Chronocity

Technical Report Towards an Open Point Cloud Map supporting on-the-fly change detection

Barbara Cemellini
Willem van Opstal
Cheng-Kai Wang
Dimitris Xenakis

TU Delft

Chronocity

Technical Report Towards an Open Point Cloud Map supporting on-the-fly change detection

by

Barbara Cemellini
Willem van Opstal
Cheng-Kai Wang
Dimitris Xenakis

GEOMATICS SYNTHESIS PROJECT 2017

Date:	June 21st 2017	
Supervisors:	Stefan van der Spek	TU Delft, Director GSP
	Peter van Oosterom	TU Delft, Team coordinator
	Wilko Quak	TU Delft, Team coach
	Stella Psomadaki	Fugro, Team coach

Preface

We are now gradually entering the era of *big data* - maybe a bit too much of a buzzword, but it is not lied. Technology is evolving fast, enabling faster and more efficient data acquisition, storage, retrieval and processing. Point cloud datasets are such a type which relies on extremely large files and lots of processing power. The rather fast evolutions in technology enable the shared idea between Delft University of Technology and Fugro of a so-called 'Open Point Cloud Map' (OPCM). This Open Point Cloud Map aims at making point cloud datasets easily available to the public, even letting them perform simple analysis. Both Fugro and TU Delft want to take lead in development of such an environment; three student teams from TU Delft thus form a partnership with Fugro to kick-off three in-depth researches which would result in one step closer to the vision of OPCM.

The *Chronocity* project is part of this tripartite synergy among the three student teams. The combined goal is to improve accessibility and availability of (massive) point cloud data in the most effective and efficient way. *Chronocity* will put focus on the time-component in these large datasets, where the other two teams respectively focus on a) combining data from different sources and exploring the ideal coordinate reference system, and b) examining the means and uses of different scale and granularities in different datasets.

The ChronoCity research team consists of four Master students in the field of Geomatics for the Built Environment. The varied backgrounds (Geoinformatics, Urban Planning, Geosciences and Industrial Design) enable a well-defined problem statement covering all sorts of aspects towards the OPCM which will be covered more completely in the final research and product.

All this takes place within the the framework of the *Geomatics Synthesis Project* 2017 at Delft University of Technology. It is the final course in the first year's study program and aims at combining all gained knowledge throughout the year. The project runs for ten weeks in small teams of up to six students. Not only actual outcomes are important, still the learning curve and personal development is of significant importance.

List of Figures

1.1 Rich picture, project process and contexts. 4

2.1 Viewport of the Esri AHN viewer, visually comparing AHN2 to AHN3. 6
2.2 OPCM is a long term vision covering many aspects, while ChronoCity solely focuses on a portion of it to step further towards this vision. 6

3.1 Top-level requirements, setup in Moscow-rules (partly taken from DID-B2). 10
3.2 New data appears if e.g. a bridge is built over a canal (previously no-return scans). 12
3.3 Classification changes without significant geometric change. 12
3.4 Simple roof extension, detectable by the algorithm. 13
3.5 Partly demolished building and then extended with a somewhat higher extension. 13
3.6 (irregular) 3D change of vegetation during a (sometimes very) short period of time. 14

4.1 Modularity for project implementation management. User has control over the orange parts, where blue parts mainly take place in the background. 15
4.2 User interaction flow diagram; blue paralleloids are so-called black boxes. These processes take place in the background. 16
4.3 The proposed 3D change detection approaches and their theoretical workflow. The thickness of the arrow indicates the priority of implementation. 18
4.4 The density function compares the number of points in AHN2 and AHN3 in the same voxel unit. Even the point resolution of two datasets may be different, the change in density function value can still indicate the positive or negative changes. 19
4.5 Geometric comparison: Euclidean distance comparison, adapted from [20]. 20
4.6 Nearest neighbour distance algorithm. Image source: [5] 20
4.7 Changes of the compared points are highlighted in green color in CloudCompare. 21
4.8 Change vector can be used to indicate the change direction. 21
4.9 Offsets and errors are unavoidable in the measurements. However, inappropriate distance threshold could ignore the real changes, as shown in the blue dash line above. 22
4.10 Octree neighbor search. Image source: [19] 22
4.11 The overall workflow of implementing three change detection approaches. 23
4.12 User interaction with ChronoCity application through its interface. 25
4.13 Example of a request procedure for a specific point cloud change. 26
4.14 Different LOD's showed on a loaded point cloud (region in Leiden). 27
4.15 How point resolution per node is changed at different depth levels. [23] 28
4.16 In this snapshot, a volume clipper has been used to select part of a building in Delft (Source data: AHN3). 29
4.17 In this snapshot, a height profiler has been used on the construction of the TU Delft Library (Source data: AHN3). 30
4.18 A point cloud being projected within iTowns, on top of street images. 30

5.1 One of the first sketches for the user interface. This is an envisioned framework which plays the starting point for implementation in *HTLM* and*CSS*. 34
5.2 Resulting interactive user interface for the Chronocity-application. It shows the viewport on the right and the sidebar on the left with the change-detection module activated. 35
5.3 Schematic system infrastructure with most important files-structure on the server-side. 35
5.4 Client-server communication, initiated by user tasks. 37
5.5 Schematic flow for the change detection process, including both the HTTP request and response as well as the (temporary) files. 38
5.6 Screenshot of the resulting view when the change detection is finished. 39

6.1 In the left side kd-tree decomposition, the nearest neighbor is across a hyperplane from the query point, which means additional backtracking is needed, but in the same cell in another decomposition as shown on the right side. Image source: [14] 43
6.2 Projection of different branching factor K of the different decompositions of the space. K = 4, 32, 128 respectively. Image source: [14] 43
6.3 When t1 and t2 points are mostly overlapped, most of the distance calculations are done within a single initial grid since the closest neighbor distance is less than the defined threshold; If no neighbor exists in the current grid(s), the search is expanding to the adjacent girds and only stop till the nearest neighbor is found. 44
6.4 TIn this example, true nearest neighbor is not within the initial search grid (medium gray). Therefore, the initial search area of the query point is set to be 3x3x3 grids and expand to the next level if no neighbor is founded. 44
6.5 FLANN performance test with different threshold distances 45
6.6 Indexed FLANN performance test with different grid sizes. 45
6.7 The performance of grid nearest neighbor algorithm in general is much faster than FLANN and indexed FLANN. This difference is more significant in even larger datasets. However, grid NN is comparatively sensitive to the threshold setting and becomes dramatically slow with a small threshold distance. 46
6.8 Green represents new points and red for the removed points. For simplicity, only the detected change points are shown. 46
6.9 To obtain the generic geometrical changes of two point clouds, both new points and removed points will be detected in the algorithm. New ground points on the footprint of the demolished building shows the tricky case of nearest neighbor change detection. 47
6.10 Green represents new points (t2 - t1), and false positive changes are detected on the rooftop of the Faculty of Architecture, TU Delft. The threshold is set to be 1m for clearness. 48

7.1 Search and Go tab. 49
7.2 Example of clipping area. 50
7.3 'Share your view' tab. 50
7.4 'Download and Export' tab. 51
7.5 Using Chronocity application to identify tree growth zone near the outskirt of city of Delft, Netherlands 52
7.6 Changes in a forest before (above) and after (below) a fire. Image source: [30] 53
7.7 Example of coastal erosion. Image source: [12] 53

Contents

List of Figures **v**

1 Introduction **3**

2 Problem statement **5**
- 2.1 Vision: Open Point Cloud Map . . . 5
- 2.2 ChronoCity project . . . 5
- 2.3 Research questions . . . 7

3 Requirements **9**
- 3.1 General . . . 9
- 3.2 User aspects . . . 9
- 3.3 3D change detection . . . 11
- 3.4 Data model and storage . . . 12

4 Conceptual analysis **15**
- 4.1 Overview . . . 15
- 4.2 User aspects . . . 16
- 4.3 3D change detection . . . 17
  - 4.3.1 Theory . . . 17
  - 4.3.2 Implementation . . . 23
- 4.4 Online viewer and data model . . . 24
  - 4.4.1 Ideal scenario . . . 24
  - 4.4.2 State of the art solutions . . . 26
  - 4.4.3 Implementation . . . 30

5 System architecture **33**
- 5.1 Modularity . . . 33
- 5.2 Online viewer . . . 33
- 5.3 File and data structure . . . 34
- 5.4 Task flow . . . 36

6 3D change detection **41**
- 6.1 Methodology . . . 41
  - 6.1.1 Euclidean nearest neighbor . . . 41
  - 6.1.2 Fast approximate nearest neighbor search . . . 42
  - 6.1.3 Grid nearest neighbor search . . . 43
- 6.2 Performance analysis . . . 44
- 6.3 Reliability analysis and identifiable changes . . . 46

7 Functionalities and use cases **49**
- 7.1 Viewer functionalities . . . 49
- 7.2 Potential use cases . . . 50
  - 7.2.1 Urban planning use cases . . . 51
  - 7.2.2 Environmental - engineering use cases . . . 52

8 Conclusions **55**

9 Recommendations **57**

Bibliography **61**

A Media outreach strategy **63**

B Project planning **65**

C Performance test **67**

List of symbols and abbreviations

2D	2 dimensional
3D	3 dimensional
AHN	“Actuele Hoogtekaart Nederland”
BSD	Berkeley Software Distribution
CRS	Coordinate Reference System
CSS	Cascading Style Sheet
DBMS	Database Management System
DBSCAN	Density-based spatial clustering of applications with noise
ECC	Error-correcting code
GSP	Geomatics Synthesis Project
HTML	Hypertext Markup language
JS	Javascript
KVP	Key-Value pair
LAS	Storage format for LiDAR point clouds
LAZ	Compressed version of LAS
LiDAR	LIght Detection And Ranging
LOD	Level of detail
LOPoCS	Light Opensource PointCloud Server
MNO	Modifiable nested octree structure
NN	Nearest Neighbor
OPCM	Open Point Cloud Map
PCL	Point Cloud Library
RoI	Region of interest
VTK	Visualization toolkit
WebGL	Web Graphics Library

1 Introduction

We are now gradually entering the era of *big data* - maybe a bit too much of a buzzword, but it is not lied. Technology is evolving fast, enabling faster and more efficient data acquisition, storage, retrieval and processing. Point cloud datasets are such a type which relies on large files and lots of processing power. The rather fast evolutions in technology enable the shared idea between Delft University of Technology and Fugro of an 'Open Point Cloud Map'. This Open Point Cloud Map aims at making point cloud datasets easily available to the public, even letting them perform simple analysis. Both Fugro and TU Delft want to take lead in development of such an environment; three student teams from TU Delft thus form a partnership with Fugro to kick-off three in-depth researches which would result in one step closer to the vision of OPCM. The ChronoCity team will focus on the time-component (Figure 1.1) in the acquainted point clouds, while the other two teams focus on location-dependency and different scales and granularities of the datasets.

The *Chronocity*-team strives to create an online interactive tool which gives the user the ability to view, explore and analyze massive point cloud datasets on-the-fly. Since the limited timespan in which the project should take place this would not yield a fully optimized application, but at least the general principles are defined and evaluated on for a more defined future in the development of the OPCM. A large portion of the efforts will go into making the data and analyses available to the public - in an interactive and user-friendly way; because without this availability, the underlying principles are not brought to the public also. Regarding these underlying principles the most important one is change detection. During the project a suitable algorithm is designed and evaluated for detecting new, removed and changed geometric points.

Reading guide

This report concludes the entire Chronocity-project and its considerations. Since the main goal of this project is creating a software packacge, this report only includes theoretical background. The actual software is made available trough *github.com/OPCM*.

The report starts with defining the current situation on massive point cloud visualization and analyses in chapter 2. Within the same chapter also an envisioning of the future OPCM usage and features is given on which multiple research questions are proposed to conclude this section. To grasp this vision and the associated research and make it useful in the ongoing process towards the OPCM, in chapter 3 the top-level requirements are defined. This chapter forms the basis and indicates the goal of the final deliverables. Then in chapter 4 multiple alternatives are explored in terms of usability, change detection, data models and online frameworks which are then evaluated on basis of the top-level requirements. With the outcomes of this conceptual analysis the system architecture is designed in chapter 5. It explains on what is actually implemented in terms of tools, task flow and communications between several modules. Because of the significant role and technical nature of the change detection algorithm this is elaborated on its own in chapter 6. This will guide you through the development of the algorithm, but also shows some results and drawbacks of the used methodology. To conclude implementation of the *Chronocity* application some example use cases are given in chapter 7 with respect to different kind of users and workfields. Development of the OPCM is likely to never end, but our project does. Therefore this report is concluded with recommendations on improving the system as it is now, but also hint towards future extensions and features.

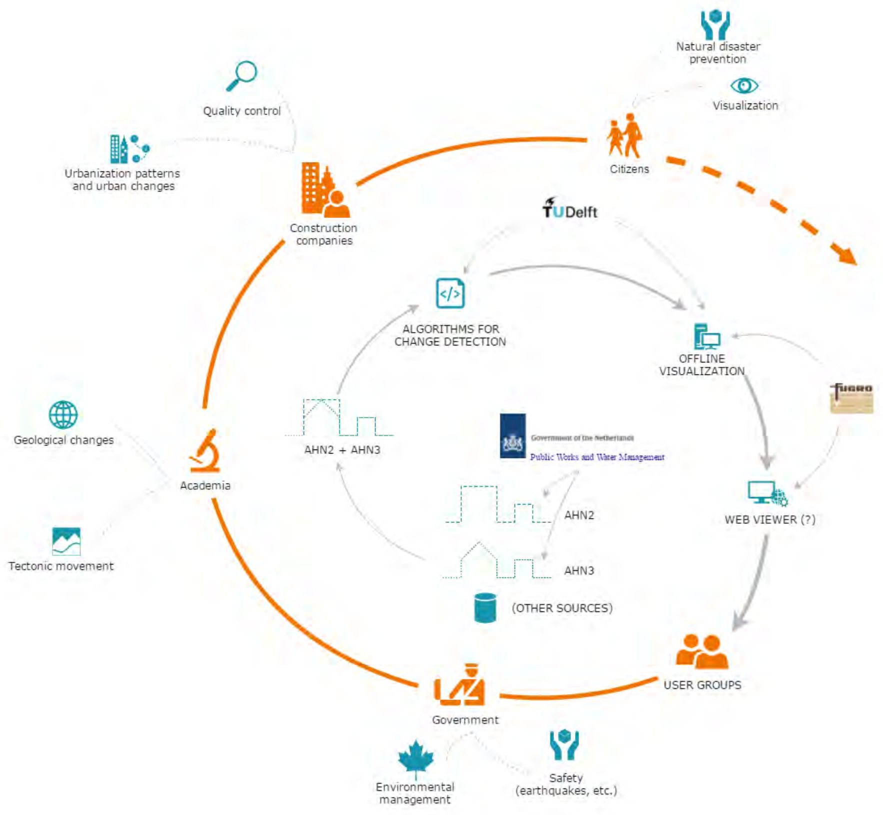


Figure 1.1: Rich picture, project process and contexts.

2 Problem statement

This chapter elaborates on background information of the topic; setting up a vision on how the OPCM will ultimately look like - and thus defining to which specifications our project should be able to adapt, but also into which extent the other synthesis groups cover certain parts. Next stage is looking into the current situation around the theme: what has already been done, and what are currently the main challenges. The section concludes with our research scope in terms of proposed research questions.

2.1. Vision: Open Point Cloud Map

The vision of the OPCM aims at making point cloud data accessible everywhere and allows the users to do some predefined or custom analysis on the data itself. This analysis mainly focuses on the aspects of time (change detection), scale and granularity, and location (different point clouds from different data sources). Nowadays, most of the viewers available are working fine for visualization purposes, but they lack real 'understanding' of the data shown. Human beings are able to perfectly interpret the changes between point cloud data (Figure 2.1); however, in order to be of actual use for further analysis and processing, the data should be machine identifiable and pinpointed in the interactive viewer.

The OPCM is envisioned as an open online platform freely available to everyone, everywhere in the world, through an internet connection. In the first stages of development, the provided data will mainly come from commercial parties. However, when technology will be further developed, resulting in good and affordable point cloud sensors, we expect the platform to be extended to host also crowdsourced data directly. In other words, the final version of the OPCM is very ambitious and allows different kind of users to upload their own data and compare them to the already existing ones. In this way, everyone can contribute to the realisation of the OPCM to create a big point cloud map of the whole world.

2.2. ChronoCity project

The ChronoCity project can be considered as a portion or a starting point for the implementation of the Open Point Cloud Map. The project is run in parallel with the location and scale/granularity groups, which together are supposed to lay the basis for the OPCM.

It is important to specify that the ChronoCity project has to be developed in a short amount of time, therefore the final result will not meet all the requirements of the OPCM vision. Basically, the aim of our project is to design and implement an efficient algorithm to detect geometrical changes between point clouds and visualise them on an online viewer. But, the actual challenge is the combination of the two parts. Today many tools are available for an online or offline visualization of point clouds such as the Esri AHN-viewer of which a screen-snippet is shown in Figure 2.1.

There has already been a lot of research and experimentation about visualisation and comparison of point cloud datasets. Some applications are web-based and working, but most of them rely primarily on small and simple datasets or are rather buggy. A very big challenge is to find and implement methods which can handle large (i.e. global) datasets and can do efficient and effective analysis on them directly, namely on-the-fly

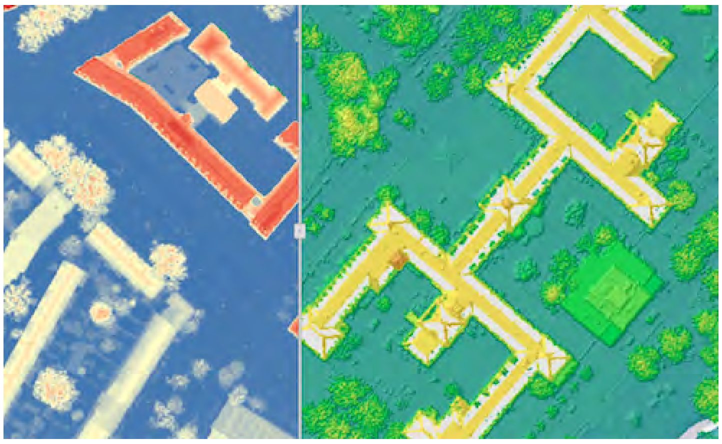


Figure 2.1: Viewport of the Esri AHN viewer, visually comparing AHN2 to AHN3.

processing.

In brief, the following diagram (Figure 2.2) summarizes the scope of the ChronoCity project within the broader vision of the Open Point Cloud Map, hopefully clarifying the links and gaps between them.

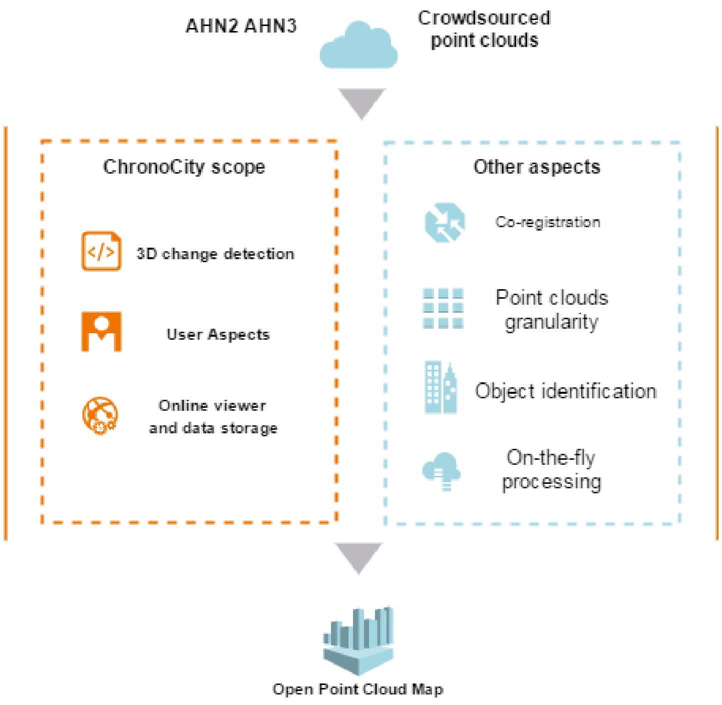


Figure 2.2: OPCM is a long term vision covering many aspects, while ChronoCity solely focuses on a portion of it to step further towards this vision.

The input data for the Chronocity project are AHN2 and AHN3 datasets, but in the broader vision of OPCM, all kind of different crowdsourced data can be input and the changes will be detected between any two point clouds collected in different time frames.

As already mentioned earlier, the ChronoCity scope is bounded by time constraints, so the main focus will be on an efficient change detection algorithm, user aspects (such as user friendliness and intuitiveness of the interface), the creation of an online viewer and efficient data storage to support it. Aspects such as co-registration of clouds with different origins, issues regarding granularity, object identification within the change detection procedure and on-the-fly processing in the viewer interface will not be addressed within the ChronoCity project. Those aspects are part of the broader vision and they will be potentially addressed in the future to achieve the 'globally' Open Point Cloud Map.

2.3. Research questions

The defined vision of the OPCM and the current developments within this, have resulted in the definition of the project scope as proposed in DID-B1/B2. This scope can be best summarized in the formulation of research questions.

How can significant changes in multiple time-changed point cloud datasets be identified and visualized conveniently?"

- How to identify the changes in 3D point cloud datasets, and what is a significant change for the end-user?
- What is the most convenient and user-friendly way to visualize these changes?
- How to properly store different time-dependent point cloud datasets for efficient (online) processing, change detection?
- *How can different (crowdsourced) point cloud datasets be combined to enable comparison?*

Like the main research question exposes, the focus of this project lies at designing and implementing a change detection methodology and (on-line) visualization of these identified changes. Yet there are important underlying principles to facilitate these procedures - being the data model and *comparable* data the most important.

The first sub-question comprises the development and implementation of the actual change detection algorithm. Where development is not necessarily building it from scratch, but also evaluating existing methodologies and possibly combining them. This would result in an executable algorithm, which is mainly executing in the background. Therefore, preferences and results for this algorithm should be communicated to the users. How this user-system communication is best designed in the second sub-question and has a high priority within this project. The system should be convenient to use for all kind of different identified stakeholders / users (DID-B3).

The combination of on-line viewing and on-line processing is a challenge on its own. Data models for viewing or processing differ a lot, but has to be incorporated within each other to yield the best possible, convenient interaction. The third sub-question will research into this field of data models and to what is best suited in our case. Because we will only deal with a small and specified portion of the OPCM, it is for now mainly a theoretical part and not our main focus. We will try to make our methodology as adaptable as possible to different data models, because simultaneously another group is finding a vario-scale structure which could be adapted in the final version of the OPCM.

Finally, the fourth sub-question (in italic) is also an important part of the OPCM; co-registration of different datasets on the same location in order to generate the comparable datasets for change detection analysis. This co-registration is not within the scope of our project and we will not dig into this question, but it is a very important assumption that our studied datasets are correctly aligned. At the same time, the synthesis location group will probably make further steps towards this issue.

3 Requirements

This chapter continuous from the previous one with specific criteria which the final product should fulfill. It starts with elaborating on the general criteria and MoSCoW-setup for all of them. After that more specific criteria will be mentioned for 1) user aspects, 2) change detection and finally 3) the data model and storage of those.

Like explained in section 2.2 there is a major difference between our vision on the finalized OPCM and what will be incorporated in this project. This entire section is applied for implementation at this moment, more generic requirements are therefore left out.

3.1. General

"We will design and implement a 3D change detection algorithm for significant change between two comparable *xyzt*-point clouds, and display them in an interactive, user-friendly viewer"

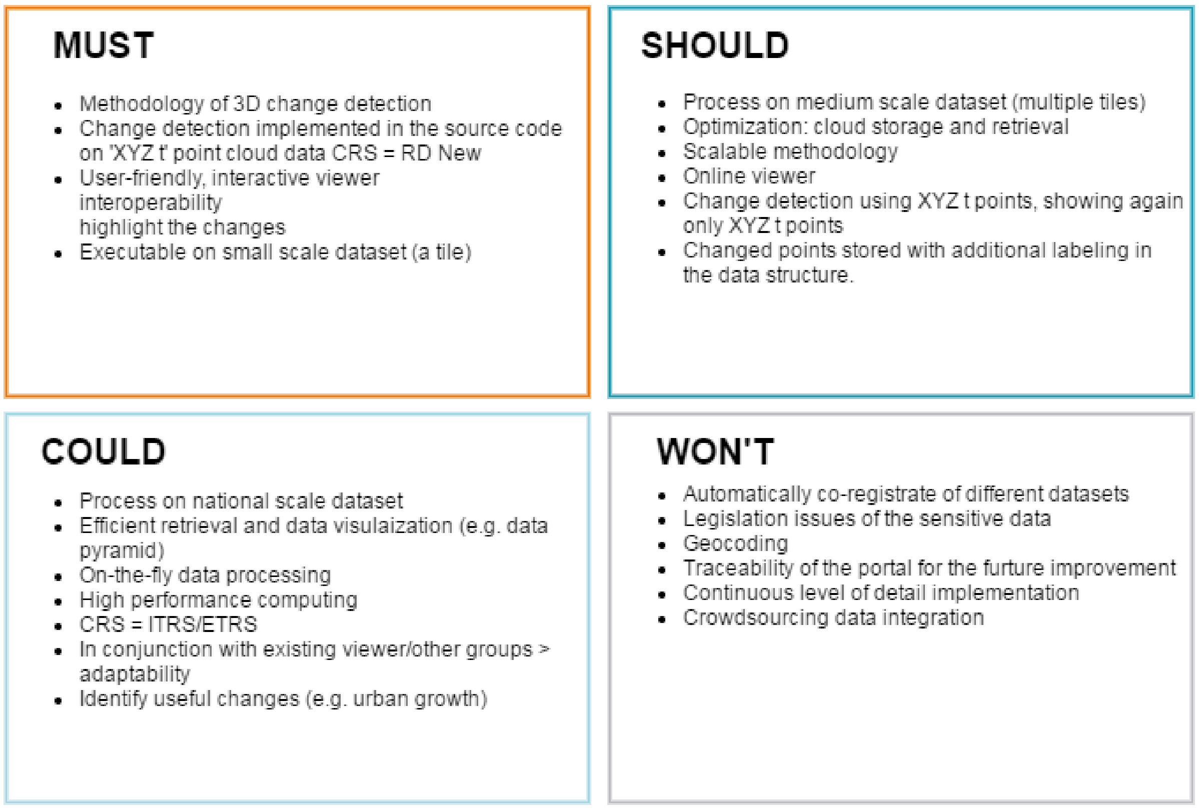
The above statement comprehends the most basic requirements of the project. With the help of the defined statement, one could say that the project development can be split up in two main parts; interactive viewer and algorithm. Now also take the needed data *flow* between those two and it ends up at the extents of the project. In other words, the final implementation of the application has to consider the three major technical aspects of a) the database, b) the data processing and c) the interface of the viewer.

