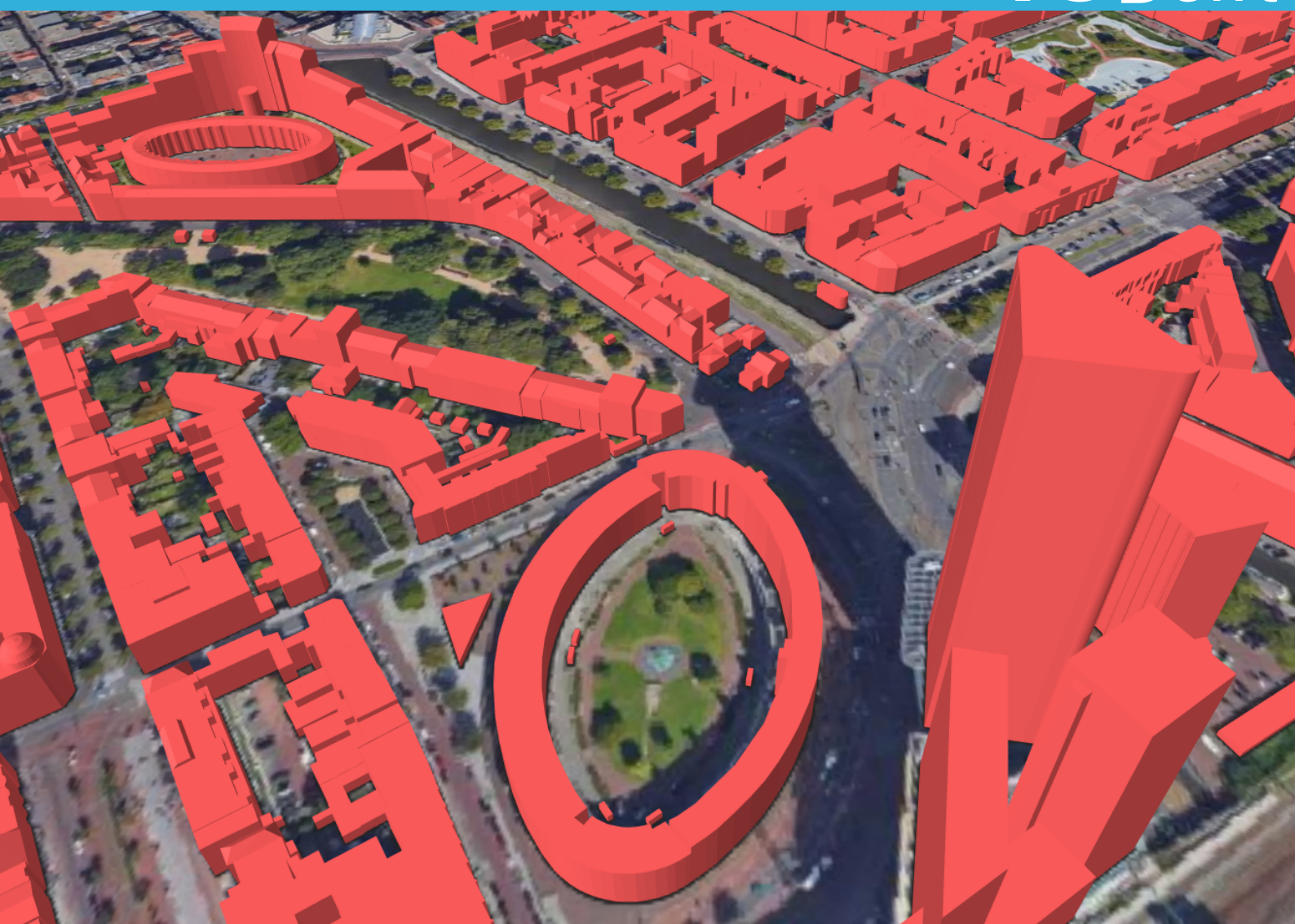


MSc thesis in Geomatics

Development of a QGIS plugin for the CityGML 3D City Database

Konstantinos Pantelios

2022



MSc thesis in Geomatics

Development of a QGIS plugin for the CityGML 3D City Database

Konstantinos Pantelios

June 2022

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

Konstantinos Pantelios: *Development of a QGIS plugin for the CityGML 3D City Database* (2022)
© ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.
To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

The work in this thesis was carried out in the:



3D geoinformation group
Delft University of Technology

with valuable support from:



Virtual City Systems
Berlin, Germany

Delft University of Technology

Supervisors: Camilo León Sánchez
Giorgio Agugiaro
Co-reader: Martijn Meijers

Virtual City Systems

External Supervisors: Claus Nagel
Zhihang Yao

Abstract

Today, in the urban planning, energy modelling and other fields, semantic 3D city models are used in various applications like visualization, data exploration, analysis and more. As a result, standard data practices needed to be set in order to facilitate the storage and exchange of these city models. For this purpose, the Open Geospatial Consortium (OGC) adopted CityGML as an international standard for effective use of 3D city models. Generally, the models are encoded in Extensible Markup Language (XML) files, however, other file encodings can also be used like JavaScript Object Notation (JSON) files with CityJSON. Moreover, CityGML can also be adapted for a database encoding like the 3D City Database (3DCityDB), on which this thesis is based upon. The benefit of using a database encoding is that databases are built to handle and organize large amount of data, which 3D city models usually consist of. The 3DCityDB is an open source project developed for PostgreSQL and Oracle databases. It is supported by other software in the 3DCityDB suite that facilitate its use in different applications. The 3DCityDB tries to simplify the complexity of CityGML, however, its approach remains difficult for users to access data directly without technical knowledge of databases, Structured Query Language (SQL), CityGML and/or 3DCityDB structure. Derived from this limitation, the primary objective of this research is to develop an approach that could simplify user interaction with the 3DCityDB from within a Q Geographical Information System (QGIS) environment. To achieve this, "3DCityDB-Loader", a QGIS plugin, is developed to handle complex server operations in the background, whilst providing a user-friendly workspace environment. The complete functionality of the plugin is segmented into client and server-side parts. This thesis focuses on the client-side development but both parts were jointly developed in a common iterative process of requirement identification, development, testing and assessment. The most important requirements for the plugin is to have layers that can interact with 3DCityDB data, be able to work with multiple users with different privileges, allow for multiple scenarios (database schemas), allow to edit attributes, handle different Levels Of Detail (LOD) and geometry representations and finally be able to operate from a Graphical User Interface (GUI) in QGIS. Regarding the client-side part of the plugin, it can manage database connections, manage the server-side installation, manage and create layers for multiple scenarios from a GUI, include CityGML generic attributes, enumerations and codelists and automatically set their relations, automatically structure a hierarchical QGIS Table Of Contents (TOC) and finally automatically apply standard colors on different features. At the time of writing this document, the plugin is at version 0.4. The limitations are mostly related to functionalities that are not yet supported, with future development being tracked from the project's GitHub repository. All in all, "3DCityDB-Loader" facilitates the use of 3DCityDB for users of different fields and expertise with the common denominator being the well-accustomed QGIS environment.

Acknowledgements

This document is my graduation thesis of the MSc Geomatics programme in Delft University of Technology. Here, I would like to express my appreciation and gratitude to everyone that contributed directly or indirectly to the completion of this research. Peoples' help and support were an important driving factor that pushed me towards a productive and consistent path.

Firstly, I give many thanks to everyone involved from Delft University of Technology. In particular, I would like to thank my main supervisor Camilo Leon Sanchez for his guidance, feedback and support throughout the span of the research. I would also like to thank my second supervisor Giorgio Agugiaro for the valuable collaboration regarding the software development phase of this research and for all constructive remarks. In addition, I thank them both for the weekly meetings that helped to keep my progress consistent throughout the year. Moreover, I would like to thank the co-reader of this thesis, Martijn Meijers, for his interest, time allocated in reviewing my research and feedback.

Secondly, I am grateful for the opportunity to cooperate with my external supervisors from Virtual City Systems. I thank both Claus Nagel and Zhihang Yao for providing their expertise and critical feedback. Their contribution as experts in the field and their association with this research was an important aspect of this endeavour. Moreover, I thank them both for helping in the testing phase of the developed software.

I would also like to thank María Aparicio Sánchez for her interest in the research and willingness to use and test the software. All testers are deeply appreciated as they helped to identify important technical issues.

Finally, I would like to express my gratitude to my parents and my girlfriend. I thank my parents for their "remote support" from the other side of Europe. I also thank my girlfriend Foteini Kourdoukla for all the love and emotional support which was motivating me from the start.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Research questions	3
1.3. Research scope	4
1.4. Research overview	4
1.5. Research structure	5
2. Related work	7
2.1. CityGML	7
2.2. 3D City Database	11
2.3. 3D City Database "Plus"	14
2.4. QGIS	15
2.5. Qt	20
2.6. Related QGIS plugins	25
2.6.1. 3DCityDB Explorer	26
2.6.2. 3DCityDB Viewer	27
2.6.3. CityJSON Loader	28
3. Methodology	31
3.1. Primary requirement identification and implementation	31
3.1.1. Database Connection	32
3.1.2. Multi-user capabilities	33
3.1.3. User privileges	33
3.1.4. Layer structure	33
3.1.5. Layer operations	34
3.2. Secondary requirement identification and implementation	35
3.2.1. Plugin structure	35
3.2.2. QGIS structure	36
3.3. Plugin use	37
3.3.1. Server-side use	38
3.3.2. Client-side use	38
3.4. Development Details	39
3.4.1. Software and tools	39
3.4.2. Testing	40
4. Plugin structure (server/client-side)	43
4.1. Server-side design	43
4.2. Client-side design	44
4.2.1. Plugin Initialization	44
4.2.2. Plugin GUI	46
4.3. Administration dialog	48
4.3.1. "Database Administration" tab	49

Contents

4.4.	User dialog	51
4.4.1.	“User Connection” tab	52
4.4.2.	“Layers” tab	59
4.4.3.	“About” tab	62
4.5.	QGIS project structure	63
4.5.1.	Layers	63
4.5.2.	Relations	68
4.5.3.	Table of Contents	69
4.6.	Software development	70
4.6.1.	Object-oriented programming	70
4.6.2.	Working directory	71
5.	Test case implementation	75
5.1.	Scenario	75
5.2.	Pipeline	76
5.2.1.	Plugin installation	76
5.2.2.	Database setup	77
5.2.3.	Updating the database	79
5.2.4.	Viewing the database	80
5.2.5.	Database maintenance	81
5.2.6.	Uninstalling the plugin’s server-side contents	81
6.	Conclusions	83
6.1.	Research questions and answers	87
6.2.	Discussion	87
6.2.1.	Limitations	88
6.2.2.	Future Development	91
A.	Reproducibility self-assessment	93
A.1.	Marks for each of the criteria	93
A.2.	Self-reflection	93
A.2.1.	Input data	93
A.2.2.	Methods	94
A.2.3.	Results	94
B.	“3DCityDB-Loader” characteristics	95
B.1.	Layer properties	95
B.2.	GUI design evolution	97
B.3.	Test data-sets	100

List of Figures

2.1. "UML package diagram illustrating the separate modules of CityGML and their schema dependencies. Each extension module (indicated by the leaf packages) further imports the GML 3.1.1 schema definition in order to represent spatial properties of its thematic classes." (Figure from [Gröger et al., 2012])	8
2.2. UML diagram of the "site" superclass along with its subclasses. (Figure adapted from [Gröger et al., 2012])	8
2.3. "UML diagram of CityGML's geometry model (subset and profile of GML3): Primitives and Composites." (Figure from [Gröger et al., 2012])	9
2.4. "UML diagram of CityGML's geometry model: Complexes and Aggregates." (Figure from [Gröger et al., 2012])	9
2.5. UML enumeration example of "CityObject"'s "relative to terrain" valid values. (Figure adapted from [Gröger et al., 2012])	10
2.6. "CityGML noise application schema – city furniture model (light yellow=CityGML CityFurniture module, light orange=CityGML Noise ADE). Prefixes are used to indicate XML namespaces associated with model elements. Element names without a prefix are defined within the CityGML CityFurniture module. The prefix noise is associated with the CityGML Noise ADE (source: Institute of Geodesy and Geoinformation Uni Bonn)." (Figure from [Gröger et al., 2012])	11
2.7. Geometry hierarchy for a solid geometry object [3DCityDB, 2021a].	14
2.8. List of QGIS project properties.	16
2.9. QGIS map canvas widget.	16
2.10. QGIS Table of Contents.	17
2.11. Visual comparison between attribute table and form.	18
2.12. QGIS Symbology properties of a layer.	19
2.13. QGIS plugin manager.	20
2.14. Qt5 push button widget example. (Figure from documentation [The-Qt-Company, 2018i])	21
2.15. Qt5 text label widget example. (Figure from documentation [The-Qt-Company, 2018h])	21
2.16. Qt5 combo box widget example. (Figure from documentation [The-Qt-Company, 2018a])	21
2.17. Qt5 tab widget example. (Figure from documentation [The-Qt-Company, 2018j])	22
2.18. Qt5 group box widget example. (Figure from documentation [The-Qt-Company, 2018f])	22
2.19. Qt5 graphics view widget example. (Figure from documentation ([The-Qt-Company, 2018d])	23
2.20. Qt5 extent group box example.	23
2.21. Qt5 checkable combo box example.	24
2.22. Qt5 standard layout structures.	24
2.23. Qt5 slot/signal implementation. (Figure from documentation [The-Qt-Company, 2018m])	25
2.24. 3DCityDB Explorer implementation example.	26

List of Figures

2.25. 3DCityDB Viewer dialog.	27
2.26. CityJSON Loader dialog.	28
2.27. An example semantic 3D city model, imported in QGIS from CityJSON file using the "CityJSON Loader" plugin. (red="Building" of LOD2, purple="BuildingPart" of LOD2)	29
3.1. Example of open issues (feedback) in GitHub repository [Pantelios and Aguiaro, 2022].	41
4.1. "3DCityDB-Loader" actions in "Database menu".	44
4.2. "3DCityDB-Loader" User pipeline.	45
4.3. "3DCityDB-Loader (Administration)" Administrator pipeline.	46
4.4. "3DCityDB-Loader" tab widgets (from "User" dialog).	46
4.5. Enabled/Disabled state of Qt5 widgets.	47
4.6. "3DCityDB-Loader (Administration)" Administration dialog at its initial state.	48
4.7. "3DCityDB-Loader (Administration)" Installation options.	50
4.8. "3DCityDB-Loader" initial GUI state ("User" dialog).	51
4.9. List of available PostgreSQL connections in "Data Source Manager" and "3DCityDB-Loader"	53
4.10. New PostgreSQL connection dialogs in "Data Source Manager" and "3DCityDB-Loader"	54
4.11. "3DCityDB-Loader" available "citydb" schemas example.	55
4.12. "3DCityDB-Loader", "User Connection" OSM base-map example. (Blue="citydb" extents, Red=database layer extents (user-selected extents))	56
4.13. "3DCityDB-Loader", Advanced options with their default values set. (Decimal precision=3, Minimum area=0.0001m ²)	57
4.14. "3DCityDB-Loader" Layer operations.	57
4.15. "3DCityDB-Loader", "Connection status" report example. (Green=passed checkpoint, Red=failed checkpoint)	57
4.16. "3DCityDB-Loader" push button widget used to close the connection to the database.	58
4.17. "3DCityDB-Loader" "Layers" tab in its initial state.	59
4.18. "3DCityDB-Loader" workspace label widgets example in "Layers" tab.	60
4.19. "3DCityDB-Loader" OSM base-map example in "Layers" tab. (Blue="citydb" extents, Red=database layer extents Green=QGIS layer extents (user-defined extents))	60
4.20. "3DCityDB-Loader" example of feature selection parameters available in user-defined extents.	61
4.21. "3DCityDB-Loader" example of multiple available layers to import.	61
4.22. "3DCityDB-Loader" example of warning message about a large amount of features.	62
4.23. "3DCityDB-Loader" "About" tab.	62
4.24. View name examples in a database (Figure from pgAdmin web view).	64
4.25. Attribute form - "Building" attributes example.	66
4.26. Example of an attempt to pass wrongful values. A value of -6 is inappropriate for the "storeys above ground" field. This caused the constraint (described in the red square) to take effect by disabling the "OK" button. The user is now forced to resolve the issue or cancel the attempt.	67
4.27. Structured ToC example.	69
4.28. "3DCityDB-Loader" Working directory diagram.	71

4.29. Metadata shown in "Plugin dialog".	72
4.30. "Qt Designer" GUI designing example.	73
5.1. Plugin workflow of the scenario example	76
5.2. QGIS plugin installation from ZIP	76
5.3. Creating schemas for users from "3DCityDB-Loader (Administration)"	78
5.4. Selected Dijkershoek extents (in red square).	79
5.5. Importing layers in QGIS from "3DCityDB-Loader"	80
5.6. An example semantic 3D city model, imported in QGIS from a 3D City Database using the "3DCityDB-Loader" plugin and visualized using the "Qgis2threejs" plugin. (red="Building" of LOD2, green="SolitaryVegetationObject" of LOD3, light brown="TINRelief" of LOD1)	81
6.1. Converting SQL queries (Listings 6.1,6.2) into QGIS no-code operations.	85
6.2. Visual comparison between Attribute Table and Form structure of a building layer.	86
6.3. Warning message before refreshing materialized views.	89
6.4. Example of QGIS 3D rendering artefacts (bottom). In FME the artefacts are not present (top).	90
A.1. Reproducibility criteria to be assessed.	93
B.1. Pre-structured QML files are stored in the "forms" directory for a multitude of different CityGML features. These hold the layer properties rules (symbology and attribute forms). Users can modify these files manually or by changing the properties from QGIS "layer properties" and overwriting them the corresponding file.	95
B.2. Available custom color schema (v0.4). This symbology is stored in the QML files (qml file) that accompany each layer (layer name).	96
B.3. Old "3DCityDB-Loader" plugin design (as of 05/01/2021 v0.1)	97
B.4. Current "3DCityDB-Loader" plugin design (as of 29/06/2022 v0.4)	98
B.5. "3DCityDB-Loader (Administration)" initial GUI state ("Database Administration" tab).	99
B.6. Railway data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).	100
B.7. House data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).	101
B.8. Rijssen-Holten data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).	102
B.9. Den Haag data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).	103

List of Tables

2.1.	<i>LOD 0-4 of CityGML with their proposed accuracy requirements (discussion proposal, based on: [Albert et al., 2003]). (Table from [Gröger et al., 2012])</i>	10
2.2.	Dictionary example of a SIG3D codelist for the CityGML "TransportationComplex" class. (Table from [Gröger et al., 2012])	11
3.1.	Identified requirement and implementation.	32
3.2.	Data-set overview. * <i>Due to size constraint, this data-set cannot be stored into the project's GitHub repository. It is stored into a private Google Drive directory.</i>	40
4.1.	QGIS libraries [QGIS-Python-API, 2022]	70

Listings

2.1. Example of an SQL statement written to perform a query to extract roof surfaces of buildings constructed from 2015 to now.	13
2.2. Example of calling a function in the 3DCityDB "Plus" to insert a building. [Agu-giario, 2018].	15
4.1. Example custom expression for "measured_height" and "measured_height_unit" fields.	67
4.2. Example custom expression of "storeys_above_ground".	67
4.3. Part of the Metadata.txt file.	72
4.4. Compiling resources from terminal.	73
5.1. SQL queries to set-up new users	77
6.1. Accessing roof surfaces of buildings constructed from 2015 to now. (Using vanilla 3DCityDB)	84
6.2. Accessing roof surfaces of buildings constructed from 2015 to now. (Using server-side of "3DCityDB-Loader")	84

Acronyms

3DCityDB 3D City Database	v
ADE Application Domain Extension	2
API Application Programming Interface	15
BAG Basisregistratie Adressen en Gebouwen / Dutch Cadastre	40
COLLADA COLLABorative Design Activity	12
CSV Comma Separated Values	2
DTM Digital Terrain Model	36
ESRI Environmental Systems Research Institute	3
GIS Geographical Information System	2
gITF Graphics Language Transmission Format	12
GML Geography Markup Language	1
GRASS Geographic Resources Analysis Support System	19
GUI Graphical User Interface	v
JSON JavaScript Object Notation	v
KML Keyhole Markup Language	2
LOD Levels Of Detail	v
LTR Long Term Release	4
LTS Long Term Support	4
OGC Open Geospatial Consortium	v
OOP Object-Oriented Programming	38
OSM Open Street Map	35
PL/pgSQL SQL Procedural Language	12
PSQL PostgreSQL Interactive Terminal	50
QGIS Q Geographical Information System	v
QML Qt Modeling Language	19
RDBMS Relational Database Management System	12
SAGA System for Automated Geoscientific Analyses	19
SFS Simple Feature for SQL	44
SFM Simple Feature Model	2
SIG3D Special Interest Group 3D	10
SLD Styled Layer Descriptor	19
SQL Structured Query Language	v
SRDBMS Spatial Relational Database Management System	3
TOC Table Of Contents	v
TU Technische Universiteit	1
UI User Interface	3
UML Unified Modeling Language	3
URI Uniform Resource Identifier	17
URL Uniform Resource Locator	21
UX User eXperience	3
VCS Virtual City Systems	4
WFS Web Feature Service	12

Listings

wms Web Map Service	15
wmts Web Map Tile Service	15
xlsx Microsoft Excel Open XML Spreadsheet	15
xml Extensible Markup Language	v

1. Introduction

1.1. Motivation

With the advancements in technology and especially computing power and software abundance, 3D city model functionality moved from only data visualisation to well refined data analysis applications. In practice city planners and other relevant actors can use city models to create simulations regarding energy consumption, traffic, growth, disaster management or any other metric that can facilitate urban management and planning [Biljecki et al., 2015].

The issue that often arises with city models is the complexity of the city as an entity. Cities are composed of many different objects which are not guaranteed to be found in every city and at the same type. Additionally, a particular type of a city model that is structured to accommodate the energy sector might not be able to be used in other applications. Lastly, there could be many different data structure formats used by different people and organizations making data communication and interoperability difficult [Stadler et al., 2009].

These are some of the motivations that led the OGC towards the adoption of the CityGML standard. CityGML is a data model that is openly available for use allowing city models to be shared between different people and organisations with ease. Moreover, it aims to solve the interoperability issue as its default XML encoding can be used universally [World-Wide-Web-Consortium et al., 2010]. Regarding the data model, CityGML tries to accommodate for as much information as possible which means that main classes are hierarchically linked to more detailed features representing the city as best as possible [Gröger and Plümer, 2012]. An example of this, is that buildings can have building parts or other, relative to the building, installations with particular attributes. It quickly becomes apparent that while this model is really comprehensive, its XML based hierarchical structure between the features, attributes and geometries is bound to produce enormous Geography Markup Language (GML) files that can be hard to work with [Lu et al., 2007]. A city center consisting by a large amount of buildings, depending on the level and type of detail, can take up many gigabytes of storage space.

This complexity hints towards the development and use of different encodings. One such encoding is based on JSON, namely CityJSON which was developed by Technische Universiteit (TU) Delft [Ledoux et al., 2019]. Another one is based on SQL with the city model data stored exclusively in a geo-spatial relational database. This approach was followed by a team at TU Berlin who originally developed the open '3D City Database' software [Yao et al., 2018].

Databases come with intrinsic characteristics that facilitate the storage, access and usage of 3D city models. To begin with, such models usually occupy large storage space in computer memory. Consequently, it is a good approach to store them in databases that have a lot of available storage and processing capabilities. In addition to storage, databases allow for data accessibility. Accessibility comes in the form of direct access or the use of software application, and in the form of local or remote access in networks. Moreover, databases can make

1. Introduction

use of embedded data structures (in PostgreSQL: B-Tree, GiST and other indices) to optimize specific queries. The use of indices can be especially useful for large city models with a lot of data entries. Overall, databases are used to organize, manage and use large amounts of data [Stolze, 2003].

A database encoding for CityGML is the [3DCityDB](#). It operates with geo-spatial relational databases like PostgreSQL/PostGIS and Oracle and follows the CityGML [OGC](#) standard. In practice CityGML files are imported into the database based on a set of mapping rules storing data into a set of predefined tables. Having city models stored in this way gives the benefit of database operations that help in data manipulation. For example the use of spatial indices can speed up spatial queries or with the use of predefined functions, users can automate specific operations. Moreover, users are able to create their own complex functions and queries to analyse data based on custom requirements. Additionally, [3DCityDB](#) is supported by a number of additional software applications that facilitate its use. With the use of its 'Importer/Exporter' application, [3DCityDB](#) provides a high level way of creating queries based on location, detail, feature etc. and allows exporting the database's contents to Keyhole Markup Language (KML), GML, JSON, Comma Separated Values (CSV) and other formats. Lastly, it allows the data model to be enhanced with different types of Application Domain Extension (ADE) using specific extension rules [Yao et al., 2018]. The "3DCityDB Importer/Exporter" is used as an indirect way to handle 3D city data with a [3DCityDB](#) database. In order to visualize or analyze data, it is required to export the data from the database by converting them into a file encoding and then be used within a 3rd party software. Moreover, it is not suited for database updates, as it is required to replace the entire schema or import the data into an empty new one. This approach require a lot of time, especially for large 3D city models.

Accessing and using the data directly from the database could overcome the above issues. However, the caveat with this approach is the complexity of [3DCityDB](#) structure.

- In particular, the database structure consist of 66 tables (for versions 4.x). These tables are mostly reserved for feature classes, but there are others used to handle relations between them, following the mapping rules of [3DCityDB](#) [3DCityDB, 2021a].
- Additionally, many of the attributes are split over multiple tables. As almost every feature class has its own table, its properties are mapped to columns in different tables, following somehow a hierarchical design pattern. Taking the "Building" class as example, the table "cityobject" contains the attributes of class "CityObject", and is connected to the linked table "building", containing some of the attributes of class "_abstractBuildng", etc. Furthermore a building is a "CityObject", which is a feature that can also have "ExternalReference", "Appearance" and "genericAttribute" classes [3DCityDB, 2021a].
- Regarding the geometry table, its structure is complex. The data is nested into multiple levels in order to handle both volumetric and surface geometries simultaneously. The top level consist of the solid geometries. Next, the composite surfaces that cover solid geometries are stored into the same table in different entries. Lastly, similarly to composite surfaces themselves, their surfaces are stored here again into different entries [3DCityDB, 2021b].
- Finally, an important notice is that the features do not follow the Simple Feature Model (SFM). The SFM is a direct way of representing earth features as vector object and is used by many Geographical Information System (GIS) software like QGIS. In short a feature class is structured as a table where each entry is a different feature instance

and the columns are composed of a unique identifier, its attributes and their geometry data [Herring et al., 2011].

Assessing from the above complexity, people that need to use this system effectively must have sufficient knowledge of Spatial Relational Database Management System (SRDBMS)s, SQL and general programming, CityGML and Unified Modeling Language (UML) comprehension. This steep learning curve means that people may not be able to use the software immediately and effectively without the active help of an expert in the field. Moreover, even expert users are not immune to this complexity as big and, by extension, error prone SQL queries, are required even for semantically simple requests. Thus, the following question arises. How can 3DCityDB be introduced to the wide audience of city planners and other users that might not have the required technical knowledge?

Nowadays, people working with geographical data use specific types of applications called GIS. Two of the most popular are the proprietary ArcGIS from Environmental Systems Research Institute (ESRI) and the open source QGIS. These example software are user-centered with GUI-based tools that leverage the access and usage of geo-spatial data to people of non-technical backgrounds. This in practice means that even though they are equipped to handle a plethora of different geographical operations using many techniques and algorithms (ranging from data management and analyses, to image and 3D processing), they do not require any programming knowledge or advanced technical skills [Steiniger and Bocher, 2009]. This is an important benefit as the offered convenience allows people with no previous experience to enter the field of geo-information fast and effectively. Their popularity caused an increased demand and offer of tutorials, courses and even certifications to be obtained from various institutions. Consequently, people in this field (like city planners) are usually proficient or accustomed in the use of GIS software [Steiniger and Hunter, 2012].

This research is going to alleviate the aforementioned limitation of 3DCityDB by developing a plugin within the QGIS software. QGIS is mainly selected due to its open source nature, meaning that the plugin, as it is open source itself, is not obstructed by pay walls and could be reached by as many people as possible. In short, QGIS can provide a recognisable user-friendly environment, while the plugin can hide from the user the complexity of the 3DCityDB schema and act as an interface between the user and the database.

1.2. Research questions

As already mentioned this research aims to make the handling of 3D city models more intuitive and approachable for users from every technical skill level. To achieve this goal, the following research questions need to be answered.

- How can QGIS be extended via a plugin to connect and use 3DCityDB in a user-friendly way?
- How can an interface be developed so that the data in the 3DCityDB can be easily accessed (both attributes and geometries) by non-expert users?
- What QGIS capabilities are considered both user-friendly and practical enough to be appreciated by both inexperienced and expert users?
- How to balance between the complexity of CityGML's database model and User Interface (UI)/User eXperience (UX)?

1. Introduction

- How to take advantage of the benefits of database stored data to be used into the [QGIS](#) environment?
- How to mediate between the possibly huge amount of data stored in a database, and the limited resources (or user's needs) in terms of data within [QGIS](#)?

1.3. Research scope

The goal of this research is to simplify user interaction with the [3DCityDB](#) using a [QGIS](#) plugin. Specifically, the core functionalities of the plugin are to allow the user to connect, load, edit feature attributes, and update the database. The main focus is on the client-side operations, the [GUI](#) structure, configuration and use of the plugin. The server-side operations are not completely out of the scope of this research as the plugin is built exclusively for those. Moreover, regarding user usage, multiple users are taken into account along with different privileges. A distinction is also being made between user types of regular users and administrator users. Lastly, regarding the [3DCityDB](#) capabilities, in this research CityGML "ADE", the "Appearance", "Address" and "ExternalReference" classes are out of scope.

More specifically, for this project the development is based on CityGML v.2.0, the [3DCityDB](#) for PostgreSQL/PostGIS versions 4.x and [QGIS](#) Long Term Release (LTR) 3.22. These versions were chosen based on their stability and time in circulation at the time of development (widely adopted).

The operating system upon which all of the above tools are installed and the plugin is developed is Ubuntu 20.04.3 Long Term Support (LTS). Nevertheless, the plugin is set to be system-independent.

Finally, the number of implemented functionalities was mainly decided upon an evaluation of priorities and is restricted more by time and less by technical limitations.

1.4. Research overview

This research was inspired by a noticeable complexity of the direct usage of 3D city models stored in databases and particularly the [3DCityDB](#). This need was identified both by personal use of such data and remarks and discussion between the supervisors of this research.

Consequently, in order to tackle this issue, the first step was to clearly define the overall goal of our endeavor, that is to simplify user interaction with the [3DCityDB](#). After deciding upon the software requirements, the next step was to identify the usage requirements. This was achieved by a number of meetings between the author, the supervisors at [TU Delft](#) and the external supervisors at Virtual City Systems (VCS), one of the companies mostly involved in the development of the [3DCityDB](#). Following these requirements, we followed an iterative process of developing the software, testing it and assessing it before refining the initial requirements and/or coming up with new ones. This cycle was repeated four times, with the fourth representing the current version of the plugin (v0.4). After the last cycle, the plugin was further developed to fix technical issues and tested also by a number of different testers.

1.5. Research structure

The first and current chapter is the introduction of the research where an overview of the current situation of the topic is briefly explained. Next follows the second chapter showcasing an in-depth analysis of literature research about existing technologies, tools and their functionalities in relation to the research topic. After that, the third chapter relates to the methodology followed to setup, experiment and produce the results. Continuing, chapter four is dedicated to more technical details relating to the developed software (programming/plugin). Chapter five describes a test case scenario on what is the intended use of the software and how to use it effectively. Lastly, in the sixth chapter, the conclusions are revealed along with a discussion section about points of interest and limitations along with a section about proposals for future development.

2. Related work

This chapter aims to identify and analyze all the elements that are relevant for this research. These elements consist of used software, their methodology, existing tools and terminology.

2.1. CityGML

The plugin related to this research is based on the [3DCityDB](#) which by itself is developed for the CityGML [OGC](#) standard. Specifically, [3DCityDB](#) supports CityGML 2.0, thus only this version of the standard is explored. Moreover, the following paragraphs illustrate a basic overview of CityGML's characteristics that are relevant for this research.

CityGML is used as a standard to visualize, store, share and overall use 3D city models and 3D models of the surface's most prevalent objects. The model is used mainly in the fields of architecture, urban development, tourism, cadastre, city management and more (e.g. [Costa-magna and Spanò](#) utilized CityGML in a case-study for Architectural Heritage). It is [XML](#)-based applied with the [GML3](#) (version 3.1.1), which is a data model used for geographical features [[Lake, 2005](#)]. The CityGML 2.0 standard was published in 2012 and is a continuation of the CityGML 1.0 published in 2008. It is worth mentioning that in 2021 CityGML moved to version 3.0 [[Kolbe et al., 2021](#)], however, due to its early stage, related software, tools and models still need to catch up. Many tools exist that are tasked to handle CityGML 2.0 data models, one of those being the "3D City Database" which is going to be described later.

As the goal of the model is to represent reality, there is a vast amount of relations set between many different types of objects. The approach that is followed in CityGML 2.0 is to group different classes in so-called thematic modules (Figure 2.1). For example, the module "Building" contains all classes that are used to model buildings, etc. All thematic modules share some common classes, contained in the "Core" module. These modules contain further compositions, aggregates and generalizations in accordance with how relations are set in reality for these objects. For example, the feature "site" relating to buildings, bridges, and tunnels (Figure 2.2).

2. Related work

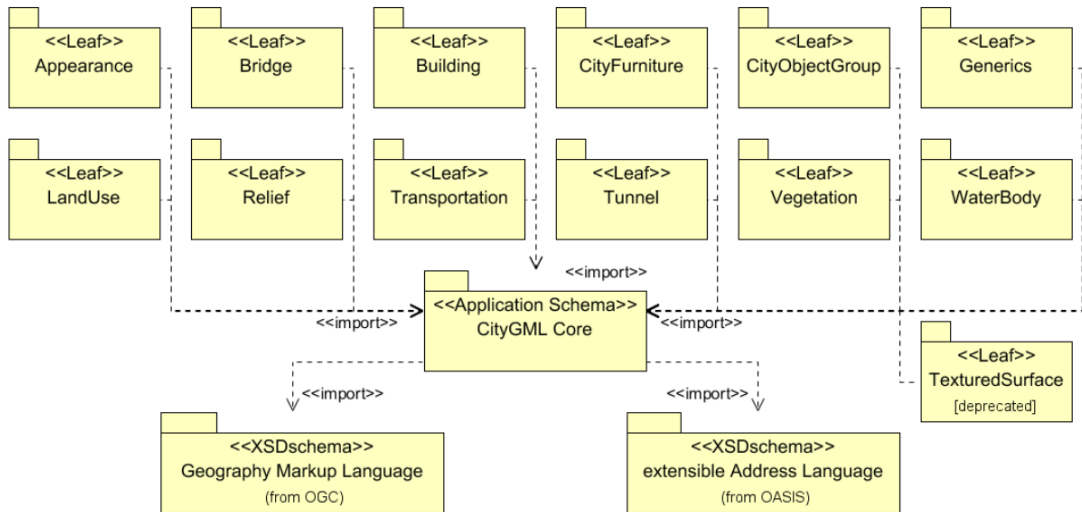


Figure 2.1.: "UML package diagram illustrating the separate modules of CityGML and their schema dependencies. Each extension module (indicated by the leaf packages) further imports the GML 3.1.1 schema definition in order to represent spatial properties of its thematic classes." (Figure from [Gröger et al., 2012])

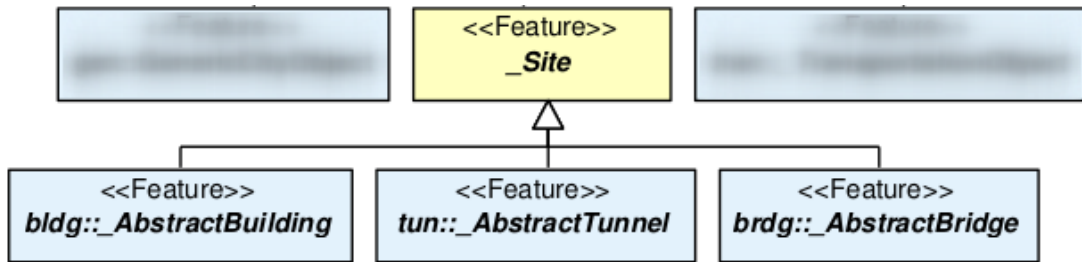


Figure 2.2.: UML diagram of the "site" superclass along with its subclasses. (Figure adapted from [Gröger et al., 2012])

Other than the thematic, CityGML also incorporates a geometry model. According to Gröger et al. [2012], "Spatial properties of CityGML features are represented by objects of GML3's geometry model. This model is based on the standard ISO 19107 'Spatial Schema' [Herring, 2001], representing 3D geometry according to the well-known Boundary Representation (B-Rep, cf. [Hughes et al., 2014])". However, only a part of it is used as a specific profile (Figures 2.3, 2.4).

