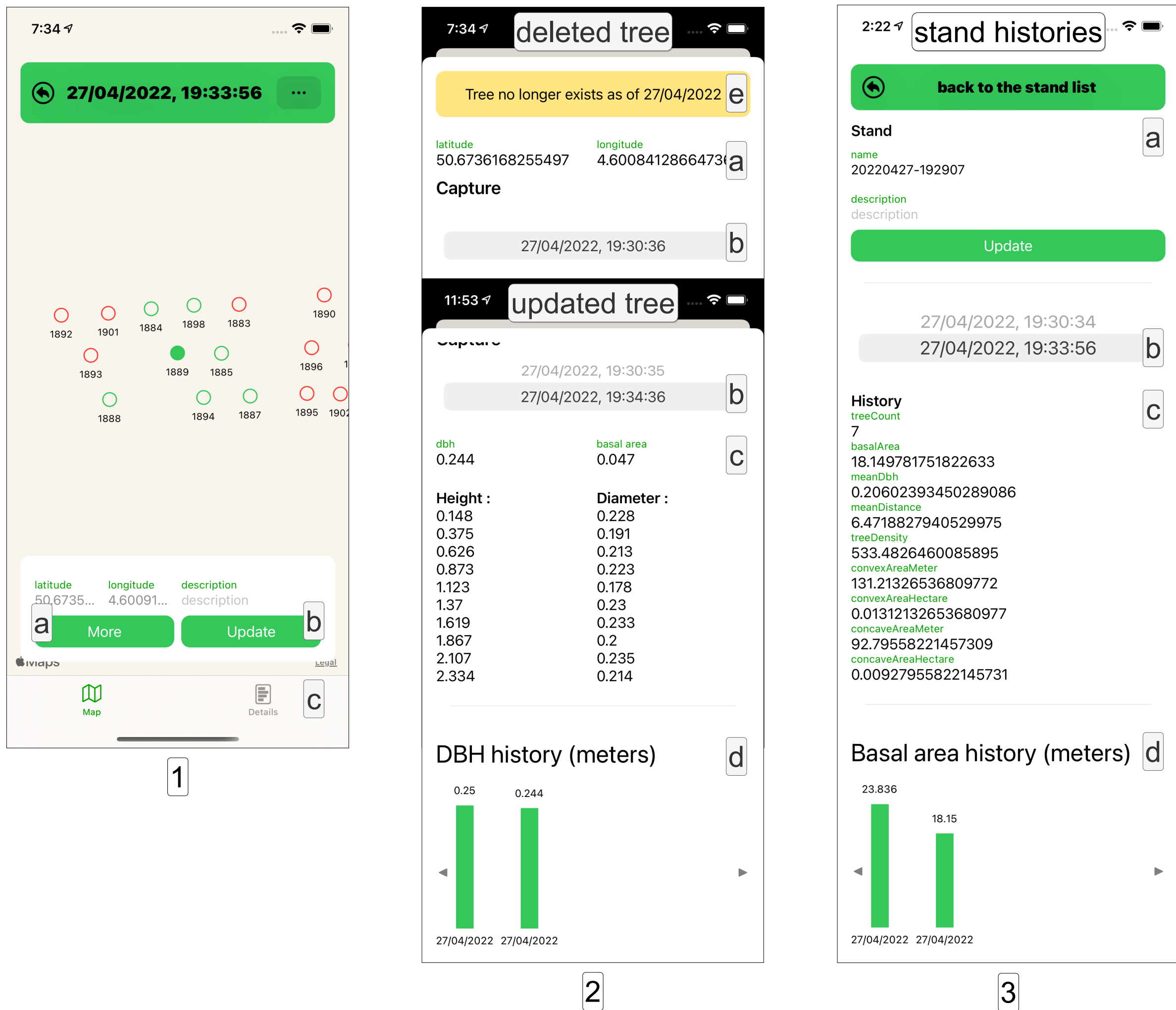


Figure 6.9: Stand and tree information screens

The map shows the trees that are part of the selected stand, as well as the live current user location. It can be zoomed in and out and moved around. The trees are represented with green circles. The circle is red if a tree has been flagged as deleted and is no longer a part of the stand. The user can perform a long press on a tree to delete it. A tree can be selected by tapping on it. Upon its selection, the map focuses on the tree. A popover at the bottom contains some information about the selected tree, and is also used to display and edit the tree description (**1.b**).

Tapping on the button "More" (**1.a**) brings the selected tree detail screen. For a deleted tree, a warning badge (**2.e**) alerts the user of the deletion date of the tree, and only the captures performed before its deletion are shown.

The information displayed on this screen is as follows:

2.a The full latitude and longitude of the tree;

2.b A picker component allowing the user to display a specific capture of that tree;

2.c The information (DBH, basal area, diameters) available for the selected capture;

2.d The DBH history of all recorded captures for that tree.

From the stand map screen, selecting the stand details tab (**1.c**) displays all the current stand measurements. The user can edit the stand name and its description (**3.a**). The stand histories are browsable in the same fashion as the trees and their captures (**3.b**, **3.c**). A diagram of the evolution of the stand basal area is displayed at the bottom of the screen (**3.d**).

