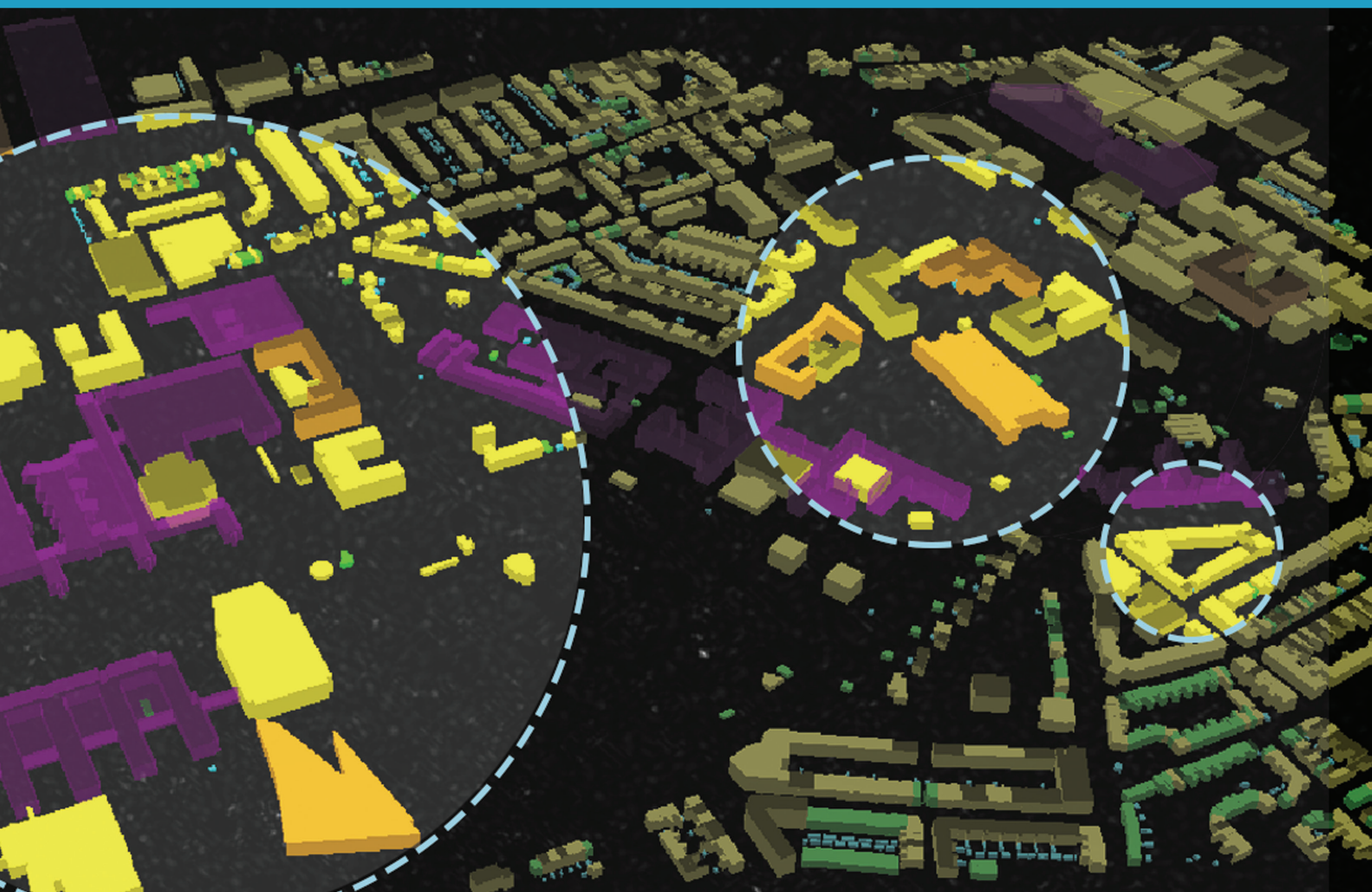
MSc thesis in Geomatics

Directly serving 3D Tiles from a Geo-DBMS

Yue Yang

2024



MSc thesis in Geomatics

Directly Serving 3D Tiles From A Geo-DBMS

Yue Yang

April 2024

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

The work in this thesis was carried out in the:

Geo-Database Management Center
Delft University of Technology

Supervisors:	Dr.ir. B.M. Meijers
	Prof.dr.ir. P.J.M. van Oosterom
Co-reader:	Dr. G. Agugiaro

Abstract

The rise of urban digital twins emphasises the critical role of representing real-world environments for decision-making and collaboration among stakeholders. This leads to an expectation of effective management and visualisation of large-scale urban data models. 3D Tiles is an open specification designed for streaming and rendering massive heterogeneous 3D geospatial datasets. It is widely adopted in fields such as urban planning and engineering construction, where geospatial data is crucial for decision-making and collaboration.

However, as the 3D data size is growing, the performance of traditional file-based solutions is facing challenges. Currently, dissemination and visualisation of 3D data are often reliant on file-based systems. Users suffer from problems such as long load times, data inconsistencies, and lack of flexibility. In this thesis, an approach for compactly storing both geometries and attributes in the Database Management System (DBMS) and efficiently serving data compliant with 3D Web standards to the client is proposed, advocating the direct serving of 3D Tiles from the database.

Acknowledgements

Within this section, I would like to take the opportunity to express gratitude for the support and guidance I have received throughout this journey.

Firstly, I would like to thank *Martijn Meijers* for this beautiful topic and support in the thesis progress. The insightful advice and well-structured feedback are crucial to project development and prototype refinement. Next, I would like to thank *Peter van Oosterom* for his critical aiding mindset and kindness in helping me. I sincerely appreciate your guidance and feedback, which deepen my understanding of Geomatics in the built environment. A great thanks goes to *Giorgio Agugiaro* for giving me valuable suggestions on my thesis project, and for attending the P4 meeting. Finally, I would like to thank *Bastiaan van Loenen* for hosting the meetings during my graduation procedure.

In addition, I would like to thank my friends in Delft. It is fortunate to meet and know you on this land. I treasure every rainy and windy day we spent together, as much as the sunny times. Special thanks to my friends in Australia who always support me despite the significant time difference. Moreover, JJ LIN's music lighted me whenever I felt demotivated. Lastly, I would like to thank my parents and my cousins for their continued support.

Yue Yang
Delft, April 2024

Contents

List of Figures

List of Tables

Acronyms

1. Introduction

1.1. Motivation and problem statement

As urban digital twins gain momentum, a wide range of use cases require 3D geospatial data visualisation to represent the real world, where rapid response to the built environment is critical. Ensuring timely updates and easy access is important to fostering collaboration among stakeholders such as citizens, municipal decision-makers, and other urban planning entities. This also promotes bottom-up strategies that promote community engagement, enabling actively shaping the development of the city.

Over the last decade, there have been endeavours to enhance the dissemination and visualisation of 3D data. For example, Cesium provides a platform that is accessible to anyone with a Cesium ion account. The platform seamlessly optimizes, hosts and streams 3D geospatial data, enabling users to create presentations using their 3D geospatial data. In addition, 3D Tiles designed by Cesium allows massive data to be divided into smaller chunks [Cesium and OGC, 2019]. This supports progressive loading and helps solve the problem of long retrieval and loading times when there are many buildings.

Figure 1.1.: 3D Tiles streaming and progressive loading on Cesium

Typically, data is maintained in a DBMS and then serves as data copies in a file format. While easy to store data, this approach raises concerns about spatial and temporal inconsistencies. The definitive mechanism to guarantee simultaneous updates to both geometry and attribute values is absent in file management. For example, information in the file may be outdated and inconsistent with the database. In addition, this file-based approach lacks efficiency and flexibility in data access. Despite the fact the 3D Tiles is optimised for massive data,

users experience a long loading time with the current 3D viewer, Cesium, because of the large number of tiles in a large-scale scene. The 3D models are divided into fixed-size tiles to organise the 3D Tiles. This makes it difficult to search for the digital representation of specified regions and filter data. Another concern is that a file-based system increases redundancy and impacts storage efficiency.

This highlights the need for alternative solutions that directly serve the data from the database. Instead of exporting 3D Tiles as separate files and then serving them, 3D Tiles are streamed directly from the database to the web application. This streamlined approach provides immediate access to the most current data and simplifies the process of accessing and visualising 3D spatial data.

Furthermore, 3D city models are not only for 3D visualisation but also often used for various Geographical Information System (GIS) simulation and analysis tasks. Due to the large size and complexity of the country-wide 3D geospatial data, the GIS software vendors and service providers face many challenges when building 3D spatial data infrastructures for realising the efficient storage, analysis, management, interaction, and visualisation of the 3D city models [Yao et al., 2018]. Spatial databases enable a wide range of geometric operations. Apart from data retrieval, storing geometries inside the database could also handle 3D data more easily and efficiently, including validating and flexibility to perform spatial analysis tasks.

However, there is currently no standard procedure for manipulating and displaying 3D Tiles for web visualisation. **This research aims to research the possibility of compactly storing geometries and attributes in the database and effectively generating data formats that comply with 3D Web standards**. We examine the benefits of serving 3D Tiles data directly from a database management system (DBMS), and explore suitable database solutions for reducing duplication, enhancing interoperability and providing fast data access.

To formulate the research question properly, an overview of the methods, related to 3D data modelling in the database as well as data transfer between client and server, is given in the next chapter.

1.2. Potential use cases

Through an exploration of direct serving 3D Tiles from Database Management Systems (DBMS), our objective is to unveil the benefits of this approach within the realm of 3D spatial data management and visualisation. In which scenarios do applications derive significant benefits from direct serving of 3D data from a Database Management System (DBMS) as opposed to file-based storage methods?

Case 1: 3D Land Administration Domain
In the context of urban development where certain buildings exceed height limitations, efficiently visualising this information is crucial for land use and planning departments. However, the large size of the dataset takes a long time to load, and the need for a fast search for specific spatial content is not met. To address this challenge, directly serving the relevant buildings as 3D Tiles based on user-defined conditions becomes imperative. By leveraging database capabilities to stream only the buildings that exceed height limitations to the web application, unnecessary data transfer and processing overhead are minimised. Spatial indexing and clustering in the database can also speed up accessing the spatial data.

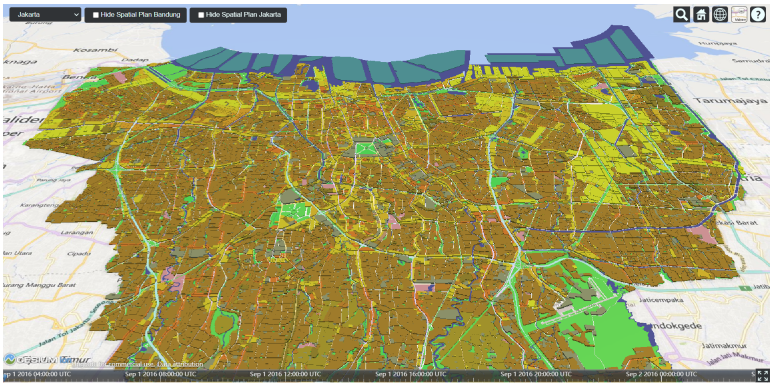


Figure 1.2.: Overview of Cesium prototype showing spatial planning information in Jakarta, Indonesia [Indrajit, 2021]

Apart from filtering objects based on attribute querying, analysis tasks based on geometric operations can be performed. For example, to examine if a building is inside a parcel, a point-in-polygon metric operation can be performed for each point constituting the footprint of the building [Biljecki et al., 2015]. In the PostgreSQL spatial database with PostGIS extension, this can be performed directly by spatial functions. Another example is performing visibility analysis. 3D city models are essential for visibility analyses, such as determining the Line of Sight (LOS) between two points and estimating sight volumes within urban environments. For instance, they are used in property valuation in urban areas, because the view from an apartment may significantly impact the property values [Kara et al., 2020].

Case 2: Municipal Engineering Construction
An example is that in a project the industry technicians change attributes in the database, however, 3D visualisation on the web client is not updated in time. The necessity of directly serving 3D Tiles from the database stems from the need for a way to guarantee the synchronisation of spatial and relational attribute data in a geometry database.

A municipality requires a comprehensive database management system to monitor and manage the utility network in this area. The updates in DB involve maintenance, repair, and changes of the utility, for example, the geospatial location of water pipes. Spatial attributes and associated attributes must be updated simultaneously to ensure data accuracy and integrity. The issue of a file-based approach is the absence of a guarantee for updating the geometry and attributes simultaneously.

Additionally, engineering constructions like tunnels often span large spatial extents, posing challenges for efficient search and fast retrieval of relevant information. A database management system (DBMS) enables efficient queries within the system and targeted retrieval of specific regions.

Case 3: Bridging multiple Digital Twin databases
Urban digital twins, as the means of monitoring physical assets and simulating dynamic scenarios, facilitate decision-making. However, there is no standard digital twin database. Challenges need to be addressed, such as scalability, reproducibility and interoperability.

It is acknowledged that the physical world is complex, and it is impossible to manage one model database for an entire digital world representation. Multiple model databases have been developed in different domains, including geospatial aspects and other non-geospatial

aspects, such as transportation, marine and so on. The main challenge is how to link these fields and obtain the required information from numerous data.

Collaboration between multiple digital twin databases can be costly. There is a gap between the need to build larger systems from multiple digital twins and the desire to help reduce costs through digital twin models. In the short term, bridging the differences between multiple databases is more manageable than standardizing an entire massive digital twin system. It is not easy to visualise the digital twins, given that these databases are not connected and are built with different standards. 3D Tiles, as an OGC standard, enhances data sharing in academia and the industry. It is a promising way to compose these pieces from multiple databases into 3D Tiles and directly serve from the database.

1.3. Objectives & Research Question

In response to the identified challenge and the imperative to address specific use cases, the main research question is formulated as follows:

How to compactly store geometries and attributes in the database and effectively serve 3D Tiles to the client?

To answer the main question, the following sub-questions are relevant:

1. How to organise the database storing raw data, for example, storing raw geometries as polygons, multi polygons or polyhedrons?
2. How to define the mapping rules for storing 3D Tiles in a relational database?
3. How to derive meshes (triangulated geometries) from raw data?
4. How to avoid potential problems such as data redundancy and data inconsistency?
5. How to define the tiling rules?
6. What kind of spatial and attribute queries could be performed based on the proposed data model?
7. What are the advantages and disadvantages of generating 3D Tiles on the fly compared to a file-based approach?

Efficiently storing and seamlessly visualising 3D geospatial data in web-based GIS applications forms the core focus of this research. The main objective revolves around finding optimal methods for storing both geometric information and associated attributes within a database while ensuring compliance with 3D web standards. This pursuit gives rise to several critical research objectives.

This research primarily emphasises two key aspects: **the compact database storage of 3D Tiles and the visualisation of geospatial data through web-based GIS**.

1.4. Thesis Outline

The remainder of this thesis is organised as follows:

Chapter 2: Theoretical Background
This chapter provides the relevant theoretical background and explains the current web technologies and DBMS methods that support the management and visualization of 3D city models in general. It explains the fundamental concepts of 3D Tiles and helps explore a DBMS approach for managing and serving 3D Tiles.

Chapter 3: 3D Tiles Approach
This chapter introduces the database model requisites, geometry storage and topological references, and hierarchical storage structures for efficient 3D Tiles organisation. It explores methodologies for organising 3D Tiles, and explains system architecture, query procedures, and visualisation strategies.

Chapter 4: Implementation and Experiments
This chapter identifies the tools used and the preparatory phases for dataset processing and describes the prototype implementation for testing the proposed storage models and web retrieval methods.

Chapter 5: Results and Analysis
This chapter describes and analyses the benchmarks used to examine the proposed approach and presents visualisation results.

Chapter 6: Conclusion and Future Work
This chapter provides the answers to the research question and discusses the contributions and limitations encountered. Finally, relevant future work is given.

2. Theoretical Background

The theoretical background chapter aims to provide a foundational introduction for the subsequent thesis sections. Section 2.1 outlines digital representation for the real world. Section 2.2 focuses on WebGIS standards and services. Section 2.3 explains conceptual and logical design, indexing and clustering methods, and geometric operations within DBMS. It also describes the current DBMS approaches for 3D city model management and visualisation. Section 2.4 focuses on 3D Tiles. These lay the foundation for further developing a DBMS approach for managing and serving 3D Tiles.

2.1. Modelling the real world

2.1.1. 3D spatial data representations

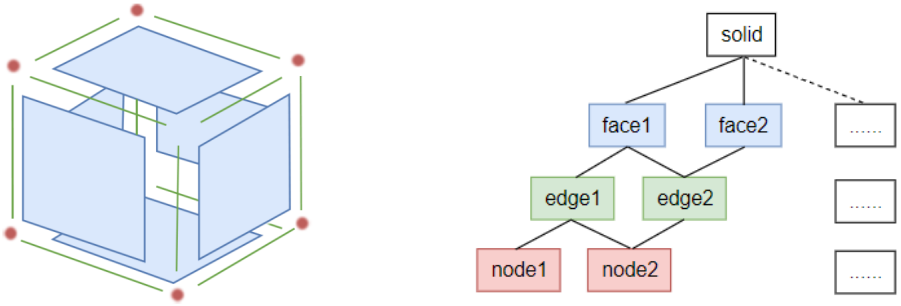
Spatial modelling involves the process of translating intricate real-world objects into digital representations. This progress requires abstracting entities resembling real-world elements into representations suitable for computer storage. There are different representation schemes for data models in the context of spatial modelling, the main difference lies in the way that they decompose and discretize the space into a defined set of elements [Arroyo Ohori, 2016].

Boundary representation

Boundary Representation (B-Rep) in computer-aided design leverages mathematical theorems like the Jordan curve and Jordan-Brouwer theorems to depict complex Three dimensional (3D) objects using 2D boundary surfaces. These structures can consist of decomposed surfaces composed of basic cells like triangles or polygons. B-rep describes the geometric shapes, interconnectivity, and relationships among vertices, edges, faces, and volumes. These elements form the representation of complex 3D objects. It forms the foundation for various operations in 3D modelling, including Boolean operations, mesh generation, etc. However, managing and creating B-rep models can be intricate, especially for complex objects or non-manifold surfaces.

Constructive solid geometry

Constructive Solid Geometry (CSG) enables complex 3D object representation through simple geometric primitives and Boolean operations. It efficiently creates intricate shapes in Computer-Aided Design (CAD) and game engines, ensuring a watertight object. Typically, a complex object is formed by combining basic geometric primitives using Boolean operations, which manipulate sets of points [Kragler, 2016]. These operations include union, difference, intersection, and others. These primitives and operations can be organised hierarchically and represented as a CSG tree.