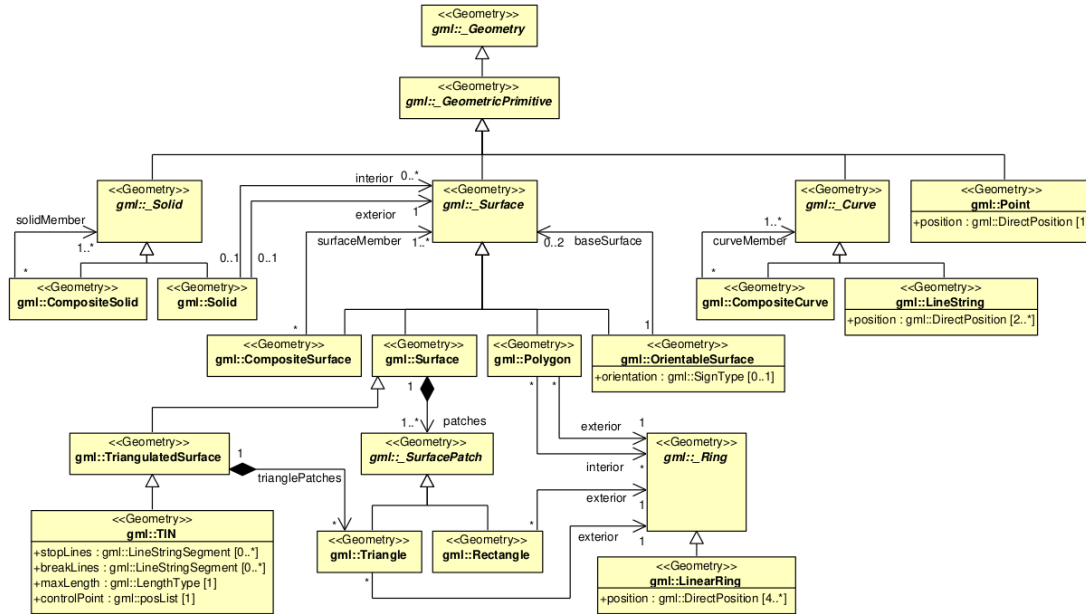


Figure 2.3.: "UML diagram of CityGML's geometry model (subset and profile of GML3): Primitives and Composites." (Figure from [Gröger et al., 2012])

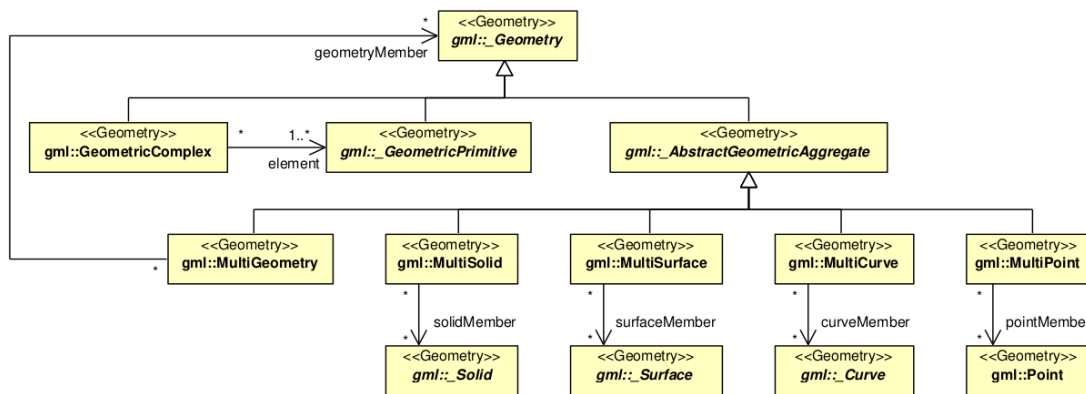


Figure 2.4.: "UML diagram of CityGML's geometry model: Complexes and Aggregates." (Figure from [Gröger et al., 2012])

As the CityGML 2.0 is built on GML3, it is able to support 3D geometries which further reveals options for different kind of levels of representations. These are defined by CityGML as "LOD" and relate to both the thematic and geometry models. CityGML supports at maximum five (5)

2. Related work

LODs as described in table 2.1. That being said, the nature of the object in relation to reality is what dictates the available LODs in the model. To give an example, a “room” feature (building composite) cannot have an LOD lower than 4 as is an interior feature. Moreover, the LODs can also be used for precision of measurement restrictions, however, according to documentation these standards are debatable.

	LOD0	LOD1	LOD2	LOD3	LOD4
Model scale description	regional, landscape	city, region	city, city districts, projects	city districts, architectural models (exterior), landmark	architectural models (interior), landmark
Class of accuracy	lowest	low	middle	high	very high
Absolute 3D point accuracy (position / height)	lower than LOD1	5/5m	2/2m	0.5/0.5m	0.2/0.2m
Generalisation	maximal generalisation	object blocks as generalised features; >6*6m/3m	objects as generalised features; >4*4m/2m	object as real features; >2*2m/1m	constructive elements and openings are represented
Building installations	no	no	yes	representative exterior features	real object form
Roof structure/representation	yes	flat	differentiated roof structures	real object form	real object form
Roof overhanging parts	yes	no	yes, if known	yes	yes
CityFurniture	no	important objects	prototypes, generalized objects	real object form	real object form
SolitaryVegetationObject	no	important objects	prototypes, higher 6m	prototypes, higher 2m	prototypes, real object form
PlantCover	no	>50*50m	>5*5m	<LOD2	<LOD2
... to be continued for the other feature themes					

Table 2.1.: LOD 0-4 of CityGML with their proposed accuracy requirements (discussion proposal, based on: [Albert et al., 2003]). (Table from [Gröger et al., 2012])

The CityGML model contains also enumerations and codelists. Enumerations are defined within CityGML itself (Figure 2.5). They consist of standard valid values for lists that are used in feature attributes. Codelists, on the other hand, are defined outside of the CityGML schema and can be custom made (Table 2.2). For instance, the Special Interest Group 3D (SIG3D) provides and maintains a complete collection of CityGML schemas ready for use. These can be accessed through their server and take on the form of simple dictionaries [SIG-3D, 2012].

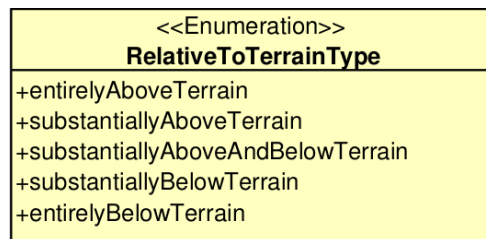


Figure 2.5.: UML enumeration example of “CityObject”’s “relative to terrain” valid values. (Figure adapted from [Gröger et al., 2012])

Code list of the TransportationComplex attribute class			
http://www.sig3d.org/codelists/standard/transportation/2.0/TransportationComplex_class.xml			
1000	private	1050	air traffic
1010	common	1060	rail traffic
1020	civil	1070	waterway
1030	military	1080	subway
1040	road traffic	1090	others

Table 2.2.: Dictionary example of a SIG3D codelist for the CityGML "TransportationComplex" class. (Table from [Gröger et al., 2012])

Lastly, CityGML supports the use of ADEs. Although, this out of scope of this research, it is worth mentioning for future development. The ADE can be used to enhance the CityGML model to accommodate for specific application in the fields of energy, pollution, transportation and more (Figure 2.6). To avoid conflicts with CityGML, ADEs use a specifically defined and different namespace. The benefit with ADEs is that they don't compromise the CityGML's standards allowing both interoperability between systems and working with specialized models.

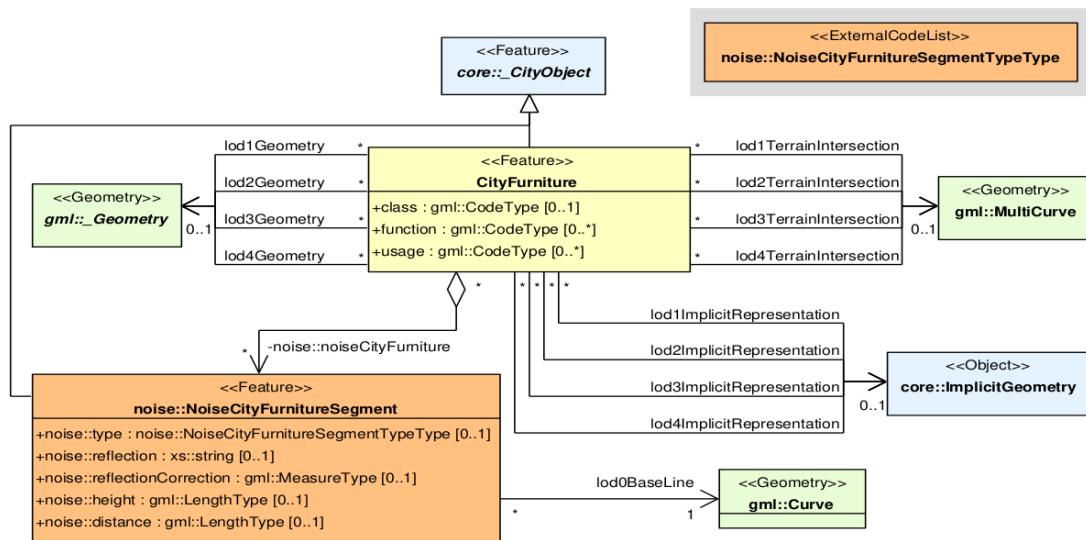


Figure 2.6.: "CityGML noise application schema – city furniture model (light yellow=CityGML CityFurniture module, light orange=CityGML Noise ADE). Prefixes are used to indicate XML namespaces associated with model elements. Element names without a prefix are defined within the CityGML CityFurniture module. The prefix noise is associated with the CityGML Noise ADE (source: Institute of Geodesy and Geoinformation Uni Bonn)." (Figure from [Gröger et al., 2012])

2.2. 3D City Database

As mentioned in the previous section, CityGML is XML-based. However, there are also other types of encodings like SQL and JSON but only SQL is going to be reviewed in this document.

2. Related work

This type of encoding (SQL) is based on [SRDBMS](#) modelling. An [SRDBMS](#) is a database that is extended to be able to manage and analyze spatial data [[Stolze, 2003](#)]. Although, it is used by various software, only the [3DCityDB](#) is in the scope of this research.

"The '3D City Database' ([3DCityDB](#)) is a free 3D geo-database solution for CityGML-based 3D city models. [3DCityDB](#) has been developed as an Open Source and platform-independent software suite to facilitate the development and deployment of 3D city model applications." [[Yao et al., 2018](#)]. Its application is based on spatially enabled PostgreSQL and Oracle Relational Database Management System ([RDBMS](#)), however, only PostgreSQL is used in this research. The database schema is properly structured to accommodate the CityGML model both for storage and processing. Moreover, combined with its software tool "[3DCityDB](#) Importer/Exporter", it adds the capabilities to import/export data and even overview the database contents. On top of that, it is also possible for developers to create and use plugins to enhance the [GUI](#). Lastly, it is also supported by other software that provide web capabilities using Web Feature Service ([WFS](#)) or the "[3DCityDB](#) web-map-client" tool. All of these software are collectively grouped into the [3DCityDB](#) suite.

The practical advantage of working with an [SRDBMS](#) over files like [GML](#) and [JSON](#) is that more often than not, 3D city models occupy large areas and by extension memory size. This can create files of many gigabytes, even for more light-weight encodings like CityJSON. Large files, depending on the system, hinder performance and can even break systems for operations of updates, spatial filters or just data exploration. On the other hand, an [SRDBMS](#) is usually deployed on servers with resource properties to handle heavy loads of transactions. Additionally, these databases support the use of data structures like spatial or otherwise indices and various types of geometries (e.g. through PostGIS). Lastly, they can be easily accessed, remotely or not, by [GIS](#) software like [QGIS](#).

That said, the CityGML data model is significantly extensive and attempting to map its relations one-to-one into a database schema is going to result to a vast number of tables and relations in between. According to [Yao et al. \[2018\]](#), *"a more compact database schema is much more efficient for querying and processing of large and complex-structured data to facilitate good performance when interacting with the database in a real-time application (cf. [[Stadler et al., 2009](#)])"*. Thus, [3DCityDB](#) maps CityGML into a denser database model that reduces operational complexity without introducing semantic ambiguity.

The [3DCityDB](#) is installed by executing the provided installations scripts. These scripts are responsible for structuring the database according to the defined mapping rules in a schema. The default schema is created and named as "citydb", however, it is allowed for multiple schema to be created that could hold different data corresponding to specific scenarios. Moreover, SQL Procedural Language ([PL/pgSQL](#)) functions are generated for use that can facilitate frequently used procedures like deleting features, cleaning the schema in the case of a faulty import and other [[The-PostgreSQL-Global-Development-Group, 2021b](#)].

After a successful installation, users can simply use the "[3DCityDB](#) Importer/Exporter" tool to insert CityGML data (from a CityGML or CityJSON file) into the database. Moreover, this process can be customized by selecting, for example, a specific bounding box or features of the CityGML modules. The software can also generate a database report to get an overview of the data that exist in the database. Finally, the last thing to often do, is to export data from the database into files. These files could be [GML](#), [CSV](#), [XML](#), [JSON](#), [KML](#), [COLL](#)aborative Design Activity ([COLLADA](#)), and Graphics Language Transmission Format ([gITF](#)).

All in all the [3DCityDB](#) suite is an excellent set of tools that manages to reduce the complexity of CityGML, facilitating its use both in desktop and web environments. That being said, regard-

ing 3DCityDB the database structure is still complex with 66 tables for v.4.3 with corresponding foreign/primary key pairs to set the connections between them. Both UML and database schema are extensively and clearly explained into the project's documentation [3DCityDB, 2021a]. However, to work directly with features using custom SQL queries, still it is required to gather all kinds of attributes, geometries, textures, addresses or other from the corresponding tables (Listing 2.1). This is required due to the feature attributes being split over multiple tables, not following the SFM approach. Furthermore, users need to navigate through the complex geometry table in order to get the required values (Figure 2.7). To handle this, a possible solution would be to create custom database functions or views. A view in PostgreSQL, is a named query that is executed every time that it is referenced [The-PostgreSQL-Global-Development-Group, 2021a]. Directly writing SQL statements can be time consuming and error prone, especially for people with little SQL experience. Users of no experience with SRDBMS are simply excluded from using it due to its technical knowledge requirements.

```

1 SELECT
2     ts.id AS roof_id,
3     co_ts.gmlid AS roof_gmlid,
4     b.id AS building_id,
5     co.gmlid AS building_gmlid,
6     b.year_of_construction,
7     ST_Collect(sg.geometry) AS roof_geom
8 FROM
9     citydb.thematic_surface AS ts
10    INNER JOIN citydb.cityobject AS co_ts
11        ON (co_ts.id = ts.id)
12    INNER JOIN citydb.surface_geometry AS sg
13        ON (ts.lod2_multi_surface_id = sg.root_id)
14    INNER JOIN citydb.building AS b
15        ON (b.id = ts.building_id)
16    INNER JOIN citydb.cityobject AS co
17        ON (co.id = b.id)
18 WHERE
19     ts.objectclass_id = 33 AND -- roofsurfaces
20     b.objectclass_id = 26 AND -- buildings
21     b.year_of_construction >= '2015-01-01'::date
22 GROUP BY
23     ts.id,
24     co_ts.gmlid,
25     b.id,
26     co.gmlid,
27     b.year_of_construction
28 ORDER BY
29     b.id,
30     ts.id;

```

Listing 2.1: Example of an SQL statement written to perform a query to extract roof surfaces of buildings constructed from 2015 to now.

2. Related work

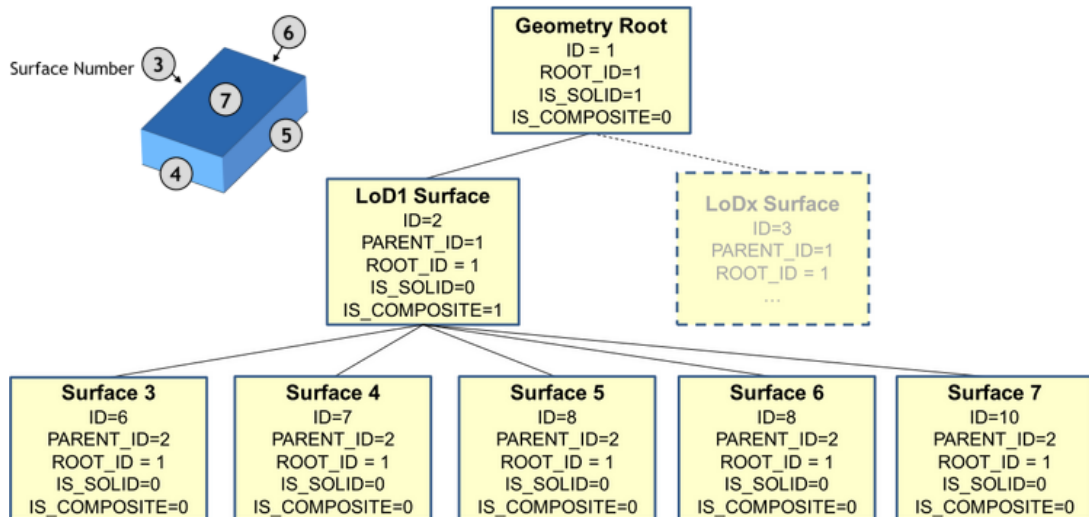


Figure 2.7.: Geometry hierarchy for a solid geometry object [3DCityDB, 2021a].

2.3. 3D City Database "Plus"

A tool that inspired part of this research, is a database schema designed to enhance the 3DCityDB, called the 3D City Database "Plus".

This open source tool was developed by the "Austrian Institute of Technology" as a proof of concept to explore new implementations of features and functionalities of CityGML for the 3DCityDB. Although it was developed for 3DCityDB v3.3.1 and energy ADE, which are out of scope of this research, useful ideas were explored that facilitate database operations. These operations relate to both default 3DCityDB and additional energy ADE tables.

The approach that is of particular relevance for this thesis is the use of database "views". As mentioned before, CityGML and 3DCityDB complexity cause (by design) elements of features to be scattered in multiple tables. This segmentation hinders the use of basic operations rendering data management workflows time consuming and difficult to implement. Views are used to "automatically" bring together all of the feature's elements in one single location. Moreover, custom trigger functions are also used in the view's contents in order to allow "update", "insert" and "delete" operations to cascade into the original tables that the view references [Aguiaro and Holcik, 2017].

Other than views, there are also PostgreSQL functions that are used to automate actions. These are implemented using PL/pgSQL which are often used to perform complex computations [The-PostgreSQL-Global-Development-Group, 2021b]. In this case, such functions are used to insert data into specific tables (Listing 2.2). The arguments of the functions are assigned to named parameters. Moreover, it is possible to omit those parameters that hold default values. Thus, no particular order of argument assignment is required to execute the function, which can be helpful to users and developers [Aguiaro, 2018].

```

1 SELECT citydb_pkg.insert_building(
2 id := 5094,
3 building_root_id := 5094,
4 class := 'Residential',
5 storeys_above_ground := 5,
6 storeys_below_ground := 2
7 );

```

Listing 2.2: Example of calling a function in the 3DCityDB “Plus” to insert a building. [Agugiario, 2018].

The implemented concepts of the last two paragraphs are valuable for this research as they try to simplify complex and time consuming procedures. This approach has many common traits with our overall objective which is the reason why these ideas were further explored and refined. It is important to note, that this software makes also use of a multitude of other functionalities, but are not described here as those are not relevant.

2.4. QGIS

QGIS is an open source GIS application solution that is platform independent and free to use. Its open source nature is what gives the software more support and development opportunities [Corrado, 2005]. As QGIS is a well-rounded and complete application, its use ranges through many industry fields, including cartography, geology, environmental engineering, urban development, risk assessment, real-estate management, architecture, transportation and more [Leidig and Teeuw, 2015]. It allows the use of many different data formats like vectors, rasters, databases and supports a multitude of file encodings. Moreover, QGIS provides a rich collection of processing, analysis, management and research tools. Lastly, it allows users to work on a lower-level (programmatically) using custom functions, scripts, actions and more.

In more detail, QGIS users can view data from spatially-enhanced tables, images for various raster formats, use irregular and regular grid meshes, use OGC Web Services like the Web Map Service (WMS), Web Map Tile Service (WMTS), WFS etc., load CSV, Microsoft Excel Open XML Spreadsheet (XLSX) spreadsheets and more. Moreover, the data can be viewed both in 2D and 3D if possible and also utilize the time dimension in a timeline series. Next, it is possible to explore data, compose maps, create, edit, manage, export, analyse data and even publish maps on the internet. Lastly, but more importantly for this research, is that QGIS allows to extend its functionality by creating and using external plugins.

The next paragraphs describe some of the QGIS’s elements that are important for this research. Also, note that the below description is focused on the QGIS Python Application Programming Interface (API) (PyQGIS). QGIS’s core code is build in C++.

To begin with, the most important element of QGIS is the project instance. To give a clear example, every time that QGIS is initiated from the desktop icon or from a saved project, QGIS runs upon this particular project instance. The project is defined by the class “QgsProject” which is the parent of most other classes (canvas, layers, styles and more) [QGIS-Python-API, 2018e]. The class has many methods that can be used to modify the project programmatically. Note, that many of these methods can also be found and accessed from the QGIS GUI properties (Figure 2.8).

2. Related work

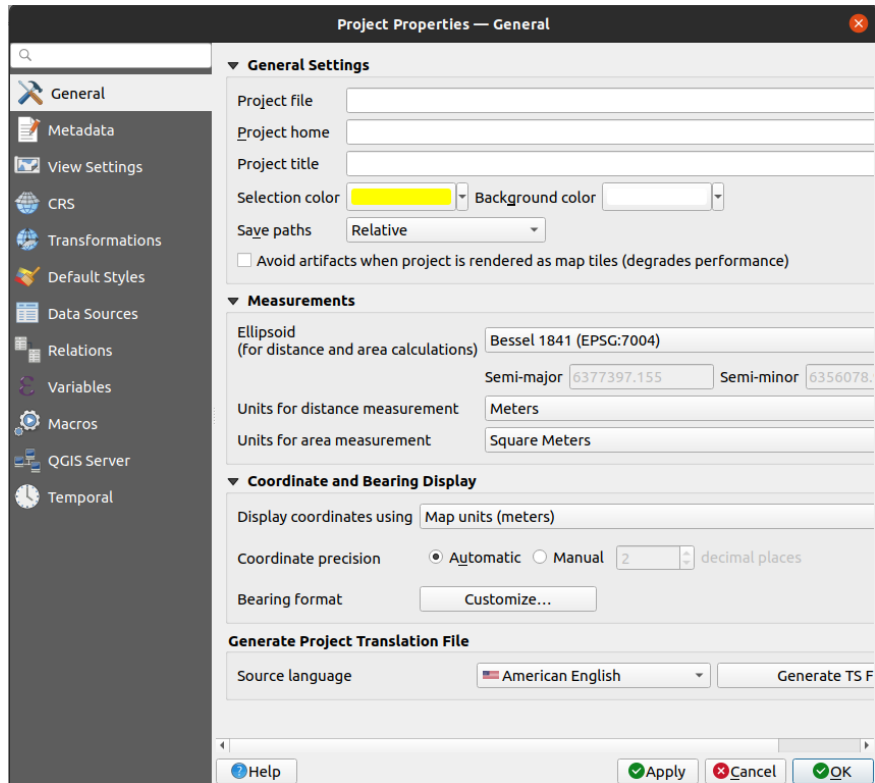


Figure 2.8.: List of QGIS project properties.

Next, the location where the layers and all other QGIS data types are displayed on is called “map canvas” and is defined by the class “QgsMapCanvas” (Figure 2.9). This is a widget type of “QGraphicsView” which is going to be described in section 2.5. The canvas class has methods that relate to the background color, tools to be used on it, annotations, rendering properties, view extents and more [QGIS-Python-API, 2018d]. Many of these methods can be, similarly as before, accessed from the QGIS GUI.

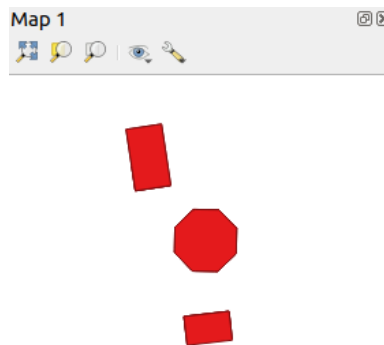


Figure 2.9.: QGIS map canvas widget.

The project's layers in QGIS can be found in the "Layers Panel". This panel can be otherwise called "TOC" and corresponds to a tree of layers defined by the classes "QgsLayerTreeUtils" and "QgsLayerTree" (Figure 2.10). In a default state, all layers are inserted into the root level of the tree, thus it hardly looks like a tree. However, it is possible to create new groups called nodes that can be nested multiple times. Using the TOC, layers can be organized into categories according to relevance which could prove really helpfully in large multi-layer projects [QGIS-Python-API, 2018c].

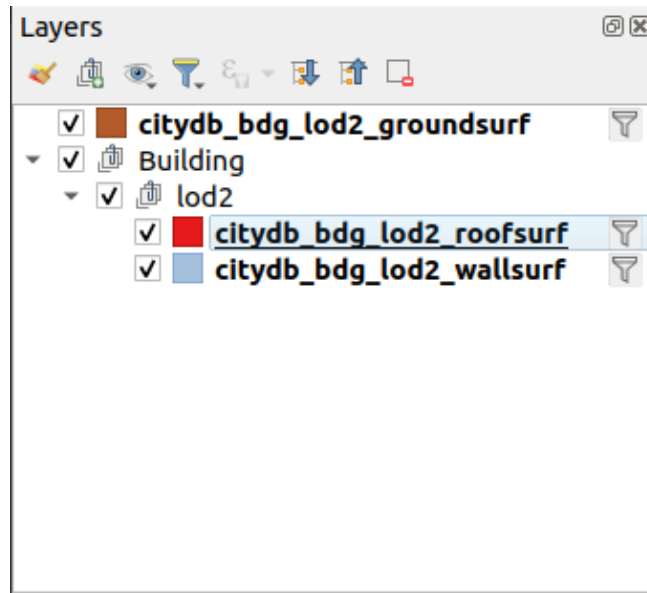
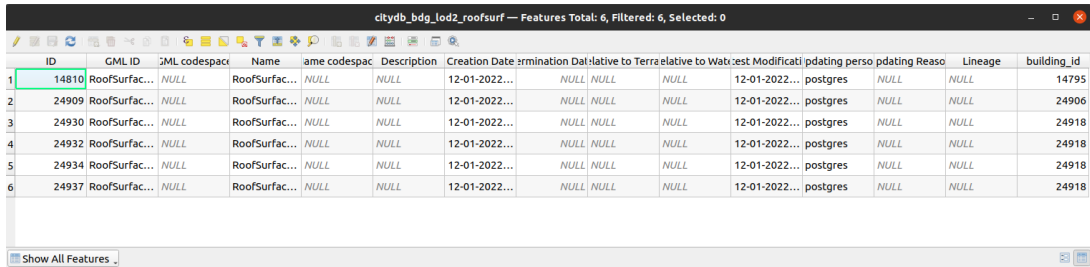


Figure 2.10.: QGIS Table of Contents.

The main types of layer in QGIS are the vector and raster layers. Only the vector layer is in the scope of this research. It is constructed from the class "QgsVectorLayer" and manages vector based data-sets. To create a layer programmatically, it is imperative to define a data provider along with a Uniform Resource Identifier (URI) of the provider's required parameters and a name. For example, a data provider could be "postgres" for PostgreSQL tables and the URI, contains information about the database connection, the table, its primary key, geometry column and even an SQL query to filter data [QGIS-Python-API, 2018g]. Moreover, multiple layers with matching keys could be linked to each other temporarily with direct joins, similar to an SQL table join or by specific project relations. The latter relations are defined by the class "QgsRelation" and can join a layer to another by association or composition [QGIS-Python-API, 2018f].

2. Related work

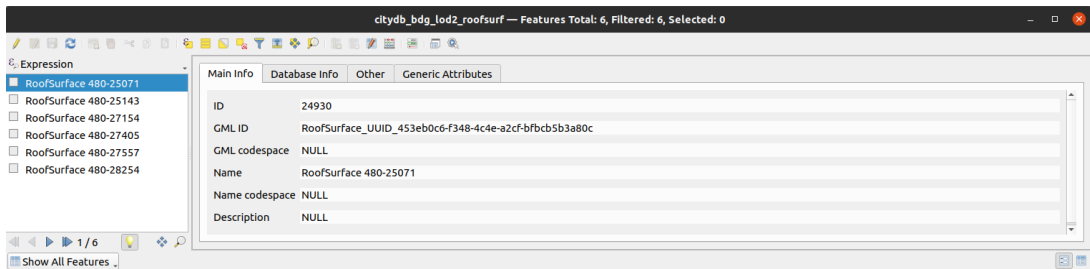
QGIS accesses the data of the layers and shows them in a table called "Attribute table" (Figure 2.11a). While the attribute table stores and displays the raw data as columns in a table called "fields", the "Attribute Form" can be used to modify the style of these fields into a comprehensive form. The attribute form can either be structured automatically, from an embedded designer process or loaded from an external UI file (.ui). These options can be found into the layer's properties under the "Attribute Form" tab. The attribute form can use special widgets to display data in different forms. It is possible to create tabs and "group boxes" to categorize and group fields (Figure 2.11b). It is also possible to display fields in accordance to their data type. For example, a field of date type could be displayed using a calendar, a number using a range, or a boolean using a "check box" and more.



The screenshot shows the QGIS Attribute Table for the layer 'citydb_bdg_lod2_roofsurf'. The table has 14 columns and 6 rows. The first row is highlighted in green.

ID	GML ID	GML codespace	Name	Name codespace	Description	Creation Date	Termination Date	Relative to Terra	Relative to Water	Best Modification	Updating person	Updating Reason	Lineage	building_id	
1	14810	RoofSurfac...	NULL	RoofSurfac...	NULL	NULL	12-01-2022...	NULL	NULL	NULL	12-01-2022...	postgres	NULL	NULL	14795
2	24909	RoofSurfac...	NULL	RoofSurfac...	NULL	NULL	12-01-2022...	NULL	NULL	NULL	12-01-2022...	postgres	NULL	NULL	24906
3	24930	RoofSurfac...	NULL	RoofSurfac...	NULL	NULL	12-01-2022...	NULL	NULL	NULL	12-01-2022...	postgres	NULL	NULL	24918
4	24932	RoofSurfac...	NULL	RoofSurfac...	NULL	NULL	12-01-2022...	NULL	NULL	NULL	12-01-2022...	postgres	NULL	NULL	24918
5	24934	RoofSurfac...	NULL	RoofSurfac...	NULL	NULL	12-01-2022...	NULL	NULL	NULL	12-01-2022...	postgres	NULL	NULL	24918
6	24937	RoofSurfac...	NULL	RoofSurfac...	NULL	NULL	12-01-2022...	NULL	NULL	NULL	12-01-2022...	postgres	NULL	NULL	24918

(a) QGIS attribute table.



The screenshot shows the QGIS Attribute Form for the selected feature 'RoofSurface 480-25071'. The form has four tabs: 'Main Info', 'Database Info', 'Other', and 'Generic Attributes'. The 'Main Info' tab is active, showing the following fields:

ID	24930
GML ID	RoofSurface_UUID_453eb0c6-f348-4c4e-a2cf-bfbcb5b3a80c
GML codespace	NULL
Name	RoofSurface 480-25071
Name codespace	NULL
Description	NULL

(b) QGIS attribute form.

Figure 2.11.: Visual comparison between attribute table and form.

In addition to attribute forms, layers can set their own symbology in regards to their appearance (Figure 2.12). Symbology can include the color, opacity, shape, conditional properties and other. This customizes the appearance of layers which is usually an indispensable requirement of any project.

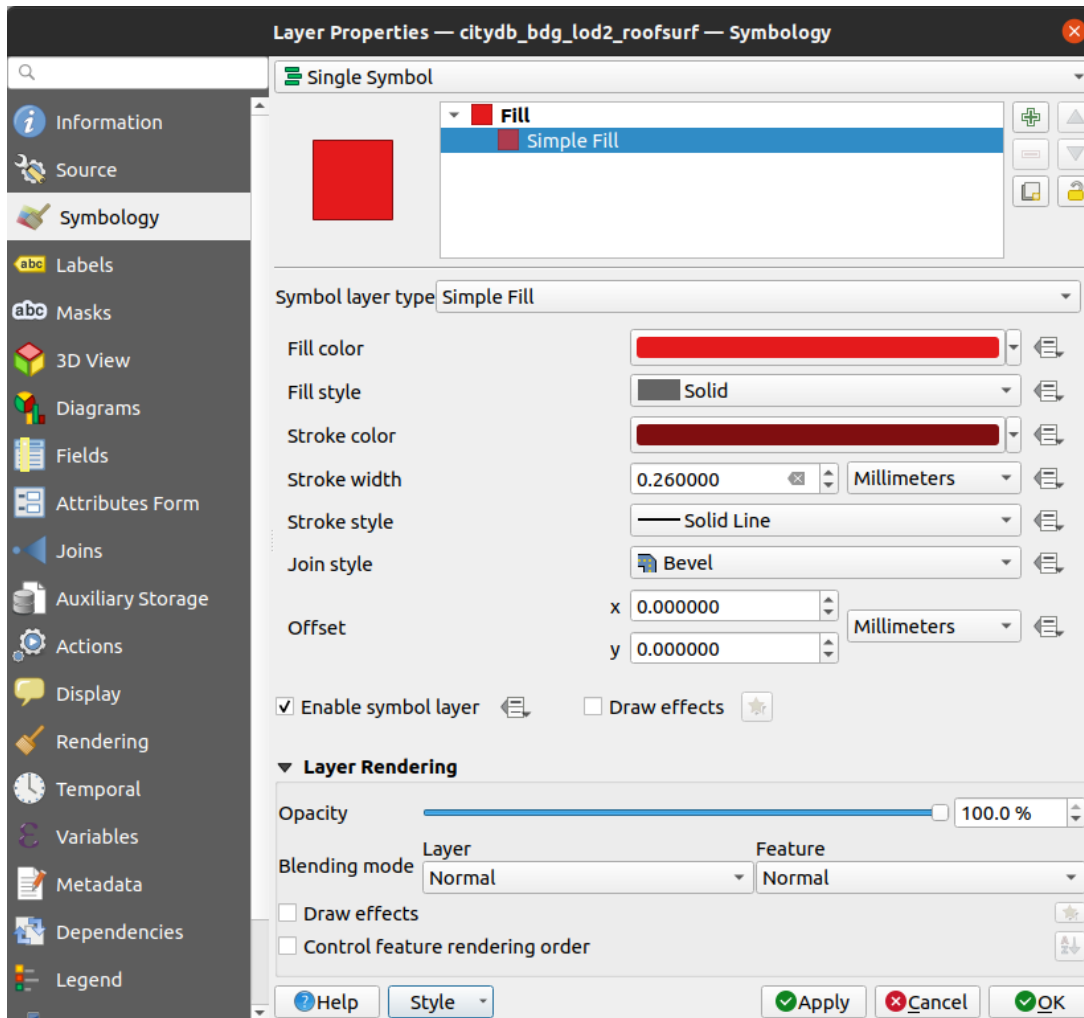


Figure 2.12.: QGIS Symbology properties of a layer.

There are also many more layer properties that are not going to be described as they are out of the scope of this research. All of these properties can be saved as styles. The styles can be saved as Qt Modeling Language (QML), Styled Layer Descriptor (SLD) files or in a PostgreSQL database. This functionality is really important as it allows customization to be created, stored and used in different and/or multiple QGIS project instances [Zipf, 2005].

Finally, QGIS extends its functionalities with the use of plugins. By default, QGIS uses some plugins like "DBManager" for database use, System for Automated Geoscientific Analyses (SAGA) and Geographic Resources Analysis Support System (GRASS) providers for enhanced processing and other options [Passy and Théry, 2018; Neteler et al., 2012]. However,

2. Related work

it also allows the use of external plugins. These plugins are created by developers that identify a problem that could be solved within the QGIS environment but there is no particular implementation at the core project for it. This is also the case for this research. Plugins could be installed by searching through the QGIS plugin repository (in [Pasotti, 2021]), from within the GUI (Figure 2.13) or from the web. Moreover, it is possible to install them either from a compressed ZIP file, or by directly inserting a plugin folder into the QGIS installation directory.

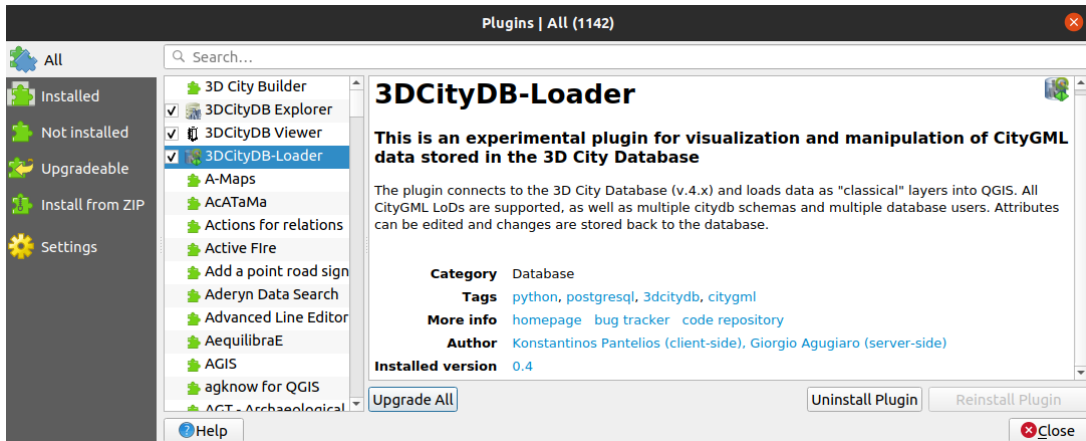


Figure 2.13.: QGIS plugin manager.