Updating a Stand

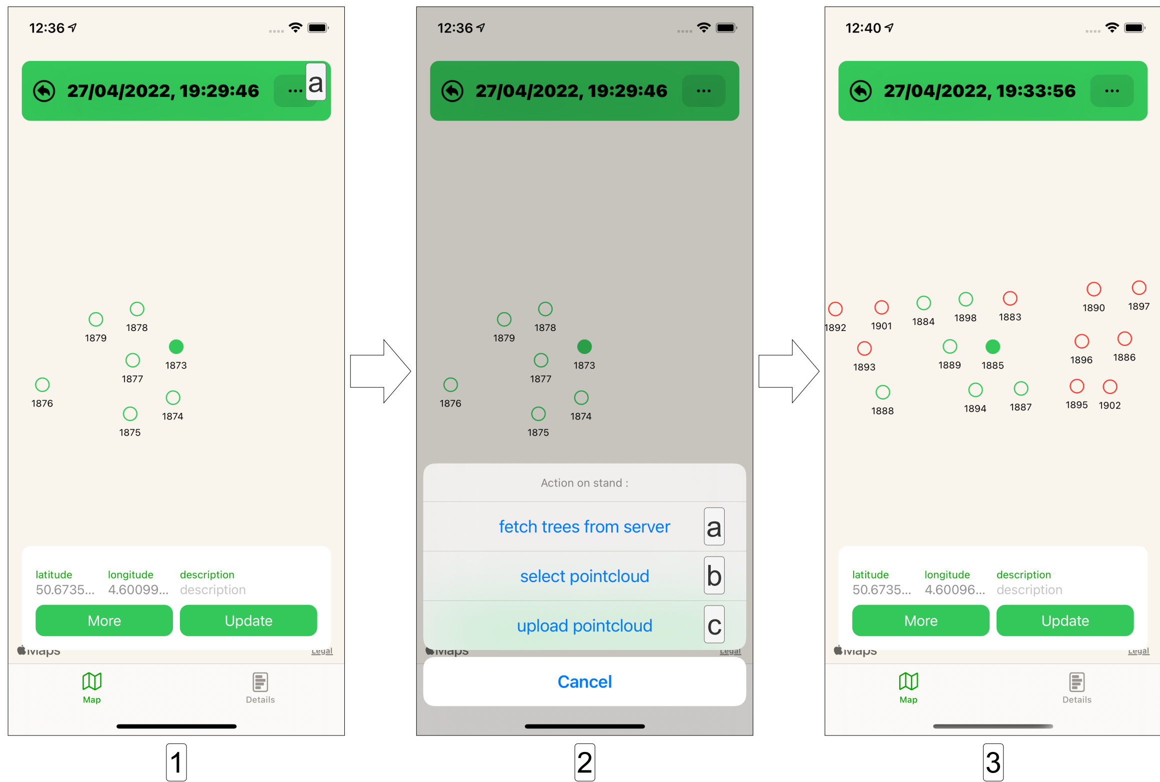


Figure 6.10: Stand update via the upload of a pointcloud file

The stand map screen (see Figure 6.10) can also be used to fetch and send new information about a stand. Locally stored data can become out-of-sync with what is stored on the server, the top-right menu (**1.a**) allows the user to refresh the locally stored trees with the ones found on the server (**2.a**). They can also update the stand and the trees it contains using a point cloud file locally stored on the device (**2.b, 2.c**).

The update of a stand is quite similar to the process of adding a new stand: the file is first uploaded, then the application waits for the backend to analyze the point cloud. Once fully processed, the application finally fetches all the computed data, updates the local database, and refreshes the view.

6.3.2 Structure

Foundation

The mobile application targets iOS versions 12 and above. It is written in the `Swift5` programming language native to the iOS mobile development platform. One might wonder why the application was not written using Web technologies such as Javascript and HTML/CSS. After all, the application is devoid of any native specific needs and could have run on a Web view. Programming in a native language was motivated by the fact that, in the future, it would be desirable to extend the application with a custom-made camera module specifically aimed at capturing point clouds of trees. As discussed in Section 3.3, it could help gather more information such as an estimate of the height of the tree using the device sensors. Such features, which are closely related to the device hardware, are next to impossible to implement without the APIs [61–63] offered by the iOS ecosystem. This choice was also motivated by the fact that it was easier to integrate a local database that is used throughout the application.

MVVM

Several patterns aim at helping in the development of scalable and testable applications. The MVC (Model View Controller) pattern is well-known amongst mobile developers, but in this case we settled on the MVVM (Model View ViewModel) pattern as it provides, among other benefits, a stronger separation of concerns [64]. This section briefly reviews both patterns, highlighting the shortcomings of the MVC pattern as well as the improvements proposed by the MVVM pattern.

The three key components of the **MVC** pattern (illustrated in Figure 6.11) are the **View**, the **Controller**, and the **Model**. The domain data and logic are handled by the Model component. This component is in total isolation from the View and the Controller. As a result, the application logic is independent of how domain data is presented. The Controller is responsible for handling user input; it holds a reference to the Model in order to modify it. The View component displays the information contained in the Model. It is subscribed to the observable Model in order to be notified of changes made to the information it contains.

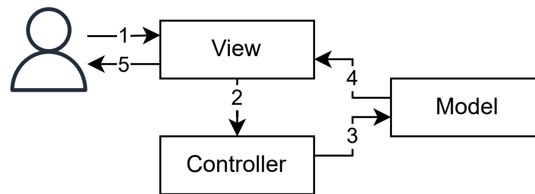


Figure 6.11: Typical execution flow using the MVC pattern

The user interacts with the View (**1**) which calls a method on the Controller (**2**). The Controller performs an action on the Model and potentially modifies the data it contains (**3**). Upon modification of the Model information, the View is notified of the change (**4**) and updates the information shown to the user (**5**).

The main issue with the MVC pattern is that the view state is fully dependent on the content of the model. Suppose a developer wants to change the color of a textfield for specific value ranges. In that case, they must either add domain-irrelevant information to the model or design a dedicated View with extra code to handle these scenarios. As a solution to this problem, the MVVM pattern introduces the ViewModel component that can manage the View state.

The three key components of the **MVVM** pattern (illustrated in Figure 6.12) are the **View**, the **ViewModel**, and the **Model**. The View holds a reference to a ViewModel. The ViewModel exposes data to the View. It acts as a bridge between the Model and the View but also handles View specific state. The View can interact with the Model using the actions described by the ViewModel. The information contained in the Model can be accessed and modified by the ViewModel.

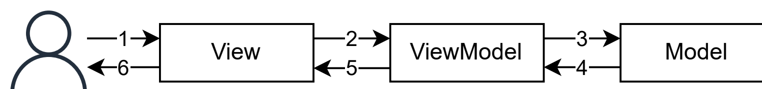


Figure 6.12: Typical execution flow using the MVVM pattern

The user interacts with the View (**1**) which calls a method on the ViewModel (**2**). The ViewModel performs an action on the Model and potentially modifies the data it contains (**3**). Upon modification of the Model information, the ViewModel is notified of the change (**4**) and updates the values published to the View (**5**) which triggers a reload of the information shown to the user (**6**).

Dependencies

We tried to include as few external libraries as possible to avoid introducing unwanted or even breaking changes to the application. We were able to only use native components offered by Apple. Here is a list of the more interesting modules used in the application:

- **SwiftUI** framework: creation of graphical interfaces.
- **Combine** framework: manipulation of data in an asynchronous environment.
- **Core Data** framework: stores data locally on the device, in a persistent container, typically for offline use.

These components and the way they interact with one another are further discussed in the last section of this chapter with a concrete example (Section 6.3.4) displaying the full one-way synchronization from the remote server to the local database.

Reactive Programming

The frontend application distances itself from not only older technologies such as Apple's `UIKit` framework [65], but also from outdated design patterns such as the observer pattern [66,67] which has been replaced in favor of the reactive programming paradigm [68].

Reactive programming is a programming paradigm built on the concept of continuous time-varying values and propagating changes. It allows developers to specify what to do and lets the language control when to do it. The paradigm makes the observer pattern redundant as it is essentially built into the language itself. In our case, the `Combine` framework used with the `Swift` language offers structures to expose observable data and consume it. This technology particularly shines with the use of asynchronous events such as downloading something from the Internet or running tasks in background threads.

