Introducing server-side support for 3DCityDB 5.0 to the 3DCityDB-Tools plug-in for QGIS

Bing-Shiuan Tsai[1], Giorgio Agugiaro[1], Camilo Leon-Sanchez [1], Claus Nagel[2], Zhihang Yao[3]

[1] 3D Geoinformation group, Department Urbanism, Faculty of Architecture and Built Environment, Delft University of Technology, Delft, The Netherlands – bstsai1022@gmail.com, {g.agugiaro, c.a.leonsanchez}@tudelft.nl
[2] Virtual City Systems, Tauentzienstr. 7 B/C, 10789, Berlin, Germany – cnagel@vc.systems
[3] Centre for Geodesy and Geoinformatics, Stuttgart University of Applied Sciences (HFT Stuttgart), Stuttgart, Germany – zhihang.yao@hft-stuttgart.de

Keywords: 3D City Database, QGIS plug-in, CityGML 3.0, PostgreSQL

Abstract
The 3DCityDB-Tools plug-in for QGIS enables users to connect to the open-source 3D City Database (3DCityDB) 4.x, load CityGML 1.0 and 2.0 data, and structure it as GIS layers within QGIS. The plug-in simplifies interaction with the complex structure of the 3DCityDB 4.x by providing a GUI-based tool and a server-side package for seamless data retrieval and management from QGIS. With the release of the CityGML 3.0 conceptual data model in 2021, the 3D City Database has been updated to version 5.0, introducing several changes to support the new characteristics of CityGML 3.0 and a significant redesign and restructuring of the database schema. However, the current 3DCityDB-Tools plug-in for QGIS does not support the latest CityGML and 3DCityDB versions. This paper presents the findings and experiences gathered to modify the plug-in's server-side architecture to cope with the new 3DCityDB 5.0. Similar to what already happens with the current plug-in version, the proposed new approach enables the generation of GIS layers following the Simple-Feature-for-SQL model, optimising query performance and improving attribute management. The resulting vector-based layers can be seamlessly imported into QGIS, allowing for interaction between QGIS and the underlying CityGML data stored in the latest version of the 3DCityDB.

1. Introduction

Semantic 3D city models are essential for visualising, analysing, and managing the built environment (Biljecki *et al.*, 2015). The Open Geospatial Consortium (OGC) has adopted CityGML as an international standard for representing 3D spatial information. CityGML data is typically encoded in XML, JSON, or SQL, such as, for the last, the 3D City Database (3DCityDB) (Yao *et al.*, 2018). CityGML 2.0 was released in 2012 and remains widely used, while CityGML 3.0, released in 2021, is gaining gradual adoption. The latter represents a major overhaul of the data model, including improvements such as a revised Level of Detail concept, an updated spatial model, and support for temporal data, versioning, and point clouds. Given the generally large size of 3D city models, a database encoding offers a scalable and structured approach to spatial data management instead of a file-based approach. The 3DCityDB is an open-source project designed for PostgreSQL, Oracle, and PolarDB/Ganos relational databases and the de facto standard solution for the SQL encoding of CityGML. It implements the CityGML standard, supporting detailed semantics and multi-level representations of city objects. The 3DCityDB Suite[1] includes a set of tools which enable CityGML and CityJSON data exchange between the database and the file encodings. 3DCityDB 4.x supports CityGML 1.0 and 2.0, whereas the latest version 5.0 (released in 2025) also supports CityGML 3.0.

The 3DCityDB 4.x consists of 66 tables storing the feature data and managing relationships between them. Attributes and geometries of the same city object (e.g., buildings, bridges, roads, etc.) are often distributed across multiple linked tables. For example, a LoD2 building can be represented as a solid, a multi-surface geometry or thematic surfaces (e.g., WallSurfaces, RoofSurfaces, GroundSurfaces). While the complexity of 3DCityDB 4.x reflects the rich structure of CityGML data, it also represents a challenge to access the stored data for those GIS practitioners who may lack advanced SQL skills. To hide the complexity of the 3DCityDB 4.x, the "3DCityDB-Tools" plug-in for QGIS has been developed to simplify the database interactions by hiding the schema complexity and providing a user-friendly, GUI-based interface within QGIS. Users can interact with CityGML data through "standard" GIS layers (Agugiaro *et al.*, 2024). The newly released 3DCityDB 5.0 (Yao *et al.*, 2025) significantly simplifies the database schema, reducing the number of tables from 66 to 17, facilitating easier data access and adding support for CityGML 3.0.

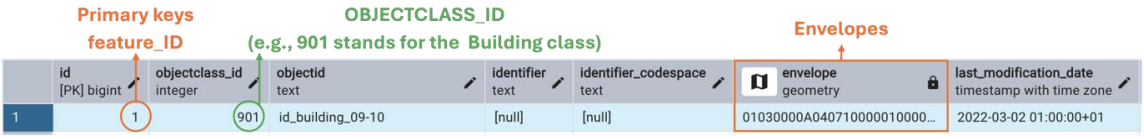
After a brief summary of the main changes between 3DCityDB 4.x and 5.0 and how such changes affect the plug-in's layer generation mechanism, the paper examines the integration of the updated schema into the existing architecture of plug-in, focusing on server-side adaptations required to generate GIS layers from spatial and non-spatial data. In particular, the current (for 3DCityDB 4.x) and the newly proposed (for 3DCityDB 5.0) approach for layer generation are presented, providing insight into the data extraction methods based on feature geometry and attribute types. The paper then presents the implementation results using test datasets and displaying the resulting GIS layers in QGIS. Finally, the current limitations are discussed, and some potential future improvements are outlined.

