City2Twin: an open urban digital twin from data integration to visualization and analysis.

Benirina Parfait Rafamatanantsoa[1], Imane Jeddoub [2], Anass Yarroudh [2], Rafika Hajji [1], Roland Billen [2]

[1] College of Geomatic Sciences and Surveying Engineering, Hassan II Institute of Agronomy and Veterinary Medicine, Rabat 10101, Morocco (rafamatanantsoabenirinaparfait, r.hajji)@iav.ac.ma
[2] GeoScITY, Spheres Research Unit, University of Liège, 4000 Liège, Belgium (i.jeddoub, ayarroudh, rbillen)@uliege.be

Keywords: Urban Digital Twin, Data integration, 3D data visualization, 3D data analysis, CityJSON, IoT.

Abstract

Urban Digital Twins have gained significant interest in the urban and geospatial fields, enabling interactive visualization and advanced analysis of cities across various domains. However, current implementation approaches are heterogeneous in terms of data and approaches. Furthermore, most implementations are based on specific needs. This project develops a comprehensive framework for Urban Digital Twins, focusing on data integration, storage, visualization, and analysis, all using open-source tools. Our approach integrates various data types, including 3D city models, dynamic air quality data, and external data imported from the client side, such as vector data, 3D city models, and point clouds. We conducted a series of experiments for each step and tackled various challenges. Many configurations are applied before integrating the 3D models, including ground reprojection, geometry type conversion, and format conversion. For the data storage and management, we performed several comparative tests between 3DCityDB and CJDB, which led us to choose CJDB for its simplicity and lightweight nature. A client interface built with the Giro3D framework (based on Three.js) connects directly to the 3D model database via a Flask server. Dynamic data is retrieved via external APIs and stored in a separate database following the SensorThings API standard, allowing time-series analysis. Our framework is standardized and designed based on open-source software, emphasizing the openness, transferability, reusability, and maintainability of Urban Digital Twin. The City2Twin project proposes significant improvements in data analysis, highlighting the importance of having a separate database for storing static and dynamic data, as well as the importance of direct interaction between the client interface and the 3D database for data updates and management. To the best of our knowledge, this work is the first Urban Digital Twin initiative that relies on the CityJSON format, which defines itself as more «developer-friendly».

1. Introduction

Urban Digital Twins (UDTs) are the new paradigm that shifts from traditional urban analytics to integrated, sustainable, and evolving methods and tools for city management and maintenance (Ketzler et al., 2020). Unlocking the potentialities of UDTs starts by defining the minimum technical requirements to implement an open, operational, and easy-to-use interface (Jeddoub et al., 2023). The shift toward a web developer-friendly approach and lightweight specifications and standards has led to addressing some of the UDT challenges related to data integration and interoperability, as well as serving users (especially cities) with integrated and dynamic urban platforms. Many questions about the suitable data models (Lehner et al., 2024), database systems (Kasprzyk et al., 2024), and visualization tools to create UDT platforms remain open to research (Gitahi and Kolbe, 2024). An examination of the literature reveals that most current implementations rely either on specific applications, such as energy use, mobility, and urban planning, etc., or on available data. Up to date, the state of the art is not well developed regarding how UDTs are implemented in practice and does not provide an in-depth analysis of the technical requirements. Furthermore, all attempts to build UDT are based on CityGML and there is no active initiative that relies on the CityJSON format, which defines itself as more «developer-friendly».

The aim of this work is to develop an open and standardized UDT framework that addresses current challenges of multi-source data integration and visualization in a lightweight format. The main steps involve the preparation and integration of the data sets in a standardized way, their storage using a database management system, as well as the visualization using open-source tools. The standardized framework is designed using an open software approach that emphasizes openness, transferability, reusability, and maintainability of UDT.

2. Related work

In recent years, several cities have taken the initiative to develop UDTs to enhance urban management. Jeddoub et al. (2023) established a comprehensive state-of-the-art review of current UDT implementations. Some of these projects are still in the prototype phase, others are under improvements, and some are already fully operational.

The approaches for implementing UDTs generally depend on specific needs and the availability of potentially heterogeneous data. For instance, the EnSysLE project is used for building energy simulation and Amsterdam 3D focuses on city planning. As a result, various methods are developed based on different use cases. Most cities, like Helsinki 3D, Livable City Digital Twin or Berlin 3D, adopt web-based visualization, while others, such as Virtual Gothenburg use game engine visualizations. Among these implementations, some rely on commercial solutions, such as 3D ArcGIS online for karlskrona city, rather than open-source alternatives, which can be costly and offer limited flexibility and customization. Several open-source tools are available for visualizing UDTs on both web and desktop platforms. Most of these viewers rely on CesiumJS with the 3D Tiles data format, such as the Urban Energy Dashboard (Würstle et al., 2020) and the Digital Twin of Sofia City (Dimitrov and Petrova-Antonova, 2021). Other viewers utilize the Three.js library, including Piero, Ninja, and 3D Bag Viewer.