These requirements are summarized in the Moscow-rules (Figure 3.1, updated version from DID-B2). Some of them are straight-forward for interpretation, but most of them will be handled more thoroughly in the upcoming sections taking into account 1) user aspects and interaction, 2) change detection and 3) data model and storage. Besides the requirements, also some boundary conditions in which the ChronoCity application will function are given.

The MoSCoW setup is a priority-list of what is taken into account in this project and especially what is not. Starting with the highest priority *must* to the lowest priority *won't*, with *should* and *could* in between respectively. The four boxes are interpreted. The aspects within *must* are working, implemented and optimized. *Should*-aspects are implemented, but may not be optimally working. *Could*-aspects are handled only in an experimental phase, and if time left implemented. The *won't* section is straightforward, and we will not implement those aspects like also being said in previous section 2.2.

3.2. User aspects

In an actual application - in which also this project would result - always the user aspect is one of the most important aspects. A perfectly working algorithm or procedure is nice to have, but yielding impact with it is the ultimate goal. That impact is only achievable if the application is actually used extensively by a user. Because of the scope of the OPCM (see section 2.1 and rich picture in Figure 1.1), users are expected to be generic; not only professional GIS users within e.g. municipalities, but also 'normal' citizens should be able



MUST

- Methodology of 3D change detection
- Change detection implemented in the source code on 'XYZ t' point cloud data CRS = RD New
- User-friendly, interactive viewer
  interoperability
  highlight the changes
- Executable on small scale dataset (a tile)

SHOULD

- Process on medium scale dataset (multiple tiles)
- Optimization: cloud storage and retrieval
- Scalable methodology
- Online viewer
- Change detection using XYZ t points, showing again only XYZ t points
- Changed points stored with additional labeling in the data structure.

COULD

- Process on national scale dataset
- Efficient retrieval and data visulaization (e.g. data pyramid)
- On-the-fly data processing
- High performance computing
- CRS = ITRS/ETRS
- In conjunction with existing viewer/other groups > adaptability
- Identify useful changes (e.g. urban growth)

WON'T

- Automatically co-registrate of different datasets
- Legislation issues of the sensitive data
- Geocoding
- Traceability of the portal for the furture improvement
- Continuous level of detail implementation
- Crowdsourcing data integration

Figure 3.1: Top-level requirements, setup in Moscow-rules (partly taken from DID-B2).

to interact with the system and yield sensible results.

Also, change detection is only one specific part of the final OPCM. Therefore, we will put focus in optimizing workflows for this, but certainly take into account adaptability of extensions. Meaning the workflow should be developed as a modular one, not being too distractive but also making sure the interface would not be broken if someone does not need the change detection module.

To reach as many users as possible, we make it an online tool functioning within the mostly used browsers. For sure, a web service has several advantages, but the most important one is that the problem of software installations and updates is solved, making it an easy to access, cross-platform app. Hence, the project strives to implement the tool within an online environment and automatize the complete data flow, but it is also known this can be hard to reach. The application will therefore be based on technologies working on the web - especially giving priority to the *viewer* - but it is not a *must* to enable full on-the-fly processing. While online data flow could be of a bottleneck with implementation, it is in this project still allowed to have some manual operation in between the viewer and algorithm.

Key point for evaluating user interaction is having a fully working prototype for at least the exploration of data in which the user should be able at least to pan/zoom/rotate, choose the datasets (time) and change display appearances. These appearances should be clear in one go, but can be different in different situations. The viewer should also be able to specify some (to be determined) algorithm parameters like type, thresholds and region of interest. At the moment of hitting 'run analysis', algorithms in the back isolate the selected points and conduct analysis after which they re-appear in the viewer. At that point the user should again be able to explore the changes, but can also possibly retrieve some more detailed information about those. This workflow has to be implemented in a user-friendly environment in which the challenge lies to give normal users an intuitive flow and professional users as much as control as they like.

Another requirement for the ChronoCity application is the option to download locally the point cloud data, to be able to continue working offline. Finally, although there are various other features for the viewer, that could be introduced as minimum specifications (e.g. computation of the vector of each change), nevertheless, only the previously mentioned specifications are considered as the most crucial ones.

3.3. 3D change detection

"The traditional remote sensing change detection techniques mainly sit within the boundary of 2D image/spectrum analysis" *Qin et al., (2016)*

Like the above statement concludes, change detection algorithms now mainly sit in the 2-dimensional field; 3D data being projected on e.g. 2D rasters on which the process takes place. Since the OPCM would yield a truly 3D environment, it would also benefit most of true 3-dimensional detection algorithms.

The algorithm should be initiated with two raw *xyzt*-point cloud datasets as an input and outputs the changed points with some information on how significant the detected change is. The latter can then be used in the viewer again for visualization purposes. It is not needed for the algorithm to also detect *objects* or classification, for now it is dedicated to geometric changes based on point geometries only. But, as it is likely for some datasets to have, e.g. classification or colours, these properties may be used to optimize the detection and/or visualization.

For now, the goal is to implement and optimize at least one *type* of algorithm. Still the modularity of the OPCM has to be taken into account, meaning it can be extended by a different one or a combination of multiples. Also, the algorithm would somehow work on its own (not integrated with the viewer), so it is possible to have the algorithm as a stand-alone executable. Main requirement for the algorithm itself is the trade-off between optimal results and usability - in which usability can be captured as amount of needed input and processing time. Possibly, the user can indicate what he needs; an amateur user is just interested in some nice and quick indications, while a professional user want to have control over the algorithm and is willing to wait a longer time for processing the results. Still for the professional user, there is the trade-off of an online tool or a dedicated (offline) software package. Therefore, processing time can not be unlimited.

Changes to be detected

The application aims at giving the user an *indication* of changed parts in the point cloud. In this experimental phase, the detected changes do not need an indication of object or suggested changes to for example a topographic map, but are mainly aimed at giving the user an indication of where to look. The to-be-detected changes can be categorized in the examples below. The coloured crosses in the figures below indicate the changed points.

The most simple changes to detect are new and removed data. New point cloud data will for example appear when a new bridge over a canal is built (Figure 3.2), the algorithm in this case highlights the new points only. You could consider a new building-story as new (elevated) data, but in this case they are not because of the older points which lie exactly below. Therefore, these points are within this project defined as 'changed' points.

Although the algorithm aims at geometric change only, it is not always the case a change in classification goes together with a change in geometry. Building a new building within an area which was previously grassland, would definitely result in geometric and classification change, but paving a road in what was previously grassland, probably will not result in a significant geometric change (Figure 3.3) since there is not really an object in place. The algorithm could have a small clause that *if* classification is available, it could indicate these (relatively simple) changes.

Then for purely geometric changes Figure 3.4 and Figure 3.5 give examples of basic changes identifiable for aerial point clouds. These will be used as testing environments and the algorithm should definitely indicate these. The second example is already quite challenging in the sense that maybe the blue area is not exceeding the threshold. Also, if classification it would be nice to also have this result.

The challenge for geometric change is possibly best described by the example in Figure 3.6; irregular objects

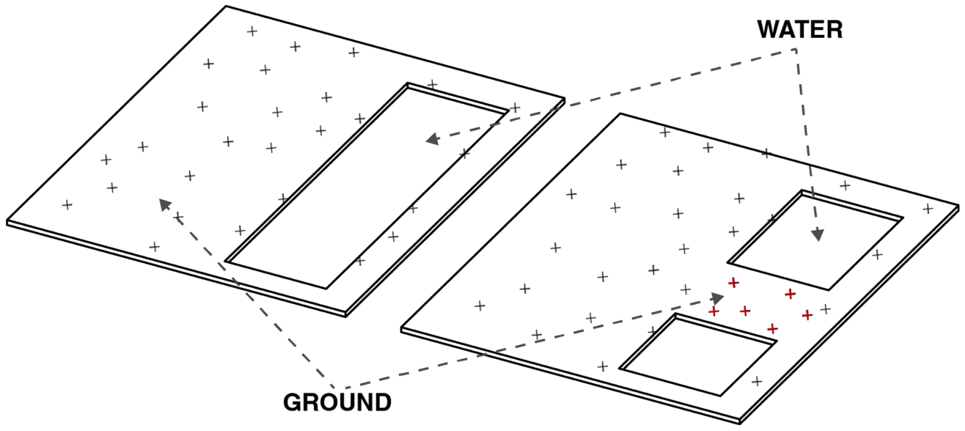


Figure 3.2: New data appears if e.g. a bridge is built over a canal (previously no-return scans).

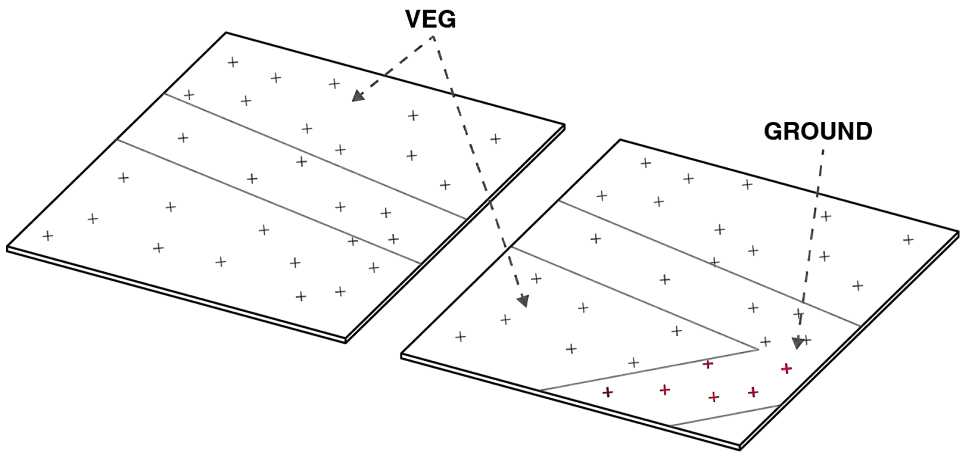


Figure 3.3: Classification changes without significant geometric change.

like vegetation or people. While buildings in aerial point clouds can still be considered as 2.5D, trees really expand and change in all directions. This would be the stress-test for the algorithm, in which the algorithm also indicates that this is a very irregular object. This latter can be very useful in visualization: switching off these irregular changes can declutter the viewport significantly.

3.4. Data model and storage

We have both the interactive viewer and the algorithm. They rely on *data* and there has to be a predefined dataflow between those two. On how the data will flow through the application will be decided later on, when there is a better evaluation of the alternatives in section 4.4. In this section we can better speak about boundary conditions for the data and the relation with other GSP groups.

First of all, an important requirement for the service is to store the detected changes, so that future users can have fast access to that without having to wait for heavy and thus, long computations. This specification is closely intertwined to the next one, which is about the selection of the point clouds needed to be compared, in terms of their timestamps. More specifically, it is clear that the more comparison combinations there are available for the user, the more detected changes will need to be stored in the end and so, a relatively more complicated database model is necessary. To tackle that, the simplistic approach of only allowing time-sequential change detections could be followed, however it is much more practical and useful for the user to be able to compare any point cloud, regardless its timestamp. Therefore, this option is considered for the online viewer, as another minimum specification.

The following requirement for the ChronoCity application is related to its performance. Not only the de-

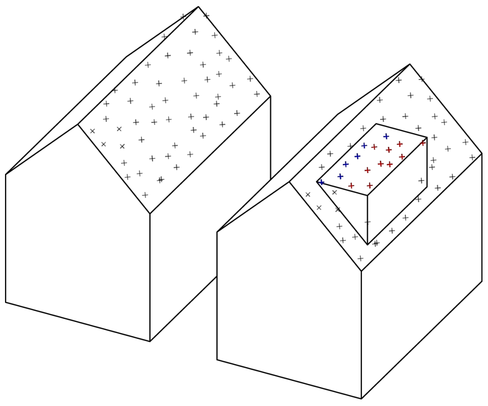


Figure 3.4: Simple roof extension, detectable by the algorithm.

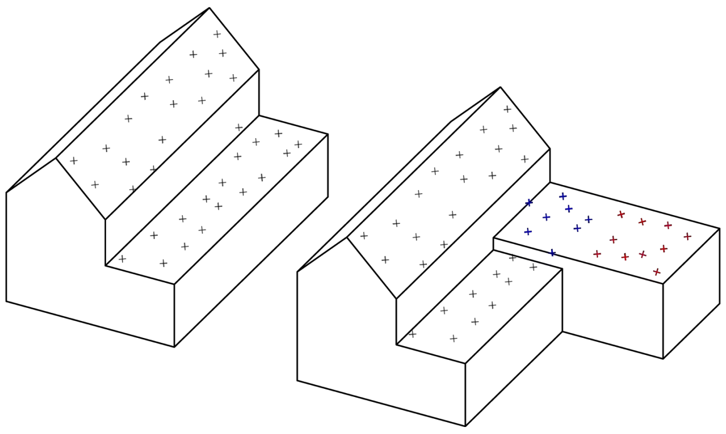


Figure 3.5: Partly demolished building and then extended with a somewhat higher extension.

signed viewer needs to be responsive, smooth and stable, but also, it has to be as customisable, extensible and scalable as possible. Point cloud data are very demanding in terms of storage and due to the fact that nowadays it is very easy to produce one set, their growth has to be sustainable. It is worth mentioning that on 2015, CycloMedia announced its support to 'nD-PointCloud' research proposal, foreseeing the generation of a point cloud of 35 trillion points [6].

Moreover, the open, online, crowd-sourced orientation of the OPCM also implies the application cannot assume perfectly co-registered and harmonized data; data is likely to be in different CRSs, lacks or has different classification values and is acquired using different types of sensors. Therefore, the ChronoCity application goes for a generic approach, on only *xyzt*-point clouds but there is already the assumption of perfect co-registration and harmonization. Also the other GSP-groups handle our assumptions partly; the scale and granularity group will strive towards an optimal data model for (online) visualization and the location-group strives for an optimal co-registration of datasets.

For now, we assume these are in current research and will be developed as implementable solutions. Therefore we take two perfectly co-registered aerial point cloud datasets as an experiment; AHN2 and AHN3 and yield results only taking into account their geometric properties. Still, test can be carried out to evaluate the procedure on their adaptability to the final vision of the OPCM.

Figure 3.6: (irregular) 3D change of vegetation during a (sometimes very) short period of time.

4 Conceptual analysis

The conceptual analysis can be seen as the first thorough exploration of alternative methodologies to reach the requirements of the final product. Based on this analysis, the best alternative is selected, optimized and implemented in the next project-phase. The section first guides you through the basic workflow, after which some important aspects are discussed separately in the paragraphs starting with the user aspects, a change-detection methodology and finally the online viewer and data model.

4.1. Overview

Because of the rather complex nature of the entire system of viewer, algorithm and dataflow between the two (section 3.4), this package can best be comprehended as a combination of (linear) modules as described in Figure 4.1. The figure describes the linear workflow where the user has some control in the orange parts and background processing is executing in the blue parts.

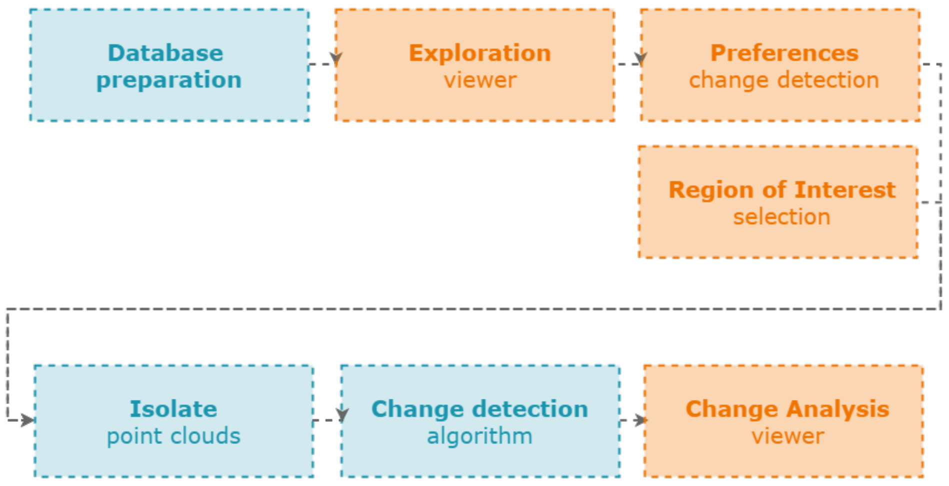

Database preparation
Exploration viewer
Preferences change detection
Region of Interest selection
Isolate point clouds
Change detection algorithm
Change Analysis viewer

Figure 4.1: Modularity for project implementation management. User has control over the orange parts, where blue parts mainly take place in the background.

Ideally, this process would all take place on-the-fly, but for the implementation of this project it is fine to have these as separate modules and thus initiate each process separately / manually. In the rest of this section all main modules are discussed and evaluated in which the orange modules are taken together as user aspects, the change detection on its own and isolation and preparation again together in the data model section.

Within *Database preparation* the two point clouds are prepared and possibly stored into a database which can be handled by the viewer-framework. Then, the user can *explore* those datasets after which he decides of doing a change detection. In that case he needs to define his *Preferences* including a specified *Region of*

Interest. These two can be seen as the parameters on which the algorithms in the background first *Isolate* the points within the region of interest and then execute the actual *Change Detection* on them. The outcome of this detection is then forwarded to the viewer again, on which the user has the ability to explore the changes and retrieve some information on them.

4.2. User aspects

The application has to be usable in order to have actual impact on the society and the environment. Usability can be separated into three main aspects; needs, user flow and acceptance. The *needs* are covered in previous deliverables and shortly repeated in our vision on the OPCM. For now it is generally based on the assumption that quick visualization of changes can yield significant advantages in existing (e.g. municipal) analyses, but it is also just fun to watch. Acceptance is in this case mostly influenced by the waiting time for the user, but also in sense of the used platforms and continuous development. In current experiments the product will be designed for desktop applications and the source code will be publicly available enabling external users to develop upon the same framework.

For now, it is important that the interface and the flow are designed as intuitive and extendable as possible. While the actual design of the interface will be implemented in the next phase, when it is clear for which viewer it has to be adapted. Still, each user-module described in Figure 4.1 contains some important user aspects which are described below and summarized in the user flow diagram in Figure 4.2.

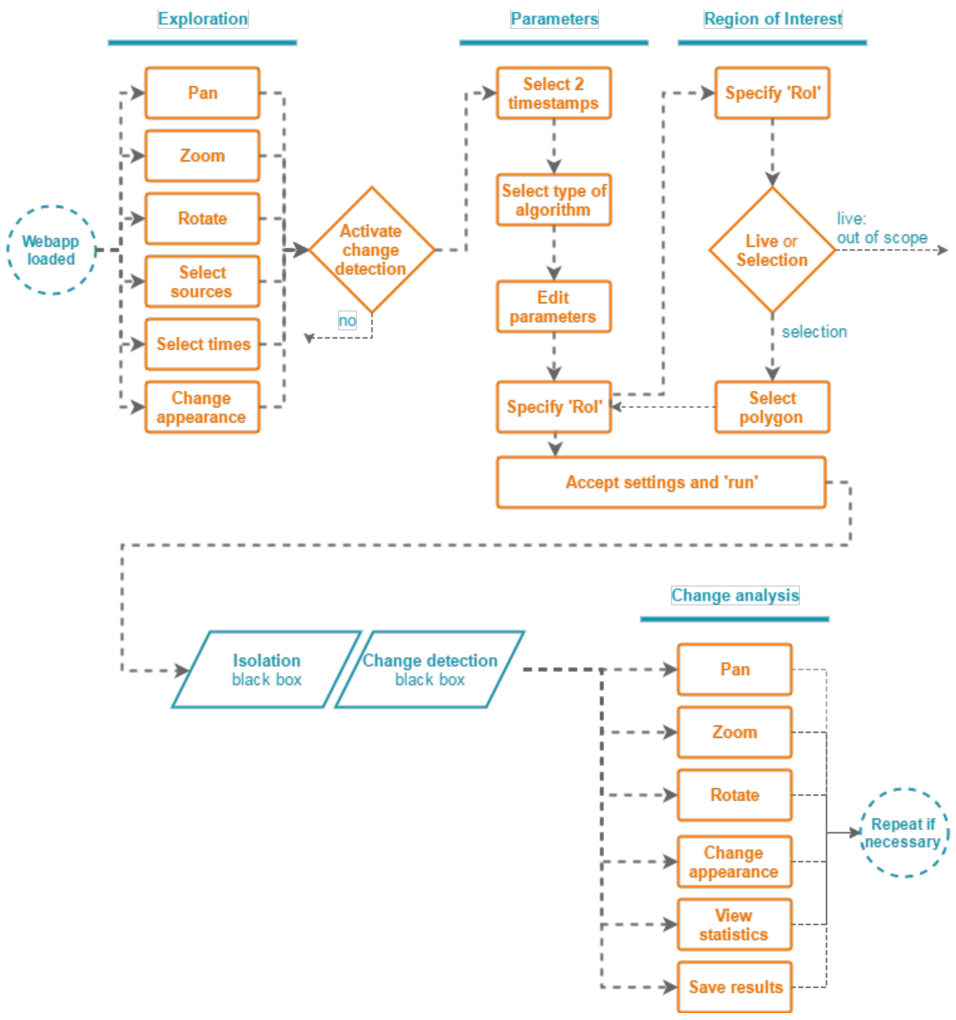


Figure 4.2: User interaction flow diagram; blue paralleloids are so-called black boxes. These processes take place in the background.

Exploration
The exploration phase is straightforward in its definition and purpose. This is also the landing-page in which the user has basic controls over the viewport being panning, zooming and rotating using standard mouse- and key-gestures. It is also possible to select different suppliers of datasets - think of official, unofficial or crowd-sources - and change the appearance of the point cloud. The latter can be a default setting, but in different conditions (e.g. urban vs. forests) a user is likely to need another representation. Also, display properties can have significant impact on the performance and therefore usability of the system. An extension on most used viewers is the ability to ‘scroll’ through time. Since this display of point clouds through time is not within the scope of this project, this will not be implemented as a slider, but rather as a selection box on which the user can specify whether to see the AHN2 or AHN3.

Change detection - parameters
Once the user wants to detect changes, he should go to a differentiated tab for change detection. This tab is dedicated to the entire detection and analysis- procedure since it is possible in the future that the interface will be extended with for example routing- or streetview-plugins which can distract the user from the proposed workflow. By enabling this tab on which the user needs to specify four main aspects before it is able to continue the actual analysis, first he needs to specify the to-be-compared timestamps. This can either be done by numerical input or using the same slider as in exploration. If multiple detection-algorithms are available, the user can specify which one to use and define some parameters for those. These would be set to a default value.

Region of Interest
Since the algorithm would not process the entire world at once, the user should also specify his region of interest. In theory, this can be based on live processing the position and angles of the viewport. Implication of this would be that the change detection should be extremely fast in order to still navigate smoothly. Therefore, in the prototype the user can only specify a fixed RoI using a polygonal selection. This would first be a rectangular *xy* bounding box, but may be extended into a free-form polygon. The interface will not let you select a portion in the *z*-direction.

Analysis
After the user specified all mandatory fields he could continue by pressing a simple button. This would initiate processes on the background on which the viewer receives back the outcomes. Then again, the user can explore using basically the same tools as in the exploration phase being it in a different colour scheme and only consisting of the changed points. If the specific algorithm is able to indicate specific kinds of changes, these can be selected. But the changes can also be exported as separate point clouds or textual reports.

4.3. 3D change detection
The focus of ChronoCity is primarily aimed at 3D geometric change detection of *xyzt* LiDAR point clouds. The aim of this section is to investigate the different techniques, algorithms and tools implemented for 3D change detection and to determine which is the most appropriate for our final goal. Many of the approaches that have been investigated lead to successful results, but each of those is suited for a specific application. The challenge is to integrate those approaches within the scope of the ChronoCity project in order to obtain a successful result and visualize the changes.