(a) object modelling (b) corresponding graph

Figure 2.1.: Boundary Representation

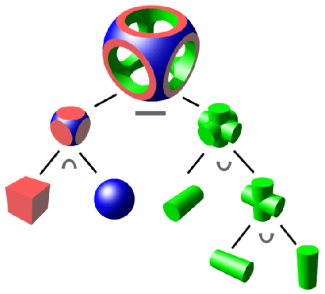


Figure 2.2.: Hierarchical CSG tree [Kragler, 2016]

Parametric approach

Parametric modelling serves as a fundamental principle in Computer-Aided Design (CAD), particularly in Mechanical CAD (MCAD). It offers a systematic approach to defining entities through adjustable parameters that encompass various measurements and geometric features. Its essence lies in its flexibility, allowing swift modifications to parameters to alter entity characteristics and relationships between components. Nevertheless, this modelling approach has its limitations of the manual definition of geometries and features by designers, which requires human labour.

In the Building Information Modeling (BIM) domain, constructive solid geometry and parametric modelling are key methods for modelling geometries. Regarding the GIS world, boundary representation is typically used, which is the geometric modelling paradigm we focus on in this project.

2.1.2. Open data models

This section discusses the practice of various open data models. Open data models provide a standardised framework for representing and organising data, fostering interoperability, accessibility, and collaboration in data sharing and exchange.

GPKG

The GeoPackage (GPKG), an Open Geospatial Consortium (OGC) standard, is a versatile container for various types of spatial data elements, including vector data, raster data, and associated attributes, into a single SQLite database file format. GPKG complies with the OGC Simple Features standard, encompassing various geometry types such as Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Among these, the MultiPolygon type is noteworthy as it can encapsulate both 2D and 3D geometries.

CityGML

Geography Markup Language (GML) encompasses more complex data structures and embedded topological information. International standards such as City Geography Markup Language (CityGML), an open data model for representing 3D city models. It defines the combination of geometric and semantic information to enable comprehensive urban modelling and analysis. Moreover, the emergence of CityGML as an open model for 3D city object representation marks a crucial advancement in standardising the storage and exchange of urban models in multiple levels of detail.

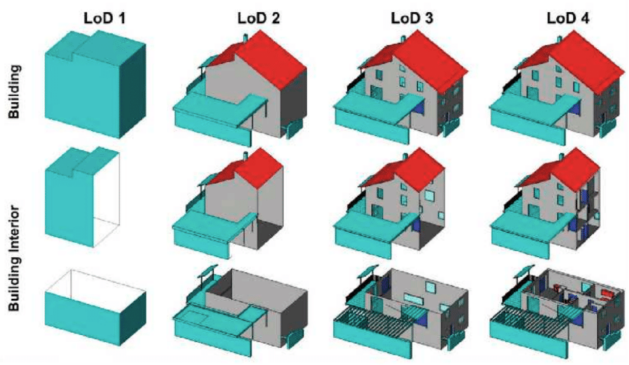


Figure 2.3.: LoD1-LoD4 represented in the CityGML standard (OGC CityGML)

CityJSON

CityGML, initially introduced by the OGC as an open data model, encountered challenges due to its XML-based format, which posed difficulties in processing and distribution. CityJSON emerged as a more streamlined alternative, utilising JSON for a simpler, lightweight representation. While CityGML offers extensive features and capabilities, CityJSON's JSON-based structure facilitates efficient dissemination online.

glTF

The Graphics Language Transmission Format (glTF) is a standard developed by the Khronos Group for efficiently representing 3D models. Its core part is a JSON file that describes the whole contents of the 3D scene [KhronosGroup, 2021]. The JSON file also contains links to the geometry and textures of the 3D objects, which are stored in dedicated binary files, as

shown in Figure 2.4. This allows the data to be stored in a compact format, resulting in smaller file sizes ideal for web-based applications. Widely adopted across the industry, glTF strikes a balance between file size and visual fidelity, making it popular for game engines, web browsers, VR, and AR platforms. Its JSON-based structure enables easy authoring and editing, facilitating seamless interchange of 3D content across different tools and workflows. glTF 2.0 is the primary tile format for 3D Tiles which is introduced in the next section.

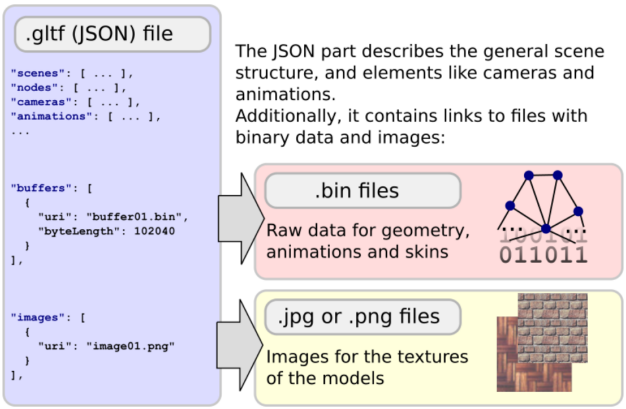


Figure 2.4.: The glTF structure

2.2. Web 3D GIS related standards and applications

The need for WebGIS (Web Geographic Information System) arises because of the necessity of modelling, representing, and simulating the dynamic world in a spatiotemporal context. There are mainly two parts that facilitate 3D WebGIS. One is standardisation introduced by international organisations like the OGC. The other is the software and hardware support visualisation of 3D formats on desktop and mobile devices.

2.2.1. WebGIS-related standards of OGC

The Open Geospatial Consortium (OGC) serves as an international standards organisation. Its specifications aim to ensure compatibility with geospatial technology. One of them is the Web Feature Service (WFS). It aims to facilitate the creation, modification, and exchange of vector-format geographic information over the Internet through HTTP protocols. A WFS encodes and transfers information in Geography Markup Language (GML).

In 2019, OGC introduced the OGC API - Feature standard, which embraces a RESTful architecture. This development represents a significant stride towards enhancing the accessibility and user-friendliness of geospatial resources on the web.

Despite years of developing and managing geographic standards that have set mature standards supporting 2D data file formats, standards that work with various 3D data formats are still on the way. The OGC web services standards provide interoperability for spatial data exchanging over the web, while limitations arise when dealing with 3D data, such as in scenarios of multi-scale 3D planning (eg: Transit-oriented development).

It is noteworthy that, in response to 3D WebGIS challenges, the OGC has introduced standards like the 3D Portrayal Service Standard 1.0 (3DPS), incorporating strategies like the Web View Service (WVS) and Web 3D Service (W3DS) for server-side and client-side rendering of 3D data.

With the increasing need for 3D web visualisation, a generic approach is needed to accessing 3D data. Continuous efforts by organisations like the Open Geospatial Consortium (OGC) are actively taken to formulate comprehensive standards for 3D data formats. 3D Tiles and Indexed 3D Scene Layers (I3S) are the latest open standards for streaming massive 3D geospatial content.

3D Tiles

3D Tiles is designed to provide efficient streaming of 3D geospatial data. Each tileset is a set of tiles that are organised hierarchically, with each tile having a corresponding bounding volume. This yields a hierarchical spatial data structure that optimises rendering and enables efficient spatial queries. When rendering, Cesium first evaluates the topological relationship between the bounding volume and the view frustum. If the bounding volume overlaps with the view frustum, then check if its geometric error falls within the predefined error limit specified in the tileset. If both conditions are met, the tile will be fetched and rendered.

3D Tiles facilitate seamless adoption and compatibility across different platforms. By offering batch 3D models, instanced 3D models, point clouds, composites, and detailed 3D Tiles style specifications, 3D Tiles presents a holistic solution for managing and visualising intricate 3D geospatial datasets.

I3S

Indexed 3D Scene Layer (I3S) is an innovative solution developed by Esri for managing vast and diverse 3D geographic datasets. A single I3S data set, known as a Scene Layer, serves as a comprehensive container capable of accommodating large amounts of heterogeneously distributed data. The delivery format and persistence model for Scene Layers, specified as Indexed 3D Scene Layer (I3S) and Scene Layer Package (SLPK), respectively, are outlined in the OGC Community Standard. Utilising JSON and binary ArrayBuffers, I3S is optimised for cloud, web, and mobile environments, leveraging modern web standards for easy handling, parsing, and rendering by web and mobile clients.

2.2.2. Visualisation of 3D city models using OGC's standard

The web visualisation of 3D City Models is made possible through the utilisation of OGC's Standard.

Web-Based 3D Routing System using WFS

Alattas et al. [2021] developed a routing system accessible through a web-based 3D Graphical User Interface (GUI). This web-based application utilises the integrated model of LADM and IndoorGML to enable indoor navigation based on user access rights within an educational building. On the server side, the system relies on a PostgreSQL/PostGIS database

with pgRouting functionality. Additionally, GeoServer is employed to implement industry-standard OGC protocols such as Web Feature Service (WFS). Apache Tomcat serves as the web server for hosting the application. This setup ensures efficient and accessible routing capabilities for users via their laptops, tablets, or mobile phone web browsers. Note that the BIM (Building Information Modelling) data is first converted into Batched 3D Model (B3DM) file and then served separately to the web server.

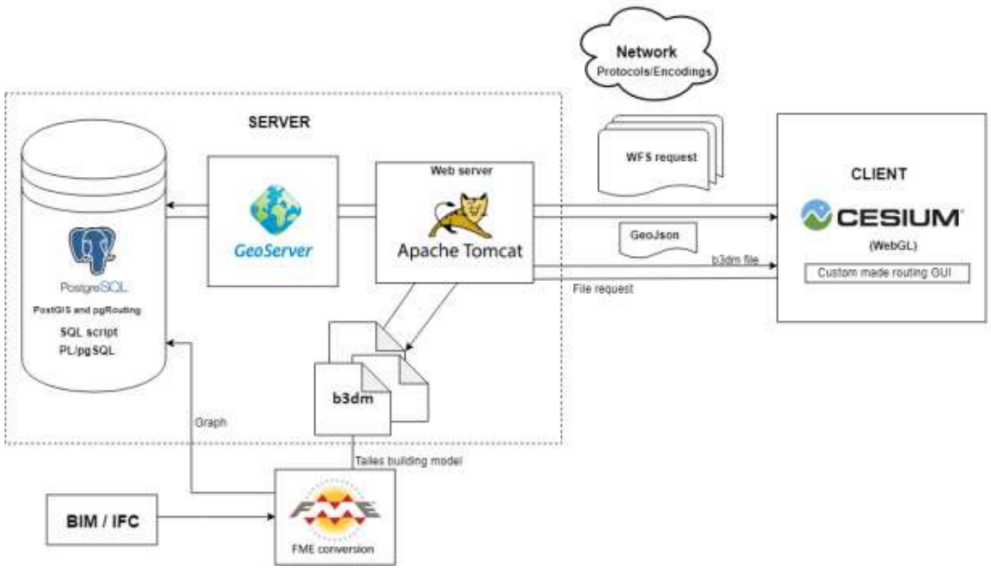


Figure 2.5.: The system architecture [Alattas et al., 2021]

Visualisation of 3D city models using 3D Portrayal Service

Koukofikis et al. [2018] explored the interoperable visualisation of 3D city models using OGC's standard 3D Portrayal Service. This experiment assessed the end-to-end process of transforming CityGML data into web-enabled visualisations using Cesium via OGC's 3D Portrayal Service. It involved converting CityGML data to 3D Tiles format, importing it into the 3D Portrayal Service Framework, and querying hierarchical 3D geometries. Additionally, an Attribute Server provided supplementary information. Through this evaluation, the experiment showcases the seamless integration enabled by OGC's 3D Portrayal Service for visualising 3D geospatial data on web platforms like Cesium. However, directly serving 3D Tiles from the database is not supported. Instead, the CityGML file is exported from the GeoRocket, a high-performance data store for geospatial files, and then converted into 3D Tiles.

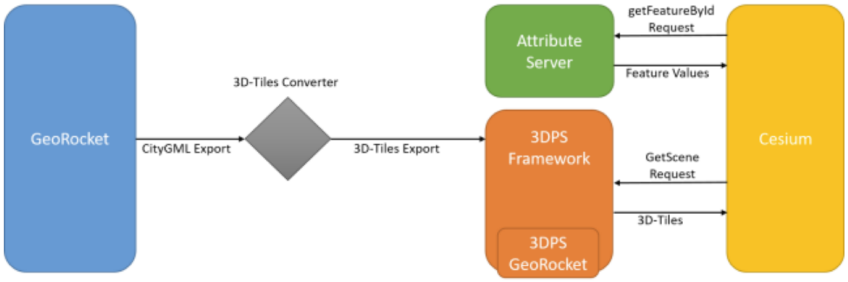


Figure 2.6.: The data flow from GeoRocket to the visualised 3D Tiles, which are requested via the 3D Portrayal Service queries [Koukofikis et al., 2018]

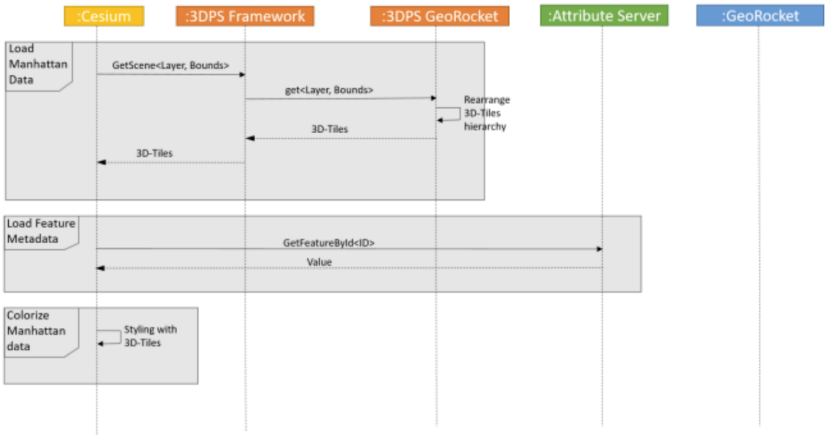


Figure 2.7.: The sequence diagram that displays data flow from GeoRocket to the visualised 3D Tiles [Koukofikis et al., 2018]

From two setups for web-based 3D visualization, it can be seen that 3D Tiles and additional features are served separately to the web application. The approaches create file representation for geometries during the process, and imply a lack of integrity maintenance. In the first scenario, b3dm (one of the 3D Tiles formats) is produced using FME, while additional features are retrieved from the database using WFS as GeoJSON format. In the second scenario, the 3D Tiles is generated using a converter and served via the 3DPS framework, while additional features are retrieved from another server.

2.2.3. WebGL-related frameworks

Today, WebGL is supported by all major desktop and mobile web browsers, and there are examples of 3D WebGIS frameworks based on WebGL.

Cesium supports the loading and display of 3D Tiles. CesiumJS is an open-source JavaScript library for creating 3D globes and 2D maps that run in browsers and across devices. The platform uses the Earth Centred, Earth Fixed coordinate system (ECEF, which is EPSG:4978), in which the centre of mass of the reference ellipsoid of Earth is taken as the origin. To render

the objects in 3D Tiles at the correct position, instead of using local coordinates, a coordinate transformation to ECEF needs to be ensured [Alattas et al., 2021].

X3DOM is an open-source JavaScript framework that enables the creation of declarative 3D scenes within web pages without the need for plugins. By integrating X3D (Extensible 3D Graphics) and DOM (Document Object Model), X3DOM allows for dynamic manipulation of 3D elements using familiar JavaScript operations, making 3D content a natural part of the web browsing experience. Three.js is another open-source JavaScript library used to create and display 3D computer graphics in web browsers.

Various commercial and proprietary solutions support geospatial applications. One is the ESRI CityEngine web viewer which is tailored for urban planning applications, facilitating the rapid prototyping and visualisation of urban environments through WebGL technology. As a market leader in geospatial software, ESRI provides CityEngine as an extension to its ArcGIS software, widely utilized in municipalities worldwide. The web viewer offers tools for plan sharing and public participation, enhancing collaboration and decision-making processes in urban planning initiatives.

Unity was originally developed as a 3D game engine for desktop PCs and game consoles. This game engine is versatile, enabling the development of augmented and virtual reality, simulations, and other applications. Unity has expanded its capabilities to support geospatial visualization through extensions like WorldComposer. This extension enables the creation of realistic 3D visualisations from geospatial data.

2.3. Database management system

An increasing number of database management systems (DBMS), such as PostgreSQL Spatial and Oracle Spatial, are incorporating the maintenance of geometry type in compliance with the OpenGIS specification [Zlatanova et al., 2004]. We focus on PostgreSQL Spatial, a free and open-source relational database management system (RDBMS) widely popular among GIS users for its powerful spatial capabilities. PostGIS and SFCGAL serve as an extension to PostgreSQL, implementing the OGC Simple Feature Specifications for SQL standards.

2.3.1. Geometrical representation in PostgreSQL Spatial

The maintenance of spatial objects typically involves three fundamental components. First, a schema within the RDBMS defines the data storage, which is often extended with specialised extensions like PostGIS. Second, a spatial indexing mechanism is employed to organize and access spatial data efficiently, enhancing the performance of spatial queries. Finally, a set of operators and functions are provided for performing spatial queries and other spatial analysis operations.

Spatial data storage: PostgreSQL spatial supports a relational model to store different types of spatial data. It supports geometric primitives such as points, lines, polygons, and also geometric aggregate (multi-geometries). Apart from working with 2-D geometries, PostGIS supports additional dimensions on all geometry types, such as a “Z” dimension to add height information. Additionally, PostGIS includes the TIN type that models triangular meshes as rows, and the POLYHEDRALSURFACE type which allows users to model volumetric objects.

Indexing: This is typically implemented using the Generalized Search Tree (GiST). Apart from the GiST index, other types like SP-GiST, BRIN, and B-tree can also be used. Spatial indexing supports quickly searching and retrieving spatial data based on its location.

Functions and operators: These functions enable users to perform a wide range of geometric operations. For example, users can filter and analyse spatial data, measure distances and areas, intersect geometries, buffering, and more. Furthermore, some of the spatial functions will automatically make use of a spatial index.

2.3.2. Topological data model

Various topological models have been proposed in the literature to address the complexity of spatial data representation.

3DFDS

Formal Data Structure (3DFDS) is a comprehensive vector model that contains geometric and semantic information and maintains 3D topology [Zlatanova et al., 2009]. This model can be implemented using an object-oriented approach, based on fundamental objects like Points, Lines, Surfaces, and Bodies, along with primitives such as Nodes, Arcs, Edges, and Faces.