The use of different standards ensures data compliance and interoperability. The CityGML[1] standard defines a conceptual model and an exchange format for the representation, storage, and sharing of virtual 3D city models. It can be encoded in several formats, including GML, which is considered verbose and heavy. An alternative encoding is CityJSON, a JSON-based format that is lighter and easier to understand. Due to its simplicity, several tools and programs have been developed to create, analyze, visualize, and edit CityJSON files, such as Ninja, 3dfier, and Cjio. Another method for storing the 3D model is database encoding, either in a relational database such as 3D City Database 3DCityDB, the most widely used, or CityJSON Database CJDB (Powałka et al., 2023) for CityJSON-based storage, or in a NoSQL database like MongoDB (Kasprzyk et al., 2024). These databases not only store the 3D model but also enable efficient management.

Dynamic data is an essential component that enriches the UDTs, and there are several ways to store and manage these data. For example, dynamic data can be integrated via a «Dynamizer» extension for CityJSON-based visualization (Boumhidi et al., 2024), or through the «Dynamizer» module, which ensures the direct integration and representation of urban object properties through time-series data in CityGML 3.0 (Kutzner et al., 2020). This data can also be stored in a database following the SensorThings API (STA) standard, which manages not only the data itself but also the various elements related to data flows (Santhanavanich and Coors, 2021).

3. Method

We have proposed a generic methodology, supported by a series of technological considerations. This methodology is divided into three major steps: (1) data preparation and integration; (2) data storage and management; and (3) data visualization and analysis. For each step, multiple tests were conducted to ensure the selection of the most suitable tools and approaches for optimal results.

3.1. Study Area and Datasets

The data used consists of two different types: static data and dynamic data. The static data represents the 3D models of urban objects according to the CityGML standard and encoded in JSON format (CityJSON). This 3D model covers the Outremeuse district in the municipality of Liège, Belgium, and contains urban objects of the type of buildings, vegetation, roads, and city furniture. This model is the outcome of the previous project introduced in (Ballouch et al., 2024).

The dynamic data used in this project represents the air quality data of the city of Liège which is provided by the ISSEP and retrieved via APIs. They are measured by sensors placed on lamp posts distributed across the city.

3.2. Data Preparation and Integration

This is a necessary step in our approach, given the heterogeneous nature and sources of the data, which include geodata and sensor data. This step involves carrying out all the necessary preprocessing and configuration.

3.2.1. Static Data Integration

When implementations are based on a specific domain, they do not allow the integration of multiple types of data. This framework, however, aims to support the integration of various data types, such as 3D city models, vector data, and 3D point clouds. Before integrating this data, it must comply with the different tools and methods used. All urban objects in the CityJSON data, such as buildings, trees, and roads, were projected onto the ground for two reasons. First, the surface of base maps in web viewers has a Z value of 0; however, our data includes elevation values, causing objects to float above the ground surface. We attempted to import a Digital Elevation Model (DEM) from online sources, but it does not perfectly match the actual terrain variation. As a result, some buildings appear to float while others sink into the ground. Another reason for this process is that roads, which are not projected onto the ground and therefore are not flat, cannot be imported into the CJDB database (as this is the chosen tool for storage). We assumed that the «CJDB import» functionality requires a flat geometry to populate the «ground_geometry» column, whereas our road objects do not have a flat geometry. To achieve the reprojection, a Python script was developed to determine the minimum Z value for each object and subtracted this value from all other Z values in the same object.

The 3D objects of the City Furniture type (lamp posts) have a geometry type called «Geometry Instance». This type of geometry is difficult to parse and load with our visualization tool. Therefore, it has been converted to a «Multi Surface» type using the CJDB export, which automatically converts geometries of the «Geometry Instance» type into a «Multi Surface» type during the export.

The database used in this project does not directly support CityJSON files. They must be converted into a CityJSON Text Sequence format with a «. city.jsonl» or «.jsonl» extension. This allows for the potentially large CityJSON file to be broken down into individual entities. This conversion was performed using the cjio library.

The measurement stations (lamp posts) on which the sensors are placed are also stored as city objects in the 3D model database. Only the stations in our study area are represented by detailed 3D models (CityFurniture), while those outside the area are represented as points. However, they must conform to the 3D database schema (i.e., adhere to the column type and data structure requirements).

3.2.2. Dynamic Data Integration

Dynamic data is retrieved via external APIs. Since this data may vary over time, it is not optimal to store it directly in the model to avoid overloading the 3D model, which is already heavy itself. Thus, the dynamic data is stored in a database following the SensorThings API standard. This database enables not only the storage, but also better management of various elements related to datastreams, such as the description of the sensors used, the observed properties, and more. Before recording the data, the entities shown in Figure 1 need to be created. For each

[1] https://docs.ogc.org/guides/20-066.html

measurement station, it must be associated with a «Thing» that represents the station, along with its «Location», the «ObservedProperty» corresponding to different pollutants and temperature, the «Sensor» measuring the observed properties, and «Datastream» that links the sensor used and the observed property for each «Thing». To optimize storage, different properties measured by the same sensor (such as the case for PM10 and PM2.5) are grouped into a «MultiDatastream». In this work, we focused on the measurement station in our study area, and the entities were created only for this station. However, it can be easily expanded to include all stations.