In this section, we will first start with discussing the theory of the most feasible 3D change detection approaches we have discovered from the literature review and from the existing algorithms. Later on, implementation strategies and the associated risks will be assessed to prioritize and direct our efforts in the next phase of the project.

4.3.1. Theory
Change detection techniques have been developed for decades and they have many useful applications such as post-disaster monitoring, urban environment changes, keep databases up to date and so on. The process of change detection is to find out the differences in the registered 3D data [20]. With so many applicable approaches and open-sourced libraries existed to identify the changes of the remote sensed data today. The following section will summarize and discuss three major methodologies that we consider as appropriate methods fulfilling the project demand. They are point density comparison, Euclidean nearest distance and

Octree structure comparison respectively (Figure 4.3).

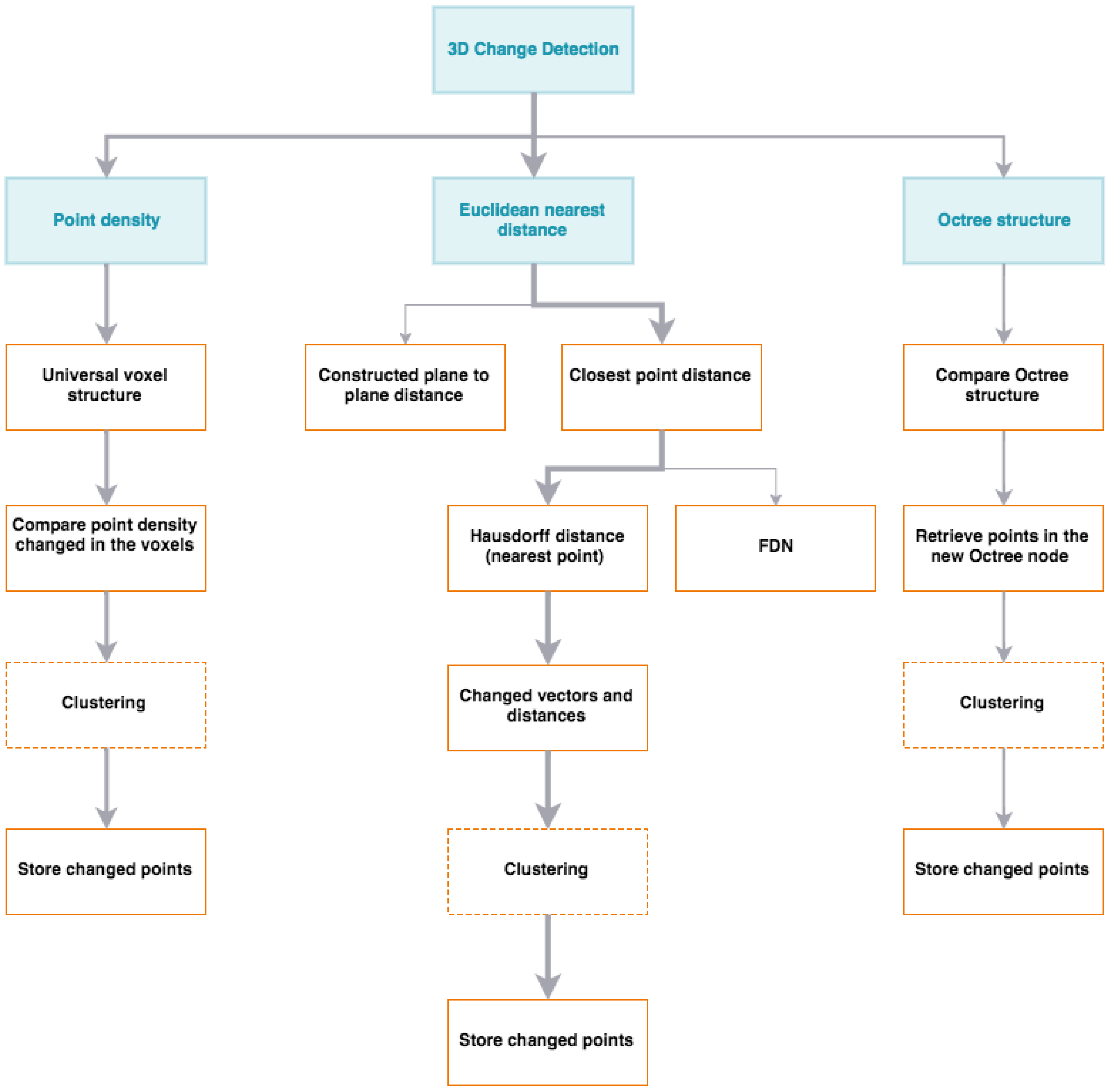

3D Change Detection
Point density
Euclidean nearest distance
Octree structure
Universal voxel structure
Constructed plane to plane distance
Closest point distance
Compare Octree structure
Compare point density changed in the voxels
Hausdorff distance (nearest point)
FDN
Retrieve points in the new Octree node
Clustering
Changed vectors and distances
Clustering
Store changed points
Clustering
Store changed points
Store changed points

Figure 4.3: The proposed 3D change detection approaches and their theoretical workflow. The thickness of the arrow indicates the priority of implementation.

Point density

Point density comparison is theoretically simple and relatively easy to implement (left of Figure 4.3). Imaging an universal voxel grid covering the entire point clouds, point density comparison is simply comparing the point densities of the two datasets of the same voxel unit. If the compared point density over the reference point density is larger than a certain threshold, we can say that it is a positive change, new points (new object detected in the real world) within this voxel. If it is other way around, then we say that it as a negative change, where the original objects were destroyed or disappeared. If this ratio keeps consistent within certain range, then the object remains unchanged (Figure 4.4).

Voxel size and threshold of the ratio are two major parameters affecting the accuracy of the results and computation time, a certain level of fine tune options should be given to the users for generating their desired results and enabling visualization if this algorithm is implemented. After the point density computation, the changed points of the compared point clouds will be retrieved from each voxel and cluster them via the algo-

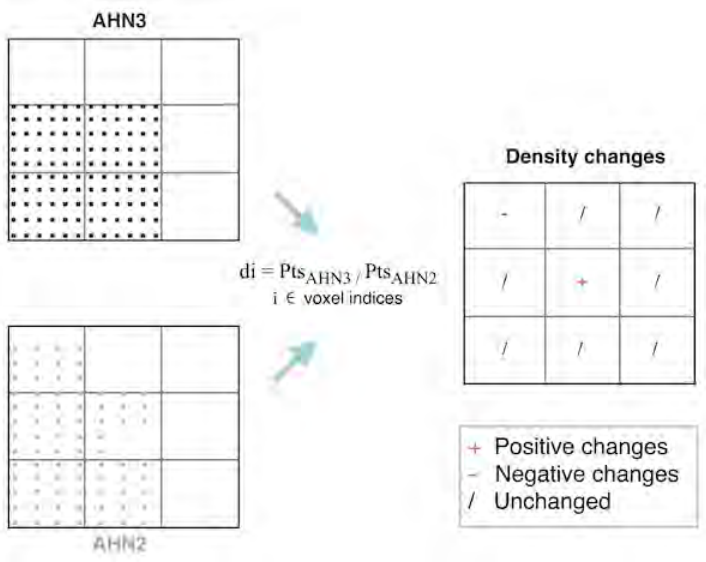


Figure 4.4: The density function compares the number of points in AHN2 and AHN3 in the same voxel unit. Even the point resolution of two datasets may be different, the change in density function value can still indicate the positive or negative changes.

rithm like region growth (based on similar features) or DBSCAN (based on density) if needed. The retrieved point will still retain the same data structure of $[X,Y,Z,t]$, but it will be stored in a new LAS or LAZ file. How the changed points are being stored will be addressed in more detail in section 3.3.2 'Implementation'. Figure 3.11 sums up the implementation procedure.

On the other hand, point density comparison may be a quick approach but can only give approximate 3D change detection results. Some issues are associated with this approach. For instance, false positive results could happen if, in reality, no objects have been changed in the specific region but only the point density scanned in the compared and reference point clouds are significantly different, due to different external conditions of two surveying operations. The opposite situation leads to false negative result, so the new object and old object are geometrically different, e.g. house changes to tree, but the scanned point density within the voxel unit happens to be similar. Another drawback of this approach is that the changes could become discontinuous due to the nature of voxelizing the real world.

Euclidean nearest distance

The issues associated with the point density approach are mainly false positive, negative detection and discontinuous change detection. Considering the vision of OPCM, which includes the changed objects identification and useful change application in the future, this approach will pose a threat for these further implementations since any feature of the point clouds is usually important for the object identification task. To overcome this drawback, Euclidean nearest distance should be considered as a more robust and rigorous alternative (middle of Figure 4.3). Qin et al. [20] see the Euclidean distance change detection approach as taking the three degrees of freedom for 3D geometry into account by computing the Euclidean distance of two surfaces, in their normal direction (Figure 4.5). Nevertheless, calculating the Euclidean distance from the reconstructed surfaces of the large scale point clouds is already a massive computation work and outside of the project scope. The alternative is simply calculating the Euclidean nearest distance of the compared point to the reference point neighbour. This is the change detection approach adopted by the 3D point cloud processing software, CloudCompare, and a common technique to perform change detection on point clouds [10].

To understand and assess this approach more thoroughly, some experiments have been conducted on CloudCompare. CloudCompare is a 3D point cloud processing software. It is written in C++ and was designed to perform comparison between 3D point clouds.

The approach we followed is based on loading two datasets, AHN2 and AHN3, of a neighbourhood in the

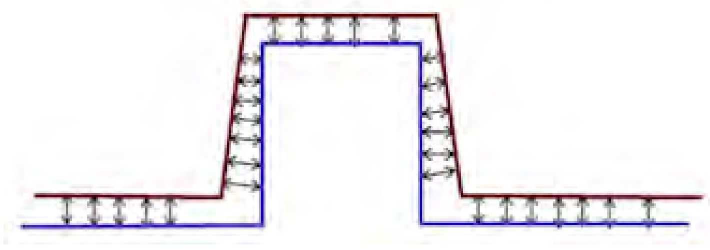


Figure 4.5: Geometric comparison: Euclidean distance comparison, adapted from [20].

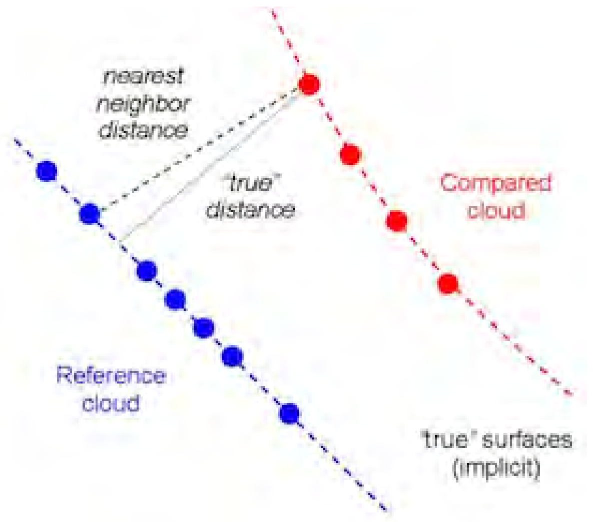


Figure 4.6: Nearest neighbour distance algorithm. Image source: [5]

city of Leiden and compute the nearest neighbour distances between the two datasets, the changes will be highlighted automatically after the processing, as shown in Figure 4.7.

This algorithm is very straightforward, it first searches for each point of the compared cloud the nearest point in the reference cloud and computes their Euclidean distance. This method is particularly suited if the reference cloud is dense enough, otherwise the distances may not be accurate enough [5].

To obtain an insight of the performance of such algorithm and assess whether it is feasible in our project we compared the processing times of two different (in terms of number of points) point clouds covering the same area.

In the first trial we tried to load a neighbourhood in the centre of Leiden both as AHN2 and AHN3 datasets. In the second trial, we tried to load an entire tile of both the AHN2 and AHN3 datasets. The testing was conducted on a Mac OS laptop with 2.5 GHz Intel Core i5 processor and 4 GB of memory. Table 4.1 reports our findings.

Even though the second trial of the experiment did not succeed because of the much larger datasets, the testing still reveals valuable insights and boundary conditions for our future implementation. Most importantly, this approach and algorithm are proved to be pretty doable when processing on a small area of interest, like the centre of Leiden. When large scale datasets are used, splitting the task into smaller chunks is needed, and this is achievable if the point clouds are properly indexed via the technique like Octree or computed via the high performance computing platform from Fugro.

However, the result from the first trial explains the real-time change detection processing is not yet possible with this algorithm. A matured software like CloudCompare still takes about 25 seconds to complete the

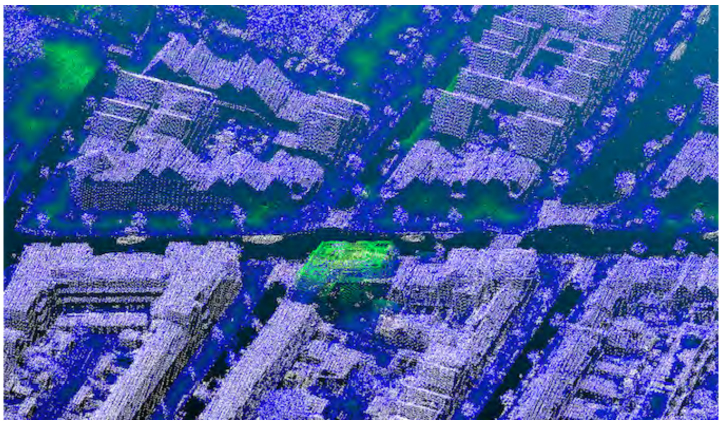


Figure 4.7: Changes of the compared points are highlighted in green color in CloudCompare.

Centre of Leiden	AHN2	AHN3
Points	4.128.749	11.581.328
Loading time	5 sec	9 sec
Change detection processing time	11 sec	
Tile	AHN2	AHN3
Points	270.501.199	689.177.299
Loading time	5 minutes	crashed
Change detection processing time	not available	

Table 4.1: Two experiments with different sizes were tested with CloudCompare.

entire change detection task. Therefore, to accomplish a satisfying user experience, appropriate system architecture design is needed. This will be discussed in more details in section 3.4 - ‘Online viewer and data model’.

Again, considering the change identification task in the future of OPCM, we hope to store as much information of the changes as possible. Other than purely identifying the changed points based on the change distance and highlighted with different colours in the CloudCompare as shown in Figure 4.7. Additional information like change vector (Figure 4.8) should also be kept in the data structure of the changed points. The data structure will ideally be [*X, Y, Z, t, dX, dY, dZ, d*], where the *dX, dY, dZ* represent the vector of change and d is the change distance. The change distance is an important threshold parameter for identifying the real changes as illustrated in Figure 4.9.

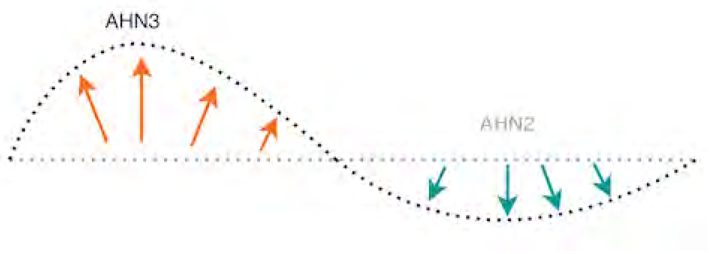


Figure 4.8: Change vector can be used to indicate the change direction.

Octree structure

Another approach that can be used for change detection is the comparison of the octree structures of the two point clouds (right of Figure 4.3), which is supported by Point Cloud Library (PCL). PCL is a large scale open project for 2D and 3D point clouds processing. It contains numerous algorithms suited for point clouds

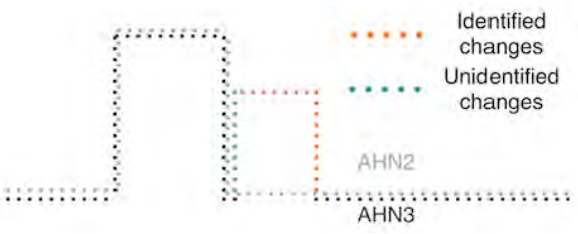


Figure 4.9: Offsets and errors are unavoidable in the measurements. However, inappropriate distance threshold could ignore the real changes, as shown in the blue dash line above.

use, such as filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation [19]. PCL is based on the 3-clause BSD license and it is free for commercial and research use and open source.

This approach recursively compares the tree structures of the octrees, so that the spatial changes can be identified by differences in the voxel configuration. For this reason, one of the main strengths of this method is that it is very fast and allows to detect changes in near real time [19].

In our research, we are particularly interested in the 'module octree' used by PCL, which provides the user with an efficient method for creating a hierarchical tree data structure from point cloud data. In this way by means of an octree structure, the space can be partitioned recursively. The root node describes a cubic bounding box which contains all the points of the cloud. Each octree node can have either eight children or no children. At every tree level, the space becomes subdivided by a factor two which results in an increased voxel resolution.

The implementation of this method (Figure 4.10) provides efficient nearest neighbours routines like 'Neighbors within Voxel Search', 'K Nearest Neighbor Search' and 'Neighbors within Radius Search' [19]. The algorithm is able to automatically adjust its dimension to the point dataset.

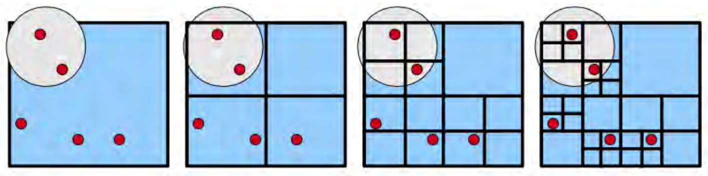


Figure 4.10: Octree neighbor search. Image source: [19]

Xu et al. [31] propose a similar approach which further supports the octree structure method with a theory and the actual study cases which focuses on detecting changes to buildings and trees in the urban environment.

In order to determine the changed areas, the octree is generated on the first datasets (i.e. at the moment *t1*) which is regarded as the reference cloud. While the second dataset (i.e. at the moment *t2*) is the compared cloud. Then, the coordinates of each point in *t2* are imported to *t1*. Because *t1* dataset was used to establish the octree, the leaf node to which the 3D coordinates of a point belong can be determined from the octree index. The points in this leaf node and its siblings form the set *N*. The 3D coordinates of a generic point can be taken as the origin and a search is conducted over a radius *r*. The points in *N* that fall within *r* form the set *N1*. In the octree indexing process two cases have to be considered:

1. The corresponding leaf node to which the 3D coordinates of a point belong does not exist in the octree index of *t1*, i.e. *N* is empty. In this case, the point is defined as *changed*.

2. The corresponding leaf node to which the 3D coordinates of a point belong exists in the octree index of *t1*, i.e. *N* is not empty. In this case, the point is defined as *unchanged*.

4.3.2. Implementation

Each of the proposed approaches has its pros and cons, also resulting in different outputs of changes. This subsection further discuss the implementation issues and discuss which is the prioritized approach for us to implement in the next phase of the project. The overall implementation workflow is summarized in the following Figure 4.11.

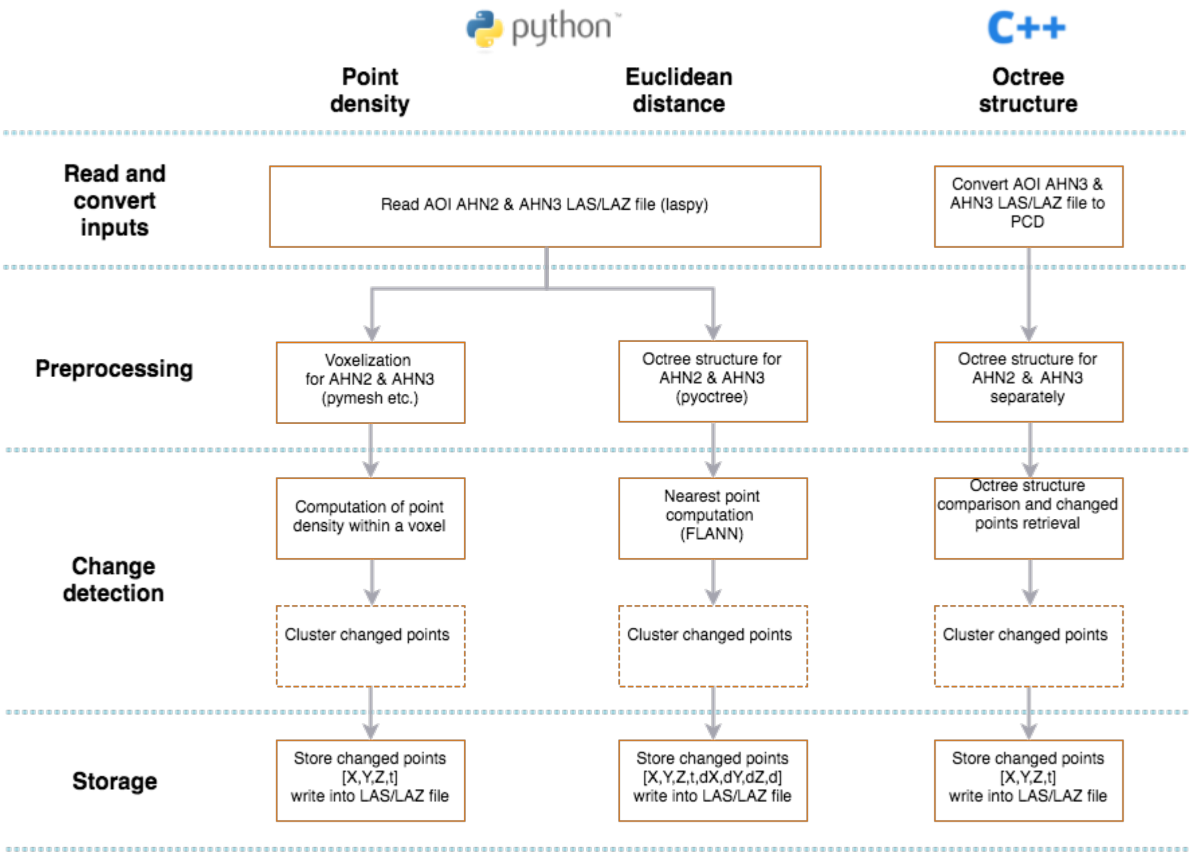

python
C++
Point density
Euclidean distance
Octree structure
Read and convert inputs
Read AOI AHN2 & AHN3 LAS/LAZ file (laspy)
Convert AOI AHN3 & AHN3 LAS/LAZ file to PCD
Preprocessing
Voxelization for AHN2 & AHN3 (pymesh etc.)
Octree structure for AHN2 & AHN3 (pyoctree)
Octree structure for AHN2 & AHN3 separately
Change detection
Computation of point density within a voxel
Nearest point computation (FLANN)
Octree structure comparison and changed points retrieval
Cluster changed points
Cluster changed points
Cluster changed points
Storage
Store changed points [X,Y,Z,t] write into LAS/LAZ file
Store changed points [X,Y,Z,t,dX,dY,dZ,d] write into LAS/LAZ file
Store changed points [X,Y,Z,t] write into LAS/LAZ file

Figure 4.11: The overall workflow of implementing three change detection approaches.

Point density and Euclidean nearest distance are programmable in Python environment, which is a big advantage because all team members already have certain level of foundation in this language. The first step for both approaches is converting the AHN2 and AHN3 in the LAS/LAZ file into a Python readable format, and this can be done with the external Python library: *laspy*. *Laspy* is a pythonic library for reading, modifying and writing LAS files, and is compatible with Python 2.6+ and 3.5+ [3].

In the preprocessing stage, voxel structure and octree structure will be created for the inputs of the point density approach and Euclidean nearest distance approach respectively. Again, external libraries like *pymesh* and *pycotree* can support this task.

In terms of change detection computation, nearest neighbor search in the second approach will be the most computationally expensive process but can also reveal a more rigorous result [10]. *FLANN*, a library for performing fast approximate nearest neighbor searches in high dimensional spaces shall be incorporated in the script [15]. On the other hand, point density calculation is computationally less expensive than the nearest point search and distance calculation. This fact makes it better than the Euclidean nearest distance with respect to the performance issue; however, the discrete and discontinuous change detection is still a big shortcoming. Clustering of the changes via the region growth, DBSCAN and many more algorithms are also doable with Python and could be an optional step if necessary.

The final part is data storage. It is important to note that the freedom of most programming languages give no constraints on which attributes or properties associated with the points to be stored in the database. However, concerning the acceptable input format of the corresponding viewer, in our case is Potree and will be explained in the following section, LAS and LAZ (compressed format) are the most important data structures for point cloud data. LAS and LAZ have defined specifically the items with their format and size to be stored [1]. [*X,Y,Z,t*] are the default items maintained in the LAS/LAZ format while change vector like [*dX, dY, dZ, d*] are not. An option would be writing these additional values into the non-corresponding items of the LAS/LAZ file while converting them back in the front-end environment.

The octree structure comparison is a mathematically and computationally simple approach and widely used in robotic vision to detect almost real-time environmental changes [20]. However, it is also the most challenging one technically since the entire library is written in C++ language, which is very new to most of the team members. Currently, *Python-pcl v1.0* is a small python binding to the point cloud library [26]. Unfortunately, the octree change detection is not covered in the current version which means we will either need to familiarize ourselves with C++ to be able to translate the algorithm into python environment or directly implement the algorithm with C++ language. This indicates a certain risk of this approach concerning limited time frame of the project. However, among three approaches, the logically simple and high-performance characteristics still very worth us investing some more exploration.