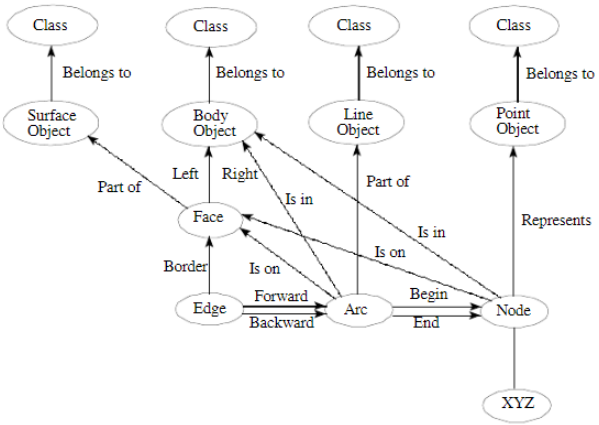


Figure 2.8.: 3D Formal Data Structure [Molenaar, 1992].

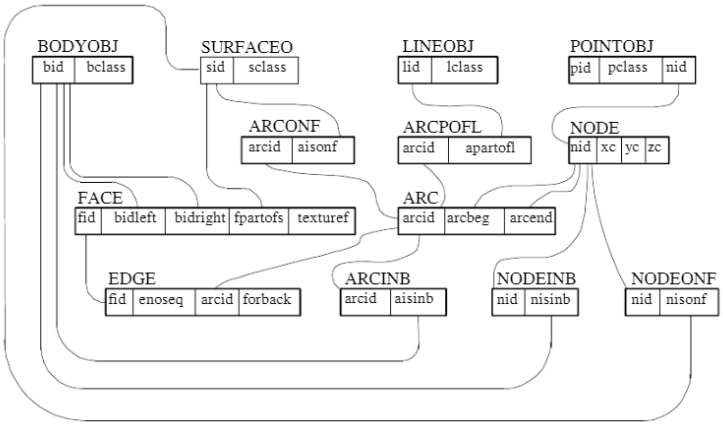


Figure 2.9.: The relational data structure of 3D FDS [Zlatanova et al., 2009]

TEN

The TEN (Tetrahedral Network), offers a different perspective by employing primitives like tetrahedrons, triangles, arcs, and nodes. However, while this method subdivides space effectively and naturally for geological applications, it creates a large volume of unnecessary data, especially when representing man-made 3D objects.

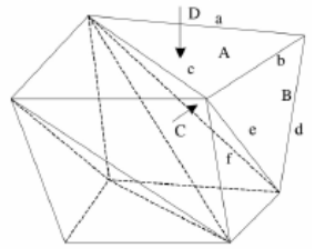


Figure 2.10.: Tetrahedral Network(TEN) [Pilouk, 1996].

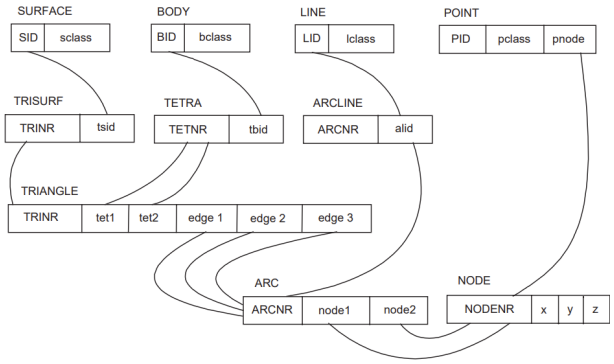


Figure 2.11.: TEtrahedral Network(TEN): relational implementation for 3D [Pilouk, 1996]. Reused from [Zlatanovaa et al., 2003]

SSM

The Simplified Spatial Model (SSM) focuses on representing geometry using planar convex faces. The SSM is designed to support web applications and focus on the visualization of queries on the screen. The model stores faces referencing to nodes, while bodies are stored associated with faces.

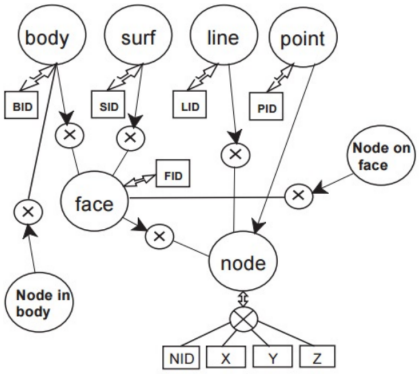


Figure 2.12.: Simplified Spatial Model (SSM) [Zlatanovaa, 2000].

UDM

In the Urban Data Model (UDM), the geometry of a surface or body is modelled through planar convex faces. These faces are linked by nodes that serve as connection points and establish connections among themselves and other faces.

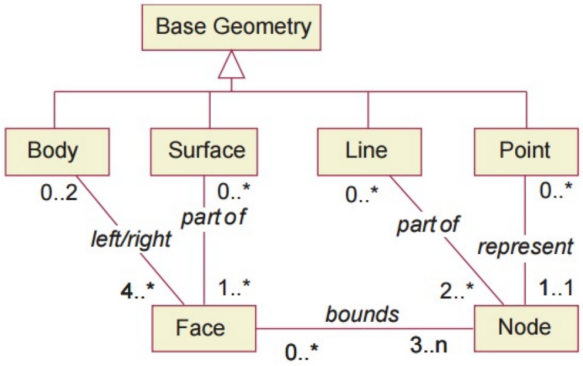


Figure 2.13.: Urban Data Model (UDM) [Coors, 2003].

	Used Primitives	Topological Tables	Explicit Relationships
3DFDS	node, arc, edge, face	arc, edge, face	node-on-face node-in-volume arc-partof-line arc-on-face arc-in-volume
TEN	node, arc, triangle, tetrahedron	arc, triangle, tetrahedron	arc-partof-line
SSS	node, face	face, line, surface, volume	node-in-volume face-in-volume
UDM	node, face	face, line, surface, volume	face-partof-surface node-partof-line

Table 2.1.: Inventory of different 3D topological structures, modified from [Van Oosterom et al., 2002]

The topological type is determined by encoding several parameters, including the dimension of the data structure, the total number of topological tables, and the specific topological rules [Van Oosterom et al., 2002]. It can be seen from Figure 2.1 that the primitives used depend on the expected type of topology, and not all primitives need to be used. For example, only the node and face are used in the SSS structure [Zlatanovaa, 2000], while all the primitives are utilised in the TEN structure [Pilouk, 1996].

2.3.3. Database for 3D city models

The 3D City Database (3DCityDB) is an Open Source software suite which allows the import, management, analysis, visualisation, and export of virtual 3D city models. This tool can automatically create database schemas for storing CityGML data for various database management systems (ORACLE Spatial or PostgreSQL). The mapping approach in 3DCityDB, for example, can utilise one table to represent multiple classes that are subtyped from a common class and at the same time belong to the same inheritance hierarchy level [Yao et al., 2018]. The 3DCityDB allows for importing and exporting of CityGML and CityqJSON

datasets. In addition, it comes with a KML/COLLADA/glTF exporter for creating tiled visualization models.

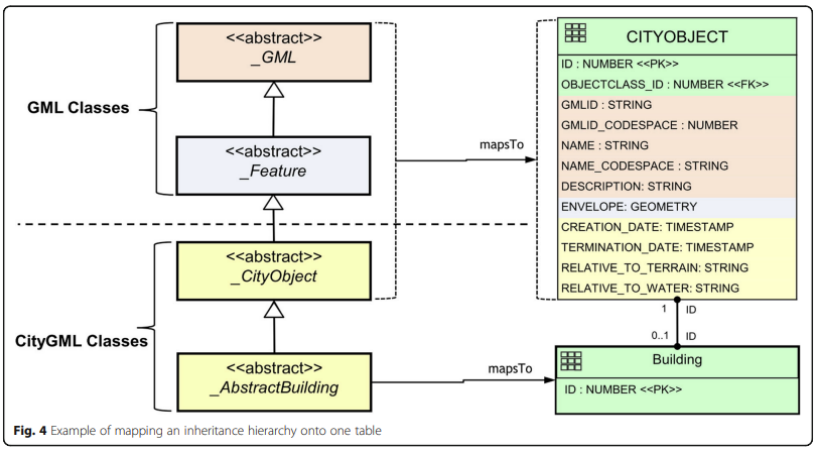


Figure 2.14.: 3D City Database (3DCityDB) example of mapping an inheritance hierarchy onto one table [Yao et al., 2018]

3DCityDB enhances web access through two services: 3DCityDB-Web-Feature-Service and 3DCityDB-Web-Map-Client. The former, compliant with OGC Web Feature Service 2.0, extends the CityGML import/export tool to enable web-based access to city objects. The latter allows 3D visualization models (such as KML/COLLADA/glTF) generated via the exporter plugin, linking with the tabular thematic data exported from the Spreadsheet Generator plugin. Note that this Web-Map-Client service first exports a file representation from the database and then serves the file to the web application.

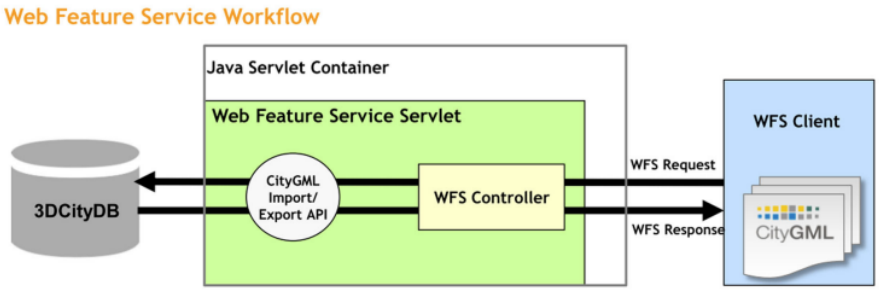


Figure 2.15.: Implementation of the 3DCityDB Web Feature Service [Yao et al., 2018]

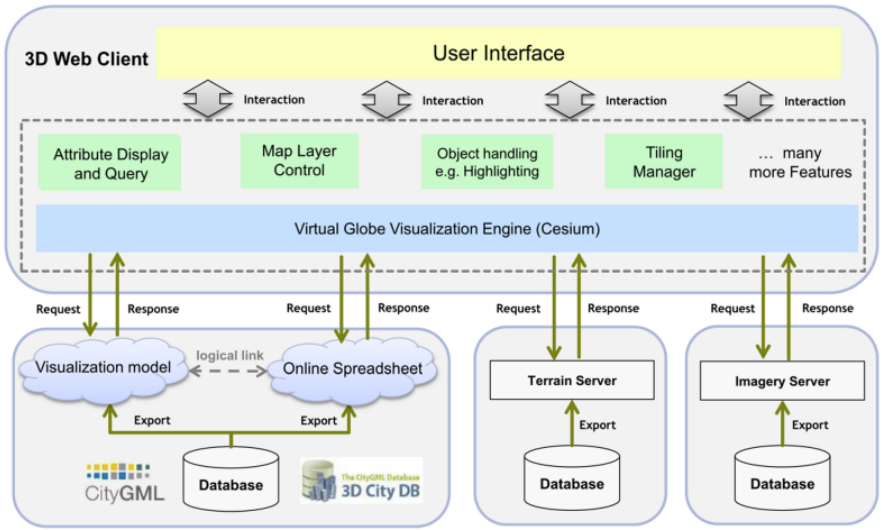


Figure 2.16.: Workflow of using 3DCityDB web client coupled with Cloud-based online spreadsheets [Yao et al., 2018]

2.3.4. Geometric operations (Triangulation)

Spatial databases such as PostGIS, Oracle Spatial, and Spatialite support geometric operations. However, performing 3D spatial operations in a database can be challenging. Existing capabilities may not adequately handle 3D geometries, so specialized tools such as Computational Geometry Algorithms Library (CGAL) or Simple Feature CGAL (SFCGAL) are required for precise operations compliant with GIS standards. Balancing accuracy and computational efficiency remains a consideration. Even so, existing database functions designed for spatial operations, such as PostGIS functions, may not support processing the current 3D geometry type.

The CGAL is an open-source library known for providing a variety of geometric algorithms and data structures covering both 3D and 2D spaces, such as triangulations, Voronoi diagrams, Boolean operations on polygons, and polyhedra. However, this library is external to the database, which supports geometric operations in file-based approaches. It should be noticed that CGAL geometric models and operations differ from those commonly used in the GIS field.

To bridge this gap, an open-source project called SFCGAL was developed, which is a C++ wrapper library that provides easy access to the powerful computational geometry algorithms library CGAL [Oslandia and IGN, 2022]. It specifically implements geometric models and API compliant with the ISO 19107 spatial schema and OGC Simple Features Access. This consistency ensures compatibility with GIS standards, enabling seamless interaction and integration with existing GIS systems.

Triangulation

Triangles are basic shapes in graphics and rendering, widely used in various application areas such as GIS, robotics, and geometric modelling. Triangulation is commonly employed by

constructing a Triangular Irregular Network (TIN) or mesh, providing a continuous surface representation.

The computational geometry triangulations aim to simplify complex polygonal shapes into a collection of triangles [Boissonnat et al., 2000]. This decomposition ensures that the resulting triangles cover the entire interior of the original polygon excluding the holes, and without intersecting or overlapping each other within the polygon. The collection of these triangles forms together the exact shape of the initial polygon.

One common method for triangulating a simple polygon is the ear-clipping algorithm. This algorithm involves a recursive process that identifies "ears" within a polygon and iteratively removes them until only one triangle remains. The concept is that any simple polygon with at least 4 vertices (without holes) has at least two ears—triangles within the polygon. Originally, the ear-clipping has an expensive time complexity of $O(n^3)$ with $O(n)$ time spent on validating newly formed triangles [Mei et al., 2012]. Some triangulation algorithms, such as the FIST Package by Held, are optimised based on Ear-Clipping. These algorithms reduce the occurrence of sliver triangles, ensuring efficient and better-shaped triangulation and avoiding requiring additional post-processing corrective steps after the initial triangulation process.

SFCGAL extensions in the spatial database also support triangulation. SFCGAL offers two-dimensional and three-dimensional built-in triangulations. The method ST_Tessellate, available from version 2.1.0 onward, takes as input surfaces such as MULTI(POLYGON) or POLYHEDRALSURFACE and returns into TIN representations from tessellation. Operating within a GIS environment, it employs the SFCGAL backend. This process supports three-dimensional attributes, preserving the z-coordinate.

2.3.5. Spatial accessing method

Spatial access methods refer to techniques used in spatial databases to organise and retrieve spatial data efficiently. It aids in quick access and rendering of specific portions of large datasets.

Spatial data sets are often large and multi-dimensional, and normal indexing has limitations when dealing with these data. Traditional indexes, such as B-trees, are not well-suited for spatial data. They rely on sorting and binary search, which are optimized for one-dimensional data. Spatial data cannot be easily sorted in a linear order that preserves spatial relationships. Thus, it requires specialised indexing techniques that can efficiently handle multi-dimensional data and spatial relationships.

Spatial indexing structures, such as R-trees, quad-trees, field-trees, and kd-trees, are commonly used to organize spatial data. These structures partition the space into smaller regions and associate spatial objects with these regions to facilitate fast search and retrieval.

R-tree is a hierarchical data structure used to efficiently index and query spatial data represented as Minimum Bounding Rectangles (MBRs). In a classical R-tree, each tuple in the tree has a unique identifier and a Minimum Bounding Rectangle (MBR) that encloses the spatial object it represents [Zlatanova and Gruber, 2001]. It organizes these rectangles in a tree structure, enabling quick spatial searches by traversing only the relevant nodes of the tree.

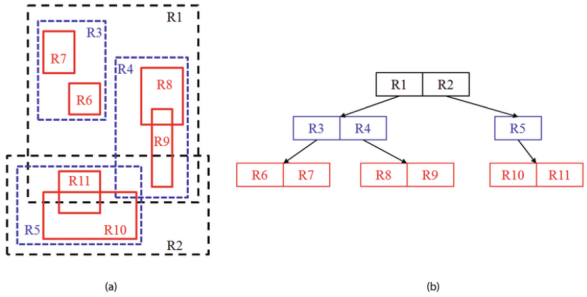


Figure 2.17.: R-Tree indexing example: 2D visualization (a), hierarchical dependencies (b) [Broilo et al., 2010]

Fast Access in spatial queries is significant. Spatial queries, such as finding nearby points or objects within a given region, require fast access to the relevant data. A spatial index helps in quickly identifying the subset of data that satisfies the query conditions.

There are clustering methods that store objects near each other in space. Spatial clustering is based on the spatial filling curve, which reconstructs cells in kD into 1D using bijective mapping. The Morton curve (also known as the Z-order curve) and the Hilbert curve are two well-known examples of space-filling curves (SFCs). For example, the R-tree can be adapted using the Hilbert curve and be mapped from N-dimensional to 1D. Additionally, clustering methods like K-means and DBSCAN are utilised for spatial subdivision or clustering.

The choice of the indexing and clustering method is related to the specific characteristics of the data in the research.

2.4. 3D Tiles

3D Tiles is widely adopted within the geoinformation and web communities. It offers scalability and interoperability, making it ideal for delivering 3D visualisation. Therefore, we choose the 3D Tiles format for our research. The b3dm and tileset.json collaboratively form 3D Tiles.

2.4.1. Elements composite of 3D Tiles

3D Tiles serve as an innovative data format used for efficiently streaming and displaying vast amounts of three-dimensional geospatial data. They enable the seamless transmission of complex 3D models, textures, and attributes, optimizing performance for visualization on various platforms. 3D Tiles supports the Hierarchical Level of Detail (HLOD), as shown in Figure 2.18. This format organizes data hierarchically, allowing for progressive loading and rendering, which means users can view details gradually as needed.

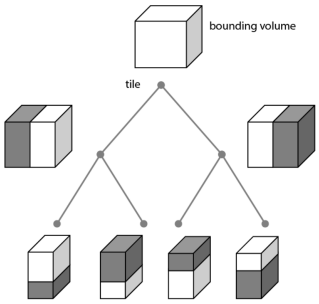


Figure 2.18.: A sample 3D Tiles bounding volume hierarchy [Cesium and OGC, 2019]

Tileset - tile structure

3D Tiles organises data on different levels, and this structure is stored in the tileset. The tileset structure commonly includes a root tile and its associated child tiles, storing renderable content within each tile, three child tiles in the example, as shown in Figure 2.19. The renderable part is stored in content. The same content with a low level of detail and with a higher level of detail are stored on different levels. The content with a higher level of detail is of smaller geometric error. In addition, each tile may refer to an external tileset, and this can combine small tilesets into a larger one.

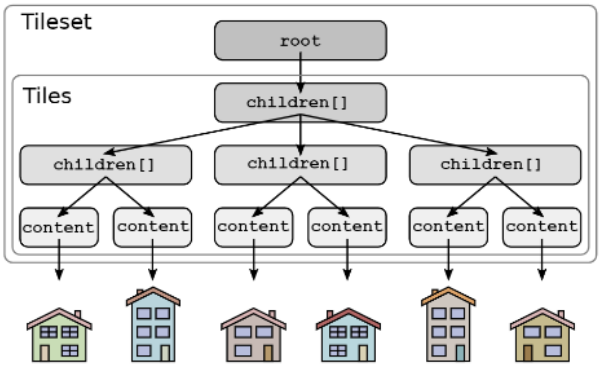


Figure 2.19.: Tile structure [CesiumGS, 2021]

In this organisation, the term "children" refers to the multiple contents under this node, and this recursive structure allows for the potential of multiple contents under each child node. Theoretically, the creation of an extensive tree hierarchy is feasible. However, it is advisable to limit the height of this tree due to practical considerations.