Creating these entities in the STA database can be accomplished using the FROST server, a server implementation of the OGC STA, which is deployed using Tomcat. The server is linked to the STA database to execute all queries for creating these entities. Observations can also be recorded via the Frost server; however, this can complicate the process since another server must be maintained. Therefore, we wrote an SQL script on our backend server that executes a query every 5 minutes to record the observations in the database.

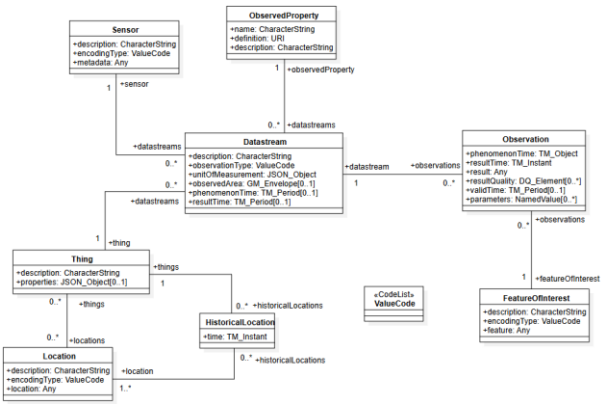


Figure 1. UML diagram of the STA data model (Source: https://developers.sensorup.com/)

3.2.3. External Data Integration

External data can be directly imported into the platform without being stored in a database, allowing users to work with their own data without the constraints of database permissions. They can import CityJSON files, GeoJSON files, or 3D Point Clouds from their local machine. For GeoJSON files, the used visualization tool supports the import of this type of data. However, there is no default support for CityJSON, so to parse and load the CityJSON data into the Three.js scene (the visualization tool adopted in this work), a library called «cityjson-threejs-loader» is used. Similarly, the visualization tool does not allow for direct visualization of point cloud files in «.LAS» format; therefore, it must be converted into a 3D Tiles point cloud using the Python library «py3dtiles».

Since the data may have a different Coordinate Reference System (CRS) than the default one (EPSG:3812), reprojection must be performed; however, this conversion is not applied to the entire 3D model but rather to the transformation matrix that positions the object correctly in the scene. The conversion can be done using libraries such as MapTiler or proj4js.

3.3. Data Storage

We used two separate databases for both static and dynamic data to effectively store and manage them. Both databases are interrelated. The data model is implemented in PostgreSQL.

3.3.1. Selection of the 3D Database

Since there are several approaches to storing 3D data in a database, a choice had to be made. Several criteria were considered, including simplicity, query speed, storage volume, and support for geospatial functionalities. We had to choose between two databases that are most suitable for our project: 3DCityDB and CJDB. Various tests and research were conducted, allowing to establish the comparison.

To better analyze the performance between the two databases in terms of query speed and storage, we conducted tests (similar to those conducted by (Powałka et al., 2023) using our own data. We maintained the same dataset for both databases, which consists of a 3D model in CityJSON format containing nearly 9,500 urban objects.

Once the model is imported into both databases, we already notice a difference in the size of the databases, as presented in Table 1. This can be explained by the complexity of the 3DCityDB schema, which stores the different elements of the 3D model in separate tables. The physical model in this database contains 66 tables, in contrast to CJDB, which compresses the model and contains only 3 tables. In CJDB, the geometries and attributes are stored in a single table called «city_object». The 3D geometries are stored in a column of JSONB type.

To analyze the query speed on both databases, we selected several queries for comparison. The first three queries are applied to attributes, while the fourth one focuses on geometries. Although the queries are not exactly the same due to differences in the database schemas, they generally return similar results corresponding to our requests. The time results of these queries are presented in Table 1.

Q1: Retrieve a building object using its ID
Q2: Add a new attribute to objects of type «Building»
Q3: Retrieve buildings with a Level of Detail 2
Q4: Retrieve geometries of 100 buildings in 3D GeoJSON format

Criteria\Database	3DCityDB	CJDB
DB volume (MB)	252	97
Q1 (ms)	33.36	33.54
Q2 (ms)	53.75	76.87
Q3 (ms)	191	302
Q4 (ms)	92	48

Table 1. Comparison of database sizes and query times for CJDB and 3DCityDB

After conducting several tests and comparisons between the two databases, we can confirm the results of the test conducted by (Powałka et al., 2023), which shows the maturity of 3DCityDB

in handling semantic surfaces and attributes (Q1, Q2 and Q3) with consistency. However, the main advantage of using CJDB lies in its simplicity of use and understanding, as well as its method of storing 3D geometries, which makes them easy and rapid to retrieve (Q4), and results in smaller sizes in the database. In addition, CJDB is the most suitable database solution for the CityJSON-based encoding with the direct approach of visualization presented in section 3.4.1. Given that most current UDT implementations are generally based on the 3DCityDB, we found it interesting to further explore the performance and limitations of CJDB. Several optimizations have been applied to the database to enhance its performance.