`UIKit` has been available since iOS 2.0 (released in 2008) and is still widely used to build GUIs (Graphical User Interface). We however decided to use the more recent (2019) `SwifUI` framework which integrates neatly with the reactive programming approach.

It provides syntactic sugar such as the `@Published` property wrapper which allows the developer to create observable objects that automatically notify the listeners when changes occur. This is especially useful to expose data in a `ViewModel` to a `View` so it can be refreshed whenever the data is updated.

In order to fully embrace the philosophy of the reactive programming paradigm, we also chose to use the `Combine` framework as much as possible when dealing with asynchronous events.

The fundamental entities `Combine` provides are `Publishers` and `Subscribers`.
A `Publisher` creates a stream where data can be sent asynchronously. It can then be used to either send information or a completion signal that notifies that it has successfully stopped publishing values or that an error has occurred. One or more `Subscribers` can bind to that `Publisher` and are notified whenever new data is available or when the `Publisher` has stopped producing values. These streams can form rather complex pipelines that use multiple data sources from various `Publishers`.

`Combine` is also extremely useful when fetching data on the network. The ViewModel would typically create a `Publisher` running in a background thread using the `ApiDataService`. It would then be informed, on the main thread, whenever the response is available or an error occurs. The View can then be updated accordingly using a `@Published` value. Both frameworks work well in tandem in order to manage the flow of data through all layers of the application.

6.3.3 Local Database

Internet connectivity being a scarce resource when out in the wilderness, we opted for the use of a local database. This allows the user to synchronize with the remote server and use local copies of their stands (along with their relevant information) when they are out on the field. Note that, for the time being, the user will need to have access to the Internet in order to share new information with the server. A solution would be to keep a history of all local modifications and synchronize with the server when it becomes available, but it would have significantly impacted the development time of the solution.

The SQLite database engine is a popular choice in the mobile development world for its "small, fast, self-contained, high-reliability, and full-featured" implementation [69].

But we decided to settle on Apple's `Core Data` framework as it is more than a simple database engine and offers a wider range of features out of the box. For instance, it is easy to describe relationships between entities. Where a `n:m` relationship would require an extra table using a purely SQL-based approach, no such table is needed with `Core Data` as this is all handled internally by the framework.

The application performs a one-way synchronization with the remote API which is designated as the source of truth. During synchronization, the local data that is not in the remote database is removed from permanent storage. The remote, up-to-date data is then stored locally. This choice, albeit apparently simplistic, allows us to avoid implementing a bi-directional synchronization with merge conflict resolution.

Models

The application handles two types of models. The first kind is referred to as a **Model** (e.g. `StandModel`, `TreeModel`, ...). It conforms to the `Codable` protocol, allowing it to be (de)coded to and from JSON data which is the format used by the REST API. The other kind of model is called an **Entity** and is used by the `Core Data` framework.

The `Core Data` entities are quite similar to the models presented in the backend database section (Section 6.2.5). The relationships are directly expressed between entities. For instance, a `StandEntity` has a field `trees` which holds a set of `TreeEntitys`. Reciprocally, each `TreeEntity` has a field `stand` that holds the `StandEntity` it belongs to. This would have been much harder to express and use had the application featured a classic SQL database engine.

6.3.4 Concrete Flow Example

This section goes over a concrete example, the upload of a new stand, in order to better understand how all the components mentioned above interact with each other. It also aims to explain how the application deals with the asynchronous nature of some of those tasks and the dependencies between them. To be more readable, the full sequence diagram has been segmented into three parts, with each segment increasing in terms of complexity when compared to the previous one. Due to the representation of many threads and their interactions, the very last segment does not use a sequence diagram because it would be too difficult to read.

Selection of the Files

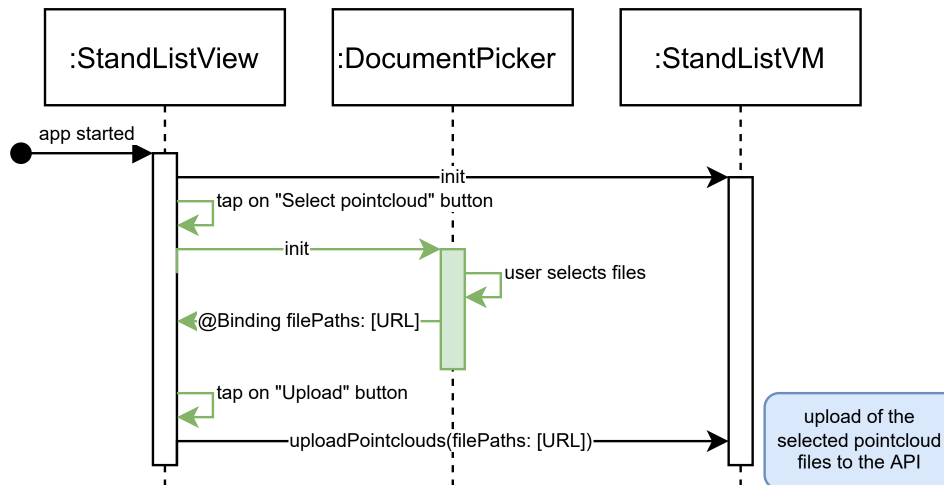


Figure 6.13: Use of a file picker to select point cloud files to upload to the API

Figure 6.13 represents the most trivial segment of this example. It mainly highlights the fact that not all operations necessarily need the use of both a View and a View-Model. Here, the View can handle the selection of a list of files without the need for a ViewModel to handle specific business logic. As a result, this picker can be reused in widely different contexts. Once the user decides to upload these files, the paths stored on the View state are only then retrieved and used by the `StandListVM`.

Upload of the Files

Now that the `StandListVM` knows which local files the user wishes to upload, those can be sent to the API. Uploading a file is an asynchronous task, which allows multiple uploads to run in parallel. The user can monitor the progress of any individual task and cancel a given task during the uploading phase.

The `ApiDataService` holds items that are at the heart of the uploading process, they are called `CancellableItems`. These `structs` contain information about a file currently being uploaded; they are created, updated, and deleted by the `ApiDataService`. The set of `CancellableItems` is `@Published` so that the `StandListVM` can follow the progress of each ongoing task.