2. The 3DCityDB 5.0 in a nutshell

The 3DCityDB 5.0 maps the CityGML classes and their properties to 17 tables. Although providing a detailed overview of the complete database schema is beyond the scope of this paper, this section focuses on the tables crucial for this work. More details can be found in the online manual[2]. The starting point is the FEATURE table, which is the entry point of all

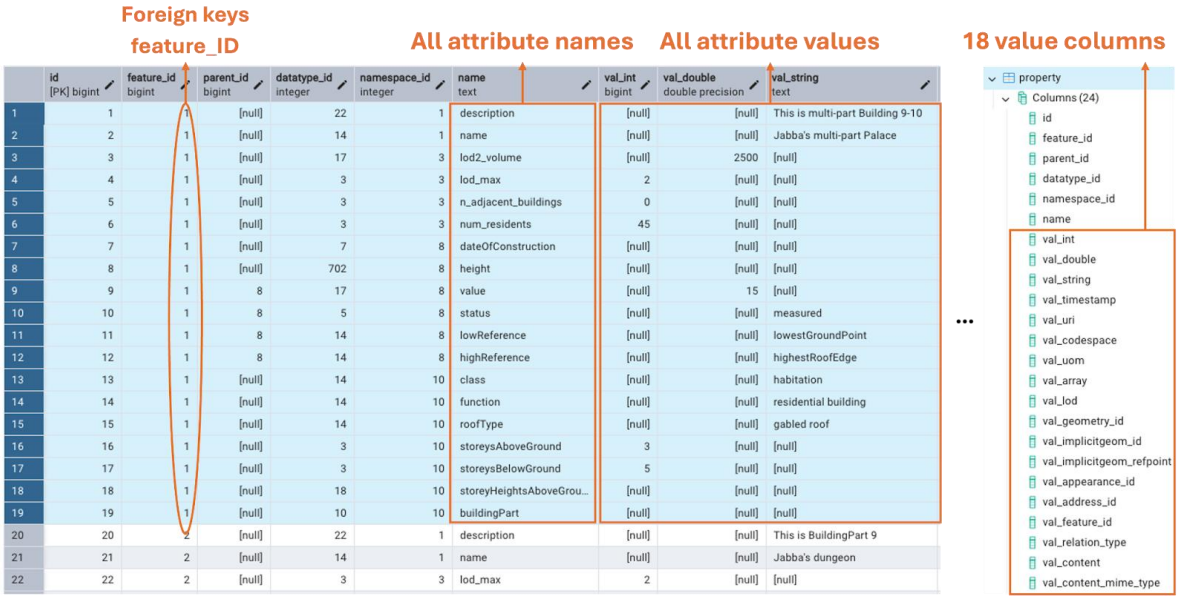
[1] https://github.com/3dcitydb/3dcitydb-suite

[2] https://3dcitydb.github.io/3dcitydb-mkdocs



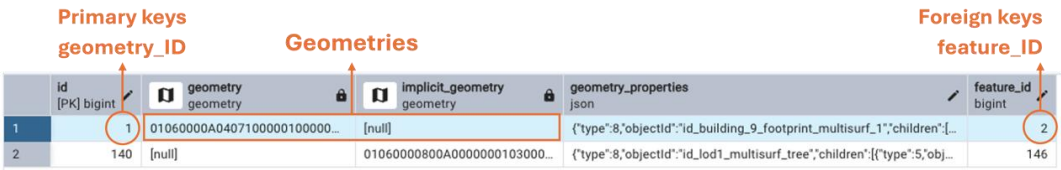
	id [PK] bigint	objectclass_id integer	objectid text	identifier text	identifier_codespace text	envelope geometry	last_modification_date timestamp with time zone
1	1	901	id_building_09-10	[null]	[null]	01030000A040710000010000…	2022-03-02 01:00:00+01

Figure 1. Example of the FEATURE table in 3DCityDB 5.0 (excerpt).



	id [PK] bigint	feature_id bigint	parent_id bigint	datatype_id integer	namespace_id integer	name text	val_int bigint	val_double double precision	val_string text
1	1	1	[null]	22	1	description	[null]	[null]	This is multi-part Building 9-10
2	2	1	[null]	14	1	name	[null]	[null]	Jabba's multi-part Palace
3	3	1	[null]	17	3	lod2_volume	[null]	2500	[null]
4	4	1	[null]	3	3	lod_max	2	[null]	[null]
5	5	1	[null]	3	3	n_adjacent_buildings	0	[null]	[null]
6	6	1	[null]	3	3	num_residents	45	[null]	[null]
7	7	1	[null]	7	8	dateOfConstruction	[null]	[null]	[null]
8	8	1	[null]	702	8	height	[null]	[null]	[null]
9	9	1	8	17	8	value	[null]	15	[null]
10	10	1	8	5	8	status	[null]	[null]	measured
11	11	1	8	14	8	lowReference	[null]	[null]	lowestGroundPoint
12	12	1	8	14	8	highReference	[null]	[null]	highestRoofEdge
13	13	1	[null]	14	10	class	[null]	[null]	habitation
14	14	1	[null]	14	10	function	[null]	[null]	residential building
15	15	1	[null]	14	10	roofType	[null]	[null]	gabled roof
16	16	1	[null]	3	10	storeysAboveGround	3	[null]	[null]
17	17	1	[null]	3	10	storeysBelowGround	5	[null]	[null]
18	18	1	[null]	18	10	storeyHeightsAboveGrou…	[null]	[null]	[null]
19	19	1	[null]	10	10	buildingPart	[null]	[null]	[null]
20	20	2	[null]	22	1	description	[null]	[null]	This is BuildingPart 9
21	21	2	[null]	14	1	name	[null]	[null]	Jabba's dungeon
22	22	2	[null]	3	3	lod_max	2	[null]	[null]

Figure 2. Example of the PROPERTY table in 3DCityDB 5.0 (excerpt).



	id [PK] bigint	geometry geometry	implicit_geometry geometry	geometry_properties json	feature_id bigint
1	1	01060000A0407100000100000…	[null]	{"type":8,"objectId":"id_building_9_footprint_multisurf_1","children":[…	2
2	140	[null]	01060000800A0000000103000…	{"type":8,"objectId":"id_lod1_multisurf_tree","children":[{"type":5,"obj…	146

Figure 3. Example of the GEOMETRY_DATA table in 3DCityDB 5.0 (excerpt).

features within the dataset. The PROPERTY table stores all feature attributes and the relations between them. The feature geometries, including the template geometries of implicit representations, are stored in the GEOMETRY_DATA table, while table IMPLICIT_GEOMETRY contains the geometry roots to reference the template implicit geometries. The following points detail these four tables:

- The FEATURE table registers all existing features stored in the 3DCityDB 5.0. Column *id* contains the primary key. Like the 3DCityDB 4.x, column *objectclass*_id adds semantic information about the different classes. The column *objectid* is used to store the feature *gmlid*. Column *envelope* stores the 3D bounding box of each feature (Figure 1).
- The PROPERTY table accommodates all feature attributes and relations following a schema-less model (Figure 2). The main columns are:
  - *id* is the primary key;
  - *feature_id* is a foreign key that links the attribute to the corresponding feature;
  - *parent_id* stores the relation of nested attributes, i.e., complex attributes that are further split into simpler parts and stored across multiple rows. "Parent" attributes appear first, followed by their "children" attributes linked via *parent_id* keys;
  - *datatype_id* is a foreign key to the DATATYPE table, which contains metadata for the data types in CityGML. Metadata is available for simple types such as integers, strings, etc., and complex types such as geometries or implicit geometries;
  - *namespace_id* specifies the namespace of the respective attribute. It can be linked to table NAMESPACE;
  - *name* stands for the property name. This can be a simple attribute name (e.g., "class", "name", or "year of construction") but also a spatial property such as "lod1Solid", specifying, for example, that the feature is represented as a solid geometry in LoD1 and providing the link to the table containing its geometry;
  - *val_** stands for a set of columns storing the attribute values. Based on the attribute type, values are stored across different columns, starting from *val_int* to *val_content_mine*. There are 18 different *val_** columns.
- The GEOMETRY_DATA table stores the geometries of existing features. It contains the primary key *id* as geometry identifiers to be joined with *val_geometry_id* or *val_implicitgeom_id* from the PROPERTY table. The *feature_id* is a foreign key directly linked with the features (Figure 3). Unlike the 3DCityDB 4.x, type solid or multi-surface geometries are stored as polyhedral surface or multi-polygon objects, respectively, instead of being decomposed into the polygons composing them (details in section 5.2.2).
- The IMPLICIT_GEOMETRY table is only referenced when features have an implicit spatial representation. Similarly to 3DCityDB 4.x, it stores the primary key of the implicit geometry attributes, and the *relative_geometry_id* key is used to join with the *id* keys from the GEOMETRY DATA table, retrieving geometries for the feature implicit representations.

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W6-2025-193-2025 | © Author(s) 2025. CC BY 4.0 License.

```
SELECT f.id AS f_id, g.geometry::geometry(MultiPolygonZ,28992) AS geom
FROM citydb.feature AS f
    INNER JOIN citydb.property AS p ON (f.id = p.feature_id
       AND f.objectclass_id = 901)                                    -- choose buildings
       AND p.name = 'boundary'                                        -- choose boundary relation properties
    INNER JOIN citydb.feature AS f1 ON f1.id = p.val_feature_id
       AND f1.objectclass_id = 712                                    -- choose roof surfaces
    INNER JOIN citydb.property AS p1 ON f1.id = p1.feature_id          -- choose LoD & geometry representation
       AND p1.name = 'lod2MultiSurface'
    INNER JOIN citydb.geometry_data AS g ON g.id = p1.val_geometry_id; -- link to GEOMETRY_DATA table
```

Listing 1. Example of a query to extract all roofs of buildings in LoD2 Multi-Surface from the 3DCityDB 5.0.

- **3DCityDB 4.x (CITYOBJECT +BUILDING tables)**

	id bigint	objectclass_id integer	gmlid character varying (256)	measured_height double precision	measured_height_unit character varying (4000)	function character varying (1000)	function_codespace character varying (4000)	lod1_solid_id bigint
1	2	26	id_building_01	49.21	ft	residential building--/\--youth hostel	http://www.sig3d.org/codelists/standard/building/2.0/...	24

- **3DCityDB 5.0 (FEATURE + PROPERTY tables)**

	(feature)id bigint	objectclass_id integer	objectclass_id integer	(attribute)id bigint	(attribute)parent_id bigint	name text	val_double double precision	val_uom text	val_string text	val_codespace text	val_geometry_id bigint
1	38	901	901	206	[null]	height	[null]	[null]	[null]	[null]	[null]
2	38	901	901	207	206	value	49.21	ft	[null]	[null]	[null]
3	38	901	901	208	206	status	[null]	[null]	measured	[null]	[null]
4	38	901	901	209	206	lowReference	[null]	[null]	lowestGroundPoint	[null]	[null]
5	38	901	901	210	206	highReference	[null]	[null]	highestRoofEdge	[null]	[null]
6	38	901	901	212	[null]	function	[null]	[null]	residential building	http://www.sig3d.org/codelists/standard/building/2.0/...	[null]
7	38	901	901	213	[null]	function	[null]	[null]	youth hostel	http://www.sig3d.org/codelists/standard/building/2.0/...	[null]
8	38	901	901	224	[null]	lod1Solid	[null]	[null]	[null]	[null]	42

Figure 4. Example of the differences between 3DCityDB 4.x [top] and 5.0 [bottom] schemas to store building data.

Listing 1 shows an example of a SQL query to retrieve all building roofs represented as LoD2 multi-surface from the 3DCityDB 5.0 schema "citydb". It involves repetitively referencing the FEATURE table to access the feature entries, the PROPERTY table for querying the feature attributes and relations between them and providing geometry roots to link the corresponding geometries from the GEOMETRY_DATA table.

3. Major differences between 3DCityDB 4.x and 5.0

In 3DCityDB 4.x, standard attributes and the LoD geometry roots are mapped to joined tables, starting from the CITYOBJECT table down to a specific thematic table, such as the BUILDING table. The feature geometries such as solids and multi-surfaces are decomposed into polygons and stored in the SURFACE_GEOMETRY table, together with all necessary hierarchical information to re-aggregate them upon export. Only the root ID of the composite geometry is referenced via foreign keys from the thematic tables. Regarding CityGML *generic attributes*, they are mapped to a single table named CITYOBJECT_GENERICATTRIB.

In contrast, the starting table in 3DCityDB 5.0 is the FEATURE table. All attributes and spatial properties are mapped to the PROPERTY table following a type-enforced EAV (Entity-Attribute-Value) model. Figure 4 illustrates schema differences between 3DCityDB versions for a building (*objectid* "id_building_01") with two functions (residential and youth hostel), one height value (49.21 ft) and a *lod1Solid* representation. In 3DCityDB 4.x, attributes are accessed via the CITYOBJECT and BUILDING tables. In 3DCityDB 5.0, attributes are retrieved from the PROPERTY table: the function values and the geometry property are attributes stored in separate rows, while the height is a complex attribute spread across multiple interconnected rows via *parent_id*. Finally, the feature geometries are not decomposed but directly stored in the GEOMETRY_DATA table.