3.3.2. Optimization of the CJDB Database

According to the CJDB database schema, the footprints of each object are stored in a PostGIS geometry-type column, called «ground_geometry». During data import, the CJDB importer retrieves the ground geometry of each object; however, in some cases, this function is not properly executed. For example, for some objects in our CityJSON data, the column «ground_geometry» contains only an empty Multipolygon instead of a Multipolygon with geometries.

We assumed that this issue occurs because, during data import, the CJDB importer tries to detect a flat ground surface, meaning that the Z values of the polygons are identical or have minimal differences. If this is not the case, it returns an empty geometry. To ensure that the footprints are always stored, regardless of the case, a Python script was written and should be executed once the data has been properly imported into the database. The script retrieves the ground geometries of each object and adds them to the «ground_geometry» column.

3.3.3. CityThings concept

The CityThings concept relies on the SensorThings and CityGML standards to establish a connection between the sensor and the 3D model on which the sensor data is measured (Santhanavanich and Coors, 2021). We extended this concept based on CityGML and 3DCityDB with light encodings (e.g., CityJSON and CJDB). It allows linking the static and dynamic databases, where an urban object (lamp post stored in CJDB) corresponds to an object where a sensor is placed (measurement station stored in STA Database as a « Thing »). By using this approach, we can retrieve information about the 3D urban object corresponding to the sensor data flows, and vice versa (see Figure 2).

Figure 2. The CityThings concept applied to a sensor in our study area.

Technically, to ensure this link, the instance of «Thing » defined in the UML diagram of the STA data model stores as a property the unique identifier of its corresponding 3D urban object (which, in our case, is the ID of the lamp post). Figure 3 shows an example of how the urban object identifier is stored in the « Thing » entity.

```
{
    "name": "Outremeuse Station",
    "description": "Outremeuse Measurement Station",
    "properties": {
        "address": "Place Delcour, Outremeuse",
        "city_object_id": 10177
    }
}
```

Figure 3. Example of storing the 3D object identifier in the «Thing» entity properties.

3.4. Data Visualization:

3.4.1. Choice of visualization tool and approach

This step involves retrieving objects from the database and visualizing them on web-based platforms. We have selected web visualization as it is more optimal and does not require any additional software or hardware installation by the user. At this stage, we have compared two existing approaches in the literature:

- The direct approach: the objects are retrieved directly from the database by running an SQL query. The returned object is in GeoJSON or CityJSON format, including both attributes and geometries. In this approach, there is a direct connection between the client and the 3D database. Updates are made automatically after any modification, such as the deletion or modification of urban objects. However, loading time and visualization can be quite slow and require more resources, as the data is loaded in a format that is not suitable for large data visualization.

- The indirect approach is the most widely adopted in most UDT implementations. It consists of exporting manually 3D models in 3D tiles format, storing them on cloud servers such as Cesium Ion and retrieving them by ID. The use of these adapted formats guarantees rapid visualization. However, a direct link between the visualization and the database where the objects are stored and managed is not assured. This means that any changes to urban objects require a new export for visualization.

Given that this project focuses on a neighborhood with around 10,000 3D objects, its scale is relatively small. Moreover, this project aims to implement a solution that is not restricted by limitations such as the lack of a direct link between the database and the client or the need for manual data export and conversion into a web-streamable format. It also aims to further explore the use of a database in the backend for the management of 3D objects within the context of UDT. Consequently, the direct approach was adopted.

Common UDT visualization tools and frameworks have been implemented and fully tested. Their differences lie in loading speed, supported formats, built-in features, and other aspects. We investigated two JavaScript libraries, CesiumJS and Three.js, to parse and load data in an interactive way. These are both web-based graphics libraries used to create interactive 3D graphics in a browser.

CesiumJS is more specialized for geospatial applications and offers advanced geospatial features. However, with the direct approach, 3D data will be retrieved and viewed in 3D GeoJSON format to facilitate data flow from the database, through the server, and to the client. This method is very slow and requires more memory. For Three.JS, while it lacks built-in support for geospatial functionalities, it can efficiently visualize data in

CityJSON format in a shorter time by using the « cityjson-threejs-loader » library. Since we conducted our first attempt with the approach based on CesiumJS and 3D GeoJSON, we observed that the approach based on Three.JS and CityJSON brings significant improvements in loading speed, memory usage, and navigation fluidity. To address the lack of geospatial functionalities in Three.js, we used the Giro3D framework. It is a recent open-source JavaScript framework designed to visualize and interact with heterogeneous data in 2D, 2.5D, or 3D. Although it is based on Three.JS, it integrates support for geospatial functionalities by using the Openlayers library.