To sum up all, after the current assessment of three approaches, the Euclidean nearest distance will be the priority for the practical implementation in the next project-phase. Other two approaches will still be tested but with less efforts and time investment, and if no successful results can be derived from them in the system design stage, which is a week long, in the next phase. The project will only focuses on implementing the already existing Euclidean nearest distance algorithm in our ChronoCity application.

4.4. Online viewer and data model

The purpose of ChronoCity is to provide the users with a tool for quick change identification between 2 point clouds, captured at different moments in time. The design of such application is technically depended on various aspects that in general are related to: a) the database and data processing and b) the interface and its capabilities. Each specific decision made regarding these aspects will give the final shape to the online viewer (ChronoCity application), with which the user will interact.

Some few examples of such decisions needed to be taken are:

- How and where will the 3D change detection algorithm be implemented?
- Which data model is best to use?
- What kind of viewer can visualize the detected changes and how?
- In which way should the viewer interact with the data and the user?

Nevertheless, the user will only interact with the application at a very front-end level, namely its interface, being unaware of most of these decisions regarding both the application's front-end and back-end engines. The online viewer will simply be his *reality*, as it is illustrated in Figure 4.12 from the blue features.

For a software in general, it is expected that the more functionalities being offered, the more complicated its design and thus, more time needed. Because of that and in order to be most effective while designing the online viewer, already well-established software packages and libraries had to be used to support the development process. Depending on the choice, some constraints will be introduced and so, these libraries and software foundations need to be carefully selected. The considerations taken into account when choosing the most suitable of them (along with the justification of each choice made), will be discussed at a later point. Before that, however, an ideal approach will be discussed, where a prior hypothesis is made, that no such technical constraints or important time limitations exist.

4.4.1. Ideal scenario

It is a fact that the development of a software is always based on some former technologies and standards. For example, the creation of a website cannot avoid the use of HTML. In this sense, even if Chronocity appli-

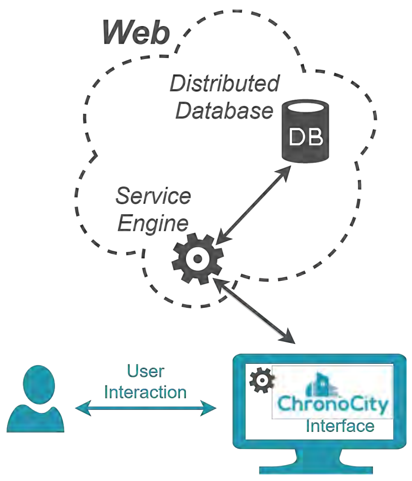


Figure 4.12: User interaction with ChronoCity application through its interface.

cation was designed from the very beginning, some crucial technologies would have to be considered.

As it has already been mentioned, this application should be available online. That alone, introduces the requirement of a web server, hosting and running an online service that is communicating both with the users and the data. The most important data to consider, would be the point clouds and that, due to their size. Hence, a powerful database would be needed. This database should not only be able to efficiently store and handle massive point clouds, but at the same time it should be able to deal with crucial matters such as: point properties, indexing, tilling, multi-user operation, scalability, different data types (vector/raster) etc. Yet, a database management system (DBMS) being specifically designed for point clouds and covering all needs above, does not exist. To avoid building one from scratch, some already existing DBMS options that have been put to the test and benchmarked, such as Oracle Exadata, might be remarkable solutions [16].

Assuming that the ideal database management system is available to implement, then the next essential hypothesis is that an efficient change detection algorithm has been developed. An algorithm being fast enough to contribute to a nice user experience and deliver changes to the user as soon as possible. The execution side (server/client) of not only this 3D change detection algorithm, but also other routines concerning various other functionalities of the final product (such as the selection/clipping of the points to compare), is depended on the final architecture of the software.

Due to the fact that servers are designed to handle high-performance computing and big data much better than the clients, by using Error-Correcting Code (ECC) memory, durable Xeon processors, parallel tasking operations, cloud storage etc, most of the computational load in an ideal scenario would be assigned to the server-side.

More specifically, the user would have two core options regarding the change detection functionality. The first option would be to generate a quick descriptive preview of the changes, based on a) the 2 input point clouds, b) the 2 different timestamps and 3) a specified sub-selection of the points within the area of interest as a sample (e.g 10.000 of the currently loaded points in the viewport). This ‘on-the-fly’ feature introduces two requirements. First, the change detection algorithm must be executed on client-side and secondly, the results need to be able to be loaded and visualised back into the client (parallel to the data being served from the web).

The second core option of the user would be to request from the server a complete and official 3D change detection on the area of interest, which could be done via a selection tool (bounding/clipping). In this case, however, because of the potentially large amount of raw points, the results could not be immediately delivered to the user. Hence, a calculation ‘ticket’ would be opened from the user and the results would be temporally stored and hosted on the server.

The temporal nature of these data are related to several factors regarding the future needs of the same results. More specifically, for a given consumption of server storage capacity, having a corresponding cost, the amount of calculated data allowed to be stored temporarily will be depended on their ‘value’, which is their future re-use. This ‘value for money’ data storage concept has already been efficiently handled by online services such as Youtube. Thus, after the changes have been calculated and temporally stored, they will be instantly available to access, at future requests, substituting also the ‘on-the-fly’ functionality of the viewer. This concept is demonstrated in the Figure 4.13 below.

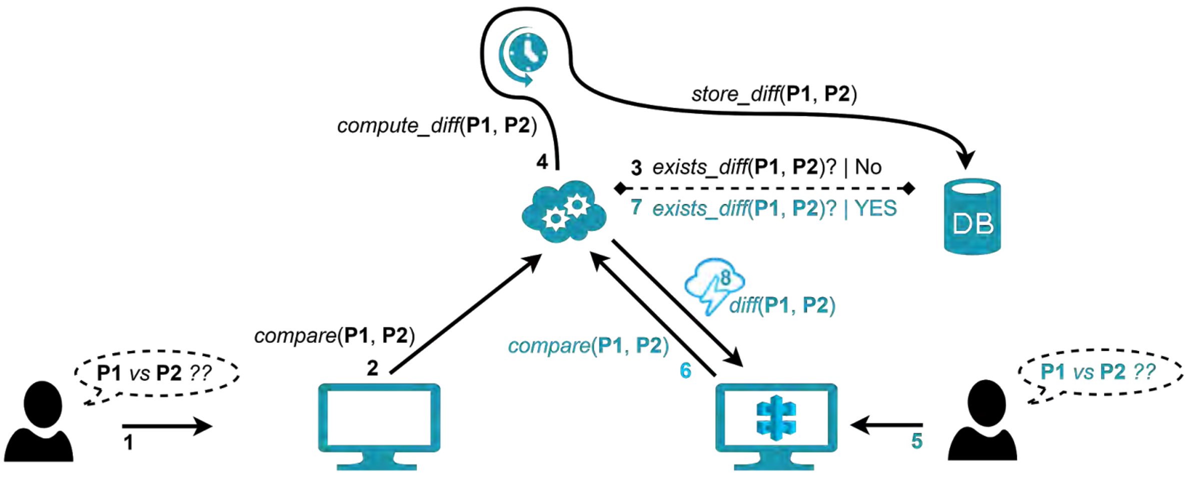


Figure 4.13: Example of a request procedure for a specific point cloud change.

In both the ideal scenario and the final implementation, another important issue to consider is how the calculated changes will be stored. Point cloud data usually contain millions of points, which translates to massive storage demands. Hence, in an open - crowdsourced data delivery, calculating and storing new information (i.e. the detected changes) should at least be done in a non-redundant way. For that reason, the selected database management system should be able to store and retrieve efficiently the computed changes, probably as point attributes or small intermediate point clouds.

4.4.2. State of the art solutions

After having specified the minimum specifications in chapter 3 for the online viewer, the next step is to identify the top worth-considering software libraries, from which the viewer could benefit. A holistic comparison and review of them, in terms of these requirements, would yield the team’s choice.A dynamic progress and research has already been made from the graphics rendering and information technology community, in terms of massive point cloud visualization (both in web-based and desktop-based platforms). Some of the most important examples include the following:

- **Plasio** project, which is a point cloud rendering platform based on WebGL, integrating the Entwine and Greyhound components from Howard Butler, Connor Manning [29],
- **PointCloudViz** LiDAR viewer for both Desktop and Web Server/Client architectures [13],
- **udWeb** viewer, which is a form of a voxel rasterizer based on a point cloud search engine indexing system [7],
- **Scanopy** (Scan Data Organisation, Processing and Display) project by the computer graphics department at TU Wien [21], which is one of the first point cloud renderers with the ability to visualize billions of points,
- **Arena4D** point cloud editing and viewing package suite by Veesus, that offers both desktop and server solutions [28],

- **CESIUM** open-source JavaScript library to render world-class 3D globes and maps on a browser. It is a framework offering visualisation capabilities for many types of data sources such as Imagery, Vector data, 3D models, Point Clouds, Temporal data, etc [4].

Nevertheless, based on the viewer's current requirements, the 'best' software candidates are not among the previous listed options, but instead, are presented below. These are:

- **Potree** WebGL point cloud viewer, mainly developed by Markus Schütz
- **iTowns** web framework by IGN and Oslandia
- Visualization Toolkit (**VTK**), being currently maintained by Kitware [9]

In addition to these 3 softwares, an additional non-rendering library designed by Oslandia will be presented (LOPoCS), which undertakes the task to bridge point cloud viewers and Postgis. In each case, the core features being offered by each option will be discussed next.

Potree

Potree is a free, open source web viewer that can visualize large point clouds by utilizing both the rendering power of the Web Graphics Library (WebGL) and the functionality of the three.js library. It is based on the Scanopy project and is still an active project (the latest pre-release version is 1.5RC), being mainly maintained by its initial creator. It is a standalone solution, as it requires no extra plugins to run and so, it can work on all operating environments and browsers that support WebGL, making it a cross-platform application. It runs entirely on the client's side, with the server having only the task to host and serve the files to the client. These files are by default stored in either LAS, LAZ (compressed format) or BIN (binary) format.

Apart from its modern and user-friendly interface, the most important aspect of its core engine, is how the point data are handled in terms of loading and visualizing. Regardless the original number of points stored in the database, the Potree viewer identifies quickly the visible regions and based on a given number of points (bucket of points) it divides the point cloud into *voxelized* regions of different Levels of Detail (LOD) as shown in Figure 4.14. These different LOD's correspond to different number of points and densities, nevertheless, their aggregation always respects the specified bucket of points.

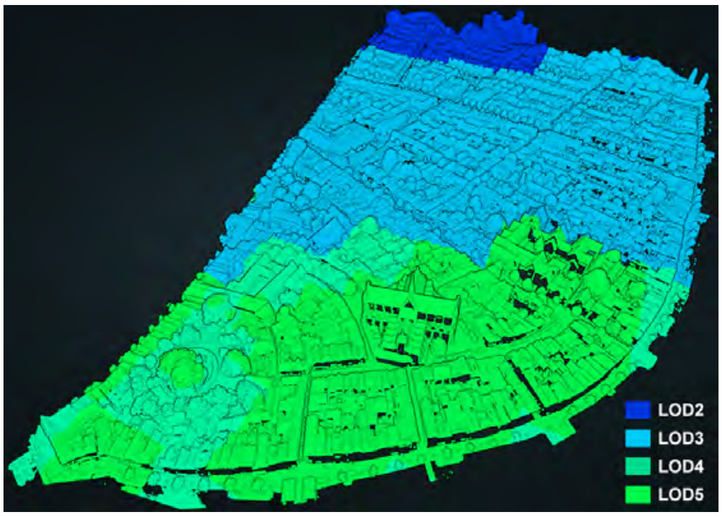


Figure 4.14: Different LOD's showed on a loaded point cloud (region in Leiden).

These *voxelized* regions are nothing more than a slightly modifiable (regarding the point cloud sub-sampling method and the corresponding chunk sizes) nested octree structure (MNO), that was introduced from the Scanopy renderer. Based on this structure, a point cloud is *unevenly* divided among different nodes, following a subsampling procedure. At every depth level of the octree, the size of the node decreases, whereas the density of the points increases, a concept that is illustrated in Figure 4.15 below. From this figure it can be seen that at each level, the number of points are changed in an exponential degree.

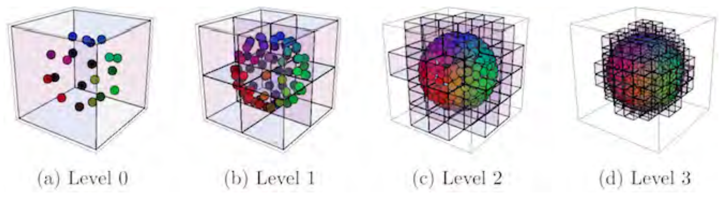


Figure 4.15: How point resolution per node is changed at different depth levels. [23]

The resolution of a node is set by a spacing parameter, that defines the minimum distance between points. As expected, different values of this parameter affect not only the number of points within a node and the final number of nodes, but also the depth of the octree structure. All these factors are directly related to the performance of the system, with several advantages and disadvantages existing for small or large spacing values. The whole process above, is executed through a conversion function, which eventually prepares the raw point cloud data to be stored in this Potree structure. It is also worth mentioning that the points are not stored twice (every point is assigned only to one node) and so, there is no data redundancy there.

Through the functionality that was described above, the Potree viewer manages to render massive point clouds in a very efficient way. Up until today, one of the most demanding and thus, impressive implementations of this viewer remains the point cloud visualization of the Actuele Hoogtekaart Nederland (2), made by the Netherlands eScience Center [11]. The viewer manages efficiently to build and deliver the 3D point cloud model of the Netherlands after having selected which nodes to utilize, out of more than 38 million LAZ files that contain about 638 billion points.

Potree is far from a simple point cloud viewer. There are plenty features offered such as: a) extensive parameterization of the point appearance in terms of color, geometry, properties, quality, background, shadings, etc, b) complete navigation controllers, c) spatial measurement tools (clipping, angle, distance, area, height profiles, volume, etc) and more.

The following two snapshots (Figure 4.16 and Figure 4.17) demonstrate some of the already mentioned features. In both cases, the classification and the elevation are considered by viewing only the building points and coloring them based on their heights. Moreover, in Figure 4.16 a volume clipper has been applied to highlight part of a building, whereas in Figure 4.17, a height profile has been computed and the user can view the properties of each contained point.

iTowns

The iTowns project is a JS/WebGL framework for 3D geospatial data visualization, supporting many different geographical data types (panoramic and oriented images, point clouds, 3D textured models, vector data, etc) and currently maintained by IGN and Oslandia. It supports many spatial measurements via a modern web interface (Figure 4.18), making it a complete package of 3D visualization tools and thus, a very good candidate. Initially, it was designed by IGN (French geographical institute) to combine street images and terrestrial lidar point cloud data that were acquired by its *Stereopolis* mapping vehicle [18]. Currently it is under active development and migration phase from version 1 to version 2.

The capabilities of iTowns are indeed many and not compared to any other online renderer, having the capability to become a complete online GIS platform. Some of the most interesting features (other than simply visualising oriented images, temporal satellite imagery and point clouds) include:

- Support of 3D textured models and annotations.
- Local file access and WFS support. Street view navigation.
- Statistics generation and visualization.

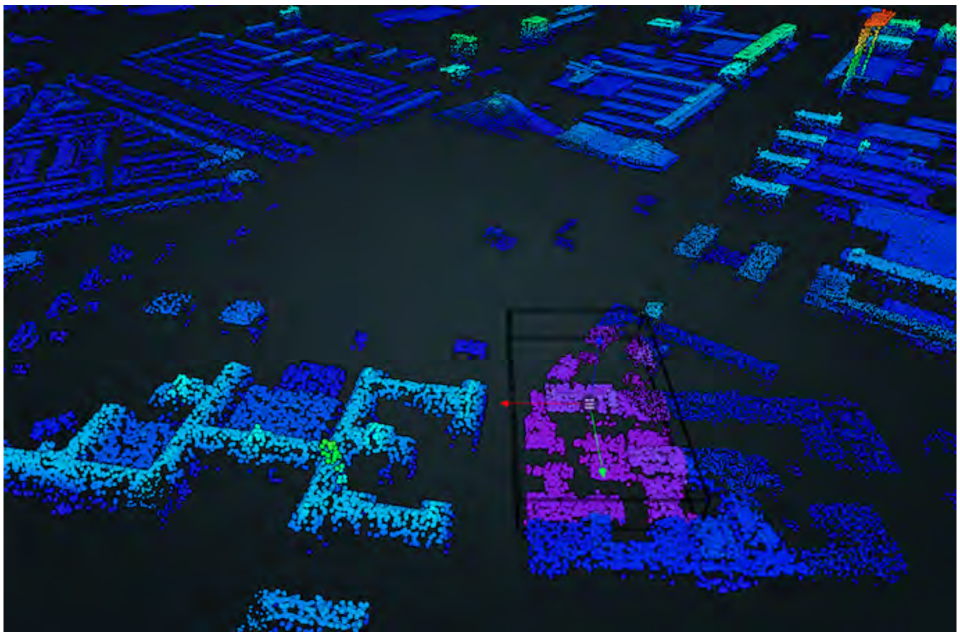


Figure 4.16: In this snapshot, a volume clipper has been used to select part of a building in Delft (Source data: AHN3).

- Flood and loudness simulation.
- Routines for automatic classification.
- A simple API interface..

Even though the list above is not complete, which makes iTowns quite promising, there is a concern about its performance in terms of user experience and speed. Because of the fact that iTowns is a relatively new project, loaded with many functionalities, it seems not ready yet to provide the user with an out-of-the-box clear and user friendly interface. Moreover, because it was originally designed to support many different types of geodata, it is a considerably heavy client. Hence, it is unknown how would it perform while serving trillions of points.

VTK

The Visualization Toolkit (VTK) is an open-source collection of image processing, computer 3D graphics and visualization algorithms by kitware [22], which all together combined, form a powerful framework. It is far from just a simple point cloud viewer. It has been heavily used for scientific visualization and out of all candidates, it retains by far the most sophisticated visualization and processing algorithms. It provides excellent support for vectors, textures, volumetric processing, etc and also advanced modeling techniques that can run in parallel processes.

Comparing to the other viewers, it is not only the oldest software, but also the most mature one. It has developed wrapper mechanisms to broadly provide its powerful engine (written in C++) to other environments such as Java, Python and Tcl. Moreover, VTK has also implemented a JavaScript class library (namely the VTK.JS), which can be integrated into any web application. For that reason, it was included in this candidate list.

The web integration above is done using the WebGL technology, however as the VTK is not directly orientated towards this web-client direction (it does not provide any functional interface out-of-the-box), such implementation demands a lot of work.

LOPoCS

The LOPoCS, standing for Light Opensource PointCloud Server, is an open source point cloud server written in Python with the ability to bridge the PostGIS spatial database extension of PostgreSQL, with (so far) Potree,

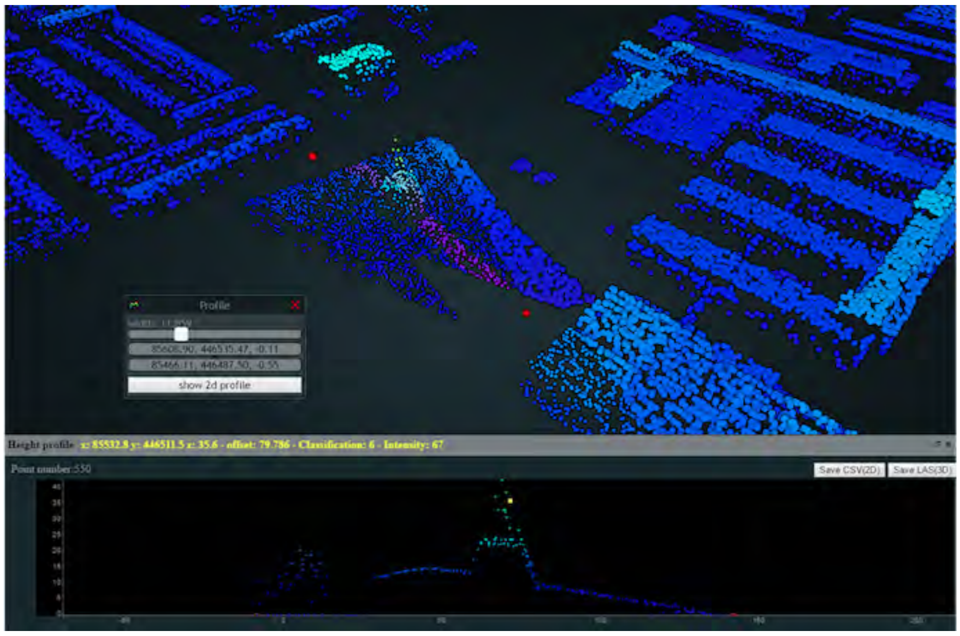


Figure 4.17: In this snapshot, a height profiler has been used on the construction of the TU Delft Library (Source data: AHN3).

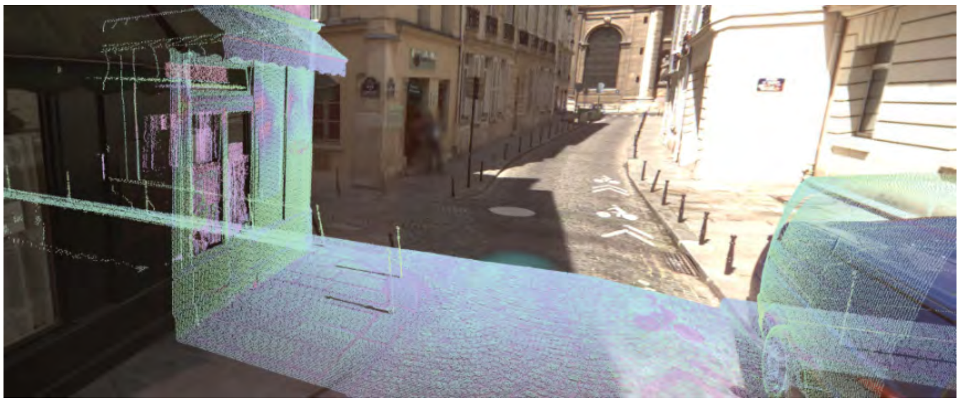


Figure 4.18: A point cloud being projected within iTowns, on top of street images.

iTowns2 and Cesium viewers. It does that by using the pgpointcloud (a PostgreSQL extension for LiDAR data) and the python module py3dtiles. Due to the fact that each viewer has its own communication specifications, the streaming of the points is done via an API that handles this harmonization issue, while everything is being delivered in chunks of compressed LAZ files [17].

This software comes from the creators of iTowns (Oslandia company) and so, the support of other point cloud renderers (other than iTowns) is a quite interesting fact. Although LOPoCS is not a visualizer but a supporting extension, it is still being listed here, as its use might be worth considering.

4.4.3. Implementation

Considering all candidate softwares above, along with the minimum requirements specified in the second section, the most applicable and capable to support the ChronoCity project, proved to be the Potree viewer. Although each one of these point cloud renderers could probably be used as the basis for this online viewer, Potree, compared to the rest of the libraries, offers several key advantages regarding the specifications set. These are highlighted next.

Robust and stable

Potree and iTowns are the renderers offering not only the most modern and user friendly web interfaces, but also the most already built-in spatial functionalities among the software packages, with iTowns having a considerable lead on both aspects. Even though both are active projects with new versions pending, iTowns project feels like it has not only stronger development focus, but also, even more potential to become in the future, the new reference point for online renderers. Nevertheless, due to the fact that Potree is much more *simple* and mature as a software, it achieves great performance and stability. The iTowns web framework seems like it is still under *experimentation* and although plentiful impressive features are being offered, many of them need a lot of improvement (e.g. navigation system, image rendering, point cloud loading). The most worth mentioning issue about Potree's performance has to do with the management of the memory after a prolong usage. It seems like there is a small memory *leak* and the *data-garbage* are not efficiently cleaned in order to support very long sessions. This, however, could be in general a WebGL issue.

Point cloud orientated

Potree was specifically designed to handle point clouds, whereas the rest of the software packages are more generic (and heavy) object viewers. Hence, it offers from scratch more features that are related to point clouds, but at the same time it remains very light-weight.

Proven design

As mentioned before, Potree viewer has already been put to the heavy test (by the Netherlands eScience Center) to handle *more than half of a trillion of points*. This is another important reason why Potree is the best candidate out of all rest. Choosing Potree as the engine of the ChronoCity application introduces several constraints regarding the database model that has to be respected, and also the functionalities that could (or not) be integrated into the web client. The most important is that, the point clouds have to be stored in the Potree Octree format as chunked LAS, LAZ or BIN files. This requires a conversion phase that needs to be done on the server, because otherwise the Potree client itself would have to be hacked for such functionality to be added. Only in case the LOPoCS bridge would be utilized, then the database model could easily change. However, there is a risk factor in such implementation as the final performance would be completely unknown beforehand and thus, that will be avoided.

5 System architecture

Where the previous chapter aimed at exploring and evaluating alternatives, this chapter continues with the implementation strategy for the actual end-product. A lot of considerations for especially the viewer- and data-part are well explored in chapter 4. Therefore, this section will mainly elaborate on *how* it is actually achieved in a practical manner. The section starts with a brief recap on the modularity of the system because this can be seen as the basis for development. The following sections will cover the online viewer, data model and concludes with the overall task flow which also touches upon the client-server communication. Actual development, performance and evaluation of the change-detection algorithm is not covered in this section. Within the system architecture, this can be seen as a *black box* and is also handled like so.