4. Layer generation in the 3DCityDB-Tools plug-in

The server side of the 3DCityDB-Tool plug-in, also called the "QGIS Package" (for 3DCityDB 4.x), is written in PL/pgSQL. The main functions offered by the "QGIS Package" are targeted at layer creation and the management of users and their database privileges. The "QGIS Package" allows users to define and create a layer by extracting a specific geometry according to its LoD and to associate it with all corresponding attributes. The resulting layer follows the Simple Feature for SQL (SFS) model. The reason is that QGIS supports only SFS vector layers. It is therefore necessary to choose beforehand one of the multiple representations that a CityGML feature can have.

Each layer consists of a view that links all underlying tables containing the feature attributes and a materialized view containing the feature geometries of the selected LoD. For example, in case of the buildings, the standard feature attributes of the "Building" class are stored in tables CITYOBJECT and BUILDING. These two tables are therefore linked. The *id* keys in the BUILDING table are then joined with the respective materialized view of geometries. Unlike CityGML "standard" feature attributes (e.g. class, function, usage, etc.), CityGML *generic attributes* are not attached to the layers. Instead, they are linked as a child table in the QGIS GUI, so a user can explore (and edit them) via a nested table. Upon layer creation, specific trigger functions are deployed to update each view (as far as the attributes are concerned).

The use of materialized views for the geometries has been chosen as a compromise to provide a better user experience at the cost of (temporarily) consuming storage space and the time needed to generate them upon layer creation. Otherwise, aggregating all geometries from the component polygons every time or performing a 3D affine transformation on implicit representation geometries would negatively affect the user experience. Additionally, a check counts the number of existing features. If there are no data regarding a specific class (e.g. "Room") in the database, these layers will not be generated. Finally, users can specify the extents of the layers to be generated. This is particularly useful for huge city models, allowing users to create materialized views for only a selected, smaller area. This approach reduces the (temporary) storage space required by the materialized views and significantly

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W6-2025-193-2025 | © Author(s) 2025. CC BY 4.0 License.

decreases the time needed to refresh them. More details are provided in Agugiaro *et al.* (2024).

5. Methodology

This section describes the methodology defined and implemented for the server-side part of the QGIS plug-in for the 3DCityDB 5.0. This results from the changes in CityGML 3.0, the 3DCityDB 5.0 schema differences, and the server-side implementation of the 3DCityDB-Tools plug-in. The methodology consists of three parts:

1) Database preparation: This paper takes this step for granted, i.e., that the 3DCityDB 5.0 has already been installed. Successively, the "QGIS Package for 3DCityDB 5.0" can be installed on top of the 3DCityDB instance.
2) QGIS Package: The new "QGIS package for 3DCityDB 5.0" installs upon the 3DCityDB 5.0 instance a set of tables and functions that follow a similar logic as in the previous version, however with slightly different steps, such as:
   a) Users can specify the geographical extents of the to-be-generated layers;
   b) An improved scan is performed to check for the existence of the feature geometries stored in the database, compatible with how geometries and LoDs are dealt with in CityGML 3.0 and 3DCityDB 5.0;
   c) A new scan has been added to check the existing feature attributes. Attributes are classified into four different classes (details will be given later on)
   d) Information collected about geometries and attributes is stored in the corresponding metadata tables. Such metadata is later needed to convert the 3DCityDB 5.0 data into layers supported by QGIS.
3) Data interaction: After collecting information on existing feature geometries and attributes, users can select the desired geometry representation of a feature class and choose which attribute to add to the respective layer for QGIS. Unlike the previous "QGIS package" for 3DCityDB 4.x, editing attribute data via specific forms in QGIS has not been implemented yet, and is left for future improvements. The following sections provide more details about each part of the methodology.

5.1 Part 1: Database preparation

Once 3DCityDB 5.0 is successfully set up, users can download the "QGIS Package for 3DCityDB 5.0" from the GitHub[3]. Detailed instructions for the setup are provided in Tsai (2024). This process creates a new schema named "qgis_pkg" in the 3DCityDB 5.0 database instance. In short, the QGIS Package provides functions to generate SQL queries dynamically to create GIS layers from data stored in 3DCityDB 5.0. Users must first create a custom schema to store the generated layer views using the script in Listing 2. The newly created schema follows the naming convention "qgis_{user_name}", referred to as "usr_schema" throughout this paper. Within "usr_schema", four auxiliary tables are generated: (1) EXTENTS (2) FEATURE_GEOMETRY_METADATA (3) FEATURE_ATTRIBUTE_METADATA (4) LAYER_METADATA. These tables correspond to the four main steps required for GIS layer creation.

```
-- create user schema for QGIS Package
SELECT qgis_pkg.create_qgis_usr_schema('user_name')
```

Listing 2. Query example to create a user schema.

[3] https://github.com/tudelft3d/3DCityDB-Tools-for-QGIS/tree/thesis_bingshiuan

5.2 Part 2: The QGIS package for 3DCityDB 5.0

5.2.1 Extents selection: Once the CityGML/CityJSON data are imported into the 3DCityDB schema, the bounding box extents for GIS layer generation can be set by the user, similar to the current QGIS plug-in implementation. The example in Listing 3 shows how to use the SQL function to set default or, alternatively, user-specified extents using the PostGIS *ST_MakeEnvelope* function. The extents specified are stored and used in all successive steps. The target data schema is called "cdb_schema", i.e. the schema that contains the 17 tables.

```
-- full-schema extents (default)
SELECT qgis_pkg.upsert_extents('usr_schema', 'cdb_schema');
-- user-defined extents
SELECT qgis_pkg.upsert_extents('usr_schema', 'cdb_schema',
   'm_view', ST_MakeEnvelope(Xmin, Ymin, Xmax, Ymax, SRID));
```

Listing 3. Query example to define the extents of the layers to be generated.

