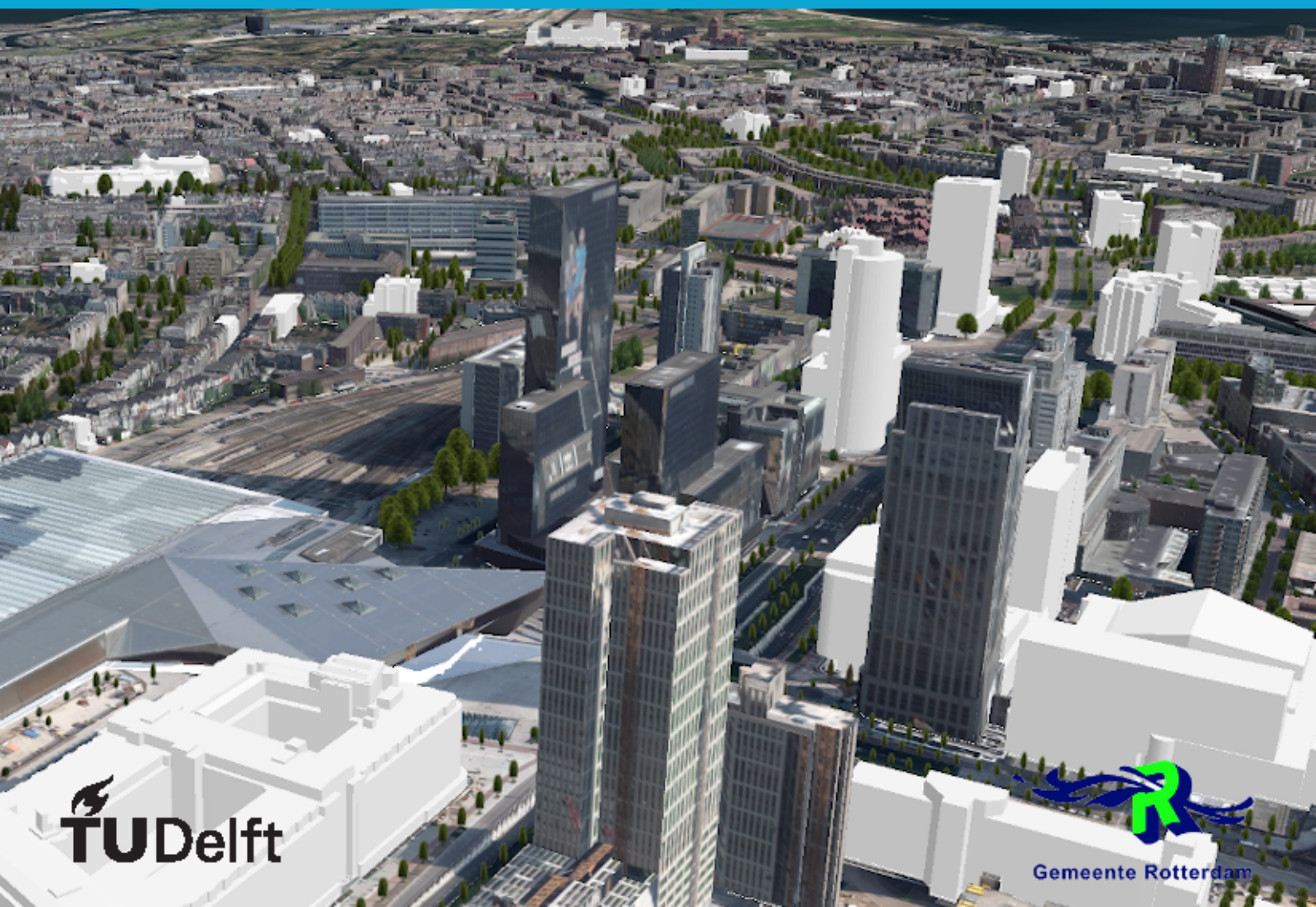
MSc thesis in Geomatics for the Built Environment

An integrative versioning workflow for 3D City Model maintenance

Konstantinos Mastorakis
2020



TUDelft

Gemeente Rotterdam

AN INTEGRATIVE WORKFLOW FOR 3D CITY MODELS VERSIONING

A thesis submitted to the Delft University of Technology in partial fulfillment
of the requirements for the degree of

Master of Science in Geomatics for the Built Environment

by

Konstantinos Mastorakis

November 2020

The work in this thesis was carried out in the:

3D geoinformation group
Department of Urbanism
Faculty of the Built Environment & Architecture
Delft University of Technology

GIS & Advies
Stadsbeheer
Gemeente Rotterdam

Supervisors:	Hugo Ledoux (TU Delft)
	Stelios Vitalis (TU Delft)
	Maarten Vermeij (Gemeente Rotterdam)
Co-reader:	Giorgio Agugiaro (TU Delft)

ABSTRACT

3D city models are continuously becoming more popular among practitioners due to the volume and versatility of information they contain, which makes them suitable to be used in various applications. However, there is no mechanism to allow maintaining them updated at the same pace that cities evolve, or when error correction is necessary, eventually diminishing their value.

Many cities around the world already possess such models which are mostly used for experimentation and research purposes. Such an example is also the city of Rotterdam, whose *3D city model* is not regularly updated and has to be outsourced for that purpose.

This thesis investigates into addressing this issue by proposing and implementing an integrative maintenance workflow. The workflow is designed to fulfill what the maintenance needs of a typical municipality are expected to be. Those needs were identified after conducting an analysis of the current situation and collecting information from practitioners within the municipality through interviews.

The workflow is a combination of *3D city model* versioning and visual editing capabilities with the aim to effectively maintain *CityJSON* encoded models in an intuitive way. Its implementation includes two prototype software implementations: a versioning component, which is utilized to create a workflow inspired by *git flow* and allows concurrent maintenance and alternative scenario testing in a non-linear and distributed way, and a visual editing component capable of editing *CityJSON* encoded *3D city models* by extending *Blender's* functionality.

Following the implementation, the workflow was tested by simulating real world maintenance scenarios. The tests demonstrate the feasibility of maintaining *3D city models* with such a workflow and more specifically the suitability of git based workflows. At the same time some key parameters of the versioning mechanism are identified which if tuned properly they can optimize the performance, behavior and robustness of *3D city model* versioning.

With both components being prototype solutions the workflow is far from operational and there is certainly a lot of space for improvement regarding both components. Utilizing the workflow in practice would be the ideal way for collecting useful feedback. Besides that, there are already extensions of *Blender* that combined with the visual updating component of the workflow can offer advanced integration of editing and analysis capabilities.

ACKNOWLEDGEMENTS

With this thesis I am completing my M.Sc. Geomatics degree which was two full years of commitment that has pushed me closer to what I find so exciting to call my profession. Even if it took longer than initially planned I deeply enjoyed investing all the effort together to compile this thesis; first because *3D city models* are amazing on their own and second because I always found great pleasure in creating something that proves useful to more than myself. And that is what I hope this thesis to be.

I feel the need to express my utmost gratitude to my mentors during this task, *Hugo Ledoux* and *Stelios Vitalis* for being always available when I needed them for giving me meaningful feedback and for guiding me through all of this yet allowing me to choose the path that I wanted. Many thanks also to *Giorgio Agugiaro* for providing me with crucial feedback that helped me do my thesis more complete.

I would like to deeply thank the whole municipality of Rotterdam which gave me the chance of carrying out my project with them, but more specifically my supervisor *Maarten* who made my stay within the premises of the municipality of Rotterdam feel like home, providing me with all I needed and trusting my ideas in the first place, *Leonard van den Welde* for helping with all technical issues and *Harmen Kampinga*, *Christian Wisse* and *Roland van der Heijden* for their time and willingness to work with me, providing me all the necessary information to identify the current situation with respect to the 3D city model and its management. The quality of my thesis would be much lower without that information. I really wish my stay within the premises was as long as originally planned but then corona virus came!

Lastly, I would like to thank my parents for all the psychological support and their belief in me and to what I am doing. In the times of Covid-19 it was more than necessary.

CONTENTS

ACRONYMS

3DCM 3D City Model . . . 1
ADE Application Domain Extension . . . 8
API Application Program Interface . . . 8
CJV CityJSON Versioning Prototype . . . 19
CLI Command Line Interface . . . 3
CRS Coordinate Reference System . . . 9
CVS Concurrent Versions System . . . 11
DSM Digital Surface Model . . . 10
ETL Extract Transform Load . . . 3
GIS Geographical Information System . . . 9
GML Geography Markup Language . . . 73
GUI Graphical User Interface . . . 31
JSON JavaScript Object Notation . . . 3
LoD Level of Detail . . . 7
OGC Open Geospatial Consortium . . . 7
OWM Oracle Workspace Management . . . 17
SCCS Source Code Control System . . . 11
SQL Structured Query Language . . . 16
SRDBMS Spatial Relational Database Management System . . . 9
VCS Versioning Control System . . . 2
XML Extensible Markup Language . . . 7

1 INTRODUCTION

1.1 MOTIVATION

3D City Models (3DCMs) are growing in popularity, especially for governmental parties and cities around the world. More and more domain experts interact with them, for various purposes ranging from visualization to infrastructure planning and energy demand estimation among others [Biljecki et al., 2015]. Their appeal lies to the amount, versatility and heterogeneity of information they contain, but more importantly to the way they can visualize this information i.e. in 3D.

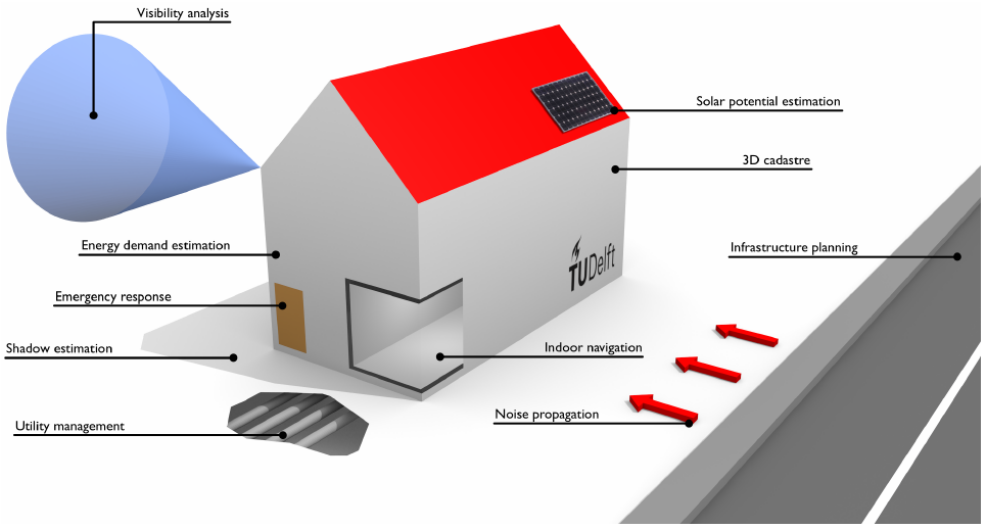


Figure 1.1: The multi-domain applications of 3D city models. Figure from Biljecki et al. [2015]

3DCMs are digital representations of actual cities that are able to store geometric (vertices), descriptive (attributes) and semantic information (categorization of objects faces). They are created from combining existing geo-information from various sources such as shapefiles, digital terrain models, satellite imagery, CAD-files, lidar data etc. [Agugiaro, 2016] [Malambo and Hahn, 2010].However, the plethora of data sources, leads to inconsistencies and big variation in the quality of 3DCMs datasets [Biljecki et al., 2016].

3DCMs creation is an expensive and time consuming process [Döllner et al., 2006]. That is due to the amount of information that has to be integrated into the model. Although it is a domain that automation will have a strong impact and some solid progress in automating the generation of 3DCMs is already done [Cao et al., 2017], manual editing is still necessary [Malambo and Hahn, 2010].

An important challenge for 3DCM is efficient and effective maintenance. The major reason for that is that cities change over time and their respective models need to be updated for these changes to be reflected in them. In addition, it facilitates

the correction of any errors that might happen upon creation [Biljecki et al., 2016]. There is no doubt that good maintenance will always lead to a more functional 3DCMs. In other words an obsolete model is certain to find less use than an up to date one.

Not taking advantage of the full potential of 3DCMs applications due to lack of (good) maintenance is equivalent to discarding information already obtained. Considering the versatility of 3DCMs and how entangled cities are with society, it is safe to say that limited maintenance translates into reduced societal impact. Furthermore, limitations in maintenance such as lack of managing tools by the organizations that possess the 3DCM, increases the administrating costs of a 3DCM. This often results into outsourcing —the biggest part of— its maintenance, by providing the necessary component-datasets to third parties. In some cases the solution for an updated 3DCM includes the (re)generation of completely new iterations of the model from scratch [Airaksinen et al., 2019], through labor intensive procedures and repetitive workflows, that a very small proportion of practitioners is able to perform.

There is a lot of research going on in the recent years regarding how 3DCMs should be maintained; much is at conceptual level and unfortunately little about the implementation level. Research has showcased the importance of keeping track of the history of maintenance [Samuel et al., 2016] [Chaturvedi et al., 2016], or to put it simpler, the chronological order of the changes in a 3DCM. For example, a building's lifecycle can be extracted by having the order of the changes performed on it along a timeline (see Figure 1.2). Or being able to recreate the city's state at any given moment by following the order of the changes reversely (see Figure 1.3). What is more, 3DCM maintenance does not necessarily has to happen by a single user at a time. It is possible for 3DCM to be maintained concurrently and collaboratively by the adapting what in software development is known as Versioning Control System (VCS) [Prieto et al., 2017]. At the moment, *Git* [Chacon and Straub, 2019] looks to be the VCS that inspires most researchers to create 3DCM-oriented VCSs.

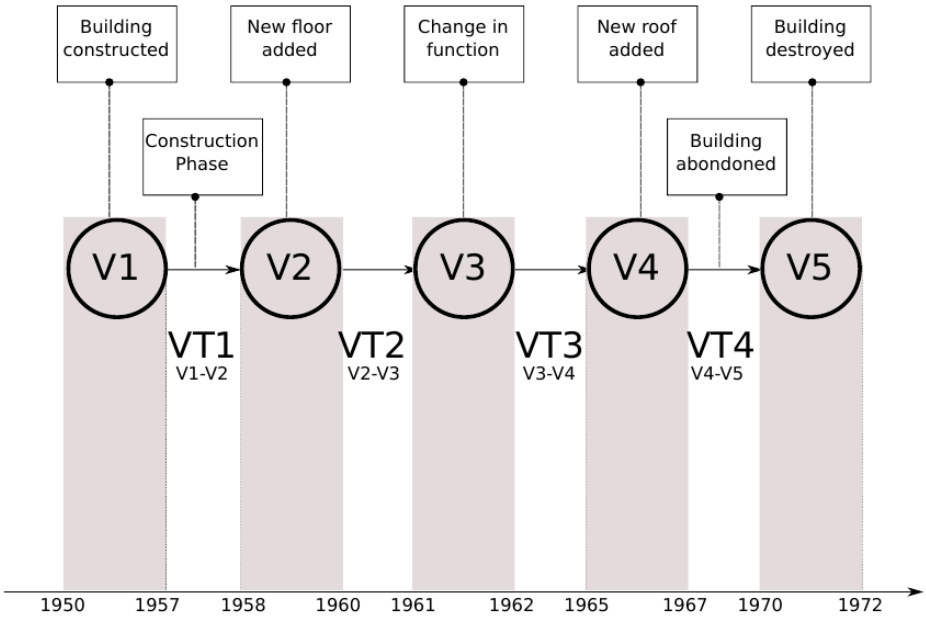


Figure 1.2: A building's lifecycle and how it is related to the order of the changes that were performed. Figure from Samuel et al. [2016]

However, the maintenance challenge in the 3DCM domain still persists on a practical level. Most research is around the standardized *CityGML* data model, whose practical shortcomings when implemented into a data exchange format lead to limited

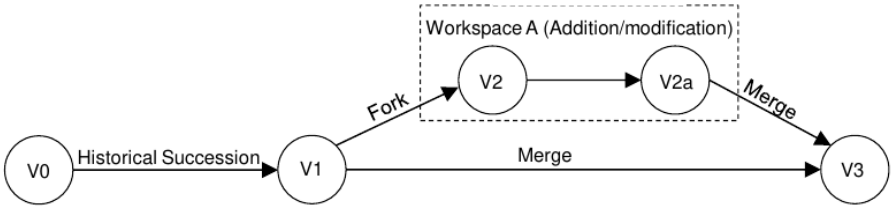


Figure 1.3: A representation of a city's evolution over time. Figure from Chaturvedi et al. [2016].

functionality [Ledoux et al., 2019].

Inspired by VCSs for software development, researchers have tried to implement workflows and data structures to enable versioning or navigation among different versions of 3DCMs [Prieto et al., 2017] [Samuel et al., 2020], some of them building directly over, or conceptually close to popular software oriented VCSs.

Although there is a certain focus on the VCS mechanisms themselves which are responsible for managing the new versions [Prieto et al., 2017] [Samuel et al., 2018] [Vitalis et al., 2019], there is no focus into improving the way new versions are generated. Most common ways of generating new versions is by directly editing a 3DCM file, updating databases where 3DCMs are stored in [Yao et al., 2018], or by using Extract Transform Load (ETL) software like *FME*.

Direct editing of a 3DCM file is rather exhaustive and error prone, while *FME* is for batch oriented workflows, which do not facilitate single objects maintenance. Importing and exporting *CityGML* files into a database is a quite complex task that few practitioners can perform.

In addition the data structure of *CityGML v.2.0* encoding itself does not include any versioning components, which is not the case with *CityGML v.3.0*. A presentation of the *CityGML v.2.0* and *v.3.0* data structures is given in Chapter 2, while further elaboration can be found in Chapter 6.

On the other hand, *CAD* software or other non 3DCM oriented software like *BIM* is a bad option for editing 3DCMs as well. That is due to the discrepancy between data models, which inevitably will result in compatibility issues and information loss [Noardo et al., 2019].

This thesis, focuses on a promising data structure for 3DCM versioning presented by Vitalis et al. [2019], that wraps around the *CityJSON* data model [Ledoux et al., 2019]. *CityJSON* is a JavaScript Object Notation (JSON)-based format, which was designed to favor software development around 3DCMs. Their work includes an implementation of the versioning data structure in a prototype Command Line Interface (CLI) software.

The aim of this thesis is to investigate how the maintaining needs of a 3DCM can be addressed by combining the implementation of Vitalis et al. [2019] data structure with a software interface that was implemented specifically for this reason and allows visual editing of *CityJSON*-encoded 3DCM. To do so a practical workflow is created and tested with scenarios that simulate the current needs of a typical 3DCM of a municipality, with respect to updating its model.

Establishing a workflow that combines visual updating and versioning of 3DCMs, will bring even more value to them by facilitating their use. Among the many ben-

efits of having an up- to-date 3DCM, the capability of extended visual interaction will increase the adoption rate of 3DCMs across different domain experts. It makes the user's experience more seamless regardless their background, by focusing the user's attention into what the 3DCM represents rather than splitting their attention between data structures as well.

1.2 USE CASE – GEMEENTE ROTTERDAM

The use case for this project is the 3DCM of *Gemeente Rotterdam*. The 3DCM was first initiated in 2010 but due to geometric defects was rebuilt from scratch in 2016. Since then, it has been updated in 2018, by integrating around 4000 buildings changes since 2016 and is planned to be updated in 2020 as well. Currently, a new iteration of the 3DCM is created in a biennial lifecycle incorporating all the changes that happened during that time. In the meantime the 3DCM remains outdated.

The basic tasks for keeping the model up to date include -among others- modeling new buildings that are built, removing demolished ones, editing the buildings that are altered 1.e. new building part added etc and editing their attributes. With the current approach, keeping the 3DCM updated is not a continuous process. It is actually a big one-off venture that requires planning in advance, takes time and is financially expensive. It requires outsourcing the necessary datasets to third parties which create the new iteration of the model.

Although the key registration dataset, namely *BAG*, which is a component of the 3DCM, is regularly updated and maintained by the municipality without any issues, the same does not happen with the 3DCM which is outsourced accompanied by elevation LIDAR data. The main reason for that is the lack of both software and established workflows for maintaining 3DCMs.

To understand the current situation in depth and identify the municipality's needs, some interviews with practitioners working with the 3DCM were made. They were focused on the technical aspects of the model itself, the management processes and its future potential. During the interviews, information about the status of the 3DCM, as well as the current processes of updating it were gathered.

Obtaining an extended view of the current processes and workflows around *Gemeente Rotterdam*'s 3DCM, helped in establishing a solid background for the proposed workflow to be built upon. Despite the fact that this project was carried out working with *Gemeente Rotterdam*'s 3DCM, the suggested workflow is highly adaptable thus case-independent, since the maintenance problem of 3DCM is universal. For the experimentation needs of this thesis, the municipality has kindly provided part of their 3DCM. The use case is thoroughly presented in Chapter 3.

1.3 RESEARCH QUESTIONS

Understanding the current needs a typical municipality has in order to maintain its 3DCM and motivated by the lack of regular maintenance similar to that of *BAG* and other key registration datasets the following research question is defined:

- To what extent can a *Git*-based versioning approach be used for the maintenance of the 3DCM of a typical municipality?

  followed by two sub-questions:

-What would be a conceptual workflow that would make this approach practical and manageable?

-How can the maintenance process be improved by combining the versioning workflow with 3D visual editing capabilities of the model?

1.4 THESIS SCOPE

The scientific and technical scope of this thesis is:

- The objects of interest for this project are buildings; not utilities, trees, city furniture etc.
- The implementation is strictly limited to cope with *CityJSON* files converted from the original *CityGML* files.
- The visualization and editing platform that is extended to import *CityJSON* 3DCM is Blender, and all tests with visual editing of the model are made in it.
- Testing will be local and no concurrent editing from many users will take place.
- Although for the 3DCM datasets used building textures are available they are out of the scope of this thesis.
- When referring to *3DCityDB* implementation —see Chapter 2— only the *PostgreSQL-PostGIS* variation is within the scope of this thesis. The alternative *Oracle* variation is out of the scope since it is a proprietary database system.

1.5 THESIS OVERVIEW

Chapter 2 presents an extended overview of related work to this thesis explaining the 3DCMs's data models and encodings. It then sets the basis on which the workflow is built upon and gives a short introduction of the workflow components.

Chapter 3 introduces the specifications of the use case in detail, and identifies the current maintenance needs of the model.

Chapter 4 establishes the conceptual layout of the workflow with all the interconnections between its components based on the maintenance needs of the 3DCM

Chapter 5 explains the implementation specific decisions of the workflow that were made and its limitations.

Chapter 6 includes all the tests that were carried out to evaluate the performance of the workflow's implementation based on the previous analysis.

Chapter 7 discusses on the findings of this thesis, elaborates based on the results of Chapter 6 and provides a realistic view of how will Rotterdam be able to benefit from the workflow.

Chapter 8 suggests what can further be done to improve the workflow, with respect to both versioning and visual editing component.

2 BACKGROUND AND RELATED WORK

2.1 3D CITY DATA MODELS AND ENCODINGS

3DCMs are digital representations of the urban environment. They are single datasets that incorporate terrain surfaces, buildings, city furniture, trees, infrastructure etc. in three-dimensions [Döllner et al., 2007]. They are encoded in various ways with or without semantic specifications of their geometries.

Under the scope of this thesis only 3DCMs that have the ability to store building surface semantics are included. So only those encodings and data models will be introduced. There are two file based and one database-based solutions: *CityGML* which is an Extensible Markup Language (XML)-based encoding, *CityJSON*, a JSON-based encoding and *3DCityDB* which extends the *PostgrSQL* relational databases to cope -primarily- with *CityGML* files.

Both file-based encodings are free, human and machine readable. There is citygml-tools which allows bi-directional conversion between the two. *3DCityDB* is also free and uses a relational database schema for mapping *CityGML* entities into tables. These solutions are presented in the following sections.

2.1.1 CityGML

CityGML is an open data model and an XML-file-based format storage for the storage and exchange of 3DCMs [Gröger et al., 2012]. It defines distinct Level of Detail (LoD)s that allow the 3D geometries to be stored in more than one representations for different applications. It is broadly used among practitioners, academia and organizations at the moment. It is adopted as an international standard, designed and maintained by the Open Geospatial Consortium (OGC).

Although a lot of work has been done with *CityGML* as a data model, its performance as an exchange format is poor [Ledoux et al., 2019]. Being an XML based encoding makes *CityGML* files quite impractical. XML language is inherently verbose as every other markup language, since every element it contains has to be inside opening and closing tags. Also the data structure of *CityGML* files contributes to redundancy in information. *CityGML* files store the coordinates of every vertex of every object inside each object replicating the same coordinates every time the same point belongs to more than one faces. This also leads to unnecessary increase of *CityGML* files' size considering the amount of objects it contains, poor editing efficiency and topology support.

What is more *CityGML* encoding is not strict about how semantics is stored leading to adhok solutions when it comes to *CityGML* file parsing, reducing the interoperability of these parsers. Finally, *CityGML* files have deep hierarchical structure, which combined with the previous fact makes software development for this encoding extremely exhausting and not robust.

All in all, *CityGML* files are extremely bulky and difficult to be parsed into a pro-

gramming environment or browsers, resulting in poor web-compatibility, exchange practicality and software support.

2.1.2 The *Scenario* Application Domain Extension

Scenario Application Domain Extension (ADE) [Schüler et al., 2018] is an extension of the *CityGML v.2.0* data model that enables testing of virtual changes (scenarios) on *CityGML v.2.0* encoded 3DCMs. These virtual changes can refer to the whole 3DCM or any subset of it and can affect both attributes and geometry. The virtually changed 3DCM can then be used for optimization applications, simulations and more. In other words, *Scenario* ADE data model is designed to be application independent. *Scenario* ADE is implemented as an extension of the *3DCityDB* schema for *PostGIS*.

It consists of two main parts: *Time series module* and the *Core module*. The former is responsible for documenting all inputs and results that have a temporal aspect accompanied by other metadata such as data acquisition method, data source, timestamp etc. The latter models the set of physical objects changed for the current scenario and can be derived by a previous scenario. Optionally, some parameters that describe initial conditions or output parameters of the scenario can be saved. There is also the capability to save the related resources (financial, time, energy etc). Last but not least, it is possible to record all the changes that happen to the (virtually) changed city objects at both geometry and attribute level.

Considering that *Scenario* ADE has a temporal component transfuses a form of versioning functionality to it; however that is not its main focus. Its main concept, i.e. testing new ideas and scenarios has influenced the design of the workflow presented in this thesis. Thus, the concept of (virtual) scenarios testing is part of the whole workflow conceptualization as shown in Chapter 4.

2.1.3 CityJSON

CityJSON is a JSON-based encoding of a subset of the *CityGML* data model. Developed and maintained by *3D geoinfo lab* of *TU Delft*. It was designed to be a compact 3DCM file exchange format friendly to programmers.