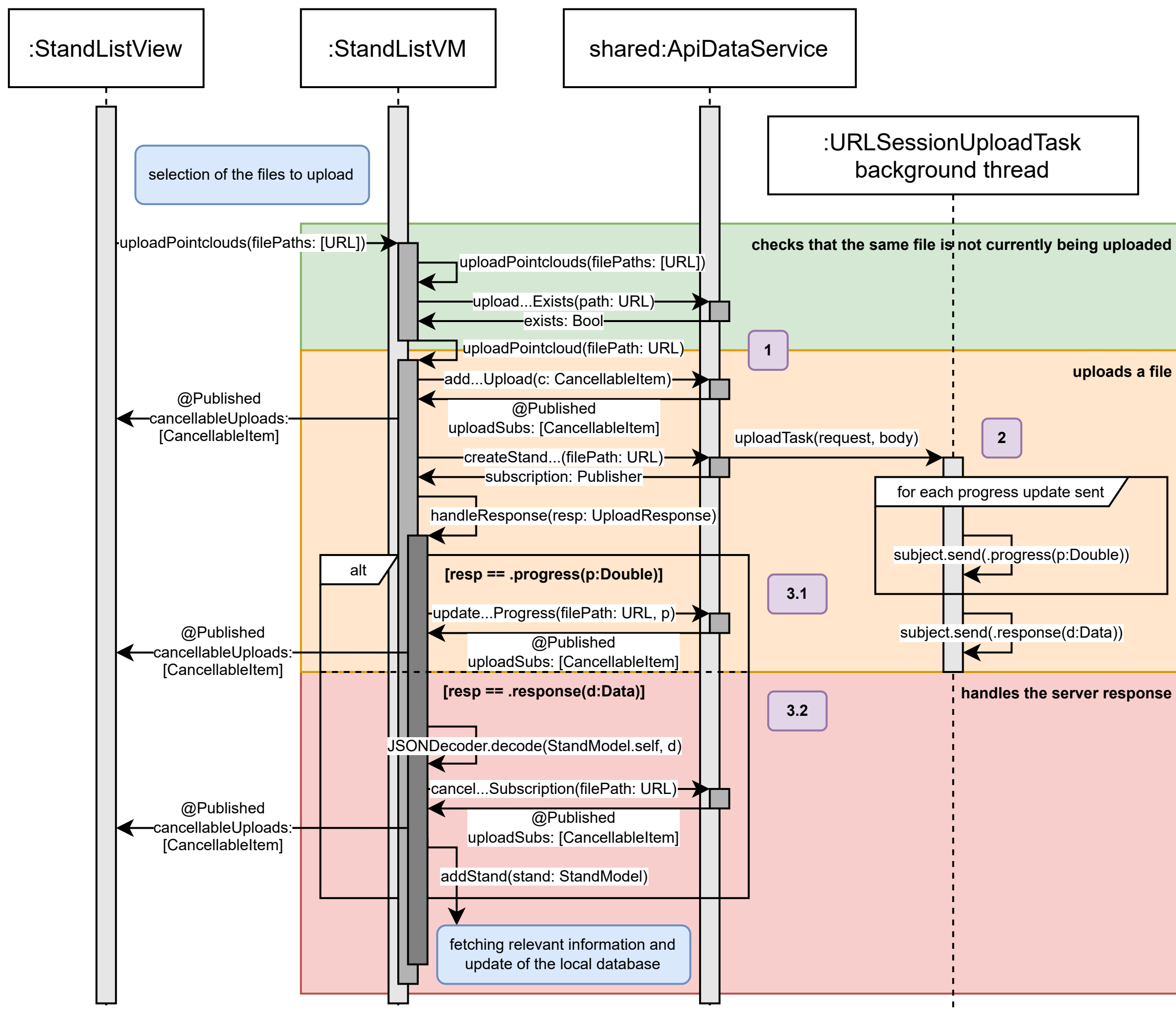


Figure 6.14: The upload of multiple pointcloud file to the API

This entire upload process is described in Figure 6.14. The ViewModel first compares the selected files to upload with the ongoing upload tasks in order not to upload the same point cloud twice at the same time **(1)**. For each file that passes that check, a `CancellableItem` is created and added to the set in the `ApiDataService`, the View-Model is notified of the change, which causes the View to update.

Although not displayed on this diagram, it is possible for a user to cancel an ongoing upload task by tapping on a `CancellableItem` displayed on the screen. The item is used to cancel the task; it is then removed from the published list of ongoing uploads. Once again, the ViewModel detects the change, which triggers the View to reload its content.

The ViewModel now calls the method responsible for the upload of the file. This results in the creation of an asynchronous upload task **(2)** that runs on a background thread. This task also sends `UploadResponses` that either inform on the `progress` of the upload, or contain the `response` data for the newly created stand.

This information is consumed by the ViewModel to update the `CancellableItem` and notify the user of the task progress **(3.1)**. Once a stand is fully uploaded, the backend eventually informs the client that the file has been fully analyzed and sends back the JSON data corresponding to the stand and all its computed measurements **(3.2)**. The ViewModel receives this data and transforms it into a `StandModel` for easier manipulation by the application. The stand upload subscription is finally removed from the list of ongoing uploads.

At this point, the stand list cannot be refreshed as the application only knows information about the stand itself. It still needs to fetch all the information tied to that stand (e.g. trees, diameters). This is explored in the following and last segment of this example.

Update of the Local Database

This section shows how the rest of the computed information is fetched and added to the local database. This operation involves a complex pipeline of `Publishers` that are all linked with one another. Indeed, a stand possesses one or more trees, but a tree also contains one or more captures, etc. This describes a tree-like structure where the stand is the root and the least dependent entities are the leaves:

```
Stands
    \----> Histories
    \----> Trees (leaf, returns true)
                \----> Captures
                            \----> Diameters (leaf, returns true)
```

The example of a full synchronization is illustrated in Figure 6.15. It essentially uses the same logic as for the addition of a stand, but deals with all available remote stands instead of a single one. It fetches each stand along with its related information in order to add them into the local database.

Figure 6.15: High-level view of a full local database synchronization

The synchronization happens in two stages. It first goes down the tree and fetches all needed information using the API.

The second stage goes up the tree to add all the mutual relationships between the retrieved entities and the corresponding parent entity. This is made easier with the use of the `reduce` function offered by `Combine` which acts as a barrier.

```
return Publishers
    .MergeMany(publishers)
    .reduce(true, { accumulator, isCurrOk in
        accumulator && isCurrOk
    })
    .eraseToAnyPublisher()
```

The call to reduce effectively turns a set of Publishers into a unique Publisher that either returns a boolean or an error. Thanks to this neat trick, the ViewModel only has to subscribe to the Publisher returned by the populateStands function to know if the synchronization was a success or if an error occurred and the information could not be saved locally.

6.3.5 Improvements

The first iterations of the application featured a **Dependency Injection** mechanism using the popular Swinject [70] package. It was later removed as it was hard to keep the mock and concrete implementations of services up-to-date as the application's structure evolved. Now that application's foundations are set, this concept should be re-introduced as it makes for easier testing of services and ViewModels.