3.4.2. 3D Data Retrieval

Initially, when the user loads the page, an HTTP request is immediately sent to the server, which is a Flask server, that retrieves 3D data from the CJDB database. This database automatically exports data in CityJSON Lines format using the CJDB Exporter function. Once the data is retrieved, it must be converted to CityJSON format to be readable by the visualization tool. This conversion can be done using CjSeq, a program for creating, processing, and modifying CityJSONSeq files, as well as converting them to CityJSON files (Ledoux et al., 2024). The CityJSON data is then sent to the client side to be parsed and loaded into the Giro3D scene.

Figure 4 illustrates the UML sequence diagram of this architecture.

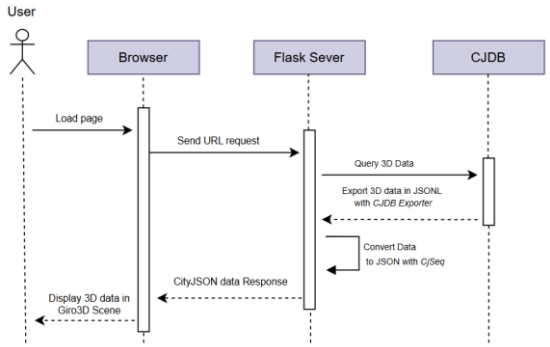


Figure 4. UML sequence diagram of 3D Data Loading.

3.4.3. Dynamic Data Visualization

The approach used for visualizing dynamic data is independent of the one applied to static data. This data is retrieved through APIs and can either be directly displayed on the interface or saved in the STA database for time series analysis.

- **Displaying Punctual Data**

First, a request is made to fetch information about available sensors, such as their 2D position, ID, address, etc. This initial request allows the sensors to be displayed in the scene and enables modifications to the database if new sensors are added or removed.

The second request is sent when the user clicks on a specific sensor. This request retrieves the latest measurements taken by the sensor, including the concentration of each pollutant, temperature, air quality index (AQI) value, and the timestamp of the measurement (aqi_timestamp). Figure 5 illustrates an example of the results from the second request, which are then displayed in a graph using the Chart.js JavaScript library.

Figure 5. Example of retrieved data for a specific sensor.

- **Displaying time series data from the database**

The time series values are retrieved from the STA database. An SQL query is executed to extract data from the «OBSERVATIONS» table, ordering the results by date and time. The results are then processed to group the data by date. Each date is associated with a dictionary containing values for various pollutant concentrations, temperature, and a calculated air quality index value (since this value is not stored in the STA database and must be calculated based on pollutant concentrations). The data is then formatted into a JSON dictionary, where the keys represent the date and time. These data are then returned and displayed in a graph (see Figure 9).

3.5. Technical Architecture of the UDT

Based on the various tests and comparisons, Figure 6 represents the final technical architecture to be adopted for the implementation of this project. The three main stages are presented along the vertical axis, and at the top of the diagram, there are the different types of data (static and dynamic) to be integrated, such as data to be injected into the database or data that will be directly visualized on the platform. The two databases (CJDB and STA Database) are connected via the CityThings concept. A Flask server is implemented to handle various requests and perform certain calculations and conversions. Finally, all data is visualized on a single platform using the Giro3D framework.

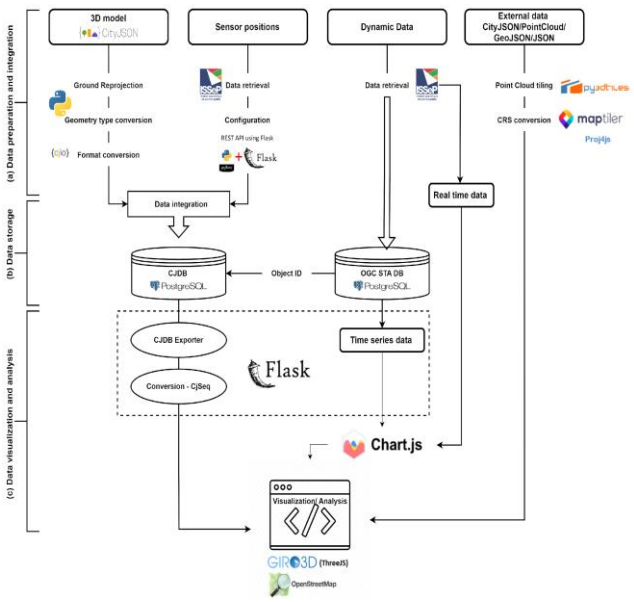


Figure 6. Technical architecture of the UDT.

3.6. Various features integrated into the platform

To provide a fully featured and user-friendly platform, a variety of functions have been implemented.