It is 6x more compact on average, than *CityGML* and incorporates a flattened-out hierarchy [Ledoux et al., 2019]. Being JSON-based means it can be parsed natively into most modern programming languages and browsers. This also means that applications that offer Application Program Interfaces (APIs) can be easily extended to support *CityJSON* files.

A key feature of the *CityJSON* data structure, is that it keeps all the vertices stored once outside the objects, in a separate list. Each object's face contains pointers that point to the index of the vertex in the global vertices list, which contains the coordinates. This not only adds tremendously to the editing capabilities of such a file, but also simplifies topology creation, and reduces the file size by not storing redundant information.

Notice in Figure 2.1 that for the *CityGML* file for every attribute-value pair that both are enclosed into XML tags. The value itself is nested into the attribute, creating a very complex file hierarchy. In contrast the *CityJSON* file is more flat and avoids using opening and closing tags by using a key-value pairs, the same structure that in *Python* is called a dictionary.

```
<cityObjectMember>
  <bldg:Building gml:id="ID_0599100000601416">
    <creationDate>2010-03-08</creationDate>
    <gen:stringAttribute name="gebouwnummer">
      <gen:value>0599100000601416</gen:value>
    </gen:stringAttribute>
    <gen:intAttribute name="laagste_bouwlaag">
      <gen:value>0</gen:value>
    </gen:intAttribute>
    <gen:intAttribute name="hoogste_bouwlaag">
      <gen:value>3</gen:value>
    </gen:intAttribute>
    <gen:intAttribute name="aantalBouwlagen">
      <gen:value>4</gen:value>
    </gen:intAttribute>
    <gen:stringAttribute name="statusOmschr">
      <gen:value>Pand in gebruik</gen:value>
    </gen:stringAttribute>
    <gen:stringAttribute name="typeOmschr">
      <gen:value>tussenpand</gen:value>
    </gen:stringAttribute>
    <bldg:yearOfConstruction>1912</bldg:
    yearOfConstruction>
```


(a) A snapshot of a 3DCM file encoded in *CityGML*
Source: Municipality of Rotterdam (originally in CityGML)

```
"CityObjects": {
  "ID_0599100000601416": {
    "address": {
      "CountryName": "Nederland",
      "LocalityName": "Rotterdam",
      "ThoroughfareNumber": "81",
      "ThoroughfareName": "2e Schansstraat",
      "PostalCode": "3025XM",
      "location": {
        "type": "MultiPoint",
        "boundaries": [
          65580
        ],
        "lod": 0
      }
    },
    "type": "Building",
    "attributes": {
      "yearOfConstruction": 1912,
      "creationDate": "2010-03-08",
      "gebouwnummer": "0599100000601416",
      "hoogste_bouwlaag": 3,
      "statusOmschr": "Pand in gebruik",
      "aantalBouwlagen": 4,
      "laagste_bouwlaag": 0,
      "typeOmschr": "tussenpand"
```


(b) A snapshot of a 3DCM file encoded in *CityJSON*
Source: Municipality of Rotterdam (converted from CityGML)

Figure 2.1: A side by side comparison between a 3DCM file encoded in *CityGML* and *CityJSON*

2.1.4 3DCityDB Database

3DcityDB is a free geo-database solution, designed for efficient querying, visualization and updating 3DCMs [Yao et al., 2018]. It is a relational database that maps the data model of *CityGML* onto a relational database schema, which lies at its core. It supports two Spatial Relational Database Management System (SRDBMS) *Oracle Spatial/Locator* —which is out of scope of this thesis— and *PostgreSQL* with *PostGIS* extension.

Its functionality includes importing and exporting from and to various common formats such as *CityGML, KML, Collada, glTF*, as well as a 3D web client for visualization and enhanced exploration of semantically extended 3DCMs. It has built-in procedures organized in six packages that allow Coordinate Reference System (CRS) transformation, statistics overview, object deletion, maximum volume calculation etc. [Yao et al., 2018]. Lastly, it makes use of the already existing functionality of the SRDBMS, such as estimating the bounding box of an object or the whole model. It is arguably a superior solution for managing 3DCMs compared to dealing directly with raw *CityGML* files.

Currently, the only versioning capabilities bestowed to *3DCityDB* by the *PostgreSQL-PostGIS* combination is the functionality of *pgVersion* module (see Section 2.4.3), an extension of the *PostGIS* schema that enables versioning for 2D Geographical Information System (GIS) data.

However, the data structures of 2D GIS data are far simpler than that of 3DCMs. Most GIS data are simple encodings of 2D objects represented by a single geometry and its attributes, stored in relatively flat structures. On the contrary, 3DCMs data structures contain various entities in a nested structure with objects often having multiple geometries with their respective attributes, plus there is also semantic information for some of these geometries surfaces. Thus, 3DCMs can not be handled by the database under no circumstances.

On a data model level there is also no support for 3DCM versioning since *3DCityDB*

currently supports *CityGML v.1.0* and *v.2.0*, which have no versioning elements within their respective data models.

2.1.5 Choosing a suitable data model and encoding

Considering the goal of this thesis is to design and implement a versioning workflow that includes visual editing capabilities of the 3DCM choosing the most appropriate data model and encoding is crucial. The data models with their respective implementations mentioned above are the currently available options for working with 3DCMs with semantics-storing capabilities. This section explains which option was selected and why.

CityGML is notoriously verbose and not programming friendly. There is also no support for versioning either at the data model or at implementation level in the current *version 2.0*. For these reasons it was disqualified as the encoding for the workflow. However, the upcoming *CityGML v.3.0* data model incorporates versioning but it will be presented in Section 2.5.

CityJSON on the other hand was conceived and implemented to solve the technical problems the *CityGML* encoding poses. Using the JSON format is what attracts developers to create software around it. There is already plenty of different software for *CityJSON* [1] and further software development around it is appealing since JSON format is widely supported in the programming world. There is also a data structure —including an implementation— to incorporate versioning into *CityJSON* files, which is presented in Section 2.5.2. These were the key reasons for choosing *CityJSON* as the most appropriate solution for the workflow.

3DcityDB was considered not a practical alternative for the nature of the thesis, since the file-based approach offers more practicality for inspecting and tweaking the data and prototyping in general. In addition, *3DcityDB* does not offer hierarchical versioning (branches); only linear versioning due to the functionality of *PostgreSQL* which by no means can handle the *CityGML* data models alone. This results in limited capabilities of extracting previous versions of the 3DCM with potential loss of information. Lastly, *3DCityDB* has no visual editing capabilities —especially with geometries— which is important for the implementation of the proposed by this thesis workflow.

2.2 3D CITY MODELS MAINTENANCE

Because 3DCMs contain a vast amount of information they need to be maintained to preserve or even increase their value over time. Maintenance in the 3DCM domain though has been ambiguously defined due to the novelty of the domain itself. It is quite common that 3DCM researchers use the term ”maintenance” in different contexts, whose meaning is adapted to the focus and the scope of their research.

In their research [Steinhage et al., 2010] attempt to automate 3DCM generation by automating 3D reconstruction of buildings with fusion of datasets such as parameterized *CAD* models, building footprints originated from GIS and Digital Surface Model (DSM) data. These datasets allow for the 3D reconstruction using various different approaches and techniques.

In the context of their research, maintenance is about managing across versions of the model created with different 3D reconstruction methods and not maintaining

1 https://www.cityjson.org/software/

a single 3DCM dataset with respect to its evolution through time. A typical result of different 3D reconstruction methods is different LoD of the resulting geometries. In Figure 2.2 an example of two alternative 3D reconstruction methods which leads to to two different LoD geometries.

Both *CityGML* and *CityJSON* data models support multi-LoD geometries within a single object; so versioning as suggested in this case could be better addressed by a versioning mechanism that wraps around any of the two data models and versions a single 3DCM which contains more than one LoDs, rather than many 3DCMs that contain a single (different) LoD.

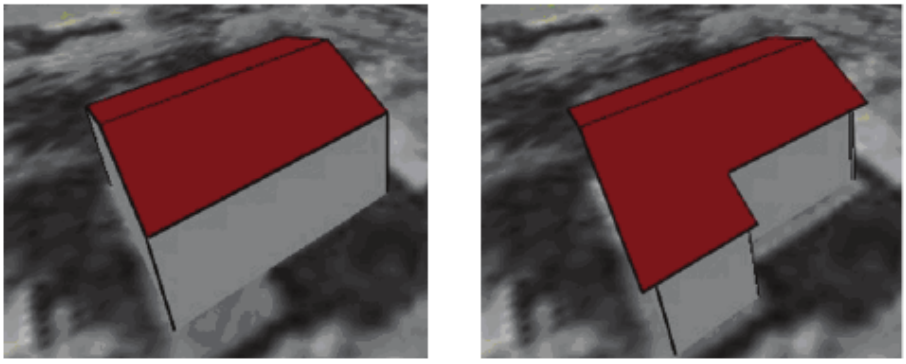


Figure 2.2: An example of two alternative model versions that were reconstructed using different 3D reconstruction approaches. Figure from Steinhage et al. [2010].

In contrast, maintenance in the context of research of Prieto et al. [2017] refers to a continuous deployment platform that wraps around *CityGML*-encoded 3DCMs. Tools used in software development are utilized, such as VCS and unit testing components and adapted to fit 3DCM maintenance needs. As explicitly mentioned in their paper, deployment means to 'make the model available to its potential users'. In this context maintenance is about creating workflows that facilitate incorporating changes to the model in an easy, fast and less error prone way, thus increasing the usability of the model across different domain experts (see Figure 2.3). The platform allows concurrent maintenance by multiple users and includes both manual (i.e. editing objects) and automated (i.e. schema validation) maintaining workflows.

2.3 VERSIONING CONTROL SYSTEMS IN SOFTWARE DEVELOPMENT

Versioning in software development has been tackled decisively through years since VCSs have been developed at least since 1975 [Rochkind, 1975], with Source Code Control System (SCCS) being one of the first [Glasser, 1978]. They are primarily used to allow developers to roll back into previous versions/instances of source code, by keeping track of every change that is made. In the following years, VCSs were extended to enable concurrent work on the same project with **?** introducing Concurrent Versions System (CVS).

According to Ball et al. [1997], alongside every change VCSs capture a lot of metadata like timestamp, author, comments etc, which if properly interpreted provides very useful information on the evolution of the software itself.

A key characteristic of VCSs is the architecture they follow: Distributed vs Central-

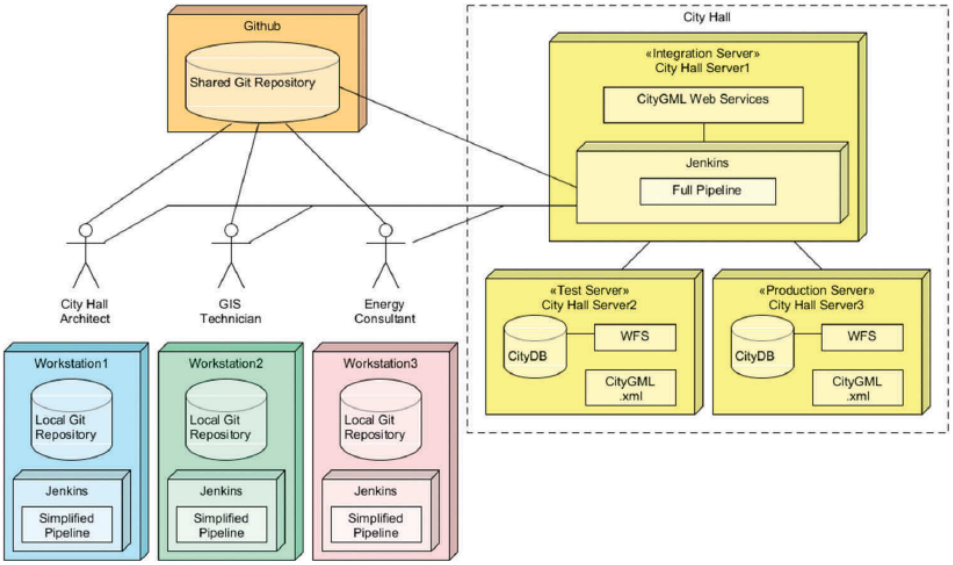


Figure 2.3: An example scenario of the components of the maintenance platform and their interaction. Figure from Prieto et al. [2017].

ized. Simply put, with a distributed architecture peers work on a local repository and commit their work / fetch data to/from an online repository in which their collaborators have access as well Figure 2.5. On the contrary, in a centralized architecture all peers work directly on the online repository Figure 2.6.

They both come with respective advantages and disadvantages. Based on their empirical study on the comparison of distributed vs centralized systems, Brindescu et al. [2014] conclude that in the former, the size of commits is smaller, split commits are more often and commits include more issue tracking labels compared to the latter. Distributed systems' ability of committing locally is the a 'killer feature' -according to the study- for developers. On the other hand the flatter learning curve of centralized is their strong point.

A very useful feature of VCSs is branching (see Figure 2.4). Branching allows to create a new copy of the project at any given moment and work on it, leaving the main copy of the project unaffected. All the changes that happen to the new copy are also tracked exactly as if they were made on the main. This feature is extremely useful in software development for trying new prototype ideas, feature extension, debugging etc. If required the new copy of the project can be merged back to the main integrating all changes to it and updating the main copy. It also allows for concurrent working on the project, without affecting each other workflow. In case the same part of the file is differently modified by more than one users, the system will prompt a conflict message upon a merge attempt, allowing for manual editing of the file at that point to resolve the conflict.

2.3.1 Git

Git is among the most popular modern VCSs [Spinellis, 2012]. Developed initially by *Linus Torvalds* for maintaining the *Linux* kernel in 2005, it has grown a lot in popularity since. *Git* was designed such that the majority of operations are executed in the local repository, allowing it to be very fast and responsive, very secure in terms of data integrity and with the ability for offline work to be done. Its users range from hobbyist software developers up to the world's biggest IT corporations

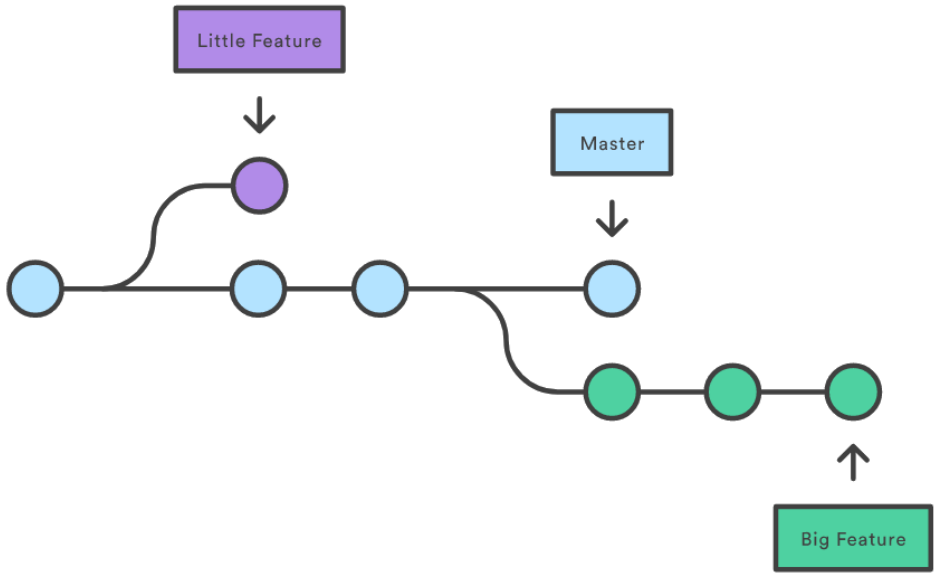


Figure 2.4: The concept of branches.
Figure from *https://www.atlassian.com/git/tutorials/using-branches*

such as *Microsoft, Apple, Amazon* etc. Some other popular VCSs implementations are *Mercurial, Subversion, Perforce, BitKeeper, Monotone*.

Git is a distributed VCS and has an explicit conceptual difference compared with other VCSs. While the rest of the VCSs store the differences -also known as delta-tables' between consecutive versions, *Git* stores the so called 'snapshots' for every version [Chacon and Straub, 2019]. It does so by taking a snapshot of the whole system (i.e. how every file in the repository looks like) every time a new commit is made and stores a reference to it. At every commit the content of the files for the whole system is hashed -hashes are used in cryptography for security purposes but in *Git* only for data integrity purposes- to verify the content of every file after any change, which makes it almost impossible for the data to go corrupt. When needed, it will calculate the differences from 'subtracting' consecutive snapshot references and computing what has been changed. To optimize storing efficiency, in case any files haven't been changed between consecutive commits, it stores a link to the previous identical file in the previous snapshot. This way storing duplicate files is avoided.

This conceptual design decision gives *Git* one of its most powerful characteristics compared to other VCSs. This is how it can deals with branching. To give an idea of how this simplifies branching, consider that having snapshots stored at any given moment reduces the task of creating a new branch in simply creating a reference to an existing snapshot.

Git-based workflows

The capabilities of *Git* combined with the fact that it is free and open-source led many enthusiasts and organizations into creating customized *Git* workflows to address their needs. Some of the most popular are the *Git flow* [2], the *GitHub workflow* [3] and the *GitLab workflow* [4] .

2 https://nvie.com/posts/a-successful-git-branching-model/
3 https://guides.github.com/introduction/flow/index.html
4 https://docs.gitlab.com/ee/topics/gitlab_flow.html

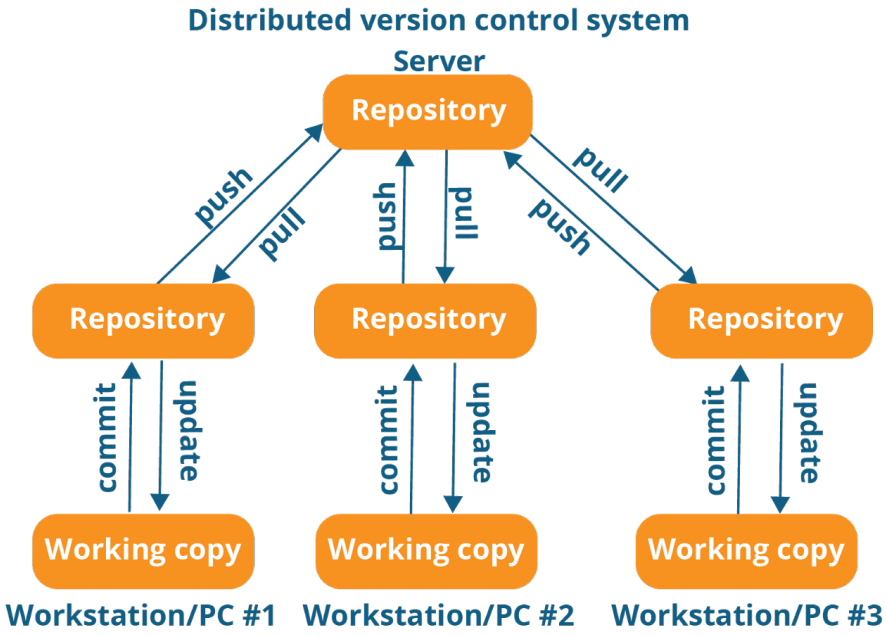


Figure 2.5: The structure of a distributed VCS.
Figure from
https://medium.com/faun/centralized-vs-distributed-version-control-systems-a135091299fo

The first has proven to be one of the most used *git* workflows adopted since its creation in 2010, while the *GitHub workflow* is a simplified alternative of the former. Finally, according to its designers the *GitLab workflow* is simplifying the task of issue tracking as development moves forward.

2.4 VERSIONING SOLUTIONS FOR GIS

The principles of software development VCSs can find great application if ported in spatial-oriented VCSs. That being said, there is a major difference between the two that has to be pointed out, in order to better understand what a spatial-oriented VCSs is about.

The difference is that inevitably, a VCS wraps around the data format it attempts to version and keep track of. In software development all the source code files are basically text files regardless the programming language. On the contrary, spatial data -both 2D and 3D- come in a multitude of formats. Thus, for different formats a VCS should have radically different implementations. This means that developing a spatial oriented VCS requires committing to single data exchange format. The chosen format deeply affects the whole developing process robustness and efficiency of the whole system. This section gives an overview of some VCS implementations for 2D GIS applications, both commercial and free.

2.4.1 GeoGig

GeoGig is an open source versioning tool that implements *Git's* principles to manage the versioning of geospatial vector data. It is a CLI distributed VCS which can be also accessed as a datastore in *Geoserver* [Franceschi et al., 2019].

It currently supports Shapefiles, PostGIS and SpatiaLite data, which are imported into a Git-like repository where all changes are tracked. It offers two storage back-

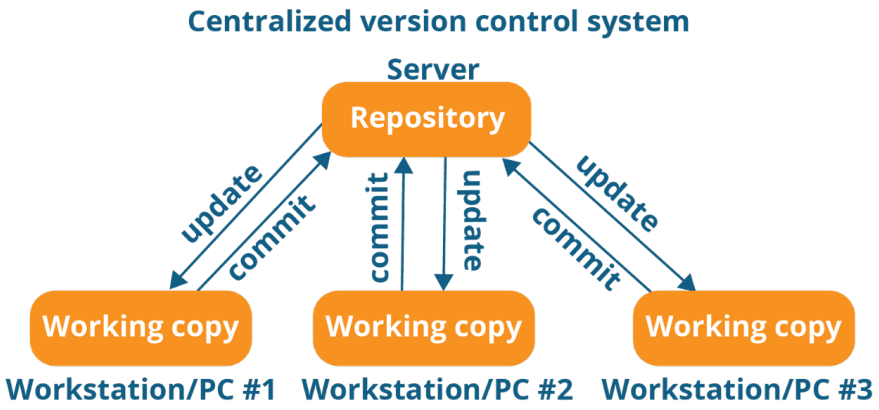


Figure 2.6: The structure of a centralized VCS.
Figure from
https://medium.com/faun/centralized-vs-distributed-version-control-systems-a135091299fo

ends, either the local filesystem or *Postgres*.

Its workflow is comprised by three steps that correspond to the three areas: *Working tree* where data is imported to be worked on, *Staging area* which is a preparation area for the data before being committed to the database and *Database* where all data is stored Figure 2.7. Upon importing, *GeoGig* internally converts it to its own binary format that can handle and keep track of.

Similar to *Git* it allows branching and merging of branches. It also takes snapshots of the data at any given moment storing all objects that form a version. For unmodified objects, pointers are utilized between versions so the same (unchanged) object is only stored once.

Geogig is a promising well documented VCS implementation, considering it is still unstable. It is considered a conceptually similar implementation to what this thesis suggests; yet oriented for fundamentally different data type.

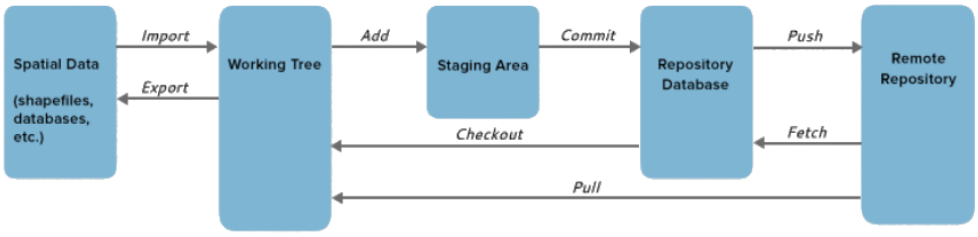


Figure 2.7: *Geogig's* three-step local workflow (import, add, commit extended with the remote repository transactions.
Figure from *Geogig's* official website

2.4.2 QGIS versioning plugin

The QGIS versioning plugin, is another approach to achieve geospatial data versioning. As its name suggests, it extends *QGIS* functionality to allow versioning of its layers. It is a distributed VCS which uses a *PostGIS* schema for versioning the database.

Developed to provide offline working capabilities and storing data in a *PostGIS* database, its approach is as follows: There are three operations for every table *insert*, *delete* and *update*. In reality though, nothing gets deleted from the tables. It is simply marked as deleted. The *update* is a sum of *insert* and *delete* operations. Those deleted and newly inserted rows are related with a parent-child relation.

For editing the data and creating new versions the plugin offers two-way functionality. *SpatiaLite* working copies for offline work, where the currently existing (not deleted) elements are exported for the user to edit. *PostGIS* working copies when there is active connection with the original database, where differences from the wanted version is stored Figure 2.8.

The database can be updated by editing the working copies and then committing back to the database as long as the working copy is up to date with it. In case of conflict the two conflicted versions are stored in a table and tagged as 'theirs' (database side) and 'mine' (working copy). The conflict is resolved by deleting one of them.

Branches are created that stem from any revision, by adding extra history columns at the version's record and and setting up a field for all features in this version to indicate the start of the branch.

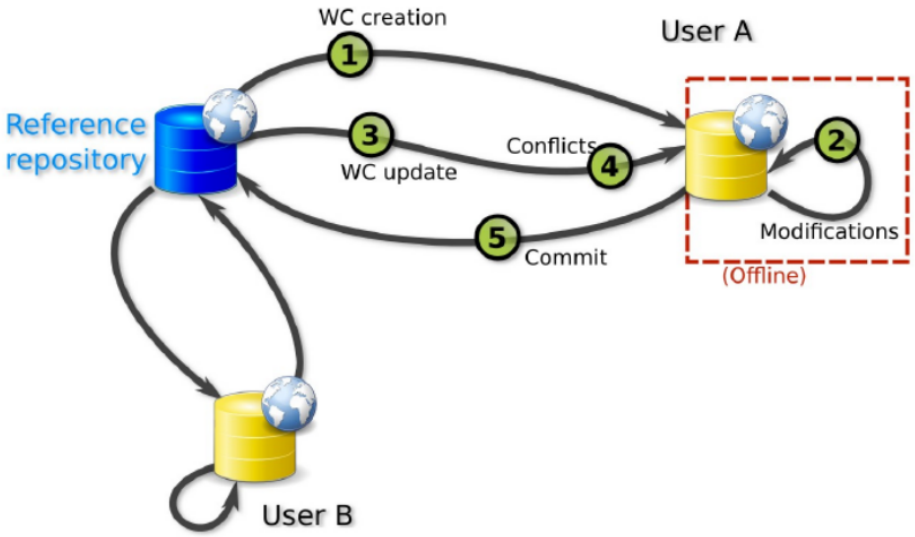


Figure 2.8: The schematic representation of the *QGIS versioning plugin*
Figure from *QGIS versioning plugin* official website

2.4.3 pgVersion

pgVersion was designed to version *PostGIS* layers concurrently in a multi-user environment. To do so it creates a new schema in a *PostGIS*-extended database, capable of managing the versioned tables.

Unlike *Geogig* and *QGIS versioning plugin* *pgVersion* is a centralized VCS, so it does not have the functionality for offline working. An uninterrupted connection is necessary between the client and the database server. The user can edit the existing and commit the new versions via direct Structured Query Language (SQL) commands or by using the pgvs plugin.

To version the layers through *QGIS's* interface, the layer should be prepared for versioning, conceptually similar to *Geogig's* importing operation. Then the versioned layer should be loaded from the database after removing the original from *QGIS*. After finishing the edits the new version can be committed back to the database as long as no conflicts arise. In case of conflict, the conflicting object(s) are shown with the author of the edit and the user can choose which one gets committed.