The application could enforce an even **looser coupling** between its different layers. For instance, at this stage, Views and ViewModels directly use Core Data entities. This makes it impossible for a ViewModel to be tested without Core Data providing entities. The same applies to the Views that cannot be previewed without injecting Core Data specific entities.

These two improvements would facilitate the addition of **test suites** for the services, ViewModel. Indeed, each component could be tested in total isolation by interacting with mock implementations that would be injected as needed. Apple's XC Test framework [71] is a good candidate for the creation of such suite. It can also "interact with an application's UI to validate user interaction flows".

The code base would also benefit from **a more abstract and generic approach**. A good candidate for such refactoring would be the ApiDataService, where each entity has its own get, put, and delete methods. This leads to the repetition of similar looking

but different enough code throughout the service. A possible solution would be to have a generic `getResource` method supplied with the entity type.

In its current state, the application does not provide an all-in-one solution. The user has to use a third party application (see Chapter 4.1) to capture and export a forest stand point cloud file to use it with our solution. The application should be extended to feature a **custom made scanner module**.

6.3.6 Summary

The application makes life easier for the forest ranger, as they no longer need to carry cumbersome tools, manually measure each tree, and record the results in a spreadsheet. The user can access the stands, their trees, and all the relevant measurements that are stored on the remote server. Thanks to the application's local database, the user can still browse local copies when out in the wilderness.

The frontend solution attempts to distance itself from outdated, soon-to-be deprecated technologies. It uses a robust MVVM structure to enforce the separation of concern principle in order to maximize the testability and scalability of the solution. It also keeps the number of external dependencies to a bare minimum by only using frameworks provided by Apple. Both these decisions attempt to make the application as future-proof as possible. These choices make it relatively easy for a developer to understand the existing modules and further improve on the solution with extensions such as a custom camera module.

This section concludes the implementation of the client-server application. The following section covers the evaluation of the solution as a whole, including the methodology we followed, the results we obtained, and the discussions that followed.

Chapter 7

Final Evaluation and Discussion

This chapter assesses the entire solution developed throughout the last chapters using real-world data gathered manually in the field. Some evaluation has already occurred in previous chapters.

- Section 4.1 determines the precision of the GPS of the iPad Pro and analyses the 3D Scanner App settings to determine the optimal combination for DBH estimates.
- Section 5.3.5 compares the local coordinate system with two projected systems.
- Section 5.4 examines the algorithms in a broad sense in order to find the best algorithms, best setting combinations, resulting execution times, and so on.
- Section 6.2.6 describes the unit testing structure of the backend components.

In this chapter, we further evaluate the application. First, we assess the expected errors of the DBHs extracted by the application and discuss the underlying reasons in Section 7.1.

Next, we evaluate the time taken to carry out the process of capturing and processing a plot with the application with respect to an inventory using traditional manual methods in Section 7.2.

7.1 DBH Extraction

We assessed the accuracy and precision of the DBHs predicted by our application using the root mean square error (RMSE) and the coefficient of determination (R^2).

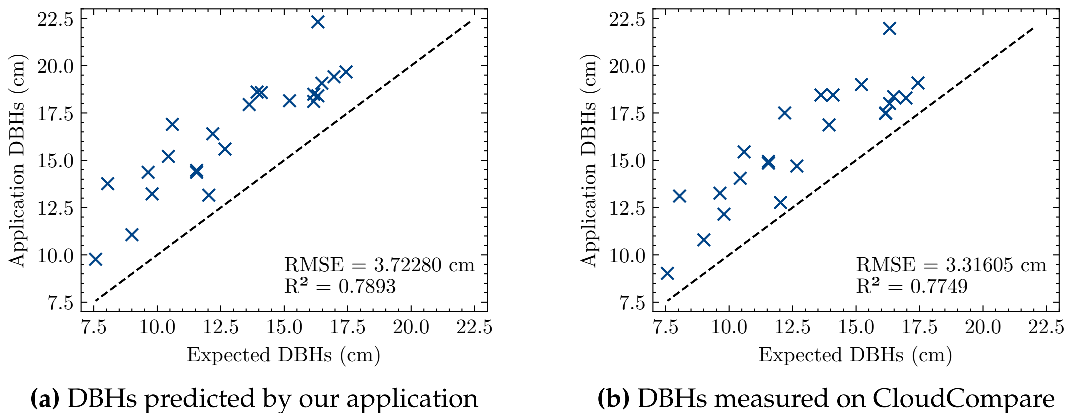
$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(e_i - p_i\right)^2} \tag{7.1}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(e_i - p_i\right)^2}{\sum_{i=1}^{n}\left(e_i - \bar{e}\right)^2} \tag{7.2}$$

where:

e = expected DBHs
p = predicted DBHs
$\bar{e}$ = mean of expected DBHs

The expected DBHs were manually measured with a diameter tape in the Bois de Lauzelle according to the methodology described in Section 3.1. The predicted DBHs are of two kinds: the DBHs predicted by our application (and its underlying algorithms) and the DBHs measured on CloudCompare with its point picking tool. The latter helps evaluate the performance of the application using a manual method to prevent any algorithmic influence.



(a) DBHs predicted by our application **(b)** DBHs measured on CloudCompare

Figure 7.1: DBHs retrieved from an iPad-generated LiDAR point cloud with a linear regression line showing the ideal relationship between predicted and expected DBHs

During the scans, we observed that the scanner had a tendency to inflate the trunks — see Figure 7.2. This is reflected in the RMSEs, as both our application and CloudCompare consistently overestimate the DBHs with RMSEs of 3.7228 and 3.31605 cm, respectively. The point cloud generated by the scanner is most likely the reason for these overestimates. Nevertheless, CloudCompare provides marginally better results, which indicates that the application (and its algorithms) has room for improvement.

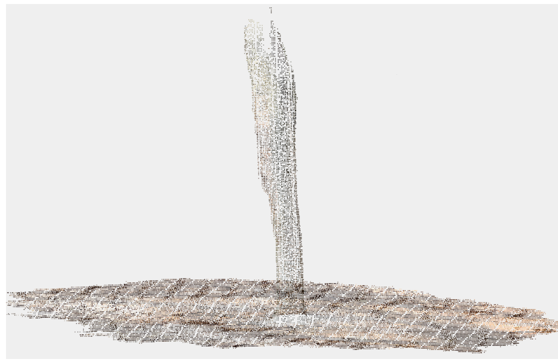


Figure 7.2: Example of a trunk being inflated by the scanning process

Similar results (RMSE of 3.64-3.76 cm) were observed in a related study [2] that evaluated the viability of the iPad LiDAR for forest inventory. They retrieved the DBHs using a circular cluster approach and a least-squares-based algorithm. In comparison, they achieved an RMSE of 1.59 cm with PLS (ZEB HORIZON) data, which costs 25.000€.