2.5. Qt

Qt is a GUI framework upon which QGIS is built. Consequently, this research is implemented using many of its elements. In general, it consists of a platform independent library used in both desktop and mobile applications. Although Qt uses C++, the package PyQt5 provides Python bindings that allow developers to code Qt applications in Python.

The QGIS installation usually comes with the option to install Qt5 Creator and Designer. These are software that help develop, test and debug Qt applications. However, in the context of this research only Qt Designer is used as a tool to structure the GUI of the plugin. The code implementation uses PyQt5 but from within the QGIS environment.

The most crucial for the research elements of a Qt applications are the widgets and the layouts. Widgets are objects that are used for a wide variety of functionalities. As explained in the documentation, *"the widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen."* [The-Qt-Company, 2018]. Windows are also considered widgets, but those don't have another widget as a parent like the rest.

A subset of windows is the dialog defined by the "QDialog" class. Dialogs are top-level windows usually used for short operations. Being top-level also means that it is the element that can hold all other widgets under it. Moreover, it can utilize specific modal properties, blocking or allowing events simultaneously to other windows [The-Qt-Company, 2018b].

One of the most frequently used widgets, is the “Push button” defined by the class “QPushButton” (Figure 2.14). Its practical application is to allow users to execute an operation by clicking on a button. These buttons, in most software are the “OK”, “Cancel”, “Exit”, “Close” but custom buttons with specific text format could be used as well [The-Qt-Company, 2018i].



Figure 2.14.: Qt5 push button widget example. (Figure from documentation [The-Qt-Company, 2018i])

Another, widely used widget is the “Text label” defined by the class “QLabel” (Figure 2.15). This is used as a placeholder for text. It also supports HTML formatted text, images and even animation that could be displayed on it. Moreover, it can allow or disallow users to select the text to be copied or redirect them into a website in case of a Uniform Resource Locator (URL) [The-Qt-Company, 2018h].

A simple rectangular label with a light gray background and a thin black border. The text "Text Label" is centered in a dark gray, sans-serif font.

Figure 2.15.: Qt5 text label widget example. (Figure from documentation [The-Qt-Company, 2018h])

In order to display a list of values, Qt uses the “Combo Box” widget defined by the class QComboBox (Figure 2.16). Upon clicking on the widget, a drop-down list is presented showing all stored values in the minimum possible space. In addition to the list element, it is also possible to provide an accompanying hidden value which is not visible in the GUI [The-Qt-Company, 2018a]. For example, in a combo box storing five random countries, one could insert their populations in order to order them by this metric.



Figure 2.16.: Qt5 combo box widget example. (Figure from documentation [The-Qt-Company, 2018a])

2. Related work

Another important widget is the "Tab Widget" defined by the "QTabWidget" class (Figure 2.17). This widget is used to paginate the GUI in order to help segment and organize the rest of the widgets by relevance. Each tab can hold its own title, icon and properties [The-Qt-Company, 2018j].

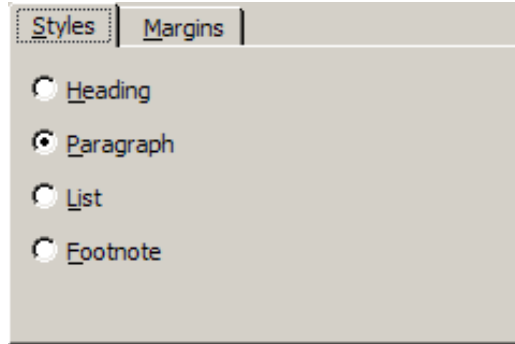


Figure 2.17.: Qt5 tab widget example. (Figure from documentation [The-Qt-Company, 2018j])

To further organize widgets, Qt uses the "Group box" widget defined by the "QGroupBox" class (Figure 2.18). Practically, this is a frame with a title that is used to describe its contents. Moreover, it can have the property to change between an enabled or disabled state. This property is universal to all widgets, however, as this widget can become the parent of other widgets, its state is inherited by those widgets too, regardless of their own state [The-Qt-Company, 2018f]. To clarify with an example, disabling a group box, disables automatically all children widgets of that group box.

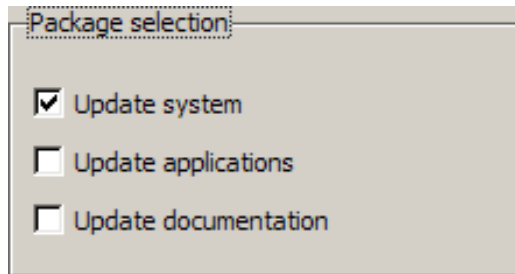


Figure 2.18.: Qt5 group box widget example. (Figure from documentation [The-Qt-Company, 2018f])

Next, the widget called "Graphics View" is defined by the "QGraphicsView" class (Figure 2.19). The QGIS's "QgsMapCanvas" class inherits the properties of this particular Qt class. It is used as a placeholder to visualize graphical items like geographical layers, or regular shapes, map images and other. It also enables users to zoom and pan through the widgets using the mouse commands [The-Qt-Company, 2018d].

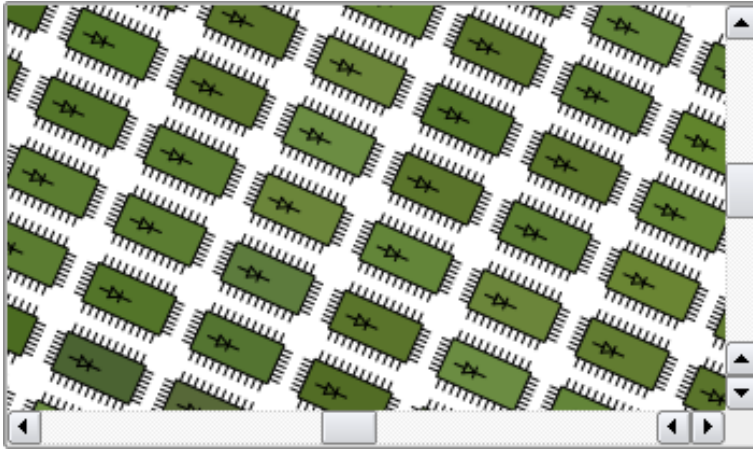


Figure 2.19.: Qt5 graphics view widget example. (Figure from documentation ([The-Qt-Company, 2018d])

Other than the canvas, QGIS has more custom made Qt widgets. One of those is the "Extent group box" which is defined by the "QgsExtentGroupBox" (Figure 2.20). It inherits the properties of a "Collapsed group box" which is another QGIS custom made widget ("QgsCollapsibleGroupBox"). This widget is used to grab and store extent coordinates in the form of N-E-S-W coordinates. It allows users to set those coordinates by manual input, by using the extents from an existing QGIS layer, by using the extents of the canvas or by drawing temporarily a square on the map canvas [QGIS-Python-API, 2018b].

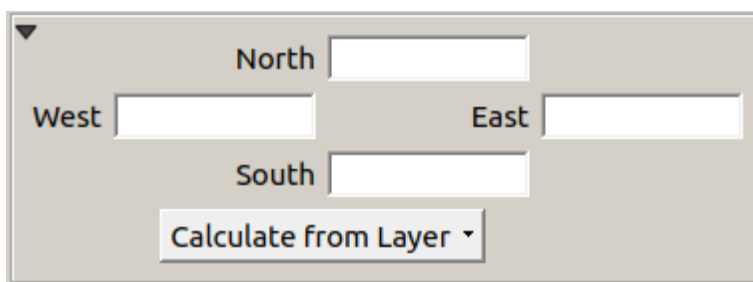


Figure 2.20.: Qt5 extent group box example.

2. Related work

Lastly, another QGIS custom made widget is the “Checkable Combo box” defined by the “QgsCheckableComboBox” class (Figure 2.21). As the name suggests it inherits the properties of the “Combo box” widget with the added value that the elements of the list can have a checked/unchecked state [QGIS-Python-API, 2018a]. This is really useful in cases where multiple elements of the list are required for an operation.

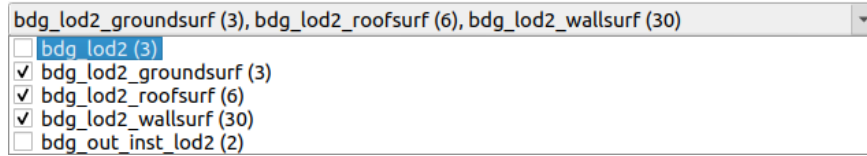
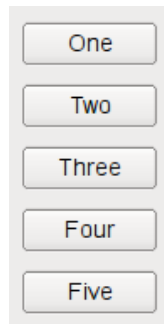


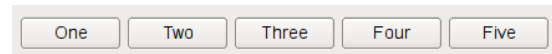
Figure 2.21.: Qt5 checkable combo box example.

Another element of the GUI, is the layout. Layouts impose size and positional properties to the widgets that they hold [The-Qt-Company, 2018k,g,e,c]. In Qt, there are different types of layouts.

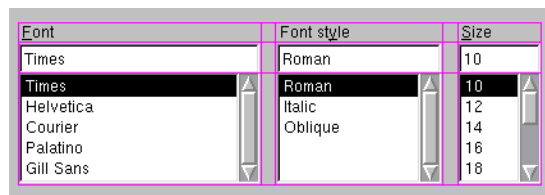
- A vertical layout, orders and spaces widgets one on top of the other (Figure 2.22a).
- On the other axis, horizontal layouts place the widgets one after the other (left or right) (Figure 2.22b).
- Grid layouts are a merge of both vertical and horizontal layouts (Figure 2.22c).
- Form layouts are a custom convenient layout type that uses two columns (left for labels, right for fields) to be used as a form with unrestricted depth (Figure 2.22d).



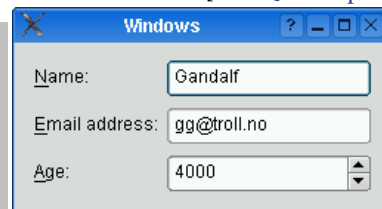
(a) Qt5 vertical layout example. (Figure from documentation [The-Qt-Company, 2018k])



(b) Qt5 horizontal layout example. (Figure from documentation [The-Qt-Company, 2018g])



(c) Qt5 grid layout example. (Figure from documentation [The-Qt-Company, 2018e])



(d) Qt5 form layout example. (Figure from documentation [The-Qt-Company, 2018c])

Figure 2.22.: Qt5 standard layout structures.

In general, a Qt GUI is initiated by executing an event loop that is constantly listening for signals that are assigned to slots (Figure 2.23). Signals and slots are a way of conditional communication between the elements of the GUI. Regarding signals, they have a similar utility to function calls. They can be emitted by the GUI's elements or by custom objects, when a change has occurred. Instantly, a slot is called to execute an operation reacting to the aforementioned change. Slots are connected to signals and are used as functions to execute other operations [Lobur et al., 2011; The-Qt-Company, 2018m].

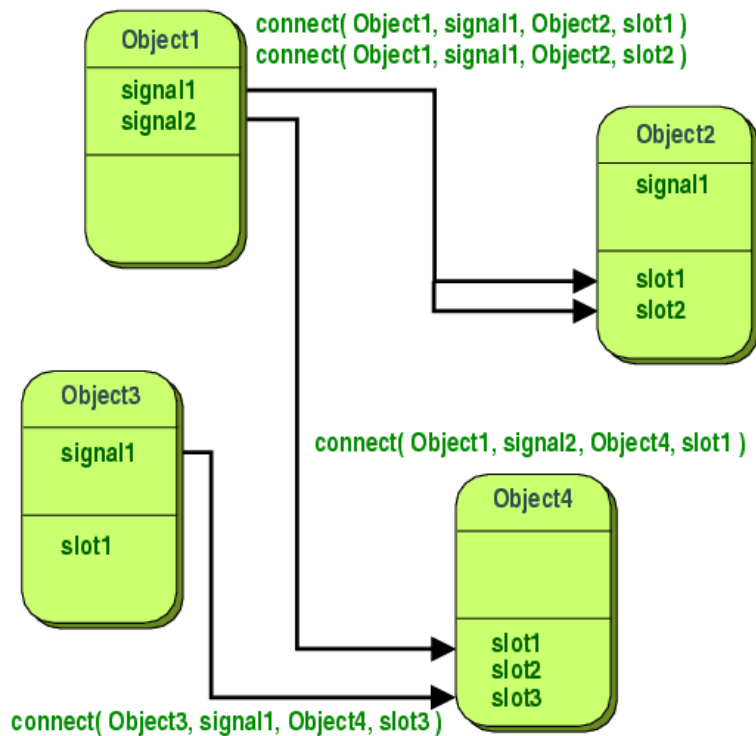


Figure 2.23.: Qt5 slot/signal implementation. (Figure from documentation [The-Qt-Company, 2018m])

Finally, it is important to note that Qt is a framework consisting of many supported versions. Although the newest one is the Qt6 (published in 2021), QGIS environment currently uses Qt5. So, this is the main reason why this research implementation is focused entirely on this version.

2.6. Related QGIS plugins

In this section, a number of existing QGIS plugins are explored to determine their relation to the objective of this research. These plugins were discovered in the preliminary research and were assessed based on their functionality. This was achieved by using the plugins in practice, consulting metadata descriptions and examining other informative documents (when applicable).

2. Related work

2.6.1. 3DCityDB Explorer

A similar QGIS plugin identified in preliminary research, is called "3DCityDB Explorer" and is openly available through GitHub [Casagrande et al., 2021](Figure 2.24). According to the author, it is currently in a development phase with the last update committed on 8th of March, 2021. In short, its functionality is to load data from a 3D City Database and modify the "genericAttribute" attributes of the underlying geometry. Furthermore, it dynamically loads data into the QGIS map canvas using a combination between a maximum number of features (set by the user) and the current extents of the map. The limitations are that it only works for "Building" features represented by LOD2 geometries excluding any composing classes like "BuildingInstallation", "Room", or other. Additionally, the layers do not contain the feature's unique attributes and "CityObject" attributes. Next, it is not compatible for user defined schemas or multiple data schemas, as only the default one ("citydb") is hard-coded to work with. Lastly, the plugin doesn't seem to account for cases of multiple database users with different access privileges.

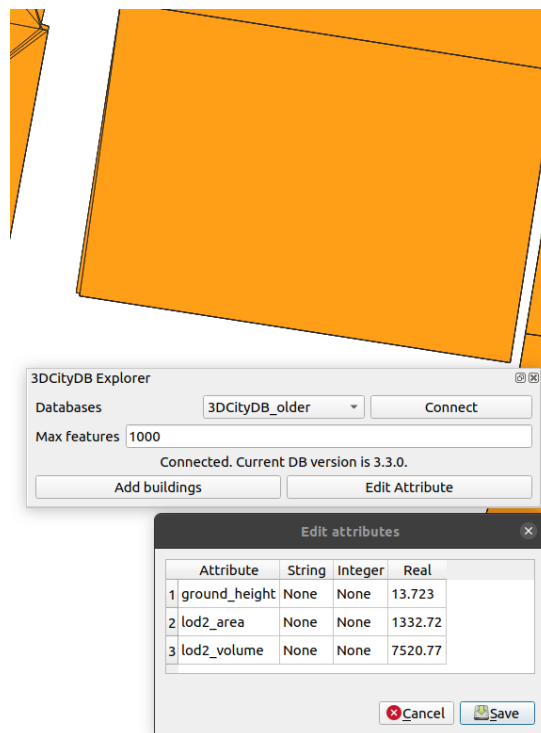


Figure 2.24.: 3DCityDB Explorer implementation example.

2.6.2. 3DCityDB Viewer

Another similar QGIS plugin is called "3DCityDB Viewer" and can also be found in GitHub [Aberham, 2021](Figure 2.25). This plugin lacks any metadata related to its development phase (no guarantee whether is active or not), yet the last update was committed on 16th of June, 2021. The functionality for this plugin is to load data from a 3D City Database. Users can select to load "Building" features based on all geometry levels and types excluding any composing classes. It is important, however, to note that features are loaded from the "citydb" table of geometry, meaning that it is not possible to access the feature's "CityObject", "genericAttribute" and/or unique attributes. Regarding the GUI, from personal experience, the plugin seems complex to navigate which could possibly be a detriment for UX. As a limitation, like the previous plugin, it cannot accommodate for multiple schemas or any other than the default "citydb" schema. Additionally, it does not provide a functionality to handle the amount of data that are loaded. In practice, upon loading a building layer, the plugin attempts to import the entire database which, in case of huge amount of data, can crash the QGIS instance. Lastly, similarly to the previous plugin, it doesn't seem to account for cases of multiple database users with different access privileges.

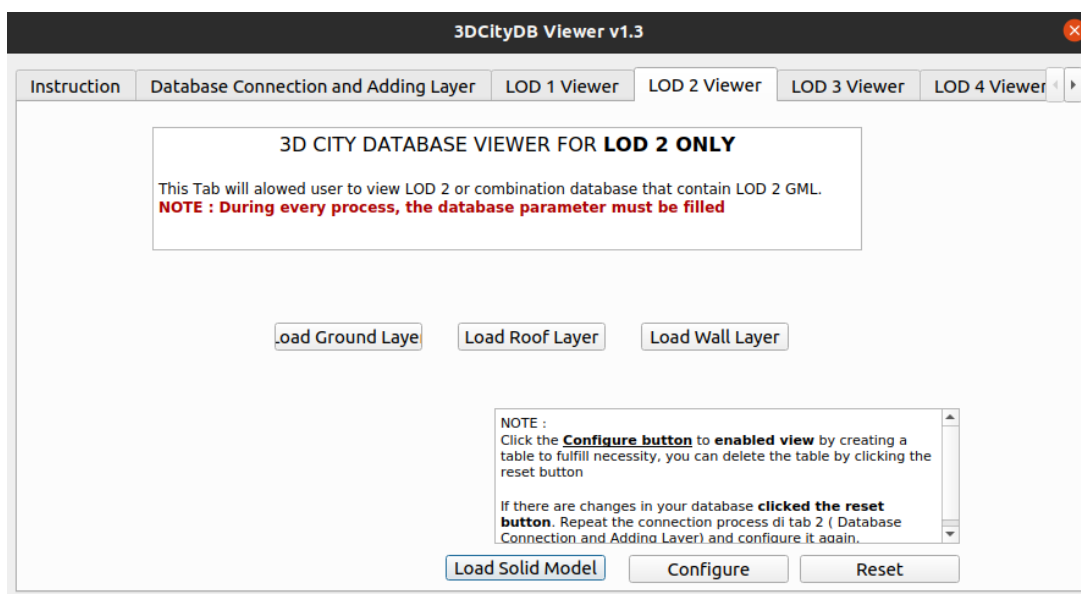


Figure 2.25.: 3DCityDB Viewer dialog.

In summary, both plugins deviate from the fundamental goal of this research, that is to facilitate operations in 3DCityDB models and increase efficiency and productivity for inexperienced users and experts alike. The functionalities are extremely limited and could only be used in very specific situations.

2. Related work

2.6.3. CityJSON Loader

Lastly, it is important to also mention the “CityJSON Loader” plugin (Figure 2.26). This plugin can be found in GitHub [Vitalis and Labetski, 2020] and is actively maintained. Although, it falls outside of the scope of this research, it can be considered to be in the same family, as its goal is to facilitate the use of semantic 3D city models in QGIS.

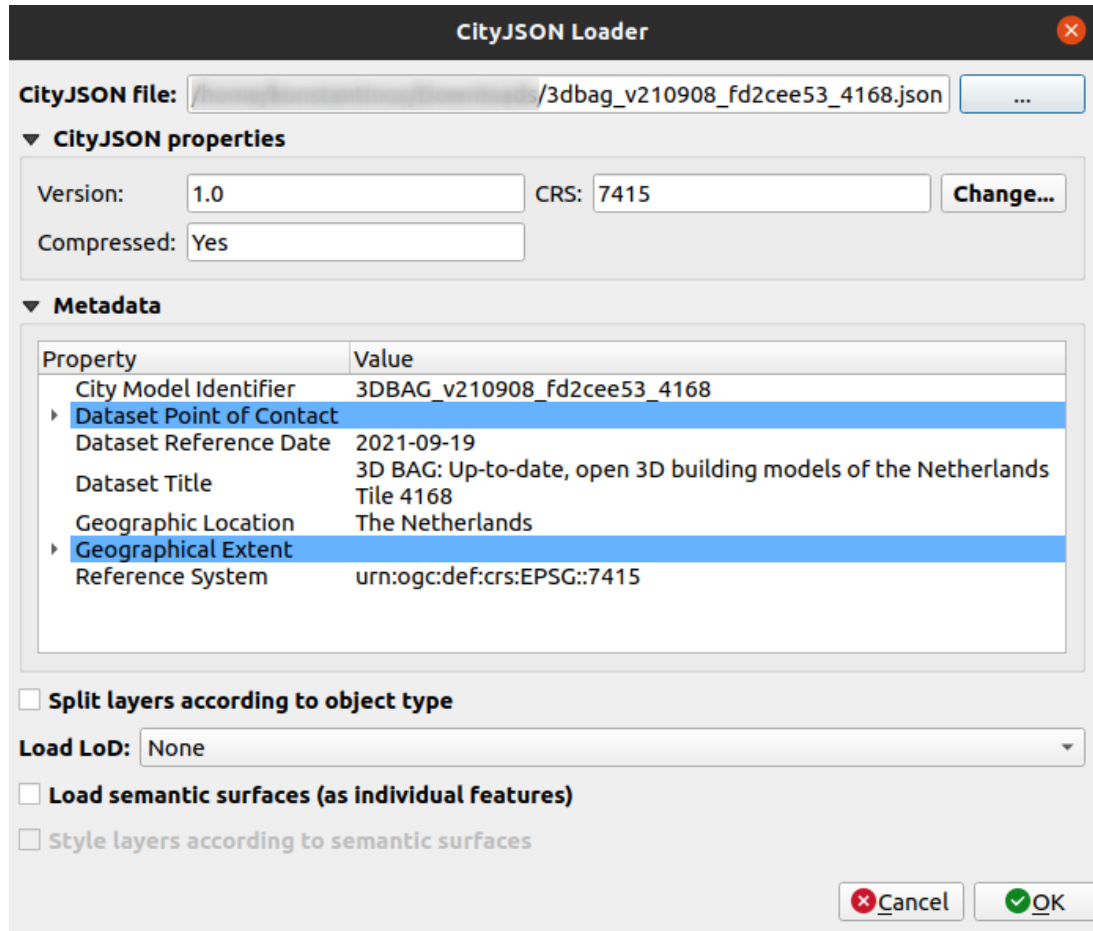


Figure 2.26.: CityJSON Loader dialog.

“CityJSON Loader” is a plugin that allows to import CityJSON files as layers into a QGIS project (Figure 2.27). Users can select CityJSON files from their system to load using different options. Firstly, they could opt-to import different layers corresponding to different feature types. Next, users can elect to either load different layers for different LOD representations, or individual layers with the LOD information being in a field. Moreover, it is also possible to add semantic surfaces as individual layers. Lastly, it allows for a predefined coloring style schema of the semantic layers which can facilitate visual consistency [Vitalis et al., 2020]

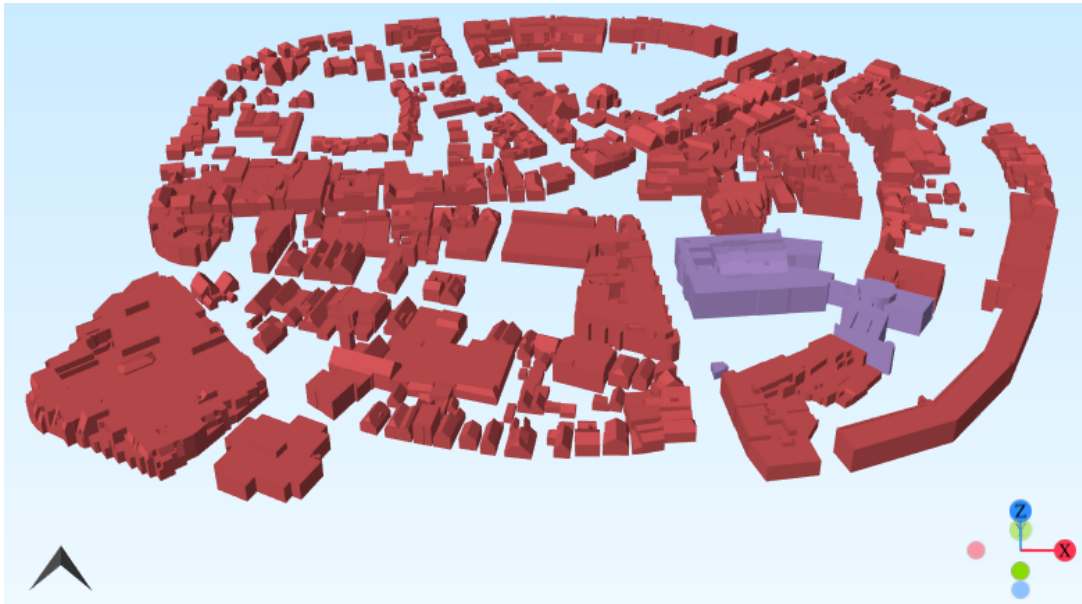


Figure 2.27.: An example semantic 3D city model, imported in QGIS from CityJSON file using the "CityJSON Loader" plugin. (red="Building" of LOD2, purple="BuildingPart" of LOD2)

It is important to note that the plugin loads CityJSON files by converting the data into vector layers stored temporarily in memory. This means that changes happening in the QGIS environment (both in geometries or attributes) are not saved in the original CityJSON file that was loaded. In order to save changes, users need to export the layers as a new file, however, QGIS does not support a CityJSON driver. Consequently, the plugin focuses only on loading data and does not allow for direct data modifications. Lastly, it is not possible to opt to load data only for a particular area. For big CityJSON files, this limitation could cause performance and stability issues for QGIS.

In general, all of the aforementioned plugins, were tried and tested in practice in order to identify the extents of their implementation. Based on this assessment, valuable experience and ideas were procured as a means to drive a more focused and clear development path.

3. Methodology

For this research, an effective and tested method is to create a type of software (plugin) that could answer the research questions. However, concluding from the already existing plugins, well defined and clear goals need to be set, alongside with simple but also complete user capabilities.

It is important to note that the implementation, during the time of the thesis, has followed an iterative process starting from identifying requirements, moving to realization, followed by testing and finally reaching to conclusions. This process was repeated four times, with each time consisting of a version of the plugin. Furthermore, this process was carried out with meeting sessions between the author and all of the supervisors of the research, with individual software development and with testing sessions for the plugin.

Lastly, the client-side of the plugin, which is the focus of this research, is jointly developed with the server-side installation. Consequently, the version updates of the plugin's GUI are linked directly to version updates of the server-side installation.

In short,

- version 0.1 was only responsible for importing layers which is the bare minimum of the functionalities (but nevertheless the final goal).
- In version 0.2 the geometry representation handling was changed and some initial ideas were tested on how to handle user privileges.
- Version 0.3 implemented a multi-user approach and a more advanced way of handling layers for multiple schemas.
- Finally, version 0.4 tackles the important distinction between user types and administrators.

In this document, only the most recent iteration (v0.4) is going to be discussed, as firstly, its the most complete one, solving most limitation of the previous versions and secondly none of the versions are backwards compatible due to significant changes in both GUI design and implementation.

3.1. Primary requirement identification and implementation

These are the most important capabilities of users (and administrators) (Table 3.1). It is considered the core of the plugin, allowing people to overcome most of the aforementioned limitations of 3DCityDB and work seamlessly within the QGIS environment.

3. Methodology

Requirement	Requirement description	Requirement implementation
Database connection	Users should be able to directly connect to a remote or local PostgreSQL 3DCityDB server.	Plugin should follow QGIS 's native approach.
Multi-user capabilities	Multiple users should be able to work simultaneously from the same server.	3DCityDB server should have user-specific work spaces (schemas).
User privileges	Users should have different types of access depending on specific privileges.	Users should be segmented into <i>Read-Only</i> and <i>Read-Write</i> .
Layers structure	Layers should be able to interact with 3DCityDB data.	Layers should be mapped to 3DCityDB data tables following the <i>SFM</i> .
Layers operations	Layers should function as regular QGIS vector layers.	Layers should be created as <i>QgsVectorLayer</i> objects that can load, view, update and delete attributes and/or features.
Plugin structure	Plugin should follow <i>UI/UX</i> design principals.	Plugin should linearly guide the user and communicate using messages and hints.
QGIS structure	Plugin should be able to automatically structure user's QGIS project instance.	Plugin should structure the project's <i>TOC</i> , import additional linked tables and setup layer properties.

Table 3.1.: Identified requirement and implementation.

The main functionalities correspond to user needs that were identified from multiple discussions and assessment between supervisors from both [TU Delft](#) and [VCS](#). Moreover, the requirement identification was held in evenly distributed time intervals throughout the research's span. This aid us to include, remove, modify and refine some of the functionalities that were identified in the initial less-mature stages of the research. It is also important to state the added value in the analysis from the comments, feedback and proposals of the supervisors who have multi-year experience in the field of 3D city models solutions and in particular the [3DCityDB](#) open source project. Lastly, [VCS](#) provided their particular expertise regarding firstly information derived from partly developing and maintaining the [3DCityDB](#) and secondly valuable knowledge acquired from working directly with clients in the commercial space.

3.1.1. Database Connection

The first major functionality is the database connection. While required and expected, this function is the stepping stone to link 3D City DataBases (PostgreSQL) with [QGIS](#). Users should be able to connect to any related database using their credentials (host, port, database, user, password) and work either remotely or locally. In a similar manner, they should also have the ability to disconnect from the database on demand. Additionally, people should be able to create new connections using different connection and user parameters. This also facilitates the multi-user usage of the plugin.

It is important to note that while the plugin allows the creation of new connection, users can and may opt to create connections using the [QGIS](#)'s methods. The plugin's approach is almost identical to the [QGIS](#) approach, meaning that it's also compatible and does not cause any conflicts. Utilizing database capabilities from [QGIS](#) also means that people have the ability to store their connection parameters into the current [QGIS](#) user's settings for future use. That said, doing this, may not be a safe approach as these sensitive information are stored in a local directory unencrypted, meaning that additional safety measures need to be taken that are out the scope of this research. Lastly, the plugin (at version 0.4) does not give the option to users to delete or edit a connection. However, these operations could be easily handled, from within [QGIS](#) itself.

3.1.2. Multi-user capabilities

For individual users who work alone in isolated environments, the implementation is straight forward. However, this is not usually the case, especially for a larger audience (e.g. Municipalities, Organizations). Consequently an important requirement is that multiple users should be able to directly access data from a 3DCityDB server simultaneously.

This aspect of the problem is related mostly to the server-side operations. The 3DCityDB instance is enhanced with multiple user-specific schemas holding elements reserved for specific users. This particular segmentation was decided upon experimentation and contemplation of the ability of multiple people to work simultaneously. By giving access to personal user space it is possible to overcome inevitable versioning conflicts that could arise from simultaneous work on the same layers.

3.1.3. User privileges

In PostgreSQL, similarly to most databases, different users can have different privileges for different use in the database. These privileges are set from the administrator or superuser of the database and dictate what users can do.

This aspect of database use is really important as the plugin is required to create schemas, types, tables, views, functions within the 3DCityDB instance. In practice, this means that a user or users need to have the necessary privileges to make the installation (or uninstallation) happen. Additionally, other users must not have the ability to fiddle with the database installation or sensitive operations. This is the main reason why the plugin should be segmented into "User" and "Administration" use.

Continuing, these valid users and their privileges need to be set by the administrator as either "Read-Only" or "Read-Write" and be stored in a specific user group in the database. "Read-Only" is for users that act as guests and are not allowed to commit any changes to the features in the database, but are allowed to view, explore and analyse the features using QGIS processes. On the other hand, "Read-Write" users can do all of the above and additionally edit attributes or delete whole features permanently.

Lastly, to facilitate the use of the plugin, upon installation, a "Read-Only" and a "Read-Write" user are created as default users for immediate use. Using these default users, a person could bypass the step of having to manually create a new user (with the necessary privileges) to work with.

3.1.4. Layer structure

The layer structure is one of the most important functionalities relating to this research.

As already mentioned, the biggest disadvantage of the current SQL based 3DCityDB structure is the segmentation of features into multiple tables within the database causing the need of big and complex queries (Listing 2.1). This means that it is important that users can access 3DCityDB data directly but in an intuitive way.

To overcome this complexity, each feature is reconstructed to a view following the SFM as defined by the OGC [Herring et al., 2011]. In practice each feature is represented by a database

3. Methodology

view (table structure) where the rows correspond to different features and the columns (fields) correspond to the feature's attributes where one of those attributes is its geometry. Note that for efficiency reasons, the geometry columns are materialized views while the rest of the attributes are contained in views. In particular, querying the geometry table on the fly would take a lot of time, so from several experiments we have decided to pre-generate the geometries. This also allows to instantiate implicit geometries (e.g. trees). Therefore, a layer is the result of linking a view with the attributes to the materialized view with the geometries.

Regarding the geometry, after experimentation and contemplation of preliminary results, we concluded that allowing for multiple geometry representation (e.g. Solid, Multi-Surface) introduces complexity without adding any functionality. Consequently, we decided to offer only one representation per LOD. As an example, LOD1 buildings can be represented in CityGML as multi-surfaces or solids. For the layers, a unique representation is offered by the layer, which is automatically determined by the server-side scripts.

Moving to the QGIS side, the layers are structured as "postgres" (database provider) vector layers (QgsVectorLayer) with direct connection to the database. This means that once a layer is imported into the QGIS project, it can function independently from the plugin, as long as the server is up. The plugin is not responsible to handle the connection to the database, as this is handled exclusively by QGIS and PostgreSQL.

Lastly, inside the scope of this research is to include the "genericAttribute" class accompanying the features. The attributes of the "genericAttribute" class are imported as a regular table layer. In order to automatically link these attributes with the other features' attributes, an associative relation can be set from within the QGIS project that connects them with a cardinality of Many-to-One.

3.1.5. Layer operations

The layers are structured in a way that mainly should allow users to load, view, update and delete but can also make use of other QGIS built-in functionalities.

Loading layers, as mentioned above, should create QGIS vector layers (QgsVectorLayer) stored in the map registry of the current project that the users works on. After loading, the layers can be visualised both in 2D and 3D view for visual exploration. Additionally, the layers should be update-able meaning that they can commit any changes of their attributes into their original 3DCityDB schema and table. This functionality can work with the implementation of trigger functions attached to the layers. These trigger function handle the "update", "insert" and "delete" operations. Note however, that the "insert" operation is currently forbidden as this implementation doesn't handle new geometries. Moreover, the "update" operation is only available for attribute updates, excluding geometries. That said, deleting feature instances is still possible (Section 2.3).

Regarding QGIS operations, the layers should function almost the same as any other QGIS vector layer. In practice this means that layers could be used in various processes and algorithms of QGIS. For example, users may opt to extract a buffer or use other geo-processing tools. They could also export the layers into a different format that is stored in a file or run their own SQL queries on them.

3.2. Secondary requirement identification and implementation

The secondary functionalities relate to UI/UX design decisions. While these do not add features to the plugin usage implementation, a clear and straight forward approach on design is often appreciated by all types of users [Joo, 2017].

3.2.1. Plugin structure

Regarding the structure of the plugin, it is important to be able to add as much concise features as possible, in a clear way that users can follow without relying too much on documentation. That said, to get a full grasp of the plugin, reading the user manual is always recommended.

Software structure

For starters, following the assumption that plugin users already have a bare minimum knowledge on the 3DCityDB software, the plugin follows a similar design approach with the application "3DCityDB Importer/Exporter". In practice, similarly to the aforementioned software, the plugin follows a Top-to-Bottom widget configuration, segmented into tabs according to their broad functionalities (e.g. "User Connection" tab, "Layers" tab). That said, the plugin consists of different dialog windows depending on the use of "User" or "Administration". These differences are explained in more detail in section 3.3.2. Taking inspiration again from "3DCityDB Importer/Exporter", the entire structure of the plugin resembles a form or a function, where the users set their needs (parameters) and based on those the plugin returns results.

Moreover, the plugin contains a specific base-map widget linked to Open Street Map (OSM). This helps users to navigate and know exactly the extents of the data in their database and/or layers. Embedding the base-map inside the plugin help also users to setup new extents without the requirement to manually set the cardinal coordinates.

Additional options

Following the server-side installation, the plugin should provide space for additional options linking to server functions. At version 0.4 (following server-side installation v.0.7.0), only one option could be available, that is to simplify the geometries. The implementation of this function was born from preliminary results, where, in short, QGIS 3D visualization was failing due to coordinates' close proximity. The function allows users to simplify the stored geometries by discarding minuscule polygons based on area and fixing coordinates based on decimal precision.

3. Methodology

User experience features

Regarding design features to enhance UX, a useful approach is to have dynamic data availability on certain filters. In this way, users can be informed on live updates about the exact available Features Types, LODs and specific layers accompanied with the number of entries.

Furthermore, the plugin is structured in a way to communicate with the user either by log messages or by message windows. Another more subtle way of user communication, is guiding the user by enabling and disabling widgets based on the current state/step in the plugin. This method is mainly used to stop users from deviating from the intended path of operations [Järvi et al., 2009]. This can help the users to understand some of the underlying operation and troubleshoot in case of issues accordingly. Specifically for developers, it is an essential tool to use and consult.