- Retrieving information related to an object by its ID

By using the raycasting principle, we can retrieve information about the clicked object such as their «object_id», the attributes of this object and the ID of the semantic surface if it exists. A «div overlay» is then displayed, containing this information.

- Retrieving information related to a specific surface

If the model contains information about surfaces (e.g., wall surfaces, roof surfaces, or other attributes), this information can be retrieved using the surface ID obtained through the raycasting.

- Attributes manipulation

The «div overlay» that appears after double-clicking on a specific object contains three buttons for modifying, adding, and deleting attributes. The process for these three functionalities is similar. After identifying the concerned 3D object, a filter is applied using its ID. Operations are then performed on this object: either assigning a new value to an existing attribute, adding a new attribute, or deleting an attribute. To update the database, endpoints are created to execute an SQL query that modifies the database. If the attributes concern semantic surfaces, we modify the «semantics» property instead of the «attributes» property.

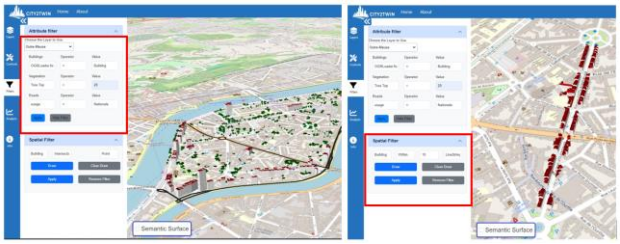
- Multi-layer management

A layer management section has been added to handle multiple data sets loaded in the scene. For each layer, we can toggle its visibility, zoom in on the layer, or download the layer for use in other applications (see Figure 8-a).

- Filtering objects

For filters based on attributes, we specify first which layer the filter will be applied to. After setting various conditions based on the object type and attributes (see Figure 7-a), these conditions are grouped and then applied to the current city model to create and load in the scene a new city model containing only the desired objects.

For spatial filters, this functionality has not yet been integrated on the client side. It must be executed on the server side of the database using PostgreSQL's spatial extension «PostGIS» and the «ground_geometry» column in CJDB, which contains the objects' footprints in PostGIS geometry format. First, we specify the type of object, the type of spatial functions (DWITHIN, DWITHOUT, INTERSECT, or DISJOINT), and the type of geometry to be drawn directly on the base map (see Figure 7-b). These parameters are then sent to the server, and a spatial query is executed in the database, which returns the desired objects.



(a) (b)

Figure 7. a) Attribute filter: buildings of type «Building», trees < 20m, and roads for national use; b) Spatial filter: buildings located within 10m around a road segment.

- Optimization of visualization

All visualization and control parameters can be adjusted based on the user's needs using the Dat.GUI library. The user can change the sky color, adjust the light intensity, display objects based on the Level of Detail (LoD), and change the colors of urban objects based on a specific attribute. Additionally, they can customize the colors according to the type of object (e.g., Building, Road, etc.) or the semantics of the building (e.g., RoofSurface, WallSurface, etc.).

- User management

To manage different permissions for modifying data in the database, a user management system has been integrated. A separate database is used to store user information, including name, email address, password, and role, which can either be «admin» or «user». Once the database schema is created, endpoints for login, logout, and registration are established. Users must have the «admin» role to access the database modification capabilities; however, they can modify the data they have imported.

4. Results and Use Cases

This section presents an overview of the results we have obtained and explains the use cases that have been implemented within the platform. These use cases demonstrate the platform's ability to operate across various fields and highlight the advantages of digital twins in urban environments.

4.1. City2Twin Platform

The City2Twin platform is designed to be simple and intuitive, enabling users to navigate efficiently while offering enhanced visualization features, as shown in Figure 8.

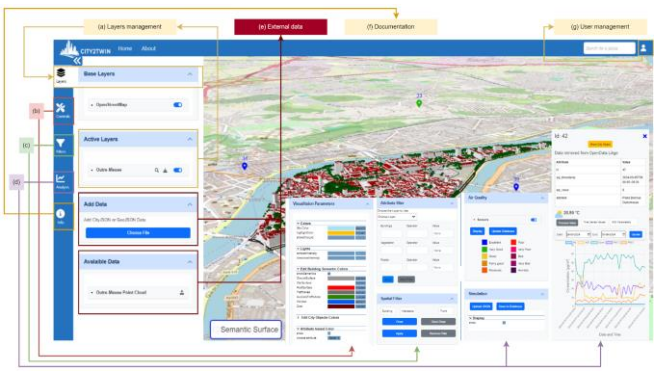


Figure 8. The CITY2TWIN overview: (a) layer management panel, (b) controls panel, (c) attribute and spatial filters, (d) urban analysis (air quality and energy simulation panel), (e) import external data, (f) documentation, (g) user management.

To ensure the platform's capability in handling large datasets, we imported a CityJSON LoD 2 file of Zurich city from the client side, containing approximately 200,000 objects and sized at 300 MB. The model was successfully loaded.