7.2 Performance & Walking Path

This section evaluates the performance of the application in terms of the time required to scan and process a plot of 23 trees compared to a manual approach. Scans used in the preceding chapters were completed by walking around each tree with the device, as detailed in Section 4.2. This section also evaluates a new walking path that involves scanning the plot row by row.

The comparisons are based on the assumption that both manual and automated approaches would be equally faster if carried out by an experienced professional.

Table 5.2 shows that, compared to the traditional manual approach, our solution cuts the total time in half. The additional time for the manual method is due to the fact

Method	Collection Time (min)	Additional Time (min)	Total Time (min)	Time Per Tree (min/tree)	Detection Rate (%)
Manual	20	5	25	1.087	100
Automated (tree-by-tree)	11.5	1.5	13	0.565	100
Automated (row-by-row)	4.65	1.5	6.15	0.267	78

Table 7.1: Comparison of the time taken by different approaches to inventory a 23-tree plot

that we missed 3 trees, which required us to double-check which ones were missing. The automated methods do not have this problem as the scanning application gives an overview of what has been scanned on the fly. On the other hand, the automated approaches require an additional time of about 90 seconds, which includes the upload of the point cloud over the cellular network and its processing.

The row-by-row walking method significantly improved the collection time. However, several trees were not detected. This is because the algorithms fail to fit circles to detect trees, as this approach results in more missing points in the trunk contours. Further work to refine the workflow for this walking path may be worthwhile as the speedup seems promising.

7.3 Summary

The evaluation of our application comes to a close in this section. The DBH error (RMSE) is 3.72 cm (DBHs are constantly overestimated), the detection rate is 100%, and the application allows inventory time to be cut in half for the surveyed plot.

Although promising, we believe that the overall evaluation is insufficient to draw definitive conclusions regarding the application. The application has to be tested on a larger scale in order to find flaws that we overlooked and gain a better grasp of its limitations. However, this preliminary assessment yields encouraging results that are consistent with previous research, indicating that the application provides a solid starting point for further research.

Chapter 8

Conclusions and Future Work

In light of concerns such as global warming and the biodiversity crisis, the demand for forest inventory tools has never been greater. The objective of this thesis was to advance the state-of-the-art in automatic forest inventory. In particular, we focused on developing a low-cost application for the iOS platform that takes advantage of the point clouds generated by the new LiDAR sensor on the 2020 iPad Pro. The application is capable of estimating typical forest attributes such as diameters and plot area, as well as offering digital storage of such data. The application also provides foresters and regular users with user-friendly interfaces. Along with this application, we introduced and evaluated a fully automatic workflow for processing geo-referenced point clouds, which was specifically tailored for the needs and limitations of the iPad LiDAR.

The application and the workflow were evaluated based on reference data acquired manually in the Bois de Lauzelle in Belgium. The workflow comprising of algorithms and methods such as voxelization, Hough Transform, and RANSAC yields promising results, provided that the proper parameters are selected. We were able to achieve a detection rate of 100%, a root-mean-square error of the diameter at breast height of 3.72 cm, and to cut the collection time of traditional (manual) methods in half. Furthermore, despite the GPS inaccuracy of 2.50 m, trees can be geo-located thanks to a point matching algorithm. Other studies observed similar results with semi-manual workflows. This thesis shows that a fully-automatic end-to-end application can achieve comparable results.

There is of course room for future work:

- A specialized scanner module would provide more accurate estimates. It would enable the estimation of more tree attributes such as height, volume, species, and above-ground biomass. Combining LiDAR data and image data to determine the species may yield improved results [72,73].
- Some refinements to the workflow could be made by evaluating novel algorithms or further optimizing the parameter choice. The server application facilitates such evaluation thanks to its customizability.
- The solution, despite promising results, requires more in-field evaluation in different forest types (other tree species, densities, branches and vegetation at low height...) to further evaluate the accuracy of the measurements it provides.
- The solution could be adapted for collaborative scanning of large trees. The server could be kept as is, however the client application should feature a two-way synchronization between the server and several clients.

8.1 Final Words

Despite a complete and functional final product, we encountered many difficulties. They were however all overcome, notably by reading research papers and documentation, and by experimenting through trial-and-error. A certain rigor, curiosity, and perseverance were all necessary for the success of this project.

The thesis allowed us to immerse ourselves in a complex yet fascinating domain. General concepts of the computer science world were explored, whether it was about web architectures, user interfaces, coordinate systems standards, and algorithms. Our skills grew along the way. We learned to weigh the benefits and drawbacks of different architectures and algorithms to identify the best solution for our needs.

In conclusion, this thesis achieved its intended purpose. We are very satisfied to have achieved a consistent, functional, and interesting solution. We believe that the application highlights the potential of the iPad LiDAR sensor for the forestry industry. The low cost of the solution and its compelling results have opened the door for a wider audience. The application requires more work, but we also believe it is a good starting point for future research in the field.

Bibliography

[1] Xiaoling Wang, Arunima Singh, Yelyzaveta Pervysheva, K. Lamatungga, Veronika Murtinová, Mohammad Mukarram, Q. Zhu, Kun Song, Peter Surovy, and Martin Mokros. Evaluation of iPad Pro 2020 LiIDAR for estimating tree diameters in urban forest. *ISPRS - International Archives of the Photogrammetry Remote Sensing and Spatial Information Sciences*, VIII-4/W1-2021:105–110, 09 2021.

[2] Christoph Gollob, Tim Ritter, Ralf Kraßnitzer, Andreas Tockner, and Arne Nothdurft. Measurement of forest inventory parameters with Apple iPad Pro and integrated LiDAR technology. *Remote Sensing*, 13(16), 2021.

[3] Fangyi Wang, Muditha K. Heenkenda, and Jason T. Freeburn. Estimating tree diameter at breast height (DBH) using an iPad Pro LiDAR sensor. *Remote Sensing Letters*, 13(6):568–578, 2022.

[4] Joanne C. White, Christoph Stepper, Piotr Tompalski, Nicholas C. Coops, and Michael A. Wulder. Comparing ALS and image-based point cloud metrics and modelled forest inventory attributes in a complex coastal forest environment. *Forests*, 6(10):3704–3732, 2015.

[5] Joanne C. White, Michael A. Wulder, Mikko Vastaranta, Nicholas C. Coops, Doug Pitt, and Murray Woods. The utility of image-based point clouds for forest inventory: A comparison with airborne laser scanning. *Forests*, 4(3):518–536, 2013.

[6] Tristan R.H. Goodbody, Nicholas C. Coops, Peter L. Marshall, Piotr Tompalski, and Patrick Crawford. Unmanned aerial systems for precision forest inventory purposes: A review and case study. *The Forestry Chronicle*, 93(01):71–81, 2017.