5.2.2 Feature geometries: Once the geographical extents are set, a schema-wise scan is performed to check the existence of feature geometries within the chosen "cdb_schema". This operation is carried out by function *update_feature_geometry_metadata*, which takes *objectclass_id* as input. For instance, the *objectclass_id* 901 stands for buildings, while 709 represents WallSurfaces. The results of this scan are stored in the FEATURE_GEOMETRY_METADATA table. The function shown in Listing 4 must be executed whenever users specify a new extent. Figure 5 shows an example of the results stored in the FEATURE_GEOMETRY_METADATA table.

```
-- full-schema extents (default)
SELECT qgis_pkg.update_feature_geometry_metadata(
'usr_schema', 'cdb_schema');
-- user-defined extents
SELECT qgis_pkg.update_feature_geometry_metadata(
'usr_schema', 'cdb_schema', 'm_view');
```

Listing 4. Example of query to check the existence of feature geometries.

Once the FEATURE_GEOMETRY_METADATA table is updated, users can choose which feature LoD geometry representation they want. The QGIS Package functions will then dynamically create and run SQL scripts to generate views (or materialized views) of the geometries. This process involves joining the FEATURE, the PROPERTY, and the GEOMETRY_DATA tables. In the case of implicit geometries, the table IMPLICIT_GEOMETRY is also used.

Although 3DCityDB 5.0 directly stores surface geometries without disaggregating them, which in theory should reduce query times compared to previous 3DCityDB version, the PROPERTY table could be huge for large datasets, affecting the query time efficiency. Tests comparing views versus materialized views show that, while views are sufficient for space features (e.g., buildings), materialized views still offer better time performance for boundary features (e.g., building roofs) and space features with implicit representation (e.g., trees), where queries involve intensive cross-referencing between FEATURE and PROPERTY tables (Tsai, 2024). Despite materialized views (temporarily) consuming storage space and taking longer to create and refresh, they are used by default in the QGIS Package for 3DCityDB 5.0 to enhance the user experience.

	id [PK] bigint	cdb_schema character varying	bbox_type character varying	parent_objectclass_id integer	parent_classname character varying	objectclass_id integer	classname character varying	datatype_id integer	geometry_name text
1	1	citydb	db_schema	0	-	502	TINRelief	11	tin
2	2	citydb	db_schema	0	-	902	BuildingPart	11	lod0MultiSurface
3	3	citydb	db_schema	0	-	902	BuildingPart	11	lod1Solid
4	4	citydb	db_schema	901	Building	15	ClosureSurface	11	lod2MultiSurface
5	5	citydb	db_schema	902	BuildingPart	15	ClosureSurface	11	lod2MultiSurface

Figure 5. Example of FEATURE_GEOMETRY_METADATA table (excerpt).

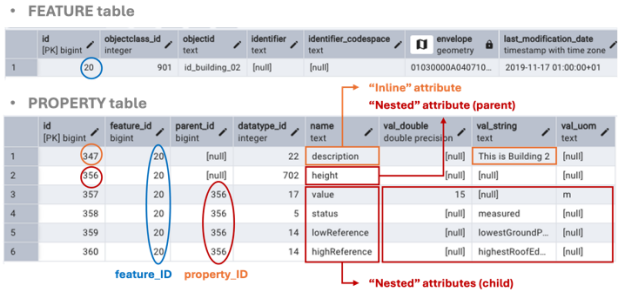


Figure 6. Examples of "inline" and "nested" attribute.

5.2.3 Feature attributes: In 3DCityDB 5.0, CityGML standard and *generic attributes* are all stored in the PROPERTY table. To comply with the SFS model when generating GIS layers, they must be joined with the (materialized) views containing the geometries. This operation is generally done by pivoting the query results on the attributes and joining them to the respective geometry (materialized) views using attribute names as column headers. Particular care is required in case of nested attributes. The "QGIS Package for 3DCityDB 5.0" first classifies all attributes in the PROPERTY table into two main types:

- "Inline" attributes are single-row records directly associated with a feature via the *feature_id*. Their data types are indicated by *datatype_id*, and the corresponding values are stored in specific columns based on these types. An example is provided in Figure 6 for the attribute "description".
- "Nested" attributes are stored across multiple rows connected through the *parent_id*. The first row represents the parent attribute name and multiple "inline" child attributes. An example is provided in Figure 6 for the attribute "height".

The first step of the attribute process is checking the existence of feature attributes within the selected "cdb_schema". This operation is performed by function *update_feature_attribute_metadata*, which iterates through the *objectclass_id* of the existing classes scanning their existing attributes and determining their type (Listing 5). The results of the attributes scan are stored in the FEATURE_ATTRIBUTE_METADATA table, providing a summary of the existing attributes and their properties. This function should be executed whenever users specify a new extent. Figure 7 shows an example of the result.

The multiplicity and attribute value columns are two additional factors to be checked before performing (if required) the pivot operation on the attribute query with the PostgreSQL *crosstab* function. When an attribute is selected, these two factors are checked and updated in the attribute metadata table. Multiplicity is the minimal and maximal number of attribute occurrences per object, determined using the *max* and *count* functions on the target attribute by referencing the FEATURE and PROPERTY tables. Once the attribute data types and multiplicity are determined, all feature attributes can be categorised as "inline-single," "inline-multiple," "nested-single," and "nested-multiple." These four attribute categories require distinct flattening strategies based on their storage structure. In particular, the *crosstab* function in PostgreSQL is used for flattening feature nested attributes and attributes with multiplicity greater than 1. Listing 6 provides an example query for pivoting buildings' "function" attribute. This attribute is classified as an "inline-multiple" attribute type with two associated value columns: *val_string* and *val_codespace*. In this case, the crosstab function is required to pivot the query result

```
-- full-schema extents (default)
SELECT qgis_pkg.update_feature_attribute_metadata(
'usr_schema', 'cdb_schema');
-- user-defined extents
SELECT qgis_pkg.update_feature_attribute_metadata(
'usr_schema', 'cdb_schema', 'm_view');
```

Listing 5. Example of query to check the existence of feature attributes.

```
-- Define a composite type to hold values in tuples
DROP TYPE IF EXISTS "citydb_901_function";
CREATE TYPE "citydb_901_function"  AS
  (val_string text, val_codespace text);
-- Flatten attributes using composite type
SELECT f_id AS f_id,
  -- Extract values from composite-type tuples
  (function_1).val_string    AS "function_1",
  (function_1).val_codespace AS "function_codespace_1",
  (function_2).val_string    AS "function_2",
  (function_2).val_codespace AS "function_codespace_2"
FROM CROSSTAB($BODY$
  SELECT -- 1: row id, 2: category,
    f.id AS f_id, p.name,
    -- 3: values
    (p.val_string, p.val_codespace)::"citydb_901_function"
  FROM citydb.feature AS f
    INNER JOIN citydb.property AS p ON
      (f.id = p.feature_id AND f.objectclass_id = 901)
  WHERE p.name = 'function'
  ORDER BY f_id, p.id ASC $BODY$)
  -- Max multiplicity defines number of value columns
  -- Source value columns cast to composite type
  AS ct(f_id bigint,
    function_1 "citydb_901_function",
    function_2 "citydb_901_function");
```