To summarize the main objective of the plugin is to have a linear structure in a way that users can clearly understand the order of operations. Having a GUI design that feels intuitive can save time and reduce frustration, thus increasing productivity [Marcus, 1995].

3.2.2. QGIS structure

The QGIS can provide many options and capabilities that cater to different user types. It is prudent to use these options to enhance the plugin's user experience.

The plugin is responsible of automatizing some simple operations into a default state, without restricting users to make their own custom changes and designs. These mainly, relate to an orderly QGIS TOC, relations for the "genericAttribute" table, color symbology and attribute forms for different features.

Continuing with the user experience aspect of the plugin, QGIS itself is used and structured accordingly. QGIS API gives developers the ability to modify almost everything in a project instance. Consequently, utilizing this ability, the plugin is used to modify the project with a default structure that could help users with some simple operations.

QGIS table of contents

For starters, when layers are imported, they are assigned to particular groups in the QGIS TOC called "nodes". These groups are structured according to a well-defined hierarchical tree starting from the root (database) and branching out until reaching the ending leaves (the layers). The layer names themselves can hint on the contents and location of the data, but having a QGIS TOC organizes the project by grouping the layers by database, schema, feature type and LOD. The benefits of this kind of organization can be mostly recognized on large project where more than one layers are imported, possibly from multiple databases or schemas [World-Wide-Web-Consortium et al., 2020]. The "nodes" and layers are also alphabetically ordered with the exception of "Relief" (Digital Terrain Model (DTM)). The "Relief" layers relate to the surface of the earth, so these layers should always stay at the bottom. This is important, as QGIS renders the layers according to the order of the layers in the TOC.

Additional layers

Secondly, with every layer import, the plugin imports also the "genericAttribute" table (if it doesn't already exist in the project). This table is used to associate these attributes through a Many-to-One relationship. QGIS has a relation class that can be used to set, either from the GUI or programmatically, a relationship between fields of referenced and referencing layers. The plugin sets up this relationship automatically in order to link the "genericAttribute" table to the main feature layer. Visually, this relationship is illustrated in the attribute form as a nested table.

In addition to these attributes, the plugin imports codelists and enumeration. For enumerations, a "Value relation" widget is set to fix a list of values (used for "CityObject"'s relative to terrain and water values) in the attribute form. For codelists however, preliminary results showed that given the custom nature of these tables, it is not possible to create a confident automated process. The main reason is that the relations are created based on layer names and key fields. In custom codelist tables both the table names and key names cannot be automatically recognized in all cases, as these values are hard-coded in the current implementation. That said, users can still import manually those tables and create their own relations from within the QGIS GUI.

Layer properties

Lastly, when QGIS vector layers are created (from PostgreSQL views) a random color is assigned to them that changes between different instances of the same layers. Moreover, the attribute form is automatically created based on field data types to corresponding widgets. However, QGIS safely assumes and assigns most fields into the "text edit" widget. This resembles the attribute table structure and doesn't help visually the user. In order to further improve the user experience the plugin sets a fixed coloring schema that is mapped and applied to each layer. Additionally, the attribute form is structured in a way that collects most relevant fields into specific tabs and group boxes. It is important to note that the above layer rules are stored locally in the plugin's installation directory as QML files (not in the database). These QML files are set based on this research's subjective rules. Users, however, should still have access to these files and can make and/or replace the default ones with their own custom made QMLs to cater to their own specific needs.

3.3. Plugin use

By assessing the requirements and their implementation (sections 3.1 and 3.2), the use of the plugin can be segmented into operations relating to the database and operations relating to the QGIS project. The database aspect is the server-side of the plugin, while the QGIS project is the client-side. As already mentioned, this research is focused on the client-side, however it is jointly developed with the server-side. Consequently, it is important to mention both aspects of it.

3. Methodology

3.3.1. Server-side use

The server-side aspect of the plugin was developed simultaneously in a way to be able to be used by the client-side and vice versa. In this research the server-side is referenced as "server-side installation" or "qgis.pkg". As the server-side operations work exclusively in the database, it is possible to operate independently from the client-side of the plugin.

In short, the server-side installation modifies a 3DCityDB instance in PostgreSQL by including additional schemas, PL/pgSQL and trigger functions, auxillary tables, database views and PostgreSQL types (similar to Object-Oriented Programming (OOP) classes).

The server-side installation is responsible for linking all the 3DCityDB data tables into single database views (corresponds to a QGIS layer) following the SFM. This approach simplifies the way of direct interaction with data which means that users can avoid extensive queries like the given example in listing 2.1.

These views are designed to be mostly update-able. In more detail, it is possible to access feature attributes and add, remove or change their values by updating them. It is also possible to delete a view's entry. However currently, by design, it doesn't allow insertion of new entries.

Next, it handles multiple users by creating user-specific schemas. In practice, each user can have a schema with reserved unique views to access and use. Although these views are reserved for users, they can be mapped to the same 3DCityDB original tables.

Moreover, the server-side installation can handle different user privileges. These privileges are set to distinguish between users that can make changes to the data in the database and user that cannot. That being said, regardless of these privilege types, users can make use of a collection of database functions.

The server-side installation structure is explained in more details in section 4.1.

3.3.2. Client-side use

The plugin has two work modes (in different window dialogs) that relate to specific user types. The first, relates to operations for administrators (Database administration) and the second for regular users (User). This distinction is necessary for database privileges reasons.

Database administration

The database administration aspect of the plugin is dedicated to database installation settings (Figure 4.3).

Only superusers, or users with necessary privileges should be able to work within this dialog window. Here administrators are responsible for the plugin server-side installation. Thus they can connect or create and store new connections for the database. The installation is separated between a "Main" single schema and one or more "User" specific schemas. Administrators have the ability to install or uninstall the "Main" installation schema and/or other "User" specific schemas. Lastly, it is important to note that an administrator should

also be able to act as a regular user, having the same options as the regular users (although this is discouraged).

User

Regular users have often less privileges than the administrators but are considered the main focus of the plugin (Figure 4.2).

Users, similarly to administrators, should be able to connect to the database and/or create and store new connections. Based on the database, they should also be able to select the schema (in [3DCityDB](#) the default one is called "citydb") where the data is stored. Additionally, they can view the extents of the data on an [OSM](#) base-map and/or set their own extents on the map. These extents should be used to assign the area in which new layers are to be created. These layers could also be deleted by the user or replaced by creating new ones. Additionally, users may opt-to simplify the generated geometries by selecting the advanced options. Next, users must also refresh the geometries to be always up to date as these are stored as materialized view.

As the layer extents can be different from the data schema's extents, similarly, users can select different extents for the imported layers. Next, users could select from a list of available feature types and a list of available [LODs](#), one or more layers to be imported. Upon import, the plugin remains open allowing users to select and import different layers of the same extents. Even after closing the plugin, upon reopening, the plugin resumes from the last user action. However, note that the action of closing [QGIS](#) resets the plugin to the its initial state. Lastly, after the import of any layer the plugin reaches the final step of its functionality completing its purpose.

3.4. Development Details

3.4.1. Software and tools

In order to realise all of the technical functionalities in this research, various software needed to be installed. First, the entire development was done with software installed on Ubuntu 20.04.03 [LTS](#) x86-64. That said, all software including the plugin itself work the same in Windows and macOS systems. The [QGIS](#), that the plugin is built upon, is version 3.22 [LTR](#). The [GUI](#) of the plugin is designed using Qt Designer with Qt version of 5.12.8. The Qt5 5.12.8 is also used for the main plugin code development in combination with the embedded Python programming language of version 3.8.10.. Regarding server/database locations, the plugin was developed and tested for a local PostgreSQL version 12.9 instance with PostGIS version 3.2.. Moreover, it is also tested in a remote PostgreSQL server of version 10.19 ([TU Delft](#) - 3D Geoinformation group server). Lastly the plugin is developed for [3DCityDB](#) versions 4.x. The "3DCityDB Importer/Exporter" version 4.3.x was used to import CityGML data-sets into the PostgreSQL database.

Other helpful tools that were used for development testing and versioning is the [pgAdmin](#) version 4.25 (used to facilitate database operations) and [Git](#) version 2.25.1 in combination with [GitHub](#) (used for code tracking and collaboration) [[Pantelios and Agugiaro, 2022](#)]. Note

3. Methodology

that the GitHub repository is going to be published after the completion of this research and studies.

3.4.2. Testing

Regarding used data-sets, the plugin was tested on various criteria.

- 1 One almost complete but small and imaginary data-set (Railway) extracted from open online source to test a large variety of features [Häfele et al., 2020].
- 2 One high detailed but small and imaginary data-set (House) to test LoD4 layers [KIT and Campus-North, 2021].
- 3 One medium-size data-set depicting real and fictional data in Rijssen-Holten extracted by the dutch 3D Basisregistratie Adressen en Gebouwen / Dutch Cadastre (BAG) (provided by the supervisors of this research to test a real world scenario).
- 4 One small data-set depicting real data in The Hague extracted from open online source to test a real world scenario from available open data [Gemeente-Den-Haag, 2021].

These come in CityGML format and were imported into both local and remote 3D City Databases using its Imported/Exporter software (v.4.3) (Table 3.2).

Data-set No	Alias	File	Size	Location
1	Railway	CityGML_2.0_Test_Dataset_2022-03-11.zip	17.2 MB	"3DCityDB-Loader" GitHub repository
2	House	FZK-Haus-LoD-all-KIT-IAI-KHH-B36-V1.zip	726 KB	"3DCityDB-Loader" GitHub repository
3*	Rijssen-Holten	RijssenHolten_all_lod.zip	227.2MB	Private directory
4	Den Haag	DenHaag_bdg_lod2.zip	12.9 MB	"3DCityDB-Loader" GitHub repository

Table 3.2.: Data-set overview.

* Due to size constraint, this data-set cannot be stored into the project's GitHub repository. It is stored into a private Google Drive directory.

The plugin is used and/or tested (in different extents) for implementation and debugging purposes by a group of people using different operating systems.

- Konstantinos Pantelios: Linux
- Camilo León Sánchez: macOS
- Giorgio Agugiaro: Windows
- Zhihang Yao: Windows
- María Aparicio Sánchez: Windows

Feedback coming from the testers was used to fix technical issues, but also add new or improved functionalities (from older to new versions). It was also used to further drive the development of the server-side installation. This feedback is being tracked in the project's GitHub repository (Figure 3.1) [Pantelios and Agugiaro, 2022].

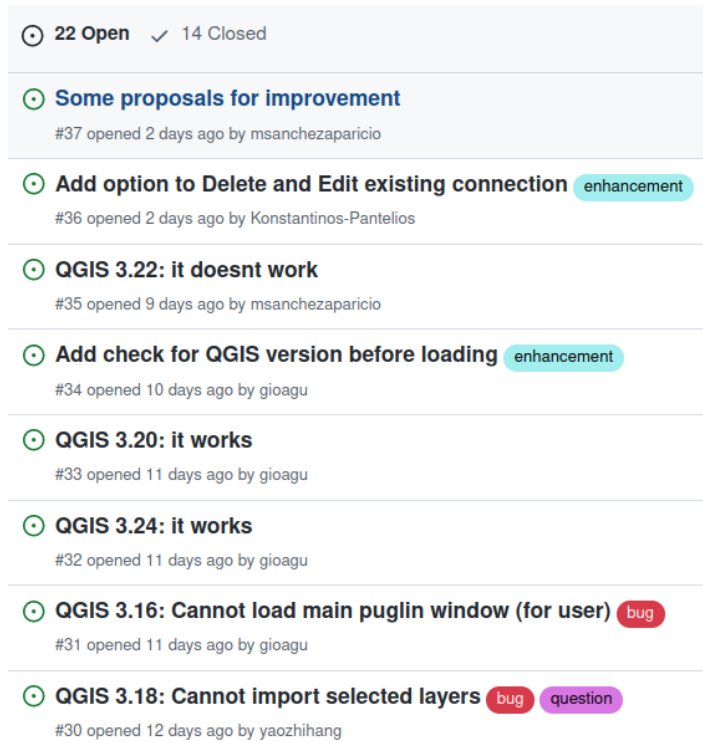


Figure 3.1.: Example of open issues (feedback) in GitHub repository [Pantelios and Agugiaro, 2022].

Additionally, throughout the development of the plugin, external supervisor Claus Nagel along with Zhihang Yao from VCS provided valuable additional feedback and ideas that were taken into account.

4. Plugin structure (server/client-side)

This chapter is a technical and detailed overview of the plugin's functionalities and use of version 0.4. The plugin that is developed in this research, at the time of the writing this document, is titled "3DCityDB-Loader" and is segmented into a server-side and a client-side aspect with the main focus being on the [QGIS GUI](#) (client-side).

4.1. Server-side design

As briefly mentioned in section 3.3.1, the server-side installation implements some of the requirements and the plugin's GUI converts them into a user-friendly no-code form. The plugin's structure is built in accordance to these implementations, thus it is important to provide an overview of the server-side functionalities and structure.

To begin with, the server-side installation tries to tackle the 3DCityDB complexity issue described in section 2.2. In practice, instead of having to work with multiple tables for single features, it makes it possible to use single views (or materialized views) bringing multiple tables together. So in the end, a view for each possible CityGML feature can be created and handled with ease. Moreover, all layers are update-able by means of triggers and functions. PostgreSQL allows regular views to be automatically update-able but those are required to be structured according to specific restrictions [[The-PostgreSQL-Global-Development-Group, 2021a](#)]. Finally, as this tool is developed for the server-side implementation of this research, it is important to also provide an overview of the technical characteristics.

Regarding version 0.7.0, to install it, a 3DCityDB installation must already exist in the database. Similarly, to 3DCityDB's installation, the schema can be created by simply executing an installation script. This results, in three default schemas being created.

The first one, in the context of this research has the alias "Main" schema and is named "qgis.-pkg", the other two are schemas that are called "User" schemas and are named "qgis.user_ro" and "qgis.user_rw".

The main schema holds all of the functions that could be used from users. Additionally, it holds PostgreSQL types for every 3DCityDB table and the trigger functions. The triggers are assigned to views and are listening to "update", "delete" and "insert" signals to perform the corresponding operations. However, only "update" for attributes and "delete" for the whole feature are currently supported. By design, it is not possible to insert new features, but the trigger exists for user communication purposes. Lastly, the "Main" schema also has a number of tables relating to codelists, enumerations, metadata and extents as templates.

The above tables are used in practice, in the "User" schemas. "User" schemas are implementing the multi-user approach, were it is possible to have multiple schemas dedicated to individual users. Using this approach it is possible to avoid conflicts that arise from multiple users working simultaneously with the same layers. These schemas, hold the materialized

4. Plugin structure (server/client-side)

views for the geometry and the regular views for complete features (attributes and geometry).

Materialized views are a type of view that instead of executing on every reference, are refreshed on demand and hold the results physically [Gupta et al., 1995]. This type of view is used for the geometry field to facilitate faster access as operations on it can be computationally expensive. Moreover, the materialized views can be indexed, just like tables, using different indices. In this case, a "B-tree" index is used on the identification column ("co_id"), and a "GiST" index for the geometries column ("geom"). Such data structures are supported in PostgreSQL by default and are also able to be used on geometry data [Pitoura, 2018].

Regarding the feature layers, these are stored as simple views that bring together the unique attributes of the feature itself, its "CityObject" attributes and a single geometry. This structure is called the Simple Feature for SQL (SFS) model, which according to definition, "A feature is an abstraction of a real-world object. Feature attributes are columns in a feature table. Features are rows in a feature table. The Geometry of a feature is one of its feature attributes; while logically a geometric data type, a geometry column is implemented as a foreign key to a geometry table. Relationships between features may be defined as foreign key references between feature tables." [Herring et al., 2011].

Lastly, in every schema, there exist auxiliary tables for specific purposes. These tables relate to the enumerations of CityGML, default SIG3D codelists, various types of extents and the metadata which consist of view details. These tables are also incorporated into the plugin's design and are used programmatically in its workflow.

4.2. Client-side design

The overall design style of the plugin's GUI is based on "3DCityDB Importer/Exporter". The reasoning behind this decision is to provide a familiar GUI to the end-user (Section 3.2.1). That being said, the final product is significantly different as it is structured according to its own requirements.

4.2.1. Plugin Initialization

To begin with, following a multi-user approach with different roles, requires clear distinction between the work space of each user. In this case, there is particular distinction between a regular user and an administrator user. These two different user types are segmented by the design of the plugin to different GUI work locations. Consequently, the plugin offers two menu options (Figure 4.1).

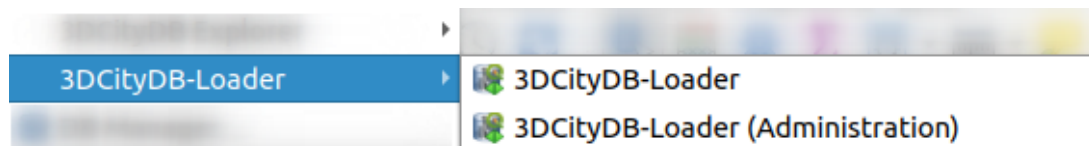


Figure 4.1.: "3DCityDB-Loader" actions in "Database menu".

The first action is used to initialize the plugin for regular users and is titled "3DCityDB-Loader" (Figure 4.2). The title is similar to the overall plugin's title due to the fact that this action is considered the core of the plugin's purpose.

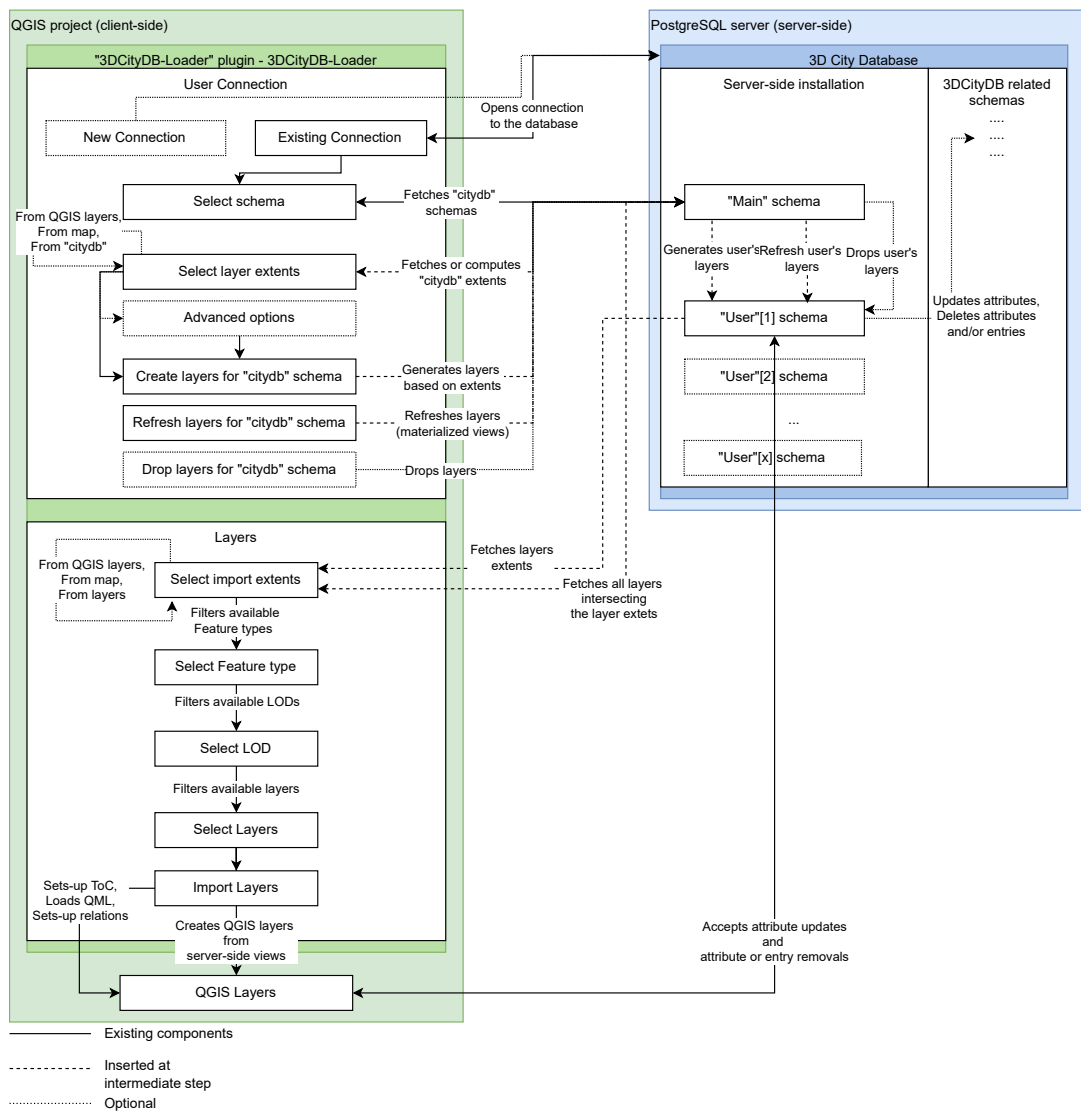


Figure 4.2.: "3DCityDB-Loader" User pipeline.

The second action is used to initialize the plugin for administrator users and is titled "3DCityDB-Loader (Administration)" (Figure 4.3). It is related only to the bare minimum database settings (installing/uninstalling necessary schemas). This action is to be carried out only by the database administrator with superuser privileges.

4. Plugin structure (server/client-side)

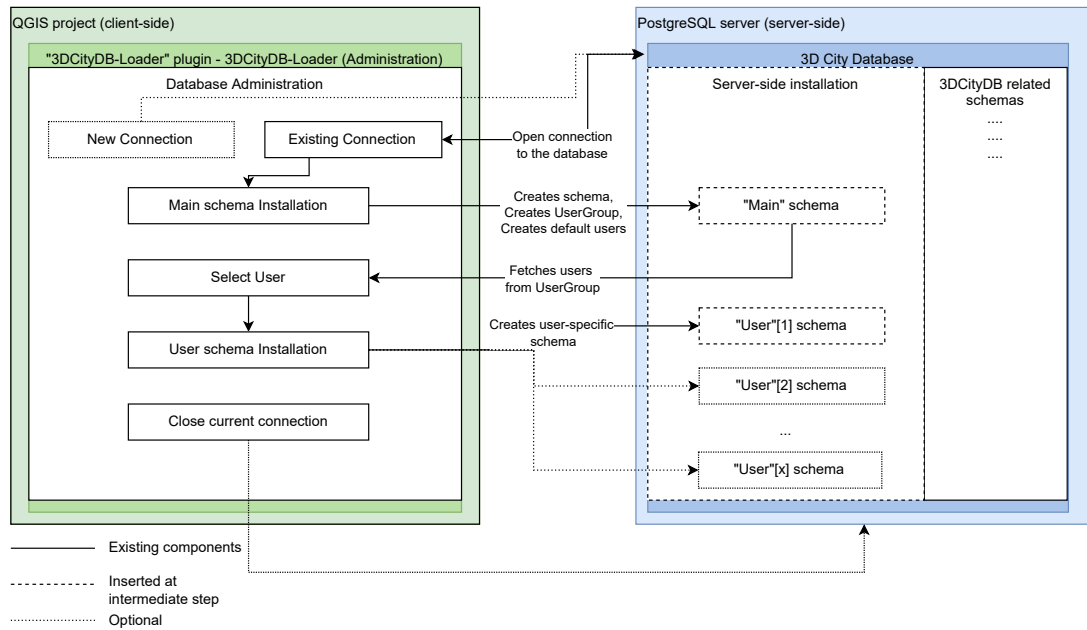


Figure 4.3.: "3DCityDB-Loader (Administration)" Administrator pipeline.

In QGIS, these actions are located into the built-in "Database" menu, inside a custom made plugin menu (Figure 4.1). Also, these actions are available from the "Database" toolbar as clickable icons as well.

4.2.2. Plugin GUI

Regarding the GUI structure, the software "Qt 5 Designer" was used to setup the structure, align the layouts and place the widgets.

Layouts are objects that are used to not only store widgets, but arrange them in a particular order inheriting also size properties. Thus, layouts are used in the plugin as a way to group widgets by relevance or function. Additionally, the widgets are further grouped utilizing the "Tab Widget" that allows the construction of paginated containers used for specific functionalities (Figure 4.4).

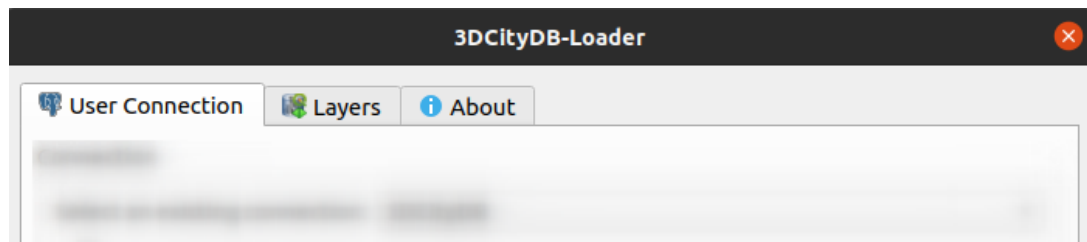


Figure 4.4.: "3DCityDB-Loader" tab widgets (from "User" dialog).

It is important to note that the GUI, is constructed from a UI file (XML-based .ui), but the slots and signals are set manually in the main code project. This approach was required as while the software "Qt 5 Designer" allows for slot/signal pairing, it is only relevant for events between the existing widgets' methods. In practice the plugin uses signal to call events that execute custom operations to serve its needs. For example, pressing a button to connect to a database, is an event that cannot be predefined in the initial UI file.

Lastly, the plugin makes use of enabled or disabled widget, in order to force a particular order of operations. Disabled widgets are grayed-out producing a clear visual cue of unavailable space in the plugin (Figure 4.5). In detail, the tabs are initialized with all but one widget group disabled. This acts as a hint for the steps that users need to follow. Usually, upon a successful pass of a step, a new widget or widget group is enabled revealing the proper way to continue forward. Consequently, users should not feel confused or lost and, as a result, it should be harder to unintentionally cause the software to fail.

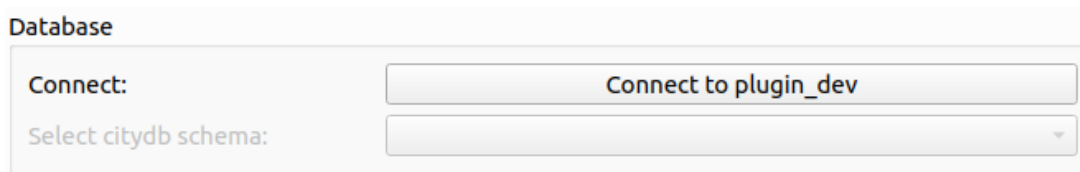


Figure 4.5.: Enabled/Disabled state of Qt5 widgets.

4. Plugin structure (server/client-side)

4.3. Administration dialog

The Administration dialog “3DCityDB-Loader (Administration)” is targeted towards database administrators (superusers) that need to setup the database appropriately (Figure 4.6). Here, only administrators or users that have specific privileges can execute operations. The functionalities are restricted from the server-side, meaning that regular users are met with the corresponding accessing errors. For the purposes of this research, administrators are those users that make use of the administration dialog.

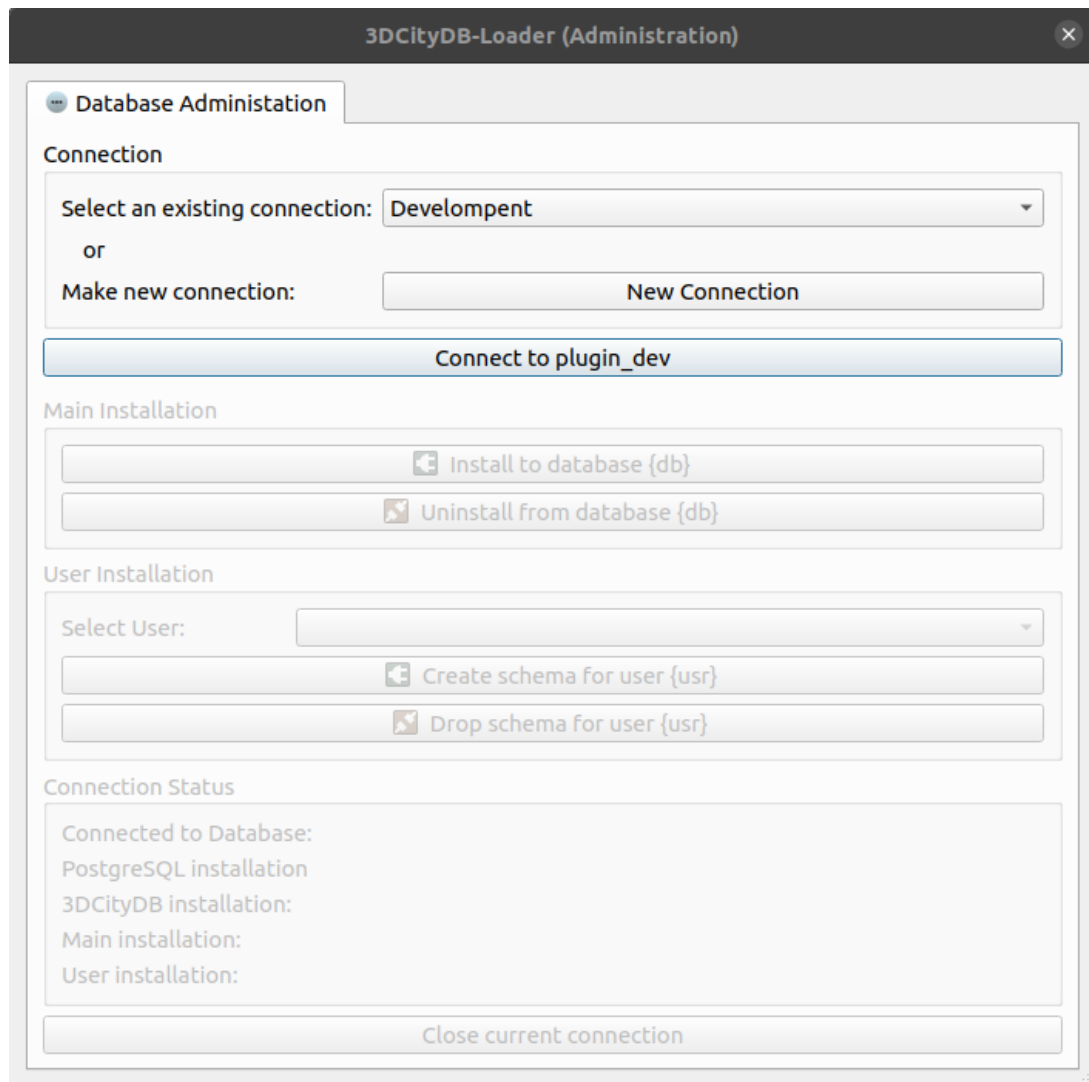


Figure 4.6.: “3DCityDB-Loader (Administration)” Administration dialog at its initial state.

The design of this dialog is similar to the “User dialog” (described in section 4.4) following the top-to-bottom approach with widgets categorized by relevance into group boxes. At v0.4

of the plugin, although only one tab exists (“Database Administration”), the tab structure remains to accommodate more tabs that could satisfy future development requirements.

This dialog’s window is set to modal mode so that signals are blocked for other QGIS applications. This restricts administrators from working simultaneously with both the administrator and user dialog.

Structure-wise this dialog inherits the properties of the user dialog that were extensively described in section 4.4.

Lastly, similarly to the user dialog, closing the window doesn’t turn off the plugin. However, what it does differently is that it closes any currently open connection to the database. Although, there is an option to close the connection manually, closing it automatically is a fail-safe in the case that administrators forget to do it. Keeping the administrator’s connection open after closing the dialog doesn’t provide any valid advantage as the dialog’s state is not important to be temporarily saved.

4.3.1. “Database Administration” tab

The only tab of the administration dialog is the “Database Administration” tab which is linked to installing/uninstalling operations (Figure 4.6).

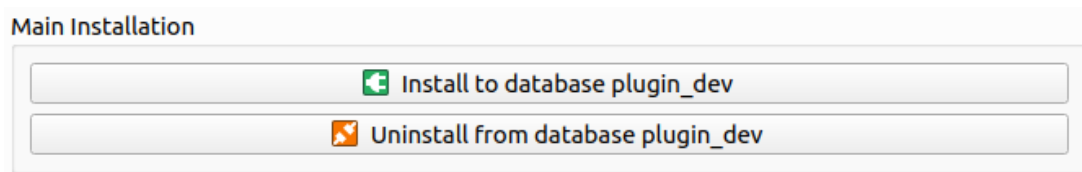
As a first step, administrators need to connect to a database by either using one of the stored database connections or a new one. The implementation here is an exact replica of the “Connection” group box in the user dialog (Figure 4.9b). To avoid excessive repetition, the process is going to be explained in section 4.4.1.

Continuing to the next step, after a successful database connection, administrators can install and uninstall the QGIS package (Figure 4.7a) The installation is done by injecting a number of SQL scripts directly into the database. As the scripts relate to different steps in the installation (creating tables, types, functions etc), plugin uses a progress bar to inform the user of how the process is moving and which script is being installed. The scripts are stored into the plugin’s installation directory in QGIS plugins.

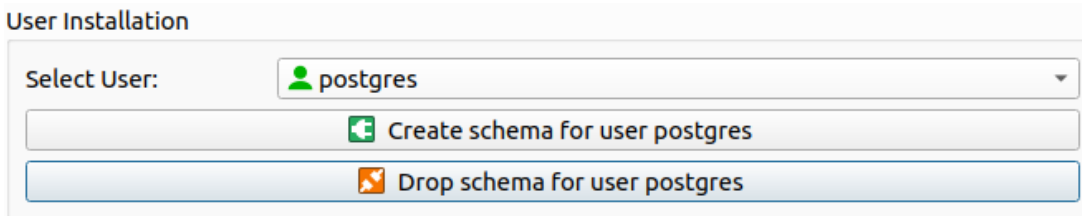
Note that uninstalling the main schema, not only drops the core schema from the plugin, but cleans as well the database from any other contents that the plugin may have already added.

A successful installation of the main schema leads to two results. First, the “User Installation” group box is enabled to allow administrators to install and uninstall the schema for particular users (Figure 4.7b). Secondly, the main installation generates a default user named “qgis_user.ro with “Read-Only” privileges and a default user named “qgis_user.rw with “Read-Write” privileges.

4. Plugin structure (server/client-side)



(a) "3DCityDB-Loader (Administration)" "Main" installation options.



(b) "3DCityDB-Loader (Administration)" "User" installation options.

Figure 4.7.: "3DCityDB-Loader (Administration)" Installation options.

It is important to note, that the plugin can be used only for users that exist in a specific user group ("qgis_pkg_usrgroup"). Consequently, if administrators want to make use of the plugin for existing users, they need to grant them the privileges manually. This could be done by executing the function "grant_qgis_usr_privileges" directly in the server (from PostgreSQL Interactive Terminal (PSQL) or pgAdmin). The available privileges, as hinted from the default users, are the "Read-Only" as "ro" and the "Read-Write" as "rw". Additionally, administrators can choose to grant the aforementioned privileges for specific "citydb" schemas (data schema). This approach can facilitate security and coordination in multi-user and multi-project applications.

In the user selection combo box, all users that are members of the "qgis_pkg_usrgroup" group are presented in the form of a list that administrators can choose from. For the selected users, it is possible to install a user-specific schema, into which the users can generate their own layers on their own extents. Additionally, it is also possible to uninstall this schema, which practically drops the schema from the database along with all contents.

Similarly to the "User Connection" tab in user dialog, at the bottom of this tab, administrators are presented with the "Connection Status" group box. Its function is the same regarding the database connection and installation, although the "Schema Support" and "Layer refresh state" are missing. On successful or failed checks, the labels dynamically change their message likewise. A detailed explanation of this part is given in section 4.4.1.

Lastly, although the dialog is set to automatically terminate any open connection upon closing, a push button to close the current connection still exists. With this button, administrators have the ability to close the connection at any point without having to close the window. Moreover, closing the connection that way resets all of the dialog's widgets at their initial state.

4.4. User dialog

The core of the plugin's functionalities are located and can be executed from the User dialog named "3DCityDB-Loader" (Figure 4.8).