[7] Tianyu Hu, Xiliang Sun, Yanjun Su, Hongcan Guan, Qianhui Sun, Maggi Kelly, and Qinghua Guo. Development and performance evaluation of a very low-cost UAV-lidar system for forestry applications. *Remote Sensing*, 13(1), 2021.

[8] L Monika Moskal and Guang Zheng. Retrieving forest inventory variables with

terrestrial laser scanning (TLS) in urban heterogeneous forest. *Remote Sensing*, 4(1):1–20, 2011.

[9] Sébastien Bauwens, Harm Bartholomeus, Kim Calders, and Philippe Lejeune. Forest inventory with terrestrial LiDAR: A comparison of static and hand-held mobile laser scanning. *Forests*, 7(6):127, 2016.

[10] Maria Immacolata Marzulli, Pasi Raumonen, Roberto Greco, Manuela Persia, and Patrizia Tartarino. Estimating tree stem diameters and volume from smartphone photogrammetric point clouds. *Forestry: An International Journal of Forest Research*, 93(3):411–429, 12 2019.

[11] G. R. van der Werf, D. C. Morton, R. S. DeFries, J. G. J. Olivier, P. S. Kasibhatla, R. B. Jackson, G. J. Collatz, and J. T. Randerson. CO2 emissions from forest loss. *Nature Geoscience*, 2(11):737–738, November 2009.

[12] Miki Fukuda, Toshiro Iehara, and Mitsuo Matsumoto. Carbon stock estimates for sugi and hinoki forests in Japan. *Forest ecology and management*, 184(1-3):1–16, 2003.

[13] John Kauffman, Virni Arifanti, Imam Basuki, Sofyan Kurnianto, Nisa Novita, Daniel Murdiyarso, Daniel Donato, and Matthew Warren. Protocols for the measurement, monitoring, and reporting of structure, biomass, carbon stocks and greenhouse gas emissions in tropical peat swamp forests. 12 2016.

[14] John A Kershaw Jr, Mark J Ducey, Thomas W Beers, and Bertram Husch. *Forest mensuration*. John Wiley & Sons, 2016.

[15] James G Dickson. *Wildlife of southern forests: habitat and management*. Surrey, BC: Hancock House Publishers, 2001.

[16] Joanne White, Michael Wulder, Andrés Varhola, Mikko Vastaranta, Nicholas Coops, Bruce Cook, Doug Pitt, and Murray Woods. A best practices guide for generating forest inventory attributes from airborne laser scanning data using an area-based approach. *Canadian Forest Service, Information Report*, FI-X:50, 07 2013.

[17] Thomas Eugene Avery and Harold E. Burkhart. *Forest measurements*. McGraw-Hill series in forest resources. McGraw-Hill, Boston, 5th ed edition, 2002.

[18] Joanne White, Michael Wulder, Andrés Varhola, Mikko Vastaranta, Nicholas Coops, Bruce Cook, Doug Pitt, and Murray Woods. A best practices guide for

generating forest inventory attributes from airborne laser scanning data using an area-based approach. *Canadian Forest Service, Information Report*, FI-X:50, 07 2013.

[19] David R Larsen. Simple taper: Taper equations for the field forester. In *In: Kabrick, John M.; Dey, Daniel C.; Knapp, Benjamin O.; Larsen, David R.; Shifley, Stephen R.; Stelzer, Henry E., eds. Proceedings of the 20th Central Hardwood Forest Conference; 2016 March 28-April 1; Columbia, MO. General Technical Report NRS-P-167. Newtown Square, PA: US Department of Agriculture, Forest Service, Northern Research Station: 265-278.*, pages 265–278, 2017.

[20] Mahadev Sharma and John Parton. Height–diameter equations for boreal tree species in ontario using a mixed-effects modeling approach. *Forest Ecology and Management*, 249:187–198, 09 2007.

[21] Roscinto Ian C. Lumbres, Young Jin Lee, Chung Weon Yun, Chang Duck Koo, Se Bin Kim, Yeong Mo Son, Kyeong Hak Lee, Hyun Kyu Won, Sung Cheol Jung, and Yeon Ok Seo. DBH-height modeling and validation foriacacia mangium/iandieucalyptus pellita/iin korintiga hutani plantation, kalimantan, indonesia. *Forest Science and Technology*, 11(3):119–125, March 2015.

[22] Erik Næsset, Terje Gobakken, Johan Holmgren, Hannu Hyyppä, Juha Hyyppä, Matti Maltamo, Mats Nilsson, Håkan Olsson, Åsa Persson, and Ulf Söderman. Laser scanning of forest resources: the nordic experience. *Scandinavian Journal of Forest Research*, 19(6):482–499, 2004.

[23] Todd J Hawbaker, Nicholas S Keuler, Adrian A Lesak, Terje Gobakken, Kirk Contrucci, and Volker C Radeloff. Improved estimates of forest vegetation structure and biomass with a lidar-optimized sampling design. *Journal of Geophysical Research: Biogeosciences*, 114(G2), 2009.

[24] Murray Woods, Doug Pitt, Margaret Penner, Kevin Lim, Dave Nesbitt, Dave Etheridge, and Paul Treitz. Operational implementation of a lidar inventory in boreal ontario. *The Forestry Chronicle*, 87(4):512–528, 2011.

[25] GL Baskerville. Use of logarithmic regression in the estimation of plant biomass. *Canadian Journal of Forest Research*, 2(1):49–53, 1972.

[26] Erik Næsset and Tonje Økland. Estimating tree height and tree crown properties using airborne scanning laser in a boreal nature reserve. *Remote Sensing of Environment*, 79(1):105–115, 2002.

[27] Ye Seul Lim, Phu Hien La, Jong Soo Park, Mi Hee Lee, Mu Wook Pyeon, and Jee-In Kim. Calculation of tree height and canopy crown from drone images using segmentation. *Journal of the Korean Society of Surveying, Geodesy, Photogrammetry and Cartography*, 33(6):605–614, 2015.

[28] © 2022 Laan Labs. 3D Scanner App — LIDAR scanner for iPad iPhone Pro. https://3dscannerapp.com/.

[29] © 2022 Apple Inc. About privacy and location services in iOS and iPadOS — Apple support. https://support.apple.com/en-us/HT203033.

[30] © 2022 Apple Inc. Core location — Apple developer documentation. https://developer.apple.com/documentation/corelocation.

[31] IGN L'Institut géographique national. G-doc - documentation geodésique. https://gdoc.ngi.be/.

[32] © 2022 Apple Inc. Apple unveils new iPad Pro with LiDAR scanner and trackpad support in iPadOS - Apple. https://www.apple.com/newsroom/2020/03/apple-unveils-new-ipad-pro-with-lidar-scanner-and-trackpad-support-in-ipados/.