Listing 6. Query example to collect and flatten the "function" of buildings from 3DCityDB 5.0.

	id [PK] bigint	cdb_schema character varying	bbox_type character varying	objectclass_id integer	classname character varying	parent_attribute_name character varying	parent_attribute_typename character varying	attribute_name character varying	attribute_typename character varying	is_nested boolean
1	40	citydb	db_schema	901	Building	-	-	description	StringOrRef	false
2	46	citydb	db_schema	901	Building	-	-	name	Code	false
3	57	citydb	db_schema	901	Building	height	Height	highReference	Code	true
4	58	citydb	db_schema	901	Building	height	Height	lowReference	Code	true
5	59	citydb	db_schema	901	Building	height	Height	status	String	true
6	60	citydb	db_schema	901	Building	height	Height	value	Measure	true

Figure 7. Example of FEATURE_ATTRIBUTE_METADATA table (excerpt).

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W6-2025-193-2025 | © Author(s) 2025. CC BY 4.0 License.

• **PROPERTY table, before**

	f_id bigint	name text	val_string text	val_codespace text
1	1	function	residential building	http://www.sig3d.org/codelists/standard/building/2.0/_...
2	20	function	youth hostel	http://www.sig3d.org/codelists/standard/building/2.0/_...
3	20	function	residential building	http://www.sig3d.org/codelists/standard/building/2.0/_...
4	20	function	hostel	http://www.sig3d.org/codelists/standard/building/2.0/_...
5	30	function	residential building	http://www.sig3d.org/codelists/standard/building/2.0/_...

• **..After flattening**

	f_id bigint	function_1 text	function_codespace_1 text	function_2 text	function_codespace_2 text	function_3 text	function_codespace_3 text
1	1	residential building	http://www.sig3d.org/c...	[null]	[null]	[null]	[null]
2	20	residential building	http://www.sig3d.org/c...	hostel	http://www.sig3d.org/c...	youth hostel	http://www.sig3d.org/co...
3	30	residential building	http://www.sig3d.org/c...	[null]	[null]	[null]	[null]

Figure 8. Example of flattening the inline-multiple attribute "function" of class Building.

The value columns are collected and cast into a composite type with its definition header added dynamically at the beginning, as the *crosstab* function only accepts one value column from the source table. The *FROM* clause includes the attribute retrieval source query enclosed within *$BODY$* tags passed into the *crosstab* function, reflecting the multiplicity (in this case, two). The resulting table columns are named according to the target attribute ("function"), prefixed with their multiplicity numbers and explicitly cast as composite types. After pivoting the query results, the individual values are extracted from their composite-type tuples. In the final *SELECT* clause, the first extracted value column adopts the original attribute name, while the remaining value columns receive suffixes derived from the target attribute name. The flattened result of the "function" of buildings can be seen in Figure 8. The QGIS Package functions apply this pivoting and column renaming approach to generate SQL queries for attribute flattening dynamically, using attribute category information and value column definitions based on the four established attribute categories.

The approach to handling feature attributes differs significantly from the one used in the "QGIS Package" for 3DCityDB 4.x, in which all CityGML standard attributes are already stored in tables, while *generic attributes* are managed through sub-tables. In the current approach, all attributes (both standard and *generic*) require flattening due to the PROPERTY table structure. However, this method gives users greater flexibility in selecting which attributes to join with geometry views.

5.2.4 Creating GIS layers: Once the metadata tables for geometries and attributes are updated, users must choose which type of geometrical representation to use and which attributes to associate with that representation. At database level, this means linking the feature geometries in the (materialized) views with the selected attributes to generate the resulting SFS-compliant layer. Given the potentially large size of the PROPERTY table, the complexity of attribute-flattening queries and the number of SQL joins involved, several tests have been conducted to explore the best way to join geometries and attributes.

Figure 9 provides an overview of the evaluated approaches. Materialised views are generally preferable as they offer faster query time performance, especially when dealing with large-scale datasets. For this reason, the chosen approach is the last one at the bottom of Figure 9, where a single SQL statement is automatically generated and executed to have all selected attributes gathered in a unique materialized view. The resulting materialized view is then joined once with the (materialized) view containing the feature geometries. This approach allows for avoiding multiple joins and therefore reduces the computation overhead when interacting with the data. More details can be found in (Tsai, 2024).

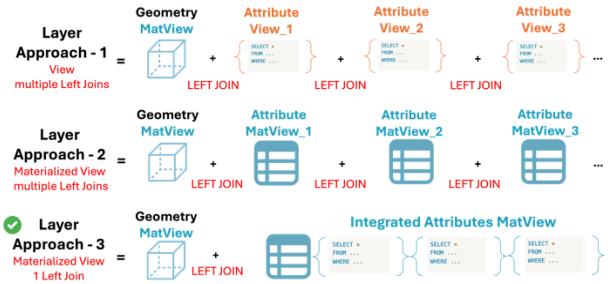


Figure 9. Different GIS layer creation approaches tested with the 3DCityDB 5.0.

In Listing 7, a user can create a layer containing the building function and height with buildings represented in LoD1 solid geometry by specifying the *objectclass_id* 901 for the Building class, the "lod1Solid" and 1 for geometry type, LoD number and the selected attribute names in an array. The resulting SQL scripts are generated and run automatically by providing such parameters to the function *create_layer*. Creating all layers for a specific CityGML class or creating the layers with all its existing attributes is also possible using the *create_class_layers* and *create_all_layer* functions, as exemplified in Listing 8. Layers are created and stored within the selected "usr_schema", and they can then be visualised directly in QGIS.

```
SELECT qgis_pkg.create_layer('usr_schema', 'cdb_schema',
    -- (Parent) objectclass_id (0 for space features)
    0, 901,
   -- Geometry type, LoD
    'lod1Solid', 1,
    -- Selected attributes
    ARRAY['function', 'height'],
    -- True to save the result in a materialized view
    TRUE);
```

Listing 7. Query example to create a single GIS layer.

```
-- Create all layers for class Building
SELECT qgis_pkg.create_class_layers(
    'usr_schema', 'cdb_schema', 0, 901);
-- Create layers for class (Building) WallSurfaces
SELECT qgis_pkg.create_class_layers(
    'usr_schema', 'cdb_schema', 901, 709);
-- Create layers for all existing CityGML classes with all
existing attributes
SELECT qgis_pkg.create_all_layer(
    'usr_schema', 'cdb_schema');
```

Listing 8. Query examples to batch-create GIS layers.