The fact that it is centralized makes *pgVersion* the most simplistic approach compared to the two previous ones. Although being simple means it is more robust for what it can do, it also means reduced functionality compared to the other two.

2.4.4 Oracle Workspace Management and ESRI ArcSDE

With Oracle Workspace Management (OWM) users work in workspaces which can be seen as individual rooms where users work, with all the changes being visible only to these users. The changes of the workspace are applied to the parent workspace via a merge transaction which makes the changes visible to the parent workspace as well [Oracle, 2020]. To be noted that in the case of Oracle products, this approach does not apply only to geospatial data but in every kind of data as long as the database schema can support it.

ESRI's equivalent for workspace is called version and is by default spatial-database oriented.The approach in this case is that every database has a default version which can not be deleted and is owned by the administrator. The different versions are stored in the geodatabase and regardless how many of them exist, each dataset is only stored once (delta tables technique) [Law, 2010].

Apart from being 2D-oriented, both systems are commercial and proprietary, meaning that they can not be freely used or further developed by anyone except their owners. They are also generic versioning solutions that are not easy to be adapted to wrap around complex data structures such as *CityGML* data model.

2.4.5 Reflections on *Geogig, QGIS versioning plugin, pgVersion*

In a 2D GIS environment these solutions offer solid overall functionality. 3DCMs data though is not 2D vector or raster data structure-wise and this means these implementations can not be used as part of a workflow implementation for versioning 3DCMs. They could be potentially tweaked and adapted to handle 3DCMs, but that would probably be more time consuming and exhausting than implementing a 3DCM oriented VCS in the first place.

VCSs have specific data models and encodings at their core. The variety of data formats that spatial information comes in, is the most limiting factor for the interoperability of implemented VCSs. Versioning of 2D data alone is challenging and it has to wrap around the respective data structures to achieve it. The mere complexity of the structure of 3DCM data models which allows them to contain so much and versatile kind of information (multi-LoD, attributes, semantics) is what vastly differentiates them from traditional 2D GIS oriented data. It is also what makes it impossible for a 2D GIS oriented VCS to be tweaked and adapted to handle them. It will probably be more exhausting than developing a new VCS specifically for that purpose.

2.5 VERSIONING SOLUTIONS FOR 3D CITY MODELS

2.5.1 *CityGML* versioning extension as part of the *CityGML v.3.0* data model

A new iteration of the *CityGML -CityGML v3.0-* has started as a project since 2013, some months after *CityGML 2.0* was adopted from the OGC [Kutzner et al., 2020]. At the time of writing this thesis, *CityGML 3.0* is not officially released yet but its aspects are almost finalized and nearly published.

According to its authors *CityGML 3.0* data model is revised and improved to meet the interoperability needs with other industry standards.

A big addition in the new iteration relevant to this thesis is two new modules that have been added for versioning support, namely *Versioning* and *Dynamizer*. These two module stem from the previous work of Chaturvedi et al. [2016], whose aim was to lay the conceptual basis for adding versioning functionality to the *CityGML* data model.

Versioning module is responsible for versioning qualitative changes such as the city's evolution over time or the lifespan of objects and managing the various versions of the city model.

Dynamizer is quantitative oriented responsible for capturing thematic values variation, changes in spatial properties and real time sensor feed.

Although versioning capabilities will be available now, which is a solid improvement of the data model, adding two different modules instead of one adds unnecessary complexity to the already complex *CityGML* data model. It might not look like a problem in the data model, but it will probably complicate the implementation of versioning in practice, even if the encoding is not XML-based. Keeping track of sensors values (i.e. what the *Dynamizer module* is responsible for) does not necessarily require a custom versioning solution adapted to 3DCMs. This data can easily be stored in external databases, simplifying the whole data model structure without losing any functionality. Database existing functionality can address the problem of keeping track of the history of such data.

In any case, since there is no implementation of the versioning modules in practice, all the above can be considered speculations. Actual testing requires *CityGML v.3.0* to be officially published so that developers are able to create software implementations for it.

2.5.2 CityJSON-based versioning solution

Vitalis et al. [2019] suggest a promising data structure that allows versioning for *CityJSON*-encoded 3DCMs. The conceptual basis of the solution is *Git*, which is presented in Section 2.3.1.

In their approach Vitalis et al. [2019], use a data structure in which all versions of the model are stored in a single versioned file called *vCityJSON*, similar to *Git's* local internal database. The *vCityJSON* file is a JSON file itself meaning that the user can directly edit it if needed, since it is human and machine readable. As shown in Listing 2.1 it is similar to a normal *CityJSON* file with the difference that it contains an extra tag named *versioning*.

```
1 {
2 "type": "CityJSON",
3 "version": "1.0",
```

```
  "metadata": {...},
  "CityObjects": {...},
  "versioning":{...},
  "vertices": [...],
  "appearance":{},
  "geometry-templates":{}
  }
```

Listing 2.1: A *vCityJSON* file.

For the validation of the functionality of the data structure CityJSON Versioning Prototype (CJV) was developed. CJV is a CLI prototype software that interacts with the data structure for realizing versioning of *CityJSON* files. It is a free software available at github.

Since the paper was published, a lot of functionality has been added that is not presented in the paper. For example it now supports branch creation and merging. Inspired from *Git's* principles, it follows the distributed architecture although functionality for remote repositories is not implemented at the moment.

CJV is an attempt to port the working principles of *Git* into the 3DCM domain to enable versioning. It is implemented in *Python 3* and operated via a CLI. The main interface is shown in Listing 2.2.

```
cjv
Usage: cjv [OPTIONS] INPUT COMMAND [ARGS]...

  A tool to create and manipulate versioned CityJSON files.

  INPUT can be either a versioned file or the word 'init'

Options:
  --help: Show this message and exit.

Commands:
  branch    Create or delete branches.
  checkout  Extract version from a specific commit.
  commit    Add a new version to the history based on the NEW_VERSION...
  diff      Show the differences between two commits.
  log       Prints the history of a versioned CityJSON file.
  merge     Merges a branch to another one.
  rehash    Recalculate all objects and commits ids as hashes.
```

Listing 2.2: A snapshot of CJV's main interface screen.

In total CJV includes eight commands, four of which *Checkout, Commit, Branch, Merge* are the fundamentals as defined in Section 2.5.3. The rest *Log, Diff, Rehash* are supportive commands that improve the user's overall experience. There is also the *init* command for initializing an empty versioned file, conceptually identical to the respective *Git* command.
CJV's workflow is as follows: The user first initializes a versioned file either empty -*init*- command, or with an instance of the model committed directly into it -*init commit*- command, which is a combination of commands. The versioned file is then the system's storing structure, similar to *Git's* internal database (i.e. the '.git' folder). As long as a new version is available from editing the current version, it can be committed back to the versioned file with respective metadata such as author, timestamp and a message by utilizing the *commit* command. All the commits are hashed for consistency and integrity purposes and each commit uses this hash as an identification as well.

To extract a version from the versioned file the *checkout* command is utilized and the version is specified with its unique hash.

Branching is also supported by utilizing the *branch* command and specifying the version from which the branch stems. The user can then select for each commit at which branch it should be committed by typing its name in the *commit* command. CJV also supports branch merging of branches. Branches with common ancestors can be merged at any time integrating all the updates of one branch into the other.

2.5.3 The fundamental transactions of a *Git*-like 3D city model oriented versioning control system

Before explaining the transactions two other terms have to be defined. **Instance**, which is the a 3DCM file representing the model at a given time instance and **versioned file**, which is file that contains multiple instances of the model.

The reason the following transactions are considered fundamental is because they are the commands that implement the fundamental interaction between the user and the VCS, so that the former can take full advantage of its versioning capabilities.

Since CJV is based on *Git*, the commands' names are identical, which also makes the transition for users already familiar with *Git* smoother.

- **Checkout**: It extracts a specific instance from the versioned file. This instance is a 3DCM file itself identified by a unique ID. It does not alter the versioned file. Compared to *Git* which uses its own internal database, here a file is used instead i.e. versioned file.
- **Commit**: Integrates a specific instance into the versioned file. In contrast with checkout when a commit transaction is executed the versioned file is altered, as in it is augmented with the new instance. Alongside the instance, metadata such as time, date, the author of the instance etc are stored in the versioned file.
- **Branch**: Creates a new branch (timeline) in the versioned file that stems from a specified instance. This allows for new instances to be committed in the new timeline, leaving the original (main) branch unaltered. There is practically no limit in the number of branches that can be created.
- **Merge**: Merges a branch into another. It does so by identifying the common ancestor of the (to be) merged versions and integrating them both into one new commit, if possible automatically. It is necessary for the branches to be merged to have a common ancestor.

2.6 3D CITY MODELS EDITING SOFTWARE (*CITYJSON*)

2.6.1 cjio

cjio (see Listing 2.3), is a free and open source CLI application that can edit and validate *CityJSON* files. Through its interface the user can select among many different operators and perform the necessary actions on the *CityJSON* file. No manual editing of the file is needed. The advantage of using software to edit the file is that the file syntax and information integrity is guaranteed since human error is avoided.

On the other hand, a major shortcoming of *cjio* is that geometries can not be edited at will, at least in a straightforward way. The user is limited to using only the predefined operators. For anything more than that direct editing of the *CityJSON* file is needed.

```
cjio
Usage: cjio [OPTIONS] INPUT COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]...

  Process and manipulate a CityJSON file, and allow different outputs. The
  different operators can be chained to perform several processing in one
  step, the CityJSON model goes through the different operators.

  To get help on specific command, eg for 'validate':

      cjio validate --help

  Usage examples:

      cjio example.json info validate
      cjio example.json assign_epsg 7145 remove_textures export output.obj
      cjio example.json subset --id house12 save out.json

Options:
  --version               Show the version and exit.
  --ignore_duplicate_keys Load a CityJSON file even if some City Objects
      have
                          the same IDs (technically invalid file)

  --help                  Show this message and exit.

Commands:
  assign_epsg               Assign a (new) EPSG.
  clean                     Clean = remove_duplicate_vertices +...
  compress                  Compress a CityJSON file, ie stores its...
  decompress                Decompress a CityJSON file, ie remove the...
  export                    Export the CityJSON to another format.
  extract_lod               Extract only one LoD for a dataset.
  info                      Output info in simple JSON.
  locate_textures           Output the location of the texture files.
  merge                     Merge the current CityJSON with others.
  remove_duplicate_vertices Remove duplicate vertices a CityJSON file.
  remove_materials          Remove all materials from a CityJSON file.
  remove_orphan_vertices    Remove orphan vertices a CityJSON file.
  remove_textures           Remove all textures from a CityJSON file.
  reproject                 Reproject the CityJSON to a new EPSG.
  save                      Save the city model to a CityJSON file.
  subset                    Create a subset of a CityJSON file.
  translate                 Translate the file by its (-minx, -miny,...
  update_bbox               Update the bbox of a CityJSON file.
  update_textures           Update the location of the texture files.
  upgrade_version           Upgrade the CityJSON to the latest version.
  validate                  Validate the CityJSON file: (1) against its...
```

Listing 2.3: *cjio* with all its available operators

2.6.2 ninja

ninja is a free and open source web browser application that visualizes *CityJSON* encoded 3DCMs and allows selection of objects by clicking on its 3D representation through the viewport. If the user wants to edit the object a *JSON* snippet appears with the actual piece of information of that object into the *CityJSON* file i.e. the part of the *CityJSON* file containing the data about it (see Figure 2.9). This method consists an improvement over *cjio* when it comes to editing the attributes and semantics of an object, although editing geometries is not possible.

Figure 2.9: The interface of *ninja*. After selecting an object the *CityJSON* code snippet appears and the user can directly edit the *CityJSON* file.

2.6.3 Blender

Blender [5] is a free and open source 3D creation platform. Its functionality includes modeling, rendering and simulations among others. By default it has no way of handling any kind of 3DCM files but its extendable through its *Python* API. As will be shown in Section 5.2.1, *Blender's* data model is mapped against *CityJSON* data model to the point that it allows 3DCM file handling.

Blender's data model is quite extensive, so in this section an overview of only what is relevant to representing geometries and their attributes will be exposed. This is by no means a detailed presentation of the full data model. For that official documentation might prove useful [6].

Blender's data structure uses the object oriented paradigm. There are many object types (classes) such as *Mesh, Light, Empty, Surface, Camera, etc.* The relevant object types in the context of this thesis are *Mesh* and *Empty*. Each object type has their own built-in properties such as transformation matrices, the *collection* they belong to, which is similar to *layers* in *CAD* software. Objects are also allowed to have relations with other objects i.e. parent-child relationship and as many other properties the user wants to add which are stored as *custom properties* within the data structure of the respective object. In the case of objects that incorporate any kind of geometry, every face of that geometry is allowed to have a distinct *material*, which allows *Blender* to render scenes more naturally, based on how this type of material interacts with light.

5 https://www.blender.org/
6 https://www.blender.org/fileadmin/verse/spec/datamodel.html

3 USE CASE: 3D CITY MODEL OF ROTTERDAM

3.1 AN OVERVIEW OF ROTTERDAM'S 3D CITY MODEL PROCESSES AND HISTORY

The very first version of Rotterdam's 3DCM was published in 2011 mostly as a proof of concept and for experimentation purposes. It used the *CityGML v1.0* encoding schema. The goal for the municipality was to experiment with the potential applications of the model across its different departments and there was a plan to use the model for buildings' volume calculation. Due to the experimental nature of the project, some necessary geometric restrictions were not met [Boeters et al., 2015]. This resulted in a defective model including among others missing walls between buildings, leading to a model that was limited to serving visualization purposes only and not for building volume calculation as initial planned.

In 2016, the whole 3DCM was completely remodeled in order to eradicate the limitations of the first version with respect to building volume calculation. The input datasets used to create the model were the official 2D base dataset provided by the dutch *Kadaster*, *BAG*, [Bakker, 2009], and lidar data collected from the municipality at a density of approx 30 points per square meter. Aerial imagery was also used both for textures (oblique images) and for modeling of the buildings (stereo-pair images).

These datasets were outsourced to an external third party for the generation of the new model iteration. The model is encoded in the *CityGML v2.0* data exchange format and then stored into the *3DCityDB*. Ever since, the model gets updated in a biennial lifecycle (see Section 3.1.1), with one update carried out in 2018 and the next one planned for late 2020 early 2021. At the moment, except buildings, the model contains trees, city furniture and underground infrastructure, which are outside of the scope of this thesis.

Currently, the main use of the model by the municipality is for visualizing new constructions in the city and analyzing their environmental impact. This can happen directly on their online 3D platform or by exporting the model into the supported by the online platform file formats and working locally. According to practitioners, an evolution in the use of the model that they would like to promote, is the utilization of the model for own-applications, such as tax estimation, planning, simulations, by the users and not only for viewing purposes.

The model contains the building objects as individual entities, with each building containing three different geometries LoD 0, LoD 1 and LoD 2. LoD 2 geometry surfaces have a semantic attribute; *WallSurface, GroundSurface* or *RoofSurface*. The building address and attributes such as *creation date* are also contained within the building object.

For the needs of this thesis the municipality has kindly provided me with a part of their latest —at the time of writing this thesis— version 3DCM of 2018. The (part of) model was provided in *CityGML v2.0* files and contains buildings from the area of *Delfshaven*.

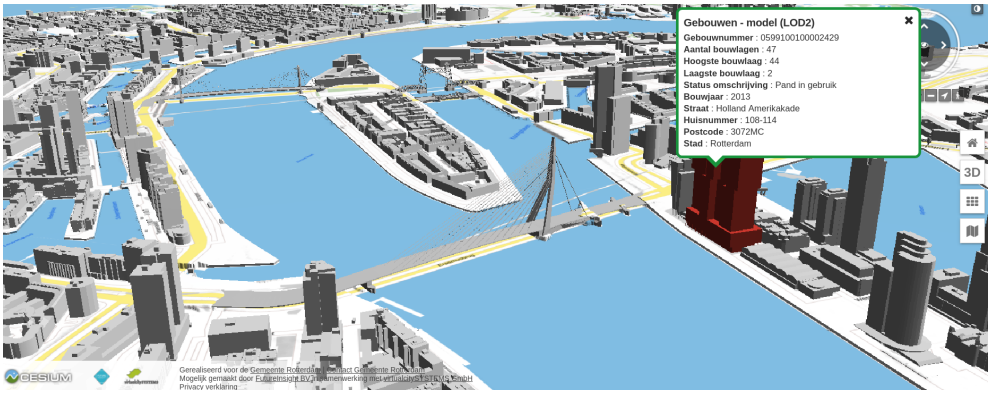


Figure 3.1: A snapshot of the Rotterdam's 3DCM through the municipality's online platform. The online platform can be publicly accessed here

3.1.1 Current maintenance procedure

From 2016 and on, the workflow of the municipality for integrating the changes in the model is the following: First, the building footprints from *BAG* —which is updated at a regular basis within the municipality— are compared with the 3DCM's objects footprint which remain static during the biennial update. There is no comparison in 3D, since *BAG* is a 2D dataset. After this side by side comparison the compared buildings will eventually fall under one out of four categories.

i. New buildings: Buildings existing in *BAG* and not in the model.

ii. Deleted buildings: Buildings existing in the model but not in *BAG*.

iii. Mutated buildings: Buildings existing in both *BAG* and the model yet with geometrical differences. Since *BAG* is a 2D dataset, the only differences that can be detected are differences in buildings' footprints.

iv. Unchanged buildings: The remaining buildings that haven't been affected and don't need any update.

It is worth mentioning here that for the 2018-2020 maintenance iteration, the buildings comparison between *BAG* and the model will be checked against the issued building permits of the past two years.

Between 2016 and 2018, 5845 buildings were created, deleted or modified in the model (see Table 3.1). A list with those buildings together with the lidar data and aerial imagery is handed in to an external contractor who (re)models the new and mutated buildings. The contractor returns the newly modeled objects in the *CityGML* data exchange format, valid against the *CityGML v2.0* encoding schema.

Then a second external contractor, hired by municipality, will incorporate the newly modeled buildings into the *3DCityDB* database (see Figure 3.2). It will also remove the deleted buildings. The deleted and mutated buildings' (initial) models are stored outside the database for history tracking purposes, alongside the list of buildings that were affected during the update.

From 2016 to 2018, around 3000 buildings needed to be changed annually, on average. The current maintenance process includes outsourcing to external parties for the main workload of the maintenance (i.e. to model new and mutated buildings). Apart from the financial cost and the prolonged two-year period that the model remains outdated, outsourcing for public agencies has another drawback. Due to european regulations regarding outsourcing arrangements, there is a limit of approximately 30000 euros imposed; that if exceeded the municipality can not

outsource the project directly to a contractor of their preference, who is already familiar with their requirements and processes. This creates the problem of re-communicating their exact needs on how the model should be updated which is time consuming, inefficient and can affect overall quality.

	2016-2018	*2018-2020*
New	3079	NA
Mutated	1228	NA
Deleted	1538	NA
Total	5845	10000 (approx.)

Table 3.1: The number of buildings that need to be updated in the last two updating iterations of the model.

In a wider perspective, there is potentially an upper limit to the amount of the buildings that can be maintained at each biennial iterations. When the number of buildings that need to be maintained exceeds the number that the financial cost can cover, some buildings will be left out of the maintenance iteration, remaining outdated for at least four years given the current processes.

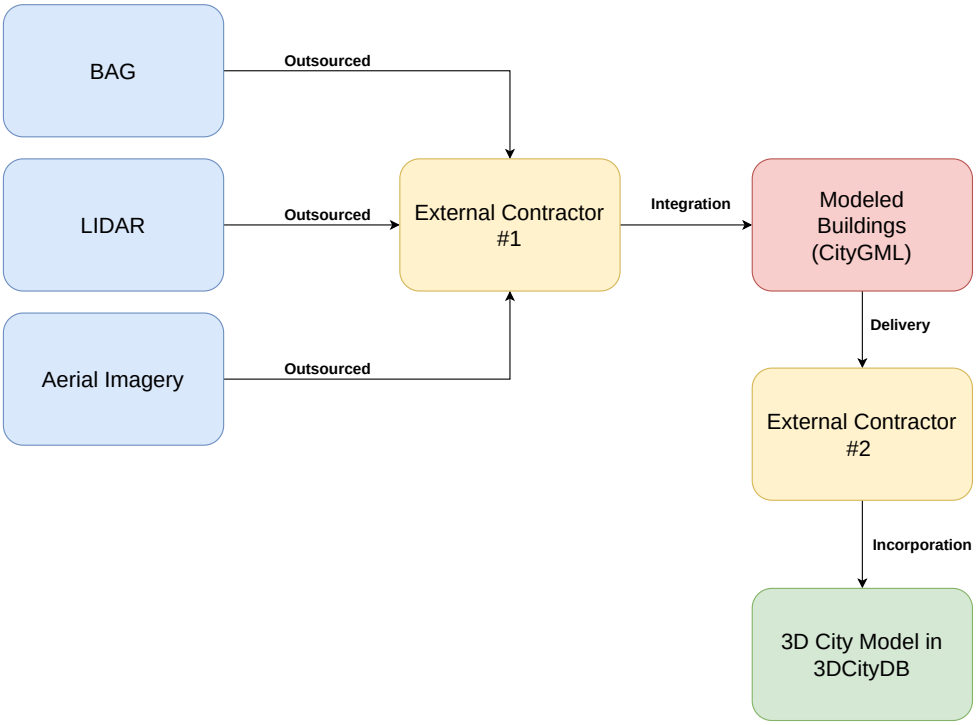


Figure 3.2: The process of incorporating the changed buildings into the 3DCM of Rotterdam. This process is repeated in a biennial lifecycle with the updated input datasets, resulting in a new iteration of the model.

3.2 IDENTIFYING THE KEY POINTS THAT WILL ALLOW THE EFFECTIVE MAINTENANCE OF ROTTERDAM'S 3D CITY MODEL

This section introduces the key points that are considered the basis for an efficient and effective maintenance of the 3DCM of Rotterdam. Following these key points as described next is expected to facilitate its wider utilization by more external parties

and domain experts. To better understand these key points, it might be useful to see them as current limitations that lead to inefficient and expensive administration of the model throughout its lifecycle. These limitations directly reflect on the willingness of practitioners to interact with it for third party applications, while simultaneously diminish the value of the 3DCM, due to data obsolescence caused by long periods of unmaintained 3DCM.

1. **Maintaining the model updated at regular time intervals**

   Arguably the most important key point to successfully maintain a 3DCM is the ability to keep all its buildings up to date with reality. As shown in Table 3.1, there are three update cases, in which maintenance of model is needed. For reference simplicity, buildings falling under these cases will be called 'pending'.

   New buildings must be modeled, missing buildings must be added, demolished buildings should be removed from the model and mutated ones should be edited. Editing a mutated building includes altering its geometry or attributes to match reality. For example when a new extension has been built or demolished or an attribute has changed for any reason.

   At the moment pending buildings are updated all at once, by outsourcing them once every two years. With this approach, the model is constantly outdated, although through the *BAG* dataset all the pending updates are known, since it is constantly maintained by the municipality. It is far from optimal to know what should be maintained up to two years in advance and yet waiting until the next iteration simply because there is no mechanism in place to integrate the updated into the model.

   Thus, it is crucial that the maintenance iterations frequency gets well-increased from a two year lifecycle to a much more frequent process. More frequent maintenance iterations means a 3DCM which follows reality closer than before, increasing its value and its appeal to interested parties. What is more, the workload of the maintenance iterations will be dramatically reduced, paving the ground for avoiding outsourcing of the maintenance.

   By dividing the massive biennial maintenance workload in smaller daily or weekly updates, allows few or even a single practitioner to handle the task successfully within the municipality. It abolishes the outsourcing necessity giving the municipality full control and supervision over the model's maintenance, similar to the *BAG* maintenance.

   In-house maintenance solves two problems at the same time for the municipality. First, it saves the financial funds that have to be spent for the outsourced maintenance. Second, there will be no buildings left unmaintained due to the financial limitations imposed by the european regulations which indirectly limits the number of buildings that can be maintained with the current approach.

2. **Keeping track of maintenance history**

   Another key point that stems from the previous one, is keeping track of the maintenance history. Keeping track of the maintenance history is equivalent to being able to go back from the current model to any previous version of it. It is very important for monitoring the advancement of the city through years, while also making sure in case the model's data is lost or corrupted it can be restored.

It can be achieved by storing the differences between consecutive versions, or by storing the whole model versions as the maintenance iterations advance. Keeping track of the maintenance history, allows future reference to the model's changes, that depicts the city's evolution over time. Trends can be identified, and urban planning decision making can happen from a much more informed point of view.

Currently, during maintenance all the deleted and the initial versions of the mutated buildings are kept, but this happens manually and outside of the main platform in which the model is stored. So, although there is a form of history kept, there is limited metadata with respect to the maintenance. Also, due to the frequency of the maintenance iteration, the temporal resolution of the city's history is severely reduced. For example, if a building's geometry has been changed more than once during the two years, the intermediate change will be lost as only the last change will be documented.

As already mentioned, every year on average around 3000 buildings need to be updated. This averages 12,5 pending buildings per day for a 240 working day year. With a VCS in place to manage the maintenance, it makes sense that the iterations remain as distributed as possible so they can remain small in size. Smaller and more frequent updates within a VCS means increased temporal resolution of the city's evolution and easier review of that progress, due to the automatic metadata creation for every iteration. It also relieves the maintainers from manually keeping track of history, because the VCS takes care of it automatically within the same platform where the model is stored. With more metadata available at a finer temporal resolution more insight can be gained and evolution patterns can be explicitly identified.

More in detail explanation of how the updates should be performed and the role of the VCS in it, will be given in Chapter 4.

3. **Testing alternative scenarios for urban planning**

As mentioned in Section 3.1, an important application for the municipality is to use the 3DCM for testing of new ideas and construction scenarios, before their potential implementation. Visualizing an idea within the model allows for environmental impact analysis and facilitates the decision making process in an unprecedented way. That is due to the fact that the model is in 3D, georeferenced in real world coordinates, thus intuitive to interpret and properly oriented which enables the planner to perform simulations, such as solar capacity, wind flow simulation in conjunction with the surrounding buildings etc.

The number of practitioners working on a planning scenario/project varies from a single person to a group of people. So, it would be beneficial that testing of scenarios can happen collaboratively and even concurrently in an effective manner, to maximize efficiency and productivity. Ideally, the whole planning procedure for the planner(s) should happen without affecting the main model -allowing any kind of maintenance to happen with no obstacles- or other practitioners working on the same project.

Keeping track of the history of the planning procedure with the same mechanism as with maintenance during the planning scenario advancement, through its different stages over time is important. Not only it allows to build the scenario step by step, but also it allows to navigate at will between these steps

for more focused analysis on different aspects of the scenario which were implemented at different phases. For better understanding this functionality can be considered as an undo button that works at any occasion.

Currently this functionality of drawing 3D objects exists within the 3D platform of Rotterdam, but with limited functionalities, compared to those mentioned above. The drawing capabilities are limited to visualization and there is no possibility to keep track of the history, either for collaborative work to happen.