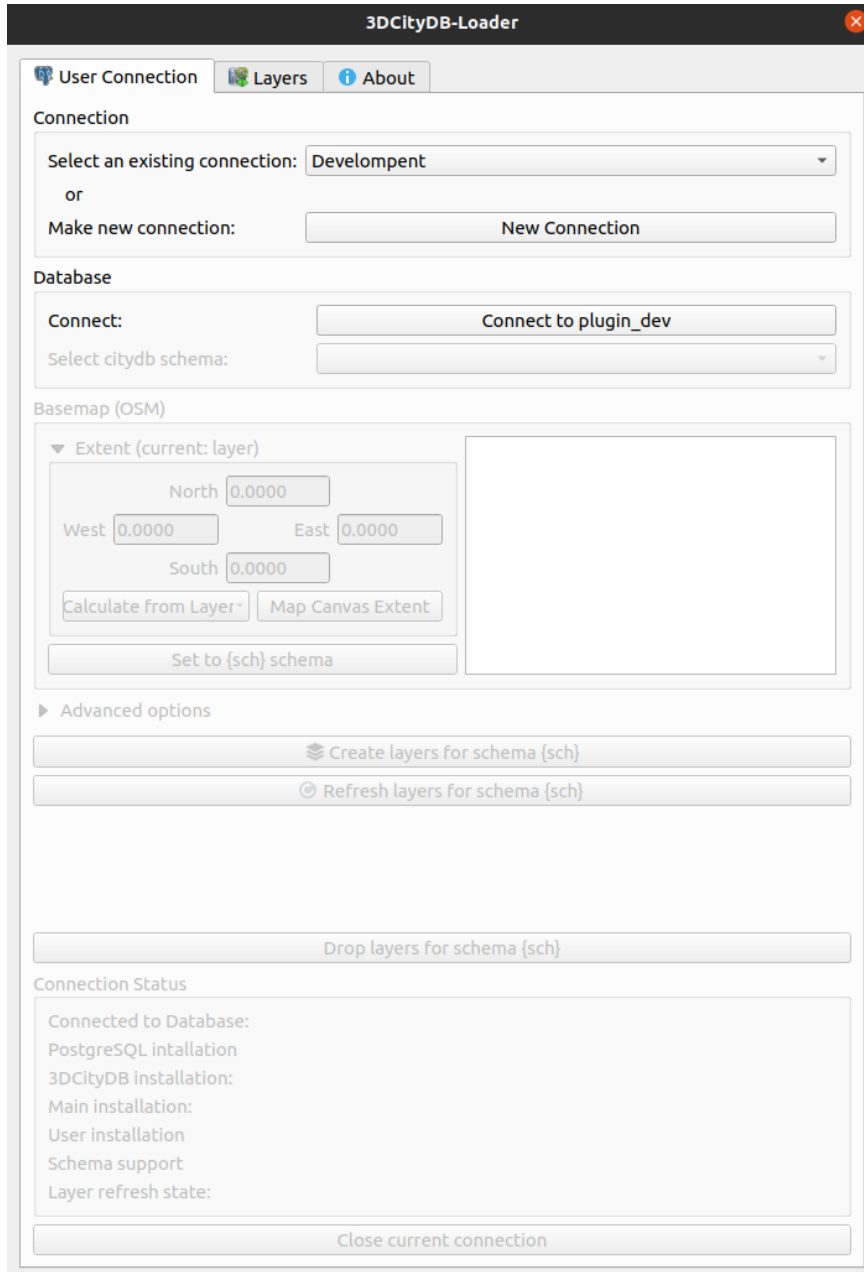


Figure 4.8.: "3DCityDB-Loader" initial GUI state ("User" dialog).

4. Plugin structure (server/client-side)

The user dialog is responsible for the intended use of the plugin from regular users. Widget tabs are used to categorize the functionalities of the widgets in regards to similarity. These tabs are the "User Connection", the "Layers" and the "About" and are going to be described in the following sections (Figure 4.4).

The dialog window is set into a non-modal mode so that signals are not blocked for other applications. In practice, users are able to execute functions outside of the plugin (use QGIS GUI), even if the dialog of the plugin is open. This mode of operation allows multi-tasking without introducing any issues from random order of operations.

Additionally, structure-wise, the dialog window has specific size constraints. Particularly, its horizontal axis (width) is fixed and cannot be resized due to the fact the widgets and all of the information contained are properly fitted without the need of any newlines. Its vertical axis (height), however, can be only expanded until a minimum size is reached that can fit all of the contents properly. This is necessary, as widgets like the QGIS custom widget of "Collapsible Group Box" ("QgsCollapsibleGroupBox") can be in an expanded or collapsed state, which changes the widget's overall height. Consequently, the vertical axis can be resized manually (user drags the window's boundaries) or automatically (user expands a widget).

Note, that these dialog policies are not dependent by any means to the objective goal of this research, but are fundamental properties of any plugin or GUI related software. This also means that future versions of the plugin that could introduce more or less widgets and overall requirements, are going to come with changes into the size policies as well.

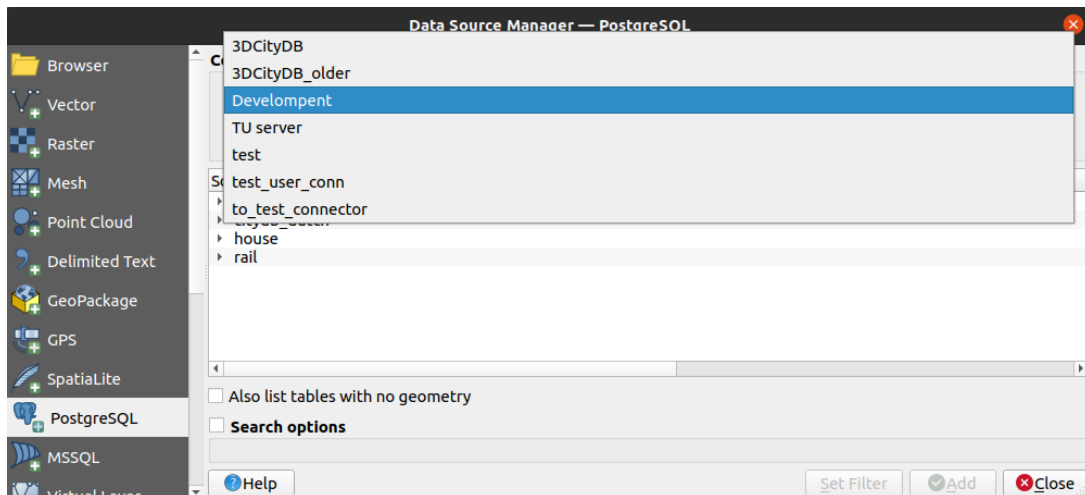
Lastly, an operational design choice that is made, is the closing behaviour. A dialog window gives users the option to close it from an embedded "X" button at the top-right corner of the window. However, in this case this button only causes the plugin's window to close, but the plugin itself is not turned off. Users can reopen the plugin from either the Database menu or the Database toolbar and resume from the same state that it was when it was closed. This is considered helpful and can save time in the case that users decide to use it again for an additional import from the same database, schema, area of interest that they are already working on. The plugin will reset manually, either by opting to close the connection to the database or by reloading the plugin by disabling it and enabling it again from QGIS's plugins list (Figure 2.13). It also resets automatically in every new project instance, as in the initialization process of a project, QGIS loads the current profile's settings anew including all installed plugins.

4.4.1. "User Connection" tab

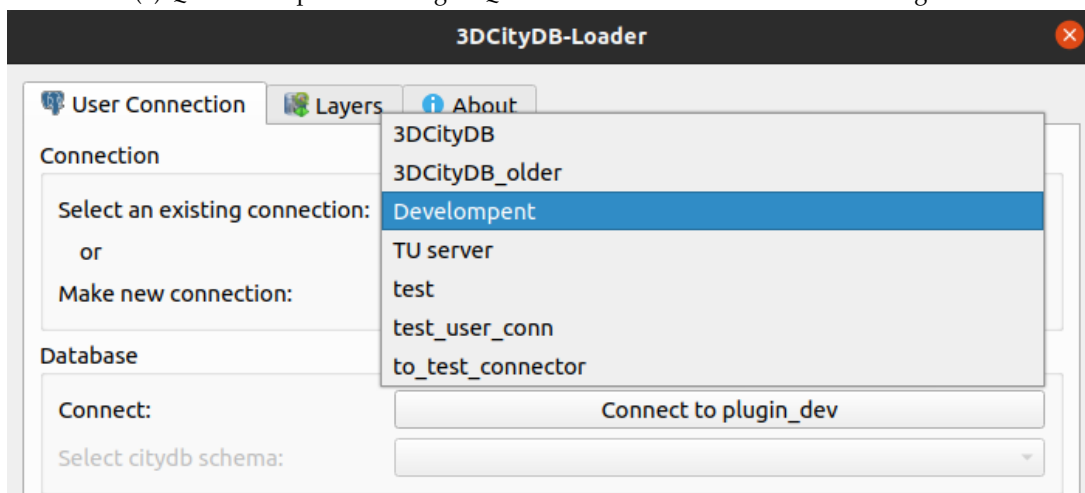
The first tab that users must fill/complete is the "User Connection" tab and consists of operations to prepare the user's working environment (Figure 4.8).

The initial state of the tab is disabled, except of the "Connection" group box, hinting users that before anything else, selecting the database to connect is required.

Here it is important to explain that the plugin reads the PostgreSQL connection settings for the current QGIS profile. These settings contain profile information about connection name, database name, hosting service, port, username and password. The plugin shows a list to users of all of the available stored connections similarly to how QGIS provides the same information from the "Data Source Manager" of PostgreSQL (Figure 4.9).



(a) QGIS list of profile's PostgreSQL connections from "Data Source Manager".



(b) "3DCityDB-Loader" list of profile's PostgreSQL connections from "Connection" group box.

Figure 4.9.: List of available PostgreSQL connections in "Data Source Manager" and "3DCityDB-Loader"

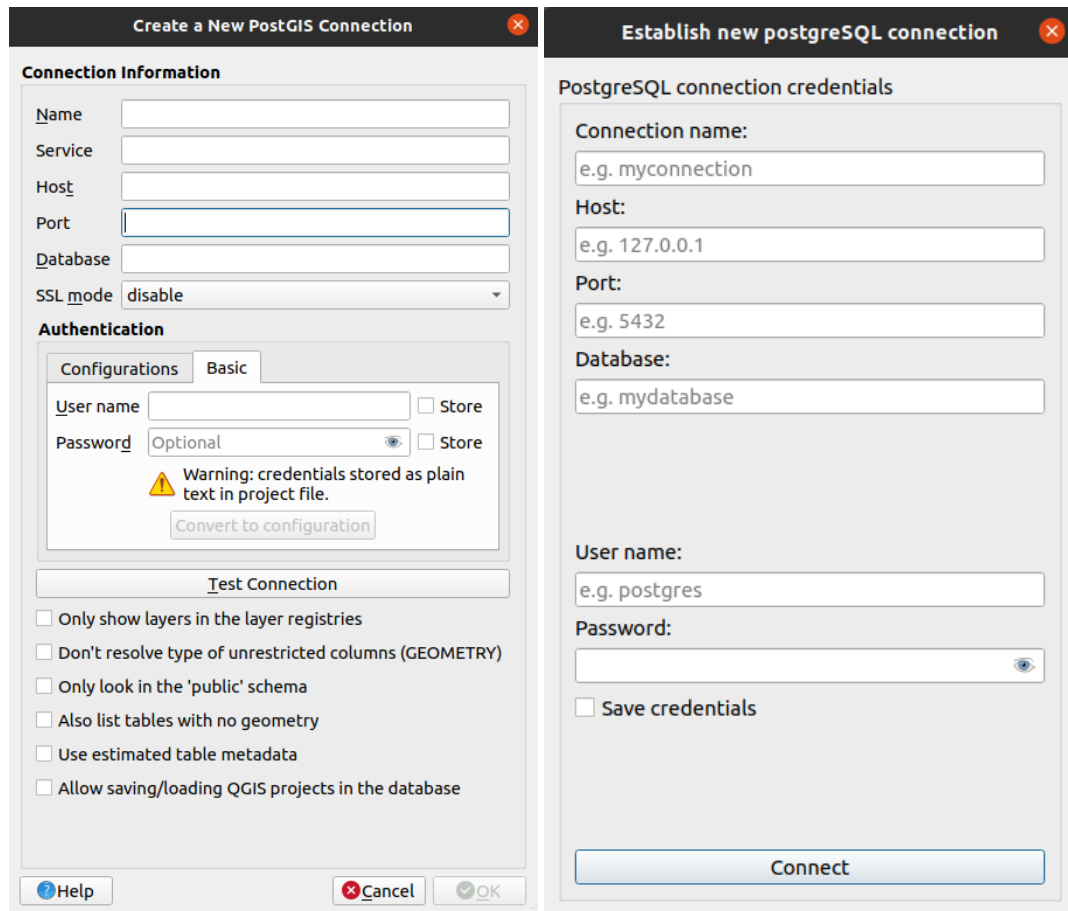
Thus, as a first step, users must choose a database and connect to it by:

- Choosing an existing connection.
- Creating a new connection.

A new connection could be created by two ways. One way, provided by QGIS built-in capabilities, is to create a new connection from the "Data Source Manager" of PostgreSQL (Figure 4.10a). On successful creation of the connection, the plugin appends it to the list of the available connections. The other way to do this is through the plugin itself. In the "Connection" group box, users can opt to create a new connection from the "New Connection" push button widget. This presents users with a new dialog (Figure 4.10b) where they can enter the

4. Plugin structure (server/client-side)

connection's credentials and elect to store their log-in information to the profile's settings for future use.



(a) QGIS new PostgreSQL connection dialog from "Data Source Manager". (b) "3DCityDB-Loader" new PostgreSQL connection dialog from "Connection" group box.

Figure 4.10.: New PostgreSQL connection dialogs in "Data Source Manager" and "3DCityDB-Loader"

The next step relates to the database in the homonym group box. Here, users can open the connection to the database. In the background there is a list of assertions that are being made to ensure the process continues as intended. These relate to:

- 1 Connection
- 2 Database instance
- 3 3D City Database installation
- 4 3D City Database version

The first check is to make sure that the connection with the database was established successfully. Another check here is that the database is indeed PostgreSQL. Next check is that

the database must be a 3D City Database (to have the correct structure). This check is implemented by calling a function that fetches the 3D City Database version. Its presence identifies the necessary installation. Lastly, the final check is based on the above-mentioned version and restricts the use of the plugin for databases of versions older than 4.x. This is an important restriction as the server-side installation is developed exclusively for the database structure of versions 4.x.

After passing all of the above checkpoints, users are being presented with the available "citydb" schemas in a combo box (list) widget (Figure 4.11). These schemas could be one or multiple and hold the primary data for use. Upon 3D City Database installation the first created schema uses the default name "citydb". However, there is the option to name the schema with a user-defined name, meaning that databases could exist without any "citydb" schema. In the context of this document and the plugin, a "citydb" schema is a primary data schema regardless of its real name.

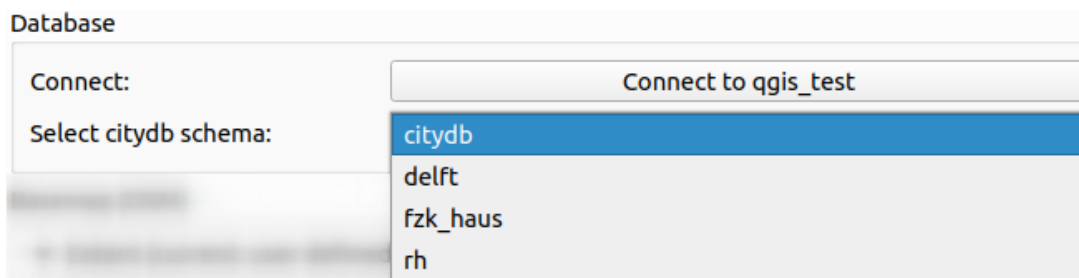


Figure 4.11.: "3DCityDB-Loader" available "citydb" schemas example.

A server-side function is used to determine and fetch all of the "citydb" schemas, however, only the selected schema is going to be put through another set of checkpoints.

- 1 Main installation
- 2 User installation
- 3 Layers
- 4 Geometries

The first check assesses that the "Main" server-side installation exists inside the database. The next check assesses that the "User" server-side installation exists inside the database. After that, a check determines that the database has generated layers (views) and/or that the extents of these layers are already computed. Lastly, as the geometries of the layers are stored as materialized views, a check determines if those have been already materialized or need to be refreshed.

If all of the checkpoints are passed successfully, users can move to the next tab ("Layers" tab). However, at the first ever use of the plugin, users need to setup the layers that are going to work on.

4. Plugin structure (server/client-side)

A base-map group box, referenced in figure 4.12, exists for two purposes. First, it shows the data schema's ("citydb") extents and presents them on an OSM base-map as a blue square. This helps users see the extents and the locations of where data exists in relation to the real world. The base-map is acquired from OSM and the coordinate reference system is set from the database. Additionally, it allows users to zoom-in and out using the scroll wheel of a mouse, and pan by pressing the scroll wheel and moving the mouse around. From this base-map users can set the extents of the database layers (views). There are three ways to do this.

- Calculate from Layer
- Map Canvas Extents
- Set to "citydb" schema

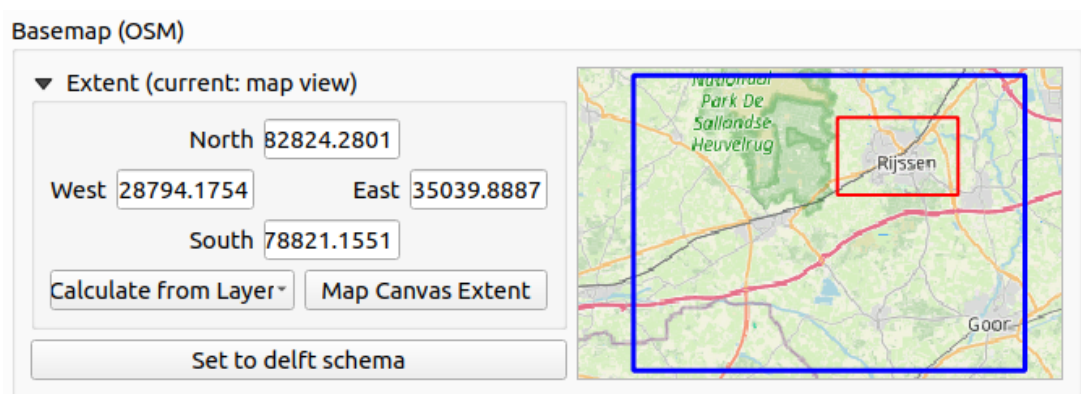


Figure 4.12.: "3DCityDB-Loader", "User Connection" OSM base-map example. (Blue="citydb" extents, Red=database layer extents (user-selected extents))

The first way allows users to select the extents from an already existing layer in the Map layer registry of the QGIS project. The second selects the extents of the current zoomed state of the map. Lastly, it allows to select the same extents as the extents of the schema itself. These user-defined layer extent selections are illustrated in the base-map using a red square.

After selecting the layers extents, users could elect to use the "Advanced options" (Figure 4.13). At version 0.4 of "3DCityDB-Loader", the only available advanced option is to use the server-side function to simplify the geometries. This is used to reduce some triangulation errors in 3D visualisation that are caused from duplicate or extremely close coordinates and really small polygons. Regardless of the advanced options, it is mandatory for users to generate the layers and after that to refresh them for the materialized view to be populated (Figure 4.14). Note that refreshing materialized views is relatively computationally expensive, thus, depending on the data size of the layers, this process may take some time. Before executing the operation, the plugin notifies the user about this behaviour. Lastly, users have the ability to delete all of the layers from their schema by clicking the "Drop layers for "citydb" schema" push button (Figure 4.14).

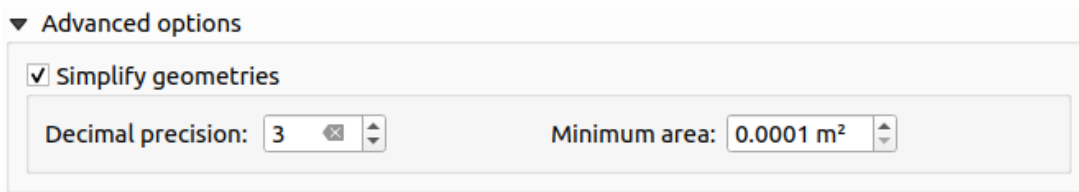


Figure 4.13.: "3DCityDB-Loader", Advanced options with their default values set. (Decimal precision=3, Minimum area=0.0001m²)

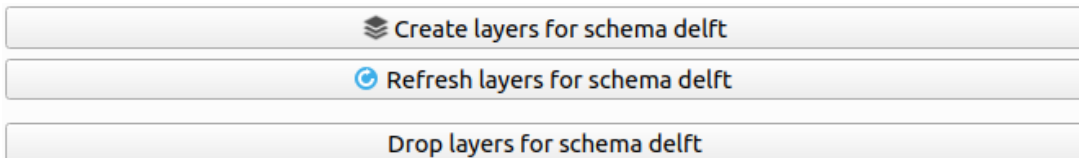


Figure 4.14.: "3DCityDB-Loader" Layer operations.

Similarly to the Administration tab (Section 4.3.1), in order to aid the user visually, all of the above steps are linked to the "Status Connection" group box at the bottom of the tab (Figure 4.15). In practice, here are illustrated the results from the various checkpoints.

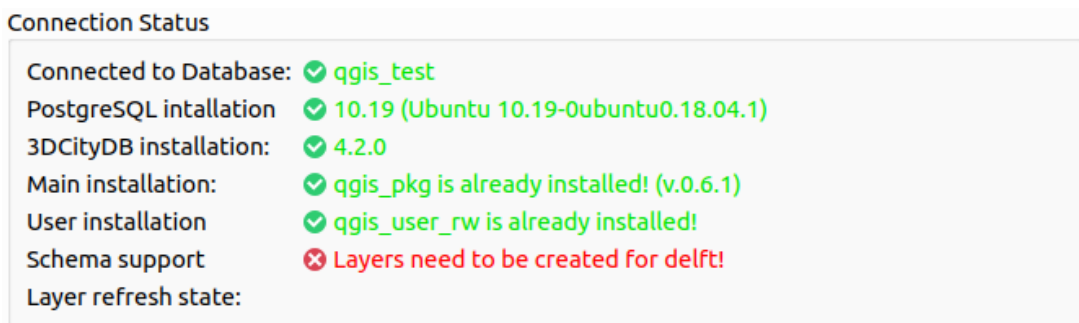


Figure 4.15.: "3DCityDB-Loader", "Connection status" report example. (Green=passed checkpoint, Red=failed checkpoint)

4. Plugin structure (server/client-side)

On successful checks the labels dynamically change their message to a short description.

- "Connected to database" label, shows the database name after a successful connection.
- "PostgreSQL installation" label, shows the versions of the PostgreSQL server after a successful connection.
- "3DCityDB installation" label, shows the version of the existing installation.
- "Main installation" label, shows the schema name and version of the server-side installation of the plugin.
- "User installation" label, shows the schema name of the connected user.
- "Schema support" label, shows that layers exists in the user schema.
- "Layer refresh state" label, shows the most recent date of layer refresh.

Similarly to successful checks, failed checks change their label message to different descriptions.

- "Connected to database" label, shows that the connection failed
- "PostgreSQL installation" label, shows that PostgreSQL couldn't be found.
- "3DCityDB installation" label, shows either that the version of the of existing installation is not supported or that the database is not 3DCityDB.
- "Main installation" label, shows that the main schema doesn't exist.
- "User installation" label, shows that the use schema doesn't exist.
- "Schema support" label, shows that layers don't exist in the user schema.
- "Layer refresh state" label, shows that layer need to be refreshed.

Note here that these messages are not strictly defined and can be changed in future development.

Lastly, similarly to the Administration tab (Section 4.3.1), users can elect to terminate the connection by clicking the "Close current connection" push button (Figure 4.16). This is important, as by just closing the dialog window, as explained in section 4.4, the connection remains open and the plugin state freezes.

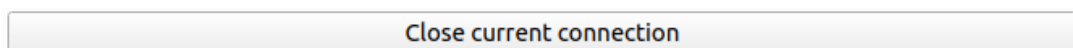


Figure 4.16.: "3DCityDB-Loader" push button widget used to close the connection to the database.

4.4.2. “Layers” tab

When all of the labels in “Connection Status” group box are green, this indicates that the “User Connection” tab is correctly setup and users can move to the “Layers” tab (Figure 4.17).

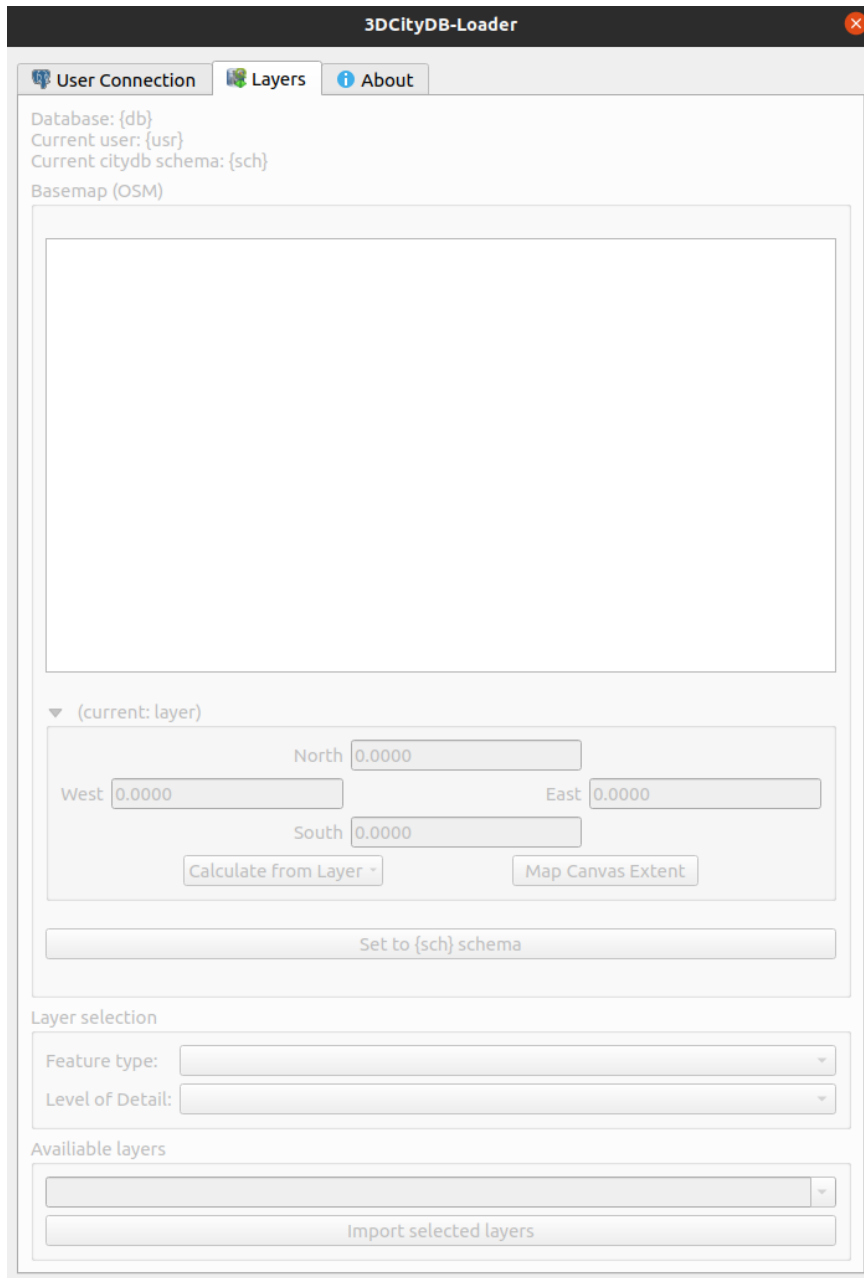


Figure 4.17.: “3DCityDB-Loader” “Layers” tab in its initial state.

4. Plugin structure (server/client-side)

In short, this tab relates to selecting and filtering the layers that users want to import. The filters are based on extents, CityGML feature types (known as modules) and geometry LOD.

Firstly, at the top of the tab, there are label widgets displaying the current database, user and data schema ("citydb") names (Figure 4.18). This exists to have users constantly informed about their working environment without having to change tabs.

```
Database: plugin_dev
Current user: postgres
Current citydb schema: citydb_dutch
```

Figure 4.18.: "3DCityDB-Loader" workspace label widgets example in "Layers" tab.

Next, follows a base-map widget similar to the one in "User Connection" tab (Figure 4.19). However, here, the user-defined extents (now in green) relate to the QGIS layer extents and not the database layer extents (in red). In more detail, the schema's extents are illustrated again as a blue square. The database layers' extents are illustrated as a red square. And lastly, the QGIS layer extents are illustrated as a green square. Similar to the other base-map, the QGIS layer extents can be set using the extents of another (existing layer in the QGIS map registry), using the window extents of the widget and using the same extents of the database layers (red square).

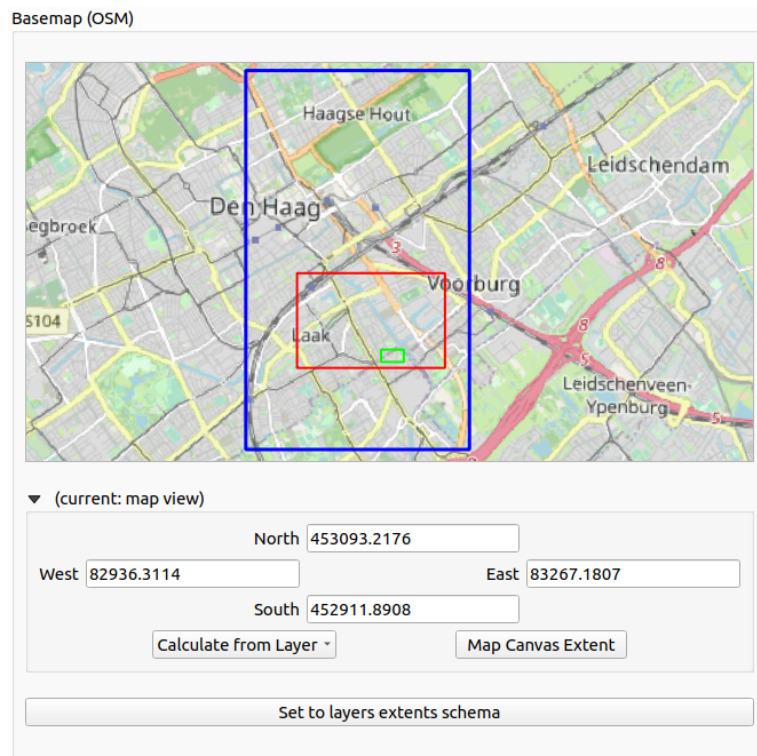


Figure 4.19.: "3DCityDB-Loader" OSM base-map example in "Layers" tab. (Blue="citydb" extents, Red=database layer extents Green=QGIS layer extents (user-defined extents))

After that, users are required to select from a particular feature type and specific LOD (Figure 4.20). Upon selection of the QGIS layer extents, the plugin dynamically checks and shows only those layers for which there is at least one record. Layers that don't have any data within the selected extents are hidden. Similarly, LOD options are also dynamic displaying only options of the selected feature's type data that exist in the extents.

(a) "3DCityDB-Loader" example of available Feature type selection in user-defined extents (3 out of 10).

(b) "3DCityDB-Loader" example of available LOD selection in user-defined extents for the selected Feature type (2 out of 5).

Figure 4.20.: "3DCityDB-Loader" example of feature selection parameters available in user-defined extents.

Lastly, at the bottom of the tab are the results of the above filters and the function to import the user's selection (Figure 4.21). Based on the user's parameters, a final list shows one or more layers that are ready to be imported into the QGIS project. Moreover, the layers are represented with the layer (view) name accompanied with the number of features. The list is also a checkable combo box ("QgsCheckableComboBox"), meaning that users can choose multiple layers to be imported simultaneously.

Figure 4.21.: "3DCityDB-Loader" example of multiple available layers to import.

Finally, upon selection of at least one available layer, the "Import selected layers" push button becomes enabled. The push button is linked to two different functions. The first one is responsible to generate QGIS layers from the PostgreSQL view and the second to structure the

4. Plugin structure (server/client-side)

QGIS project to accommodate the new layers accordingly. More about the second event are discussed in section 4.5.

It is also important to note that the filters are activated upon any new user-defined extents selection. A spatial intersection is queried for all generated layers, which results to the number of available features found inside the extents. This number determines the availability of the layers. In that way, it provides a dynamic approach that gives users insight about the size of the data that are going to be imported. Moreover, acknowledging the fact that a large data-set could cause memory issues within QGIS, the plugin notifies the user of large imports before proceeding the operation (Figure 4.22). This approach could potentially prevent QGIS project crashes but it doesn't restrict the user.

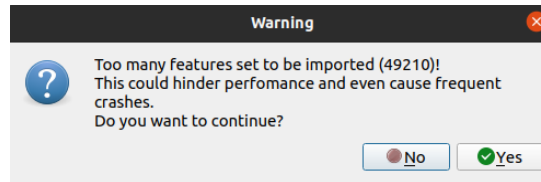


Figure 4.22.: "3DCityDB-Loader" example of warning message about a large amount of features.

4.4.3. "About" tab

The "About" tab is the last tab of the user dialog and it contains the plugin's metadata (Figure 4.23). These are presented as informative text and users don't need to do anything particular in this tab.

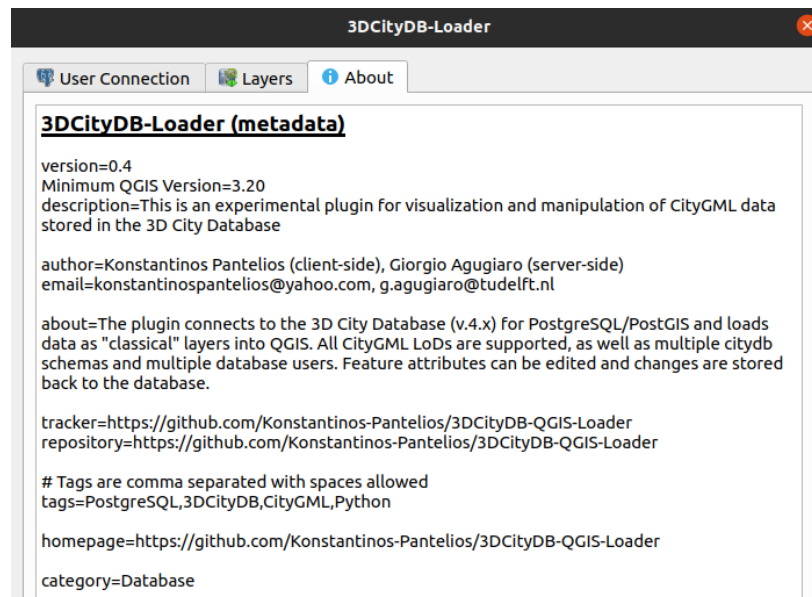


Figure 4.23.: "3DCityDB-Loader" "About" tab.

4.5. QGIS project structure

Upon every import of new layers, the plugin restructures the QGIS project in way that organizes them for easier handling. This is targeted towards a better user experience as the extensive amount of different layers, attributes, representations or even schemas that come with CityGML and the 3DCityDB encoding could be daunting to use effectively.

4.5.1. Layers

The first QGIS structuring relates to the views that are imported as layers.

Data source

To begin with, the plugin uses QGIS's vector layers class (QgsVectorLayer) to create layers from the views inside the database. This is done by providing the data source name which in this case is "postgres", the layer name and a URI to the constructor. The URI mainly consists of connection and data source information.

Connection parameters include:

- Database name
- Host
- Port
- User
- Password

Data source information include:

- Schema
- Table
- Geometry column
- SQL query
- Primary key column

In the case of the plugin's implementation, a table is a view which is used to generate the QGIS layer. The SQL query is used to spatially filter the view to receive only data that intersect the user-defined extents.

4. Plugin structure (server/client-side)

Layer naming syntax

Regarding the name of the layer, it inherits the name of its source view. This is helpful for users as by design of the server-side installation, the name is formatted in a way to hint the data contents. The convention is that a view name starts with the "citydb" schema that relates to, followed by the feature name or alias, next the LOD and lastly but conditionally the geometry type if it is relevant (Figure 4.24). The above are separated by underscores but note that multi-worded arguments are also separated by underscores.

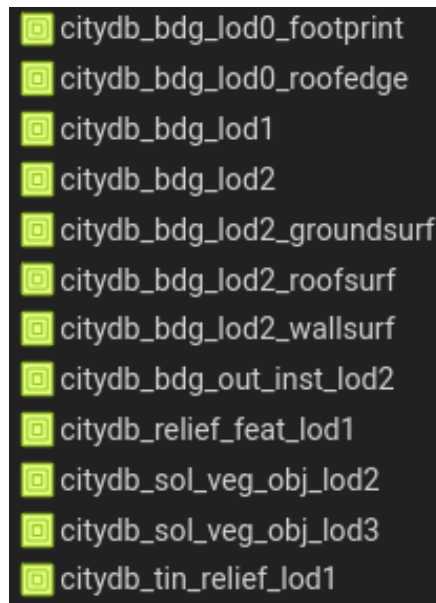


Figure 4.24.: View name examples in a database (Figure from pgAdmin web view).

Layer properties

In QGIS, layers are instantiated with default properties regarding their styling/symbology, variables, attribute form, actions and other. That said, QGIS allows for custom properties to be saved and even exported as styles into XML-based QML files, SLD files or UI stored inside the database. This plugin utilizes the QML approach with files being prepared for every layer and stored into the plugin's installation directory. The QML files are structured to stylize the symbology (only color) and the attribute form based on the proposed format. Finally, these files are mapped by name to their corresponding views and are loaded into the layers soon after their creation in the QGIS project.

Layer symbology

Regarding symbology, a color schema is proposed in order to map specific colors to specific features. Each feature is created with its own color. The first reason for this decision was that the plugin, at the current version 0.4 does not support CityGML "Appearances", thus custom

colors are not transferable. The second reason is that in new layer instances random colors are assigned regardless the layer name or data source.