4.2. Use Cases

In the fourth sidebar (see Figure 8-d), two applications are presented. The first section contains the application dedicated to air quality analysis, while the second focuses on energy simulation in buildings.

4.2.1. Air Quality Analysis

The dynamic data are used to perform air quality analysis. As explained in the methodology, data is retrieved via an external API and can either be stored or directly visualized. This near real-time visualization informs users of the current air quality in the area. Regarding time series data from the database, various analyses can be conducted. The X-axis can represent dates and times in chronological order, different hours of the day (from 00:00 to 23:00), or days of the week, while the Y-axis can represent temperature, AQI values, or concentrations of pollutants. The data can also be filtered for a specific period.

By combining different parameters on the X and Y axes and adjusting date ranges, a more comprehensive and advanced air quality analysis can be performed (see Figure 9). For example, it is possible to identify the times of day or days of the week when air quality is at its best, or simply to track the evolution of a parameter over time. Seasonal trends can also be evaluated if sufficient data is available in the database. These analyses support decision-makers and governments in making informed choices to improve citizens' quality of life.

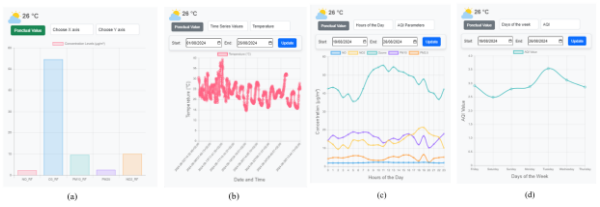


Figure 9. (a) Concentrations of each pollutant at time t; (b) Temperature variations from 08/01 to 08/26; (c) Average hourly concentrations calculated between 08/19 – 08/26; (d) Average daily air quality value calculated between 08/19 – 08/26.

4.2.2. Energy Simulation

Simulation results are typically provided in CSV or JSON files. These files contain key-value pairs, where the key represents the building ID, and the value is the associated result. Although a real simulation cannot be performed due to the lack of necessary data in our 3D model, an approach has been developed to support data from these simulations. For this purpose, we created a JSON file containing random data for testing (considered as a result of a simulated energy demand). This file can be injected into the model for visualization, as shown in Figure 10, and can also be stored in the database as an attribute of urban objects.

Figure 10. Visualization of integrated simulation data on the model.

5. Discussion

5.1. Used Tools

Based on the tests conducted on both databases, we observed the importance of having an easily understandable schema, like the one proposed by CJDB. However, one of the challenges encountered with this database is the issue of schema consistency. Unlike 3DCityDB, where attributes are stored in a separate table with constraints on variable types, in CJDB, they are stored in a JSONB column in the same table as the urban objects. Consequently, no constraints on the attribute value types are applied. We had to implement these constraints manually; for instance, when updating an attribute value, it first compares the type of the old value with that of the new one.

Although CesiumJS is the most widely used library, Three.js was chosen for this project to reduce loading times and improve performance while maintaining a direct link between the client and the database. Additionally, Three.js enables compatibility with CityJSON, allowing us to benefit from its various advantages, such as support for handling semantic surfaces, which is very challenging to achieve with CesiumJS and the 3D GeoJSON format.

5.2. Evaluation

To assess the contributions of this project, we compared it with existing operational viewers. Based on this comparison, we conclude that City2Twin has introduced significant improvements in data analysis, particularly with attribute and spatial queries, as well as the semantic surfaces. Regarding attribute manipulation, it offers a more intuitive interface compared to the way attributes are modified in Ninja and also implements constraints on value types to reinforce data consistency.

The developed approach has demonstrated the importance of having a 3D database that directly interacts with the client interface, allowing for the manipulation or modification of urban objects without relying on manual exports and file hosting on a server. In addition to this database-based approach, it also supports files imported from the client interface (such as 3D point clouds, CityJSON, or GeoJSON). Thus, it can function as a simple 3D visualization tool on the web without the need for database configuration.

Although dynamic data can be injected into the model itself via an extension, the use of the STA Database has facilitated the management and storage of this data. This standard not only allows for the storage of data but also includes the management of various elements related to datastreams with their descriptions, such as the sensor used, the observed properties, the station and its location, etc.

The developed framework is a generic framework that can be used for other applications requiring dynamic data or not: e.g., mobility, energy demand, urban planning, etc. All the tools and solutions used in this project are open-source and free, facilitating the development of extensions for specific needs.