4. **Motivate different domain experts to use the model for custom (own) external applications**

   The key point presented here, is not a key point that will directly have an impact at the maintenance of the model itself, as the rest of the needs presented in this section. It is though a way of indirectly increasing the model's value, which will consequently help the maintenance procedures to be improved.

   There is a big amount of different domain experts that can benefit from working with (a part of) the 3DCM for different custom applications at a local level. That is because different domain experts have completely different workflows that the online platform's functionality can not cater for. Within or without the municipality, the capability to export part of the model and incorporate it into a custom application workflow is crucial for the maximization of the model's value and the raise of domain experts interest on the model.

   At the moment, the capability of extracting specific objects already exists at Rotterdam's online 3D platform, although there is no option for editing the model prior to exporting. There are many available export formats for the user to select according to their needs. Yet when working with online platforms handling 3D data of this size, the user experience can be compromised due to the limitations of web browsers to cope with 3D information and limited internet connection bandwidth, leading to limited use of the interface, thus less custom applications. It is worth mentioning that at the same time, the functionality to draw and import shapes/objects into the online platform for visualization already exists; but it is also limited by the browsers limitations when used as a 3D object editor.

   As long as web browsers can't manipulate 3D data of such size adequately, a more responsive and edit-capable interface would improve the overall user experience; attracting more domain experts to use it for extracting subsets of the model for own use, due to the advanced editing capabilities and exporting options.

   Ideally, the user should directly download the desired area of the city from the municipality servers, then import it into the aforementioned interface, perform the necessary edits, select which buildings they want to export and create a new file that can be used for further processing. Having the choice to export in various different formats would enable domain experts to integrate the extracted output in their workflows easier.

5. **Incorporating the results of custom external applications as part of maintenance if necessary**

   This key point is about addressing the challenge of being able to incorporate any useful results that come from the external applications that the model will

be used for. For example a building's geometry in higher LoD after enriching the current one for an architectural project. Or watertight 3D surfaces as a result of a wind simulation project.

It is realistic to assume that any useful results coming from external projects will almost always be a subset of the original 3DCM, even if the 3DCM is split into tiles. Therefore, for this key point to be feasible there needs to be a mechanism in place will be able to incorporate these changes into a new instance of the model without affecting the rest of the buildings.

In contrast with key point 4, in this case the effect of this key point on the maintenance of the 3DCM is direct and justifies even more the existence of key point 4 in the first place.

Regardless the importance and the impact of such functionality this key point is out of the scope of this thesis and the proposed workflow will not address it. That is because it is quite a challenge to design and implement such a mechanism within a VCS platform in the time-frame of an M.Sc. thesis. It is certainly though a very interesting project worth researching. More will be discussed in Chapter 7.

The five key points explained previously are summarized in Table 3.2 alongside with their expected effect.

#	*Key Point*	*Expected Effect*
1	Maintaining model updated at regular intervals	Standardize the current maintaining procedures
2	Keeping track of maintenance history	Monitor the evolution of the urban fabric
3	Testing alternative scenarios for urban planning	Enhance the planning process while promoting concurrent collaboration
4	Motivate domain experts to use the model in custom applications	Raise the interest of third parties to experiment with the 3DCM
5	Incorporate the results of the external applications	Allow the work of third parties to enrich the 3DCM with more information

Table 3.2: A summary of the key points and their expected effect on the maintenance of the 3DCM

4 THE PROPOSED 3D CITY MODEL MAINTENANCE WORKFLOW

The maintenance workflow suggested in this thesis has a VCS at its core. The VCS is responsible for keeping track of all the changes while storing all the different instances. To create new instances the maintenance workflow also incorporates a 3D visual editing platform. A graphical environment for editing an instance of a 3DCM into the next, is considered necessary for the proposed workflow. Editing a 3DCM through a Graphical User Interface (GUI) is far more intuitive for the user than a CLI. It allows direct inspection and visualization of the model and the actions carried out on it respectively.

The term maintenance in the 3DCM domain does not have a universal meaning among researchers, as seen in Section 2.2. From the perspective of this thesis, maintenance is about facilitating the task of integrating changes in a 3DCM at a frequency that minimizes the duration for which the model remains outdated therefore maximizing its value.

To understand the structure and the operation of the VCS of the workflow suggested in this thesis, two main concepts must be introduced, the *core* workflow and the *multi-branch* structure. The *core* workflow describes the procedure of extracting a 3DCM instance from the VCS, updating it into a new instance and committing it back to the VCS. The *multi-branch* structure defines four branches and their interrelations, to form a conceptual approach for an effective and practical maintenance of the 3DCM that provides unobstructed and concurrent maintenance of the model by many different actors. These are inspired and adapted from the *git-flow* (see Section 2.3.1).

4.1 INTRODUCING THE *CORE* WORKFLOW

To grasp the function of the *core* workflow (see Figure 4.1), the fundamental VCS transactions introduced in Section 2.5.3 must be kept in mind. The *core* workflow only utilizes the *commit* and *checkout* transactions out of the four fundamental ones. The absence of the *merge* and *branch* is due to the fact that the *core* workflow conceptualization does not include any branches.

To explain the conceptual steps of the *core* workflow, it is assumed that the versioned file is initialized and an instance (at least) has been committed to it. The steps to complete one circular operation of the workflow are the following:

1. A 3DCM instance is exported by executing the checkout transaction on the versioned file resulting in an exported 3DCM instance.

2. The exported 3DCM instance is imported into the 3D visual editing platform (ex. *Blender* [Blender Foundation, 2019]) where it can be geometrically modified and and attributes can be edited, so that all the changes are incorporated in the model.

3. The current state of the model is exported from the visual editing platform in an updated 3DCM instance.

4. The updated instance is committed back to the versioned file alongside some metadata such as the author's name, timestamp and a message describing the updates performed. The VCS takes care of identifying what has been changed thus keeping track of the history of the model.

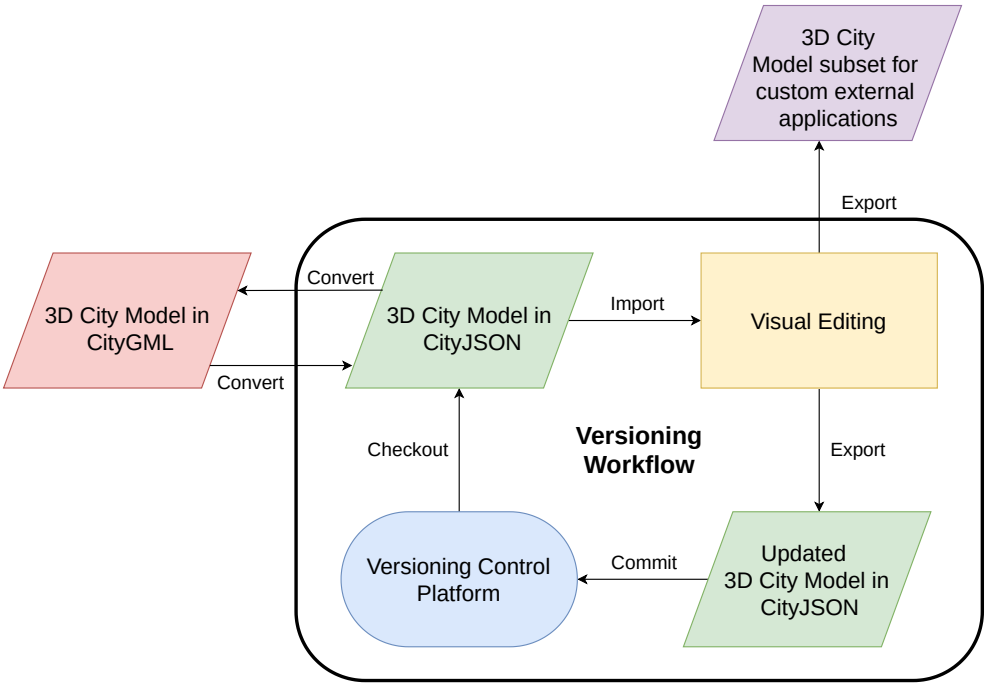


Figure 4.1: The *core* workflow

The conceptual steps explained above belong to the VCS environment of the *core* workflow (see Figure 4.1). These are the steps that have to be performed every time the 3DCM needs to get updated.

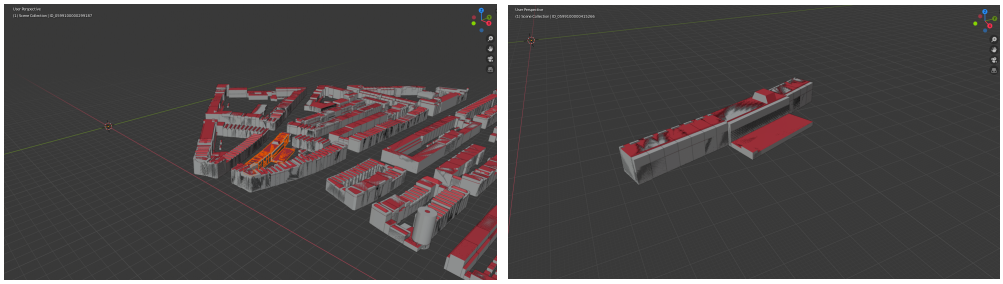
They address key points 1 and 2, as defined in Table 3.2, because they allow the 3DCM to be maintained while the VCS is keeping track of the 3DCM evolution over time.

There are however two extra steps that are not involved in the maintenance process directly. Those are the initialization step that creates the versioned file where all the new instances will be committed to and the extraction of isolated part(s) for custom application step that is optional.

The initialization step needs to be executed for the whole VCS platform to be used. It is an once-off transaction that can be considered insignificant, yet included in the schematic representation of the *core* workflow as it belongs there.

Extracting a subset of the 3DCM for custom external application is an optional step of the *core* workflow that is based on the functionality of the visual editing platform. This functionality can provide a part of the 3DCM to be used for external custom applications. For example, an architect might need to isolate and extract a neighborhood in which they wish to visualize a project at their own custom software or for analysis purposes. Or an urban planner might want to do the same at a whole region for a big scale gentrification.

It addresses key point 4 from Table 3.2 since it provides external parties with parts of the model at various data exchange formats. This functionality is not directly



(a) The selected *CityObjects*. **(b)** *Blender's* scene with only the selected *CityObjects*.

Figure 4.2: The subset before and after cropping

```
{
"type": "CityJSON",
"version": "1.0",
"metadata": {...},
"CityObjects": {
  "ID_0599100000415249": {...},
  "ID_0599100000415250": {...},
  "ID_0599100000415251": {...},
  "ID_0599100000415252": {...},
  "ID_0599100000415260": {...},
  "ID_0599100000415261": {...},
  "ID_0599100000415262": {...},
  "ID_0599100000415263": {...},
  "ID_0599100000415264": {...},
  "ID_0599100000415265": {...},
  "ID_0599100000415266": {...},
  "ID_61c18aaa-1d05-4a6e-9cd4-3adb20b44e04":{
    "geometry": [...],
    "type": "BuildingPart",
    "parents": "ID 0599100000415253"},
  "ID_774bdcc9-2ece-4289-8bd7-912af31f481a": {
    "geometry": [...],
    "type": "BuildingPart",
    "parents": "ID 0599100000415253"
    },
 "ID_0599100000415253": {[...]}
 },
"vertices": [[...]]
}
```

Listing 4.1: The new cropped instance exported in *CityJSON* format.

related to the maintenance of the 3DCM as it does not contribute to the maintaining of the 3DCM itself. What it does though is reaping the benefits of using a 3D visual platform as an editing platform, at no extra cost since the platform offers that functionality anyway. Considering that the municipality of Rotterdam wishes to attract more domain experts to utilize the model for own custom applications, it is a functionality that will motivate external domain experts to at least experiment with the potential applications of the model. In turn, it is expected that the interest of a broader range of practitioners to use the model will be increased; increasing the need for better and more effective maintenance as well, in an indirect way.

It is also the basis for satisfying the fifth key point (see Table 3.2), which although it out of the scope of this thesis.

4.2 INTRODUCING THE MULTI-BRANCH STRUCTURE (BASED ON THE GIT WORKFLOW)

The *multi-branch structure* is a system of branches inside the VCS, designed to parallelize the *core workflow*, so that multiple practitioners can concurrently and unobstructedly work on the model's different maintenance needs; with the capability to integrate all their work together in a single 3DCM instance, when needed.

The connection between the *core* workflow and the *multi-branch structure* is shown in Figure 4.3. In this graph, every circle represents a commit i.e. a new updated state of the model. The *core workflow* —its VCS environment more specifically (see Figure 4.1)— is the mechanism that allows the creation of every next commit (circle in Figure 4.3. In other words, between every two consecutive commits within the same branch in the *multi-branch structure* the *core workflow* needs to be carried out.

This thesis proposes that the *multi-branch structure* is comprised by four branches: *main, maintenance, scenario* and *release* as shown in Figure 4.3. They are considered to cover the maintenance needs of a 3DCM, while facilitating the concept of scenario testing for future projects and applications. The workflow is adaptable and flexible in the sense that more branches can be used if necessary. The branches with their interactions are the following:

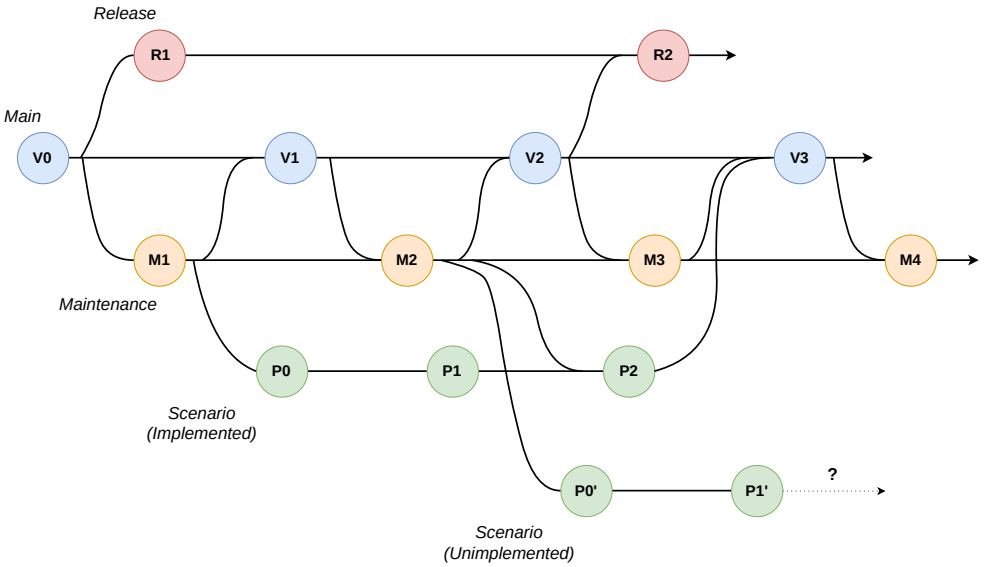


Figure 4.3: The multi-branch schematic representation of the branches and their interrelations.

- **Main**: Main branch with the latest "true" up to date version of the 3DCM. It is the 'central' branch of the VCS, acting as a reference basis for the 3DCM. *Maintenance* and *release* branch stem from it. *Maintenance* and *scenario* can merge back to it while *release* no.

- **Maintenance**: Stems from *main* branch and merges back into it at (preferably) similar time intervals (see Section 4.2.1). The role of this branch is to be the workspace in which the maintenance —due to the actual changes— is carried out before getting integrated back to *main* branch. For example when a building's geometry is mutated due to a new compartment or a building is demolished completely (see Figure 4.4).

  To update *main* branch its latest instance is branched into *maintenance* branch.

After conducting the required changes within *maintenance* branch, the new (updated) instance is merged back to the *main* branch, thus updating it.

It can also merge into *scenario* to integrate all the maintenance changes that have taken place while *scenario* was evolving. This should happen only when the tested -in *scenario* branch- scenario is going to be realized, and the merge should happen right before it is merged into *main*.

In case of massive volume of updates (e.g. a new acquisition method that leads to all buildings being measured more accurately, thus every building geometry has to be updated) that should happen at a relatively small amount of buildings per commit. That way, in case of errors or for future reference to the changes, identifying the responsible commit will be easier. This method also increases the "resolution" of the update in the sense of splitting a bulky massive update into smaller more manageable from every aspect.

- **Scenario**: Stems from *maintenance* branch and merges into *main* if a proposition is realized. It is the workspace that any new idea or scenario can be tested, without affecting either the *main* or *maintenance* branch. Except testing new ideas for evaluating their multi-aspect impact prior to construction, this branch is the workspace for conducting simulations such as wind flow analysis or solar capacity of the buildings etc, where a 3DCM comes pretty handy.

  Instead of merging directly into *main* branch, the structure is designed such so that *maintenance* first merges into it to integrate all the changes that happened due to maintenance; right before merging the realized proposition branch into *main*. That is in order for the implemented scenario to be integrated into *main* as smoothly as possible.

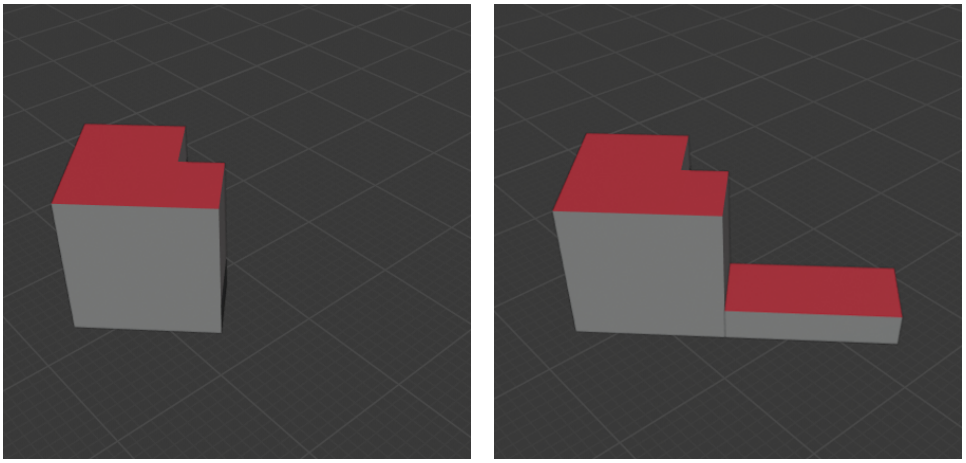
  Every different project that has to be tested and evaluated will be contained in a distinct *scenario* branch. The (new) *scenario* branch which will stem from the latest *maintenance* instance available, unless otherwise required. It will be named accordingly based on the project's name.

  This functionality addresses key point 3, by assigning a whole branch for testing new scenarios and simulations, leaving the main maintenance process of the 3DCM unaffected.

- **Release**: Stems from *main* branch but doesn't merge anywhere. It is an one direction branch in the sense that it shouldn't merge back, because its role is to contain standalone instances of the model that can be given to the public for general, educational or research purposes.

  At this point someone could argue that *release* is not needed, since the same functionality could be covered by *main* with the use of appropriate tags. There are certain reasons though that *release* is a really good option. For example, *main* contains the full 3DCM, which might contain certain attributes for every building that if published would possibly cause privacy issues. Consequently, any sensitive piece of information or personal data should be wiped out of the model before publishing. So a "trimmed" version of the model is needed in which the 3DCM is clean of any kind of personal data.

  Second, having a dedicated branch for this purpose simplifies the whole function of the workflow with more clear and dedicated roles for every branch.



(a) A building at its initial state

(b) The same building with an extension

Figure 4.4: An example of a mutated building at which a compartment has been added

4.2.1 Maintenance iterations frequency

The frequency at which maintenance iterations occur, will affect the workload of each iteration and potentially the number of practitioners needed to carry out the task in a given time period. In the long term it all reflects into efficiency of the VCS. Assuming that the model can be maintained at will, the optimal frequency would be the one that minimizes the duration that the 3DCM remains outdated, yet making sure that there are enough changes to justify a maintenance iteration for that given frequency.

There are two distinctive alternatives approaches to define the maintenance iteration frequency: Launch an iteration when a fixed number of buildings need to be maintained and launch an iteration at fixed time intervals.

Apart from defining a kind of standardization of the maintenance iterations frequency, the two alternatives proposed have a second goal under the scope of this thesis. The goal is to give an idea to the reader, of how a considerably irrelevant decision affects the VCS function and can create a completely different user experience. As an example, favoring the fixed time interval might create the need for more practitioners to maintain concurrently which can complicate the integration of the changes.

All in all, if the workflow were to be implemented in practice, finding the optimal maintenance iteration frequency would be a matter of time and experimentation within the respective agency. Adapting the maintenance task to meet their management and administration needs would dictate what applies best.

Fixed number of buildings

A simple arbitrary approach to get an estimation of the frequency at which the iterations should happen is to divide the number of total changed buildings per year by the working days of the year to calculate the average number (x) of changed buildings per day. An example of a mutated building is given in Figure 4.4. Then as long as there are at least x buildings pending at any given day they should be updated and committed to the VCS. Considering that the rate at which buildings change is not stable, the frequency will also not be stable.

The benefit of this approach is that it favors consistency with respect to the workload

of each iteration. A consistent -thus predictable- workload allows better resource allocation, in this case practitioners man-hours. Based on real data from the municipality of Rotterdam, it is expected one practitioner alone to be able to carry out the whole iteration within a working day if based on their average daily pending buildings estimation. Furthermore, the number of buildings changed at each iteration will be consistent by definition, which in the long term facilitates the review of the city's history by practitioners.

The drawback of this approach is that since the number of pending buildings does not grow at a steady rate, there could be a long period between two iterations. At the same time it is possible the number of the pending buildings to rise steeply without any warning, increasing the workload and potentially requiring more than one practitioners to address the situation. Finally the number of the pending buildings should be constantly monitored.

Fixed time interval

The second approach to define the maintenance iteration frequency is to define a fixed time interval at which an iteration should be launched. The advantage of this approach is that it is simpler than the previous with respect to estimating when an iteration should be launched. There is no need for estimating the average number of affected buildings per day and keeping track of the changed building count on a daily basis. It also keeps the duration at which the model remains outdated consistent.

The downside of this approach is that there is no way to predict the workload of each iteration, since the rate of building change is unstable. Assuming that the iteration is planned to happen in a single workday the number of practitioners required to carry out the task might vary between iterations, so resource allocation optimization might be compromised. In case though that assigning more than one practitioners is an affordable option, this approach can become quite efficient.

Although, there is an emerging downside by having more than one practitioners maintaining the 3DCM concurrently. That being the chance of merging conflicts (see Section 4.2.2 increases a lot, which will require the practitioners to be familiar with conflict resolution strategies. In the case of a single practitioner the chance of a creating a conflict is minimal.

4.2.2 Managing Merging Conflicts

A merging conflict occurs when the VCS does not know how to integrate information coming from two -to be merged- branches. For example, when the same piece(s) of information have been edited in two different ways -and branches- and there is an attempt to merge these two branches, as shown in Figure 4.5. On the contrary, there is no conflict in Figure 4.6 since different pieces of information have been changed.

In distributed VCSs, the work happens locally with each user having their own copy of the remote repository as a local repository. Keeping these two synchronized makes conflicts practically unavoidable. Keep in mind that for every user the number of total branches that interact (local ones and the remote) grows proportionally; since every one of them has a copy of those branches stored locally at their systems as explained in Section 2.3.

The more local repositories needed to be synced with the remote, the more chances of two or more users editing the same piece of information. In other words, ev-

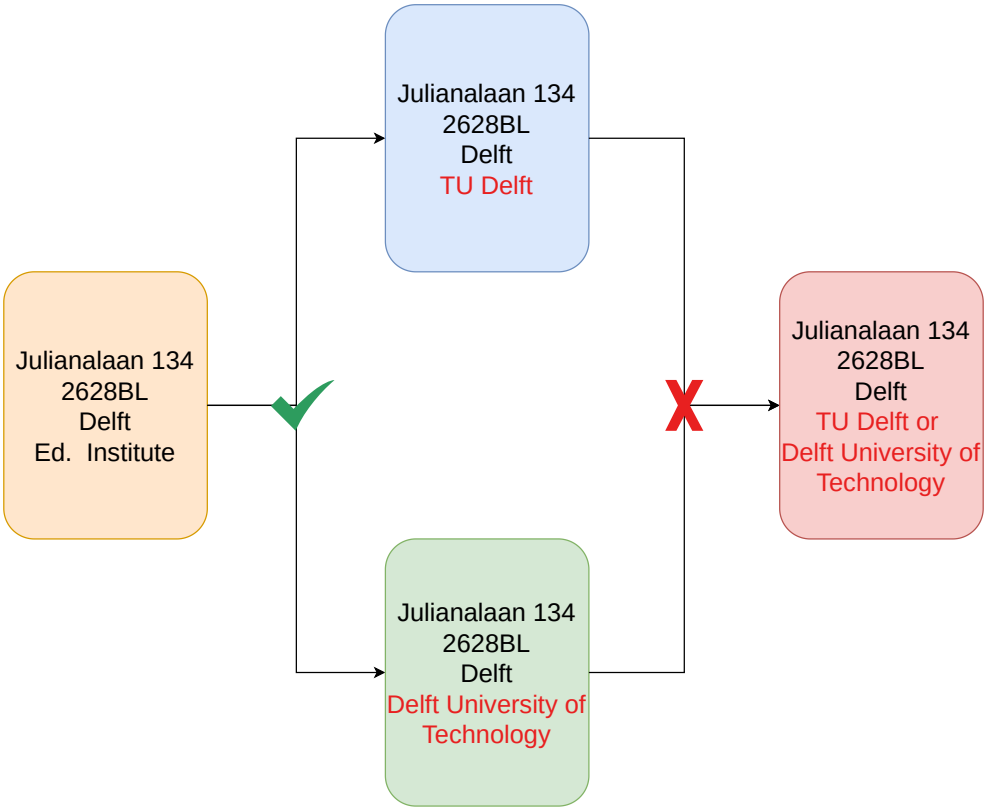


Figure 4.5: The same piece of information *Ed. Institute* has been edited differently in two different instances *TU Delft* and *Technical University of Delft*, creating a merge conflict upon merging.

ery attempt to synchronize each local repository with the remote, regardless of the direction of the synchronization (local to remote or vice versa), is prone to merge conflicts, as long as more than one users can influence the remote repository.

Before going more in depth with identifying the points of the workflow at which conflicts can occur, it is important to make some assumptions based on the maintenance needs introduced in Section 3.2. Those assumptions are related to the number of actors that will be working concurrently at the same branch and their actions. The assumptions are the following:

1. There will be no practitioner working directly on the *main* branch. That is because based on the *multi-branch* structure, the *main* branch is not directly updated. Any form of updating of the *main* branch happens only via merging the *maintenance* branch and the *planning/proposition* branch into it.

2. The *maintenance* branch can be maintained by either one or many practitioners (depending on the iteration frequency and resource allocation of the agency), in order to keep the model up to date with reality by implementing the required updates into it.