Layer attribute form design

QML files store also information about the structure of the attribute form (Figure 4.25). Comprehensive attribute forms are especially important for data entry operations. The default attribute form is restructured to not only accommodate better specific field types, but also to organize the fields according to relevance. The general format followed by all layers is that CityGML "CityObject" information are located at the top, segmented by tabs into more specific groups. In more detail, the "Main Info" tab contains feature identification data (Figure 4.25a). The "Database Info" tab includes data related to database operations (Figure 4.25b). The "Relation to surface" tab includes the relations to surfaces data (Figure 4.25b). Lastly, "Generic Attribute" tab, as the name suggests, includes a nested table of all available generic attributes from the "genericAttribute" class (Figure 4.25d). Below, there are tabs for the class, function and usage of the feature, however, depending on the feature, these may be absent (Figure 4.25e). Last at the bottom, a group box is used to store all feature specific attributes (Figure 4.25f). The QML files are also equipped with specific aliases that replace the field name to facilitate comprehension in the form.

4. Plugin structure (server/client-side)

Database ID	81
GML ID	GMLID_BUI130363_1235_6047
GML codespace	NULL
Name	Boatshouse KIT/KHH-1
Name codespace	NULL
Description	Simple Boathouse on the lake side

(a) Attribute form - "Main Info" tab example.

Creation date	17-03-2022 08:05:30
Termination date	NULL
Last modification	17-03-2022 08:05:30
Updating person	postgres
Reason for update	NULL
Lineage	NULL

(b) Attribute form - "Database Info" tab example.

Relative to terrain	(no selection)
Relative to water	(City)Object entirely above water surface

(c) Attribute form - "Relation to surface" tab example.

id	133018
parent_genattrib_id	NULL
root_genattrib_id	133018
attrname	lod2_volume
datatype	6
strval	NULL
intval	NULL

(d) Attribute form - "Generic Attributes" tab example.

Class	Unknown
Class codespace	NULL

(e) Attribute form - class/function/usage/ attributes example.

Year of construction	NULL	Year of demolition	NULL
Storeys above ground	NULL	Storeys below ground	NULL
Height	NULL	UoM	NULL
Storey height above ground	NULL	UoM	NULL
Storey height below ground	NULL	UoM	NULL
Roof type	NULL	Codespace	NULL

(f) Attribute form - feature-specific attributes example.

Figure 4.25.: Attribute form - "Building" attributes example.

Layer attribute form constraints

Some CityGML attributes are accompanied with a field in regards to their unit of measure (e.g. height, meters). However, it is noted that as the attribute's value can differ from entry to entry, the unit of measure remains the same. In QGIS, it is necessary that these two field are linked. In more detail, as an example, the unit of measure of height can be undefined only if the height value itself is undefined. Likewise, the height value can be updated only if it is accompanied by a corresponding unit of measure value (Listing 4.1). This and other restrictions can be set by utilizing custom expressions that act as constraints (Figure 4.26).

Figure 4.26.: Example of an attempt to pass wrongful values. A value of -6 is inappropriate for the "stores above ground" field. This caused the constraint (described in the red square) to take effect by disabling the "OK" button. The user is now forced to resolve the issue or cancel the attempt.

```
1 ("measured_height" IS NOT NULL AND "measured_height_unit" IS NOT NULL)
2 OR
3 ("measured_height" IS NULL AND "measured_height_unit" IS NULL)
```

Listing 4.1: Example custom expression for "measured_height" and "measured_height_unit" fields.

Similar constraints are also for fields like the storeys above or below ground where the value can be either undefined or non negative (Listing 4.2, Figure 4.26).

```
1 ("storeys_above_ground" IS NULL) OR
2 ("storeys_above_ground" >= 0)
```

Listing 4.2: Example custom expression of "storeys_above_ground".

4. Plugin structure (server/client-side)

These constraints are saved and are loaded automatically from the [QML](#) files that accompany the layers.

Auxiliary layers

Lastly, upon every import of a layer, layers for the CityGML "genericAttribute", enumerations and codelists are also created and added to the project. However, these layers are added once, thus, the plugin checks if the layers have already been added to the project. Although, these layers are different from the feature views as they are geometry-less tables, the [QGIS](#) layers are created in the same way as described before (as vector layers without geometries).

4.5.2. Relations

The "genericAttribute", enumerations and codelists are tables that are linked to the feature's attributes, thus, depending on the table structure, different widget types are used to represent them in the attribute form. These can be linked using special kind of [QGIS](#) relation objects.

Regarding the "genericAttribute" table, [3DCityDB](#) uses a table with a "many-to-one" [QGIS](#) relation between multiple generic attributes per feature. Consequently, a similar relationship is created in [QGIS](#) with the "QgsRelation" class. The referencing layer is the "genericAttribute" table accompanied with its "cityobject.id" field as the key. The referenced layer is set to the feature layer with the "id" field as the key. Finally, the relationship type can be described as composition. This relation object is then assigned to a "Relation editor" widget type with a many-to-one cardinality. In practice, the attribute form holds the generic attributes as a nested table (Figure 4.25d).

Concerning the enumerations, these are tables of standard and normative CityGML values for feature attributes. The relationship here is one-to-one, meaning that it could be represented by the widget type of "Value Relation". The key column is set to the field "value" and the value columns is set to the field "description". Moreover, as all enumerations are stored under one table in the database, the filter expression could be user to get the appropriate values (e.g. name = 'RelativeToWaterType' for relative.to.water field).

Setting up relations was also tried for codelists but with no implementation. This is due to the fact that they do not have standard hard-coded values (like enumerations) and can vary in different databases. That being said, a default table (SIG-3D) is still added into the [QGIS](#) project and users have the ability to set up the connection themselves from the attribute form properties.

Finally, it is important to mention that the above relations are constructed on the fly with every feature layer creation. This is different than the rest of the attribute form which is loaded from prepared [QML](#) files. The relations links use layer instances identifications instead of layer name, meaning that they cannot be saved into [QML](#) as in a new import (of the same feature layer) a new instance is being created with a different unique identifier. So, the relations are being created and loaded immediately but after loading the [QML](#) file.

4.5.3. Table of Contents

A QGIS project stores layer object into its layer tree which consists of the TOC. The table of contents is used to categorize layers in nodes (groups) by relevance using comprehensive names (Figure 4.27). By default, however, new layers are assigned to the tree's root which is not helpful. The plugin, for every layer creation, creates or modifies the TOC according to a specific structure. The first level of the tree holds nodes that are named after the database of the layer's data source. The database nodes are hierarchically at the top, followed by the "citydb" schema that the layer is linked to in combination with the current user (joined by "@" symbol). This level contains, additional nodes named after the Feature type (CityGML module), a node holding the "genericAttribute" table and a node for look-up tables (enumerations and codelists). The Feature type nodes, contain additional nodes based on the layer's LOD which they hold finally the imported layer.

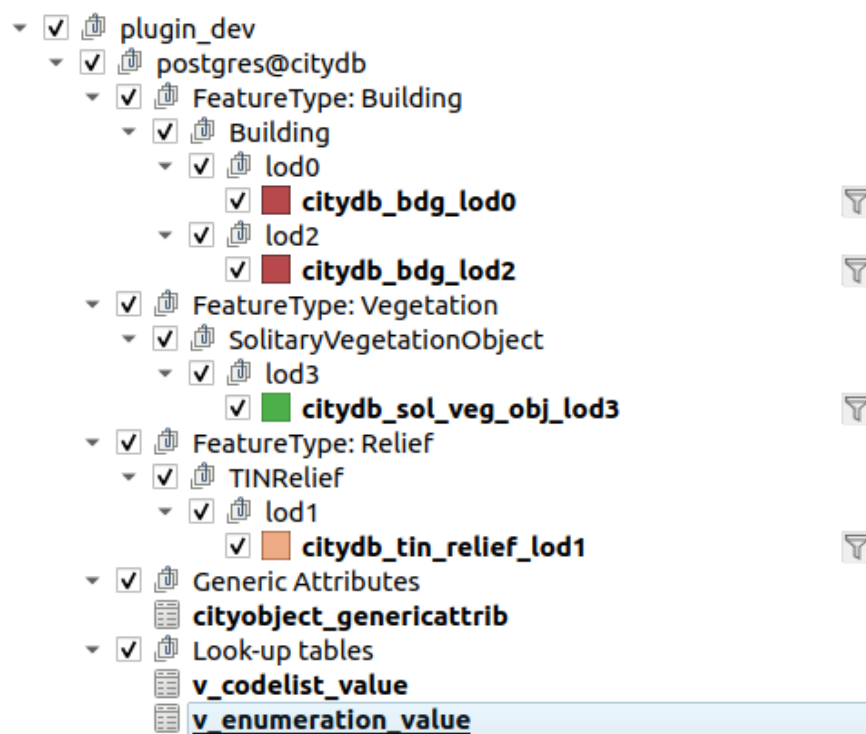


Figure 4.27.: Structured ToC example.

To keep the TOC consistent, the nodes are being alphabetically ordered every time that it is built or modified. The ordering is executed as a recursive function visiting not only the groups, but the layers as well. However, noting the fact that QGIS draws the layers in the canvas based on the TOC's layer order, an exception for "Relief" is made. Although, in 3D view the "Relief" can be illustrated at the same time with other Feature types, in 2D view it overlaps them. Consequently, the "Relief", is always considered as the last Feature Type regardless of alphabetical order.

4. Plugin structure (server/client-side)

4.6. Software development

As already mentioned in section 3.4.1, the plugin was mainly developed using the “Python” programming language utilizing the QGIS’s Python API. Additionally, the QGIS plugin “Plugin Builder 3” was used to generate a blank plugin template. This template is used as the base upon which “3DCityDB-Loader” is built. Moreover, the template contains the plugin’s main class which is used by the QGIS to recognize and execute the plugin. It also contains metadata information about authors, categories, version key words and other. Lastly, it provides some scripts for building and deployment, but those were not used in the development phase of the plugin.

4.6.1. Object-oriented programming

The programming style of the plugin follows OOP [Wegner, 1990] concepts derived from the native Python API of QGIS and extended to fit special cases. These special cases relate to specific needs of the plugin that the built-in classes cannot handle on their own.

Built-in QGIS classes are stored in specific libraries with modules segmented by relevance (Table 4.1). These classes are used extensively in the development of the plugin and can relate to all elements of a QGIS project (e.g. settings, map, widgets, relations, layers and more). Moreover the plugin uses the “Psycopg 2” Python library which handles the PostgreSQL database communication [Di Gregorio and Varrazzo, 2021].

Library	Description
core	The CORE library contains all basic GIS functionality.
gui	The GUI library is build on top of the CORE library and adds reusable GUI widgets.
analysis	The ANALYSIS library is built on top of CORE library and provides high level tools for carrying out spatial analysis on vector and raster data.
server	The SERVER library is built on top of the CORE library and adds map server components to QGIS.
3d	The 3D library is build on top of the CORE library and Qt 3D framework and adds support for display of GIS data in 3D scenes
processing	The PROCESSING library is build on top of the CORE library and adds support for processing algorithms.

Table 4.1.: QGIS libraries [QGIS-Python-API, 2022]

In more detail, other than the main plugin classes and the native QGIS ones, the code makes use of custom classes relating to the database views, CityGML feature types, database connection and multi-threading workers.

The “view” class is used to convert metadata about the database views from table form to objects with attributes for easier access and handling in the code. Additionally, these are complemented by the “feature type” class which mainly acts as a container grouping the view objects by type. The “connection” class is mainly used to store user credentials that are used to execute transactions with the database. Lastly, “multi-threading workers” are

classes that are used to execute processes in a separate thread. These classes are reserved for operations that are usually resource-consuming [Malakhov, 2016].

4.6.2. Working directory

As mentioned in the introductory paragraph of this section, the software is built on a specific template. This template was further modified to account for the plugin's unique requirements. That said, some of the initial files are kept in the plugin's main directory for future use. Consequently, these inactive elements that were not used, are not going to be included in this overview. The most important elements of the directory are presented in figure 4.28.

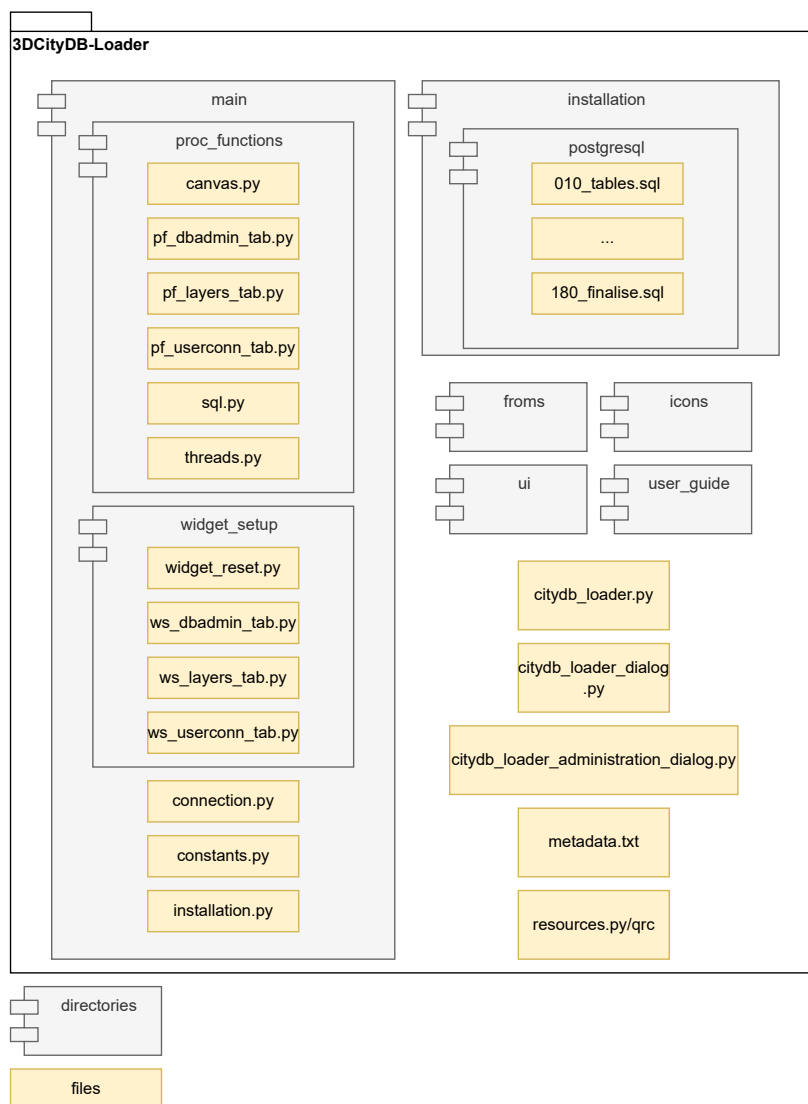


Figure 4.28.: "3DCityDB-Loader" Working directory diagram.

4. Plugin structure (server/client-side)

To begin with, at the top level of the working directory there exist the most generalized Python files. These relate to the plugin's class, the dialogs and the resources. It also contains a metadata text file (Listing 4.3) which is used by the QGIS "Plugin dialog" to display relevant information (Figure 4.29).

```
1 [general]
2 name=3DCityDB-Loader
3 qgisMinimumVersion=3.20
4 description=This is an experimental plugin for visualization and manipulation of
5   CityGML data stored in the 3D City Database
6 version=0.4
7 author=Konstantinos Pantelios (client-side), Giorgio Agugiaro (server-side)
8 email=konstantinospantelios@yahoo.com, g.agugiaro@tudelft.nl
9
10 about=The plugin connects to the 3D City Database (v.4.x) and loads data as "
11   classical" layers into QGIS. All CityGML LoDs are supported, as well as
12   multiple citydb schemas and multiple database users. Attributes can be
13   edited and changes are stored back to the database.
14 tracker=https://github.com/Konstantinos-Pantelios/3DCityDB-QGIS-Loader
15 repository=https://github.com/Konstantinos-Pantelios/3DCityDB-QGIS-Loader
```

Listing 4.3: Part of the Metadata.txt file.

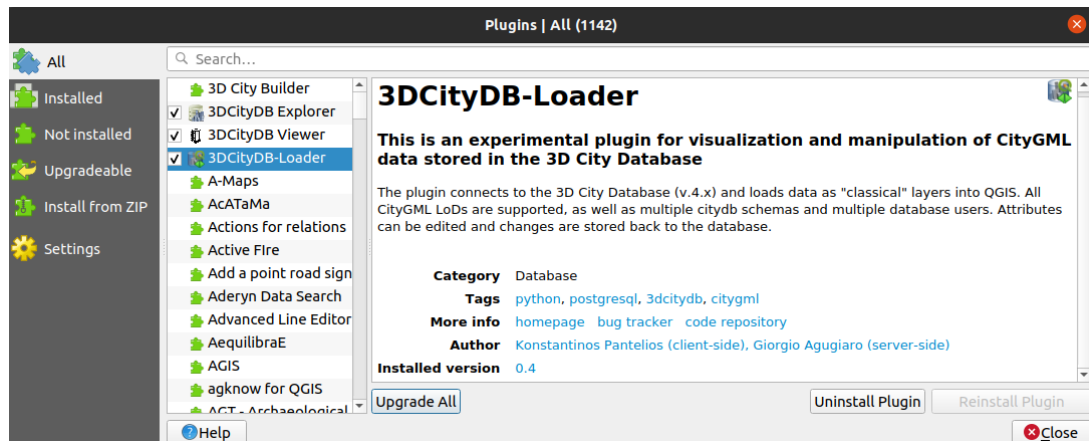


Figure 4.29.: Metadata shown in "Plugin dialog".

The origin file of the plugin is the *"citydb_loader.py"* which contains its main class. The class loads the dialogs for the different actions (Sections 4.4 and 4.3) and executes the main event loop. It also contains, as methods, the custom slots which are connected to predetermined Qt signals. Moreover, these slots call functions stored in the *"widget_setup"* directory. Lastly, the class in this file is also responsible for unloading the plugin from the QGIS project when requested.

The file *"citydb_loader_dialog.py"* holds the class that builds the GUI for the user dialog (Section 4.4). Similarly, the file *"citydb_loader_administration_dialog.py"* holds the class that builds the GUI for the administration dialog (Section 4.3). In both files, the GUI is loaded from a pre-build UI file that was designed in "Qt 5 Designer" (Figure 4.30)

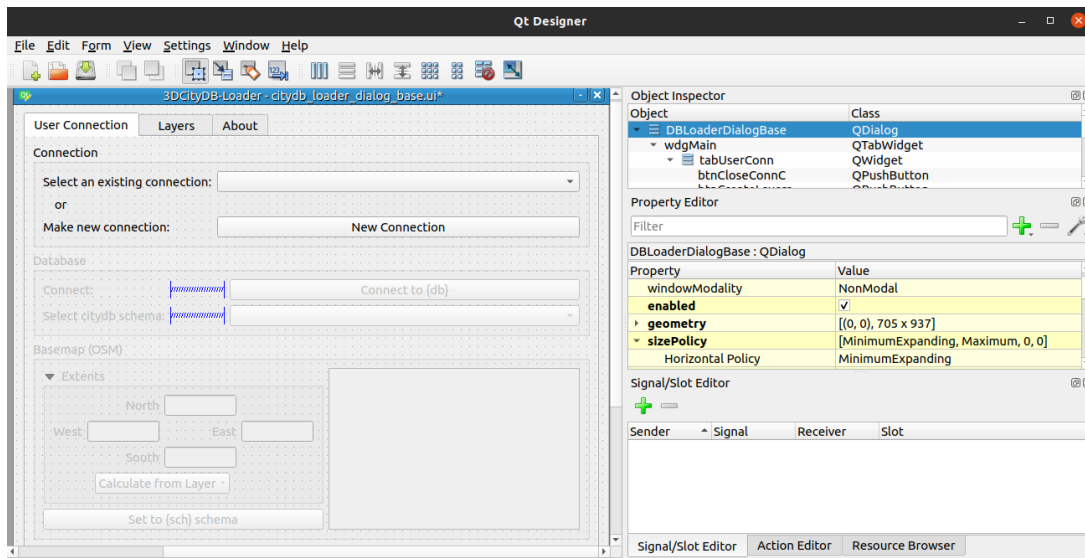


Figure 4.30.: "Qt Designer" GUI designing example.

The *"resources.qrc"* file contains the relative paths of resource objects. In this case these objects are image icons that are used in the plugin. These resources are then compiled into the *"resources.py"* file using the PyQt5 resource compiler (Listing 4.4).

```
1 pyrcc5 -o resources.py resources.qrc
```

Listing 4.4: Compiling resources from terminal.

Regarding the sub-directories:

- The *"forms"* directory is a container storing QML files that are attached to imported layers.
- The *"icons"* directory is a container storing icons used in the plugin's GUI.
- The *"ui"* directory is a container storing the .ui GUI files built from "Qt 5 Designer".
- The *"user_guide"* is a container storing a file used to describe and document the project.
- The *"installation"* directory contains the installation scripts for the server-side QGIS package.
- The *"main"* directory contains the building code blocks of the plugin.
- The *"widget_setup"* sub-directory contains code that is responsible to handle changes in the plugin's state.
- The *"proc_functions"* sub-directory contains code responsible for low-level specialized processing operations.

5. Test case implementation

In this chapter, a possible scenario is going to be simulated to show the plugin's use. This can be considered as a guideline to understand the process and order of operations. Additionally, although the scenario is fictional, it is complete, exploring most available features of the plugin. Lastly, the approach that is used here exhibits the intended use of the plugin.

5.1. Scenario

An employee of an organization is tasked with the mission to give access to information to a client about the buildings of the village of Dijkerhoek in Rijssen-Holten, The Netherlands, which was recently redeveloped. The redevelopment was an increase to the maximum plot ratio in the area, which resulted to the construction of additional floors on the existing buildings. The employee has intermediate skill in QGIS, but has inadequate knowledge on programming and database operations. The organization has a PostgreSQL, 3DCityDB enabled database containing CityGML buildings and other features for the entire country (The Netherlands), however, it is not yet updated to the new state. Moreover, the same database stores data for other applications in different schemas. The database is managed by an administrator. Lastly, the organization operates using QGIS.

From the situation described above, it is clear that there are some issues that need to be addressed. The first issue is that the client is only interested in the specific small area while the organization's database has vast amount of data for the entire country. The second issue is that the database is not up-to-date, so the data need to be updated before passing them to the client. The third issue is that the client must access the data but without being able to make any changes as external actor. The fourth issue, is that the employee doesn't have the appropriate skills to access, update and prepare the necessary data programmatically, as he/she doesn't have any knowledge on SQL, PostgreSQL or 3DCityDB.

5. Test case implementation

5.2. Pipeline

The above problem could be easily solved by the employee following the process and logic that is described below (Figure 5.1).

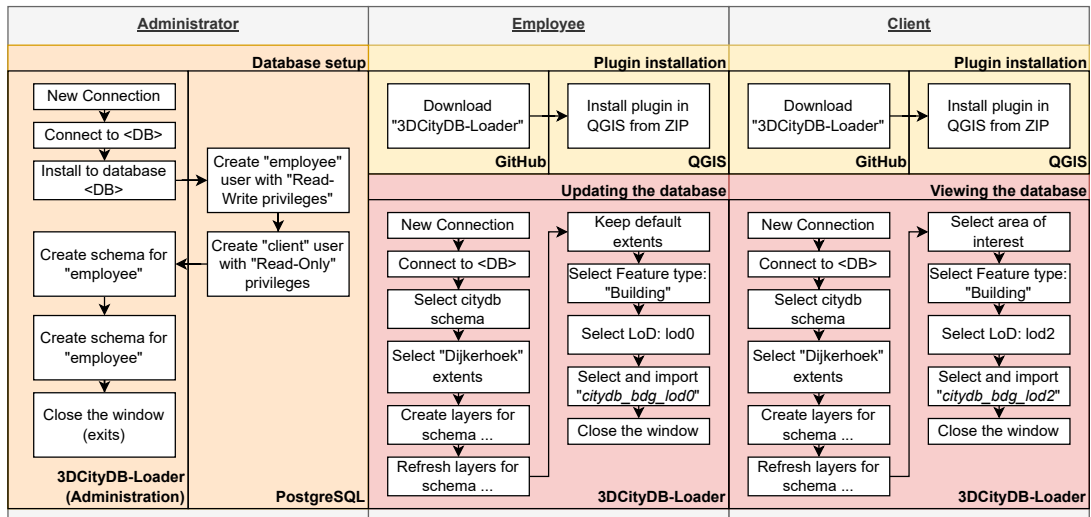


Figure 5.1.: Plugin workflow of the scenario example

5.2.1. Plugin installation

First, suggested by the database administrator, the employee can find the [QGIS](#) plugin "3DCityDB-Loader" from Github and download it. Then the plugin could be installed using [QGIS](#)'s "Install from ZIP" option (Figure 5.2).

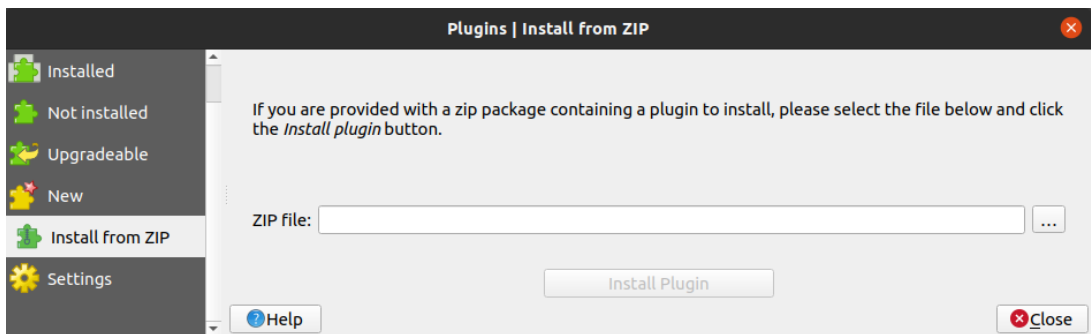


Figure 5.2.: QGIS plugin installation from ZIP

After a successful installation, the plugin's actions are available for use in the Database menu and toolbar.

5.2.2. Database setup

As the database need to be configured, the only eligible person to do this is the administrator. To begin with, the administrator needs to use the "3DCityDB-Loader (Administration)" action from the Database menu. Here, in order to connect to the organization's database, the connection parameters need to be inserted. These parameters could already exist but in this scenario, its the first time that the administrator uses QGIS for database related operations. Thus, by clicking "New Connection", it is possible to add the necessary credentials to establish the connection. For security reasons, the administrator can opt to not save the user name and password.

Next step for the administrator is to install the "Main" schema ("qgis_pkg") by clicking the named button (Figure 4.7b). Installing the "Main" schema, also creates in the database a new user group for users that are going to work with the plugin. Moreover, it creates two new default users with specific privileges. The administrator can opt to create new custom named users for the employee and the client. The privileges can be assigned by directly executing the function "grant_qgis_usr_privileges" (outside of the plugin) (Listing 5.1). A user named "employee" is created with "Read-Write" privileges as he/she needs to commit changes into the database. A user named "client" is created with "Read-Only" privileges as the organization shouldn't allow people outside its workforce to make modifications in its database. Additionally, using the same function, the administrator can limit the access to both employee and client only to the specific schema that holds the required data.

```

1 -- Setting-up employee user
2 CREATE USER employee WITH PASSWORD 'wg76sdft5';
3 SELECT qgis_pkg.grant_qgis_usr_privileges('employee','rw','citydb');
4
5 -- Setting-up client user
6 CREATE USER client WITH PASSWORD 'kj23jimh4';
7 SELECT qgis_pkg.grant_qgis_usr_privileges('client','ro','citydb');

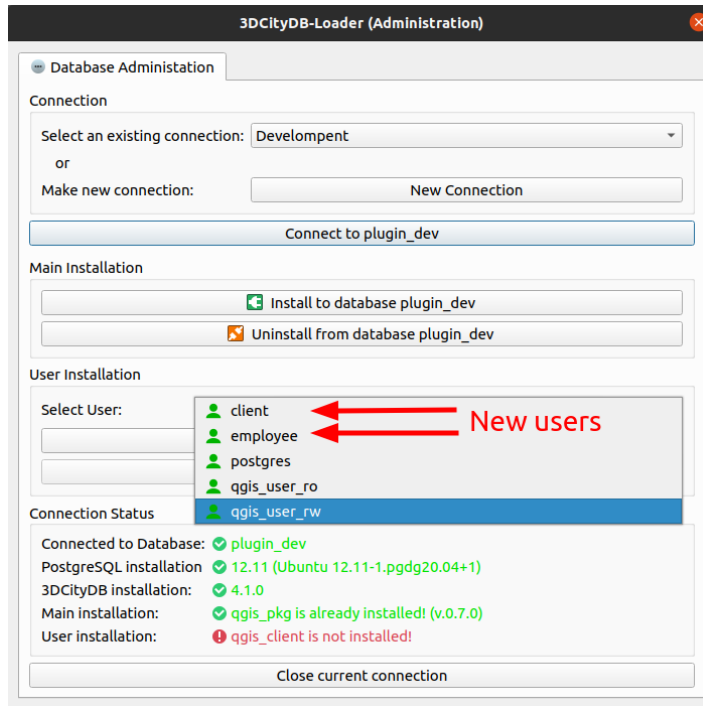
```

Listing 5.1: SQL queries to set-up new users

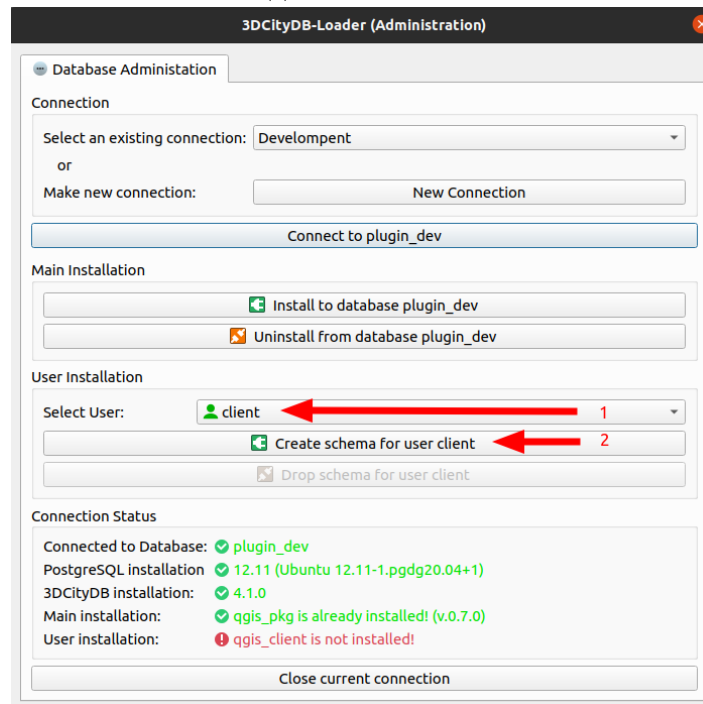
Now, that the relevant users are assigned with their respective privileges, the administrator can continue in the plugin by selecting and creating a schema for each user. This is done by simply selecting the user (to create the schema for) from the combo box widget and clicking the button to do so (Figure 5.3).

At last, if the setup was successful the Connection Status report should show that the installation exists in green text. The administrator can now close the dialog which by design also closes the connection to the database.

5. Test case implementation



(a) Available users



(b) Create schema option for selected user

Figure 5.3.: Creating schemas for users from "3DCityDB-Loader (Administration)"

5.2.3. Updating the database

Now the database is ready to be used with the plugin. The employee can either open the plugin from the toolbar or the Database menu. Either way, the initial state is in the "User Connection" tab where the following steps need to take place.

Regarding the connection, the administrator gives the employee's user credentials along with the name of the schema that he has access to. Using these credentials the employee can connect to the database using the same process as the administrator above (clicking "New Connection" and filling the credentials). As the employee may spend multiple days updating the database, it is helpful to store the credential for easier access.

Next, by selecting the proper "citydb" schema storing the required data, the employee can see its extents in the base-map. However, generating layers for the entire country can cost a lot of time. As the area of interest lies only at the extents of Dijkerhoek, the employee can zoom-in and set the extents only for there (Figure 5.4).

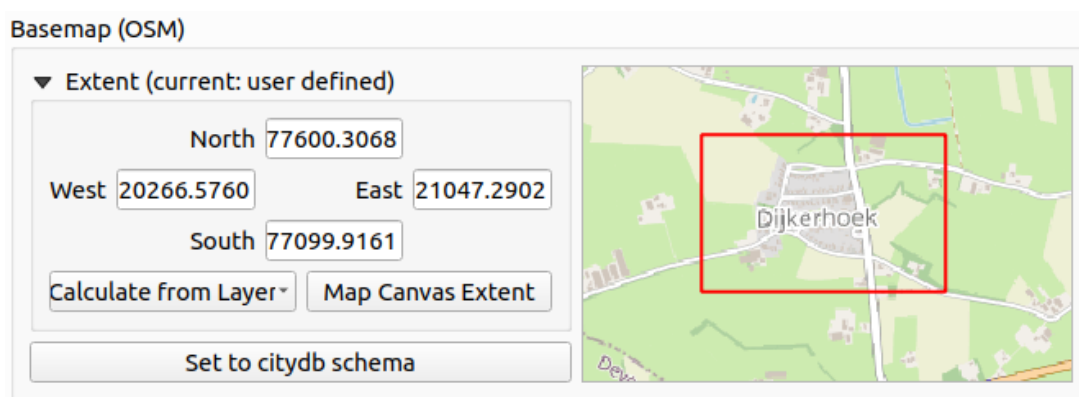
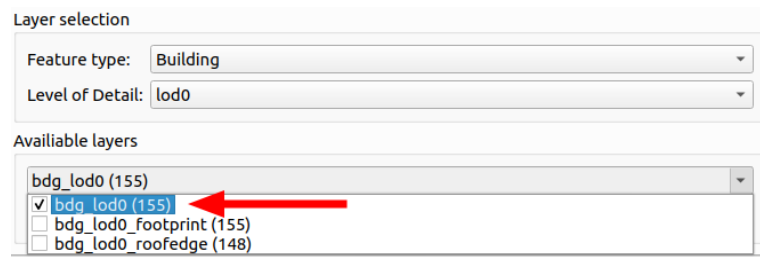


Figure 5.4.: Selected Dijkerhoek extents (in red square).

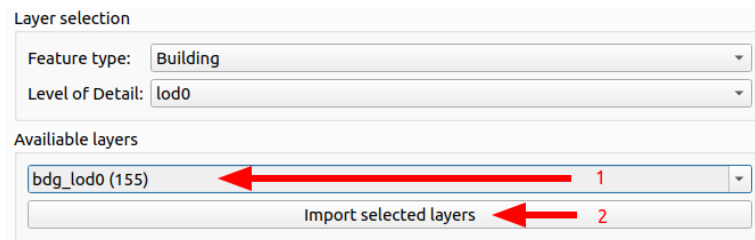
In the next step, regarding the advanced options the employee's job is unrelated to the available function, so this step is skipped. To move further, consulting the "Connection Status" report and judging from the available and unavailable widget, the employee presses the "Create Layers" button. This generates the layers into the specified extents, but the geometries in the database still need to be populated. This step may not be well understood by a beginner user like the employee of this scenario. Thus, again, the "Connection Status" reports hints the employee to click the "Refresh Layer" button. Finally, when all steps are completed successfully, the labels in the report should be in a success state hinting to move to the next tab.

Moving to the "Layers" tab, the employee, is presented with the extents of the created layers, the extents of the entire schema and extents that can be used to spatial filter the layers to be imported. In this case, the extents of the user are the same as the extents of the layers in order to import every object that was generated. As the employee only cares about building attributes, he/she chooses from the "Feature type" combo box the "Building" and from "Level of Detail", LOD0. Choosing the above parameters, result to the three layers being available to import: `bdg_lod0`, `bdg_lod0.footprint` and `bdg_lod0.roofedge`. From the three, the employee chooses to import only the first one by selecting it and clicking the "Import" button.

5. Test case implementation



(a) Available layers



(b) Import selected layers

Figure 5.5.: Importing layers in QGIS from "3DCityDB-Loader"

Now, the layer has been imported in QGIS along with the "genericAttribute", codelist and enumeration tables. The employee can update the attributes by moving from feature to feature and changing the necessary fields in the attribute form. Saving the edits, updates the database with the new values in the "citydb" schema and table that the layers is originated from. Finally, having updated the database, the client can now access the data.

5.2.4. Viewing the database

This process is similar to the process above with the significant difference that the client doesn't have the privilege to update the database. The database administrator makes the user credential known to the client. The client downloads and install the plugin in the same way as the employee. Using the credential, the client can connect to the database and choose the schema directed by the administrator. Next, determining the area of interest the client can select the entirety of Dijkershoek, or in this case, a bigger buffering area to get additional data that could be otherwise useful. After that, hinted by the "Connection Status" report, the client can generate and refresh the layers. In the next step, the client can choose to load data from the extents concerning only the area of interest. For these extents, the client can further filter the data by Feature types and LOD. As one of the needs is to generate a 3D model for urban plans, the client first chooses and imports buildings of LOD2. However, wanting to have a more comprehensive 3D city model of the neighborhood, the client can re-open the plugin to choose and import vegetation object of LOD3 and tin relief of LOD1. Finally, the client's QGIS project end up with a multitude of layers that could be used for analysis of urban development and more. Moreover the client can visualise and explore the data in 2D or in 3D with custom styles and by using either the built-in 3D map of QGIS, or another plugin (e.g. "Qgis2threejs")(Figure 5.6).