5.1. Modularity

Within the conceptual analysis it is said the implementation of the Chronocity-application is of a modular nature. This is not only due to convenient development in a remote project team, but also enables a better support for future plugins and extensions. When starting development, the modularity as shown in Figure 4.1 is kept like it is. Therefore you could think of the system in four main modules being 1) a convenient online viewer, 2) point cloud isolation based on user-input, 3) a change-detection module handling these isolated datasets and 4) overall data preparation for the online environment. All of these are developed as stand-alone modules at first and some of them merged after reaching a working state.

5.2. Online viewer

The online viewer is the environment where all user interactions take place and is in fact the only place where user input is generated and data is given back. It needs to work flawless and convenient. Like explained in the previous conceptual analysis this needs a proven framework which is also easy to implement within a short timespan. For these reasons, the choice for *Potree* was quickly made. *Potree* handles most of the hassle with the visualization of (massive) point cloud data, where we could focus on achieving a working and convenient user interface.

Potree is completely designed for displaying point clouds, but in the current development phase it is still mainly a *visualization* tool rather than an *analysis* tool. Also, while it is under heavy technical development, it still lacks a intuitive user interface for the generic user. These two aspects - user interface and introducing some (external) analysis functionalities - form the greatest challenges within this module. Luckily, the framework is designed around common standards like *HTML5*, *CSS* and *JavaScript* and is therefore easy to customize. Albeit with an investment of time in discovering the different libraries and *classes*. As a starting point for the development of the convenient interface, some sketches are being made of which one is shown in Figure 5.1. Based on these sketches, extensive customization is carried out to achieve a suitable result which can be seen in Figure 5.2. With the help of *JavaScript* the interface is made even more fluent and intuitive by implying some automation on the interface which has control over the *Potree* canvas on which the actual data is being showed. While designing and developing, particular attention is given to the fact the OPCM is meant for a generic user. Most of the standard *Potree* controls are handled in the background and do not extract attention of the actual interface. Also, most of the interactions are handled in the so-called sidebar. This frame is designed as being easily extensible by third-party modules, without affecting the rest of the interface.

All source-code is made available on *Github*.

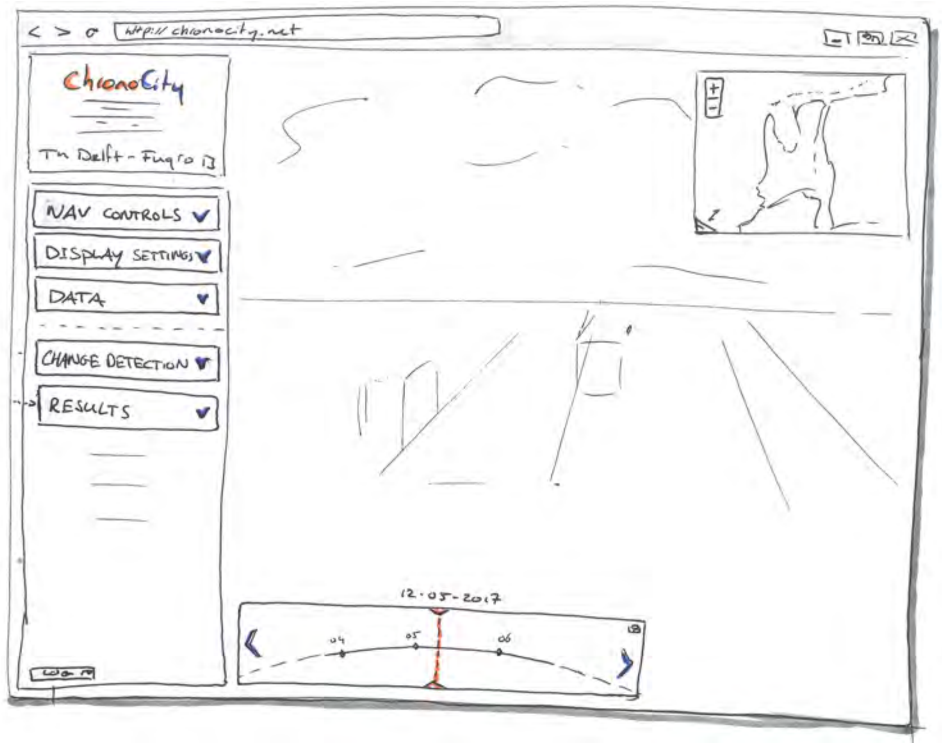


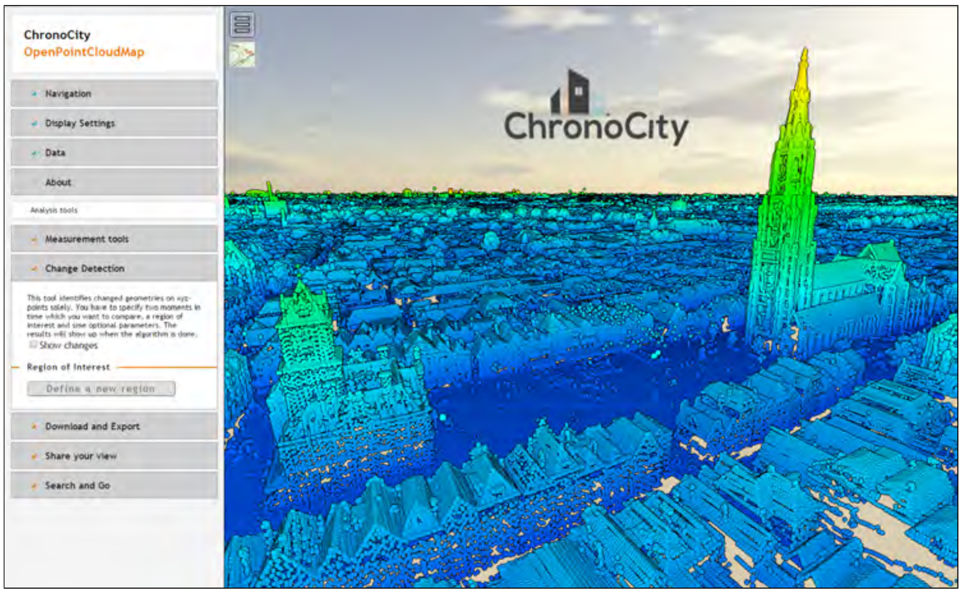
Figure 5.1: One of the first sketches for the user interface. This is an envisioned framework which plays the starting point for implementation in *HTLM* and*CSS*.

5.3. File and data structure

The used architecture as it is developed now, is probably not the best suitable for further development towards our vision of the OPCM. In the previous conceptual analysis this was elaborated on in the sense of *file-* or *database* structures, but also on how to store the actual changed points and its properties. Like with the difference in our vision of the OPCM and what will actually be implemented within the project, within the data model also significant trade-offs are taken. There is a theoretical *ideal* situation, but since the limited timespan also practical implications play a significant role in the development.

The use of *Potree* as the viewing framework is a practical one, it is fully functional in delivering small chunks of point clouds to the client. Where the interface could be simply customized, the portion of handling the requests and delivery is more difficult without any prior knowledge. We therefore need to adapt to an existing file structure which *Potree* can handle. Currently it supports the use of *Greyhound-* and *Arena4D*-databases for streaming the data directly. But still the most-used principle is the use of indexed LAS-files (section 4.3). This is simply because this is the 'proven' method. The other alternatives are still under heavy development and are difficult to get to work or customize and update. Although it has been concluded in section 4.3 that a DBMS would be the best option for streaming (massive) point clouds, in this case the practical solution of indexed LAS-files is implemented. Valuable time could now be spent on visualization, change detection and get the full package to work together. This practical decision has greatly influenced the other aspects of the project. On one hand, file structure (*HTML, JS*) for the viewer is mainly fixed, but now also the point cloud data should be stored in the predefined *Potree* (octree-)structure.

These decisions imply the file structure shown in Figure 5.3 which covers the most important files on the server. Since it is designed as an online tool we can only rely on and control the server-side aspects. Because of the modular development, the file structure on this server can be split up in three parts; 1) the viewer frame-



This tool identifies changed geometries on xyz-points solely. You have to specify two moments in time which you want to compare, a region of interest and sine optional parameters. The results will show up when the algorithm is done.

Figure 5.2: Resulting interactive user interface for the Chronocity-application. It shows the viewport on the right and the sidebar on the left with the change-detection module activated.

work, 2) the processing framework with its algorithms and 3) a data part which is both used by the viewer, but can be updated by the processing framework. Figure 5.3 shows these parts where the viewer framework is mainly stand-alone. Besides it needs access the point cloud data trees it has nothing to do with the files processed by the algorithm. The other parts - data and processing - do heavily communicate with each other; how that is actually done is in-depth covered in section 5.4.

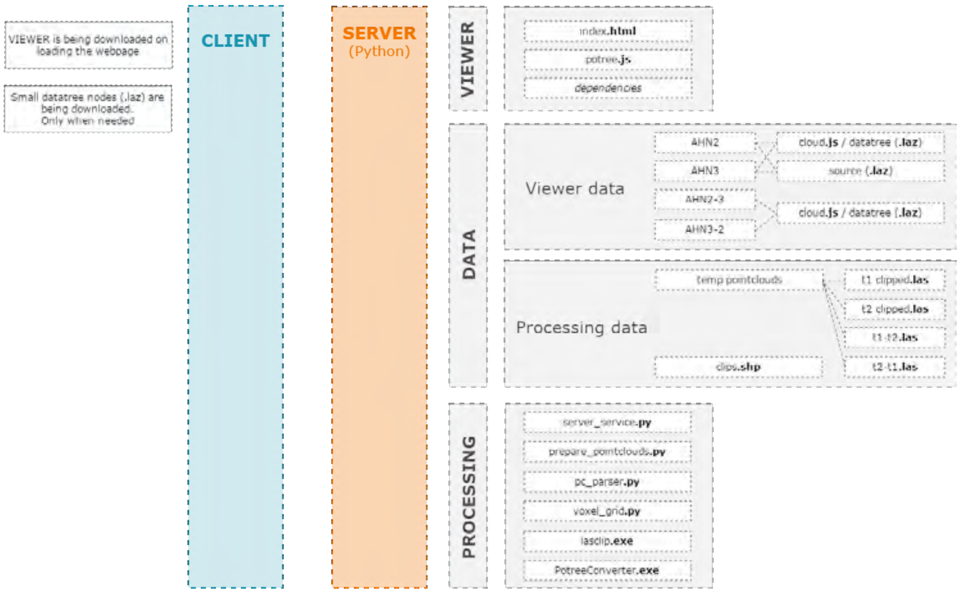


Figure 5.3: Schematic system infrastructure with most important files-structure on the server-side.

As for decomposition, the viewer has its own place together with its (online) dependencies. *index.html* is the web application where a user ends up and initiates all interactions. Then the two most important datasets are the two base maps *AHN2* and *AHN3*. In the current demo, these are the only base maps which can be viewed and analyzed, but this can be simply extended by preparing more. An important notice in this data part, is the storage of not only the *Potree* data tree with its corresponding *cloud.js*, but also the 'raw' point cloud data. Where the data tree is used for streaming information to the client and does not contain áll data, this source data does. It is important the algorithm used this full dataset for its detection because it is heavily dependent on distances (chapter 6) and therefore thinning a dataset has significant impact on the results. Since this full dataset is not used by *Potree* itself, it could be migrated to a database solution but - as for the viewing, practical implications of existing tools for *clipping* the dataset it is now still stored as LAZ files on the same server. The data with the analyzed regions is only stored as the *Potree* data tree since it is not needed to extract or clip data from these. Further consideration about these files are elaborated on in the next section. The other spatial data which is stored on the server consists of some temporary LAS files for keeping track of the algorithm and an ESRI Shapefile which stores the analyzed regions and is used for the clipping procedures (Figure 5.5).

Change storage

In this development phase, the viewer relies on four datasets; two base layers and two layers which store the results. This is mainly due to the practical implications of *Potree* which is able to load multiple datasets at a time, but for convenient development for toggling the times and showing different results it is practical to store all of these in separate layers. The two combined results needs slightly more storage due to their headers and such, but comparing to the overall size this is not significant. On the other hand, these two separate files enables straightforward adaptation into the interactive interface.

The base layers are tried to be kept untouched with respect to their original source. The generated resulting files contains not only the geometry of the changed points, but also an indication of the significance of change. Since the LAS/LAZ format is strictly formulated in sense of the available properties, this indication should fit within this format. The best solution for this would be using the slot *UserData*. It is available for your own definition and has size to put in a distance measurement or a binary representation of for example a vector. However, again practical implementation has been found out to be important. An important point for the entire project is bringing the results conveniently to the user; a bottleneck in the visualization of results is the LAZ format itself. Within *Potree* material/appearance settings are incorporated for a few standard LAS attributes. Unfortunately *Userdata* is not one of them and it has been found out it is very difficult to adapt this within this short amount of time without any prior knowledge on binary parsing. Mainly for this reason, the change measurement is stored as an RGB-value. *Potree* already has implemented this attribute as visualization value and can therefore be easily adapted to. For now, this is a fine solution since the source data does not have meaningful RGB-values assigned. Although they had values which seems to represent fly-paths, their values are set to almost white. In this way, the changed points can have hues of red and green which pop out over the base layers. In this stage, the results therefore do not have any numerical definition of the change measurement; those are parsed to achieve a gradient color in reds and greens.

With this approach, fact is that every analysis between two base layers results in again two new datasets. Although in tested cases the resulting files are not significant in size, this would cause the need for a lot of storage. For example; also having the AHN1 as a base layer and analyzing changes between all possible combinations would result in having six new data sets. A better approach could be appending information on each point for its change over a longer period of time. A major challenge in this is the LAS format itself and its available attribute slots. Also in the future of the OPCM where data sets are rapidly followed up - maybe in terms of hours - this approach could cause significant amount of data. Therefore, and to keep it straightforward changes between to distinct point clouds are stored as can be seen in Figure 5.3.

5.4. Task flow

As already mentioned in section 4.2 user flow and interaction plays an important role in the overall usability of the system. Actual user flow is implemented in the interface according to Figure 4.2 and will not be elaborated on in this section. Best is to explore this yourself and some of the functionalities are explained in chapter 7 of this report. What is actually important to notice is the communication between the client and server and the flow in the background happening upon initialization of the change detection.

Figure 5.4 shows a schematic overview of this communication. It can be clearly seen most of the most-used processes are happening at the client-side except for the change detection. This is of benefit when multiple users access the server at the same time, but also implies heavy computations take place at the users computer and therefore has significant impact on the performance.

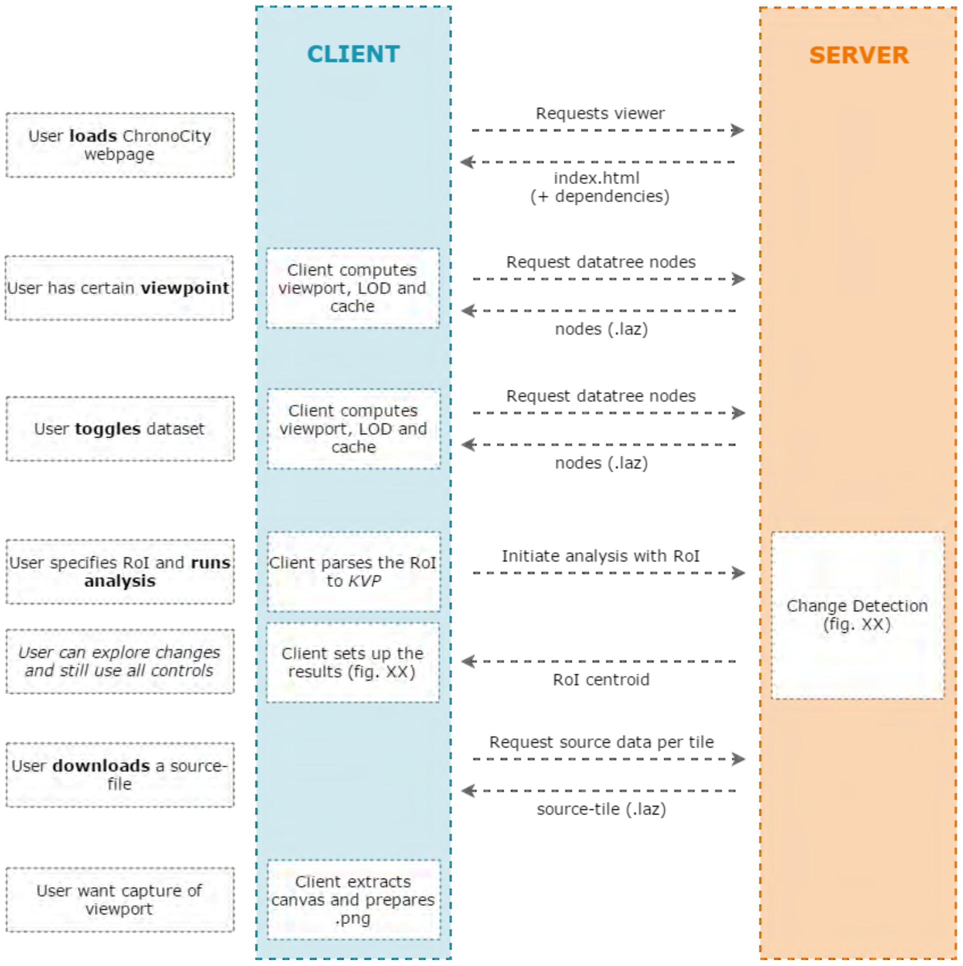

CLIENT
SERVER
User **loads** ChronoCity webpage
Requests viewer
index.html (+ dependencies)
User has certain **viewpoint**
Client computes viewport, LOD and cache
Request datatree nodes
nodes (.laz)
User **toggles** dataset
Client computes viewport, LOD and cache
Request datatree nodes
nodes (.laz)
User specifies RoI and **runs analysis**
Client parses the RoI to *KVP*
Initiate analysis with RoI
Change Detection (fig. XX)
User can explore changes and still use all controls
Client sets up the results (fig. XX)
RoI centroid
User **downloads** a source-file
Request source data per tile
source-tile (.laz)
User want capture of viewport
Client extracts canvas and prepares .png

Figure 5.4: Client-server communication, initiated by user tasks.

The Chronocity-application is initialized by requesting the *index.html* in the client's browser. The server responds by sending the index and the linked files and scripts. At this moment, no point cloud data is being streamed to the client. Then *Potree* scripts computes the viewing position and target of the user. Based on that, the script requests specific data tree nodes which are displayed to the client. This is a continuous process: when the user changes its view or toggles another base layer, *Potree* requests for different nodes. The script which decides on which and how many nodes are returned is actually the power of *Potree* and streaming large amounts of points to the client; it is untouched.

Typical for the Chronocity-application are the processes which take place on initializing the change detection. The user first specifies its region of interest and accepts this. On that moment, a HTTP 'GET' request is being send to the processing-server. This request only contains a KVP with a 3D representation of the RoI-polygon. The processing-server now got the request of doing a change detection and starts processing (which is elaborated on below). Once finished, the server responds to the client with the centroid of the RoI. Now, the client knows it has results and starts preparing the interface for the change-analysis (Figure 4.2). This setup includes changing appearance to RGB-values, zooming into the RoI and toggling base layers. A bottleneck in this approach is the visualization-nature of *Potree*. The change-detection updates the *cloud.js* file in the viewer, but the interface cannot be updated conveniently. Therefore the browser window is in this development stage refreshed without retrieving cached memory.

Where Figure 5.4 includes 'change detection' as a black box, these processed are schematic visualized in Figure 5.5. This process fully takes place on the server, but does not necessarily needs to be placed within the same as the viewer. The only constraint is to have access to the raw data and the *Potree* data tree. The process

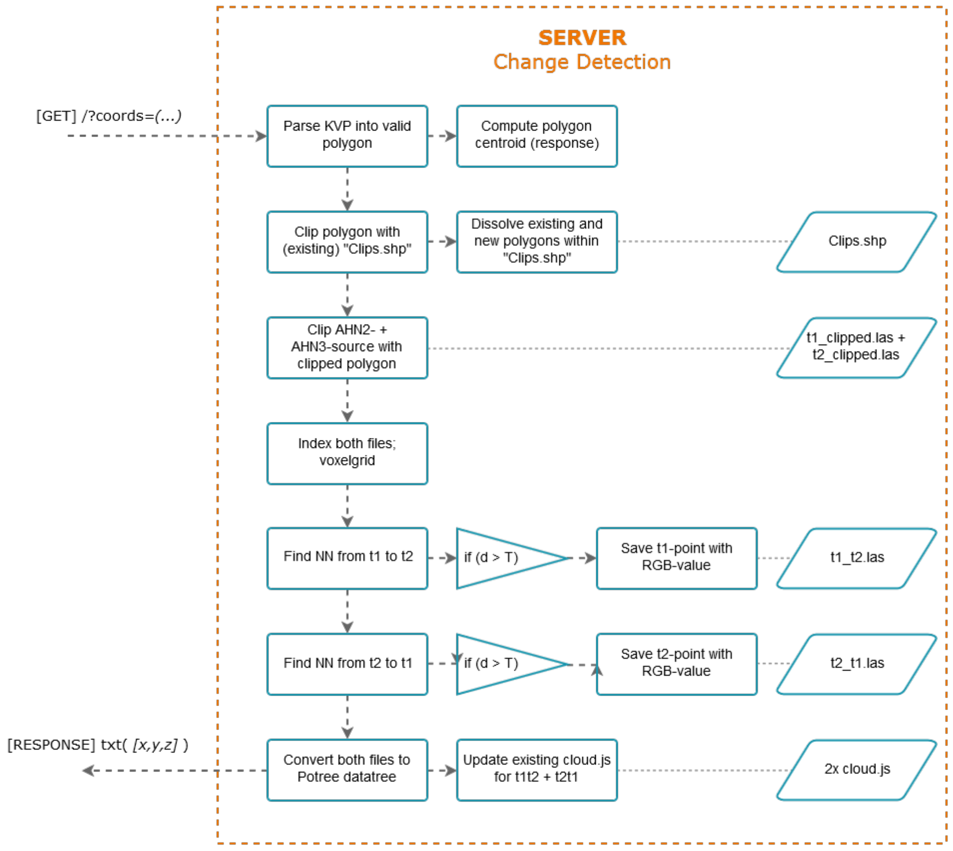
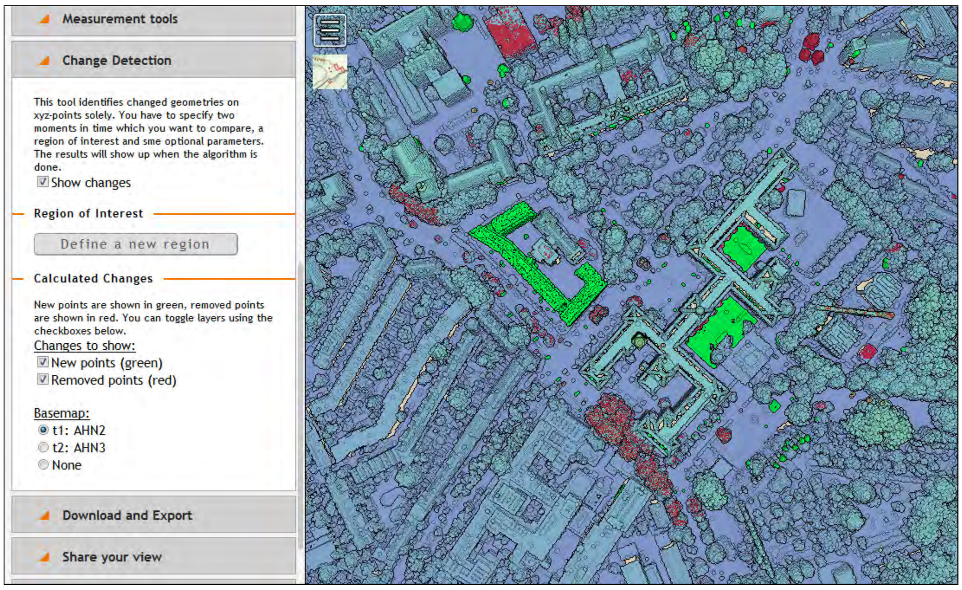

SERVER
Change Detection
[GET] /?coords=(...)
Parse KVP into valid polygon
Compute polygon centroid (response)
Clip polygon with (existing) "Clips.shp"
Dissolve existing and new polygons within "Clips.shp"
Clips.shp
Clip AHN2- + AHN3-source with clipped polygon
t1_clipped.las + t2_clipped.las
Index both files; voxelgrid
Find NN from t1 to t2
if (d > T)
Save t1-point with RGB-value
t1_t2.las
Find NN from t2 to t1
if (d > T)
Save t2-point with RGB-value
t2_t1.las
[RESPONSE] txt([x,y,z])
Convert both files to Potree datatree
Update existing cloud.js for t1t2 + t2t1
2x cloud.js

Figure 5.5: Schematic flow for the change detection process, including both the HTTP request and response as well as the (temporary) files.

starts by retrieving the HTTP-request with a single KVP *?coords=*. The coordinates are parsed into a polygon which is used for clipping the RoI. Before the actual clipping with raw data takes place, the 'new' RoI is compared to *Clips.shp* which contains the area which is already analyzed, saved and available. Simply put; the new RoI is clipped by these areas to make sure no double analysis is carried out and reducing data redundancy. Once clipped and stored as a variable, this new RoI is of course also appended to *Clips.shp* for future analyses. Once the clipped RoI is there, both base layers are clipped by it using LASclip in a parallel process. The resulting clipped files are temporarily stored on the server to conveniently use them in the algorithm, but is also a left-over from the modular development. Next to that, if the following processes would fail it is still available to no redo it from scratch. Then both files are indexed using a regular grid which lowers computational time in the algorithm. Both this indexing and actual neighbor finding is elaborated on in the next chapter. Both neighbor-finding processes would result in the resulting, colored LAS files which are again temporarily stored on the server. These resulting files can already be viewed in many point cloud viewers, but is useless in *Potree*. To be able to use it in the viewer, it needs to be converted and appended to the data tree. This is handled by the PotreeConverter tool, the main job of which is to convert LAS/LAZ files into valid Potree databases. It is worth mentioning that, appending a new point cloud B to an existing Potree point cloud database A, requires that the bounding box of B is included in the bounding box of A. For that reason, the very first (base) Potree database was initialized using a bounding box that included the entire Netherlands. With that being said, the point cloud is converted and appended to the data tree in -incremental mode.