3. More than one practitioners are expected to be working concurrently on the *planning/proposition* branch. That is solely due to the fact this branch was created for ideas and scenarios to be tested, which means its functionality would be limited if only one practitioner/domain expert is able to work on it.

4. The *release* branch has no special needs in terms of managing. Since the municipality can decide whenever they want to publish a new "trimmed" version of the model, which is expected to happen at long time intervals of possibly

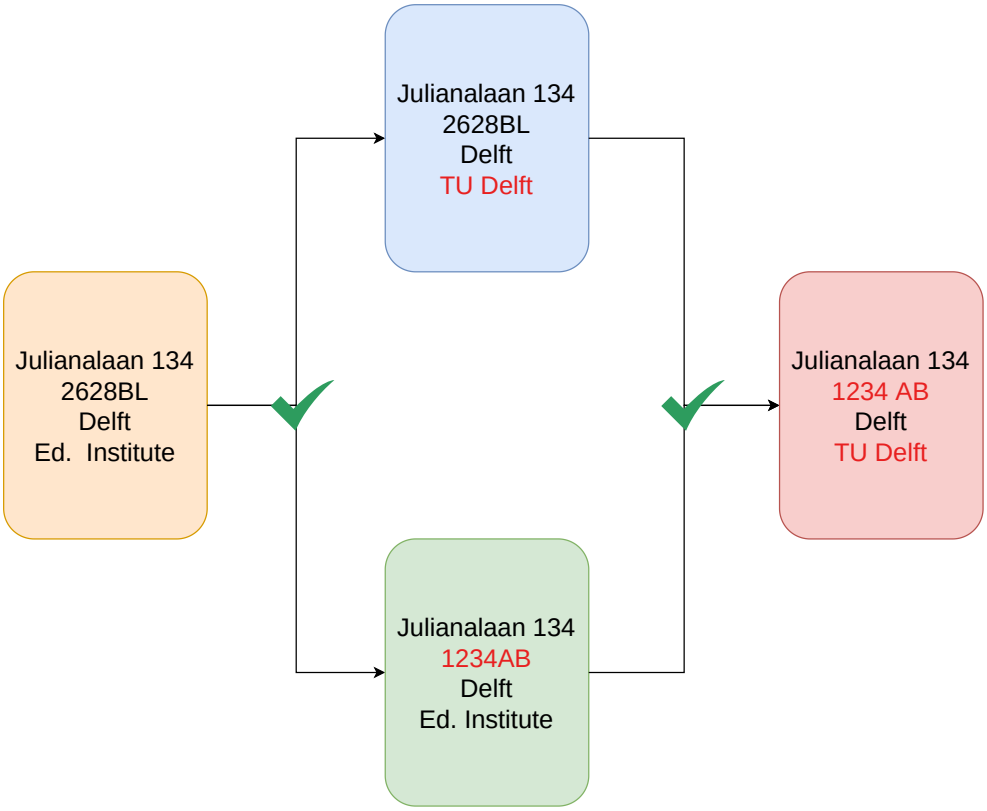


Figure 4.6: *2628 BL* and *Ed. Institute* were changed in the two different instances to *1234AB* and *TU Delft*. They are different pieces of information so no conflict is created upon merging.

few years, the workload of committing a "trimmed" version into the *release* branch is considered negligible. Furthermore, the workflow of trimming the 3DCM off of personal information could be easily automated; minimizing the workload of creating a publishable instance.

5. The users will be asked to synchronize their local repository with the remote every time the initialize a working session, so everything is up to date with the remote repository upon the beginning of their work (pull), to avoid extra conflicts. Session, in this context is the period between two consecutive commits of a user to the VCS. Furthermore, the VCS will require that the local repository is synced again with the remote repository right before uploading (push) the changes in the local repository to the remote.

Branch	*Number of practitioners*
Main	0
Maintenance	1 or >1
Scenario	>1
Release	0

Table 4.1: The number of concurrent users per branch.

Based on these assumptions the points of the workflow that conflicts can occur can be identified. For explanation purposes, two categories of conflicts are defined: *Single user induced conflicts* that happen when merging between branches within a local repository and *Multi-user induced conflicts*, that happen when attempting to synchronize between a local and the remote repository clashing with what other users have already pushed on it.

Single user induced conflicts: Every local repository has a single user, meaning that conflicts that arise in this situation are single user dependent. Based on the sync assumption, such conflicts can happen in the following case:

1. Merging *maintenance* into *planning/proposition* when the affected in the *scenario* objects were also changed in the *maintenance*. For example, when testing a new idea in the *scenario* and during the evolution of the new idea within *scenario*, the affected objects were also affected by *maintenance* as well, creating ambiguity for the VCS.

   This case though, can only be valid when the same user works on *maintenance* and *scenario* during the same session. In any other case that work is done on a single branch there is no chance of a single user induced conflict.

Multi-user induced conflicts: The role of the remote repository is to be commonly accessed and updated by all users. For the remote repository to be updated, the local repository changes have to be uploaded (pushed) into it. Since by definition there are more than one users uploading to it, conflicts will occur.

According to the fifth assumption, the VCS will require the local repository to be counter-checked against any possible updates in the remote repository since the first sync upon the beginning of the session. This is the point at which all the conflicts between local and remote will become apparent. As long as they are resolved the local changes will be uploaded (pushed) to the remote repository without any problem.

Keeping in mind the interrelations between branches as they were introduced earlier, the potential conflict points are the following:

1. Based on the *multi-branch structure*, the most conflict-yielding situation is that of multiple users working on the same project inside the same *scenario* branch. Editing the same object(s) with another user that has already uploaded (pushed) their work on the remote repository -in not an identical way- will create a conflict upon trying to sync the local repository with the remote.
2. If only a single user is responsible to carry out the maintenance task, the situation is simplified since there will be no concurrent updating in the maintenance branch, thus no conflict chances.

   In case of more than one users working on the *maintenance* branch concurrently as mentioned in the second assumption, conflicts might occurs upon merging the distinct updated instances of different users. In this case though, since the end result of the updates can be known beforehand (i.e. what should be updated into what), the maintaining process could be easily standardized or divided between the users. This way they would never get to work on the same object. These two strategies can limit the amount of conflicts, or potentially eliminate them.
3. When a proposed scenario or idea is realized, the *maintenance* branch has to be merged into *scenario*, so the *scenario* can then be merged into *main*. If the buildings affected by the newly implemented proposition have been changed in *maintenance* while the *scenario* was evolving, there will be conflicts upon merging *maintenance* into *scenario*. The conflicting objects will be those buildings that were affected by the new idea but also maintained along the process.

A summary of this cases is presented in Table 4.2

#	Potential conflict cases
1	Multiple users working on *scenario*
2	Multiple users working on *maintenance*
3	Merging *maintenance* to *scenario*

Table 4.2: Cases in which conflicts are expected to occur.

4.2.3 The 'smallest entity' problem

To define what a conflict is, the term (same) *piece of information* was used, which if changed in more than one ways yields a conflict. The term piece of information in this context is equivalent with the smallest entity that the VCS can identify as altered.

While in the software development domain that piece of information is every character of every line of the source code, the situation in the 3DCM domain is not such. Source code files have no hierarchical entities, meaning that all the information lies on a single hierarchical level with respect to how the computer parses source code.

In contrast 3DCM file structures are hierarchical structures meaning that one information entity lies inside another and so on. This means that a VCS designed to handle 3DCM files has to (arbitrarily) establish what is considered as this smallest entity (in terms of hierarchy levels), which can perceive as altered. For example, a building can be an entity, which incorporate attributes, geometries (different LoD), which are individual entities as well, which in turn contain different faces which are also individual entities and so on. A schematic representation of the hierarchy structure of a 3DCM is shown in Figure 4.7.

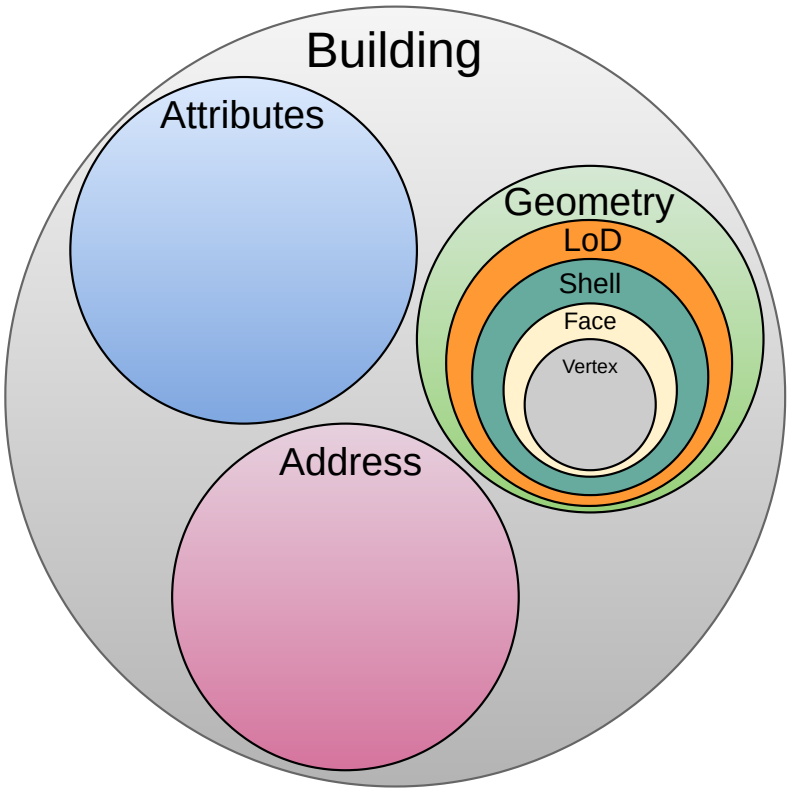
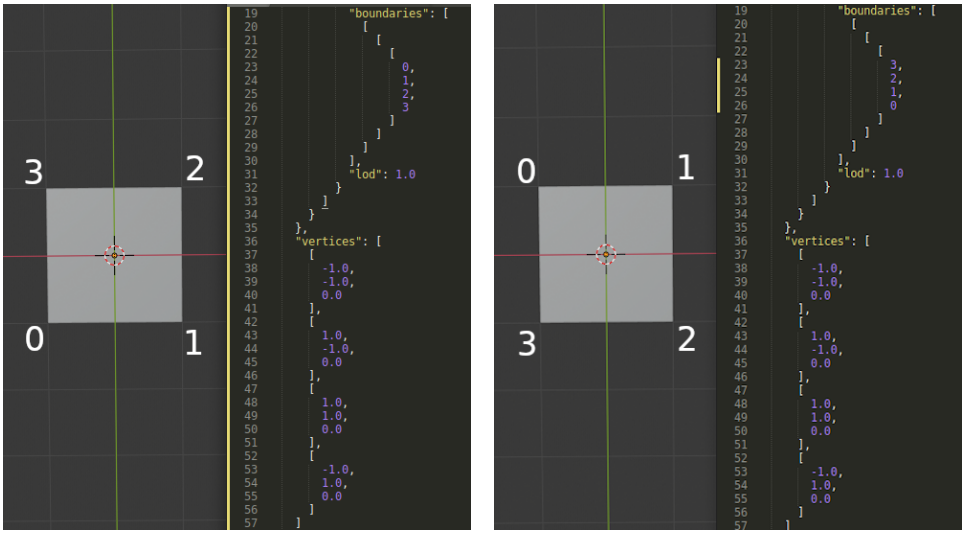


Figure 4.7: The hierarchy levels formed by the different entities within a 3DCM data structure. In the VCS proposed the smallest entity is the 'Building' level (gray circle).



(a) A plane surface with its vertices stored in default order

(b) The same surface with the order of the vertices reversed

Figure 4.8: The same planar surface saved with its vertices in default and reverse order

The hierarchical level of the smallest entity, at which the VCS will be able to perceive conflict is of utmost importance for the efficiency and robustness of it. For example, if the building entity level is considered as the deepest level for the VCS, -meaning the building object is the smallest entity- removing a complete LoD geometry and changing one of its attributes in different instances will yield a conflict, as for the VCS the same building object entity is changed (see Figure 4.4).

This will translate into seemingly more conflicts, while in reality not the same piece of information -as a human would see it- has been altered. Having more conflicts that are actually not real conflicts requires more time to handle them, affecting the efficiency of the VCS. The advantage though, of choosing a higher entity level as the deepest, is that the structure of the VCS mechanism that identifies conflicts can remain simpler and that the user's supervision required for conflict resolving at a higher hierarchical, offers a kind of quality control of the end product.

Ideally, it would be very convenient for the VCS to be able to distinguish changes at the deepest hierarchical level possible. It would eliminate the fake conflicts due to smallest entity problem. There is however a big challenge to overcome to do so. That being the data structure of 3DCM files, with geometrical entities that do not have to be in a strict specific order in some cases, for example the order of the faces or even the order of vertices of a face of a geometry (see Figure 4.8).

In order for the VCS to avoid these kind of conflicts, it should be able to identify the difference between an actual geometric difference in the model and a difference in the file itself due to reordering its records, which imposes no change in the model whatsoever. It would be no surprise if manually choosing the correct instance between two conflicting ones would prove more robust than developing a mechanism to identify between the types of difference explained above.

That being said conflicts created due to arbitrary selection of the smallest entity can be counteracted with two fundamentally different approaches or a combination of the two.

1. Solving the problem on the data structure level: Adjusting the 3DCM data structure so that hierarchy becomes as flat as possible avoiding nested entities

thus conflicts due to the depth of entities. The benefit of this method is that it tackles the problem at its root, before it even happens. The disadvantage though is that the data structure has to be severely compromised with respect to what is considered best for storing and organizing the information.

2. Solving the problem on the software level: Allow the conflicts to happen due to the data structure of the 3DCM and resolve them *a-posteriori*. The disadvantage with this approach is that conflicts will happen although not the same piece(s) of information have been altered. But the data structure can remain as primarily designed, which is a very big advantage. Also, software can be customized to handle conflicts as required, based on different resolving conflict policies, which makes this approach far more flexible than changing the data structure (see Section 4.2.4)

4.2.4 Resolving conflict policy

Resolving a merge conflict inside a VCS is all about choosing which version between the conflicting ones should be kept. Using the example in Figure 4.5, resolving the conflict means choosing between *TU Delft* and *Delft University of Technology* i.e. what instance (blue or green) will become accepted.

It is not necessary that one of these two options is selected. It is possible to choose one third option, that is not included in neither of the two conflicting instances, if required. But in most cases it will be one of the two conflicting instances that contains the piece of information that will finally be accepted.

In theory, there are endless approaches in resolving conflicts, when it happens on the software side. From zero tolerance approaches that put the merging completely on hold and let the user decide 100% what should be done, to completely automated approaches in which the VCS's behavior upon conflicts can be designed at will.

The main benefit of the zero tolerant approach is the complete supervision and control of the conflict resolution by the user, while the main drawback is that manual conflict resolution requires a lot of time. On the other hand, complete automation of conflict resolution has the opposite characteristics. This means that the conflict resolving algorithm follows predefined steps to resolve the conflict but it is almost guaranteed that the resolution will not be meaningful.

Nevertheless, in practice, it is not possible to automate the conflict resolution in such a way so it produces the same results as manual conflict resolution. That is because the VCS is a structure that has no cognitive abilities, so it is unable to know what should be kept and what should be discarded upon a conflict.

Due to this reason all the conflicts that will occur as described in Section 4.2.2 are expected to be resolved manually by the user. Since manual editing conflicts are time-consuming yet unavoidable, it is important for the workflow to be followed such so that conflicts remain as limited as possible.

One example to understand how the number of conflicts can be affected by the workflow customization, is related to the number of concurrent users in the *maintenance* branch. *Maintenance* is a branch that a single user can potentially carry out the required changes; given the correct maintenance frequency. Opting for more than one users increases the change of the same part of the model being edited, thus creating conflicts that could be avoided. In case it is necessary for two or more

users to work on the *maintenance*, it is possible to still avoid conflicts, if there is a previously agreed plan that makes clear who is working on which buildings, so that there is no overlap in their work areas.

5 IMPLEMENTATION

The workflow proposed in Chapter 4 was implemented as a proof of concept. The aim of the implementation was to showcase the feasibility of the concept, testing its performance and identifying both practical and conceptual shortcomings. The prototype software used to implement the concept is CJV that was introduced in Chapter 2 and *Up3date*, a *Blender* add-on able to visualize, edit and export instances of 3DCM encoded in *CityJSON v.1.0* format.

Both CJV and *Up3date* have been developed to work with 3DCM instances encoded in the *CityJSON v.1.0* data exchange format. *CityJSON* was selected due to its programming friendliness and for its data structure itself that makes geometry storing very efficient and easy to be parsed.

5.1 THE IMPORTANCE OF VISUAL EDITING CAPABILITIES

There are plenty of ways to edit a *CityJSON* file. The simplest of them is by manual direct editing of the file itself since it is a JSON-based human readable text file. This is however the most error prone method since it requires the user to be very familiar with the *CityJSON* file structure. It is also rather exhaustive to constantly make sure that the syntax of the file remains valid, throughout the process.

A second method is to use cjio. This method is arguably an improvement over working with direct manual editing. The syntax integrity is guaranteed since all the modifications are done by a script but the user still needs to be somehow familiar with the *CityJSON* file. That is knowing the ID of the objects they want to edit since that is the way to identify them through the interface of *cjio*. So far there is no way to edit a 3DCM directly though a GUI. The closest alternative to it is ninja.

The motivation for implementing a tool that would allow visual editing capabilities can be attributed mainly to two reasons. First, as a proof of concept initiative since there was no software capable of visual editing of a 3DCM. Second, due to the fact that by having visual editing capabilities makes the whole interaction between the user and the model far more intuitive. That is because the user can visualize the whole 3DCM and perform all the changes graphically, which are shown on the model in real time. It increases the inspection capabilities of the model as well since the user can simply navigate around and interact with all the objects. Finally, it attracts users that have no familiarization with the 3DCM files structure to work with it and simplifies the editing procedure for the experienced users.

All in all, visually editing a 3DCM is less error prone since the user never gets to interact with the file itself, while all the changes made can be monitored visually in a 3D environment making the 3DCM editing procedure much more trivial.

5.2 INTRODUCING *UP3DATE*

Blender (see Section 2.6.3) is a powerful free and open source 3D software suite capable of producing professional results in the 3D domain.

Although *Blender* was never designed for handling 3DCMs or big data in general, it was chosen to serve as the underlying platform because of its versatility and its extended functionality and because of the fact that it is free and open source, so it can be endlessly extended and improved.

Up3date is an add-on for *Blender* that extends it to handle 3DCMs encoded in *CityJSON v1.0*. It is developed in *Python* via *Blender's* API. It is compatible with *Blender v.2.80* or greater. It is able to import a 3DCM file encoded in *CityJSON v1.0* into *Blender's* scene enabling all *Blender's* functionality on it and export it back into a new *CityJSON v1.0* file. It stores all the information contained in the file (except textures and template-geometries) leading in zero loss of information during editing and exporting back into *CityJSON* format. All semantic information, object attributes and multiple LoD geometries are maintained upon importing and exporting.

It is designed such that any given *Blender* scene can potentially be exported into the *CityJSON* format. This means that *Up3date* can be utilized as an effective converter between *CityJSON* and all supported by *Blender* formats and vice versa.

The source code of *Up3date* is free and open and can be found at github.

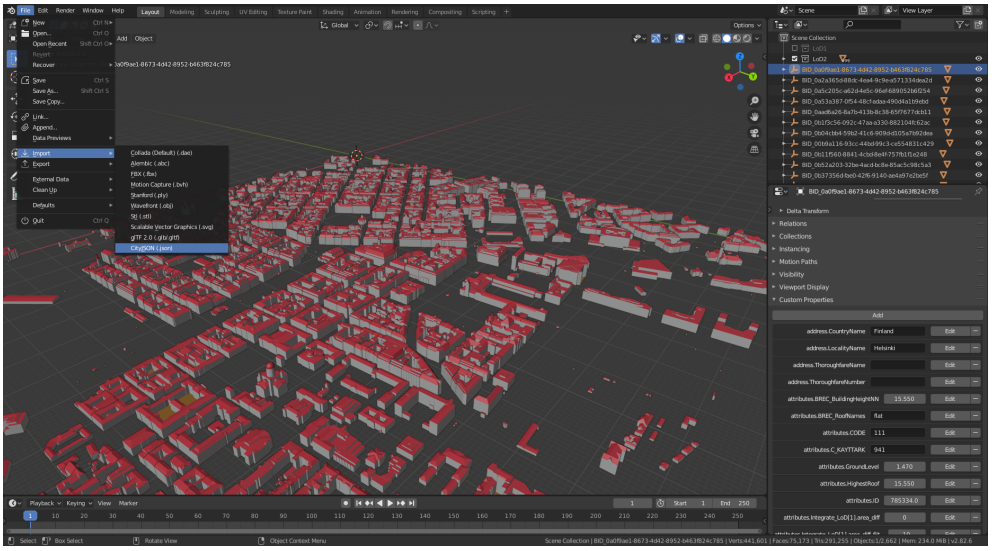


Figure 5.1: A screenshot after importing a 3DCM into *Blender* using *Up3date*

5.2.1 Implementation specifications

A single object can have only one representation in *Blender*, so there need to be a workaround for the multi-LoD geometries that can be found in *CityJSON* files. This workaround allows *Blender* to store all of them, in order for the whole importing editing and exporting procedure to become lossless.

First, each *CityObject* corresponds to an *Empty Blender* object that contains all its attributes as *custom properties* in *Blender's* data model. In case the attributes have nested structures themselves name prefixes are used at the beginning of the name of the *custom property* within *Blender*.

Each geometry —different LoD— of each *CityObject* is a distinct *Mesh* in *Blender*, which is related to the *Empty* object that represents the *CityObject* as its child. In the case of *CityObjects* that are *building parts* again an *Empty* object represents that *CityObject* which has as many children —*Meshes*— as the LoD count of its geometry. Finally, the whole *Empty* object that represents the *building part-CityObject* will be a child of the *CityObject* that represents the entire building entity.

Finally, different LoDs (i.e. LoD 0, LoD 1, LoD 2 etc.) are stored in different *Blender's collections* for organization and visualization purposes. Their attributes such as their LoD and geometry type (*Solid*, *MultiSurface* etc.) are stored as *custom properties* within the *Mesh* objects data structure itself.

Surface semantics, are stored as *materials* for every face of every LoD 2 geometry.

A summarizing of the entity mapping is given in Table 5.1.

CityJSON	**Blender**
CityObject	*Empty* object (all attributes stored as *custom properties*)
Geometry (LoD)	*Mesh* object (all attributes stored as *custom properties*)
Semantics	*Materials* of LoD 2 geometry faces

Table 5.1: Mapping between *CityJSON* and *Blender* entities

6 TESTING

6.1 DATASETS AND PREPARATION

The performance of the implementation within the scope of this thesis is an indication of how well the workflow handles the fundamental transactions for both the visual editing and the versioning component. To evaluate it, some of the scenarios described in Chapter 4 were tested with real 3DCM datasets provided by *Gemeente Rotterdam*. The aim of testing is to simulate expected interactions between users and model, identify existing shortcomings both in *Up3date* and CJV and interpret them with respect to the conceptual workflow.

The datasets on which the testing was carried out contain building objects, with all their attributes and semantics but no textures, although there are available textures in the 3DCM official portal of Rotterdam. They were created from the latest information collected from *Gemeente Rotterdam* and they were delivered encoded in *CityGML* format.

Each dataset contains a different area of the city, i.e. the complete model is divided into tiles. Without any specific reason, the given datasets are the tiles *B-3_18*, *B-3_19*, *B-3_20*, *B-3_21*, *B-3_54*, according to *Gemeente Rotterdam* identification system. They all contain three different LoD, those being LoD 0, LoD 1, LoD 2. They were converted into *CityJSON* encoding using the free software citygml-tools.

For consistency purposes, all the files prior to testing were imported into *Up3date* and exported back into *CityJSON*, without carrying out any changes. That is because the order that the objects are stored in the original file and the file created after exporting through *Blender* is not the same. Omitting this process would lead to false conflicts being detected by CJV compromising testing. For reference reasons this dataset will be called "normalized".

The order that the *CityObjects* are stored after exporting from *Up3date* is different because of the way the exporter writes the *CityJSON* file. More specifically, the function used to write the *CityJSON* file does it in an alphabetical order. So technically, "normalizing" a *CityJSON* file is nothing different than rearranging the order of its *CityObjects* in an alphabetical order.

The original dataset as well as the "normalized" dataset are visually and with the respect to the information the contain identical as expected. Their differences are in the order of the objects in the files and in the number of vertices stored in case the original file had duplicate vertices, and consequently the file size (see Table 6.1).

The tests that will be carried out in this chapter include initializing the repository of the VCS, creating the *multi-branch* structure suggested in Chapter 4 and testing its functionality with respect to the expected use of the workflow. They were conducted on a complete dataset i.e. a complete tile provided by the municipality in *CityGML* and converted into *CityJSON* encoding.

	B3_18 CityGML	*B3_18 CityJSON converted*	*B3_18 CityJSON exported*
# CityObjects		782	782
# vertices		89954	23623
file size	23.9 MB	4.3 MB	3.4 MB

Table 6.1: A comparison of statistics between the original *CityGML* file, the *CityJSON* file as converted from *CityGML* and the *CityJSON* exported from *Up3date* file using the original as input. The statistics for *CityJSON* files were calculated with *cjio*

6.2 INITIALIZE REPOSITORY AND CREATE THE *MULTI-BRANCH* STRUCTURE

To carry out all testing the repository has to be first initialized and the *multi-branch* structure to be created within it. In this case CJV is used as the implemented VCS and the system's platform is a *vCityJSON* file.

Using the *init commit* command a *vCityJSON* file is created (see Listing 6.2) with the initial normalized version of the dataset. The four branches of the *multi-branch* structure are created next, each one stemming accordingly, using CJV's *branch* command. For each version committed to it, the versioned file stores the version's metadata such as author, date, message etc.

```
cjv init commit B3_18_normalized.json main -a "Konstantinos Mastorakis" -m "
    Initial Commit" -o versioned.json
Opening B3_18_normalized.json...
Appending vertices...
Removing duplicate vertices...
Updating main to b9ca079909bf4654ca67f08cab9e8cae4545a749
Saving to versioned.json...
```

Listing 6.1: The initialization *init commit* command to create the versioned file and commit the initial instance to it.