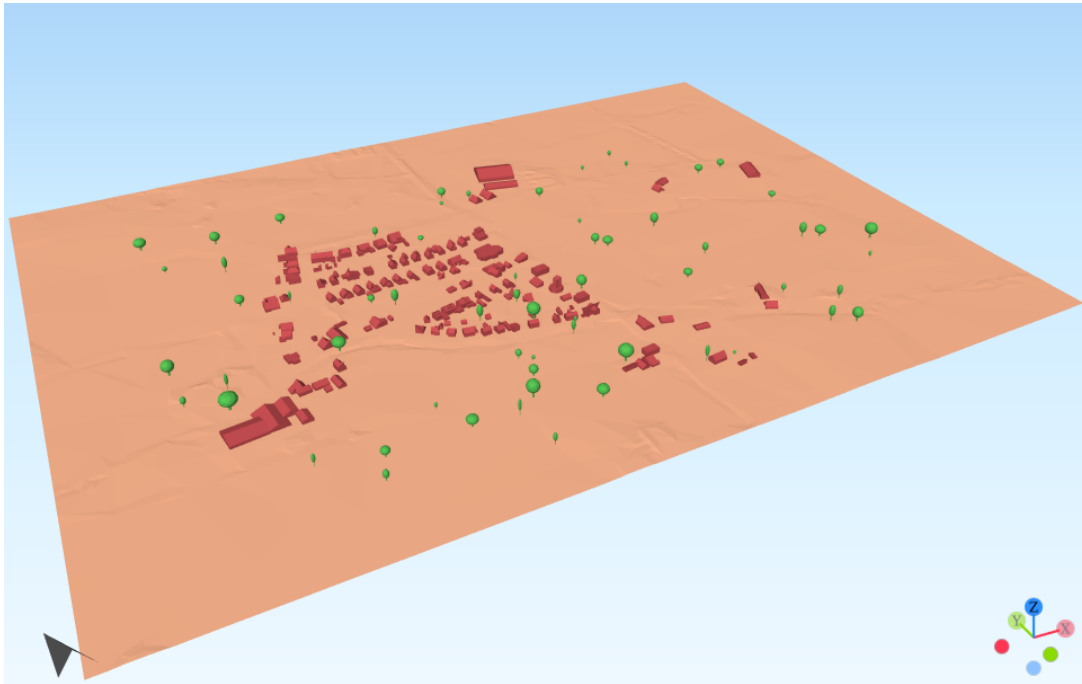


Figure 5.6.: An example semantic 3D city model, imported in QGIS from a 3D City Database using the "3DCityDB-Loader" plugin and visualized using the "Qgis2threejs" plugin. (red="Building" of LOD2, green="SolitaryVegetationObject" of LOD3, light brown="TINRelief" of LOD1)

5.2.5. Database maintenance

After the client's contract ends, the administrator can remove the client user from the database by using the administration action of the plugin. To do this, the administrator just clicks the "Drop schema" button for the selected user which in this scenario is the client. Note, however that this doesn't delete the user from the database. The user "client" could be reused in the future for the same client or other clients with a new password. Alternatively, the administrator can remove the user from the database manually.

5.2.6. Uninstalling the plugin's server-side contents

The scenario ends by the administrator deciding that a new database is going to be exclusively used for 3DCityDB models in combination with the plugin. Consequently, the old installation needs to be removed from the previous database. This can be easily handled by first deleting the "qgis_pkg_usrgroup" user group along with all the users that were created for this purpose, which in this case are the default users "qgis_user_ro", "qgis_user_rw", "employee" and "client". Next, the "Uninstall" button can be finally pressed to clear the database from any content that the plugin has added. In this scenario, the additional schemas are the main schema "qgis_pkg" and only the user schema "qgis_employee" (the client's schema has already been removed in the previous section 5.2.5).

6. Conclusions

This research is focused on the application use aspect of [SQL](#) encoded CityGML models specifically using the 3D City Database open source structure. This endeavour utilized the widely used and easy-to-learn [QGIS](#) as a base for a plugin to bring CityGML at a more approachable to regular users plane. Although a couple of other plugins already exist, their functionalities are extremely limited and the provided user experience is low. The plugin "3DCityDB-Loader" that is created in the context of this research, tries to solve the limitations of the existing plugins while adding a more complete set of functionalities.

Specifically, the "3DCityDB-Loader" seems to be able to reduce the complexity of directly accessing [3DCityDB](#) data. The synergy between the client and server-side aspect of the plugin allows the use of complex operation without imposing any advance technical requirements to user. The client-side [GUI](#), manages to translate the server-side structure into user-friendly widgets. Additionally, the [GUI](#) follows specific software design principals that could facilitate its use and comprehension by the users. In the end, as a result of this research, the plugin is able to provide direct access to [3DCityDB](#) data locally or remotely. This approach could save users time off of their workflows and also attract users with little to no programming experience. To illustrate this point, the example query of section 2.2 is now simplified and reduced in size (Listing 6.2).

That being said, even with simplified queries, users still need to have at least some basic understanding of [SQL](#) in order to work with them. The "3DCityDB-Loader"'s [GUI](#) can solve this issue by utilising the capabilities of [QGIS](#) (Figure 6.1). Moreover, by using [QGIS](#) as the base software on which the plugin is deployed, users can take advantage of a multitude of provided processing and analysis algorithms. These tools can be then used on the [3DCityDB](#) data that are directly accessed using the plugin.

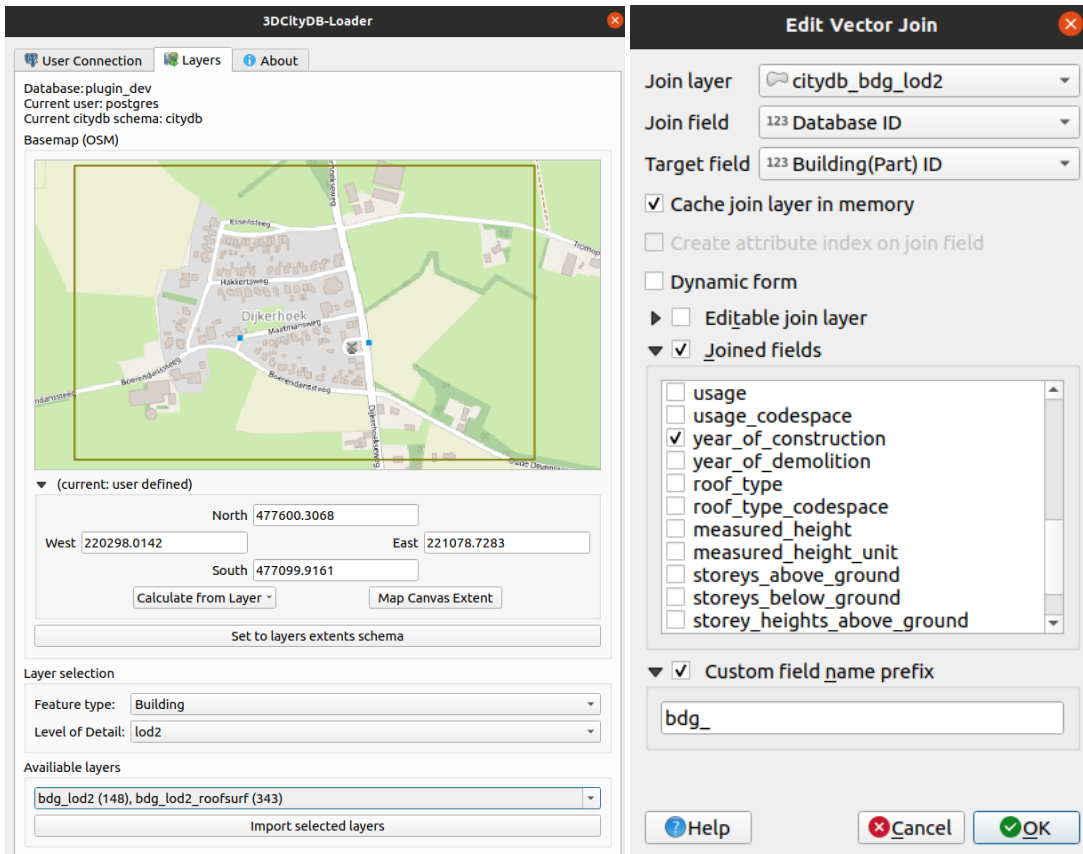
6. Conclusions

```
1 SELECT
2   ts.id AS roof_id,
3   co.ts.gmlid AS roof_gmlid,
4   b.id AS building_id,
5   co.gmlid AS building_gmlid,
6   b.year_of_construction,
7   ST_Collect(sg.geometry) AS
   roof_geom
8 FROM
9   citydb.thematic_surface AS ts
10  INNER JOIN citydb.cityobject AS
   co_ts
11    ON (co_ts.id = ts.id)
12  INNER JOIN citydb.
   surface_geometry AS sg
13    ON (ts.lod2_multi_surface_id
   = sg.root_id)
14  INNER JOIN citydb.building AS b
15    ON (b.id = ts.building_id)
16  INNER JOIN citydb.cityobject AS
   co
17    ON (co.id = b.id)
18 WHERE
19   ts.objectclass_id = 33 AND --
   roofsurfaces
20   b.objectclass_id = 26 AND --
   buildings
21   b.year_of_construction >=
   '2015-01-01'::date
22 GROUP BY
23   ts.id,
24   co_ts.gmlid,
25   b.id,
26   co.gmlid,
27   b.year_of_construction
28 ORDER BY
29   b.id,
30   ts.id;
```

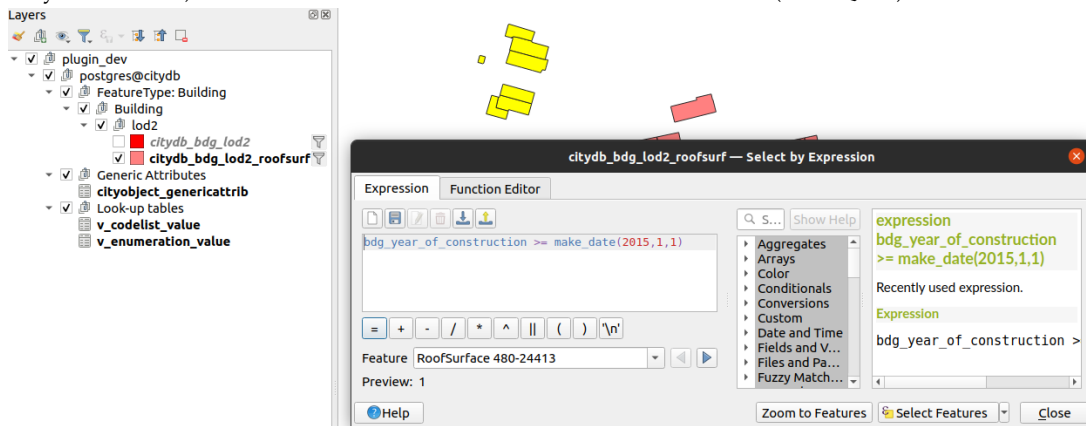
Listing 6.1: Accessing roof surfaces of buildings constructed from 2015 to now. (Using vanilla 3DCityDB)

```
1 SELECT
2   rs.id AS roof_id,
3   rs.gmlid AS roof_gmlid,
4   rs.building_id AS bdg_id,
5   b.gmlid AS bdg_gmlid,
6   b.year_of_construction,
7   rs.geom AS roof_geom
8 FROM
9   qgis_user_ro.
   citydb_bdg_lod2_roofsurf AS rs
10  INNER JOIN qgis_user_ro.
   citydb_bdg_lod2 AS b
11    ON b.id = rs.building_id
12 WHERE
13   b.year_of_construction >=
   '2015-01-01'::date
14 ORDER BY
15   b.id,
16   rs.id;
```

Listing 6.2: Accessing roof surfaces of buildings constructed from 2015 to now. (Using server-side of "3DCityDB-Loader")



(a) Loading layers for roofs and buildings (from "3DCi- (b) Joining layers to get "Year of Construction" field (from QGIS).



(c) Applying selection filter (in yellow) for building roofs build from 2015 (from QGIS).

Figure 6.1.: Converting SQL queries (Listings 6.1,6.2) into QGIS no-code operations.

6. Conclusions

Furthermore, the plugin manages (using the [QGIS Python API](#)) to automatically modify the [QGIS](#) project to accommodate for the loaded data. This restructuring is meant to simplify the various relations that exist between the layers and their attributes. For people with little experience in the CityGML and/or [3DCityDB](#) schema structure, it is easy to miss some of the features' underlying elements. Such elements could be enumeration, codelist tables and other less obvious classes like the generic attributes. The "3DCityDB-Loader" handles these situation by automatically bringing-in these tables and setting-up the appropriate relations between them and the referenced layers. That said, as of the plugin's version 0.4, some of these cases are not supported. There are explained in the limitations (Section 6.2.1).

Additionally, in a similar manner, the plugin sets-up some of the properties of the [QGIS](#) layers. Specifically, by automatically structuring the fields of the layer into a comprehensive attribute form, users may find it easier to navigate through them. As the attribute data relate to different types and CityGML classes, it could be confusing to display them as a single table (Figure 6.2a) and in some cases impossible (e.g. nested table like "genericAttributes"). Consequently, the plugin utilizes the "attribute form" feature which facilitates readability and adds functionality (Figure 6.2b).

id	Database ID	GML ID	Name	Description	Year of construction	Year of demolition	Stores above ground	Stores below ground	Height	Roof type	UoM	Storeys above ground	Storeys below ground	UoM	Storeys above ground	Storeys below ground	UoM
1	29336	NL.IMBA...	Build...		12-0...				14 m	slant...		0					
2	28509	NL.IMBA...	Build...		12-0...				14 m	slant...		0					
3	24918	NL.IMBA...	Build...		12-0...				45 m	slant...		1					
4	3141	NL.IMBA...	Build...		12-0...				45 m	slant...		2					
5	14089	NL.IMBA...	Build...		12-0...				45 m	slant...		2					
6	27299	NL.IMBA...	Build...		12-0...				45 m	slant...		3					
7	31961	NL.IMBA...	Build...		12-0...				45 m	slant...		2					
8	14070	NL.IMBA...	Build...		12-0...				45 m	slant...		3					

(a) Attribute Table of a building layer.

Building 480-1015

Database ID: 29336
 GML ID: NL.IMBAG.Pand.174210000014553
 Name: Building 480-1015
 Description: NULL

Class: Unknown
 Usage: NULL

Feature-specific attributes

Year of construction: 2005
 Year of demolition: NULL
 Stores above ground: NULL
 Stores below ground: 0
 Height: 45
 UoM: m
 Storey height above ground: NULL
 UoM: NULL
 Storey height below ground: NULL
 UoM: NULL
 Roof type: slanted
 Codespace: NULL

(b) Attribute Form of a building layer.

Figure 6.2.: Visual comparison between Attribute Table and Form structure of a building layer.

6.1. Research questions and answers

- Q1 How can QGIS be extended via a plugin to connect and use 3DCityDB in a user-friendly way?
- A Create easy-to-use plugin for QGIS and 3DCityDB using Qt5 and the Python API as they are open source and widely adopted.
- Q2 How can an interface be developed so that the data in the 3DCityDB can be easily accessed (both attributes and geometries) by non-expert users?
- A Layers must be constructed from multiple 3DCityDB tables as a combination of attributes and geometries following the SFS model. Moreover, layers should be user-owned and stored in user-specific schemas.
- Q3 What capabilities are considered both user-friendly and practical enough to be appreciated by both inexperienced and expert users?
- A View features for analysis, Update/Delete/Insert for modification, Multi-user schemas for management, User privileges for control, TOC for organization, Attribute form for diligence.
- Q4 How to balance between the complexity of CityGML's database model and UI/UX?
- A Simple to use widgets in the GUI should be linked to complex server-side functions. Additionally, a GUI should be designed to guide the user with hints (disabled widgets), notification and messages.
- Q5 How to take advantage of the benefits of database stored data to be used into the QGIS environment?
- A Utilization of the SFS model into 3DCityDB with materialized views for the geometry field. Use of database views as QGIS vector layers. Make use of PL/pgSQL server-side functions to handle complex operations.
- Q6 How to mediate between the possibly huge amount of data stored in a database, and the limited resources (or user's needs) in terms of data within QGIS?
- A Provide users with the means to select sub-areas of interest. Give the option to generate layers for a specific area (server-side). Give further the option to load layers from an even smaller area (client-side).

6.2. Discussion

The results seems to prove that a comprehensive GUI works well together with complex server-side operations. Additionally, as both software were jointly developed, at the time of conducting this research, this plugin is the front-end of what could be considered a full-stack application.

Although the server-side installation can stand on its own, the plugin adds important functionalities to help also users with little or no programming skills, or no need/desire to work via a command console. Working directly from within the database, while it allows for more

6. Conclusions

freedom of operation, it adds more complexity that could hinder the workflow of even experienced users. The plugin successfully handles automatically many queries (spatial, semantic filters) of the database using a straight forward GUI to get the user's input in a comprehensive way. Moreover, the automatic re-structuring of the QGIS project which is unique to the plugin, can potentially save for users a lot of preparation time.

As proposed in chapter 5, it can find practical implementation as a tool for research and analysis applications. Finally, this method alleviates most of the complexity of the practical use of data in 3DCityDB, while utilizing the database's inherent benefits of storing and querying datasets. These advantages could become a turning factor for individuals and organizations to move to an SRDBMS way of storing 3D city models and attract more users to join the community of 3DCityDB. In particular for open source projects, users are a crucial part of their existence, as some users give feedback for future development, other users volunteer as developers to maintain and drive the development forwards, and others even provide funds as incentive to drive more focused innovations.

That being said, the results of this research are evaluated based on user feedback of the testers and the assessment of the involved actors (author, supervisors). The evaluation consisted mainly of opinions on existing features, difficulty level of use, practicality of use and comparison with other tools. To reinforce these findings, it would be beneficial to realize or use existing standardise metrics in order quantify the results.

Furthermore, an important part of the methodology consisted of meetings between the relevant actors. The contents of these discussions relate to the requirement identification (Table 3.1), reasoning about the state of software development and assessment of preliminary results. Although the results coming out of these discussion (implemented software) are tracked in project's GitHub repository, the meeting contents themselves were not thoroughly registered. Those are stored as personal notes into a private directory.

Finally, regarding the iterative process of development, it is important to recognise the value of the assessment and requirement identification phase. In this research, the development process went through four different cycles. A well-organized path and planning of intermediate goals can determine the amount of cycles required to reach expected results. In this case, for a research that is linked to experimental software development, it proved to be beneficial. Additionally, this process resembles the "scrum philosophy" used often in software development which "employs an iterative, incremental approach to optimize predictability and to control risk" [Schwaber and Sutherland, 2011]. It would be interesting to assess if a dedicated down-scaled scrum approach can achieve better synergy with researches of similar nature.

6.2.1. Limitations

At this stage of the plugin (v0.4) and server-side installation (v0.7.0), there are some limitations that need to be addressed.

One limitation is that the plugin in combination with the server-side installation was developed exclusively for PostgreSQL databases. However, 3DCityDB also supports the Oracle database which could also benefit from a QGIS integration. Oracle databases have different structure and functionalities from PostgreSQL, thus more research is needed to attempt a migration. Additionally, PostgreSQL was favoured over Oracle, as is an open source project similar to 3DCityDB and the "3DCityDB-Loader" plugin. Lastly, the exact possibilities of an Oracle integration were not explored as it is out of scope for this research.

Regarding the multi-user approach, users (administrators) need to take action outside of the plugin's GUI in order to prepare or clean the database. Users need to be created either from an SQL query or from the PSQL terminal application. Then the "Main" schema needs to be installed by the plugin or manually in order to get access to the PL/pgSQL functions that grant and revoke the Read-Only and/or Read-Write privileges. However, this function needs to be called manually outside of the plugin's environment. Leaving the plugin's environment at an intermediate stage could be confusing for users or lead to disorderly operations that could cause software issues (crashes or unintended behaviour). That said, this consists of a functionality limitation derived from a design decision.

The above limitations relate more to the plugin's current capabilities. In the next paragraphs, follow some limiting factors that could hinder the user experience.

First of all, although materialized views have the benefit of fire-and-forget use, they are computationally expensive for geometries (at the first time or on every refresh). Depending on the extents of the data and the system's hardware specifications, it could consume a lot of time from the user and processing power from the server. As a preventative measure, when such operation is called, the plugin warns the user about it with a pop-up message giving the option to cancel it (Figure 6.3).

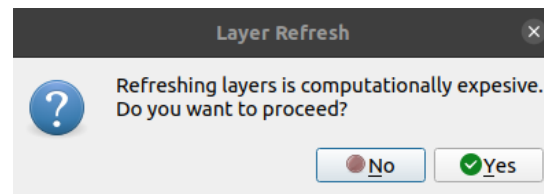


Figure 6.3.: Warning message before refreshing materialized views.

In relation to the above, although the users have the ability to choose their own extents in order to avoid generating layers for the entire schema's extents, there is nothing that obstructs them from not utilizing this feature at all. This in practice means, that unintentionally or not, users could generate and refresh layers for a vast area and huge amount of data, reserving unusual amounts of server resources. Once started, this process can only be killed manually, by terminating the running transaction from within the server. This results to a database connection error, which is getting caught by the plugin and blocks any other incoming transactions. In this case, it is suggested for the user to click the "Close current connection" button as regardless of an existing open connection, it resets the plugin to its initial state.

Next, regarding data visualisation in 3D, QGIS may fail to render successfully every feature. In practice, this situation was mainly observed in wall surfaces where the rendered polygon boundaries seem to differ from their actual boundaries (Figure 6.4). The current assumption is that this potential issue with 3D visualizations is caused from precision loss of projected coordinates. Moreover, we can guarantee that this issue doesn't originate from invalid data, as the same features are being correctly rendered in other software. The additional software that was tested is the "FME" and the "Google Earth". Although, this issue isn't relevant to the plugin's methodology and implementation, it is important to disclose it so that users are aware of it.

6. Conclusions

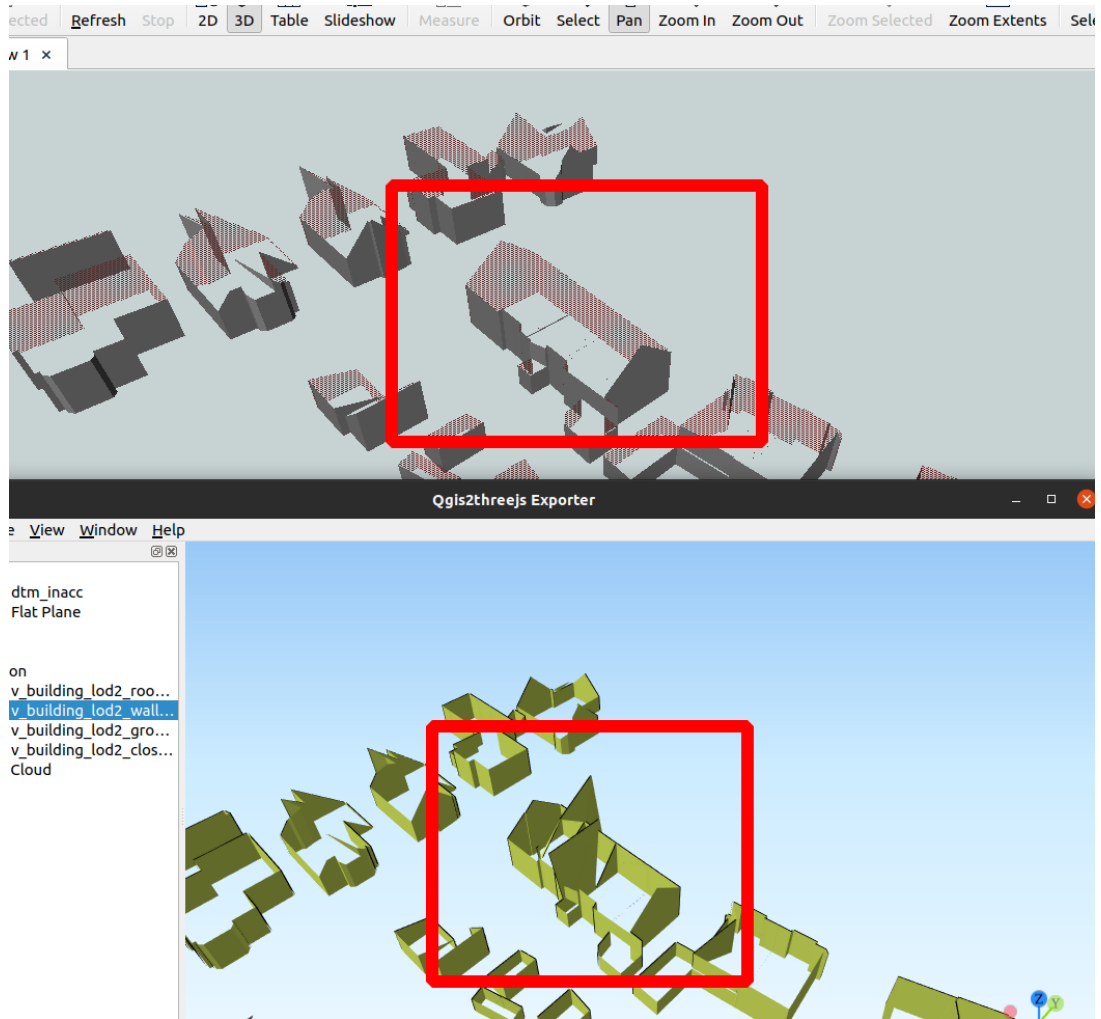


Figure 6.4.: Example of QGIS 3D rendering artefacts (bottom). In FME the artefacts are not present (top).

Lastly, another limitation coming from QGIS, concerns the inability for the plugin to use the QGIS defined push button of "Draw on Canvas". This widget corresponds to a built-in method to the "QgsExtentGroupBox" class, which allows for a custom temporary square to be drawn by left-clicking dragging and releasing on a map. This square is used to extract and display the N-E-S-W points in the group box. However, "Draw on Canvas" does not currently work as intended. In the core QGIS code, there is a hard-coded "True" value that causes the parent dialog to toggle its visibility in order, presumably, to let the user draw on the map. But in our case the parent dialog contains the canvas (base-map) that we need to draw upon. Note that QGIS uses a canvas object to draw the project's layer on. The plugin uses its own canvas object to draw the extents and the OSM layer on. Re-opening the plugin allows us to draw on the plugin's canvas but with the caveat that the drawing tool never closes (it also caused a lot of QGIS crashes).

6.2.2. Future Development

For future development, it would be interesting to focus on solving or accounting for the above-mentioned limitations. Additionally, derived from discussion between the relevant actors of this research, new functionalities are identified that could further evolve the usability of the plugin.

To begin with, as mentioned before, the plugin and its server-side operations are developed for PostgreSQL. A similar research could be conducted to access the possibility to migrate the methodology for Oracle database use. Depending on the similarities between the two database types, this endeavour could either result to a seamless integration to the current plugin or reveal the need to create a new fully dedicated one. Either way, doing so, is going to benefit the Oracle based community of [3DCityDB](#) users.

Next, in the scope of this research, the development is applied for CityGML 2.0. In short, the main reason for this choice is that this version is tried and tested through the years. In fact, this version is in circulation for almost 10 years already, meaning that most 3D city models, up to this date, are structured according to this standard. That being said, now that the new CityGML 3.0 is published, the [3DCityDB](#) is going to follow to match the new version. Thus, for future development, it is required to assess if the methodology of this research could be replicated for the CityGML 3.0 and the related [3DCityDB](#) future version.

In the last paragraphs, the future development is focused solely on adding or amending functionalities of the plugin. These ideas are tracked in the "Issues" tab of the GitHub repository of this project [[Pantelios and Agugiario, 2022](#)].

To start with, the CityGML elements of "Appearances" and "Addresses" are not yet supported. As these are different elements than the classes of feature types, a different approach needs to be tested and implement for use. Additionally, this issue also relates to the [ADE](#) functionality, which also needs to be assessed.

Derived from the corresponding limitation, administrators need to temporarily exit the plugin to handle users manually. This approach could be refined and changed so that these operations can be handled from the GUI. A very important design principle of this plugin is to make the order of operation as intuitive as possible. Consequently, it is prudent to align this part of operations into this principle.

Regarding the plugin's base-maps, a new functionality is to add a geocoder to help user navigation. Geocoders are useful tools that convert an address (full or partial) like a country or a city into map coordinates zooming into that location [[Florczyk et al., 2009](#)]. This gives users the option to navigate on the base-map without having to rely on visual cues.

Next, another functionality, similarly to "3DCityDB Importer/Exporter", is to let users choose to create layers only for particular Feature types. For example, in the scenario simulation in chapter 5, the employee only wanted to update some buildings, but ended up creating layers for all available feature types in the area. It would save some resources if he/she could elect to create layers only for the "Building" feature type. This might not show any noticeable difference for the particular scenario, however it would, for the case where a huge amount of data is present in the area.

In this research, particularly in the server-side operation, the PostgreSQL's functionalities are extensively used (triggers, views, etc). It would be beneficial to explore how other functionalities could be used to solve some of the limitations of the plugin or introduce new features.

6. Conclusions

For version 0.4, an applicable idea is to use multiple simultaneous transactions that each generates or refreshes a different layer. This could prove to be more time efficient from the current implementation of sequential transactions.

Finally, regarding the properties of the layers that are saved as QML files, they must be tested for all possible features. Currently, specific layer properties like the custom color symbology and field constraints are only tested for features that existed in the tested data-sets. However, the data-sets didn't contain every single CityGML feature, meaning that for the missing features the custom properties remain to be tested.

A. Reproducibility self-assessment

A.1. Marks for each of the criteria

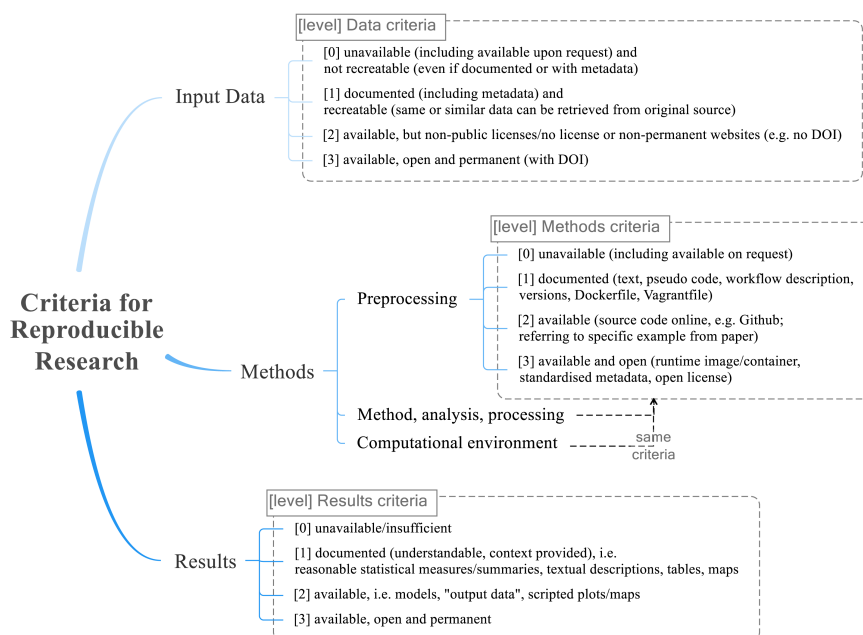


Figure A.1.: Reproducibility criteria to be assessed.

2/3 Input data

2/3 Preprocessing

3/3 Methods

3/3 Computational environment

3/3 Results

A.2. Self-reflection

A.2.1. Input data

Regarding the data used in the research, these were procured from various open online sources. However, in order to group and organise them, they were stored in the project's

A. Reproducibility self-assessment

GitHub repository [Pantelios and Agugiaro, 2022]. It is important to note that one data-set (Rijsen-Holten), which was provided by the supervisors of this research, is stored in a private Google Drive directory due to size limitations. Although, all URLs are open to anyone, their location are not permanent meaning that access to them in the future could not be possible.

A.2.2. Methods

In the context of this research preprocessing is considered the requirement identification phase which was conducted in meetings and discussion. Information about those is not available and poorly documented. That being said, the preprocessing phase also contains the initial prototype software which is tacked in the project's GitHub repository [Pantelios and Agugiaro, 2022].

Regarding the methods that were developed for this research, the same GitHub repository was used for code collaboration, versioning and storage. The software developed in this research is considered open source. It is a free software which can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation version 2 of the License. Additionally, the software is pre-compiled and installation-ready.

About the computational environment, everything was developed or built-upon open source software that can be accessed from multiple sources (e.g. Docker images, source code, dedicated installers).

A.2.3. Results

Finally, as results can be considered the final "3DCityDB-Loader" plugin accompanied with this research document. Both of these are stored and are publicly available into the educational repository of Delft University of Technology.

B. "3DCityDB-Loader" characteristics

This part of the appendix contains some generic details regarding the plugin structure and [GUI](#).

B.1. Layer properties

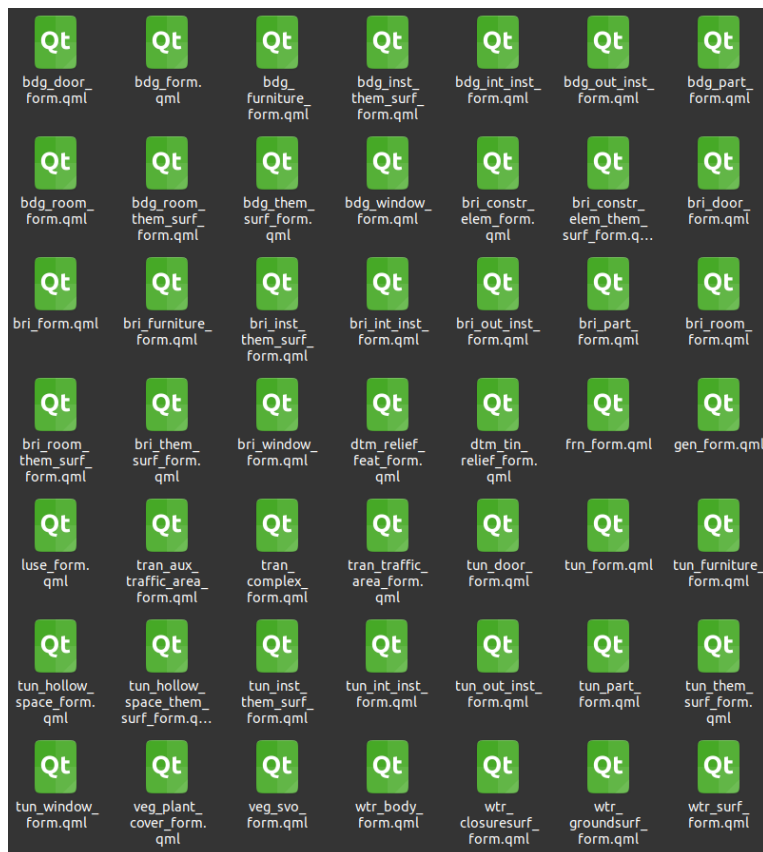


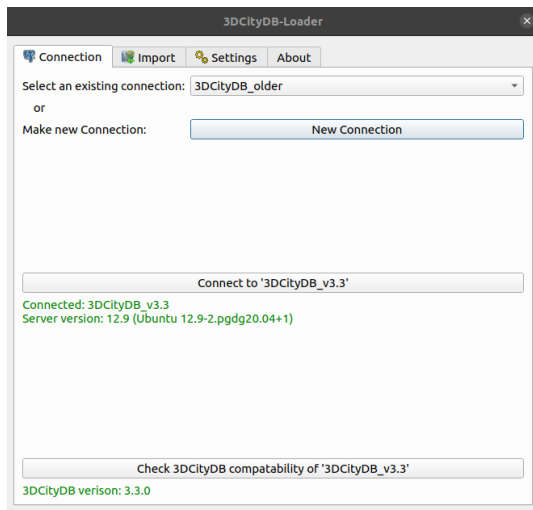
Figure B.1.: Pre-structured QML files are stored in the "forms" directory for a multitude of different CityGML features. These hold the layer properties rules (symbology and attribute forms). Users can modify these files manually or by changing the properties from QGIS "layer properties" and overwriting them the corresponding file.