The content in the tileset refers to an external URL or URI, which can be a B3DM(Batched 3D Model), PNTS(PointCloudTileset), I3DM(Instanced 3D Model), CMPT(Composite), or GLB(Binary representation of glTF). It can also point to an external tileset (the uri of another tileset JSON file), enabling storing each city in a tileset and then having a global tileset of tilesets.

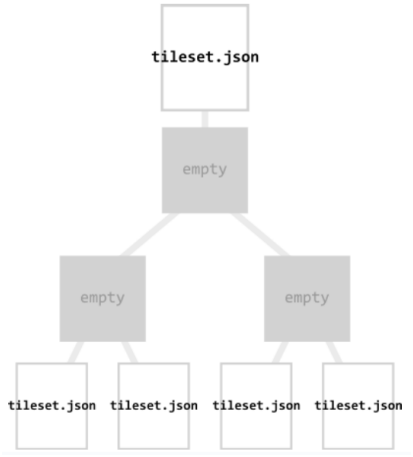


Figure 2.20.: A tileset that refers to other tilesets [Cesium and OGC, 2019]

Geometric Error and Screen Space Error

The Geometric Error (GE) represents the degree of approximation or deviation of the tile's geometry from the actual scene geometry, aiding in determining which tile should be rendered. For a tileset, i.e. the geometric error on the top level, it determines whether the root in the tileset should be rendered based on Screen Space Error (SSE). For a tile, it is used to check if its children tile should be rendered. Note that geometric error points to the next level below. root.children can be considered as sub-tiles. Each Tile can also have its children, thus forming a recursive tree structure. Generally, the children tiles' geometric error is smaller than its parent tile's.

Screen space error (SSE) is crucial to manage the level of detail based on the perceived error in geometry simplification. Each tile has a geometric error property which compares the simplified geometry to the real geometry, in metres. At runtime, the geometric error is converted into SSE. This is used for determining when to load a tile's children. In general, a higher geometric error means it will load its children sooner when the camera is zooming in.

The calculation of geometric error in 3D Tiles-based rendering systems involves various factors. The provided formula indicates one possible way to compute Screen Space Error (SSE) using parameters like screen height, tile distance, and field of view angle, determining an exact standard formula for SSE is not universally defined:

$$\text{sse} = (\text{ geometricError} * \text{screenHeight}) / (\text{ tileDistance} * 2 * \tan(\text{ fovy } / 2))$$

screenHeight : the height of the rendering screen in pixels
tileDistance : the distance of the tile from the eye point
fovy : the opening angle of the view frustum in the y-direction

The exact implementation is experience-based and tailored to specific applications. For instance, when constructing an octree for managing levels of detail, a typical approach is

to designate a geometric error for the root node and subsequently halve this error for each descending level in the tree.

Bounding volume and view frustum

A Bounding Volume is a simplified volume, like a bounding box, a bounding sphere that approximates the spatial extent of an object or group of objects in a scene. A view frustum is defined by the camera position, orientation, and field-of-view angle.

When rendering, the bounding volume is first read, and based on the bounding volume and the current view frustum, it is determined whether the bounding volumes of tilesets and tiles intersect with the view frustum.

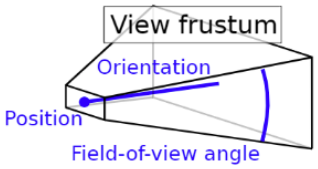


Figure 2.21.: View frustum [CesiumGS, 2021]

Content in 3D Tiles

The 3D Tiles specification covers a variety of file formats, including Batched 3D Model(B3DM), PointCloudTileset(PNTS), Instanced 3D Model(I3DM), Composite(CMPT), and binary representation of glTF(GLB). It is designed to efficiently manage and transport a variety of 3D geospatial data types. These formats play a key role in representing different aspects of 3D models, point clouds, instances, and composite scenes in the 3D Tiles ecosystem.

3D Tiles specification version 1.1 is backwards compatible with 1.0, and the basic structure of the file format remains consistent. The standardized structure of B3DM, PNTS, I3DM, CMPT, and GLB ensures interoperability and seamless data transfer, regardless of which specification version is used. This cross-compatibility supports iterative work on projects based on earlier versions of 3D Tiles.

B3DM is specifically studied. It is a binary blob (Binary Large Object) in little-endian. The Information in the b3dm is shown in Figure 2.22. It contains the header and the body part. The body comprises the Feature Table, the Batch Table, and the binary glTF. The encoding of a b3dm body will be explained in Section 3.4.3.

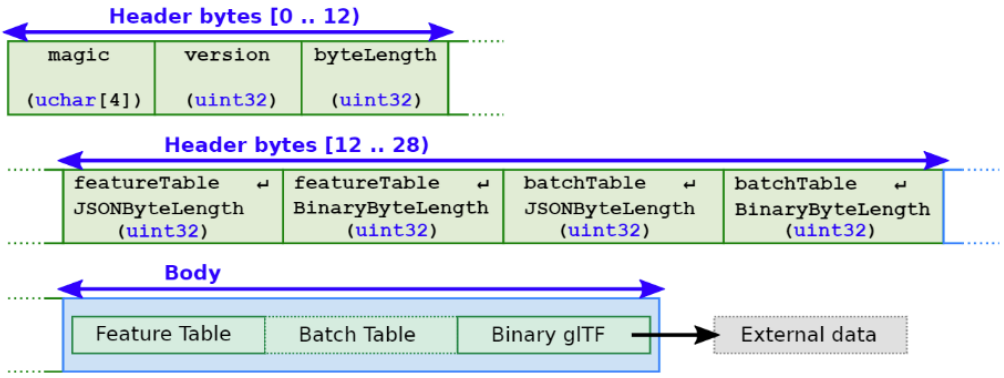


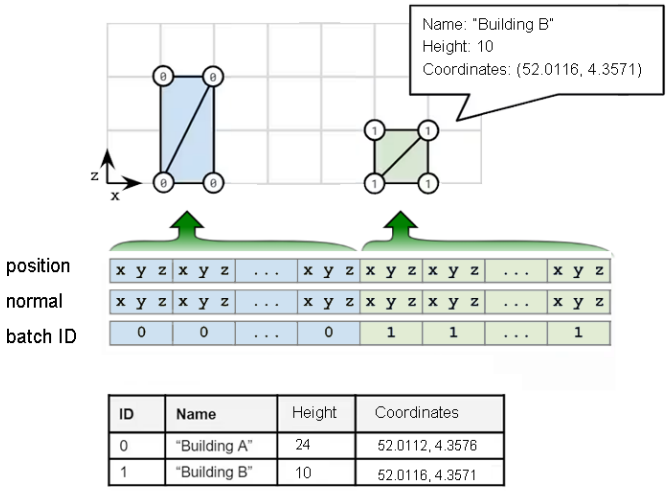
Figure 2.22.: Layout of a b3dm [CesiumGS, 2021]

The fields in a b3dm header are summarised as follows:

- The **magic** value is used to identify the content as a b3dm file. It is a 4-byte ANSI string.
- The **version** identifies the total length of the file.
- The **byteLength** identifies the version of the Batched 3D Model format.
- The **featureTableJSONByteLength** and **featureTableBinaryByteLength** identify the JSON section and binary section length of the Feature Table in bytes, with data type uint32.
- The **batchTableJSONByteLength** and **batchTableBinaryByteLength** identify the JSON section and binary section length of the Feature Table, with data type uint32.

A Feature Table contains position and appearance properties required to render each feature in a tile, and the Batch Table describes additional application-specific properties for each feature.

In a b3dm, batch ID identifies the same model. It can determine the selected model with the same batch ID at run time. The batch ID links the geometric data in a binary glTF with properties in a batch table, as shown in Figure 2.23. This allows the model's appearance to change dynamically based on its property, for example, styling buildings according to height values.



ID	Name	Height	Coordinates
0	"Building A"	24	52.0112, 4.3576
1	"Building B"	10	52.0116, 4.3571

Figure 2.23.: batch ID linking geometry and property, modified from [CesiumGS, 2021]

2.4.2. Coordinates system

There are two main aspects related to the coordinate system in the thesis. One is orientation associated with using z-up or y-up, and the other is the coordinate transformation among different coordinate reference systems.

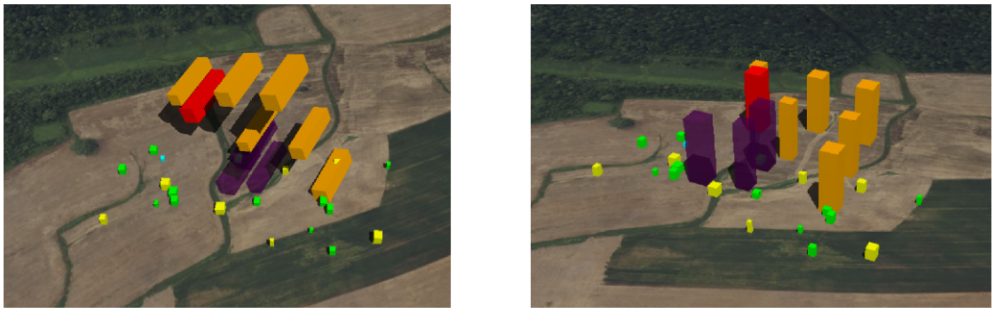
Choices between z-up or y-up coordinate system

glTF is a file format designed for efficient transmission of 3D scenes and models. By default, glTF files use a right-handed coordinate system, where the y-axis is up. However, 3DTiles uses a coordinate system with z-up. Both the z-up and y-up follow the right-hand rule.

To keep the consistency, a transformation by rotating around the x-axis by /2 radians is done at runtime. This is equivalent to matrix transform:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Additionally, for high-precision rendering purposes, vertex positions can be defined relative to a centre position specified by RTC_CENTER. This centre position serves as the reference point to which all vertex positions are relative after the coordinate system and glTF node hierarchy transformations have been applied, aiding in maintaining precision during rendering processes. In 3D Tiles 1.0, this transformation that is used to translate model vertices is in featureTable. In 3DTile 1.1, the RTC_CENTER can be added to the translation component of the root node of the glTF asset.



(a) before y-up to z-up transformation (b) after y-up to z-up transformation

Figure 2.24.: right-handed coordinate system

Coordinate reference system

Cesium is a JavaScript library that supports visualization and analysis on a high-precision WGS84 glob and is shared with users on desktop or mobile. It streams 3D Tiles and other standard formats from Cesium ion or another source. Cesium ion is a SaaS platform that offers a seamless solution for optimizing, hosting, and streaming 3D geospatial data. When the user uploads the content, Cesium ion will optimize it as 3D Tiles, host it in the cloud, and stream it to any device.

As CesiumJS is a runtime engine, it deals with data that has been transformed into specific coordinate systems. It is essential to understand the coordinate reference systems of the geospatial datasets and handle transformations between different CRS to ensure accurate geospatial representations.

EPSG codes define the coordinate system of the dataset. In the context of Cesium, a tileset's global coordinate system is in a WGS 84 earth-centred, earth-fixed (ECEF) reference frame (EPSG 4978), coordinates as XYZ. The region bounding volume specifies bounds using a geographic coordinate system (latitude, longitude, height), specifically EPSG 4979. A Cartesian in CesiumJS is in EPSG:4978. While it is in EPSG:4979 if it is a Cartographic in CesiumJS.

```
new Cesium.Cartographic(longitude, latitude, height)
```

A position defined by longitude, latitude, and height.

Name	Type	Default	Description
longitude	number	0.0	optional The longitude, in radians.
latitude	number	0.0	optional The latitude, in radians.
height	number	0.0	optional The height, in meters, above the ellipsoid.

Figure 2.25.: A position defined by lat, long, and height as shown in Cesium.Cartographic

To be specific, EPSG 4978 represents the ECEF (Earth-Centered, Earth-Fixed) CRS, which is a 3D Cartesian coordinate system with its origin at the centre of the Earth. This system is used for various applications involving satellite tracking, navigation, and geodesy. EPSG 4979 is similar to EPSG 4978. However, it uses three-dimensional Cartesian coordinates that include ellipsoidal height as the third dimension, representing the height above the

reference ellipsoid. Additionally, EPSG4326 represents a geographical coordinate system that uses longitude and latitude to represent points on the Earth. This is a two-dimensional coordinate system. EPSG:4326 uses the WGS 84 (World Geodetic System 1984) ellipsoid as its reference ellipsoid, hence the alias "WGS 84 coordinate system".

When displaying multi-source geospatial data on Cesium, conversion from source CRS to target CRS is required to ensure a correct and unified coordinate system

2.4.3. 3D Tiles indexing

3D Tiles indexing plays a crucial role in optimizing the delivery and rendering of 3D Tiles or any spatial data for applications like CesiumJS, especially in scenarios where there are bandwidth limitations or slow internet connections. This reduces the amount of data that needs to be transmitted over the network.

glTF indexing designed for web transmission

Cesium is designed based on WebGL which is the core to render the objects, and glTF is a model that complies with WebGL well. The design of glTF, to be specific, the internal structure of glTF, mimics the memory buffers. This is commonly used by graphics chips for real-time rendering. Indices themselves take space, and finding the positions that indices refer to also costs time. However, the most important idea of indices in glTF is not about saving space or 'geometric' efficiency, it is more about the renderer mechanism. glb data is mesh, and there are two ways to deliver the mesh data to GPU, DrawArrays, and DrawElements. The latter needs an index buffer to which indices in a glb are related. This reduces memory usage and improves GPU efficiency, prompting assets to be delivered to web or mobile clients and displayed quickly with minimal processing [KhronosGroup, 2021].

In computer graphics, triangle strips are used to optimize memory usage and improve rendering performance. This allows encoding a mesh of n faces with N+2 vertices. Modern graphics interfaces provide primitive types of triangle strips to relieve the bottlenecks and save graphics memory [Treumer et al., 2023]. Instead of specifying each triangle individually, triangle strips allow consecutive triangles to share vertices, reducing redundancy in vertex data and optimizing memory usage.

A hierarchical schema

Massive 3D model data is loaded by loading 3D Tiles, which speeds up the loading and rendering. The most important concept is hierarchy.

The 3D Tiles model is a multi-resolution hierarchical structure. From the bottom to the top of the pyramid, the resolution, i.e. level of detail, is getting lower and lower, while the range of representation is fixed. Building a hierarchy on varying accuracy levels enables clients to visualize large-scale 3D data by combining three-dimensional data with different precision on hierarchical levels and dynamically loading data in real-time, which reduces memory consumption [Renxin et al., 2019].

The 3D Tiles technology supports different indexing mechanisms to build tile hierarchy. It usually uses a multidimensional indexing method to build the hierarchy that forms multiple resolution levels. The 3D Tiles technology mainly adopts the index mechanism such as

octree structure, and quadtree structure. These hierarchical spatial partitioning structures represent regions in 2D and 3D space, by recursively subdividing space into quadrants or octants. In 3D Tiles 1.1, quadtree and Octree can be mapped into 1D, where the tile in a tileset can be called randomly regardless of the level depth[CesiumGS, 2022], as shown in Figure 2.26.

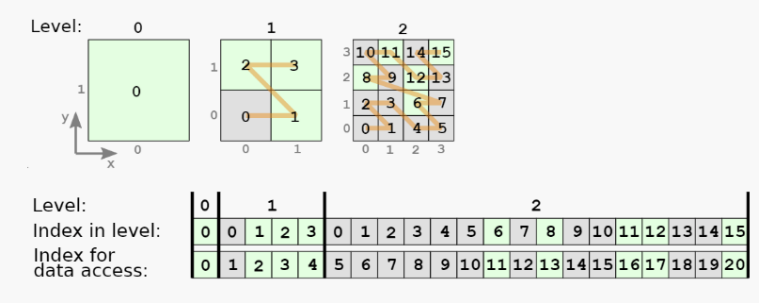


Figure 2.26.: SFC Z-order encoding quadtree scheme [CesiumGS, 2022]

3. 3D Tiles Approach

This chapter details the proposed methodology used within this thesis project. Section 3.1 gives the motivation for managing and serving 3D Tiles from the database, compared to the typical file-based approach. The database storage model is described in Section 3.2. Section 3.3 describes the steps to prepare the model for further approach development. Section 3.4 explains the approach and possible alternatives that can be followed to prepare 3D Tiles in the database. Section 3.5 explains the methodology used for querying and visualising the 3D Tiles.

3.1. Motivation

The main motivation for using a DBMS approach to serve 3D Tiles directly from the database stems from limitations present in current file-based solutions. The typical file-based approach unavoidably creates file representations (data copies). More specifically, the methods presented in Section 2.2.2. The typical approach takes up valuable disk space and is burdensome to manage. This also raises concerns about data inconsistency. Besides, it is not flexible in querying and fetching certain regions. In many applications, loading the data on the web client takes a long time as the data size and coverage area increase. This suggests that an efficient query and data visualisation requires a different approach than file-based ones.

3.1.1. Requirements

The 3D Tiles approach is expected to fulfil the following requirements:

- **3D Tiles storage:** It should support organising the object-oriented tileset structure and 3D geometry into a relational database. Additionally, it provides an option to temporarily store tile data in the local database, and ready to be delivered to the web application.
- **Geometric computation and attribute enriching:** It is expected to compute mandatory information from the raw data for generating b3dm format, including normal computation, triangulation, and calculation of the total spatial extent of objects in the same tile. The database also utilises spatial functions and operators, processing queries on geometries to enrich attributes.
- **Spatial clustering:** It is expected to cluster objects into groups on multiple levels, with each level representing distinct scales or granularities. This hierarchical structure facilitates faster spatial search and retrieval by encompassing broader spatial extents at higher levels. Moreover, it enables the potential implementation of a multi-level representation of data across various levels of detail.

- **Tile organisation:** It should support custom tiling objects based on a hierarchical index structure. The spatial extent for each tile is determined based on the groups defined on the same level (for example: the bottom level), guaranteeing meaningful spatial areas within each tile.

- **Web connection and streaming:** It should support connections to a web server that efficiently sends the 3D Tiles to the web client based on user demand.

Why serve a single LOD of a b3dm from PostgreSQL Spatial?

- Multiple source data can be served as 3D Tiles, such as point clouds, BIM, and 3D buildings (KML/COLLADA, Wavefront OBJ, CityGML, CityJSON). We focus on serving volumetric objects with 3D Tiles due to their rich geometric and semantic information and their central role in the management of the built environment.