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-19T19:52:43.562161Z",
                  "message": "Initial Commit",
                  "parents": [],
                  "objects":[...],
                  }},
      "tags":{},
"branches": {
    "main":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "maintenance":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "scenario":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749"
      }},
"vertices":[...]
}
```

Listing 6.2: The versioned file after committing the first instance of the 3DCM and creating the *multi-branch* structure. Notice that all branches have the same initial instance since it was the first instance committed to the *main* and all the rest branches stem directly or indirectly from it.

6.3 EXPORTING A SUBSET OF THE 3D CITY MODEL

The main concept behind conducting this test is the practicality of working in a subset of the file. Although it is theoretically possible to be always working directly on the full dataset it is way more convenient for the user to work only on their area of interest. It is also more efficient from a computing cost aspect, since the data size reduces dramatically. For instance, within the municipality of Rotterdam there are approximately a hundred thousand buildings.

What is also useful and worth mentioning is that *Blender* can export into various formats. In cooperation with *Up3date* a (subset of a) 3DCM could potentially be converted into many (if not all) the available formats supported by *Blender*. This would facilitate and motivate external applications in which a (very) small part of the 3DCM is needed in different 3D formats in every occasion. Testing of these format conversions is out of the scope of this thesis though.

As shown in Figure 4.2a, *CityObjects* from *ID_0599100000415249* to *ID_0599100000415266* are selected randomly for demonstration purposes and the rest are deleted from *Blender's* scene (see Figure 4.2b). The new scene is then exported into a *CityJSON* file with *Up3date* exporter (Listing 4.1).

At this point it should be mentioned that the current implementation of the VCS i.e. CJV does not have the ability to merge the changes of a subset back to the complete dataset. So this test only deals with testing the exporting functionality of *Up3date* when it comes to exporting subsets. More on that will be discussed in Chapter 7.

6.4 TESTING THE FUNDAMENTAL MAINTENANCE OPERATIONS

In this section the most prevalent cases that the user will come across with respect to the 3DCM maintenance are simulated. For example editing a building object's attributes or geometry. In theory, there is no limit in the combination of actions that can be executed as maintenance operations. Practically all combinations of actions can be broken down to simple actions which are simulated here.

As already mentioned in Chapter 3 there are three categories of buildings that need to be maintained. *New, mutated,* and *deleted*. All the tests carried out here wrap around these three categories.

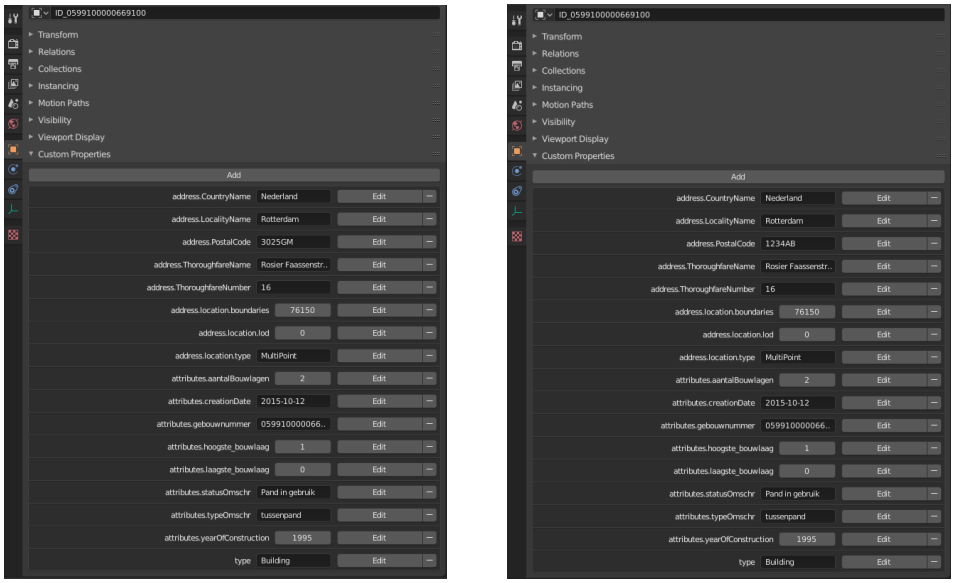
It is assumed that there is only one user working at a time, so that conflicts due to multiple users working on the same objects are diminished.

6.4.1 Visually editing attribute

Editing an attribute is the simplest action the user can perform on a 3DCM. Some of those attributes are arbitrarily assigned by external parties and are more prone to change (address of a building), while others are practically fixed (building's creation date). An implemented maintenance workflow can not be functional without addressing the necessity of editing attributes.

In this test a building's attribute is edited and then the new version of the 3DCM is exported with *Up3date* and fed into the CJV for versioning. More specifically the object *ID_0599100000669100* is randomly selected and its attribute *address.PostalCode* is changed from *3025GM* (Figure 6.1a) to *1234AB* (Figure 6.1b). This test is equivalent

to any other attribute being changed, since there is no difference in the mechanism, regardless how nested an attribute is. After editing the attribute a new instance is exported with *Up3date* which is committed to CJV.



(a) The object's *ID_0599100000669100* attributes before editing.

(b) The object's *ID_0599100000669100* attributes after editing

Figure 6.1: Editing the address.PostalCode attribute of the *ID_0599100000669100* object.

As shown in Listing 6.4 the commit is successful and updates the *maintenance* branch with a new instance. CJV recognizes the object whose attribute is edited and prints a status message. There is though no extra details regarding what part of the object was affected. In Listing 6.5 the updated versioned file is shown in which two versions are now present.

```
{
"ID_0599100000669100":{
        "geometry":[...],
        "address":{
                "CountryName":"Nederland",
                "LocalityName":"Rotterdam",
                "ThoroughfareNumber":"16",
                "ThoroughfareName":"Rosier Faassenstraat",
                "PostalCode":"1234AB",
                "location":{...}
        },
        "type":"Building",
        "attributes":{...}
        }
}
```

Listing 6.3: The new instance as exported from *Blender* into a *CityJSON* file after editing the *PostalCode* attribute.

```
cjv versioned.json commit B3_18_attrib_edit.json maintenance -a "Konstantinos
    Mastorakis" -m "ID_0599100000669100_postal_code_changed"
Opening versioned.json ...
Opening B3_18_attrib_edit.json ...
Appending vertices ...
Removing duplicate vertices ...

Changes:

        changed: ID_0599100000669100 (7af9636544ab267.. ->b5ccd65188e66b3..)

1144 objects not changed.
```

```
Updating maintenance to a493d4a48ff40151bfd7a57b49bb28b99975402b
Saving to versioned.json...
```

Listing 6.4: The commit of the object's *ID_0599100000669100* edited attribute instance in the versioned file.

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{...},
            "a493d4a48ff40151bfd7a57b49bb28b99975402b":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-20T13:09:57.609656Z",
                  "message": "ID_0599100000669100
                      postal code changed",
                  "parents": "b9ca079909bf4654ca67f08cab9e8cae4545
                      a749",
                  "objects":[...],
                  }},
      "tags":{},
"branches": {
    "main":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "maintenance":"a493d4a48ff40151bfd7a57b49bb28b99975402b",
    "scenario":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749"
      }},
"vertices":[...]
}
```

Listing 6.5: The new version of the edited attribute instance committed in the versioned file.

6.4.2 Visually editing geometry

Alongside attribute editing, editing the geometry of objects in a 3DCM is also a fundamental action. As buildings get altered or even higher accuracy raw data are collected over time, it is really useful to be able to update the existing objects with new geometric information.

Edit part of the building object's geometry

For this test object's *ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008* LoD-2 geometry is selected randomly. The face that represents the roof is deleted and the new 3DCM is exported. Although in practice deleting the roof of a building is not a common edit, it is better for visualization purposes and allows easier visual comparison, since a whole face is missing. Any alteration of one or more vertices of a face, instead of deleting a face is considered equivalent for the VCS.

Notice how the object's faces are reduced from 44 to 43 in Listing 6.6 and Listing 6.7. Again CJV identifies correctly which object is changed Listing 6.8, but as with attribute edit, there is no insight in what specifically is changed.

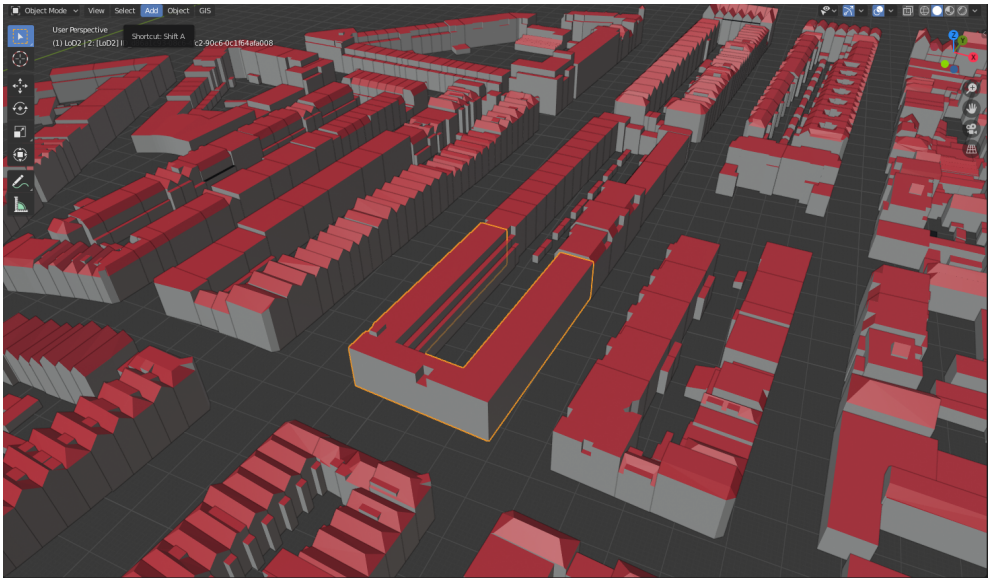


Figure 6.2: The object's *ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008* screenshot before deleting the roof face in *Blender*.

```
{
"boundaries":{...
                "36":[...]
                "37":[...]
                "38":[...]
                "39":[...]
                "40":[...]
                "41":[...]
                "42":[...]
                "43":[...]
                "44":[...]
},
        "semantics":{...},
        "textures":{},
        "lod":2
},
"type":"BuildingPart",
"parents":"ID_0599100000688769"
}
```

Listing 6.6: The object's *ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008* screenshot before deleting the roof face, in the *CityJSON* file.

```
{
"boundaries":{...
                "36":[...]
                "37":[...]
                "38":[...]
                "39":[...]
                "40":[...]
                "41":[...]
                "42":[...]
                "43":[...]
 },
        "semantics":{...},
        "textures":{},
        "lod":2
},
"type":"BuildingPart",
"parents":"ID_0599100000688769"
}
```

Listing 6.7: The object's *ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008* screenshot after deleting the roof face, in the *CityJSON* file.

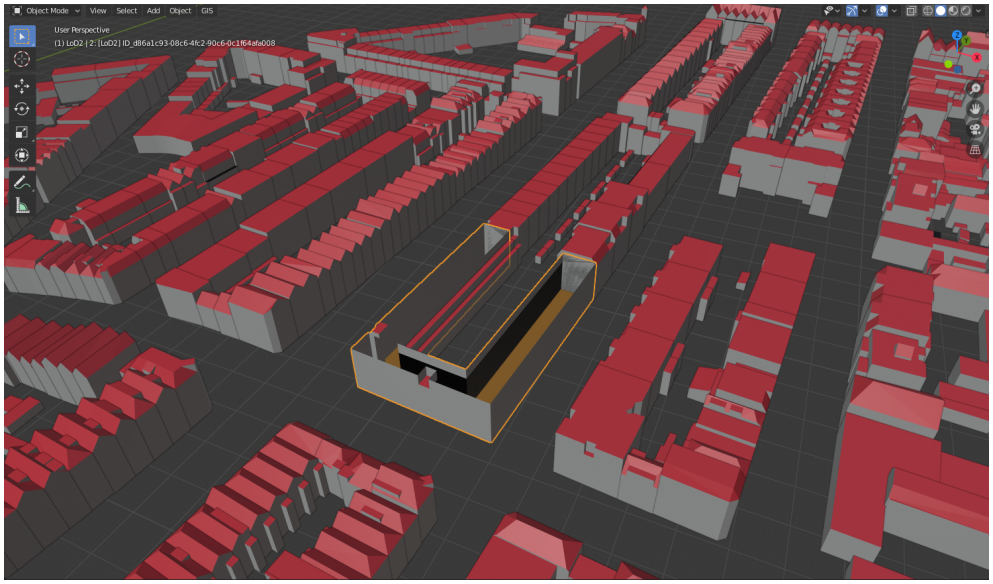


Figure 6.3: The object's *ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008* screenshot after deleting the roof face in *Blender*.

```
cjv versioned.json commit B3_18_geometry_edit.json maintenance -a "Konstantinos
    Mastorakis" -m "ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008 edit geometry"
Opening versioned.json...
Opening B3_18_geometry_edit.json...
Appending vertices...
Removing duplicate vertices...

Changes:

        changed: ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008 (39e1c319b1deb1a.. ->6e8a0c51235b7d6..)

1144 objects not changed.
Updating maintenance to 5c0a626ea58438ac24af8d3059ff9487f1d7d42d
Saving to versioned.json...
```

Listing 6.8: The commit of the edited geometry of object's *ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008* in the versioned file.

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{...},
            "a493d4a48ff40151bfd7a57b49bb28b99975402b":{...},
            "5c0a626ea58438ac24af8d3059ff9487f1d7d42d":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-20T14:08:16.169781Z",
                  "message": "ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa
                      008 edit geometry",
                  "parents": "a493d4a48ff40151bfd7a57b49bb28b99975
                      402b",
                  "objects":[...],
                  }},
      "tags":{},
"branches": {
    "main":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "maintenance":"5c0a626ea58438ac24af8d3059ff9487f1d7d42d",
    "scenario":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749"
```

```
        }},
  "vertices":[...]
}
```

Listing 6.9: The updated versioned file after committing the new instance.

Delete whole building object

Apart from deleting only parts of an object's geometry, in some cases the whole object has to be removed from the 3DCM because it is not longer existent, for example if it is demolished. Furthermore, in some cases it might be more efficient to remove an object and replace it with its newer version than updating the existing one. In both these cases the whole building object should deleted at its entirety.

For this test object *ID_848b57d6-528f-4c1d-ab65-27cbb29633d1* is chosen and completely eliminated from the 3DCM. The updated instance is then committed to the versioned file.

Notice that in Listing 6.12, CJV identifies two differences, although only one object is deleted. This happens because the *ID_848b57d6-528f-4c1d-ab65-27cbb29633d1* object is a child of the *ID_0599100000701839* object. So by eliminating the first CJV sees the parent object as changed as well, which is expected.

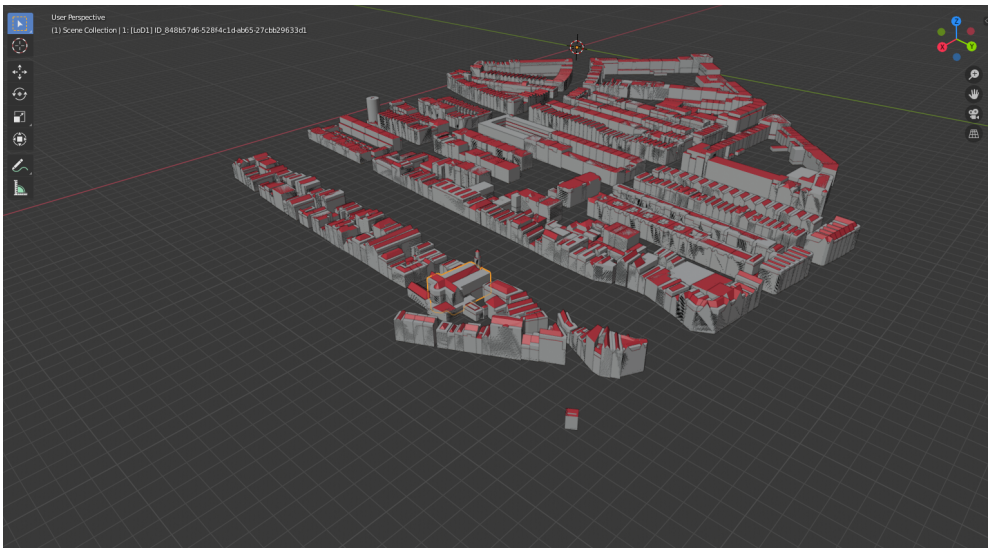


Figure 6.4: A screenshot of the 3DCM before deleting object *ID_848b57d6-528f-4c1d-ab65-27cbb29633d1* in *Blender*.

```
{
  "ID_848b57d6-528f-4c1d-ab65-27cbb29633d1":[...]
  "ID_855d2dc8-fc25-4a95-ac57-ad7af7aa0a5d":[...]
  "ID_870f7d50-a0ea-4d0f-a070-d5d804536ed9":[...]
  "ID_878ff256-afb7-44af-a863-e8ec6a0bf5bd":[...]
  "ID_892dacc2-4138-4de3-b36a-ad029ff2d79f":[...]
  "ID_8974a5bc-00a7-462c-8f77-41f3470cb4dc":[...]
  "ID_8ae195a2-55ac-4566-937a-184d8a28294d":[...]
  "ID_8b8150af-189b-4b6a-ab5c-8f46cc79266c":[...]
  "ID_8e2e4513-b726-4b2d-bf52-bd48a484c777":[...]
  "ID_8e5517ea-f145-4640-864d-a20f75514968":[...]
}
```

Listing 6.10: The object's ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 screenshot as part of the model before deleting it from the 3DCM in the *CityJSON* file.

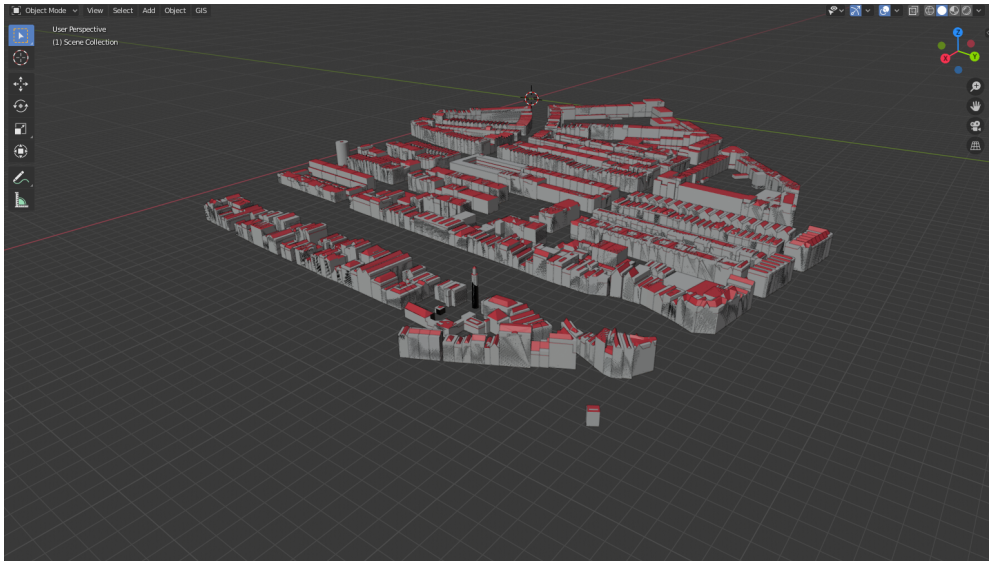


Figure 6.5: A screenshot of the 3DCM after deleting object *ID_848b57d6-528f-4c1d-ab65-27cbb29633d1* in *Blender*.

```
{
  "ID_855d2dc8-fc25-4a95-ac57-ad7af7aa0a5d": [...]
  "ID_870f7d50-a0ea-4d0f-a070-d5d804536ed9": [...]
  "ID_878ff256-afb7-44af-a863-e8ec6a0bf5bd": [...]
  "ID_892dacc2-4138-4de3-b36a-ad029ff2d79f": [...]
  "ID_8974a5bc-00a7-462c-8f77-41f3470cb4dc": [...]
  "ID_8ae195a2-55ac-4566-937a-184d8a28294d": [...]
  "ID_8b8150af-189b-4b6a-ab5c-8f46cc79266c": [...]
  "ID_8e2e4513-b726-4b2d-bf52-bd48a484c777": [...]
  "ID_8e5517ea-f145-4640-864d-a20f75514968": [...]
  "ID_8f69f6c7-f4ff-4c28-9f1c-b497c656fded": [...]
}
```

Listing 6.11: The object's *ID_848b57d6-528f-4c1d-ab65-27cbb29633d1* screenshot after deleting it from the 3DCM in the *CityJSON* file.

```
cjv versioned.json commit B3_18_object_deleted.json maintenance -a "Konstantinos
    Mastorakis" -m "ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 deleted"
Opening versioned.json...
Opening B3_18_object_deleted.json...
Appending vertices...
Removing duplicate vertices...

Changes:

        changed: ID_0599100000701839 (2f429e144117ec6.. ->87ece8e7c59117f..)
        deleted: ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 (e8c609003e7b250.. )

1143 objects not changed.
Updating maintenance to b927d455c7971f3cbba5f83988658646177eaa52
Saving to versioned.json...
```

Listing 6.12: The commit of the deleted object's ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 instance in the versioned file.

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{...},
```

```
                "a493d4a48ff40151bfd7a57b49bb28b99975402b":{...},
                "5c0a626ea58438ac24af8d3059ff9487f1d7d42d":{...},
                "b927d455c7971f3cbba5f83988658646177eaa52":{
                        "author": "Konstantinos Mastorakis",
                        "date": "2020-08-20T14:47:02.495869Z",
                        "message": "ID_848b57d6-528f-4c1d-ab65-27cbb2963
                            3d1  deleted",
                        "parents": "5c0a626ea58438ac24af8d3059ff9487f1d7
                            d42d",
                        "objects":[...],
                        }},
        "tags":{},
  "branches": {
      "main":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
      "maintenance":"b927d455c7971f3cbba5f83988658646177eaa52",
      "scenario":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
      "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749"
        }},
  "vertices":[...]
  }
```

Listing 6.13: The updated versioned file after committing the deleted object's *ID_848b57d6-528f-4c1d-ab65-27cbb29633d1* instance.

6.4.3 Adding a new building object

Adding a new object in a 3DCM via a GUI is a fundamental operation for the maintenance and update of a 3DCM. Either a new building is built and needs to be inserted in the model, or a new area is mapped, the need to add new objects is imperative.

With this test a new building addition into the 3DCM is simulated using *Up3date*. The updated model is committed into the versioned file afterwards.

Adding a building is slightly more complicated than deleting or editing one. That is due to specific parameters which have to be given by the user in order for the editing software to be aware what kind of object is inserted; which in turn affects the structure within the *CityJSON* file.

Within the context up *Up3date*, to insert a new —potentially multi-LoD— building the following steps have to be followed:

1. Every LoD/geometry has to be added as a mesh in the respective *Blender collection* —there is one collection for every LoD type which in the case of Rotterdam is 0,1,2— so three different collections.

   The mesh should be named in a predefined way for *Up3date* to parse it correctly. For example, a single LoD0 geometry should be added in the *LoD0* collection and named as **0**: [LoD**0**] ID_of_object preserving also the spaces. The zeroes in bold represent the LoD of the geometry and should be 1 for LoD1 and so on.

   For every mesh/geometry two more things needs to be added as *custom properties*: **type : the_surface_type** [1] and **lod : the_number_of_lod**

2. An empty object has to be created within the *main scene collection* with ID_of_object as a name. That object is an umbrella for the various LoD geometries.

   All the attributes are stored within this object as *custom properties*. In case the attributes are nested for example the *postal code* of an *address* then the at-

1 *Surface, MultiSurface, CompositeSurface or Solid* is accepted

tribute key should be *address.postalcode* so *Up3date* can understand the nested attribute structure and handle it accordingly.

3. If the semantics of the building's surfaces are known they can be assigned as materials to the respective faces for LoD2 or higher. The only information *Up3date* parses is the name of the material/semantics [2] so it is the only thing that the user needs to worry about.

For this test a new building object is inserted in the 3DCM named *CubicHouse* having only an LoD2 geometry for which semantics are also available. For simplicity purposes the geometry is considered to be an arbitrary cube. The updated model is the committed to the VCS.

Missing output from *Up3date* due to bug that I am currently working on. Will be there in final version

6.4.4 Updating *main* branch after the maintenance is completed

As long as the maintenance cycle is completed all the changes should be passed to *main* according to the suggested workflow. To make this happen within the VCS, *main* should be updated to point at the latest instance committed to *maintenance*. It is equivalent with what is called as *fast-forward merge* in *Git*.

Due to CJV being a prototype software there is not a *fast-forward merge* command implemented yet. To simulate the exact same functionality *main* is deleted and then created again to stem from the last instance of *maintenance*. This action produces exactly the same result as fast-forwarding *main* would.

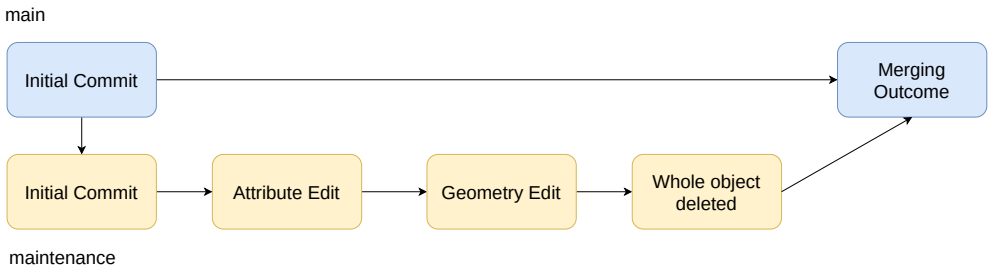


Figure 6.6: The *fast-forward merge* command. In practice, no actual merging has occurred within the VCS. It is just the update of *main* branch head to point at the last instance of *maintenance*.

```
Found 4 versions.

* version b927d455c7971f3cbba5f83988658646177eaa52 ( maintenance , main)
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T14:47:02.495869Z
  Message:

        ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 deleted

* version 5c0a626ea58438ac24af8d3059ff9487f1d7d42d
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T14:08:16.169781Z
  Message:

        ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008 edit geometry

* version a493d4a48ff40151bfd7a57b49bb28b99975402b
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T13:09:57.609656Z
  Message:

        ID_0599100000669100_postal_code_changed
```

2 The types of building surface semantics are *RoofSurface, WallSurface, GroundSurface, ClosureSurface etc.*

```
* version b9ca079909bf4654ca67f08cab9e8cae4545a749 (scenario, release)
  Author: Konstantinos Mastorakis
  Date: 2020-08-19T19:52:43.562161Z
  Message:

      Initial Commit
```

Listing 6.14: The *log* command for *main* after simulating the *fast-forward merge*.

6.5 SIMULATING THE CREATION AND ADOPTION OF NEW SCENARIOS

As introduced already, there is a dedicated branch (*scenario*) for testing every new scenario or idea. In terms of editing the 3DCM to implement the new scenario, the process is a combination of the fundamental maintenance operations (see Section 6.4).