Once finished with this final process, the server responds to the client with the centroid of the RoI as a string-value. During the process, the client has no on-going communication with the server, it therefore has no information in what stage of the process it is. This is not very convenient but again a practical implication of the technologies. Using websockets in stead of HTTP request would solve this problem, but seemed hard to implement and get it to work cross-platform. This implies that the user is able to still use the interface for exploring and other functionalities. Sending a new request to the server would result in a crash, but is worked around by disabling this as long as the client has no response. The centroid-response is used for resetting the user's view to its RoI upon refresh. Also, with the use of Javascript the interface and settings are modified like the user does not have the feeling it was ever away from it. After the process is done and responded, the user can analyze the results, but also start a new request.



This tool identifies changed geometries on xyz-points solely. You have to specify two moments in time which you want to compare, a region of interest and sme optional parameters. The results will show up when the algorithm is done.

☑ Show changes

New points are shown in green, removed points are shown in red. You can toggle layers using the checkboxes below.

Changes to show:
☑ New points (green)
☑ Removed points (red)

Basemap:
● t1: AHN2
○ t2: AHN3
○ None

Figure 5.6: Screenshot of the resulting view when the change detection is finished.

Next to the ability to conduct change detection, there is also the option to geocode, share your view on the OPCM or take screenshots of the viewport. These functionalities are still experimental and can be seen as separate from change detection. Both the screenshots and sharing your view are happening at client-side only. Sharing the view will compose a new URL with three KVP's; camera target, camera position and visible layers. By copying this link and sending it to a friend, he will get exactly the same view as you shared. Within *potree.js* this URL is handled and parsed. Settings are regarding the parameters set to the correct values. For the geocoding use is made of the PDOK Geolocation service. The implementation within the Chronocity-application is still very immature, but does the job. On accepting the search field, it sends a HTTP request to their server and gets a JSON response containing top ten results and their properties. From those, currently only the point coordinate of the best result is taken and parsed into a camera view.

6
3D change detection

The ChronoCity project is a portion of a starting point for the vision of the Open Point Cloud Map and run in parallel with the location and scale/granularity groups. A very big challenge of point cloud processing is to find and implement methods which can handle large (i.e. global) datasets and can do efficient and effective analysis on them directly, namely on-the-fly processing. In this chapter, we will discuss the implemented 3D change detection algorithm and starts from its theory. Why we choose to identify geometric changes and how can we turn the massive change points into more meaningful information. Performance and reliability of the algorithm are also the key concerns of the project and will be addressed in this chapter.

The ideal change detection will be an application specific approach and be capable of identifying meaningful changes like how many newly built buildings in the city and how many were demolished. However, this kind of time series analysis on the semi-structure datasets like point clouds remains a big challenge within the scope of object identification, which still need a lot of post processing like noises removal, clustering, semantic tagging, and advanced machine learning process. Meanwhile, concerning the vision of OPCM and the limited project time frame, our algorithm purely aims at identifying generic geometrical changes between point clouds in an efficient and scalable way

In the meantime, other point information like classifications, colors which may indeed indicate some changes of the surface materials, are also not used in the change detection since in many point cloud sets as well as AHN2, these information are incomplete or missed. Geometrical changes will therefore be the initial step to directly reflect the existing change fact in order to carry out further advanced applications like environmental monitoring, land subsidence monitoring, urban growth examination, construction management and so on.

6.1. Methodology

Three approaches of the 3D change detection has been proposed and discussed in chapter 4. They are 1. Point density comparison 2. Euclidean nearest neighbor and 3. Octree structure comparison respectively. Although, one of the main strengths of the octree structure comparison is that it is fast and allows to detect changes in near real time [19], heavy dependence on the C++ environment and no expertise in this language are the main cause why it is not implemented. We also didn't implement Point density comparison concerning that the discrete and discontinuous effects are not a favorable change detection features. The final but also the most robust option is then the Euclidean nearest neighbor.

6.1.1. Euclidean nearest neighbor

Euclidean nearest neighbor, or more precisely, Hausdorff nearest distance, is a common technique to perform change detection on two point cloud sets [10]. It is also one of the underlying algorithm adopted by the 3D point cloud processing software, CloudCompare, for showing the difference among different point sets. Qin et al. (2016) see the Euclidean distance change detection approach as taking the three degrees of freedom for 3D geometry into account by computing the Euclidean distance of two surfaces, in their normal direction, see Figure 4.5. This approach gives a more accurate result than purely calculating the point distances since inconsistent point coverage of the same object exist in many aerial point clouds like AHN2 and AHN3 because of the different surveying conditions at the moment. Nevertheless, calculating the Euclidean distance from the reconstructed surfaces of the large scale point clouds is a massive computation work and is outside of the project scope. The alternative is simply calculating the Euclidean nearest distance of the compared point to the reference point neighbor.

In essence, the idea of nearest neighbor change detection is that for a given set of reference points $P = \{p_1, p_2, ...p_n\}$ in metric space M and compared points $Q = \{q_1, q_2, ...q_n\}$ and $q_n \in M$ find the element $NN(q_n, P) \in P$ that is the closest to q_n with respect to a metric distance $d : M \times M \to \mathbb{R}$

$$NN(q_n, P) = argmin_{x \in P} d(q_n, x) \tag{6.1}$$

If $d(q_n, x)$ is larger than a certain defined threshold distance, we consider it as a change point.

In order to get the optimum speed of search, simple brute force Euclidean distance calculation is not an ideal option. There have many nearest neighbor algorithms been studied. One of the commonly used techniques is partitioning trees like kd-tree [2] or octree tree. When dealing with large amount of high dimensional data, partitioning trees increase the search performance by subdividing the space and indexing them. Search range is decreased because of this binary tree structure. However, there are many variants for different type of datasets and applications and getting the optimum algorithm parameters is not an trivial thing which is often subjected to manual process [14] .

With regard to this, an open source library: Fast library for approximate nearest neighbor (FLANN) is experimented in the project because of its automated configuration procedure of setting the algorithm parameters.

6.1.2. Fast approximate nearest neighbor search

FLANN is an open source library written in C++ but provides python binding. The biggest advantage of FLANN is an automated configuration procedure for finding the best algorithm to search a particular data set. The underlying indexing approaches are randomized kd-tree algorithm and the priority search k-means tree algorithm. It provides approximate nearest neighbor search because exact search is too costly, non-optimal neighbors in some cases can be orders of magnitude faster than exact search [14] .

The standard kd-tree generally splits the data at median into halves in the dimension with the greatest variance of the data set. Each of the two halves of the data is then recursively split in the same way to create a fully balanced binary tree. The data point is then stored in each leaf node, though in some implementations, each node may contain more than one point. Given a query point (compared point), the first candidate of the nearest neighbors will fall within the first node descent down the binary tree.

However, it is important to know that the first candidate is not necessary the nearest neighbor; it must be followed by a process of backtracking in which other nodes are searched for better candidates. This backtracking process is often the reason why calculation takes long especially in high dimensional datasets. To overcome this issue, approximate search terminates the search after a specified number of nodes are searched. By limiting the amount of backtracking, the certainty of finding the absolute minimum is sacrificed and replaced with a probabilistic performance [24] .

In the FLANN implementation, multiple randomized kd-trees, which the split dimension are chosen randomly instead of only chosen from the highest variance, are constructed , and approximate search is performed in parallel of these multiple trees. To put it simply, by setting up multiple random decomposition trees, the probability of finding the nearest neighbor of the query point being in the same cell is increased as shown in Figure 6.1.

FLANN also implements another algorithm: the priority search k-means tree, for finding the approximate nearest neighbor when higher precision is needed. It starts by partitioning the data points at each level into K distinct regions by means of k-means clustering and repeats the procedure in each region till the number of points in a region is smaller than K as shown in Figure 6.2, where K is called branching factor and is critical for obtaining good performance.

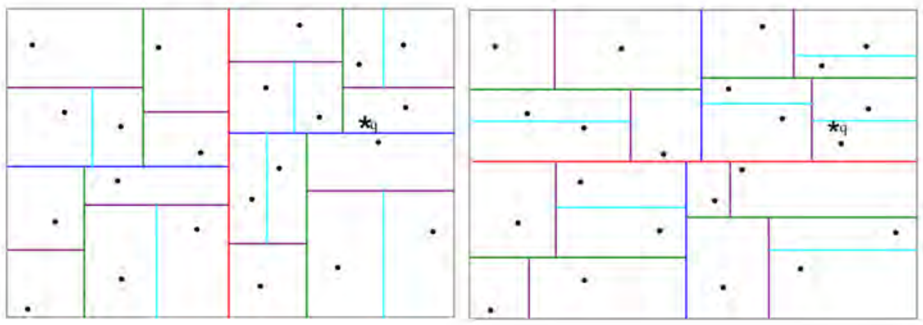


Figure 6.1: In the left side kd-tree decomposition, the nearest neighbor is across a hyperplane from the query point, which means additional backtracking is needed, but in the same cell in another decomposition as shown on the right side. Image source: [14]

When perform the nearest search, it starts from the closest leaf, and adding all the unexplored branches along the path to a priority queue. This queue is then sorted in increasing distance from the query point. Then the algorithm resumes this traversing and always starting with the first branch in the queue. The nearest neighbors are found when the number of points in a queue is smaller than K

FLANN has been implemented and experimented in our algorithm to perform change detection because of its automatic configuration feature. Figure 6.5 illustrates the initial small scale performance test we have done. It performs very well when the number of points is small; however, the processing time still increases dramatically in a larger dataset. Although FLANN already has automatically indexing function, the result of this test motivates us to implement grid indexing, which subdivides the space with equal size grids, and FLANN will perform the nearest neighbor search in each grid separately. The result of this experiment is presented in Figure 6.6 and shows a significant improvement. However, it is still not efficient enough to carry out the change detection in even larger datasets concerning the scale of OPCM.

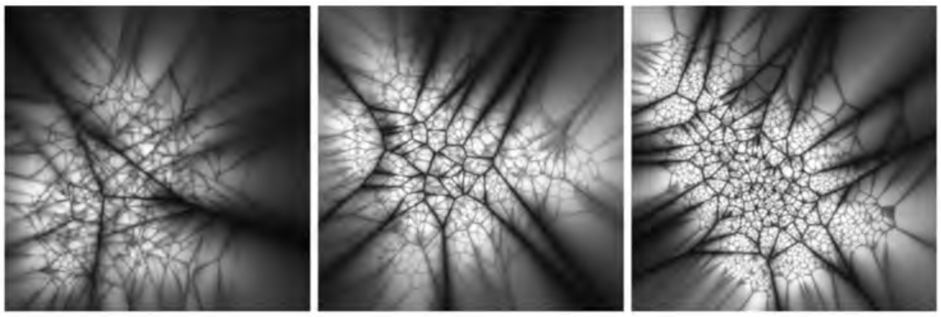


Figure 6.2: Projection of different branching factor K of the different decompositions of the space. K = 4, 32, 128 respectively. Image source: [14]

6.1.3. Grid nearest neighbor search

After the testings with FLANN, we feel a faster approach is needed. We therefore design and develop a grid nearest neighbor search. Like most of the nearest neighbor search algorithms, our algorithm partitions the space occupied by the t1 and t2 points with 3-dimensional grid implementation. The grid size is also the defined threshold distance. Nearest neighbor calculation is performed within in each indexed grid. To speed up the calculation, every distance calculation takes the square root (to get a true Euclidean distance for storage) only when $d(P_c, P_r)^2 \geqslant threshold\ distance^2$.

Less neighbor distances need to be calculated for each compared point (query point) in each indexed grid significantly increase the speed. Furthermore, the assumption is that in most of areas, points of different time frames are mostly overlapped, with only a small distance change, because there are no changes happened. This is saying that in each indexed grid, the compared point quickly find out the neighbor distance is less the threshold, therefore it is unchanged, and can move on to the next compared point. When no neighbor exists in the current grid, the search will expand onto adjacent grids until the nearest neighbor is found. The concept of this algorithm is projected on a 2D grid and shown in Figure 6.3

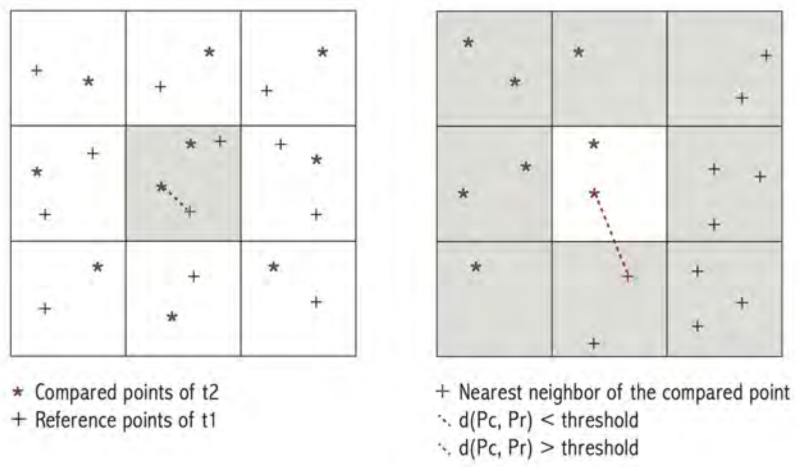


Figure 6.3: When t1 and t2 points are mostly overlapped, most of the distance calculations are done within a single initial grid since the closest neighbor distance is less than the defined threshold; If no neighbor exists in the current grid(s), the search is expanding to the adjacent girds and only stop till the nearest neighbor is found.

This approach significantly reduces the amount of calculations need to be done for each compared point. However, chances are that the exact nearest neighbor is not within the current search grid(s) but within the adjacent grids, see Figure 6.4. To overcome this issue, an ideal scenario will be that the search of the nearest neighbor is always completed only after the next level grids of current grid(s) are explored. However, according to our experiments, this will increase the calculation time by 50 percent. This fact discourages us from implementing this robust mechanism but only carries out the cubic expansion when no neighbor is found in the current grid(s). Namely, this algorithm is also an approximate nearest neighbor for the sake of performance issue.

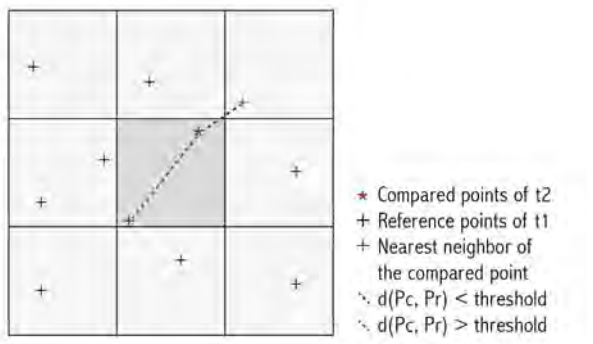


Figure 6.4: TIn this example, true nearest neighbor is not within the initial search grid (medium gray). Therefore, the initial search area of the query point is set to be 3x3x3 grids and expand to the next level if no neighbor is founded.

6.2. Performance analysis

The aim of these tests is trying to understand whether the algorithm can process large scale point clouds efficiently, to the scale like city of Delft or an entire Hague area, and should still produce reliable results which will be discuss in the next section. All of these initial performance tests have been done on the MAC OS X with

8 GB 1867 MHz DDR3 memory and with 2.7 GHz Intel Core i5 processor.

The small scale performance testings of the open source library FLANN has been shown in Figure 6.5. Although FLANN has superior performance regardless of the threshold distances and when the number of points is less than one million, the processing time is increased dramatically over this range. We have also experimented with implementing simple grid indexing. Figure 6.6 shows this experimental result and the performance improvement is significant. Nevertheless, it is still not the optimum solution to process point cloud datasets.

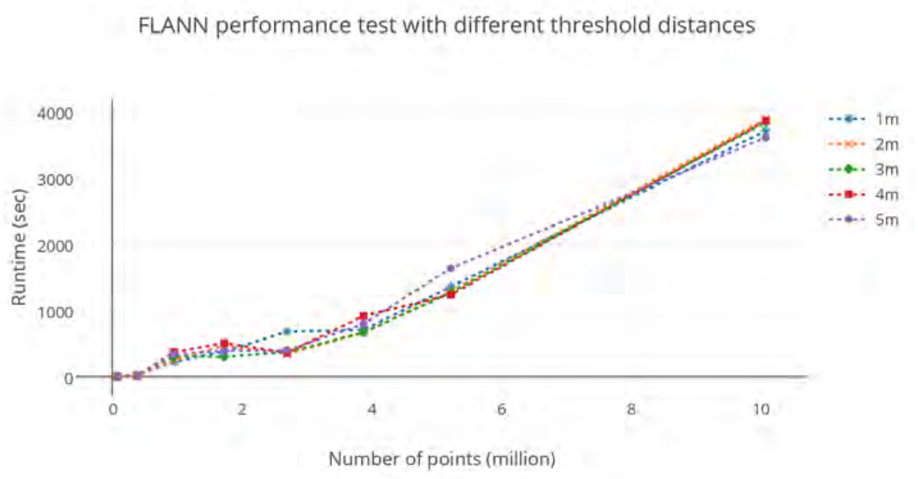


Figure 6.5: FLANN performance test with different threshold distances

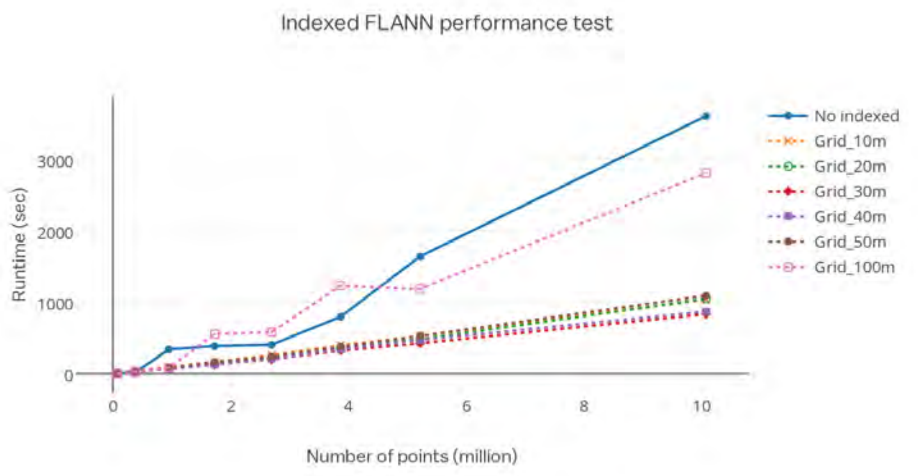


Figure 6.6: Indexed FLANN performance test with different grid sizes.

On the other hand, grid nearest neighbor search is very sensitive to the threshold distance since it directly affects the grid size and thus has an impact on the processing speed. In this small scale performance test, we have found the algorithm has the best performance when the threshold is set at 5m, see Figure 6.7. In general, good performance of the grid NN search is why this is the final implemented algorithm.
Nevertheless, it is important to note that trying to increase the performance in the python environment has certain limitations and can never be comparable with the statistic complied type language like C language. Consequently, our focus has always been using the existing libraries and resources to build up and design our algorithm.

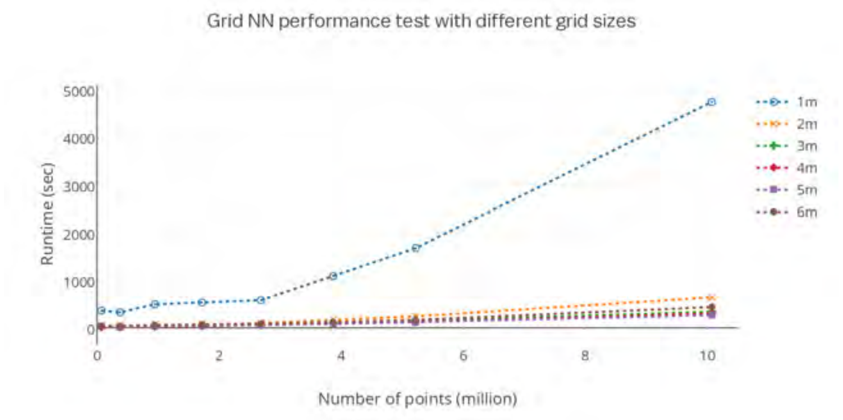
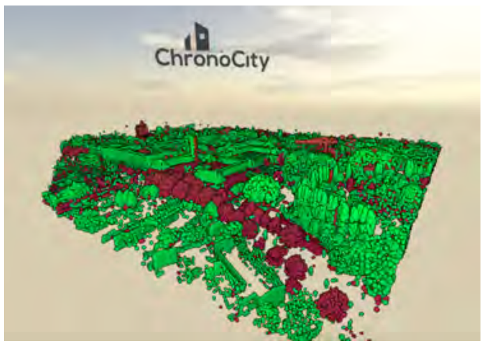


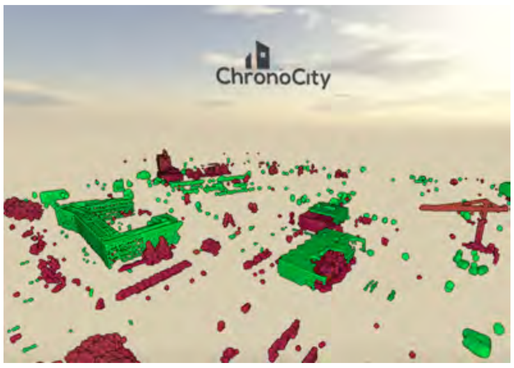
Figure 6.7: The performance of grid nearest neighbor algorithm in general is much faster than FLANN and indexed FLANN. This difference is more significant in even larger datasets. However, grid NN is comparatively sensitive to the threshold setting and becomes dramatically slow with a small threshold distance.

6.3. Reliability analysis and identifiable changes

The final implemented algorithm is grid nearest neighbor search. With a small distance threshold setting, nearest neighbor search can perform reliable change detection where many of the geometrical changes as well as noises are detected and shown, see Figure 6.8a. However, detection overloaded will thus become an undesired effect. To keep the user experience of the ChronoCity application simple and straightforward, and concerning the algorithm efficiency, we have set the threshold distance at 5m as a default value. Although this setting sacrifices some detail changes, only the significant changes are presented enables cleaner user interface and is also more user friendly as can be seen in Figure 6.8b.



(a) Threshold distance set to be 1 m. Changes are everywhere.



(b) Threshold distance set to be 5 m. Only significant changes are detected and shown.

Figure 6.8: Green represents new points and red for the removed points. For simplicity, only the detected change points are shown.

One of the most important characteristics of the Euclidean nearest neighbor change detection is that in some cases, new points, defined as t2 - t1, are less visible than the removed points, defined as t1 - t2. This situation can be best illustrated in Figure 6.9. For instance, when t2 points are collected from the footprint of the demolished building, these changes are less complete because the nearest neighbors (t1) of the compared points (t2) usually lie directly around building footprint, which means only the very center points on the footprint with the nearest distances larger than the threshold will be identified.