- The methodology and implementation are based on the PostgreSQL geospatial database. The reason is that it is one of the most widely used open-source database management systems.

- The b3dm (Batched 3D Model) format is one of the tile formats specified within the 3D Tiles specification 1.0. In 3D Tiles, ”batched” refers to grouping multiple 3D models into larger batches. The b3dm file format stores a batch table containing property (attribute) information for each model, identified by unique batch IDs. For example, the batch table contains a series of height values of building model data, enabling the visualisation of buildings with varying heights. The b3dm is used for storing and transmitting 3D models in a batched, binary format that can be easily streamed and rendered on web clients.

- By commencing the prototype with the initial version of 3D Tiles, version 1.0, it establishes a robust starting point that ensures compatibility with existing tools and workflows. As the new 3D Tiles specification evolves, it is feasible to enable iterative improvements while maintaining compatibility with the initial version.

- To prioritize simplicity and streamline the development process, a decision has been made to concentrate on a single Level of Detail (LOD). By focusing on a single LOD, complexity is reduced. This aims at directly serving 3D Tiles from the database.

3.1.2. Approach and variations

The DBMS approach is proposed as opposed to the typical file-based approach. The overview of the proposed approach and the variations is shown in Figure 3.1.

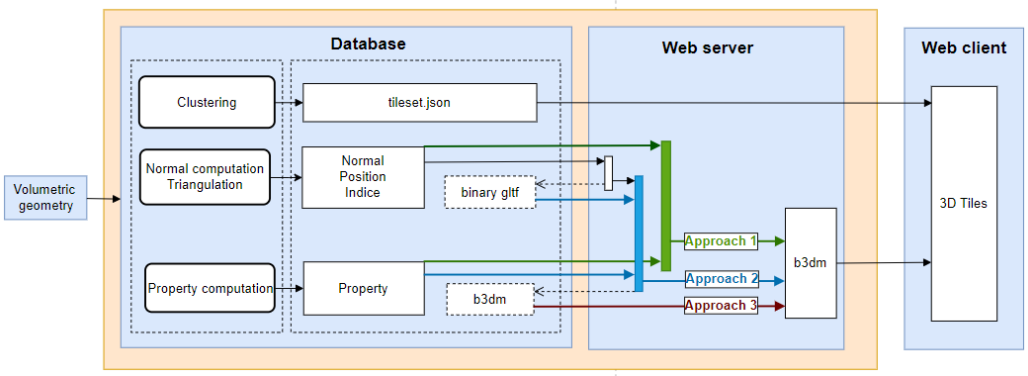


Figure 3.1.: The overview of the 3D Tiles approach (Simplified)

Approach 1

The first approach, the on-the-fly approach, encodes the complete b3dm in the web server. This is achieved by generating normal, position and triangle indices in the database, serving this information from the database and then composing the b3dm in the web server.

The on-the-fly serving approach is expected to be a flexible method to meet user needs for region and property selection.

Approach 2

The second approach, the precomposed geometry approach, assembles the geometric information in the database. After generating normal, position and triangle indices in the database, the geometric information is encoded and stored as a binary glTF in the database. The binary glTF and properties are served from the database, and the properties are combined with glTF in the server to compose a b3dm.

Regarding serving 3D Tiles containing different properties, this approach is expected to provide efficient response times. The reason is that the geometric information in the b3dm is encoded as a binary glTF in advance.

Approach 3

The third approach, the precomposed b3dm approach, completes the creation of b3dm beforehand and stores it in the database. The b3dm is then queried and directly sent to the web server from the database.

This approach should provide an efficient response time as the b3dm encoding computation is not needed in the web server.

3.2. Storage model for database

This section describes the conceptual schema for mapping an object-oriented 3D Tiles structure within a relational database. Conceptual Schema is summarised as follows:

Object: 3D coordinates of each object body are stored as an array of coordinates (nodes). References to faces at the lower level are maintained through object IDs.

Face: Details about triangulated faces are maintained on this level. Indices of the triangulated faces are stored as an array of indices. Faces are expected to keep a counterclockwise orientation when viewed from the outside of a three-dimensional enclosure.

Property: Properties are linked to the corresponding object.

Hierarchy: This table supports efficient queries based on minimum bounding regions on hierarchical levels. Objects clustered into the same cluster are considered to be in a tile.

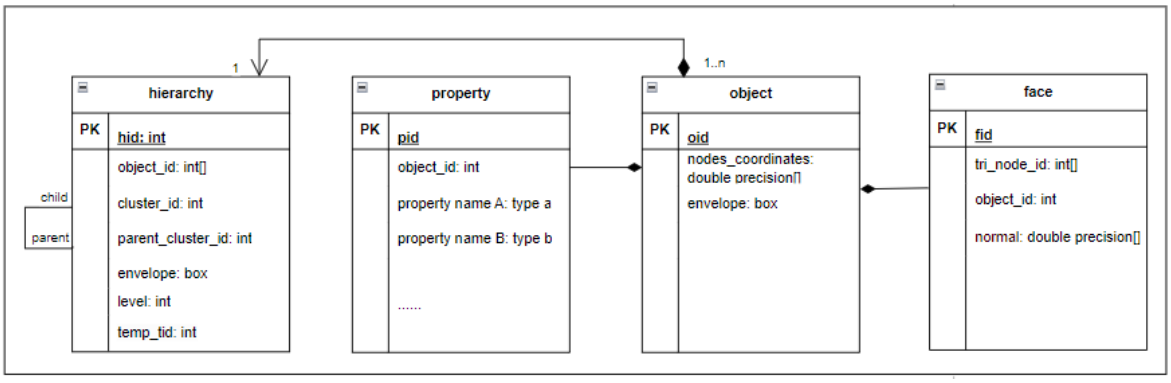


Figure 3.2.: An overview of the data storage model

3.2.1. 3D data storage model

The geometry data in the b3dm format is contained in a binary glTF format that represents the mesh, and the initial problem revolves around developing a compact storage method for the mesh in the database.

Generally, there are two aspects regarding storing 3D data in the database, one is maintaining the data as geometry, and the other is maintaining the coordinates and its topological relationship. By translating topological structures into geometric primitives, it becomes feasible to establish a DBMS view on a topological primitive, thus presenting it as a geometric primitive [Van Oosterom et al., 2002]. This combines the advantages of topological structure for less redundancy and the benefits of explicit geometric primitives.

Data model examples in Oracle Spatial

Proposed by Zlatanova et al. [2004], there are two approaches to storing 3D geometry in Oracle Spatial. One is storing one simple geometry (i.e. as a set of 3D polygons), and the other is storing one geometry as a collection of simple geometry types (i.e. 3D collection and 3D multipolygon).

The first approach encodes 3D objects using multiple rows in the geometry table, linking a set of polygons to specific 3D objects. This representation is a bit inefficient, but it manages the topology by storing relationships between faces and the object. Topological queries can be completed by comparing IDs of polygons that reference the same object.

TIN Z (((0 0 0,0 0 1,0 1 1,0 0 0)),((0 1 0,0 0 0,0 1 1,0 1 0)), ((0 0 0,0 1 0,1 1 0,0 0 0)),((1 0 0,0 0 0,1 1 0,1 0 0)), ((0 0 1,1 0 0,1 0 1,0 0 1)),((0 0 1,0 0 0,1 0 0,0 0 1)), ((1 1 0,1 1 1,1 0 1,1 1 0)),((1 0 0,1 1 0,1 0 1,1 0 0)), ((0 1 0,0 1 1,1 1 1,0 1 0)),((1 1 0,0 1 0,1 1 1,1 1 0)), ((0 1 1,1 0 1,1 1 1,0 1 1)),((0 1 1,0 0 1,1 0 1,0 1 1)))

Table 3.1.: TIN representation of a unit cube in PostgreSQL Spatial

In the second approach, the 3D object is described in a single row, since all the information about the polygons is decoded in the Oracle Spatial geometry type. This approach reduces the number of records to represent a geometry compared to the first approach. However, redundant coordinates are stored because each triple of coordinates (x,y,z) is repeated at least three times in the list of coordinates.

Both representations have their trade-offs. The first approach sacrifices some efficiency due to multiple rows per 3D object but maintains topology. The second approach reduces the number of records but suffers from coordinate redundancy.

Additionally, it is noteworthy that Relational Database Management System (RDBMS) lacks inherent navigational capabilities within the system due to its adherence to the relational principle. As a result, external procedural languages or iterators are required to navigate the database. In contrast, Object-Oriented Database Management Systems (OO-DBMS) can navigate within the system, inherent to the object-oriented paradigm.

3D data storage model design - A balance between geometry and topology

Based on the second data model design proposed by Zlatanova et al. [2004], we propose an approach which finds a balance between geometrical structures that provide explicit object representation and topological structures which help reduce redundancy.

It is efficient to store normal information for b3dm files based on face level because triangles on the same planar surface share the same normal.

Regarding how to store the triangles, the first option is storing the geometry as TIN. This type allows to model triangular meshes as one row in the database. However, triangulating duplicate coordinates in waste storage increases computational overhead without adding useful geometric information.

An alternative approach is to establish the relationship on topological primitives. First, the primitive point(node) is used to store coordinates in 3 dimensions. Next, it's crucial to ensure support for the representation of complex features in the database. The topological structure is stored as node-face-body. To directly reconstruct the boundary representation, both the topologies of the original non-triangulated geometry and the triangulated mesh are stored.

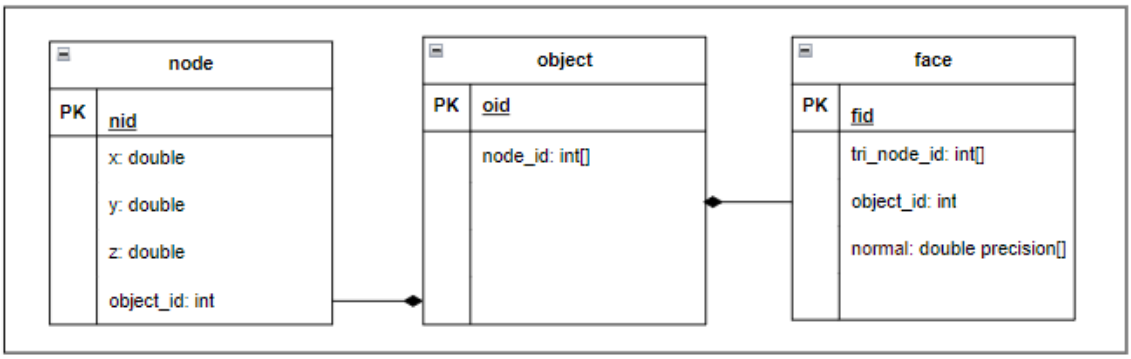


Figure 3.3.: UML diagram for Node-Face-Object approach

However, it needs to access the table nodes and faces to reconstruct the mesh from the database. Scanning through rows of records in the node table can be time-consuming. This indicates implementing indexing on key columns in the node table, or processing all coordinates of the same object in a batch. Moreover, non-triangulated geometry is not frequently queried in the 3D Tiles database, and it can be derived from the triangulated geometry. Thus, node_id in the table object is discarded.

The third, and is also the approach adopted in the project, is to store the coordinates for each object, and the topology of the triangulated objects. This is a bit object-oriented, but it helps save the time to traverse. The physical objects are represented by abstractions (object body topology) and simple elements (nodes). Coordinates of a non-triangulated object are stored as nodes in the table object, and indices of each triangulated face are stored as tri_node_id in the table face.

- Node coordinates in the table object are stored as an array of double arrays, i.e. [[x1,y1,z1],[x2,y2,z2],[x3,y3,z3]]. According to the test, whether double precision[] or double precision[][] is declared in PostgreSQL, the field in the result table is of double precision[] type.

- The tri_node_id is stored as an array of int. These int numbers are indices of the triangle vertices. Vertices are shared by multiple faces in the same object. Thus the indices point to the coordinates of an object that are stored in the nodes_coordinates column in the object table.

- The two columns are linked with the same identifier, the ID number of the object. It is the primary key of the object table and is referenced by a foreign key in the face table.

- Normals are stored as an array of doubles. Normals are shared by triangles on the same face. This helps save storage space.

By establishing object and face tables and their relationships, we store the geometric information that composes a b3dm. They are coordinates of an object, normals, and triangulated topology of faces.

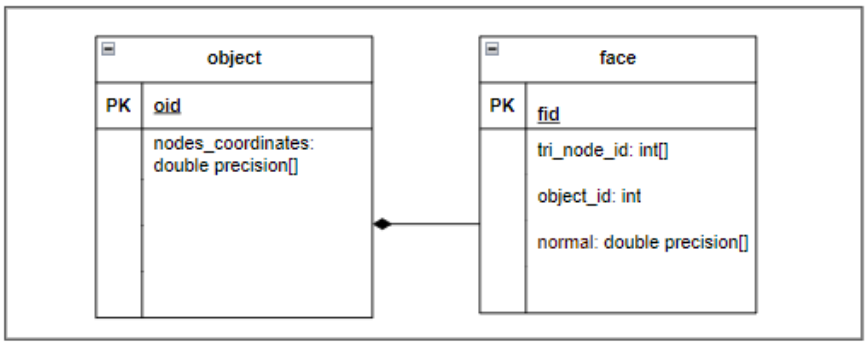


Figure 3.4.: UML diagram for Face-Object approach

An L-shaped polyhedron is provided as an example, and depicted in Figure 3.5. In the face-object approach, the list of 12 unique vertices of the polyhedron object is stored in the object as one row. Its mesh topology, namely the indices, is stored in the face as 8 rows represent 8 faces.

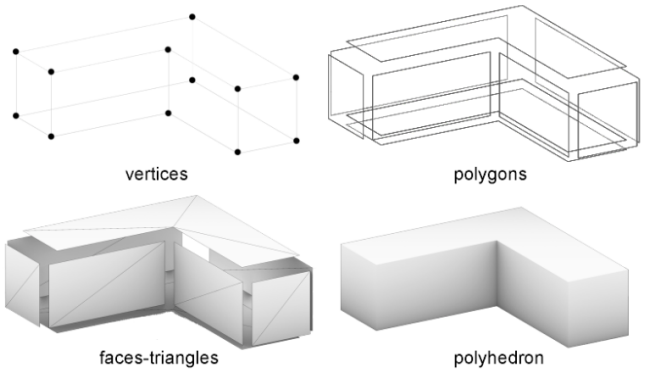


Figure 3.5.: Representation of an L-shaped polyhedron

3.2.2. Hierarchy storage database model

To facilitate spatial queries, hierarchical clusters that are used for speeding up spatial queries are explicitly stored in the database. This involves clustering spatial data at different hierarchical levels and mapping each level into the database. Additionally, object-oriented 3D Tiles structures need to be represented in a relational database.

Hierarchical data in relational databases

Hierarchical relationships are typically defined through self-referencing foreign keys, linking each entity (row) to its corresponding higher-level entity. Figure 3.6(a) shows examples of a rooted tree, which is a directed graph.

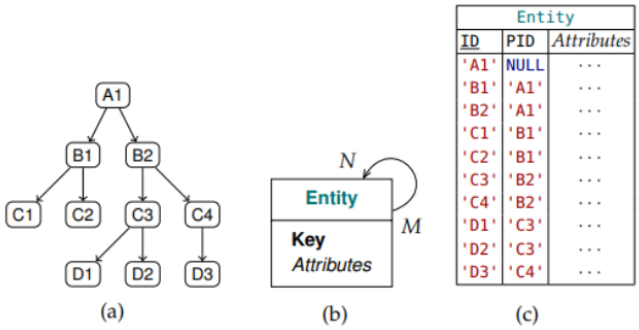
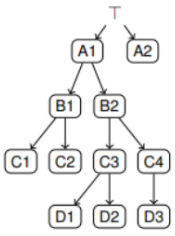


Figure 3.6.: An example of a hierarchy, and how it may appear in entity/relationship models (b) and relational database tables (c) [Brunel, 2017]

To better describe the hierarchy storage database design, the terminology for Trees is listed:

> Ancestor-descendant relationships are established through unique paths. If the path from T to node v passes through node u, node u is called an ancestor of v, and v is called a descendant of u.
> A node's parent is its directly connected ancestor. Each node, except the root, has a single parent. Children are nodes directly connected to a parent. A node that has no children is called a leaf.
> The depth of a node v is the number of nodes on the path from T to v.
> The height of the subtree rooted at the node is the depth of the deepest node in the node's subtree.
> Size: the number of nodes in the subtree rooted at the node.
> Degree: the number of children of the node.



Node	IS_LEAF	IS_ROOT	DEPTH	SIZE	DEGREE	HEIGHT
A1	FALSE	TRUE	1	10	2	4
B1	FALSE	FALSE	2	3	2	2
C1	TRUE	FALSE	3	1	0	1
C2	TRUE	FALSE	3	1	0	1
B2	FALSE	FALSE	2	6	2	3
C3	FALSE	FALSE	3	3	2	2
D1	TRUE	FALSE	4	1	0	1
D2	TRUE	FALSE	4	1	0	1
C4	FALSE	FALSE	3	2	1	2
D3	TRUE	FALSE	4	1	0	1
A2	TRUE	TRUE	1	1	0	1

Figure 3.7.: Projecting basic properties of the nodes in the relational table [Brunel, 2017]

The hierarchical structure of the tree is explicit and clear. It offers lightweight database storage because the parent-child relation is enough to build the hierarchy.

Another approach is to build an order index for each record using nested sets, for example, systematic traversal of index structure with depth-first search. However, this approach is not dynamic and needs periodic re-clustering.

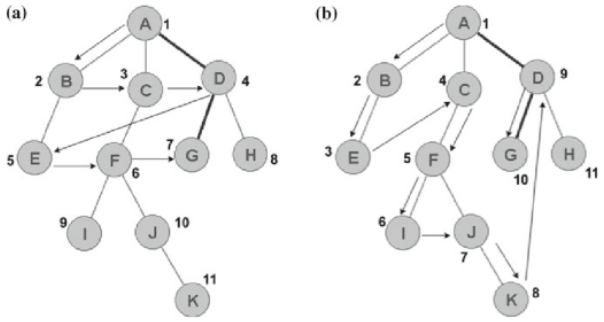


Figure 3.8.: (a) Breadth-first search, (b) Depth-first search [Khemani et al., 2019]

Hierarchy cluster storage model design - managing hierarchical data in RDBMS

The hierarchy storage design starts with mapping the hierarchy of subdivisions into a relational schema associated with objects. Objects are organized into hierarchical clusters, with fewer subdivisions at lower depths and more at higher depths. This means the same objects are assigned multiple times to one cluster on different hierarchical levels. Each cluster is a node value in the direct tree graph.