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{...},
            "a493d4a48ff40151bfd7a57b49bb28b99975402b":{...},
            "5c0a626ea58438ac24af8d3059ff9487f1d7d42d":{...},
            "b927d455c7971f3cbba5f83988658646177eaa52":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-20T14:47:02.495869Z",
                  "message": "ID_848b57d6-528f-4c1d-ab65-27cbb2963
                      3d1  deleted",
                  "parents": "5c0a626ea58438ac24af8d3059ff9487f1d7
                      d42d",
                  "objects":[...],
                  }},
      "tags":{},
"branches": {
    "main":"b927d455c7971f3cbba5f83988658646177eaa52",
    "maintenance":"b927d455c7971f3cbba5f83988658646177eaa52",
    "scenario":"b927d455c7971f3cbba5f83988658646177eaa52",
    "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "scenario 1":"b927d455c7971f3cbba5f83988658646177eaa52"
      }},
"vertices":[...]
}
```

Listing 6.15: The two branches that the individual versions will be committed at.

The major differentiation between *maintenance* and *scenario* is that within *scenario* the changes are carried out by multiple users who work concurrently and are different from the changes carried out for *maintenance* purposes, so they have to be merged together with them before passing to *main*, if the scenario gets realized.

This translates into potential conflicts, first between the concurrent users within *scenario* and second upon merging *scenario* into *maintenance*. For demonstration purposes a merging operation with no conflicts will be shown in this example in which two different sets of changes are first integrated into *scenario* and then passed into *maintenance*.
For this test two sets of changes will be committed into *scenario*, simulating two users working on different objects of the model implementing the scenario for visu-

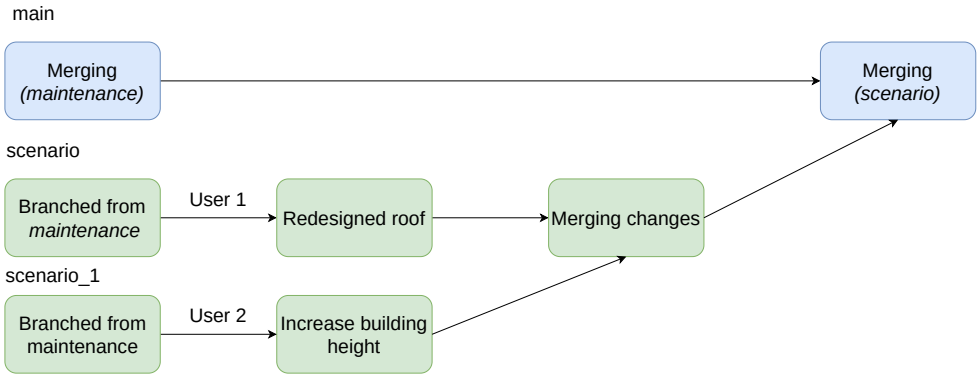
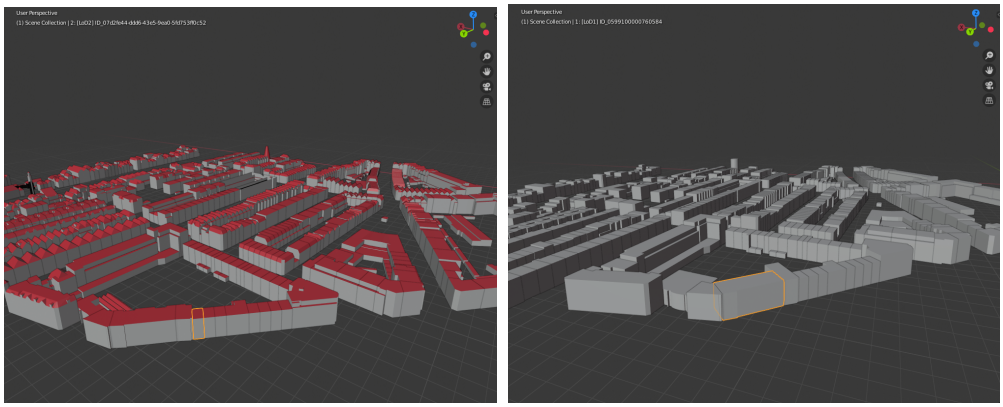


Figure 6.7: A schematic representation of the test in a sequence of commits. The merging commits in *main* or in reality fast-forward merges.

alization and testing purposes. In practice though since there is no remote repository for the individual users to push their work a workaround is needed. To simulate this functionality two branches are created *scenario* and *scenario_1* each one simulating each individual user and pointing to the last instance of *maintenance*. (Listing 6.15).



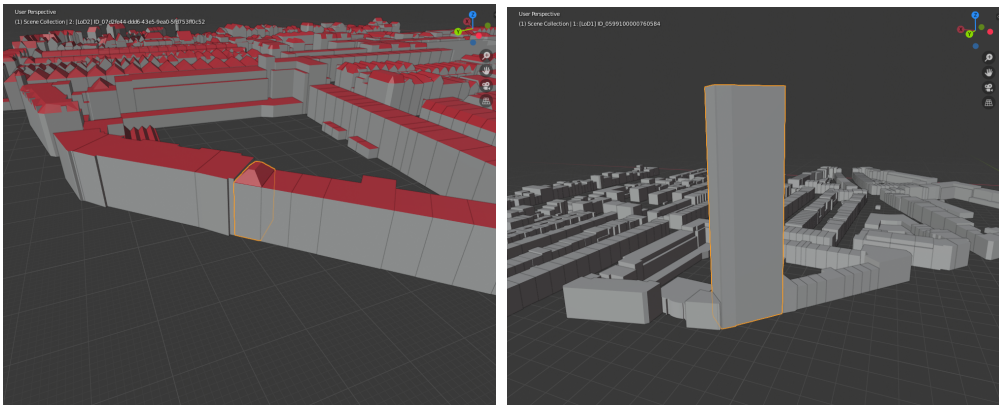
(a) Building's ID_07d2fe44-ddd6-43e5-9ea0-5fd753ff0c52 LoD 2 geometry whose roof will be reconstructed for hosting solar panels

(b) Building's ID_05991000000760584 LoD 1 geometry whose upwards extension needs to be tested

Figure 6.8: The two buildings that have to be edited for the implementation of the new scenario

Each set of changes will be carried out in these branches respectively; then the branches will be merged together into one (*scenarion* in this case), which is the equivalent of pushing the two local branches into the remote *scenario* branch. These changes can be then passed to *maintenance* simulating the adoption and the realization of the scenario. To do the *merge* command is used. The difference between a realized and a non-realized scenario is that there is no update of *maintenance* to match *scenario*.

When using *merge*, the direction of the merging should be chosen. This affects the branch in which the outcome of *merge* will be committed. Similar to *Git*, the outcome of *merge* is identical regardless the merging direction (*scenario* to *maintenance* or vice versa). In this case the direction of the merge is from *scenario* to *maintenance*, since integrating the changes into *maintenance* is the desired outcome.



(a) Building's ID_07d2fe44-ddd6-43e5-9ea0-5fd753ff0c52 LoD 2 geometry whose roof shape is redesigned

(b) Building's ID_0599100000760584 LoD 1 geometry after extending its height

Figure 6.9: The two building geometries after being edited implementing the new scenario

6.5.1 Scenario explanation

The concept of the scenario whose approval and realization will be simulated is the following: The shape of a building's roof needs to be decided to maximize its solar capacity depending also on a building nearby that will be extended upwards. The exact shape of the roof in conjunction with the effect of the increased height of the neighboring building should be carried out to the updated model(s).


```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{...},
            "a493d4a48ff40151bfd7a57b49bb28b99975402b":{...},
            "5c0a626ea58438ac24af8d3059ff9487f1d7d42d":{...},
            "b927d455c7971f3cbba5f83988658646177eaa52":{...},
            "c5814a2b2529c01d592eaacedbab3d436286fb7b":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-26T16:21:06.232885Z",
                  "message": "ID_0599100000760584_lod_1_geometry
                      _extruded",
                  "parents": "b927d455c7971f3cbba5f83988658646177
                      eaa52",
                  "objects":[...],
                  },
            "e52c328ce44fc3e0a39cd9af7245cbb6d0476609":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-26T16:56:46.839261Z",
                  "message": "ID_0599100000760585_lod_2_roof
                      _redesigned",
                  "parents": "b927d455c7971f3cbba5f83988658646177
                      eaa52",
                  "objects":[...],
                  }},
      "tags":{},
"branches": {
    "main":"b927d455c7971f3cbba5f83988658646177eaa52",
    "maintenance":"b927d455c7971f3cbba5f83988658646177eaa52",
```

```
      "scenario":"e52c328ce44fc3e0a39cd9af7245cbb6d0476609",
      "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
      "scenario_1":"c5814a2b2529c01d592eaacedbab3d436286fb7b"
        }},
  "vertices":[...]
  }
```

Listing 6.16: The *vCityJSON* file after committing both instances at their respective branches.

In this hypothetical scenario one user is responsible to implement the shape of the roof of the building (Figure 6.8a) and the second to implement the upwards extension of the second building (Figure 6.8b) so that the solar capacity analysis can be performed. The two users will commit their respective instances (Figure 6.9a, Figure 6.9b) to the individual branches with then will be merged into one.

More specifically the instance depicted in Figure 6.9a will be committed at *scenario* and the instance depicted in Figure 6.9b in *scenario_1*. Next, branch *scenario_1* will be merged into *scenario* (Listing 6.17). The log output for *scenario* is shown in Listing 6.18. Finally, *main* will be *fast-forwarded* to *scenario* Listing 6.19 simulating that the implemented in the 3DCM changes will be realized.For better understanding see Figure 6.7 which is a schematic representation of the merging mechanics of the whole test.

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "b9ca079909bf4654ca67f08cab9e8cae4545a749":{...},
            "a493d4a48ff40151bfd7a57b49bb28b99975402b":{...},
            "5c0a626ea58438ac24af8d3059ff9487f1d7d42d":{...},
            "b927d455c7971f3cbba5f83988658646177eaa52":{...},
            "c5814a2b2529c01d592eaacedbab3d436286fb7b":{...},

            "e52c328ce44fc3e0a39cd9af7245cbb6d0476609":{...}
            "a675e34ac692c0418a82420bb706987e5df3ec47":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-26T16:59:23.047245Z",
                  "message": "Merge scenario_1 to scenario",
                  "parents": {"c5814a2b2529c01d592eaacedbab3d43628
                      6fb7b", "e52c328ce44fc3e0a39cd9af7245cbb6d04
                      76609"},
                  "objects":[...],
                  },},
      "tags":{},
"branches": {
    "main":"b927d455c7971f3cbba5f83988658646177eaa52",
    "maintenance":"b927d455c7971f3cbba5f83988658646177eaa52",
    "scenario":"a675e34ac692c0418a82420bb706987e5df3ec47",
    "release":"b9ca079909bf4654ca67f08cab9e8cae4545a749",
    "scenario_1":"c5814a2b2529c01d592eaacedbab3d436286fb7b"
      }},
"vertices":[...]
}
```

Listing 6.17: The *vCityJSON* file after merging *scenario_1* to *scenario*.

```
Found 7 versions.

* version a675e34ac692c0418a82420bb706987e5df3ec47 (scenario)
  Author: Konstantinos Mastorakis
  Date: 2020-08-26T16:59:23.047245Z
  Message:

      Merge scenario_1 to scenario
```

```
* version c5814a2b2529c01d592eaacedbab3d436286fb7b (scenario_1)
  Author: Konstantinos Mastorakis
  Date: 2020-08-26T16:21:06.232885Z
  Message:

        ID_0599100000760584_lod_1_geometry_extruded

* version e52c328ce44fc3e0a39cd9af7245cbb6d0476609
  Author: Konstantinos Mastorakis
  Date: 2020-08-26T16:56:46.839261Z
  Message:

        ID_0599100000760585_lod_2_roof_redesigned

* version b927d455c7971f3cbba5f83988658646177eaa52 ( maintenance , main)
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T14:47:02.495869Z
  Message:

        ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 deleted

* version 5c0a626ea58438ac24af8d3059ff9487f1d7d42d
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T14:08:16.169781Z
  Message:

        ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008 edit geometry

* version a493d4a48ff40151bfd7a57b49bb28b99975402b
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T13:09:57.609656Z
  Message:

        ID_0599100000669100_postal_code_changed

* version b9ca079909bf4654ca67f08cab9e8cae4545a749 (release )
  Author: Konstantinos Mastorakis
  Date: 2020-08-19T19:52:43.562161Z
  Message:

        Initial Commit
```

Listing 6.18: The *log* command output for *scenario* after merging *scenario_1* into it

```
Found 7 versions.

* version a675e34ac692c0418a82420bb706987e5df3ec47 (scenario, main)
  Author: Konstantinos Mastorakis
  Date: 2020-08-26T16:59:23.047245Z
  Message:

        Merge scenario_1 to scenario

* version c5814a2b2529c01d592eaacedbab3d436286fb7b (scenario_1)
  Author: Konstantinos Mastorakis
  Date: 2020-08-26T16:21:06.232885Z
  Message:

        ID_0599100000760584_lod_1_geometry_extruded

* version e52c328ce44fc3e0a39cd9af7245cbb6d0476609
  Author: Konstantinos Mastorakis
  Date: 2020-08-26T16:56:46.839261Z
  Message:

        ID_0599100000760585_lod_2_roof_redesigned

* version b927d455c7971f3cbba5f839886586461 77eaa52 (maintenance)
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T14:47:02.495869Z
  Message:

        ID_848b57d6-528f-4c1d-ab65-27cbb29633d1 deleted

* version 5c0a626ea58438ac24af8d3059ff9487f1d7d42d
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T14:08:16.169781Z
  Message:

        ID_d86a1c93-08c6-4fc2-90c6-0c1f64afa008 edit geometry

* version a493d4a48ff40151bfd7a57b49bb28b99975402b
  Author: Konstantinos Mastorakis
  Date: 2020-08-20T13:09:57.609656Z
  Message:

        ID_0599100000669100_postal_code_changed

* version b9ca079909bf4654ca67f08cab9e8cae4545a749 (release)
  Author: Konstantinos Mastorakis
  Date: 2020-08-19T19:52:43.562161Z
  Message:

        Initial Commit
```

Listing 6.19: The *log* command output for *main* after merging *scenario* into it

6.6 TESTING FOR CONFLICTS

This section aims at understanding when the VCS considers an object changed which potentially leads to merging conflicts. By altering the information included in the 3DCM in various ways, the performance of the VCS can be tested in practice and more insight can be gained that can help improving the later.

The conflicts created can be divided into two categories. False conflicts, these happening when the smallest entity issue introduced in Chapter 4 and true or expected conflicts that happen because the exact same piece of information is changed in two different ways prior to merging.

These tests will be carried out on a new versioned file for simplicity purposes. For consistency purposes the initial file in the new versioned file will be the same as before.

6.6.1 Mingle order of attributes

This test aims in understanding if by changing the order of two *CityObject's* attributes in the *CityJSON* file itself, the *CityObject* is understood as altered by the VCS.

If this happens it means that editing the 3DCM instance file with a software other than *Up3date* for example, which potentially mingles the order of attributes, would make the object(s) appear as changed upon committing to the respective branch and yield a conflict if the same object has been edited differently by another user in the meantime.

```
{
"ID_0599100000012851":{
 "geometry":[...],
 "address":{...},
 "type":"Building",
 "attributes":{
        "yearOfConstruction":1910,
        "creationDate":"2017-08-15",
        "gebouwnummer":"0599100000012851",
        "hoogste_bouwlaag":2,
        "statusOmschr":"Pand in gebruik",
        "aantalBouwlagen":4,
        "laagste_bouwlaag":1,
        "typeOmschr":"tussenpand"
       }}
}
```

Listing (6.20) The original attribute order of *CityObject* ID_0599100000012851.

```
{
"ID_0599100000012851":{
  "geometry":[...],
  "address":{...},
  "type":"Building",
  "attributes":{
    "creationDate":"2017-08-15",
    "yearOfConstruction":1910,
    "gebouwnummer":"0599100000012851",
    "hoogste_bouwlaag":2,
    "statusOmschr":"Pand in gebruik",
    "aantalBouwlagen":4,
    "laagste_bouwlaag":1,
    "typeOmschr":"tussenpand"
  }}
}
```

Listing (6.21) The changed attribute order of CityObject ID_0599100000012851.

```
cjv versioned_conflicts.json commit B3_18_normalized_reorder_attribute.json
    master -a "Konstantinos Mastorakis" -m "
    ID_0599100000012851_attributes_swapped"
Opening versioned_conflicts.json...
Opening B3_18_normalized_reorder_attribute.json...
Appending vertices...
Removing duplicate vertices...

Changes:

        changed: ID_0599100000012851 (d6ade9370ad2e5c.. ->d200c4bb3f915ef..)

Updating master to 74e5d00d9e5c6dd654d786b2908ffd2294032f88
Saving to versioned_conflicts.json...
```

Listing 6.22: The commit of the swapped attributes' instance of object's ID_0599100000012851. CJV perceives the object as changed.

6.6.2 Mingle order of faces

This case is the exact same test as the previous one but instead of attributes the order of the faces of an object's geometry is swapped. Knowing that CJV uses hashes it is

```
{
"ID_0599100000012859":{
  "geometry":{"type": "Solid",
        "boundaries": [
          [
            [
              4312,
              4313,
              4314,
              4315,
              4316,
              4317,
              4318,
              4319,
              4320,
              4321,
              58,
              4322,
              4323
            ]
          ],
          [
            [
              14927,
              14928,
              14929,
              14930,
              14931,
              14932
            ]
          ]
  }}
}
```


Listing (6.23) The original face order of CityObject ID_0599100000012859.


```
{
"ID_0599100000012859":{
  "geometry":{"type": "Solid",
        "boundaries": [
          [
            [
              14927,
              14928,
              14929,
              14930,
              14931,
              14932
            ]
          ],
          [
            [
              4312,
              4313,
              4314,
              4315,
              4316,
              4317,
              4318,
              4319,
              4320,
              4321,
              58,
              4322,
              4323
            ]
          ]
  }}
}
```


Listing (6.24) The changed face order of CityObject ID_0599100000012859.

Figure 6.11: Swapping the first and second face of the LoD 2 geometry of CityObject ID_0599100000012859.

expected that the outcome will be the same as before, meaning that the VCS will see the object as changed. For demonstration purposes though the test is carried out.

```
cjv versioned_conflicts.json commit B3_18_normalized_reorder_face.json  master
      -a "Konstantinos Mastorakis" -m "
     ID_0599100000012859_lod_2_faces_swapped"
Opening versioned_conflicts.json...
Opening B3_18_normalized_reorder_face.json...
Appending vertices...
Removing duplicate vertices...

Changes:

        changed: ID_0599100000012859 (d278de805af79dd.. ->c33e4caba5ad000..)

Updating master to 824798c78cbe2a4ad3ed912d19f1a0211b42e885
Saving to versioned_conflicts.json...
```

Listing 6.25: The commit of the swapped faces of object's ID_0599100000012859 LoD 2 geometry faces version. CJV perceives the object changed.

6.6.3 Edit different piece of information within the same object in (false conflict)

In this case a different sub-entity of the object is changed in two branches and then a merge attempt is executed to check the outcome.

```
{
"type": "CityJSON",
"version": "1.0",
"extensions": {},
"metadata": {...},
"CityObjects": {...},
"versioning": {
      "versions": {
            "8bf0de99a8ad1fb9fb3d8927c34cda7a082a5dfd":{...},
            "74e5d00d9e5c6dd654d786b2908ffd2294032f88":{...},
            "824798c78cbe2a4ad3ed912d19f1a0211b42e885":{...},
            "2217ff724ce1e07d96765b240fa28191e5e45fb8":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-28T12:39:11.061604Z",
                  "message": "ID_0599100000012851_postal_code
                      _changed",
                                                          "
                      parents":"824798c78cbe2a4ad3ed912d19f1a0211b
                      42e885",
                  "objects":[...]
            },
            "68258e642350d97d9bd48e9ca7266409ab28c2de":{
                  "author": "Konstantinos Mastorakis",
                  "date": "2020-08-28T12:40:03.307291Z",
                  "message": "ID_0599100000012851_number_changed",
                  "parents": "824798c78cbe2a4ad3ed912d19f1a0211b42
                      e885",
                  "objects":[...]
                  },},
      "tags":{},
"branches": {
    "master":"2217ff724ce1e07d96765b240fa28191e5e45fb8",
    "testing":"68258e642350d97d9bd48e9ca7266409ab28c2de",
      }},
"vertices":[...]
}
```

Listing 6.26: The versioned file after committing both instances into their respective branches.

More specifically a second branch is created in the versioned file and then two new instance are committed; one into each branch respectively. In the first instance object's *ID_0599100000012851 PostalCode* is changed from *3025TW* to *9876CD*. In the second, attribute *ThoroughfareNumber* is changed from *126* to *150* (leaving the postal code unchanged to the original value *3025TW*). Then, both instances are committed to their respective branches (see Listing 6.26). Finally, *merge* command is used to test the behavior of the VCS. As shown in Listing 6.27 the VCS can not merge the two instances raising a conflict.

```
cjv versioned_conflicts.json merge testing master -a "Konstantinos Mastorakis"
Opening versioned_conflicts.json...
Common ancestor: 824798c78cbe2a4ad3ed912d19f1a0211b42e885
There are conflicts!
- ID_0599100000012851
Forgive me for not being able to resolve them right now...
```

Listing 6.27: CJV raising a conflict upon merging the two branches.

After carrying out the three conflict-related experiments presented so far, it is safe to conclude that any change within the records of the 3DCM or their order will be perceived as a changed object from CJV.

Concluding, it is of course expected a conflict to be raised when editing the same record within the same object in two different instances and attempting to merge them. That is the desired behavior of a VCS.

The problem is that CJV will also raise a conflict in case different records within the same object are edited which is not optimal since the user has to manually create a new instance and integrate these two changes together.

7 DISCUSSION

3DCMs and their manipulation is an emerging domain that has been growing very fast recently. That is due to technologies that allow massive spatial data collection becoming more accessible and computers' hardware becoming more powerful and capable of handling it. In addition, 3DCMs become more and more popular due to the intuitiveness of dealing with visualized 3D information plus the fact that all the information is integrated into a single dataset.

3DCMs have to catch up with reality in order for them to be as useful as possible i.e. they need to be maintained regularly. This thesis suggested a prototype workflow for the successful and efficient maintenance of *CityJSON* encoded 3DCMs. This workflow includes the use of two prototype software components: One for editing a 3DCM through a GUI and one for implementing a VCS.

Originating from software development, the use of a VCS capable to handle 3DCM files is considered the most appropriate solution for effective maintenance. That is because VCS have the ability to keep track of all changes automatically while they guarantee data integrity. They also allow concurrent maintaining in an efficient way, by identifying potential information mismatch (i.e.conflicts). These characteristics make a VCS the most important component of a 3DCM maintenance workflow.

7.1 CONCLUSIONS

A major change that the workflow introduces is the regularity of maintenance. With respect to 3DCMs, maintenance can be considered as a cornerstone. It is the —conceptual— foundation for promoting integration of the suggested workflow into the existing workflows of public organizations or private companies who work with this kind of information. Not only it will grant them full control over the 3DCM and its evolution, but it will also save them considerable financial resources currently used for outsourcing in some cases.

Being prototype means that the workflow is far from becoming operational. Arguably, one of the most important conclusions which affects the whole behavior of the VCS has to do with the smallest entity issue as it was described in Chapter 4. In other words, the hierarchical level within a 3DCM file at which the VCS can interpret a change is crucial for an efficient and smart solution.

That being said, what is the smallest entity has to be defined in advance, since one could argue that it could be the attribute's level, someone else the index of the vertices within the faces of each geometry and a third party the vertices coordinates themselves (although they are kept outside the *CityObjects* in the case of *CityJSON*).

For comparison, in software development oriented VCSs that level is every simple character in a text file (i.e. the smallest entity possible). Doing the same is not as trivial with 3DCMs oriented VCS because there are many different types of information (descriptive, geometrical, semantic, numerical, text etc) which is nested within each other as well.

As shown during testing, having an object higher into the *CityJSON's* hierarchy as the smallest entity is simpler from a developing point of view. However it raises conflicts that from a human perspective should not be raised (i.e. false conflicts), therefore increasing the workload for the maintainer.

Smaller entity of storage vs smaller atity of comparison?

On the other hand going deeper into the hierarchy means that the VCS will become smarter in the sense of understanding and interpreting what part of information is changed within a *CityObject*; reducing or even ideally eliminating the raise of false conflicts. However, from a developing point of view it would increase the complexity of the VCS software.

For this automation to be meaningful the developing of the VCS functionality to match this performance should be flawless, which is not easily achievable in the context of 3DCM data structures. Although automating procedures is always a tempting path to follow with big data, in the 3DCMs domain —and with their maintenance as well— it looks like human supervision will always be necessary to some extent. That is due to the nature and versatility of the data.

Ideally, the "golden ratio" should be achieved between a smart VCS yet leaving space for (at least) some human supervision. For that "golden ratio" to be defined, further research is needed with trial and error to be probably the best approach.

7.1.1 To what extent can a *Git*-based versioning approach be used for the maintenance of the 3D city model of a typical municipality?

The main goal of the thesis was to investigate to what extent a *Git*-based versioning approach is suitable for the maintenance of the 3DCM of a typical municipality. After designing, implementing and testing such a workflow, perhaps the most important generic conclusion is that a*Git*-based VCS with visual editing capabilities is a very promising combination for tackling the task of maintaining a 3DCM updated.

What this thesis has shown, is that maintaining 3DCMs can become a relatively simple task —compared to what is now— that does not require the maintainers to be experts with 3DCM data models, encodings and specifications.

The workflow and specifically its versioning component was customized according to the needs of the municipality of Rotterdam, in order to keep it simple yet manageable and practical.

A quite positive surprise was the realization that some of the key points for the effective maintenance for the municipality of Rotterdam are coinciding —at a conceptual level— with some of *Git's* functionality. More specifically, key point 3,4 and 5 from Table 3.2 are really close to what *Git* operations *branch*, *fork* and *pull request* respectively.

It has to be mentioned here that upon the identification and formulation of all five key points in Table 3.2, *Git's*functionality was not taken into account at all. This fact confirms that a *Git*-based versioning approach for maintaining a 3DCM of a typical municipality; assuming that those key points will remain more or less the same for other municipalities as well.

Regarding the visual editing capabilities of the workflow another useful realization was that these capabilities might be more necessary than initially thought. Developing the visual editing component of the workflow started as a proof of concept. However, after using it for generating the new instances for the versioning compo-

nent it became clear that some geometrical edits are almost impossible without a visual editor. For example changing the shape of a roof (see Figure 6.8a, Figure 6.9a).

In that sense, what was developed to improve the interface via which the user can edit a 3DCM, turned out to have more capabilities than other 3DCM editing software. Last but not least, having the capability to edit a 3DCM via a GUI drastically improves the user experience with respect to updating and editing a 3DCM, while it provides with intuitive inspection of the edits in real time.

7.2 PRACTICAL COMPARISON WITH OTHER POTENTIAL SOLUTIONS

Before carrying any comparison with other potential solutions it has to be clarified that currently there are no other implemented solutions that allow versioning of 3DCMs. What is more, discussion about versioning itself has little to no meaning in an abstract context. In order for a comparison to be meaningful it needs to compare two implementations of versioning; since in a conceptual and data model level everything is more or less an abstraction with respect to versioning.

With this in mind, the following comparisons are carried out with a speculation of what a versioning implementation of the alternative data models would be based on the respective data models. In other words, until a solid implementation of versioning is available for the alternative data models, no real comparison can exist.