[33] © 2022 iFixit — Licensed under Creative Commons. 12.9" iPad Pro 2020 teardown: What does the LiDAR scanner look like?

[34] Daniel Huber. The ASTM E57 file format for 3D imaging data exchange. In *Three-Dimensional Imaging, Interaction, and Measurement*, volume 7864, page 78640A. International society for Optics and Photonics, 2011.

[35] Kenton McHenry and Peter Bajcsy. An overview of 3d data content, file formats and viewers. *National Center for Supercomputing Applications*, 1205:22, 2008.

[36] ASPRS. LAS specification version 1.2. The American Society for Photogrammetry & Remote Sensing Bethesda, MD, USA, 2008.

[37] Martin Isenburg. LASzip: lossless compression of LiDAR data. *Photogrammetric engineering and remote sensing*, 79(2):209–217, 2013.

[38] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700, 1987.

[39] Yongxiang Fan, Zhongke Feng, Abdul Mannan, Tauheed Ullah Khan, Chaoyong

Shen, and Sajjad Saeed. Estimating tree position, diameter at breast height, and tree height in real-time using a mobile phone with RGB-D SLAM. *Remote. Sens.*, 10(11):1845, 2018.

[40] Martin Mokros, Tomás Mikita, Arunima Singh, Julián Tomastík, Juliána Chudá, Piotr Wezyk, Karel Kuzelka, Peter Surový, Martin Klimánek, Karolina Zieba-Kulawik, Rogerio Bobrowski, and Xinlian Liang. Novel low-cost mobile mapping systems for forest inventories as terrestrial laser scanning alternatives. *Int. J. Appl. Earth Obs. Geoinformation*, 104:102512, 2021.

[41] Christoph Gollob, Tim Ritter, Ralf Kraßnitzer, Andreas Tockner, and Arne Nothdurft. Measurement of forest inventory parameters with Apple iPad Pro and integrated LiDAR technology. *Remote. Sens.*, 13(16):3129, 2021.

[42] Julián Tomastík, Simon Salon, Daniel Tunák, Frantisek Chudý, and Miroslav Kardos. Tango in forests - an initial experience of the use of the new google technology in connection with forest inventory tasks. *Comput. Electron. Agric.*, 141:109–117, 2017.

[43] Geomatics Guidance Note. 7, part 2: Coordinate conversions & transformations including formulas, 2019.

[44] PROJ contributors. PROJ coordinate transformation software library, 2022. https://proj.org/.

[45] Wuming Zhang, Jianbo Qi, Peng Wan, Hongtao Wang, Donghui Xie, Xiaoyan Wang, and Guangjian Yan. An easy-to-use airborne LiDAR data filtering method based on cloth simulation. *Remote Sensing*, 8(6), 2016.

[46] Biao Xiong, Weize Jiang, Dengke Li, and Man Qi. Voxel grid-based fast registration of terrestrial point cloud. *Remote. Sens.*, 13(10):1905, 2021.

[47] Tiago de Conto, Kenneth Olofsson, Eric Bastos Görgens, Luiz Carlos Estraviz Rodriguez, and Gustavo Almeida. Performance of stem denoising and stem modelling algorithms on single tree point clouds from terrestrial laser scanning. *Comput. Electron. Agric.*, 143:165–176, 2017.

[48] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[49] Kenneth Olofsson, Johan Holmgren, and Håkan Olsson. Tree stem and height

measurements using terrestrial laser scanning and the RANSAC algorithm. *Remote. Sens.*, 6(5):4323–4344, 2014.

[50] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[51] MB Wright. Speeding up the hungarian algorithm. *Computers & Operations Research*, 17(1):95–96, 1990.

[52] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[53] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[54] PDAL Contributors. PDAL point data abstraction library, August 2020.

[55] Jean-Romain Roussel, David Auty, Nicholas C. Coops, Piotr Tompalski, Tristan R.H. Goodbody, Andrew Sánchez Meador, Jean-François Bourdon, Florian de Boissieu, and Alexis Achim. lidR: An R package for analysis of airborne laser scanning (ALS) data. *Remote Sensing of Environment*, 251:112061, 2020.

[56] Tiago de Conto. TreeLS: Terrestrial point cloud processing of forest data, 2020. R package version 2.0.4, https://github.com/tiagodc/TreeLS.

[57] Martin Isenburg. LAStools — efficient LiDAR processing software. http://rapidlasso.com/LAStools.

[58] Daniel Girardeau-Montaut. CloudCompare — open source project, 2022. Version 2.12, http://www.cloudcompare.org/.

[59] Liang-Jie Zhang. SOA solution reference architecture. In *2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA*. IEEE Computer Society, 2007.

[60] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[61] Apple Inc Core Image documentation. Core image, 2022. https://developer.apple.com/documentation/coreimage.

[62] Apple Inc Core Motion documentation. Core motion, 2022. https://developer.apple.com/documentation/coremotion.

[63] Apple Inc RealityKit documentation. RealityKit, 2022. https://developer.apple.com/documentation/realitykit.

[64] Artem Syromiatnikov and Danny Weyns. A journey through the land of model-view-design patterns. In *2014 IEEE/IFIP Conference on Software Architecture, WICSA 2014, Sydney, Australia, April 7-11, 2014*, pages 21–30. IEEE Computer Society, 2014.

[65] Piotr Wiertel and Maria Skublewska-Paszkowska. Comparative analysis of UIKit and SwiftUI frameworks in iOS system. *Journal of Computer Sciences Institute*, 20:170–174, Sep. 2021.

[66] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In Jens Knoop and Uwe Zdun, editors, *Software Engineering 2016, Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. Februar 2016, Wien, Österreich*, volume P-252 of *LNI*, pages 69–70. GI, 2016.

[67] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. 2010.

[68] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013.

[69] Richard D Hipp. SQLite, 2022. https://www.sqlite.org.

[70] Yoichi Tagaya Jakub Vano and 39 contributors. Swinject, 2022. https://github.com/Swinject/Swinject.

[71] Apple Inc XCTest documentation. Xctest, 2022. https://developer.apple.com/documentation/xctest.

[72] Nicholas R Vaughn, L Monika Moskal, and Eric C Turnblom. Tree species detection accuracies using discrete point lidar and airborne waveform lidar. *Remote sensing*, 4(2):377–403, 2012.

[73] Mohd Iz'aan Paiz Zamri, Florian Cordova, Anis Salwa Mohd Khairuddin, Norrima Mokhtar, and Rubiyah Yusof. Tree species classification based on image analysis using improved-basic gray level aura matrix. *Computers and Electronics in Agriculture*, 124:227–233, 2016.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**