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W6-2025-193-2025 | © Author(s) 2025. CC BY 4.0 License.

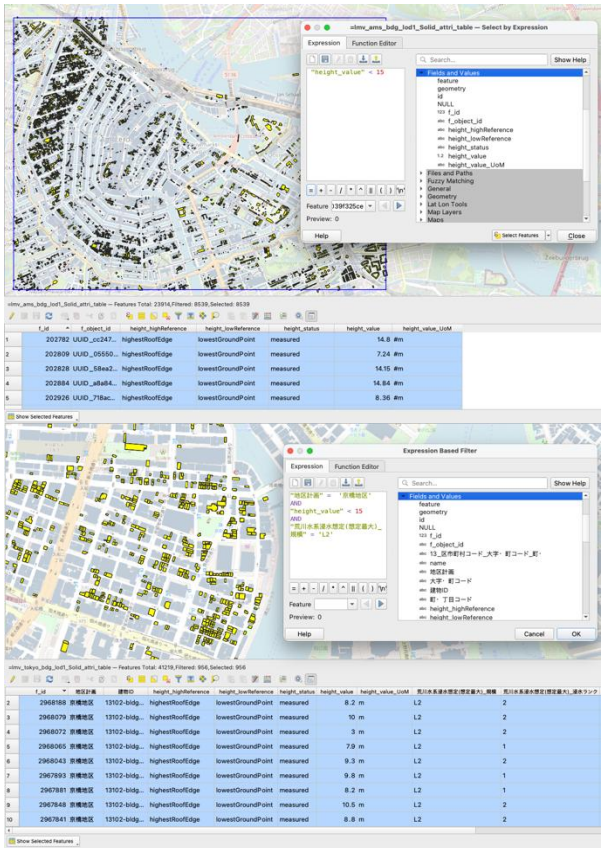


Figure 10. Generated layers loaded in QGIS and visualized in 2D: [top] Amsterdam, [bottom] Tokyo.

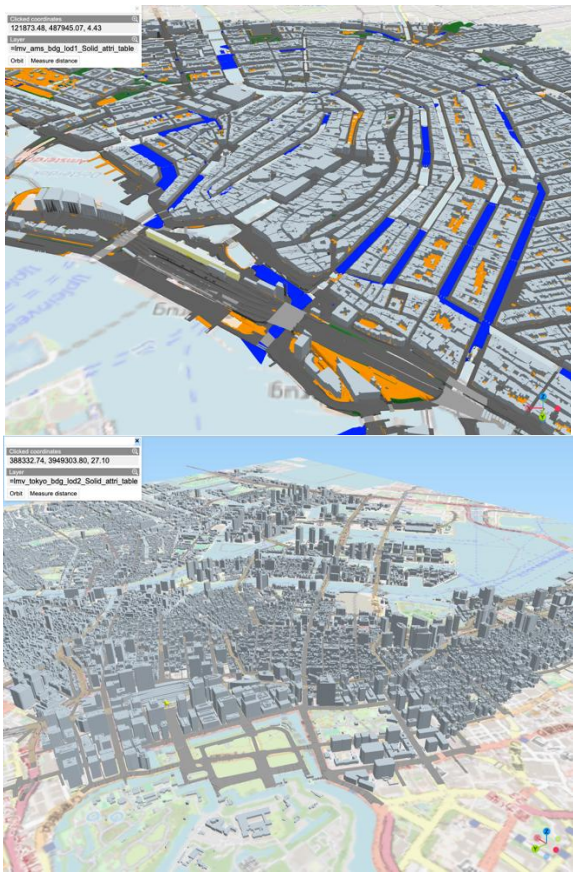


Figure 11. Generated layers loaded in QGIS and visualized in 3D: [top] Amsterdam, [bottom] Tokyo.

5.3 Part 3: Interacting with 3DCityDB 5.0 data from QGIS

The generated layers can be loaded into QGIS via drag-and-drop after establishing a PostGIS connection. As the feature attributes are flattened into column headers, their values can be directly queried, providing an intuitive interaction with the CityGML data stored in 3DCityDB 5.0. For instance, in Figure 10 (top), buildings in Amsterdam are selected based on their heights below 15 metres. A more advanced example is shown in Figure 10 (bottom), where buildings in Tokyo are selected based on their heights below 15 metres, their location within the Kyo Bashi District Plan (京橋地区計画), and categorisation under the flooding scale (規模) "L2" for the Arakawa River flood inundation risk area (荒川洪水浸水想定区域(想定最大規模)). These layers enable interaction with 3DCityDB data from QGIS, both in 2D (Figure 10) and in 3D (Figure 11). Using SFS-compliant layers has the additional advantage that "standard" QGIS tools and plug-ins can be used, therefore further enhancing the utility of QGIS for different applications.

Several input datasets, varying in terms of standard (CityGML 2.0 and 3.0), extents and geographical location (e.g. Amsterdam, portions of New York, Munich, and Tokyo), and feature classes (e.g. Building, Vegetation, Transportation, Relief, etc.) have been tested. Only few screenshots are shown here. More details are provided in Tsai (2024).

6. Current limitations

Some limitations remain in the current implementation of the "QGIS Package for 3DCityDB 5.0". For example, data editing via the GIS layers is not yet supported. The layers are generated by joining feature geometries and attributes into materialized views, which do not support automatic updates to the underlying 3DCityDB 5.0 tables. To overcome this limitation, two potential approaches have been considered so far:

(1) Intermediate Tables with Trigger Functions: This method involves creating an intermediate temporary table that duplicates the flattened data of generated GIS layers instead of resorting to materialized views. User modifications made in QGIS are stored in this intermediate table, and trigger functions propagate these changes back to the original tables within the "cdb_schema". Although trigger functions could be created and associated directly with the materialized views, the approach using tables seems preferable as it is similar in terms of (temporary) storage consumption for the duplicated data, but it does not require any joins between geometries and attributes.