6. Conclusion

This project focused on developing a framework for the implementation of an UDT that encompasses various data integration, storage, visualization, and analysis. It has led to a functioning open UDT, «City2Twin», that tackles the gap in the literature regarding the technical requirements and enhances the UDT developments while taking advantage of lightweight standards and solutions. The developed framework is based on the CityJSON encoding. This format offers several advantages due to its lightweight nature and simplicity, facilitating its manipulation and integration into a visualization platform. For data storage and management, after various tests and comparisons between CJDB and 3DCityDB, the CJDB database was chosen for its simplicity, lightweight nature, and compatibility with an approach based on CityJSON encoding. Direct interaction with the database is a major advantage, allowing for continuous maintenance and efficient data updates without the need to export files manually. Additionally, the use of a separate database (STA Database) with the «CityThings» concept prevents the direct storage of dynamic data within the model, which can already be heavy. It also enables effective management of the various elements related to datastreams.

In conclusion, the City2Twin framework demonstrates a robust approach to urban management through the integration of 3D city models and dynamic data. Although this work successfully met our main objective, future improvements could focus on optimizing data retrieval methods and expanding the platform's capabilities to support more advanced urban analysis and several types of data.

References

Ballouch, Z., Jeddoub, I., Hajji, R., Kasprzyk, J.-P., Billen, R., 2024. Towards a Digital Twin of Liege: The Core 3D Model based on Semantic Segmentation and Automated Modeling of LiDAR Point Clouds. *ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci.*, X-4-W4-2024, 13–20. doi.org/10.5194/isprs-annals-X-4-W4-2024-13-2024.

Boumhidi, K., Nys, GA., Hajji, R. (2024). Integrating Dynamic Data with 3D City Models via CityJSON Extension. In: Kolbe, T.H., Donaubauer, A., Beil, C. (eds) *Recent Advances in 3D Geoinformation Science. 3DGeoInfo 2023. Lecture Notes in Geoinformation and Cartography. Springer, Cham*. doi.org/10.1007/978-3-031-43699-4_45.

DImitrov, H., Petrova-Antonova, D., 2021. 3D city model as a first step towards digital twin of Sofia City. Presented at the *Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., - ISPRS Arch.*, 23–30. doi.org/10.5194/isprs-archives-XLIII-B4-2021-23-2021.

Gitahi, J., Kolbe, T.H., 2024. Requirements for Web-Based 4D Visualisation of Integrated 3D City Models and Sensor Data in Urban Digital Twins, in: Kolbe, T.H., Donaubauer, A., Beil, C. (Eds.), *Recent Advances in 3D Geoinformation Science, Lecture Notes in Geoinformation and Cartography. Springer Nature Switzerland, Cham*, 707–725. doi.org/10.1007/978-3-031-43699-4_43.

Jeddoub, I., Nys, G.-A., Hajji, R., Billen, R., 2023. Digital Twins for cities: Analyzing the gap between concepts and current implementations with a specific focus on data integration. *Int. J. Appl. Earth Obs. Geoinf.* 122, 103440. doi.org/10.1016/j.jag.2023.103440.

Kasprzyk, J.-P., Nys, G.-A., Billen, R., 2024. Towards a multi-database CityGML environment adapted to big geodata issues of urban digital twins. The *Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci.*, XLVIII-4-W10-2024, 101–106. doi.org/10.5194/isprs-archives-XLVIII-4-W10-2024-101-2024.

Ketzler, B., Naserentin, V., Latino, F., Zangelidis, C., Thuvander, L., Logg, A., 2020. Digital Twins for Cities: A State of the Art Review. *Built Environment 46*, 547–573. doi.org/10.2148/benv.46.4.547.

Kutzner, T., Chaturvedi, K., Kolbe, T.H., 2020. CityGML 3.0: New Functions Open Up New Applications. PFG 88, 43–61. doi.org/10.1007/s41064-020-00095-z.

Ledoux, H., Stavropoulou, G., Dukai, B., 2024. Streaming CityJSON datasets. The *Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci.*, XLVIII-4-W11-2024, 57–63. doi.org/10.5194/isprs-archives-XLVIII-4-W11-2024-57-2024.

Lehner, H., Kordasch, S.L., Glatz, C., Agugiaro, G., 2024. Digital geoTwin: A CityGML-Based Data Model for the Virtual Replica of the City of Vienna, in: Kolbe, T.H., Donaubauer, A., Beil, C. (Eds.), *Recent Advances in 3D Geoinformation Science, Lecture Notes in Geoinformation and Cartography*. *Springer Nature Switzerland, Cham*, 517–541. doi.org/10.1007/978-3-031-43699-4_32.

Powałka, L., Poon, C., Xia, Y., Meines, S., Yan, L., Cai, Y., Stavropoulou, G., Dukai, B., Ledoux, H., 2023. cjdb: a simple, fast, and lean database solution for the CityGML data model.
Santhanavanich, T., Coors, V., 2021. CityThings: An integration of the dynamic sensor data to the 3D city model. *Environment and Planning B: Urban Analytics and City Science 48*, 417–432. doi.org/10.1177/2399808320983000.

Würstle, P., Santhanavanich, T., Padsala, R., Coors, V., 2020. The Conception of an Urban Energy Dashboard using 3D City Models., 523–527. doi.org/10.1145/3396851.3402650.