From land changes to water body is another interesting change detection case. For most of aerial point clouds, water surface is not recorded due to the insufficient energy of LiDAR pulse after penetrating the water body. This phenomenon indicates that if the area of interest covers water body in t2, there are no points

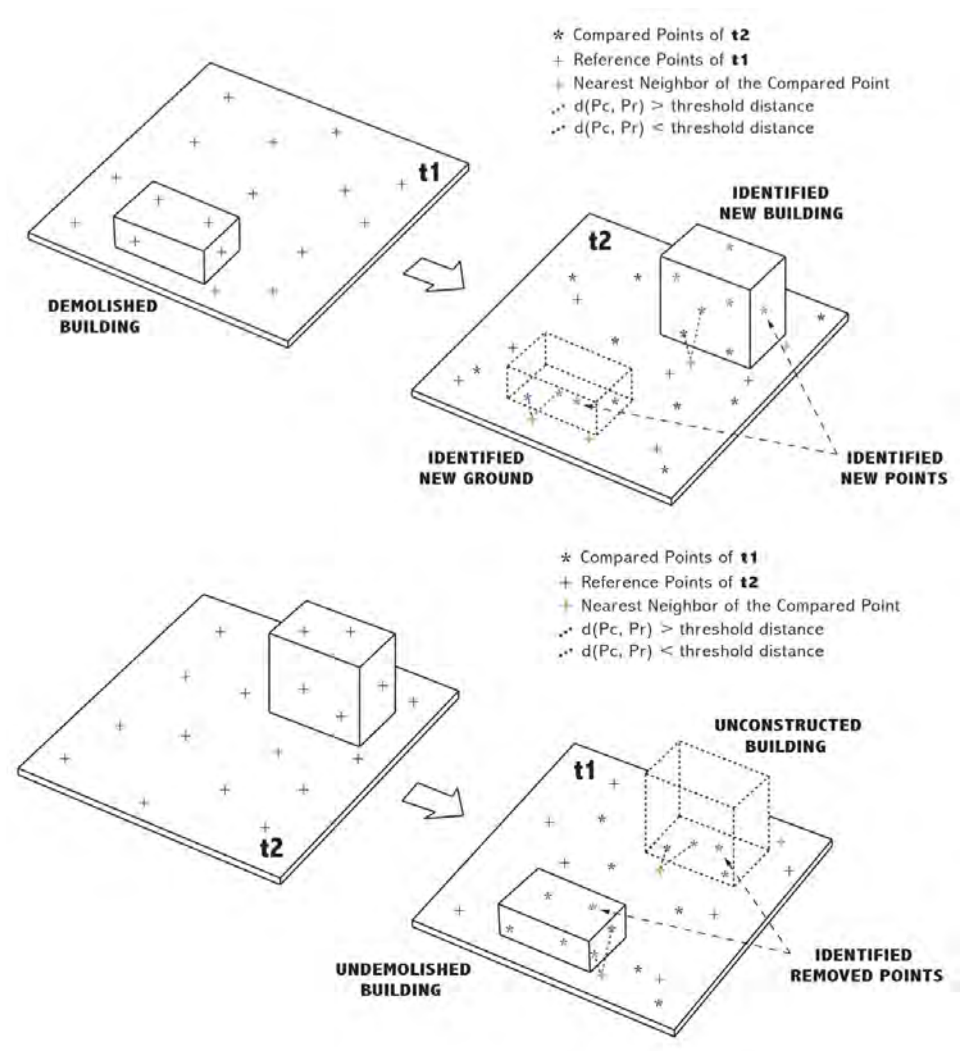


Figure 6.9: To obtain the generic geometrical changes of two point clouds, both new points and removed points will be detected in the algorithm. New ground points on the footprint of the demolished building shows the tricky case of nearest neighbor change detection.

available for performing change detection. However, this problem can be solved if we perform the change detection as t1 - t2, which will result in the removed points.

On account of this, to obtain a more generic change detection, new points and removed points are equally important and both are calculated in our application, see Figure 6.9.

Another issue comes with change detection of aerial points clouds is false positive and negative changes. This means in some occasions, detected changes are caused by uneven point densities of t1 and t2 over the same area caused by different viewing angles of different surveying conditions. Therefore, we have found many objects, especially the building walls are very common, are detected as changes but not true in reality, see Figure 6.10. This remains one of the biggest challenges to be solved if we want to detect changes by automated algorithm. Meanwhile, Euclidean nearest neighbor is also very sensitive to outliers.

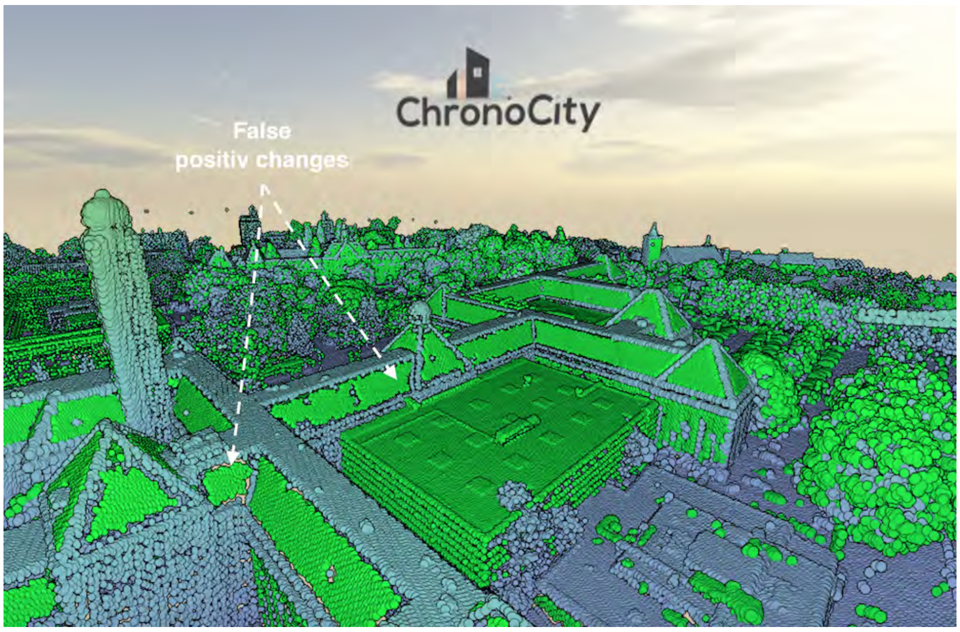


Figure 6.10: Green represents new points (t2 - t1), and false positive changes are detected on the rooftop of the Faculty of Architecture, TU Delft. The threshold is set to be 1m for clearness.

7 Functionalities and use cases

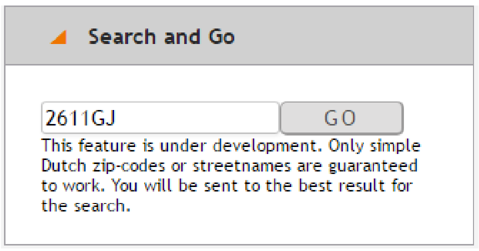
This chapter contains two main sections. The first one has the purpose of showing some examples of the viewer performance such as getting in touch with the user friendly interface, discover the main functionalities and play with different materials and textures. The description is enriched with screen shots in order to help the reader getting to know the final product. Then, the second section enumerates a number of use cases for which the ChronoCity viewer can be useful for in the fields of urbanism, engineering and the environment.

7.1. Viewer functionalities

The ChronoCity interface allows the user to perform a set of simple operation to obtain the changes between two selected datasets. The main functions include: visualization of the area of interest, selection of the area in which the changes will be computed, data download, selection of the datasets to visualise and shared view with other users.

Visualization

The first thing that one can do when opening the viewer is 'play around' with the zoom, rotate and pan commands and get a general idea of the contents. The interface has been developed keeping the user needs in mind so a strong focus is given to the ease of use. In order to zoom to an area of interest two ways are possible: manually (zoom and pan) or inserting a *geocode* in the 'Search and Go' tab (Figure 7.1).



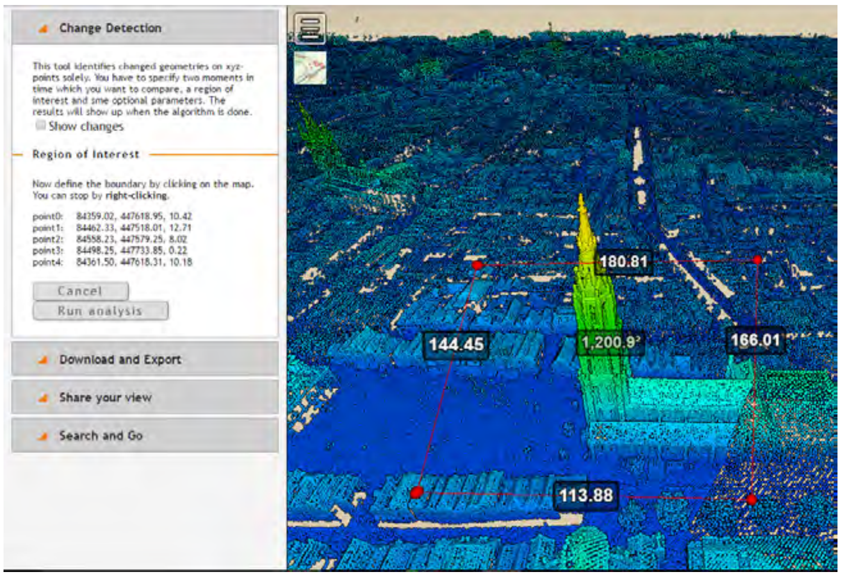
This feature is under development. Only simple Dutch zip-codes or streetnames are guaranteed to work. You will be sent to the best result for the search.

Figure 7.1: Search and Go tab.

This operation is very straightforward and the viewer will immediately point the user to the right location. Nevertheless, this feature is not complete because the dataset is taken from PDOK so only Dutch zip-codes and street names are guaranteed to work properly.

Selection of the area of interest

Once the user has individuated the desired location, there is the possibility to draw a polygon enclosing exactly the area of interest needed, this functionality is called *clipping* and allows the user to clip a chunk of the point cloud. It is important to keep in mind that self-intersecting polygons are not valid, so the user should be careful in drawing the perimeter of the polygon since it could compromise the clipping operation. Figure 7.2 shows the clipping tool in action.


Change Detection

This tool identifies changed geometries on xyz-points solely. You have to specify two moments in time which you want to compare, a region of interest and sme optional parameters. The results will show up when the algorithm is done.

☐ Show changes

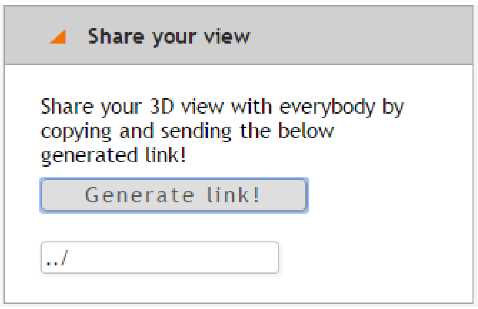
Region of Interest

Now define the boundary by clicking on the map. You can stop by **right-clicking**.

```
point0:   84359.02, 447618.95, 10.42
point1:   84462.33, 447518.01, 12.71
point2:   84558.23, 447579.25, 8.02
point3:   84498.25, 447733.85, 0.22
point4:   84361.50, 447618.31, 10.18
```

Figure 7.2: Example of clipping area.

We can see that while drawing the region of interest, the coordinates of each corner point of the polygon are also displayed in the lateral menu. This information will be sent to the server when the change detection is activated. In addition, the user can choose to share his view with other people simply by generating the link in the 'Share your view' tab (Figure 7.3).


Share your view

Share your 3D view with everybody by copying and sending the below generated link!

Figure 7.3: 'Share your view' tab.

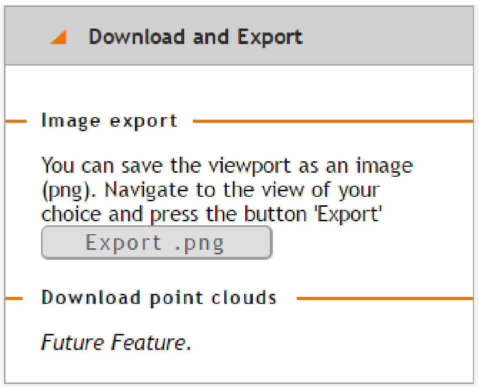
Run and export

In the envisioned OPCM, multiple datasets gathered from different periods of time will be presented and the user has to specify two moment of time (i.e. two datasets) to compare. However, in the current development, the available datasets are only AHN2 and AHN3. The change detection result of the area of interest will show up only when the algorithm is complete, so it is important to keep in mind that the waiting time varies according to the dimension of the region of interest. However, by toggling the 'Show changes', the users can also see the preprocessed changes which have been done by others.

An additional functionality would be the possibility to download the changes. By now it is only possible to export the changes in .png format, but a future improvement would be the possibility to download the point cloud dataset containing the changes (Figure 7.4).

7.2. Potential use cases

The ChronoCity viewer is able to individuate, highlight and monitor every kind of change over a specific period of time on the surface of the Earth (such as buildings, roads, street furniture, forests, mountains, etc.). There is a multitude of applications for which the viewer can be crucial for. As a consequence, also the user



You can save the viewport as an image (png). Navigate to the view of your choice and press the button 'Export'

Future Feature.

Figure 7.4: 'Download and Export' tab.

groups are extremely diverse spanning from governmental bodies, academia, construction companies, environmental bodies and ordinary citizens that are curious about our brand-new viewer. Below a sequence of use cases and user groups will be explained in detail to give an idea of the versatility of the viewer. However, it is important to keep in mind that the changes detected are bigger than 5 meters by default, so applications requiring very detailed changes are not included in the use cases. Nevertheless, the settings could be potentially changed in the future in order to reach a higher resolution for applications requiring a higher level of detail.

7.2.1. Urban planning use cases

In the field of urbanism there are a lot of applications that include the detection of changes in the built environment. Monitoring the changes over time can help to identify urbanization patterns, control soil consumption, monitor volume changes within a city, keep an eye on the subsidence of the terrain and assess the damages after an environmental disaster.

Urban growth and soil consumption

Cities are constantly expanding in terms of territory and number of inhabitants and for the first time in the history most people on Earth live in cities [27].The patterns of expansion are unpredictable, they can be both in terms of volume of existing buildings and in terms of growth towards the outskirts of the city. The ChronoCity viewer can help the user to visualise those patterns and use them to make policies preventing uncontrolled sprawl. Users such as governmental bodies can monitor the changes and take measures accordingly, while the academia can study the urbanization patterns theories stemming from the observations made by the viewer.Moreover,by identifying the directions of the expansion, urban planners can locate functions accordingly, for example in case a new residential area is built within the city, it will be necessary to locate also services for the citizens such as hospitals, supermarkets, shopping malls, parking and make sure that the amount of green space is enough with respect to the number people living in the area.

Another use case which is strictly connected to urban sprawl, is the prevention of soil consumption. In recent years soil consumption saw a rapid enhancement due to the construction of new buildings in areas that before were 'green'. The soil is a not renewable source and therefore, it must be protected from negative impacts for sustainable use for humans, animals and plants [25]. In this case the ChronoCity online viewer can be used by policy makers and planners to individuate the new building popping up and take measures to slow down (or even stop) the phenomenon.

Volumetric changes in buildings

The detection of volumetric changes in buildings is useful for comparing the volume to the total usable square meters of floor area which provides a metric of building efficiency. For example, between two buildings with the same volume, the one with more square meters (i.e. more floors) seems far more cost-efficient. Our viewer can highlight both negative and positive changes, i.e. volume increase and decrease in case of buildings. So it can be a tool to have a first glance of the variation over time of the building volumes.

Assessment of damages after an earthquake or a landslide

Scanning an area before and after a calamity event (like an earthquake or a landslide) is helpful to generate a 'picture' of the situation and to assess the damages for the subsequent reconstruction. Assuming that the datasets contained in the ChronoCity viewer are like the ones mentioned above, once the calculations are completed, the buildings that are damaged (in case of an earthquake) or the dimension of the material moved (in case of a landslide) can be easily identified. Therefore, urban planners and environmental bodies can use our viewer to estimate the extent of the damage and start the recovery/rebuilding procedures accordingly.

7.2.2. Environmental - engineering use cases

The possible use cases of the ChronoCity viewer expand also to the field of engineering and of the environmental protection. Also in this case, a number of applications can be listed, among them the monitoring of the growth of the trees, the damages caused by a fire in the forest, the control of the coastal erosion and the movement of sand piles along the coast.

Three growth monitoring

It is of primary importance to monitor the growth of the trees both in rural areas and within the cities. In rural areas the monitoring is needed to assess the health of the vegetation and to check whether interventions are needed to protect the forest to disappear. On the other hand, within cities, trees growth should be monitored mainly for the safety of the citizens and to make sure that the branches do not expand too much and interfere with buildings and with high-voltage cables. The ChronoCity viewer is a tool capable of visualizing and highlighting the areas of growth and make sure they do not interfere with the surroundings.

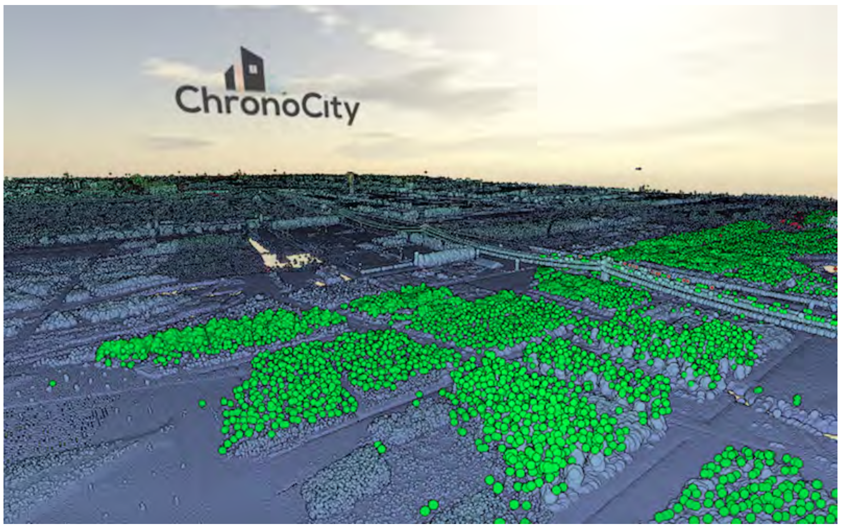


Figure 7.5: Using Chronocity application to identify tree growth zone near the outskirt of city of Delft, Netherlands

Fire damage recovery

Another use case, still connected to the monitoring of the trees is the assessment of the damages caused by a fire in the forest. By scanning the area before and after the disaster it is possible to highlight the areas of change and arrange the planting of new trees where needed (Figure 7.6). Once again, our viewer can be useful to detect the most damaged area and plan an appropriate intervention to restore the forest with new trees.

Coastal erosion monitoring

Coastal erosion can be a natural phenomena or a man-induced activity, in both cases it is important to keep an eye on it to make sure that the phenomenon does not become harmful causing a big environmental damage [8]. For this reason environmentalists could use our viewer to survey the coasts and spot extreme cases of coastal erosion.

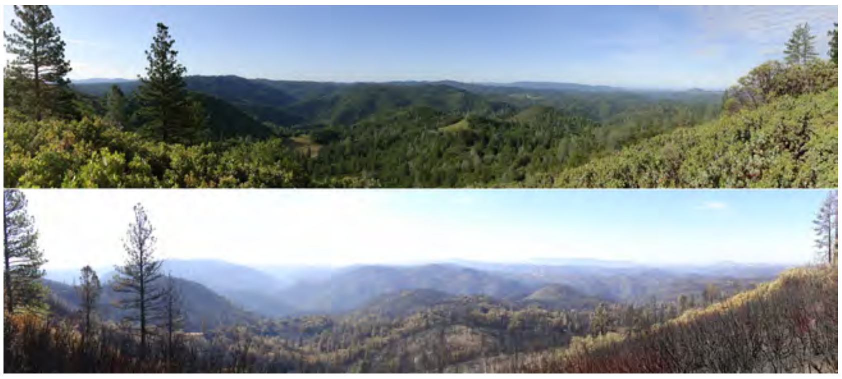


Figure 7.6: Changes in a forest before (above) and after (below) a fire. Image source: [30]

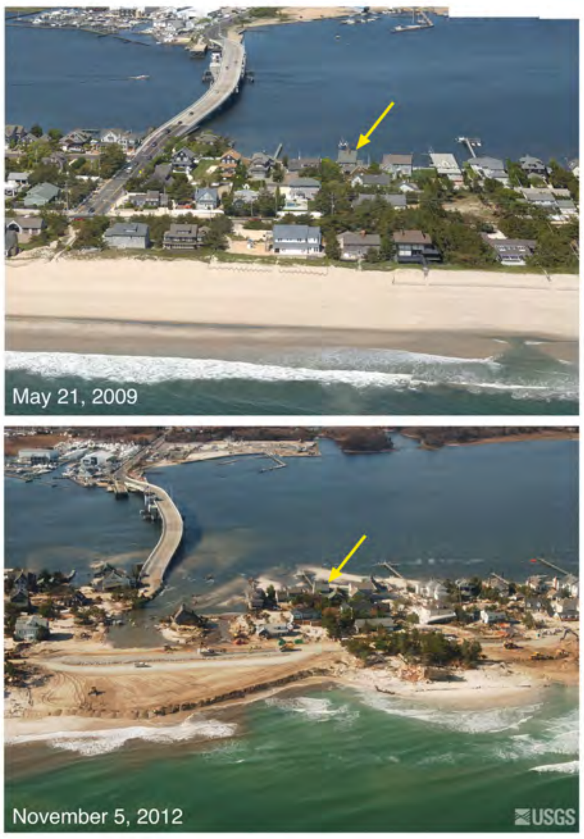


Figure 7.7: Example of coastal erosion. Image source: [12]

8 Conclusions

Conclusively, in this chapter, the ChronoCity project is going to be briefly evaluated in several aspects regarding its development and final implementation. In overall, we can say that the final product does cover the set project's goal, which is the design and development of a 3D change-detection algorithm, the results of which should be displayed in an interactive and user-friendly viewer. Going one step further, however, the final implementation not only covers all the «MUST» requirements, that were prioritized within the MoSCow-rules diagram, but additionally all the «SHOULD» and even most of the «COULD» aspects too. The approach to each «COULD» feature is discussed shortly below.

Process on national scale dataset; Assuming that the corresponding base point clouds (e.g. AHN2, AHN3 etc..) are available on our web server, then the application does support at the moment change detection for the entire Netherlands. Any future detected changes are appended (without data redundancy issues) to the previously detected changes.

Efficient retrieval and data visualization; The application was tested using a custom internet connection throttling of 1.5Mb/s (3G speed) and the data retrieval rate was more than sufficient. Furthermore, for a nice user experience and smooth operation, a system having at least 3GB of available Ram is recommended, while based also on the CPU capabilities, the maximum number of loaded points can be selected by the user.

On-the-fly data processing; In the final implementation, all operation components have been integrated in a single (fully automated) processing-line. This starts from the user region selection, up until the visualization of the new detected differences.

High performance; The longest and "heaviest" operation is the change detection processing running on server and, as it has already been mentioned, big efforts have been made towards developing a fast algorithm. Nevertheless, for an even better performance the system needs to be scaled and re-optimized, which is something that will be later discussed in more detail.

CRS Harmonization | Adaptability with the other groups; A hypothesis was made that the CRS harmonization will be handled by the PC Coord team. In general, however, as each team faces an entirely different aspect of the problem, it is indeed hard to develop (as a whole) a single compact and adaptable solution.

Identify useful changes; By the name of the application alone (i.e. ChronoCity) a user might expect that it is designed to be solely useful for urban analysis. Although urban areas are probably most important, because the majority of people live there, nevertheless, this service could be used (as mentioned in the previous chapter too) in many more user cases.

With the above being said, the general evaluation of the final application is considered to be quite positive. Most requirements have been successfully covered and the application from theory, was even transmuted into a published web service running on www.chronocity.net. However, there are some specific (technical) aspects which should be respectively mentioned and evaluated. These will be discussed below.

Potree as the Viewer Engine
As it has already been mentioned, the fact that the ChronoCity application is based on the Potree Viewer, introduced several constrains in terms of (future) code customizations. One of the most important ones is the fact that the database can only be of a specific format (LAZ/LAS/BIN). In order to integrate another type of database, such as a custom one which might support a common world-CRS (PC Coord project), lots of “hacking” needs to be done on the parser of the viewer. Contrariwise, using an open source software with many developers to support it, has some straightforward advantages. Possible bugs can be easier and quicker to identify and fix, while additionally, support from more experienced developers can be valuable too.

Data Redundancy
One critical aspect that the team had to consider during the design and development of ChronoCity application was the data redundancy and how to avoid it as much as possible. Point clouds can be of massive size and so, storing same chunks multiple times can be problematic for a database. Although one solution could be to update existing points (which are in LAZ format) and store there the detected differences as point attributes, this approach could not effectively support scalability. For example, it should be possible for a base layer (e.g. AHN2) to be compared with more than one layers (e.g. AHN3, AHN4, .. AHN10). However, all the information about these differences cannot easily fit into the initial LAZ file (e.g. AHN2) and so, this approach had to be rejected.

In the end, to store the points that had changed more than our set threshold, was inevitable. However, our tests showed that comparing to the size of the base point cloud, the size of the produced point cloud (including only the changed points), was in average very small (60-80 times smaller). Yet another problem regarding the data redundancy was the repetition of similar/overlapping user requests. The algorithm should not only avoid recalculating the same areas, but also avoid storing the same detected differences multiple times. To handle this problem, after every user’s change detection request, the server first compares the selected area with all previously made requests (saved in a shapefile) and only their difference is used for the new clipping process. Then, the initial selected area is dissolved with the clipping-history file, for all future requests.

Grid Indexing
To make change detection calculations run faster, a three-dimensional grid indexing was used. Although this methodology does offer a good searching performance, the closest point is not always correctly identified. The closer a point to the border is, the bigger the probability that its real closest point is lying in another grid. Because it is important this error not to affect our change threshold, the size of each grid has been set to be the same as the change detection threshold (5 meters). Because of that relation and due to the fact that the searching process always starts at a 3x3 grid-neighborhood (expanding only if no points were discovered), this error never leads to accepting as non-changed, a changed point. It might only affect the accuracy of the estimation of the changed distance and that should be mentioned.