(2) Incremental View Maintenance (IVM): Inspired by the IVM extension for PostgreSQL [4], Incrementally Maintainable Materialized Views (IMMVs) could be explored. IMMVs use database triggers to detect and apply only incremental changes rather than recomputing entire views, offering improved efficiency over typical materialized view refresh methods. As a matter of fact, this seems to be the most promising solution, however, at the time of writing (spring 2025), there are still limitations regarding definitions of supported views. For example, inner joins are supported, but outer joins are not. The latter, however, are extensively used in our approach.

Although it is already possible to edit the feature attributes from QGIS if the PROPERTY table is attached to the geometries as a sub-table via adding relations, this only enables attribute editing on a single-feature basis.

[4] https://github.com/sraoss/pg_ivm

Another limitation is that features without direct geometry representations (e.g., traffic spaces in transportation data) cannot be visualized via GIS layers. Two potential approaches are being considered for future development:

(1) Feature bounding box envelopes: Using the feature bounding boxes stored in the FEATURE table provides a fast and straightforward approach to create layers without joining the GEOMETRY_DATA table. However, bounding-box envelopes only offer coarse geometries, limiting spatial analyses.
(2) Aggregation of geometries of child classes: Aggregating child feature geometries offers another way to create layers by representing parent features through their child geometries (e.g., aggregating traffic areas for traffic spaces), which however requires additional processing via SQL queries.

A final limitation is due to the column name length. PostgreSQL has a limitation of 63 bytes for table column names. Names exceeding this length are automatically truncated, potentially causing ambiguity errors. For example, the attribute "Arakawa River flood inundation risk area" (荒川洪水浸水想定区域(想定最大規模))". In such a case, users must manually define abbreviations for these long names using the function in Listing 9, which stores the abbreviations in the FEATURE_ATTRIBUTE_METADATA table. The abbreviations are used when the name length exceeds the limit. In the previous example, its abbreviation is used as shown in Figure 10. Currently, attribute abbreviations must still be set manually, but automating this process remains a task for future development.

```
SELECT qgis_pkg.update_nested_attri_abbr(
    'usr_schema', 'cdb_schema', oc_id, -- objectclass_id
    'nested_parent_attribute', -- original attribute name
    'abbreviation');           -- user-defined abbreviation
```

Listing 9. Query example to manually assign an abbreviation for a nested parent attribute.

7. Conclusion and Outlook

This paper has presented the methodology and the implementation of a SQL-based server-side "QGIS package" that allows to interact with CityGML data stored in the recently released 3DCityDB 5.0: after scanning the contents of the whole database (or only within a specific user-defined bounding box) in terms of available geometrical representations and attributes, the users only need to choose the feature class(es) (e.g. Buildings), the geometrical representation (e.g. LoD1 solid), and the attributes (e.g. name, class, year of construction). A single SQL function then takes the users' input and automatically generates and executes the necessary scripts to create SFS-compliant layers, which can be them easily loaded into QGIS. The main challenge of this work has been converting CityGML attribute data that can use complex types into a simple, flattened format compatible with the SFS model used by QGIS for vector data. Attributes with multiplicity bigger than 1 (e.g., building function) or nested (e.g., building height) have required particular care.

The presented approach bridges the gap between complex data models and simplified GIS layers that users can intuitively view, query, and potentially edit, e.g. in QGIS. Web Feature Service (WFS) providers have struggled to fully support complex data structures in QGIS, limiting their functionality to simple GML features. Solutions such as the QGIS GML Application Schema Toolbox (GMLAS[5]) have been employed to manage complex features, but they introduce limitations in user experience, data navigation, and editing capabilities. Recent enhancements proposed by the QGIS-DE user group attempt to handle complex features by converting XML structures into JSON-formatted strings[6], however, this approach is still in development and has some limitations, such as single-geometry constraints per WFS layer and restricted editing and filtering capabilities.

In contrast, the approach presented in this paper provides a complementary, server-side solution by linearising complex features within the 3DCityDB 5.0 database. Flattening "nested" attributes into standard GIS layers simplifies the sometimes complex structure of CityGML data, offering direct and intuitive interaction within QGIS attribute tables. This significantly improves data accessibility and query efficiency, however, at the cost of some (temporary) storage space for the materialized views and the preprocessing time to generate them.

Finally, enabling editing capabilities for the layers could further enhance the user experience within QGIS. Changes would propagate to the 3DCityDB tables and could be then, for example, exported to CityGML files for broader data sharing.

References

Agugiaro, G., Pantelios, K., León-Sánchez, C., Yao, Z., Nagel, C., 2024. Introducing the 3DCityDB-Tools plug-in for QGIS. *Recent Advances in 3D Geoinformation Science - Proceedings of the 18th 3D GeoInfo Conference*, Springer, pp. 797–821. https://doi.org/10.1007/978-3-031-43699-4_48

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., Çöltekin, A., 2015. Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information, 4*(4), 2842-2889. https://doi.org/10.3390/ijgi4042842

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Labetski, A., Vitalis, S., 2019. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data Software and Standards*, 4(1), pp. 1–12. https://doi.org/10.1186/s40965-019-0064-0

Tsai, B.-S., 2024. 3DCityDB-Tools plug-in for QGIS: Adding server-side support to 3DCityDB v.5.0. MSc thesis. Delft University of Technology. https://resolver.tudelft.nl/uuid:5992ba24-8618-48d7-9e24-28839b5da16b

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., 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 Softw Stand*, 3(5):1–26. https://doi.org/10.1186/s40965-018-0046-7

Yao, Z., Nagel, C., Kendir, M., Willenborg, B., Kolbe, T.H., 2025. The new 3D City Database 5.0 – Advancing 3D city data management based on CityGML 3.0. *ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci.* Proc. of the 20th 3DGeoInfo and 9th Smart Data Smart Cities conference (same volume as this paper).

[5] https://brgm.github.io/gml_application_schema_toolbox
[6] https://github.com/qgis/QGIS-Enhancement-Proposals/issues/277