B. "3DCityDB-Loader" characteristics

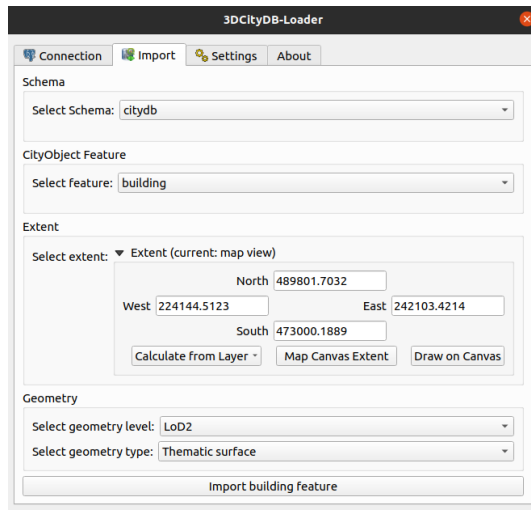
root_class	layer_name	qml_file	hex_code	alpha_value	color
Building	bdg_lodx	bdg_form.qml	#ff0000	1	Red
Building	bdg_lod0_footprint	bdg_form.qml	#ff0001	1	Red
Building	bdg_lod0_roofedge	bdg_form.qml	#ff0002	1	Red
Building	bdg_lodx_roofsurf	bdg_thematic_surface_form.qml	#ff0003	0.5	Red
Building	bdg_lodx_wallsurf	bdg_thematic_surface_form.qml	#ff0004	0.5	Red
Building	bdg_lodx_groundsurf	bdg_thematic_surface_form.qml	#ff0005	0.5	Red
Building	bdg_lodx_closuresurf	bdg_thematic_surface_form.qml	#ff0006	0.5	Red
Building	bdg_lodx_outerceilingsurf	bdg_thematic_surface_form.qml	#ff0006	0.5	Red
Building	bdg_lodx_outerfloorsurf	bdg_thematic_surface_form.qml	#ff0006	0.5	Red
Building	bdg_out_inst_lodx	bdg_out_installation_form.qml	#51fff6	1	Cyan
Building	bdg_window_lodx	bdg_opening_form.qml	#2bcaff	0.5	Blue
Building	bdg_door_lodx	bdg_opening_form.qml	#9b3c25	1	Brown
Building	bdg_room_lod4	bdg_room_form.qml	#ffff00	0.5	Yellow
Building	bdg_room_lod4_ceilingsurf	bdg_room_thematic_surface_form.qml	#ff9100	0.5	Orange
Building	bdg_room_lod4_intwallsurf	bdg_room_thematic_surface_form.qml	#ff9100	0.5	Orange
Building	bdg_room_lod4_floorsurf	bdg_room_thematic_surface_form.qml	#ff9100	0.5	Orange
Building	bdg_furniture_lod4	bdg_furniture_form.qml	#9b3c25	1	Brown
Bridge	bri_lodx	bri_form.qml	#969696	1	Grey
Bridge	bri_out_inst_lodx	bri_out_installation_form.qml	#51fff6	1	Cyan
Bridge	bri_constr_elem_lodx	bri_constr_element_form.qml	#969696	1	Grey
CityFurniture	city_furn_lodx	city_furn_form.qml	#d5b43c	1	Gold
Generics	gen_cityobj_lodx	gen_cityobj_form.qml	#5c5c5c	1	Dark Grey
Tunnel	tun_lodx	tun_form.qml	#969696	1	Grey
Tunnel	tun_lodx_roofsurf	tun_thematic_surface_form.qml	#969696	0.5	Grey
Tunnel	tun_lodx_wallsurf	tun_thematic_surface_form.qml	#969697	0.5	Grey
Tunnel	tun_lodx_groundsurf	tun_thematic_surface_form.qml	#969698	0.5	Grey
Tunnel	tun_lodx_closuresurf	tun_thematic_surface_form.qml	#969699	0.5	Grey
Tunnel	tun_lodx_outerceilingsurf	tun_thematic_surface_form.qml	#969700	0.5	Grey
Tunnel	tun_lodx_outerfloorsurf	tun_thematic_surface_form.qml	#969701	0.5	Grey
Tunnel	tun_out_inst_lodx	tun_out_installation_form.qml	#51fff6	1	Cyan
Transportation	railway_lodx	railway_form.qml	#585858	1	Dark Grey
Relief	relief_feat_lodx	relief_feat_form.qml	#176626	0.5	Green
Relief	tin_relief_lodx	tin_relief_form.qml	#0d6616	0.75	Green
Vegetation	sol_veg_obj_lodx	sol_veg_obj_form.qml	#00ff00	1	Green
WaterBody	waterbody_lodx	waterbody_form.qml	#003ad5	1	Blue
WaterBody	waterbody_lodx_watersurf	waterbody_form.qml	#003ad5	0.5	Blue
WaterBody	waterbody_lodx_watergroundsurf	waterbody_form.qml	#0d6616	0.5	Green

Figure B.2.: Available custom color schema (v0.4). This symbology is stored in the QML files (**qml_file**) that accompany each layer (**layer_name**).

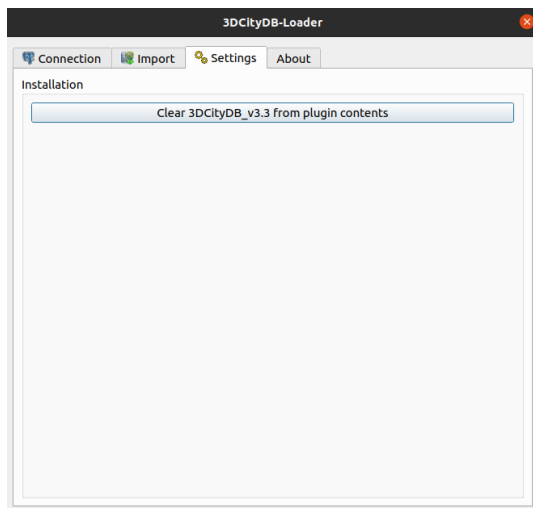
B.2. GUI design evolution



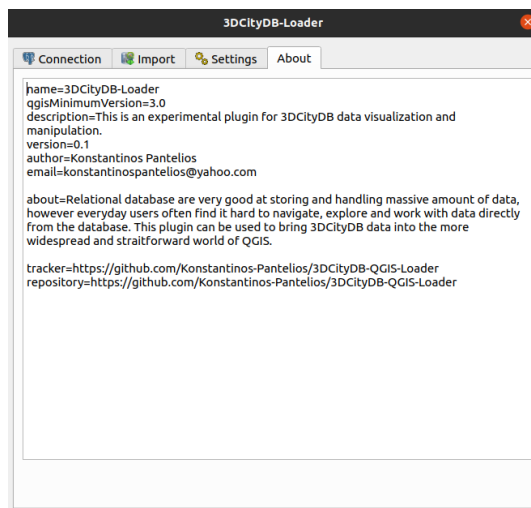
(a) Connection Tab



(b) Import tab



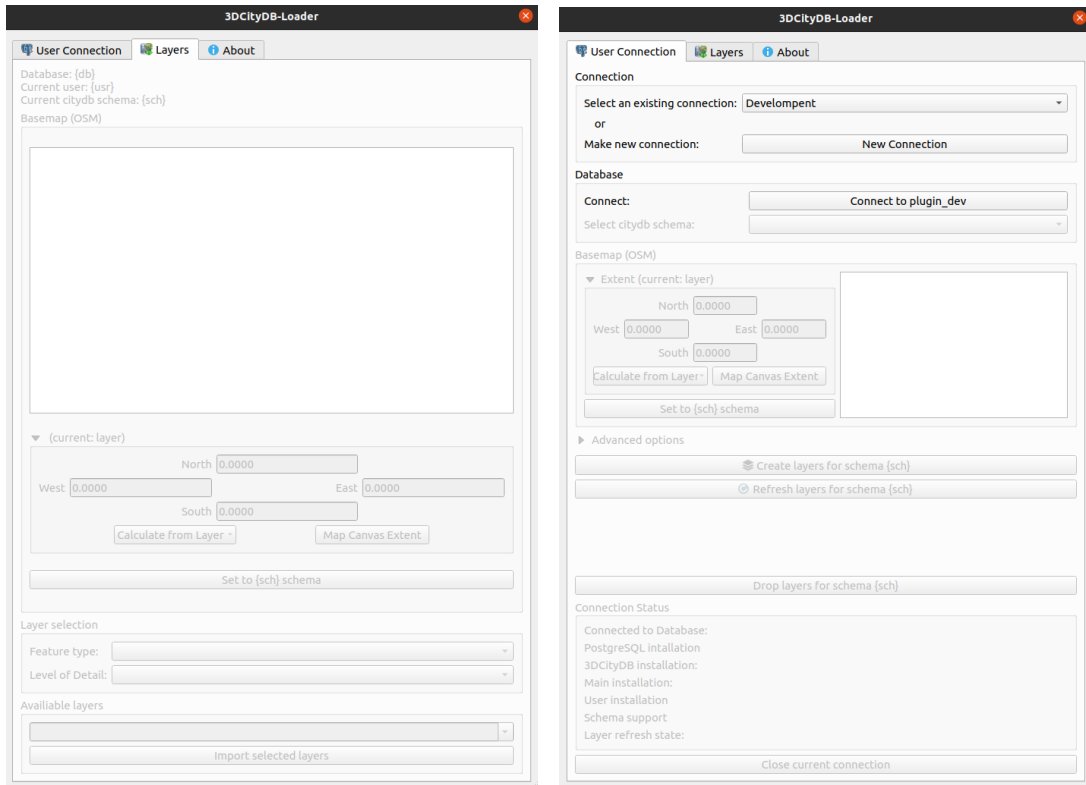
(c) Settings tab



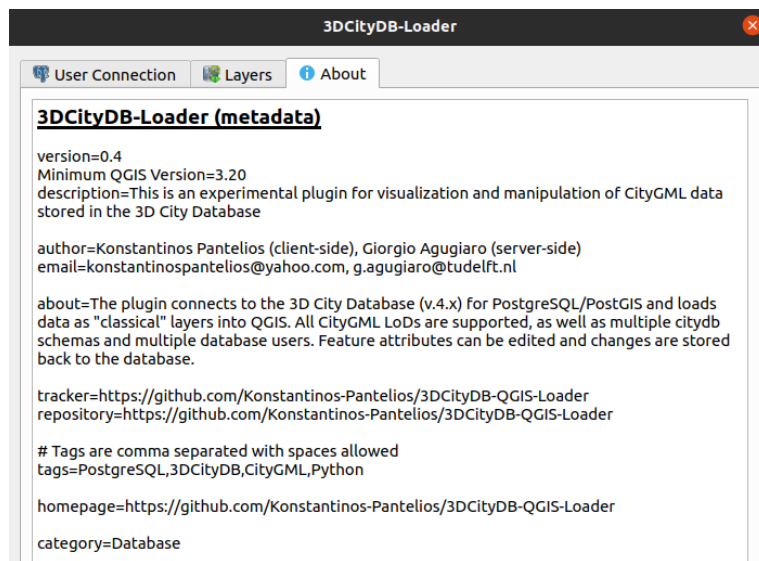
(d) About tab

Figure B.3.: Old "3DCityDB-Loader" plugin design (as of 05/01/2021 v0.1)

B. "3DCityDB-Loader" characteristics



(a) "3DCityDB-Loader" initial GUI state ("User Connection" tab). (b) "3DCityDB-Loader" initial GUI state ("Layers" tab).



(c) "3DCityDB-Loader" initial GUI state ("About" tab).

Figure B.4.: Current "3DCityDB-Loader" plugin design (as of 29/06/2022 v0.4)

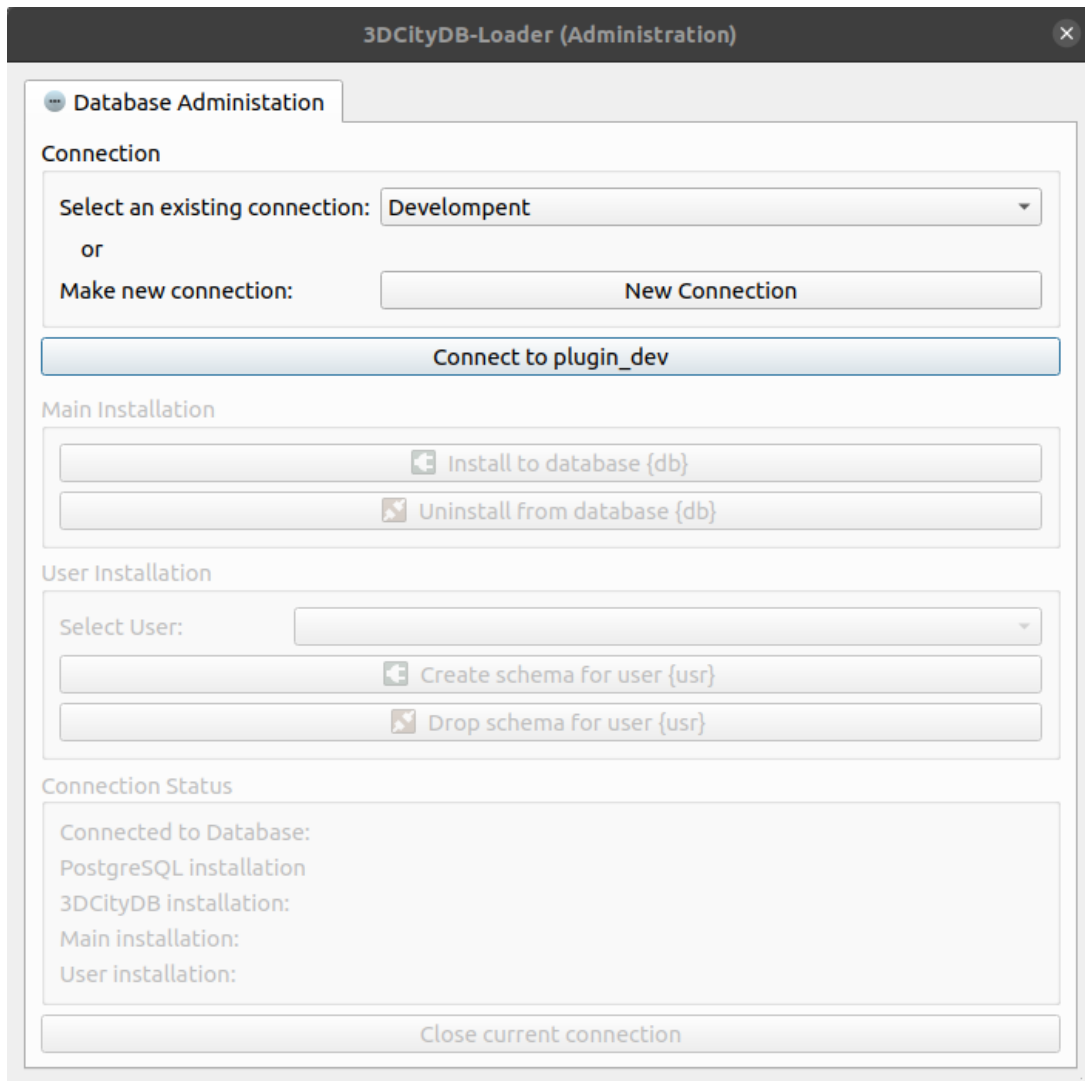


Figure B.5.: "3DCityDB-Loader (Administration)" initial GUI state ("Database Administration" tab).

B. "3DCityDB-Loader" characteristics

B.3. Test data-sets

The images below illustrate the test data-sets loaded in QGIS using the "3DCityDB-Loader" plugin. The "Qgis2threejs" plugin was used for 3D visualisation.

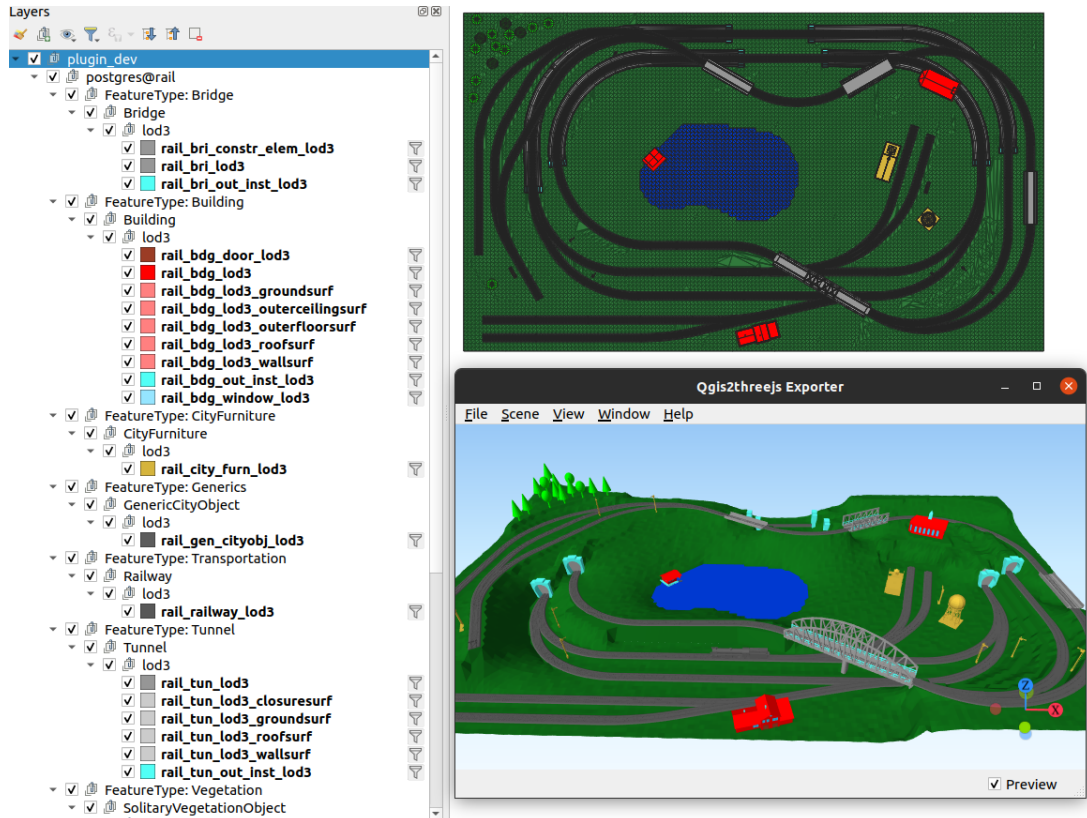


Figure B.6.: Railway data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).

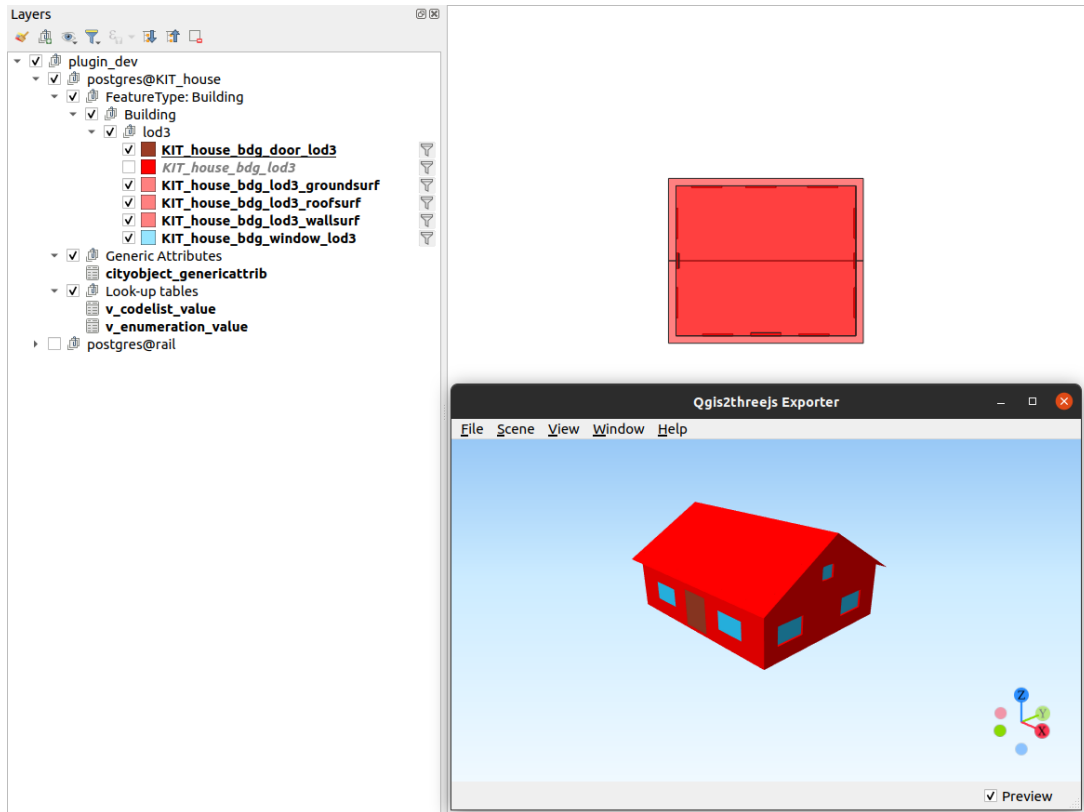


Figure B.7.: House data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).

B. "3DCityDB-Loader" characteristics

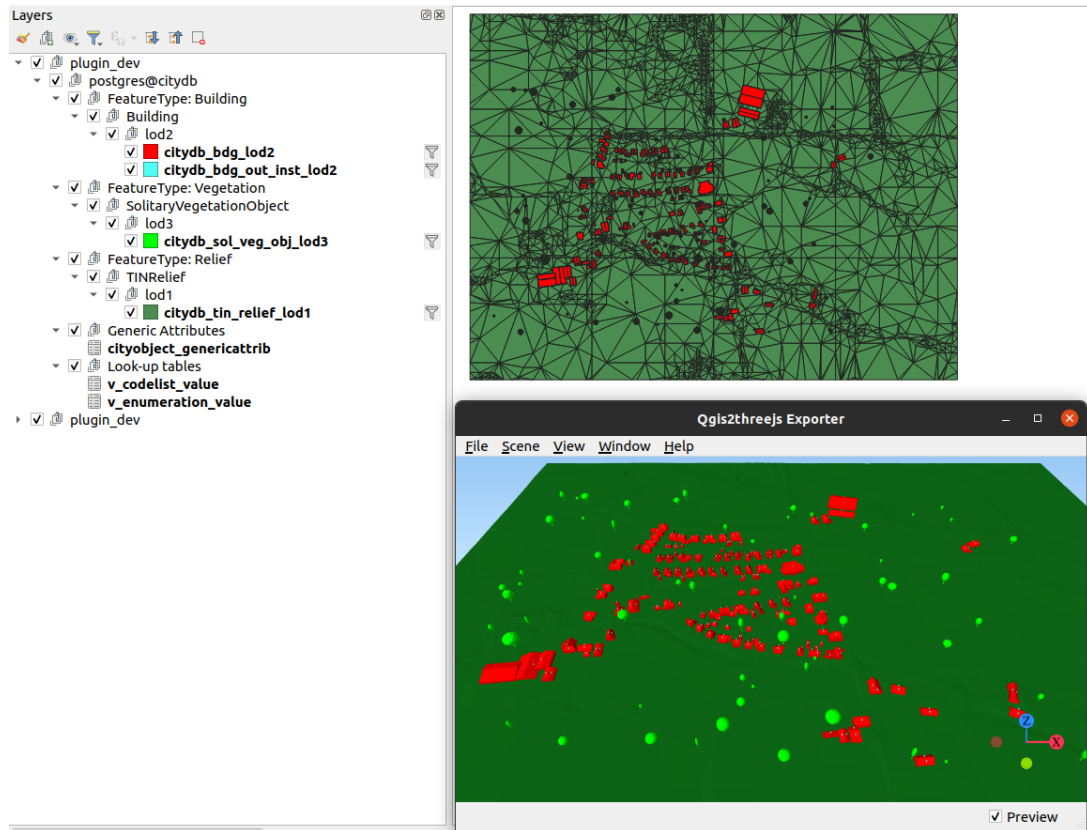


Figure B.8.: Rijssen-Holten data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).

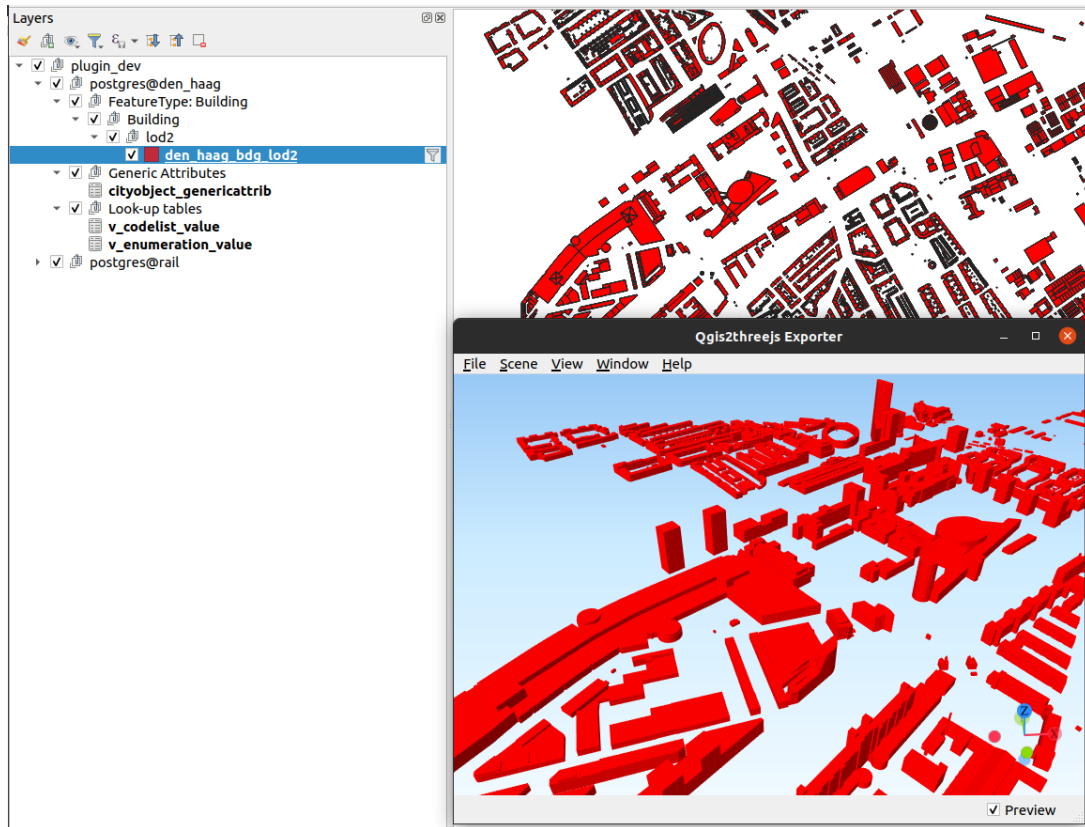


Figure B.9.: Den Haag data-set loaded in QGIS with "3DCityDB-Loader" (Table 3.2).

Bibliography

- 3DCityDB (2021a). 3d city database documentation. <https://3dcitydb-docs.readthedocs.io/en/latest/3dcitydb/index.html>. Accessed: 2022-05-03.
- 3DCityDB (2021b). Geometry schema. <https://3dcitydb-docs.readthedocs.io/en/latest/3dcitydb/schema/geometry.html>. Accessed: 2022-06-13.
- Aberham, C. A. (2021). 3dcitydb-viewer. <https://github.com/aberhamchristomus/3DCityDB-Viewer>. Accessed: 2022-05-03.
- Agugiario, G. (2018). Citygml 3d city database utilities package postgresql version. https://github.com/gioagu/3dcitydb_utilities/blob/master/manual/3DCityDB_Uilities_Package_Documentation.pdf. Accessed: 2022-05-17.
- Agugiario, G. and Holcik, P. (2017). 3d city database extension for the citygml energy ade 0.8 postgresql version. https://github.com/gioagu/3dcitydb_energy_ade/blob/4074ae7c0bffc48d464aca0c942aec2fc9585e7b/manual/3DCityDB_Energy_ADE_0.8_Documentation.pdf. Accessed: 2022-05-17.
- Albert, J., Bachmann, M., and Hellmeier, A. (2003). Zielgruppen und anwendungen für digitale stadtmodelle und digitale geländemodelle. *Erhebungen im Rahmen der SIG 3D der GDI NRW*.
- 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.
- Casagrande, L. et al. (2021). 3dcitydb-qgis-explorer. <https://github.com/3dcitydb/3dcitydb-qgis-explorer>. Accessed: 2022-05-03.
- Corrado, E. M. (2005). The importance of open access, open source, and open standards for libraries. *Issues in science and technology librarianship*.
- Costamagna, E. and Spanò, A. (2013). *CityGML for Architectural Heritage*, pages 219–237. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Di Gregorio, F. and Varrazzo, D. (2021). Psycopg – postgresql database adapter for python. <https://www.psycopg.org/docs/>. Accessed: 2022-05-17.
- Florczyk, A., López-Pellicer, F., Gayán-Asensio, D., Rodrigo-Cardiel, P., Latre, M., and Nogueras-Iso, J. (2009). Compound geocoder: get the right position. In *GSDI 11 World Conference and the 3rd INSPIRE Conference*.
- Gemeente-Den-Haag (2021). 3d stadsmodel den haag 2018 citygml. <https://denhaag.dataplatform.nl/#/data/36049d1a-4a0f-4c5d-8adb-21dbfb7252f9>. Accessed: 2022-06-12.

Bibliography

- Gröger, G. and Plümer, L. (2012). Citygml–interoperable semantic 3d city models. *ISPRS Journal of Photogrammetry and Remote Sensing*, 71:12–33.
- 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.
- Gupta, A., Mumick, I. S., et al. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18.
- Herring, J. (2001). *The OpenGIS Abstract Specification, Topic 1: Feature Geometry (ISO 19107 Spatial Schema)*, volume 5. Open Geospatial Consortium.
- Herring, J. et al. (2011). *Opengis® implementation Standard for Geographic information - Simple feature access - Part 2: SQL option [corrigendum]*. Open Geospatial Consortium.
- Hughes, J. F., Van Dam, A., McGuire, M., Foley, J. D., Sklar, D., Feiner, S. K., and Akeley, K. (2014). *Computer Graphics: Principles and Practice*. Pearson Education.
- Häfele, K.-H., of Technology, K.-I., and for Applied-Computer-Science, I. (2020). First v2.0 citygml file (aka railway). <https://nervous-ptolemy-d29bcd.netlify.app/samplefiles/>. Accessed: 2022-06-12.
- Järvi, J., Marcus, M., Parent, S., Freeman, J., and Smith, J. (2009). Algorithms for user interfaces. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering, GPCE '09*, page 147–156, New York, NY, USA. Association for Computing Machinery.
- Joo, H. (2017). A study on understanding of ui and ux, and understanding of design according to user interface change. *International Journal of Applied Engineering Research*, 12(20):9931–9935.
- KIT and Campus-North (2021). Fzk haus. https://www.citygmlwiki.org/index.php?title=FZK_Haus. Accessed: 2022-06-12.
- Kolbe, T. H., Kutzner, T., Smyth, C. S., Nagel, C., Roensdorf, C., and Heazel, C. (2021). *OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard*. Open Geospatial Consortium.
- Lake, R. (2005). The application of geography markup language (gml) to the geological sciences. *Computers & Geosciences*, 31(9):1081–1094. Application of XML in the Geosciences.
- Ledoux, H., Ohori, K. A., 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):1–12.
- Leidig, M. and Teeuw, R. (2015). Free software: A review, in the context of disaster management. *International Journal of Applied Earth Observation and Geoinformation*, 42:49–56.
- Lobur, M., Dykhta, I., Golovatsky, R., and Wrobel, J. (2011). The usage of signals and slots mechanism for custom software development in case of incomplete information. In *2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*, pages 226–227.
- Lu, C.-T., Dos Santos, R. F., Sripada, L. N., and Kou, Y. (2007). Advances in gml for geospatial applications. *Geoinformatica*, 11(1):131–157.

- Malakhov, A. (2016). Composable multi-threading for python libraries. In *Proceedings of the 15th Python in Science Conference, Austin, TX, USA*, pages 11–17.
- Marcus, A. (1995). Principles of effective visual communication for graphical user interface design. In BAECKER, R. M., GRUDIN, J., BUXTON, W. A., and GREENBERG, S., editors, *Readings in Human–Computer Interaction*, Interactive Technologies, pages 425–441. Morgan Kaufmann.
- Neteler, M., Bowman, M. H., Landa, M., and Metz, M. (2012). Grass gis: A multi-purpose open source gis. *Environmental Modelling & Software*, 31:124–130.
- Pantelios, K. and Agugiario, G. (2022). 3dcitydb-loader for qgis. <https://github.com/Konstantinos-Pantelios/3DCityDB-QGIS-Loader>. Accessed: 2022-06-12.
- Pasotti, A. (2021). Plugin repository. <https://plugins.qgis.org/plugins/>. Accessed: 2022-05-03.
- Passy, P. and Théry, S. (2018). *The Use of SAGA GIS Modules in QGIS*, chapter 4, pages 107–149. John Wiley and Sons, Ltd.
- Pitoura, E. (2018). Access path. In *Encyclopedia of Database Systems*, pages 22–23. Springer New York, New York, NY.
- QGIS-Python-API (2018a). Class: Qgscheckablecombobox. <https://qgis.org/pyqgis/3.16/gui/QgsCheckableComboBox.html>. Accessed: 2022-05-03.
- QGIS-Python-API (2018b). Class: Qgscollapsiblegroupbox. <https://qgis.org/pyqgis/3.16/gui/QgsCollapsibleGroupBox.html>. Accessed: 2022-05-03.
- QGIS-Python-API (2018c). Class: Qgslyertree. <https://qgis.org/pyqgis/3.0/core/Layer/QgsLayerTree.html>. Accessed: 2022-05-03.
- QGIS-Python-API (2018d). Class: Qgsmapcanvas. <https://qgis.org/pyqgis/3.0/gui/Map/QgsMapCanvas.html>. Accessed: 2022-05-03.
- QGIS-Python-API (2018e). Class: Qgsproject. <https://qgis.org/pyqgis/3.0/core/Project/QgsProject.htmls>. Accessed: 2022-05-03.
- QGIS-Python-API (2018f). Class: Qgsrelation. <https://qgis.org/pyqgis/3.0/core/Relation/QgsRelation.html>. Accessed: 2022-05-03.
- QGIS-Python-API (2018g). Class: Qgsvectorlayer. <https://qgis.org/pyqgis/3.0/core/Vector/QgsVectorLayer.html>. Accessed: 2022-05-03.
- QGIS-Python-API (2022). Qgis python api documentation. <https://qgis.org/pyqgis/3.22/>. Accessed: 2022-05-17.
- Schwaber, K. and Sutherland, J. (2011). The scrum guide. *Scrum Alliance*, 21(1).
- SIG-3D (2012). Index of /codelists/standard. <http://www.sig3d.de/codelists/standard/>. Accessed: 2022-05-03.
- Stadler, A., Nagel, C., König, G., and Kolbe, T. H. (2009). *Making Interoperability Persistent: A 3D Geo Database Based on CityGML*, pages 175–192. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bibliography

- Steiniger, S. and Bocher, E. (2009). An overview on current free and open source desktop gis developments. *International Journal of Geographical Information Science*, 23(10):1345–1370.
- Steiniger, S. and Hunter, A. J. S. (2012). *Free and Open Source GIS Software for Building a Spatial Data Infrastructure*, pages 247–261. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Stolze, K. (2003). Sql/mm spatial: The standard to manage spatial data in a relational database system. In Weikum, G., Schöning, H., and Rahm, E., editors, *BTW 2003 – Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW Konferenz*, pages 247–264, Bonn. Gesellschaft für Informatik e.V.
- The-PostgreSQL-Global-Development-Group (2021a). Create view. <https://www.postgresql.org/docs/current/sql-createview.html>. Accessed: 2022-05-17.
- The-PostgreSQL-Global-Development-Group (2021b). Sql procedural language. <https://www.postgresql.org/docs/current/plpgsql-overview.html>. Accessed: 2022-05-17.
- The-Qt-Company (2018a). Qcombobox class. <https://doc.qt.io/qt-5/qcombobox.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018b). Qdialog class. <https://doc.qt.io/qt-5/qdialog.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018c). Qformlayout class. (<https://doc.qt.io/qt-5/qformlayout.html>). Accessed: 2022-05-03.
- The-Qt-Company (2018d). Qgraphicsview class. <https://doc.qt.io/qt-5/qgraphicsview.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018e). Qgridlayout class. <https://doc.qt.io/qt-5/qgridlayout.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018f). Qgroupbox class. <https://doc.qt.io/qt-5/qgroupbox.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018g). Qhboxlayout class. <https://doc.qt.io/qt-5/qhboxlayout.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018h). Qlabel class. <https://doc.qt.io/qt-5/qlabel.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018i). Qpushbutton class. <https://doc.qt.io/qt-5/qpushbutton.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018j). Qtabwidget class. <https://doc.qt.io/qt-5/qtabwidget.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018k). Qvboxlayout class. <https://doc.qt.io/qt-5/qvboxlayout.html>. Accessed: 2022-05-03.
- The-Qt-Company (2018l). Qwidget class. <https://doc.qt.io/qt-5/qwidget.html#details>. Accessed: 2022-05-03.
- The-Qt-Company (2018m). Signals and slots. <https://doc.qt.io/qt-5/signalsandslots.html>. Accessed: 2022-05-03.

- Vitalis, S., Arroyo Ohori, K., and Stoter, J. (2020). Cityjson in qgis: Development of an open-source plugin. *Transactions in GIS*, 24(5):1147–1164.
- Vitalis, S. and Labetski, A. (2020). Cityjson-qgis-plugin. <https://github.com/cityjson/cityjson-qgis-plugin>. Accessed: 2022-05-03.
- Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1(1):7–87.
- World-Wide-Web-Consortium et al. (2010). Xml path language (xpath) 2.0. <https://www.w3.org/TR/xpath20/>. Accessed: 2022-05-03.
- World-Wide-Web-Consortium et al. (2020). G64: Providing a table of contents. <https://www.w3.org/TR/WCAG20-TECHS/G64.html>. Accessed: 2022-05-03.
- Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubaauer, 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):1–26.
- Zipf, A. (2005). Using styled layer descriptor (sld) for the dynamic generation of user- and context-adaptive mobile maps – a technical framework. In Li, K.-J. and Vangenot, C., editors, *Web and Wireless Geographical Information Systems*, pages 183–193, Berlin, Heidelberg. Springer Berlin Heidelberg.

Colophon

This document was typeset using \LaTeX , using the KOMA-Script class scrbook. The main font is Palatino.