System Stability
It is a fact that one of the most common (and at the same time worst) problems during the development of a software, is the so-called “bug”. Code that is not understandable by the machine, leading to instability. Assuming that during the server/client production, the team had to write more than a thousand lines of code at different technical layers and “stacks” (using CSS, HTML, JS, Python, JSON, XML, Bash etc), then the possibility of a single bug is expected to be high. However, in favor of a reliable system and functionality, the team during the development handled most of the possible exceptions that could threaten the stability of the system. With that said, all data processing and flow (between client/server) were tested and found to be running reliably. Even if a user, after sending a change detection request to the server, would close his client, the server would still continue with the given task until the results are saved.

At this point, the evaluation of the project is concluded and based on all aspects discussed above, it can be stated that the initial purpose statement was overall achieved. At the same time and through a fruitful collaboration, the team gained valuable experience in the context of this synthesis project. Not only in “purely” technical terms (geodata acquisition, visualization and processing), but also in terms of project management and decision making, interaction with professionals, presentation.

9 Recommendations

In this final chapter, a few logical and practicable suggestions will be provided for both the server and the client. These could not only enhance the performance and usability of the developed ChronoCity application, but also improve the user experience of its future operators.

Further improve change detection performance

One of the main research goals defined in the ChronoCity project is to identify an effective way to detect changes within 3D point cloud datasets. In our approach, the closest distance to a point within the same neighborhood at the comparable point cloud, is calculated. This distance is then compared to a chosen threshold (i.e. 5m), which rejects or accepts the given point as changed. Although this methodology is quite simplistic and prone to many parameters (e.g. broken line of sight during scanning on one of the two point clouds), it does offer a good general estimation. This process is currently being done on all points within the selected area, producing a new point cloud dataset that is complete (contains all changed points) and available to the users. However, which users do really want that detailed information?

Since, there is an important tradeoff between performance (in terms of execution time) and number of points being searched, it might be clever to identify how complete this change detection search should be. One of the reasons why CloudCompare performs better than our grid nearest neighbor search algorithm, is because it uses a thinned version of the point cloud. This introduces of course the problem of identifying the best sampling algorithm. Nevertheless, in our case too, doing a sampling before the change detection processing might be a good approach. Lastly, it should be mentioned here that the point clouds being generated by the PotreeConverter are normally already thinned. Hence, using those for change detection, might also be a solution.

Post-processing of detected changes

After changes have been calculated, they are stored and become available to the users. However, a more sophisticated post-processing filtering can also be easily applied to our implementation. Of course, its importance has to be evaluated along with the extra time cost that will be introduced, but some approaches may include:

- The identification of the most important changes (e.g. exclude small/unimportant items that add unnecessary noise to the outcome). A good candidate for such filtering might be the (very fast) density-based spatial clustering of applications with noise (DBSCAN) along with a minimum number of points threshold.
- The correction of (false-positive) changes due to scanner's blocked line-of-sight.
- Different thresholding based on the classification of the point.

Possibly one of the most crucial aspects when evaluating the ChronoCity software, is the performance of the server, where most of the calculations are executed. The recommended system specifications for a client to run smoothly have already been mentioned. Although these specs can nowadays be easily found on common computer systems, a server capable of hosting the ChronoCity change detection engine, demands much faster and thus, expensive hardware. In general, the design of our algorithm, the number of change-detection requests and the size of the selected areas, determine what exactly is needed.

Split each request into optimized point chunks

Regarding the developed change detection algorithm, it is designed to heavily utilize the computer's RAM by loading all objects for quick referencing. Additionally, although most of the change detection engine is in Python, not only the calculations are powered by the low-level Numpy and LASTOOLS libraries, but also, several heavy tasks (like clipping) are divisible into different CPU processes (multicore design). During the development, the profiling of the code suggested that the RAM resources (and so the execution time) are consumed by the number of points in a nonlinear way. For example, in our benchmarking tests, although a system of 32GB Ram could (on average) check 52m points in 14 minutes (42k points/second), the same system could (on average again) only check 15m points in 10 minutes (25k points/second), due to the GRID Indexing that takes place before the change detection. However, if the points exceeded the RAM capacity, then there was a RAM bottleneck heavily affecting the performance (in these cases the hard disk's pagefile was utilized, but its IO speeds are far from comparable to a RAM). All above lead to the fact that the number of calculated points can and should be optimized. Even if a user selected a very big area, the points within it should be split into smaller optimized chunks.

Scale the performance using multiple processes

The previous recommendation was about optimizing the point cloud chunks. Nonetheless, the next logical step is to divide them into different parallel workers (processes) and thus, utilize the CPU as much as possible too. At the moment, our implementation is heavily RAM orientated, executing parallel processes only for LASTOOLS (as they are already designed to use minimal RAM and mostly CPU resources). However, after having optimized the point chunks and due to the fact that our regular grid indexing can easily support task dividing, parallel workers can be initiated.

Change detection scheduler

When a system has no tasks to perform, an idle "blank" process is only consuming the resources. Considering that as an opportunity, the server could be scheduled to pre-calculate differences that might interest future users. These could be big urban areas, areas where lots of requests have already been received but are still not completed, etc.

Bridge the service to the existing AHN2/AHN3 databases

One of the core features being currently offered by the ChronoCity viewer is the option to download the original point cloud datasets. Openlayers are used as a basemap and there, all available tiles are accessible for downloading. To have a reference for these tiles to the original AHN2/3 database is just a matter of a link. That would also solve the problem of storing "again" the entire AHN2/3 database on the ChronoCity server. However, in this approach there is a problem.

During the change detection process, the base point clouds need to be quickly retrievable by our engine. Therefore, either the converted Potree database should be used instead of the original tiles (as proposed in the beginning of this chapter), or the engine should be able to rapidly get the requested tiles. Due to the fact, that the point clouds might each time exceed tenths of gigabytes of size, the previous means that the engine should either be physically connected to the original database, or lie behind a gigabit connection with it.

The following recommendations concern particularly the client-side and thus, are more directly linked to the user experience. For that reason, they are equally important to consider.

User point cloud upload

Right now, the Potree viewer offers to the users 2 point cloud base maps, the AHN2 and AHN3. However, for a complete functionality, the application should be able to accept new point clouds, that are imported by the users themselves. This demands some interface modification, because the user should be able not only to view the basemaps of his choice, but also to select them for change detection. It should be mentioned that this dynamic selection is already supported by the server.

Show already analyzed extents on the map

When a change is calculated, it then becomes available for all future users to view. However, it would be quite helpful for them to be able to easily locate (on the map) the extents of all the previously calculated changes. For example, if a user would navigate to a region that has been processed but no changes were found, then he might expect that this region has not been processed at all.

Sub-change detection based on classification

Right now, the change detection considers the whole point cloud and thus all classes. That information is saved within the new point cloud and thus, the users can activate/deactivate in the produced point cloud, their preferred classes. Assuming a user was interested to detect changes only in a specific class (e.g. buildings), then to have a choice to send to the server a sub-change detection request, might be very useful as results would be generated much faster (and possibly not have to wait for 1,2 or even more hours).

Statistics, filtering, annotations

Another feature that would be very convenient to the user would be the generation of quick statistics. For example, a user might be interested to filter the changed buildings and calculate the percentage of urban growth. These changes could be visualized either on screen (using graphs or annotations), or the user could even download a generated statistical report.

ChronoCity viewer documentation

For either a simple user or an advanced one (who might even want to install the application), a well-structured documentation of how to use (or setup) this application might be very handy. Tips and HOW-TOs could be also helpful. Lastly, as this web service was designed to be open for the public, a multilingual coverage might also make sense.

Deal with bugs

Finally, one of our most strong recommendations would be to first deal with already existing bugs. Not necessarily coming from us, but also other dependencies. For example, Potree has a known WEBGL issue of memory “garbage” management/cleaning. Namely a memory leak problem, a solution to which, might make the browsers be responsive for much longer. Another issue with Potree is that it favors the Chrome browser. In the other browsers, it does not perform that well or at all.

Bibliography

[1] American Society for Photogrammetry and Remote Sensing (ASPRS) (2010). *LAS SPECIFICATION VERSION 1.3 – R11.* `http://http://www.asprs.org` [May 17, 2017].

[2] Bentley, J. L. (1975). *Multidimensional binary search trees used for associative searching.* IEEE Transactions on Pattern Analysis and Machine Intelligence, (36(11)):509–517.

[3] Brown, G., Butler, H (2012). *Laspy 1.2.5 Documentation.* `http://laspy.readthedocs.io/en/latest/` [May 17, 2017].

[4] Cesium Consortium (2011). *WebGL Virtual Globe and Map Engine.* `https://cesiumjs.org/` [May 18, 2017].

[5] CloudCompare Version 2.6.1 (2016). *User Manual.* `http://www.cloudcompare.org/doc/qCC/CloudCompare%20v2.6.1%20-%20User%20manual.pdf` [May 17, 2017].

[6] CycloMedia Technology B.V. (2015). *CycloMedia support research proposal H2020 FET nD-PointCloud.* `http://nd-pc.org/documents/Cyclomedia.pdf` [May 20, 2017].

[7] Euclideons Unlimited Detail (2015). *Euclideon udWeb Test Application.* `http://udserver.euclideon.com/demo` [May 20, 2017].

[8] Flanders Marine Institute (VLIZ) (2008). *Natural causes of coastal erosion.* `http://www.coastalwiki.org/wiki/Natural_causes_of_coastal_erosion` [June 20, 2017].

[9] Kitware. (1998). *Visualize Your Data With VTK.* `http://www.vtk.org/` [May 20, 2017].

[10] Marc, R., Thibault, G. (2005). *Change detection on points cloud data acquired with a ground laser scanner.* International Archives of the Photogrammetry, pages 30–35.

[11] Martinez, O. R., Verhoeven, S., Van Meersbergen, M., Schütz, M., Van Oosterom, P., Gonçalves, R., Tijssen, T. (2015). *Taming the beast: Free and open-source massive point cloud web visualization.* `http://repository.tudelft.nl/islandora/object/uuid:0472e0d1-ec75-465a-840e-fd53d427c177` [May 20, 2017].

[12] METEORÓPOLE (2012). *Sandy: antes e depois.* `"http://meteoropole.com.br/2012/11/sandy-antes-e-depois/` [June 20, 2017].

[13] Mirage-Technologies (2014). *PointCloudViz.* `http://www.pointcloudviz.com/`.

[14] Muja, M., L. D. G. (2014). *Scalable Nearest Neighbor Algorithms for High Dimensional Data.* IEEE Transactions on Pattern Analysis and Machine Intelligence, (36(11)).

[15] Muja, M., Lowe, D. G. (2013). *Fast Library for Approximate Nearest Neighbors.* `http://www.cs.ubc.ca/research/flann/` [May 17, 2017].

[16] Oosterom, P. V., Martinez-Rubi, O., Ivanova, M., Horhammer, M., Geringer, D., Ravada, S., Gonçalves, R. (2015). *Massive point cloud data management: Design, implementation and execution of a point cloud benchmark.* Computers and Graphics, (49):92–125.

[17] Oslandia. (2016). *LOPoCS : Stream your Point Cloud from Postgis.* `http://www.oslandia.com/lopocs-stream-your-point-cloud-from-postgis-en.html`.

[18] Picavet, V., Bredif, M., Konini, M., Devaux, A. (2016). *ITowns, framework web pour la donnée géographique 3D.* Revue XYZ, pages 49–52. `http://recherche.ign.fr/labos/matis/pdf/articles_revues/2016/XYZ147_Juin2016_iTowns.pdf` [May 17, 2017].

[19] Point Cloud Library (Ed.). (2011). *Spatial change detection on unorganized point cloud data.* `http://pointclouds.org/documentation/tutorials/octree_change.php`.

[20] Qin, R., Tian, J., Reinartz, P. (2016). *3D change detection approaches and applications.* ISPRS Journal of Photogrammetry and Remote Sensing, (122):41–56.

[21] Scheiblauer, C. (2014). *Interactions with Gigantic Point Clouds (Doctoral dissertation, Vienna University of Technology).* `https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/scheiblauer-thesis-thesis.pdf` [May 20, 2017].

[22] Schroeder, W., Martin, K., Lorensen, B. (2006). *The Visualization Toolkit.*

[23] Schuetz, M. (2016). *Potree: Rendering Large Point Clouds in Web Browsers (Unpublished master's thesis). Vienna University of Technology.* `https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/SCHUETZ-2016-POT-thesis.pdf` [May 20, 2017].

[24] Silpa-Anan, C., H. R. (2008). *Optimised KD-trees for fast image descriptor matching.* 2008 IEEE Conference on Computer Vision and Pattern Recognition.

[25] State Office for the Environment, Agriculture and Geology - Germany (2017). *Environment - Soil Protection and Soil Consumption.* `https://www.umwelt.sachsen.de/umwelt/en/14148.htm` [June 19, 2017].

[26] Stowers, J. (2013). *Python Bindings to the Point Cloud Library.* `http://strawlab.github.io/python-pcl/` [May 18, 2017].

[27] Swilling, M. (2016). *The curse of urban sprawl: how cities grow, and why this has to change.* `https://www.theguardian.com/cities/2016/jul/12/urban-sprawl-how-cities-grow-change-sustainability-urban-age` [June 19, 2017].

[28] Veesus LTD (2017). *Powerful Software Solutions for all Point Cloud data.* `http://veesus.com/veesus-arena4d-software/` [May 20, 2017].

[29] Verma, U. (2015). *In-browser LAS/LAZ point cloud renderer.* `https://github.com/verma/plasio`.

[30] World trippers (2008). *September 21, 2015 - The Butte Fire: Before and After.* `"http://www.worldtrippers.com/05house/15/l150921.htm` [June 20, 2017].

[31] Xu, H., Cheng, L., Li, M., Chen, Y., Zhong, L. (2015). *Using Octrees to Detect Changes to Buildings and Trees in the Urban Environment from Airborne LiDAR Data.* Remote Sensing, (7(8)):9682–9704.

A

Media outreach strategy

This project is carried out with two main aspects in head. One of them being the technical part in which the generic change detection has been developed and implemented in an online viewer. Besides developing this in an academic way, it also contains a large portion of a learning-curve. The other main aspect is practically the main reason why Fugro and TU Delft initiated the OPCM project; making massive point clouds available to the public. This second aspect is not solely on his own, it could be a great practical framework for academics and technicians to make on-the-fly analyses available to everyone. Therefore, the media outreach strategy aims at two different target-groups being 1) the general public who should be acquainted with the 3D environment and 2) the academics who should be aware of the current possibilities and the proposed framework for on-the-fly analysis.

General public

As we speak today, the general standard for visualizing geo-information are 2D maps and features. Slowly the trend is going into the 3D geo-field as technologies enable streaming this massive amounts of data to the client, enabling even more possibilities. Take for example the streaming of meshes and textures in the recently launched *Google Earth Online*. Next stage in these developments is up to *ChronoCity* and its full 3D *xyz*-world.

For the general public, the *ChronoCity* web-application is designed for ease-of-use and mainly focuses on exploring the world around them. Main goal for this target-group is reaching as many users as possible, this way fundings for further development or extensions on the framework can be justified easier. Enabling users to actually úse the application is dependent on three aspects; availability, features to be used and ease-of-use.

Starting with availability the most straightforward issue is having the application up and running on a server which is able to fulfill the potential demand on storage but also processing capability. This can be costly, but processing unit can be easily implemented at Fugro's or TU Delft own services. While the OPCM increases in data size and potential user uploads, this actual data can possibly be hosted across multiple servers from different companies or contributors. By decreasing costs and dividing storage across multiple companies, this can be a valuable investment in the future of 3D world and the companies' public relations strategy. Therefore (hosting-)companies will be invited to reserve storage on their servers and contribute to this ongoing developments. Then for accessibility it is important that users know the application exists and where to find it. The first user group should first be activated by *us*, which can be yielded by demonstrations on-site at schools or congresses and (online) promotions. For this purpose, the poster is also designed as being *activating* and should excite users to browse to the application. After the initial user group is happily using the application, hopefully the word will be spread through mainly social media and conversations. For this reason, the capabilities are also extended with a 'Share your view'-module in which users can simply generate a link with a specific view and configuration. This link can then be shared among friends or pages to again activate new users.

Of course, exploring a 3D point cloud world is nice but it is not likely users keep continuously exploring the same data over and over again. Not only new data should be added at a regular pace, also features and modules should be extended and developed continuously. By for example incorporating road-networks also routing can be done within the framework. Such type of feature will generate a continuous flow of visits because of the user's needs. Comparing to a 2D routing application, the application ads the ability to fly-through your route beforehand (and thus achieving more spatial reference). Another extension could be the ability to embed certain areas and the viewer on a company's website. Contact information usually contains a section with a 2D map of the location. *ChronoCity* enables the embedding of for example the companies headquarter, indicating entrances and parking lots. While the application and its modules are under continuous development, still the ease-of-use should be guaranteed. For this reason, online demonstration videos will be made available next to a textual description of available tasks and workflows. Still, for an application like this it should be noted that using it should be intuitive; when it seems to not work as expected the page is closed fast enough.

Academics and developers

ChronoCity needs continuous development of features and technical improvements. Besides that, *ChronoCity* is built as a demonstrational framework and it could result in not being the final implementation of the OPCM. Still by activating companies and academics, it serves the goal of making clear what is possible in an online environment. For these academics the currently implemented change detection algorithm is inferior to the framework. This framework is actually activating use while the change detection delivers hand-on results increasing the curiosity even more.

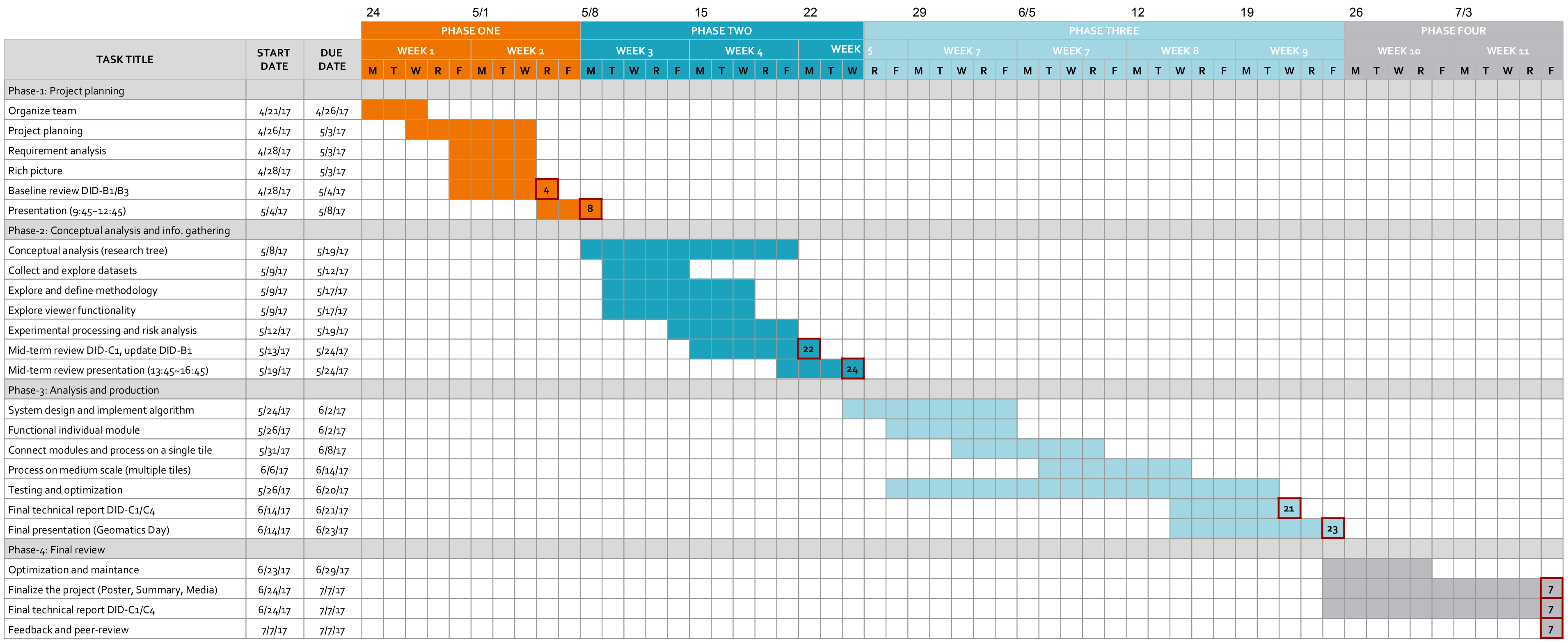
Because academics also belong to the general public, above statements also apply to them in terms of accessibility and ease-of-use. The difference between the two user groups is that the academics should be drawn to the back-end and should be activated for developing new features and algorithms. For this reason an extensive API should be written for the basic *Potree* features while it is currently lacking any explanation. Developers and academics will be invited to cooperate through e-mail and company visits, but also through the application itself by inviting them to develop new modules and redirecting them to the API. With these invitations, they will also be redirected to the technical report and executive summary to get a more thorough understanding.

While extending the current *ChronoCity*-application is nice for the general public, it can also be valuable for developers. It can bring complex algorithms directly to the user, delivering feedback for future improvements or debugging. This direct connection between developers and users will hopefully generate enough value for the companies to make hosting available, but also activates other parties to contribute to the OPCM with datasets. It is actually this continuous flow between the parties which yield value for both.

Materials

Demo	www.chronocity.net
Demo video	www.youtube.com/watch?v=lqZBHaU7G08
Poster	wvanopstal.stackstorage.com/s/NtYNt8IhLEMA83J
Sourcecode	www.github.com/willemvanopstal/ChronoCity
Sample data	*see github*

B
Project planning



TASK TITLE	START DATE	DUE DATE
Phase-1: Project planning		
Organize team	4/21/17	4/26/17
Project planning	4/26/17	5/3/17
Requirement analysis	4/28/17	5/3/17
Rich picture	4/28/17	5/3/17
Baseline review DID-B1/B3	4/28/17	5/4/17
Presentation (9:45–12:45)	5/4/17	5/8/17
Phase-2: Conceptual analysis and info. gathering		
Conceptual analysis (research tree)	5/8/17	5/19/17
Collect and explore datasets	5/9/17	5/12/17
Explore and define methodology	5/9/17	5/17/17
Explore viewer functionality	5/9/17	5/17/17
Experimental processing and risk analysis	5/12/17	5/19/17
Mid-term review DID-C1, update DID-B1	5/13/17	5/24/17
Mid-term review presentation (13:45–16:45)	5/19/17	5/24/17
Phase-3: Analysis and production		
System design and implement algorithm	5/24/17	6/2/17
Functional individual module	5/26/17	6/2/17
Connect modules and process on a single tile	5/31/17	6/8/17
Process on medium scale (multiple tiles)	6/6/17	6/14/17
Testing and optimization	5/26/17	6/20/17
Final technical report DID-C1/C4	6/14/17	6/21/17
Final presentation (Geomatics Day)	6/14/17	6/23/17
Phase-4: Final review		
Optimization and maintance	6/23/17	6/29/17
Finalize the project (Poster, Summary, Media)	6/24/17	7/7/17
Final technical report DID-C1/C4	6/24/17	7/7/17
Feedback and peer-review	7/7/17	7/7/17

C

Performance test

FLANN Performance test with different threshold distances								
Points (ML)	0.078388	0.384031	0.950422	1.726824	2.694288	3.87379	5.221704	10.078696
1m	2.77952814	20.8843291	228.545063	381.717324	464.327004	707.018024	1357.45762	3699.8056
2m	2.73766589	18.4662271	249.523816	465.290654	343.102483	671.01777	1286.30471	3879.59856
3m	2.7664578	17.9327271	305.808019	301.9628	367.597127	664.317624	1270.68101	3831.51652
4m	2.75077391	19.5193031	372.21327	505.467192	355.527093	914.893647	1233.99662	3858.19985
5m	2.80977488	17.4517579	339.124372	385.128013	402.777019	796.991714	1635.03195	3599.84729

Indexed FLANN Performance test with different grid sizes								
Points (ML)	0.078388	0.384031	0.950422	1.726824	2.694288	3.87379	5.221704	10.078696
10m	8.59865999	36.6771648	89.2797649	161.160886	257.133104	400.287111	531.949271	1039.50884
20m	6.22690201	32.183368	80.6211891	142.738088	218.285608	362.295279	476.570539	1045.48037
30m	4.02778697	26.3274319	74.0240688	123.385541	200.191415	326.104548	427.05317	834.70263
40m	4.28711605	22.798038	62.0437961	130.914461	213.772122	344.224407	471.879088	865.908694
50m	3.95889616	25.8085499	76.2528429	162.142648	236.359093	381.701562	531.491191	1092.93999
100m	3.55991697	28.378283	78.8935099	553.841228	578.578861	1227.1249	1183.11043	2803.25

Grid NN Performance test with different threshold distances								
Points (ML)	0.078388	0.384031	0.950422	1.726824	2.694288	3.87379	5.221704	10.078696
1m	358.014745	325.357011	492.618744	530.282284	583.729533	1090.11216	1669.55506	4736.34962
2m	29.6016109	38.092612	62.6342981	75.315798	96.5981369	165.232066	240.676724	639.973188
3m	15.917522	15.7301331	36.236625	47.5516829	66.0178242	98.2573631	137.385016	340.023626
4m	18.7095702	19.5363369	25.8212271	36.3829648	52.3233449	78.4734261	113.794272	300.797425
5m	35.9907629	12.3301101	27.236551	38.773787	57.33006	81.5651331	114.764287	266.44215