7.2.1 CityGML v.3.0

The most obvious comparison would be with an implemented workflow that would wrap around the new *CityGML v.3.0* data model and encoding. In my opinion, both the data model of *CityGML v.3.0* and the Geography Markup Language (GML) encoding have some specific characteristics that make *CityGML v.3.0* inferior to *CityJSON* for the purpose of versioning.

At the moment, there is no software available that implements *CityGML v.3.0* versioning module in practice. Someone could argue at this point that *CityGML v.3.0* is not yet released officially for that to happen. However, practice has shown that the GML encoding imposes many difficulties when it comes to developing software for handling such files. That is the case with *CityGML v.2.0* as well that limited software can support that encoding. It is certain that these difficulties will continue to exist also with *CityGML v.3.0*.

In the case the aforementioned difficulties were to be surpassed, there is a second point —in my opinion even more important— because it is encoding independent and has to do with the data model of *CityGML v.3.0* itself. The *CityGML v.3.0* versioning module inherently creates redundancy of information which for versioning solutions can complicates things a lot and create break points.

According to the *CityGML v.3.0* versioning data module apart from the different versions of the objects, the transition record between them is also kept within the same structure. Obviously, these two need to be always "synced", otherwise great amount of confusion will arise. Keeping them synced from an implementing point of view adds a lot of complexity and potentially compromises the robustness of the whole versioning platform. The amount of potential break points is increased every time a new version of an object is created.

What is even more limiting is the fact that the transaction types between two versions of an object are prescribed as well by the data model. What this means is that the user is not free to assign the type of transaction they wish but to choose from predefined ones. In the case a new type of transaction emerges it is not possible to record that accurately and it would require the data model to be revised or an extension to be created, which adds unnecessary complexity. In comparison, the *CityJSON* data structure for versioning does not require any record of such transition.

A minor point but worth mentioning is the lack of prediction of a mechanism for handling the versions within the *CityGML* file from the data model. Although this mechanism could theoretically be implemented in the future, users must be editing directly on the *CityGML* file until it is in place, which inevitably leads to higher error and breaking the file probability, especially with a complex data model such as the *CityGML v.3.0* versioning module.

Summarizing, *CityGML v.3.0* is considered inferior to *CityJSON* for a versioning implementation mainly due to the redundancy of information it introduces followed by an already notorious encoding format among the software developing community. Last but not least, it is unnecessarily strict when it comes to what can be the types of transactions with no tangible benefit.

7.2.2 3DCityDB

According to the official website, there is no support for versioning within the *3dCityDB* which currently supports the *CityGML v.1.0* and *v.2.0* formats. Most probably *CityGML v.3.0* will be supported in the future, but for now there is nothing officially stated about that in their website.

If *CityGML v.3.0* gets supported by *3DCityDB* implementing versioning capabilities as *CityGML v.3.0* data model dictates would also mean that the characteristics of the latter mentioned in Section 7.2.1 are inherited to *3DCityDB* as well. In my opinion, using a versioning mechanism besides what *CityGML v.3.0* data model suggests, that would also take advantage of the inherent benefits of a database when it comes to versioning, might lead to a better solution.

In the case that *3DCityDB* gets to support the *CityJSON* data model and format, will be a big improvement from a software development point of view. That makes the development of a (*Git*-based) versioning solution more feasible, if the *CityGML v.3.0* data model proves to be not the suitable approach for versioning in practice.

Since it is not viable for the 3DCM of a municipality to be kept in raw 3DCM files, making the use of database is perhaps the best way to follow. Eventually, 3DCM versioning software will be implemented and accepted, it is just not there yet.

7.3 MINGLING THE ORDER OF FACES

As seen in Section 6.6.2 changing the order of the faces within a geometry of any object, makes the VCS see the whole object as changed. This means that many false conflicts will be yielded due to this. In case of using different software to edit the same file, the order of its contents is very likely to change, creating a lot of unnecessary conflicts when integrating the changes back to the VCS and creating potential data corruption or loss if the conflicts are not handled properly . That is why the dataset was ”normalized” before doing any tests on it in Chapter 6 so the results

would be valid.

One way to avoid that, would be to order the faces of every geometry in a universal way prior to committing to the VCS, so it can detect what is actually changed. An example of such ordering is the *lexicographical order* [1]. An approach like that is very likely to address this challenge effectively.

Regardless that, a question of more or less philosophical nature arises at this point: *If the order of the faces/vertices of a geometry/face is changed, can the geometry/face be considered the same?*

Being philosophical, there is no right or wrong answer to this question from a human perspective. For the versioning component of the workflow though, the answer to the question at the moment is negative. From a developing point of view that answer makes things simpler, but on a practical level it will certainly create lots of false conflicts. This translates into considerably more workload and increased error possibility.

For a more efficient and robust system it looks like the answer to that question with respect to the versioning component should be positive. That way the robustness of the VCS will increase, allowing a facilitating the use of different editing software and workflows.

If the answer for the VCS remains as is, it will be up to the practitioner to interpret if there is an actual change to the *CityObjects*. This will be rather exhaustive procedure especially when there are many *CityObjects*. What is more, it will compromise the history tracking mechanism of the VCS since it will present false changes that have never taken place.

The exact same applies for mingling the order of attributes (see Section 6.6.1) with the only difference that tackling this issue is more straightforward. That is due to descriptive data having simpler data structures and *key* field within the dictionary is text rather than an identification number for indexing. A simple alphabetical ordering of the attributes would solve the problem effectively.

7.4 WHAT CAN ROTTERDAM EXPECT IN PRACTICE: CHALLENGES AND IMPROVEMENTS

As already mentioned in Chapter 3, currently the municipality of Rotterdam outsources the task of updating their 3DCM once every two years. Adopting the suggested workflow —with the necessary customization to fit their existing workflows— will enable the municipality to —potentially— manage the 3DCM completely on its own in the future.

However, there is no change that comes at no cost. Such a transition is certain that will impose plenty of challenges, especially since there is no pre-existing workflow due to outsourcing. The challenges are created first due to the prototype nature of the workflow and second to the required resources the municipality would need to invest, in order for the practitioners to familiarize, understand and use the workflow effectively.

The more impeding challenges are expected to be those that are related to the prototype nature of the workflow. Even if the practitioners were sufficiently trained and

1 https://en.wikipedia.org/wiki/Lexicographic_order

familiar with the workflow they would not be able to overcome these challenges, since they are inherent to the workflow.

7.4.1 Tile versioning vs Full Model versioning

One of the biggest challenges is that the versioning workflow works with a single dataset i.e. 3DCM. The 3DCM of the municipality is split in tiles for better administration. A single file for the whole city of Rotterdam would be almost impossible to be (visually) processed due to limitations of *Blender* to handle so much information.

There are two alternatives to overcome this challenge:

a) *Every tile is versioned separately and all the latest versions are combined to compose the full model:*

   In this case every tile has its own versioning structure (i.e. VCS) and the latest instance of each VCS is combined to create the whole 3DCM. With this approach keeping track of the history of the changes of the 3DCM as a whole becomes problematic due to many different VCSs storing "their own" history. In addition, working on an area that belongs in two or more tiles —for example when testing a new scenario— will certainly create issues such as creating duplicate new *scenario* branches among the affected tiles' VCSs.

How do you oversee this?

b) *The whole model is versioned and then the visual editor only works with a subset or tile of the whole model:*

   This approach is more solid from a conceptual point of view. That is because a single VCS structure is utilized to manage all the changes of the full 3DCM, so there is only one VCS structure to be managed. The biggest challenge in this case is technical. That is the capability of working on a subset of the full 3DCM and then merging it (i.e. the changes carried out) back to the full 3DCM, in order to incorporate the updates into it.

   The challenge lies in developing a merging mechanism to "pass" these changes back to the full 3DCM. That requires a mechanism of merging 3D geometries among others, in order for change detection to be possible and stored within the VCS. It will also extent the functionality of the workflow to address key point #5 (see Table 3.2), which will dramatically increase the enrichment potential of the 3DCM by external projects that use the 3DCM as their base. More details on that in Section 8.2.2.

7.4.2 Identifying the optimal "smallest entity" in practice

The fact that *CityObject* is the "smallest entity" of the VCS (see Section 4.2.3) is expected to raise many conflicts. Many of them will be false conflicts from a human perspective. Training practitioners to familiarize with what conflicts are in the first place, how to identify a false from an actual one and how to resolve them is undoubtedly going to obstruct the adoption of the workflow. It is very possible after testing the workflow in practice, another entity to be selected as the "smallest entity" to optimize the performance of the VCS.

In my opinion, an extensive time period of continuous testing of the workflow in real conditions including recurring feedback would set the foundation for solid and meaningful improvement of the workflow from a developing aspect. Consequently, this whole testing and feedback venture creates some resources costs that have to

be invested for this cause. That way, the workflow will improve with respect to the above-mentioned aspects —and possible many other to be revealed during testing— allowing it to come closer to becoming operational.

7.4.3 What is now possible

Even without any further investment, the municipality can still use the visual editing component of the workflow for achieving a "limited" maintenance of the 3DCM. Instead of relying on an external party in order to provide them with the updated iteration *CityGML* file, in-house generation will be possible and potentially updating the database with the new instances.

Based on the current situation, this can happen by following (a variation of) the conceptual steps shown in Figure 4.1 whose implemented equivalent is shown in Figure 7.1.

Those steps are:

1. The *CityGML v2.0* file is converted to a *CityJSON v1.0* using citygml-tools.
2. The (normalized) converted file is imported into *Blender* for editing.
3. After editing is over, a new *CityJSON v1.0* instance is created from *Blender*. *In case the municipality wishes to only use the visual editing capability of the workflow then the new instance is converted back to CityGML and imported into their database implementation.*
4. The new instance is committed back to the versioning control platform, from where it can be retrieved (checkout command of CJV) and converted —if necessary— back to *CityGML v2.0* encoding to be compatible with the existing platform of the municipality.

Regardless what they opt for, either with the "limited" or full adoption of the workflow for maintaining the 3DCM the municipality will be able to —at least— keep the model updated at a regular time intervals, within its premises and while having the absolute control over it. That alone is a major improvement of managing the 3DCM.

Furthermore, history of the maintenance can be kept automatically without the need to keep the obsolete objects stored separately, which is the current practice. In addition, new ideas and scenarios can be tested by creating new branches at no extra expense and without disrupting the actual maintenance. The workflow is highly scalable and adaptive which means that new branches can be created and deleted at will and for whatever use.

Last but not least, outsourcing will not be necessary anymore. With no outsourcing comes no financial burden for outsourcing, relieving the budget of the municipality. The same budget could be invested on the pilot testing of the workflow which was described in Section 7.4.2, however that is completely up to the municipality.

7.4.4 What is very likely to be improved

The four steps presented in Section 7.4.3 is what currently needs to be done in order for compatibility between *CityGML* and *CityJSON* to be established so that the 3DCM can be edited before converting back to *CityGML* for inserting the new instance in the database.

However, the process that these steps describe is very likely to be simplified in the

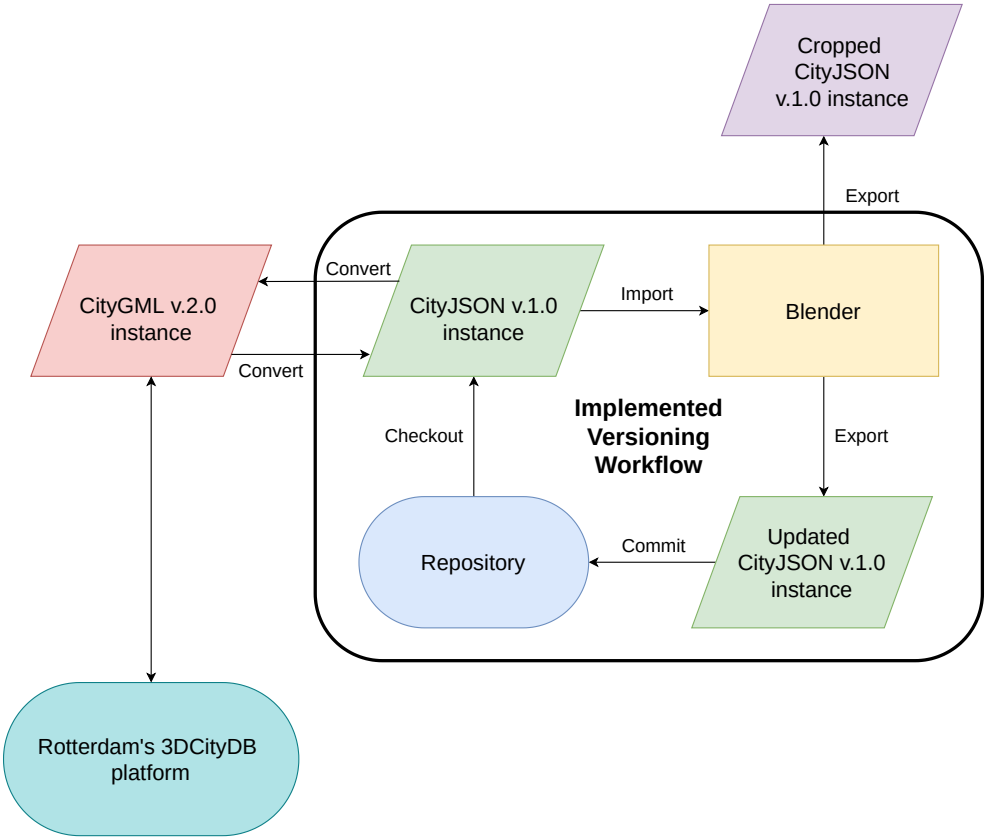


Figure 7.1: The implemented *core* workflow in conjunction with Rotterdam's 3D platform.

future. *3DCityDB* currently used by Rotterdam to store the 3DCM utilizes *citygml4j*[2] functionality. *citygml4j* also supports parsing from *CityJSON* format. Thus, it is very possible that *3DCityDB* will fully support the *CityJSON* file format in the future. That would render step #1 unnecessary simplifying the whole task of integrating the workflow with *CityGML* files.

2 https://github.com/citygml4j

8 FUTURE WORK

8.1 *GIT-FLOW* CRITICISM AND ALTERNATIVES

The versioning component of the suggested workflow is based on the *git-flow* (see Section 2.3.1). That is simply because it is one of the most successful and adopted git workflows to date.

However, there is some criticism on the *git-flow* from the developers of similar git based workflows. For example the developers of the *GitLab workflow* propose their workflow as an improvement of the *Git flow* integrating it with an issue tracking system and utilizing *master/main* branch should more than *git-flow*.

In my opinion, an issue tracking system with 3DCMs does not offer important improvements, since the maintenance transactions of a 3DCM do not vary too much in contrast with software development (new features). In other words the *GitLab flow* might be better for software development but not such a big improvement for the 3DCM domain.

Regardless that, it would be interesting to implement a 3DCM oriented workflow based on it and investigate the performance and functionality in practice.

8.2 FURTHER DEVELOPMENT OF THE WORKFLOW

The fact that 3DCM maintenance is a relatively new domain with no universally accepted or standardized solutions so far creates a lot of space for improvement. More specifically with respect to the workflow suggested there are additions and improvements that could boost its functionality. These aspects are discussed in the next sections.

8.2.1 Integrating validity check within the workflow

It is really useful to have every new 3DCM instance checked for the validity of its objects geometry to reassure that all geometries are valid against a standard. It can be considered as a quality control which at every step makes sure no degradation of (at least) geometric information occurs.

There is already an implemented solution for validating *CityJSON* encoded 3DCM files named val3dity. Integrating *val3dity* in the workflow that is suggested in this thesis would make the workflow more complete and robust.

8.2.2 Merging back subsets to the repository

At the moment the workflow is capable only for exporting a subset of the 3DCM for external applications. Having the ability to merge back the (extra information included in a) subset into the VCS means that any application stemming from the 3DCM can potentially contribute back to it. For example a more detailed model of

a building due to an architectural project that enhanced the LoD 2 of the model for that building. Or incorporate the "watertight" geometries of the buildings included in a subset that was used for a wind flow or flood simulation.

Of course this requires to make sure that the augmented subsets fulfill some predefined quality standards so that there is no information degradation or loss.

To incorporate subsets used for external applications back into the VCS, the subset needs to be in *CityJSON* format. If not, it has to be converted. When converting between formats, there is always the case of information loss because of the different data models among 3D formats that do not hold the same information.

Supposing that all these prerequisites are met, there is then the need for a sophisticated 3D merging algorithm that merges the augmented/changed objects with the existing ones. A solution to that could be something similar to what Doboš and Steed [2012] suggest. In any case, solving this problem is far from trivial, but will bring the managing of the 3DCM to a whole new level, so in my opinion it is somehting definitely worth researching into.

Another advantage of being able to merge back subsets into the repository via the VCS is that maintainers will be able work on a subset of the whole 3DCM rather than on its entirety. At the same time the challenge mentioned in Section 7.4.1 is addressed in a very efficient way. From a computational point of view this will have a big impact in efficiency and time costs, as well.

8.2.3 Combine GIS software

3DCMs are already geo-referenced at their actual real world coordinates. That could be the basis for spatial analysis using GIS applications. Investigating into augmenting the visualization platform with GIS capabilities could lead into big improvements for the visualization platform itself while allowing users to run further analysis in the 3DCM. From the most simple task of inserting a 2D map as a base reference below the model for better spatial understanding up to calculating optimal routes within the 3DCM and so on.

There is already a GIS-oriented free and open source add-on for *Blender*, namely BlenderGIS. In my opinion, it is worth doing some investigation into combining its functionality with *Up3date* or even integrate (part of) them into a new more versatile and functional add-on that serves both purposes.

8.2.4 Updating *BAG* as a consequence of the 3D city model maintenance

In the case of the municipality of Rotterdam *BAG* is one of the main component datasets for the creation of the 3DCM. Currently it is updated separately inside the municipality then outsourced to external parties every two years for the new iteration of the model to be created upon. *BAG* contains all the descriptive non-spatial information (attributes) and the geometrical footprint for each building and this information is passed into the model during the creation process Figure 3.2.

The idea is that since from the maintenance of the 3DCM all the history is kept, a log file can be created from the VCS that can be used as a guide to update *BAG*. That would reverse the current procedure of consulting *BAG* to identify which buildings have been changed and then update the 3DCM based on that. It would bring the 3DCM into the center of the spatial datasets' ecosystem, making it the main dataset according to which everything else is maintained.

Updating *BAG* would increase the robustness of the whole maintaining domain of the 3DCM, because it converts the versioning platform into a hub from which all satellite datasets can benefit from. The model can be continuously updated by the municipality with simple controllable workflows (introduced in Chapter 4) without the need for outsourcing. Last but not least, the whole BAG update process will not affect the versioning of the 3DCM at all, which minimizes the complexity of implementing this approach.

8.2.5 Creating a generator for automatic generation of instances for the *release* branch

The *release* branch is designed and implemented in the *multi-branch* structure to provide the actual 3DCM stripped off of any sensitive information for educational purposes, application developers etc.

Removing the sensitive records is all about wiping some attributes from the respective *CityObjects*. Implementing the software for this to happen automatically and integrate in into the suggested workflow would be really useful and increase the value of the workflow.

8.2.6 Handling building textures

Building textures were out of this scope of this thesis as mentioned in Section 1.4. However, building facade textures are already available for the city of Rotterdam and it would be an upgrade to handle them as well within *Blender* upon the editing process.

Since they are stored externally it makes sense for them to be excluded from the VCS. However, *Blender* supports textures. Perhaps the most suitable approach is to enrich building objects with texture is by using *Blender's UV sphere* functionality.

LIST OF FIGURES

LISTINGS

LIST OF TABLES

BIBLIOGRAPHY

Agugiaro, G. (2016). First steps towards an integrated CityGML-based 3D model of Vienna. volume III-4, pages 139–146.

Airaksinen, E., Bergström, M., Heinonen, H., Kaisla, K., Lahti, K., and Suomisto, J. (2019). The Kalasatama Digital Twins Project - The final report of the KIRA-digi pilot project.

Bakker, N. (2009). Key registers as base of the dutch SDI. *GSDI World Conference*.

Ball, T., Kim, J., Porter, A., and Harvey, P. S. (1997). If your version control system could talk.

Biljecki, F., Ledoux, H., Du, X., Stoter, J., Soon, K. H., and Khoo, V. H. S. (2016). The most common geometric and semantic errors in CityGML datasets. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-2/W1:13–22.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., and Çöltekin, A. (2015). Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, 4(4):2842–2889.

Blender Foundation (2019). Blender 2.82 reference manual.

Boeters, R., Arroyo Ohori, K., Biljecki, F., and Zlatanova, S. (2015). Automatically enhancing CityGML LOD2 models with a corresponding indoor geometry. *International Journal of Geographical Information Science*, 29(12):2248–2268. ISSN: 1365–8816 (Print), 1362–3087 (Online).

Brindescu, C., Codoban, M., Shmarkatiuk, S., and Dig, D. (2014). How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press.

Cao, R., Zhang, Y., Liu, X., and Zhao, Z. (2017). 3d building roof reconstruction from airborne LiDAR point clouds: a framework based on a spatial database. *International Journal of Geographical Information Science*, 31(7):1359–1380.

Chacon, S. and Straub, B. (2019). *Pro Git*. Apress.

Chaturvedi, K., Smyth, C. S., Gesquière, G., Kutzner, T., and Kolbe, T. H. (2016). Managing Versions and History Within Semantic 3D City Models for the Next Generation of CityGML. In *Advances in 3D Geoinformation*, pages 191–206. Springer International Publishing.

Doboš, J. and Steed, A. (2012). 3D Diff. In *SIGGRAPH Asia 2012 Technical Briefs on - SA '12*. ACM Press.

Döllner, J., Baumann, K., and Buchholz, H. (2007). Virtual 3D City Models as Foundation of Complex Urban Information Spaces.

Döllner, J., Kolbe, T., Liecke, F., Sgouros, T., and Teichmann, K. (2006). The virtual 3D city model of Berlin - Managing, integrating, and communicating complex urban information. *Proceedings of the 25th Urban Data Management Symposium UDMS*.

Franceschi, S., Adoch, K., Kang, H. K., Hupy, C., Coetzee, S., and Brovelli, M. A. (2019). OSGEO UN Committee Educational Challenge: A use case of sharing software and experience from all over the world. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-4/W14:49–55.

Glasser, A. L. (1978). The evolution of a Source Code Control System. *ACM SIGSOFT Software Engineering Notes*, 3(5):122–125.

Gröger, G., Kolbe, T. H., Nagel, C., and Häfele, K.-H. (2012). *OGC City Geography Markup Language (CityGML) Encoding Standard*. Open Geospatial Consortium, 2.0.0 edition.

Kutzner, T., Chaturvedi, K., and Kolbe, T. H. (2020). CityGML 3.0: New Functions Open Up New Applications. *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88(1):43–61.

Law, D. (2010). Versioning 101: Essential information about ArcSDE geodatabases.

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Labetski, A., and Vitalis, S. (2019). CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(1):4.

Malambo, L. and Hahn, M. (2010). LiDAR assisted CityGML creation.

Noardo, F., Biljecki, F., Agugiaro, G., Arroyo Ohori, K., Ellul, C., Harrie, L., and Stoter, J. (2019). GeoBIM Benchmark 2019: Intermediate Results. In *14th 3D GeoInfo Conference 2019*, volume XLII-4 of *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 47–52. ISPRS.

Oracle (2020). Understanding Versioning.

Prieto, I., Izkara, J. L., and Béjar, R. (2017). A continuous deployment-based approach for the collaborative creation, maintenance, testing and deployment of CityGML models. *International Journal of Geographical Information Science*, 32(2):282–301.

Rochkind, M. J. (1975). The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370.

Samuel, J., Périnaud, C., Servigne, S., Gay, G., and Gesquière, G. (2016). Representation and Visualization of Urban Fabric through Historical Documents. In Catalano, C. E. and Luca, L. D., editors, *Eurographics Workshop on Graphics and Cultural Heritage*. The Eurographics Association.

Samuel, J., Servigne, S., and Gesquière, G. (2020). Representation of concurrent points of view of urban changes for city models. *Journal of Geographical Systems*.

Samuel, J., Servigne, S., and Gesquière, G. (2018). Urbanco2fab: comprehension of concurrent viewpoints ofurban fabric based on git. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-4/W6:65–72.

Schüler, N., Agugiaro, G., Cajot, S., and Maréchal, F. (2018). Linking interactive optimization for urban planning with a semantic 3D city model. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-4:179–186.

Spinellis, D. (2012). Git. *IEEE Software*, 29(3):100–101.

Steinhage, V., Behley, J., Meisel, S., and Cremers, A. B. (2010). Automated updating and maintenance of 3D city models. *ISPRS Volume XXXVIII-4-8-2/W9, 2010*.

Vitalis, S., Labetski, A., Arroyo Ohori, K., Ledoux, H., and Stoter, J. (2019). A data structure to incorporate versioning in 3D city models. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, IV-4/W8:123–130.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., and Kolbe, T. H. (2018). 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3(1).

A REPRODUCIBILITY SELF-ASSESSMENT

A.1 MARKS FOR EACH OF THE CRITERIA

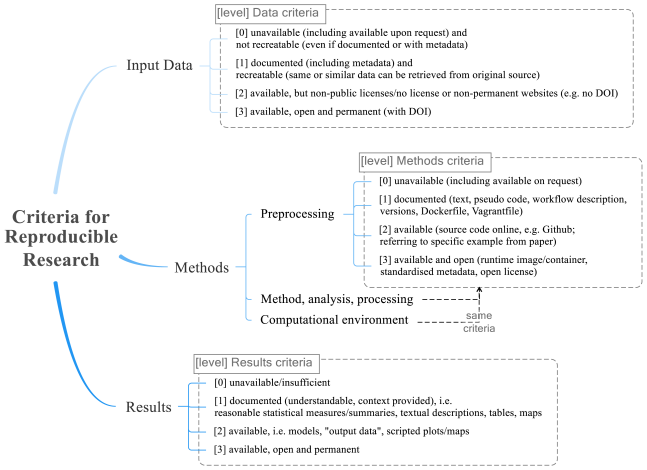


Figure A.1: Reproducibility criteria to be assessed.

A.2 REPRODUCIBILITY OF THESIS/RESULTS

Criteria	Score
Input data	2
Pre-processing	3
Methods, Analysis, Processing	3
Computational Environment	3
Results	2

Table A.1: Self-assessment of scores regarding the criteria presented in Figure A.1

A.3 SELF-REFLECTION ON THE REPRODUCIBILITY

The data, processes and software tools were all retrieved, inspired and implemented without any restrictions. More specifically the 3D City Model of Rotterdam is publicly available for download through the official website. All the software used is also free and open source publicly available at *github*. Every part of what is introduced and used in this thesis can be accurately reproduced at any time and without any restrictions.

COLOPHON

This document was typeset using LaTeX. The document layout was generated using the arsclassica package by Lorenzo Pantieri, which is an adaption of the original classicthesis package from André Miede.