Parent-child relation
In a relational database, hierarchical structures become flat tables. This involves encoding the hierarchy into one or more table columns using basic SQL data types. One commonly used method is the basic self-referencing table model, as described previously.

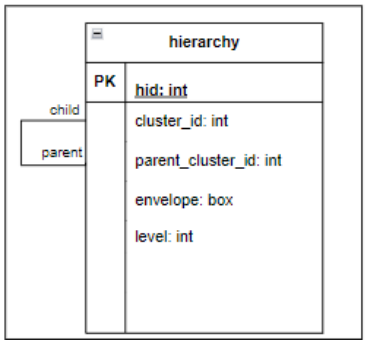


Figure 3.9.: Table hierarchy (initial design)

The table hierarchy supports the management of geometry subdivisions. This is useful to improve querying performance, as it stores bounding boxes hierarchically, similar to R-tree.

- Envelope: The minimum bounding box for the supplied geometry within the cluster on the current depth.
- Level: the depth of the node value (cluster) in the node's subtree. It defines its level with respect to the top level (level = 1).

The field envelope stored in the table hierarchy is a 3D box. The type box is chosen because it is a simple geometric type, compared with normal geometry type and array type. It is the double-precision (float8) minimum bounding box for the supplied objects in the cluster, as a geometry, defined by the corner points of the bounding box. It is computed by taking the 3D extent of the objects that are assigned to the same cluster. This 3D extent is also used for the bounding volume provided in the tileset view.

One-to-many relation
One object is assigned to the clusters on different levels, as many times as the depth of the hierarchical tree. As explained in Section 3.2.1, the table object supports the encoding of geometries, and the table face is associated with this table. The attribute levelN_cluster_id (where N = 1,2,...) is added to the table object to represent the one-to-many relationship, where X refers to the level of the cluster, i.e. depth of the node value.

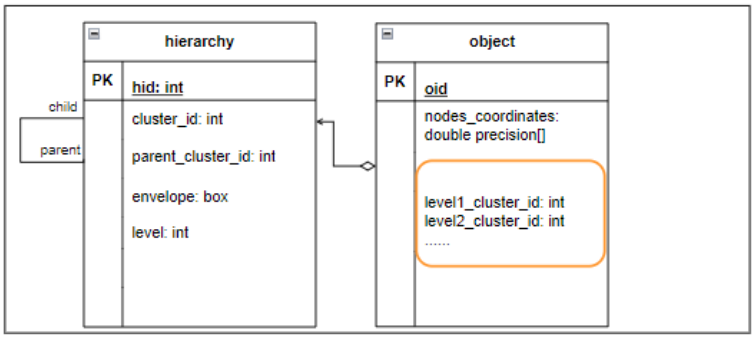


Figure 3.10.: Schema design (cluster that objects are assigned to is associated with table object)

The table hierarchy is significant for fast querying. It clusters objects in a close region into groups on a hierarchical level representing different granularities. To find the cluster that an object is assigned to, one needs to access both object and hierarchy tables, and then query the objects that share the same cluster_id on the same level. This is a bit inefficient.

Storing cluster_ids as multiple rows provides more querying flexibility. However, these objects are frequently queried as a batch. Additionally, storing the object_id as an array in one row is possible to reduce storage overhead. Thus, the object_id records are stored as an array of int in the hierarchy table, enhancing querying efficiency. The attributes that represent the corresponding clusters in the table object are removed.

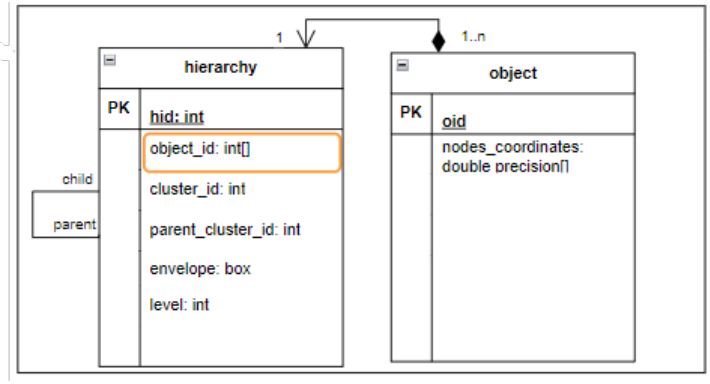


Figure 3.11.: Schema design developed (objects that are assigned to the same cluster are associated with table hierarchy)

Tile storage based on the hierarchical cluster storage

The structure of 3D tiles is a tree structure that includes one root with multiple children nodes. Its organisation can be derived based on the storage models above.

The temp_tid column is added. This represents a temporary ID of the tile. It is an increment number starting from 1 within the same level in the hierarchy table, where the number 1 indicates a root tile. This identifies a unique tile within the same hierarchy level, and it is used to form a URI reference within the tileset pointing to the renderable content.

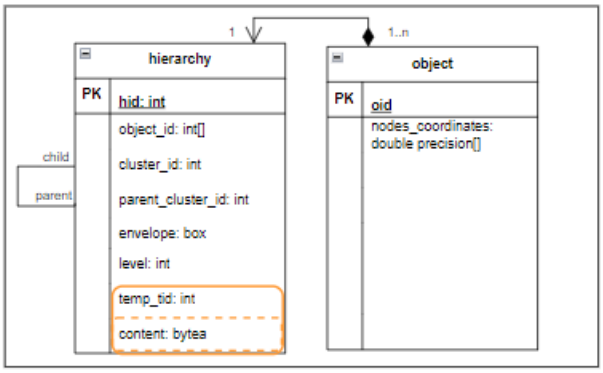
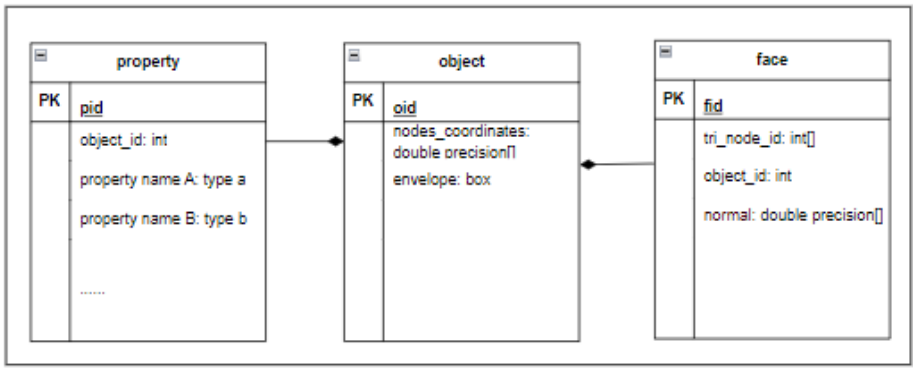


Figure 3.12.: Table hierarchy table adapted to tile organization

As explained in Section 3.1.2, two of the 3D Tiles methods are to serve precomposed b3dm or glTF from the database. To store the renderable content (b3dm or binary glTF), an additional content column (data type: bytea) is designed. This is where the renderable content is stored, and this field is optional. It serves as the temporary storage for tile data in the local database, making it readily available for delivery to the web application.

3.2.3. Associations with attribute

As explained, this table object supports the encoding of geometries. To further provide visualisation of geometries based on their properties, the table property is introduced. To store the attributes associated with the object, a foreign key obejct_id is created in the table attribute to link oid in the object table. The property of an object can be height, the year of construction, building type, etc. The attribute can be generated with SQL functions and operations or linked to an existing attribute table to incorporate with.



(a) Property table

Properties example Name and type:	
"height"	float
"construction_year"	int
"building type"	text

(b) Examples of property name and type

Figure 3.13.: Introduce table property into schema design

3.3. Preparation of the 3D model

To develop and examine the 3D Tiles approach, concave (eg: a cube) and convex geometries (eg: an L-shaped polyhedron) have been created and tested by executing the following SQL query. 3D models representing the physical objects should also be collected and preprocessed to examine the 3D Tiles approach.

```
SELECT ST_GeomFromText('POLYHEDRALSURFACE Z(
((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)))');
```

```
SELECT ST_GeomFromText('POLYHEDRALSURFACE Z(
((2 0 0, 2 2 0, 0 2 0, 0 3 0, 3 3 0, 3 0 0, 2 0 0)),
((2 0 0, 2 0 1, 2 2 1, 2 2 0, 2 0 0)),
((2 2 0, 2 2 1, 0 2 1, 0 2 0, 2 2 0)),
((0 2 0, 0 2 1, 0 3 1, 0 3 0, 0 2 0)),
((0 3 0, 0 3 1, 3 3 1, 3 3 0, 0 3 0)),
((2 0 0, 3 0 0, 3 0 1, 2 0 1, 2 0 0)),
((3 3 0, 3 3 1, 3 0 1, 3 0 0, 3 3 0)),
((3 0 1, 3 3 1, 0 3 1, 0 2 1, 2 2 1, 2 0 1, 3 0 1)))');
```

3.3.1. Geometry validity in the database

To facilitate geometric operations in the database, validation should be performed to guarantee geometry validity and minimal errors. ST_IsValid can be used to check if a given geometry is well-formed and valid according to OGC rules in 2D space. This function helps flag any invalid geometries. Note that for geometries with 3 and 4 dimensions, validity is still only tested in 2 dimensions. This can cause the geometry to self-intersect when collapsed to lower dimensions. Another function is ST_IsValidReason. It returns a reason why a geometry is invalid, such as self-intersection, invalid rings, etc.

3.3.2. Is it a valid polyhedron?

Considering the diverse 3D data models, geometries are expected to be harmonised into the same geometry type for further processing.

The first geometry type candidate is a polyhedron. The reason is that spatial functions take polyhedral surfaces as inputs. Note that restrictions are applied to the input data when creating a polyhedron. The faces must be planar, the edges may not cross, and a polyhedron must be watertight [Teunissen and van Oosterom, 1988]. The normal vector of each face should point outwards.

To check if a polygon is planar, ST_IsPlanar supported by SFCGAL backend is used. It checks if a geometry is planar (in 3D space) and doesn't drop the z-coordinate. It supports 3D geometries and can handle Polyhedral surfaces, Triangles, and Triangulated Irregular Network Surfaces (TIN).

We try to create a valid polyhedron from a volumetric geometry bounded by several polygons represented by Multipolygon Z. However, tilted planar surfaces are considered non-planar because of floating-point precision in some cases. Thus, the restrictions to reconstruct a valid polyhedron are not always satisfied for the input data. This suggests that a more robust approach to performing geometric operations is at the face level rather than the volume.

3.4. Developing 3D Tiles database

This procedure describes how to create the information that makes up 3D Tiles in a database. It includes feature generation for composing a b3dm, and tilest organisation for defining a hierarchical data structure.

3.4.1. Feature generation

This phase is to generate the mandatory data required for composing a b3dm. This section introduces geometric operations, including normal computation, triangulation, and property enrichment. As explained in Section 2.4, the actual renderable model geometry is contained in a binary glTF with ”POSITION”, ”NORMAL”, ”_BATCHID” (and ”indices”). The per-feature properties are defined in a Batch Table.

Normal computation

To determine a surface’s orientation toward a light source for shading, normal is required. Given that a polygon is a closed figure on a coordinate plane, the calculation of the normal for each vertex in a polygon is simplified to calculate the face normal. The process involves selecting two non-parallel edges from consecutive edges, determining edge vectors, applying cross products, and normalising vectors. This method yields a normalised vector perpendicular to the mesh face.

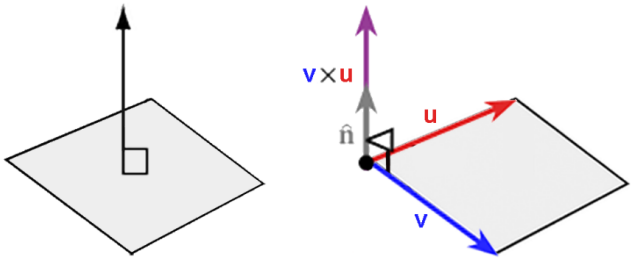


Figure 3.14.: Normal vector of a polygon

The procedure is described in the Pseudo-code as in Algorithm 3.1. Specifically, given three points defining a triangle: p1, p2, and p3 , and if the vectors:

$$U = p2 - p1$$
$$V = p3 - p1$$

The cross product of these vectors gives the normal vector N of the triangle:

$$N = UXV$$

The components of the normal vector (n) in three different axis directions can be calculated by:

$$Nx = UyVz - UzVy$$
$$Ny = UzVx - UxVz$$
$$Nz = UxVy - UyVx$$

Ultimately, normalisation is implemented to save the data size of the normal data to compose a b3dm. the unit normal vector N’ is obtained by normalising the components (Nx, Ny, Nz)

of the calculated normal vector N. This is typically done by dividing each component by the magnitude of the vector:

$$N' = (Nx/\|N\|, Ny/\|N\|, Nz/\|N\|),$$
where $\|N\|$ represents the magnitude of the vector **N**

Algorithm 3.1: CalculateSurfaceNormal

```
Function CalculateSurfaceNormal(Polygon):
    Set Normal to (0, 0, 0);
    if Polygon.vertexNumber < 3 then
        return null;
    Set p1 to Polygon.verts[0];
    Set p2 to Polygon.verts[1];
    Set p3 to Polygon.verts[2];
    Set Vector U to (p2 − p1);
    Set Vector V to (p3 − p1);
    Set Normal.x to (Uy × Vz) − (Uz × Vy);
    Set Normal.y to (Uz × Vx) − (Ux × Vz);
    Set Normal.z to (Ux × Vy) − (Uy × Vx);
    Set Magnitude to √(Normal.x² + Normal.y² + Normal.z²);
    if Magnitude == 0 then
        return null ;
    Set Normal to (Normal.x/Magnitude, Normal.y/Magnitude, Normal.z/Magnitude);
    return Normal;
```

It is expected that the normals point outwards when looking at the mesh from the outside. However, the above algorithm only ensures this if the vertices of each face go around in a counterclockwise direction.

Triangulation

In b3dm, geometries are stored as meshes composed of triangles. In most cases, 3D models are stored as non-triangulated geometries in the database, this suggests a necessary step to implement triangulation. This involves the process of breaking down a 3D model or scene into a series of triangles that can be easily processed and rendered.

In PostgreSQL, two of the triangulation functions are the ST_DelaunayTriangles supported by PostGIS and the ST_Tesselate supported by SFCGAL:

- ST_DelaunayTriangles function supports computing Delaunay triangulation based on a given set of input vertices within a geometry. This function generates non-overlapping triangles that are bounded by the convex hull of the input vertices.
- ST_Tesselate takes both MULTI(POLYGON) or POLYHEDRALSURFACE as inputs. It returns a TIN representation.
- According to the PostgreSQL manual, both functions support 3d and do not drop z-coordinates, indicating that these two functions are promising methods for triangulating 3D geometries in the database.

To test the applicability of these two functions to 3D geometries, we conducted a series of comparative tests. The tested geometries are as introduced in Section 3.3, which are a unit cube, an L-shaped polyhedron, and building footprint polygons.

Testing on convex geometry (axis aligned)
Testing of triangulation functions on faces of a unit cube is executed as in Appendix B.3.1, and query results are shown in Figure 3.15.

The test shows that ST_DelaunayTriangles returns an empty geometry collection When dealing with vertical polygons, while ST_Tesselate works in this case.

	delaunay_result text	
1	GEOMETRYCOLLECTION EMPTY	Vertical faces
2	GEOMETRYCOLLECTION EMPTY	
3	GEOMETRYCOLLECTION EMPTY	
4	GEOMETRYCOLLECTION EMPTY	
5	GEOMETRYCOLLECTION Z (POLYGON Z ((0 1 0,0 0 0,1 0 0,0 1 0)),POLYGON Z ((0 1 0,1 0 0,1 1 0,0 1 0)))	Bottom
6	GEOMETRYCOLLECTION Z (POLYGON Z ((0 1 1,0 0 1,1 0 1,0 1 1)),POLYGON Z ((0 1 1,1 0 1,1 1 1,0 1 1)))	Top

(a) Results from ST_DelaunayTriangles

	tesselate_result text
1	TIN Z (((0 0 0,0 0 1,0 1 1,0 0 0)),((0 1 0,0 0 0,0 1 1,0 1 0)))
2	TIN Z (((0 0 0,0 1 0,1 1 0,0 0 0)),((1 0 0,0 0 0,1 1 0,1 0 0)))
3	TIN Z (((0 0 1,1 0 0,1 0 1,0 0 1)),((0 0 1,0 0 0,1 0 0,0 0 1)))
4	TIN Z (((1 1 0,1 1 1,1 0 1,1 1 0)),((1 0 0,1 1 0,1 0 1,1 0 0)))
5	TIN Z (((0 1 0,0 1 1,1 1 1,0 1 0)),((1 1 0,0 1 0,1 1 1,1 1 0)))
6	TIN Z (((0 1 1,1 0 1,1 1 1,0 1 1)),((0 1 1,0 0 1,1 0 1,0 1 1)))

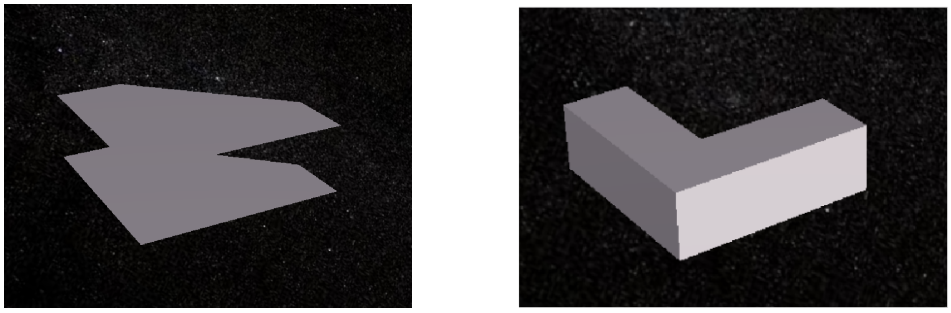
(b) Results from ST_Tesselate

Figure 3.15.: Triangulation of the faces making up a unit cube

Testing on concave geometry (axis aligned)
To further discuss the effectiveness of handling concave and convex surfaces, tests on an L-shaped polyhedron and building footprint polygons are also conducted. Triangulation results are as shown in 3.16.

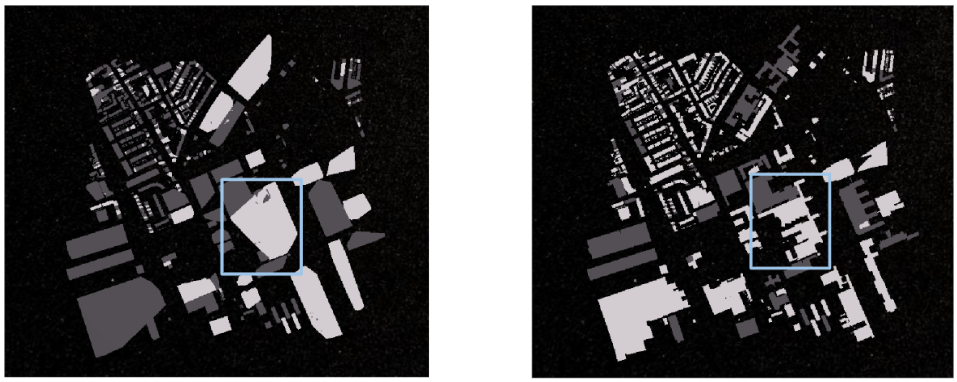
This means ST_DelaunayTriangles does not work correctly with a convex surface. Tests of building footprint polygons further prove that applying ST_DelaunayTriangles to complex concave surfaces results in convex surfaces, while ST_Tesselate can correctly triangulate both convex and concave surfaces, as shown in 3.17.

Testing on a rotated geometry
Tests on a rotated unit cube are also conducted to investigate how floating-point issues affect triangulation functions. The cube is first rotated 30 degrees along the X-axis, and then 30 degrees along Y-axis. This is achieved using ST_RotateX and ST_RotateY in the database. SQL scripts see Appendix B.3.3. Both of the functions work with the rotated cube faces. The execution timing is shown in table 3.2, where ST_Tesselate is faster than ST_DelaunayTriangles.



(a) Results from ST_DelaunayTriangles (b) Results from ST_Tesselate

Figure 3.16.: Triangulation of faces making up an L-shaped polyhedron



(a) Results from ST_DelaunayTriangles (b) Results from ST_Tesselate

Figure 3.17.: Triangulation of building footprint polygons

	ST_DelaunayTriangles	ST_Tesselate
Execution time	0.631ms	10.909ms

Table 3.2.: Comparison of triangulation time complexity on a rotated cube

	ST_DelaunayTriangles	ST_Tesselate
Convex polygon	√	√
Concave polygon	×	√
Vertical polygon	×	√
tilted polygon	Depends on tilted degree	Depends on coordinates precision

Table 3.3.: Comparison of triangulation applicability

However, in some cases, tilted faces are not accepted by ST_Tesslelate. For example, a cube generated in Python, as shown in 3.18, is not accepted by ST_Tesslelate because the points are not in the same plane. This shows that the flatness tolerance of ST_Tesslelate is higher, which means the total change allowed by the actual plane to its ideal plane is smaller.

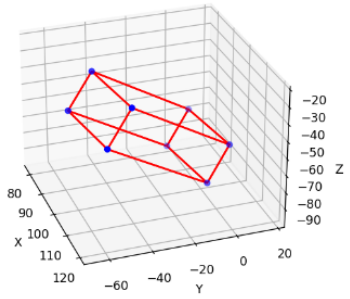


Figure 3.18.: A tilted cube visualised using Matplotlib

Table 3.3 illustrates the applicability of the triangulation function to different cases. As can be seen from the table, ST_Tesselate is more robust in terms of handling different geometries, despite being slower than ST_DelaunayTriangles. ST_DelaunayTriangles results in pitfalls when working with concave faces, while ST_Tesselate performs well. However, the triangulation functions supported by PostGIS and SFCGAL extensions do not always work in 3D. ST_DelaunayTriangles gives empty geometry collection when working with vertical faces, and the ST_Tesselate function fails with tilted faces because of points not lying in the same plane. The possible reason for this is floating-point precision.

To make the SFCGAL function ST_Tesselate work, a planar face input must be ensured, because it strictly checks if a geometry is planar. An adaptive approach with ST_Tesselate is proposed to address this issue. We propose an adaptive method where the non-planar geometries are projected to the XY plane to ensure a planar face. The adaptive step is explained as follows, and Figure 3.19 gives an overview of the steps.

- ST_IsPlanar can be used to check if a face is planar because ST_Tesselate only works when a planar face is ensured. However, there is no proof that ST_Tesselate will certainly accept a face determined as planar by ST_IsPlanar. We choose to project all the faces to make this a robust approach. This means we translate a 3D case into 2D and implement ST_Tesselate.
- For vertical non-planar faces, it can result in invalid geometries when projected onto the XOY plane. In this case, a rotation concerning the x-axis and y-axis is implemented to overcome this issue. After triangulating the projected rotated geometries, triangles

are rotated back to their original position.

- For non-planar faces, the PostGIS function ST_Force2D is used to project non-planar polygons onto the XOY plane. It drops the Z dimension and projects all 3D point coordinates onto a 2D (X, Y) plane.
- Note that the rotation step greatly increases the time complexity. An examination of how often this step is executed will be given in Section 4.2.2.

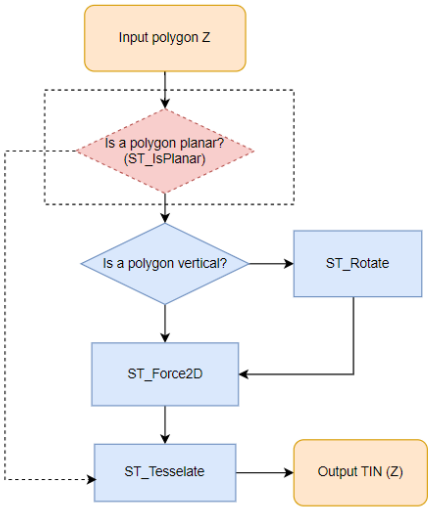


Figure 3.19.: Overview of the steps in the triangulation

Attention should be paid to the orientation of geometries. The first aspect is whether the topology of the original geometry forms a correct orientation. Another aspect is that triangulation can introduce inconsistent orientation between the triangulated faces and non-triangulated faces. The orientation test with ST_Tesselate shows that the geometry before and after tessellation keeps the same orientation, which applies to both 2D and 3D space.

Property enrichment

In a b3dm, property values are stored in a Batch Table, indexable by batchId. This can be used for declarative styling, where the model appearance changes based on predefined rules at the run time.

To enrich the property of the geometries stored in the database, geometric functions enabled by PostGIS and SGCGAL are used. Some of these functions work with polygons and accept geometry types such as a polyhedral surface or a solid. Although a polyhedral surface from multiple 3D polygons is problematic, as explained in section 3.3, a valid polyhedron can be created based on the 2D footprint polygon and height. Examples of rich attributes include height, area, volume calculation, etc.

PostGIS geometries can be retained or dropped when the geometric information required and properties expected in 3D Tiles are complete. In our case, the main purpose is to serve 3D Tiles directly from the database and perform selective visualisation with user-defined

queries. Thus, all of the object geometries are deleted from the tables. This means no PostGIS geometry (Point, line, surface, body, etc.) representing the 3D buildings are explicitly stored in the database. A geometry reconstruction can be performed if further enrichment at the face or volume level is needed. This is enabled with the node coordinates stored in the table object and triangulated geometry topology stored in the table face.

After completing the required geometric information and expected properties in 3D Tiles, the data stored in the database is ready to be composed into tiles in the format of b3dm.

3.4.2. Tileset organisation and tile creation

The organisation of tiles is stored in a tileset JSON. In a tileset, URIs are used to reference tile content, an actual renderable model (eg: b3dm). A hierarchical k-means clustering is proposed for clustering objects and organising 3D Tiles. An example of tileset JSON is provided in Appendix A.2.

Indexing and clustering objects in PostgreSQL

We aim to find an approach natively supported in the PostgreSQL database to index and cluster the geometries. This is important for efficient spatial indexing and speeds up queries within a scene. By reducing the number of objects that a given query needs to examine, spatial queries can be optimised. In addition, it expects a flexible subdivision so that geometries are not split and 3D buildings are not 'destructed' during the subdivision.

Exploring Spatial Indexing in PostgreSQL
R-tree, natively supported with Gist in Postgres, is an option. In an R-tree, spatial data is organized into nested rectangles, with smaller rectangles nested within larger rectangles. Earlier versions of PostGIS used PostgreSQL R-Tree indexes. Later, PostgreSQL R-trees were dropped and Generalized Search Trees (GiST) were used to provide faster search performance. The update of the R-tree index involves the balancing and splitting of the R-tree. Although query efficiency is reduced after large-scale updates and spatial indexes need to be rebuilt, R-tree indexing satisfies 3D city model scenarios that do not require frequent updates.

However, the R-tree structure is implicitly stored in the database when indexed with Gist. This does not directly support the further organisation of 3D Tiles. Additionally, data in R-trees is organised in pages with a variable number of entries, but Gist does not support custom minimum and maximum entries. This indicates a lack of support for customising the number of clusters in a tileset.

Another option is Quadtree or Octree, commonly used for spatial partitioning and data organisation in multidimensional spaces. A Quadtree is a tree structure partitioning a two-dimensional space into regions. It recursively subdivides a space into four equal quadrants or sub-regions. Similarly, an Octree is a tree data structure partitioning a three-dimensional space into eight octants or sub-regions. An Octree has nodes that can have up to eight children, representing the eight subdivisions of a 3D space. Each level of the tree represents a deeper level of spatial detail. However, neither Octree nor Quadtree is natively supported in the Postgres database.

Another direction is using the cluster method. The approach involves partitioning 3D models into independent blocks based on their minimum bounding boxes and evaluating the threshold between the blocks to determine the clusters. The functions *ST_ClusterDBSCAN* and *ST_ClusterKMeans* are two clustering algorithms supported by PostGIS.

ST_ClusterDBSCAN function implements the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. DBSCAN identifies clusters as areas where there is a high density of points separated by areas of low density. It classifies points as core points, border points, or noise points, depending on their neighbourhood density. Core points are those surrounded by a minimum number of points within a specified radius, while border points lie within the neighbourhood of core points but do not meet the density criteria themselves. It is noted that DBSCAN doesn't require the number of clusters as input.

In contrast, *ST_ClusterKMeans* aims to partition the data into a predetermined number of clusters (k) by iteratively assigning points to the nearest centroid and updating centroids to minimize the within-cluster sum of squares. In terms of 3D Tiles organisation, *ST_ClusterKMeans* is an ideal method compared with the *ST_ClusterDBSCAN* function. The reason for this is that K-means with a custom number of clusters ensures that all objects are partitioned into clusters without labelling any point as an outlier. However, *ST_ClusterKMeans* does not support custom minimum and maximum entries for the cluster. The number of objects in each cluster can vary significantly, resulting in large variations in tile size and hence the need for post-processing.

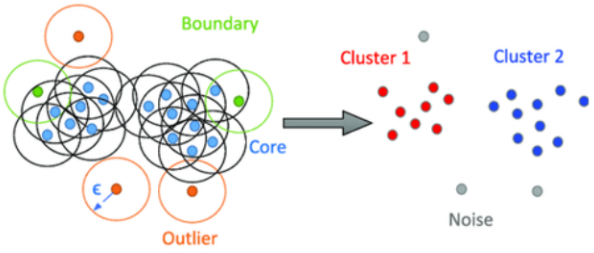


Figure 3.20.: Illustration of DBSCAN clustering algorithm [Khater et al., 2020]

Hierarchical k-means partition
The clustering method proposed is inherited over the R-tree with user-defined cluster numbers. The 3D model is partitioned via top-bottom clustering, as shown in Figure. The top-level shows a coarse partition, while the bottom level is fine-grained. Note that the clusters at the lower level cannot cross boundaries imposed by top levels.

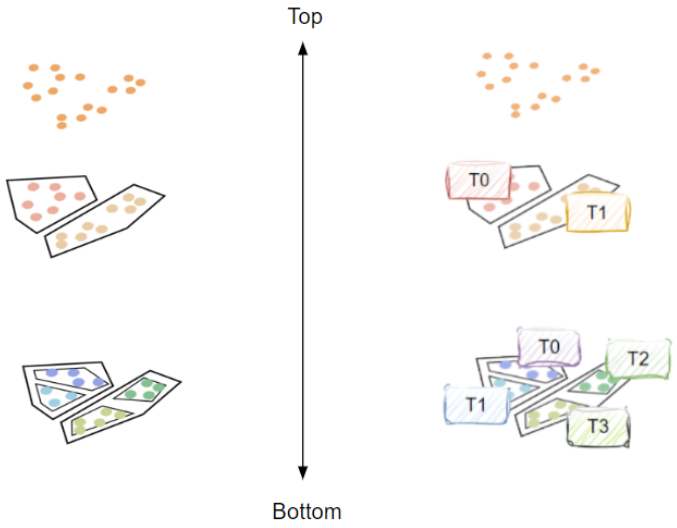


Figure 3.21.: Illustration of hierarchical k-means partition, the circle represents the centroid of the object

First, based on the centroids of the minimum bounding boxes of the objects, the 3D geometries are clustered into multiple blocks. Next, the overall minimum bounding boxes of each block are calculated. Clustering is continued until the stop condition is reached, which can be a customised number of clusters. The number of resulting clusters should not exceed a threshold, which correlates with the number of objects at the current level. A post-processing intervention is required to balance the objects assigned to each cluster.

The partitioning process occurs from the top level to the bottom level. These results are stored within a table hierarchy in the database and used to speed up queries, such as whether a specific domain is within or intersects the search region. The bounding volume of a cluster can be a convex hull of any shape, but considering query efficiency, the bounding box is calculated as a rectangle, as in an R-tree.

Utilizing hierarchical k-means clustering, we group spatial objects based on spatial proximity. Objects are assigned to groups at each level of the hierarchy, ensuring spatial coherence across different scales. This hierarchical tiling strategy optimises spatial querying and further supports the organisation of a 3D Tile.

Tiling based on hierarchical k-means clustering

This step aims to organise a tileset JSON based on the constructed hierarchy in the previous section. This tileset JSON is created as a view relying on the data defined in the table hierarchy in the database, where the tile-tileset structure is stored.

To perform large-scale visualisation of data, 3D tiles use the HLOD (Hierarchical level of detail) technique to subdivide the dataset into manageable file sizes. This technique involves organising objects in a scene into a hierarchical structure, where models at different levels have different levels of detail. This technique involves dynamically switching between different LODs of a 3D model based on the distance of the object from the viewer.

As for objects at a single level of detail, the objects can be organised into tiles based on the clusters on one of the hierarchical levels, as depicted in Figure 3.22. Regarding determining the root tile, one can customise a certain tile as the root node and set other tiles in its surrounding environment as child nodes.

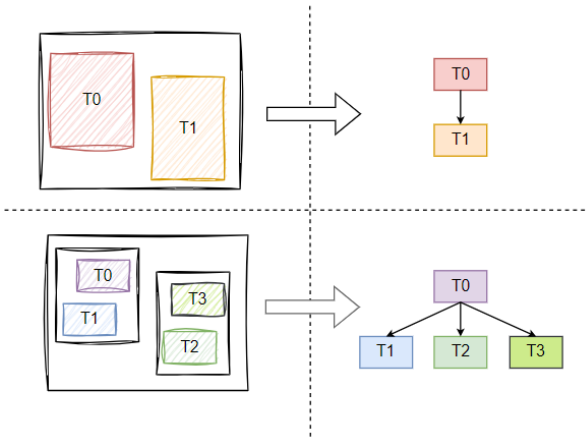


Figure 3.22.: Tile hierarchy represented in a tree structure

The hierarchical k-means partition approach can used for optimising 3D Tiles generation. Regarding serving large-scale 3D city models as 3D Tiles, the number and size of tiles affect the visualisation performance. Determining the optimal cluster number is a user-driven process, involving tests to balance tile size and the total number of tiles. After the initial clustering, the partitions can be refined based on user-defined criteria.

3D Tiles configuration
In this section, key configurations of the tileset are described. These configurations determine how to visualise the data at different levels, including bounding volume, geometric error value and refinement strategy.

bounding volume
The bounding volume defines the spatial extent of a 3D scene. Which tile at the current level is loaded is determined by calculating whether the tile's bounding volume intersects the view frustum. Generally, there are three types of bounding volume in 3D Tiles, which are box, region, and sphere. We select the box type as the bounding volume in the tileset organisation. Because it is simple and easily calculated from the result of the *ST_3DExtent* function. Regarding the hierarchical tile structure, the content of each child tile is within the bounding volume of its parent tile. An example of the box type is provided in Table 3.4. In addition to box types, regions and spheres are also supported (see Appendix A.1).

This bounding volume is defined as a box, and the array defines the properties of the bounding box:

> The first three numbers [0, 0, 10] represent the centre of the bounding box in 3D space. This means the box is centred at coordinates (0, 0, 10).
>
> The next three numbers [20, 0, 0] describe the half-lengths along the x-axis, which is 20 units. This means the box extends 20 units in the positive and negative x-

```
boundingVolume: {
    "box" : [
        0, 0, 10,
        20, 0, 0,
        0, 30, 0,
        0, 0, 10
    ]
}
```

Table 3.4.: Example of bounding volume with type box

direction from the centre.

The following three numbers [0, 30, 0] describe the half-lengths along the y-axis, which is 30 units.

The final three numbers [0, 0, 10] describe the half-lengths along the z-axis, which is 10 units.

Geometric error and refinement strategy
As explained in the previous section, one can custom the tileset, so that the parent tile depicts the important domain, and the child tiles provide surrounding details. This method incorporates the setting of geometric errors.

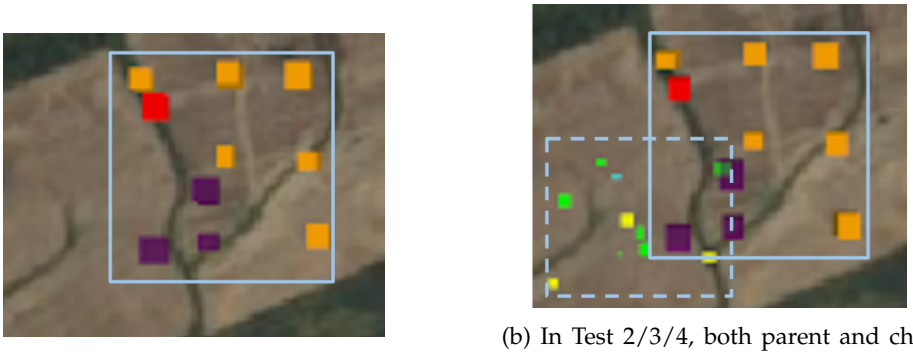
The geometric error is used to determine whether a tile content should be rendered. Tiles closer to the root are set to larger geometric errors, while tiles closer to leaf nodes are set to smaller geometric errors. This section gives the comparative test results between geometric errors. For details of the tested 3D Tiles example, please refer to Appendix A.2.

	tileset	root	leaf node
1	200	50	0
2	200	200	0
3	200	200	50
4	200	200	200

Table 3.5.: Geometric error configuration for tileset, root and leaf node

Refinement determines whether a lower-resolution parent tile remains rendered when a higher-resolution child tile is selected to render. Two types of refinement are allowed, replacement ("REPLACE") and addition ("ADD"). In this example, the tile uses additive refinement, rendering itself and its children simultaneously. In contrast, if a tile uses replacement refinement, when refined it will be replaced by its children.

In addition, the transformation matrix in the tileset.json can be set. It uses the geographic coordinates of the centre point as the origin and calculates a 4 × 4 transformation matrix. Another approach is to use the Ceisum engine to calculate the transformation matrix at runtime. When the transform is not defined, it defaults to the identity matrix as shown below.



(a) In Test 1, only parent tile (orange, red, purple blocks) displays

(b) In Test 2/3/4, both parent and children tiles (green, yellow blocks) display

Figure 3.23.: Geometric error comparison test (same view frame screenshot)

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

3.4.3. b3dm Encoding

The encoding of a b3dm follows three steps. First, encoding binary glTF with POSITION, NORMAL, _BATCHID, and indices. Next, further complete b3dm encoding by composing header, featureTable, batchTable and binary glTF. Finally, the header is added.

Binary glTF

As depicted in Figure 3.24, the glb file contains three parts: a 12-byte header, chunk0 (JSON) and chunk1 (Binary Buffer).

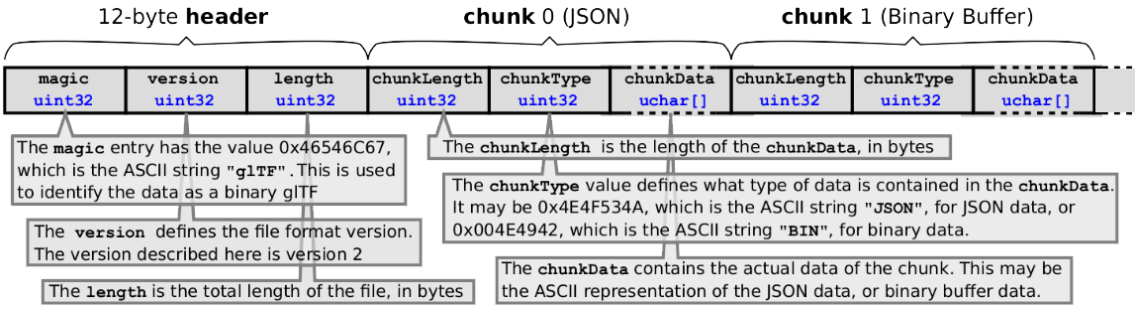


Figure 3.24.: binary glTF encoding [KhronosGroup, 2021]

The 12-byte header is composed of three parts:

- The magic entry, value 0x46546C67

- Version, for example, version 2
- The length of the total length of the binary file

Chunk0 contains three parts:

- The length of the chunkData of chunk 0
- The chunkType, 0x4E4F534A, which is the ASCII string "JSON". It defines what type of data is contained in the chunkData
- The ASCII representation of the JSON data

Chunk1 contains three parts:

- The length of the chunkData of chunk 1
- The chunkType, 0x004E4942, which is the ASCII string "BIN"
- The ASCII representation of the binary buffer data

3D data, including POSITION, NORMAL, _BATCHID and indices, is stored in bin data in chunk 1, where the JSON data in chunk 0 refers to. In the JSON data, meshes provide the index of primitive attributes and indices that point to the accessors. The index of bufferView can be found in the corresponding accessors, which point to bufferViews. This bufferView provides information on the byteLength, byteOffset, and byteStride, indicating how to unpack the 3D data in binary.

```
{
    "meshes": [{
      "primitives": [{
        "attributes": {
          "POSITION": 0,
          "NORMAL": 1,
          "_BATCHID": 2
        },
        "indices": 3,
        "material": 0,
        "mode": 4
      }]
    }]
  }
```

Figure 3.25.: An example of accessors in JSON data

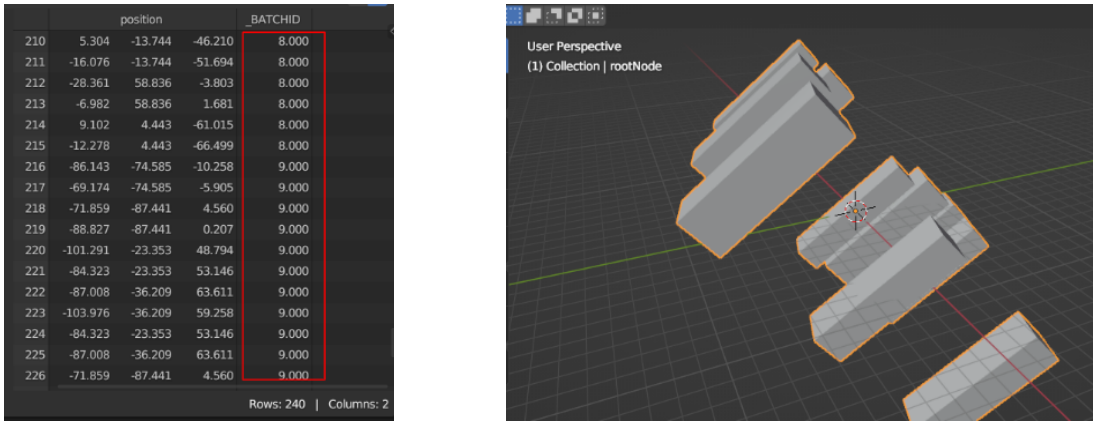
In accessors in JSON data, it defines a method for retrieving data as typed arrays from a buffer view. The componentType values specify how the data is stored within the GLB file. For example, for attribute POSITION, the component type is 5126 for foat, the data type in accessors is VEC3 for 3D vectors, and the target in corresponding bufferViews is 34962. This property is used by the renderer to recognize the data that the buffer view refers to. 34962 stands for ARRAY_BUFFER, indicating that the data is used for vertex attributes.

In the example, the Component Type is set as 5123 unsigned short, ByteStride 2. It requires $0 \leq$ number $\leq$ 0xffff, i.e. range from 0 to 65,535. This indicates the indice number trying to

Primitives	Data type	Component Type	Number of components	ByteStride	Target
POSITION	VEC3	5126 float	3	4	34962
NORMAL	VEC3	5126 float	3	4	34962
_BATCHID	SCALAR	5126 float	1	4	34962
indices	SCALAR	5123 unsigned short	1	2	34963

Table 3.6.: Data Specification Examples

pack into an unsigned short integer format exceeds the range of unsigned short (5123). However, when working with a large dataset, ushort format is not applicable. To accommodate larger values within the GLB file, it is replaced with using unsigned int (5125), ByteStride 4. However, it is noticed that changing the datatype from ushort to uint increases the file size.



(a) POSITION and _BATCHID

(b) Visualisation in blender

Figure 3.26.: A binary glTF exmaple of 10 cubes

Not all of the attributes are compulsory, and keeping indices in the file is not mandatory. Both indexed and non-indexed methods are available to draw a geometry on Cesium or a WebGL application. Take a cube as an example. In the non-indexed approach, 36 positions are required without encoding indices. In the indexed approach, the 3D model is represented by 24 positions and 36 indices.

Apart from indices, it is allowed to remove _BATCHID and NORMAL, but to keep POSITION. However, recomputing "NORMAL" from scratch takes time. _BATCHID is the key to link geometry in the binary glTF and properties in the Batch Table. Indices are needed to reduce the data size and improve render performance.

Feature Table and Batch Table

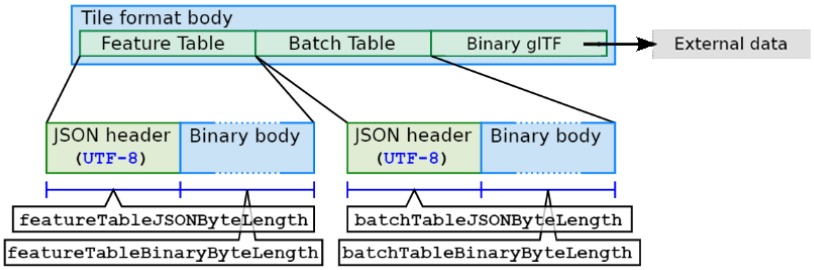


Figure 3.27.: Feature Table and Batch Table, modified from [CesiumGS, 2021]

If a b3dm includes a Feature Table, the b3dm header will also contain featureTableJSONByteLength and featureTableBinaryByteLength (uint32 fields), which can be used to interpret the corresponding part of the Feature Table. The data is stored in a JSON object, for example, "BATCH_LENGTH":10. This is used for global semantics BATCH_LENGTH in a b3dm. The data can also be referenced in the binary body.

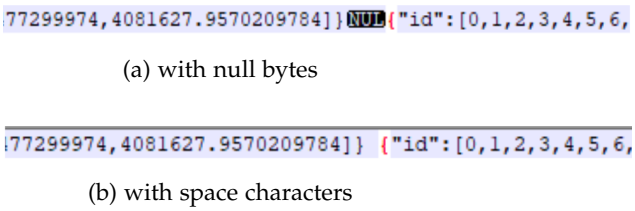
If a tile format includes a Batch Table. The JSON Header and Binary Body are specified according to batchTableJSONByteLength and batchTableBinaryByteLength (Data type: unit32) in the b3dm header. It can be represented as an array of values. For example, "height" : [10.0, 20.0, 15.0]. The length of each array is equal to batchLenghth, which is specified in each tile format. In b3dm, it is the number of models in the tile. For example, 10 means there are 10 objects, and the min and max values of batch ID in the binary glTF should be 0 and 9. Another way is referencing data in the binary body, represented by a JSON object with byteOffset, componentType, and type properties.

Padding rules

Encoding the binary glTF with featureTable and batchTable should be aligned to a set of padding rules. The rules are summarised as follows:

1. The total byteLength must be aligned to an 8-byte boundary.
2. The Feature Table JSON must be padded with trailing Space chars (0x20) to satisfy alignment requirements of the Feature Table binary (if present).
3. The Batch Table JSON must be padded with trailing Space chars (0x20) to satisfy alignment requirements of the Batch Table binary (if present).
4. The binary glTF must start and end on an 8-byte alignment.

Originally, glb is aligned to an 4-byte boundary. If packing a glb to b3dm, it is necessary to check if it is aligned to an 8-byte boundary. Otherwise, it requires padding, either to JSON chunk data or bin chunk data. One possible way is adding the padding bytes in JSON chunk data. Consequently, this affects the total length in the header chunk of glTF, and the length in json or bin chunk. To be specific, pad binary glTF with null bytes (0x00), i.e. 00000000 or space characters (0x20), i.e. 00100000. However, the Feature Table and the Batch Table are only allowed to pad with space characters (0x20).



(a) with null bytes

(b) with space characters

Figure 3.28.: Screenshot of padding results

3.5. Web server query and visualisation

After the procedure above, the 3D Tiles data is ready to deliver for viewing and analysis. To directly serve 3D Tiles from the database, a connection between the database and the web client via a Flask web server needs to be set up.

3.5.1. Direct web access

In the typical approach, 3D Tiles are produced through a conversion process using specialised software (eg: FME). Attributes associated with the geometries are stored separately in a database, while geometries are stored in files next to the database.

In the proposed approach, 3D tiles from the database are directly served for viewing and analysis. Figure 3.29 illustrates communication between the client-side (CesiumJS) and server-side connecting to the 3D Tiles database with the Flask API. CesiumJS, as a web interface based on Javascript, can connect with servers by sending and receiving HTTP requests. Flask Python library establishes a server API that queries the database and sends 3D Tiles to the web client. It also cooperates with a set of processing for 3D Tiles generation outside DBMS. By communicating with the database, it avoids 3D Tiles file export before transferring to Cesium and contributes to a cleaner system architecture with geometry stored in the DBMS.

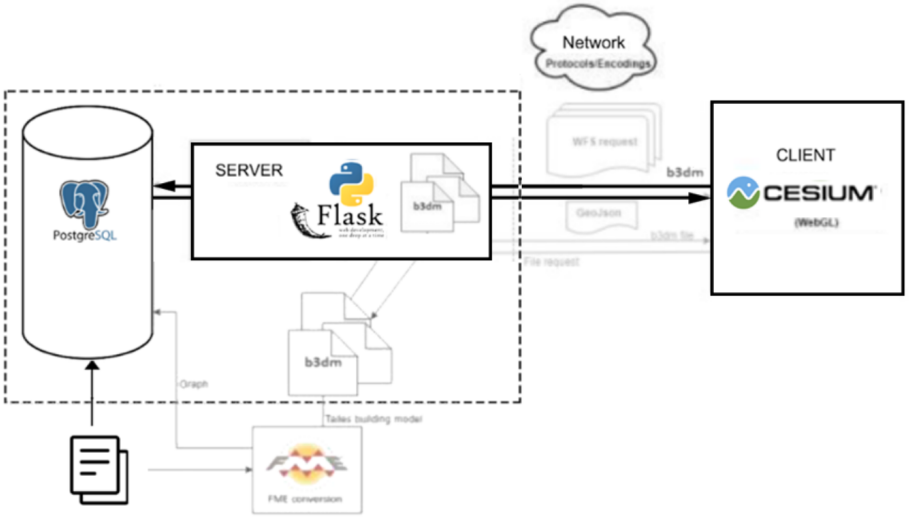


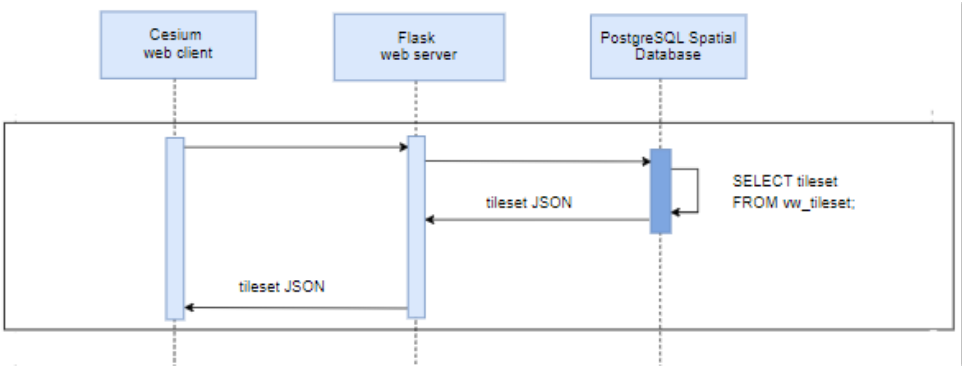
Figure 3.29.: The API requests established between Cesium and Postgres through Flask

CesiumJS, an open-source Javascript library for 3D visualisation on the web, is chosen because of its originally native support for 3D Tiles. It is noted that there are more options to load 3D Tiles for viewing and analysing apart from Cesium, such as 3DTilesRendererJS [NASA AMMOS, 2020]. QGIS 3.34 has also been considered. However, the 3D Tiles functionality in QGIS mainly supports loading data from Cesium ion and Google 3D. This is not ideal for Flask API testing that serves 3D Tiles directly from the database, as it complicates the data request procedure.

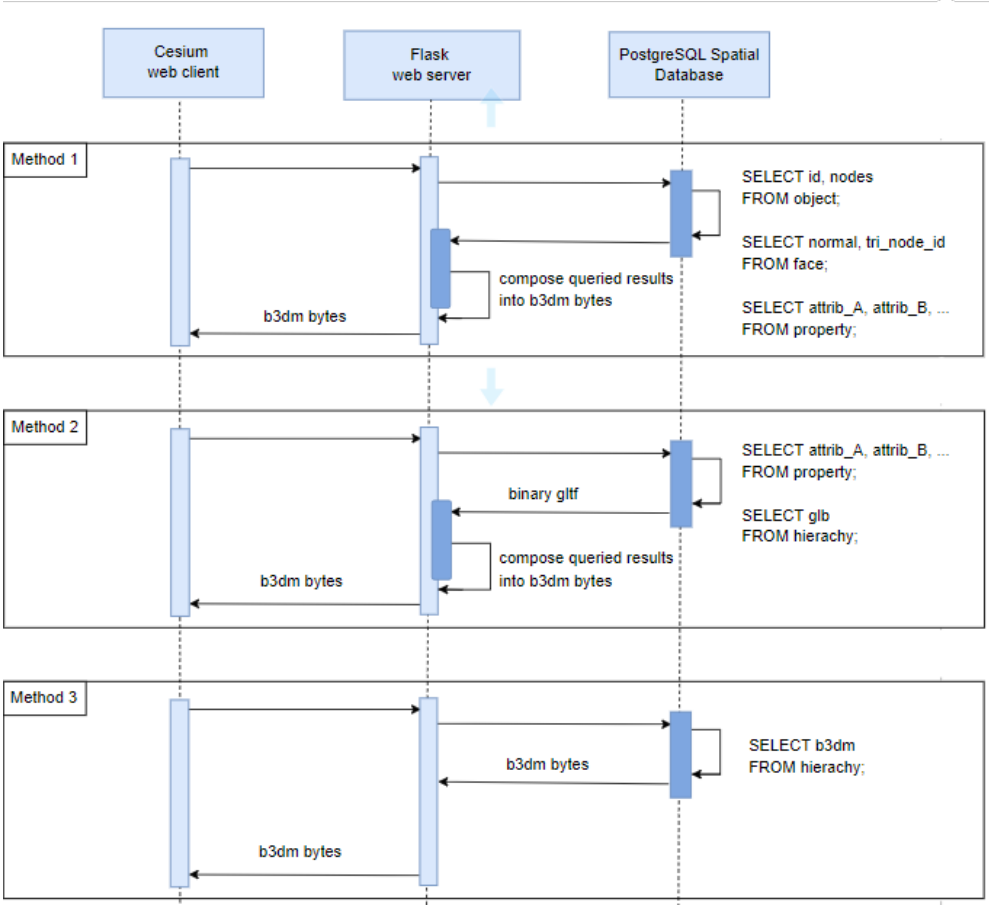
As explained in the previous Section 3.1, the on-the-fly method and its variations are proposed to serve the b3dm from the database. The differences lie in whether the binary glTF and the properties are composed or not. The sequence diagram shows how API requests 3D Tiles in Figure 3.30. The tileset is called first, and then tiles (b3dm) are called referred to by URIs in the tileset. The Ceisum provides a progressive loading mechanism that delivers large data into smaller tiles. This ensures the web client displays the data stably and effectively. Note that the issue can occur when geometry or attributes are simultaneously updated by the user. However, in this prototype, the operations only focus on data access and exclude web-side user updates.

3.5.2. Attribute and spatial query

Queries based on the spatial extent or attribute conditions are needed to query a subset of the available data, as presented in Figure 3.31



(a) request a tileset



(b) request a b3dm

Figure 3.30.: Sequence diagram representation of the way the web server works

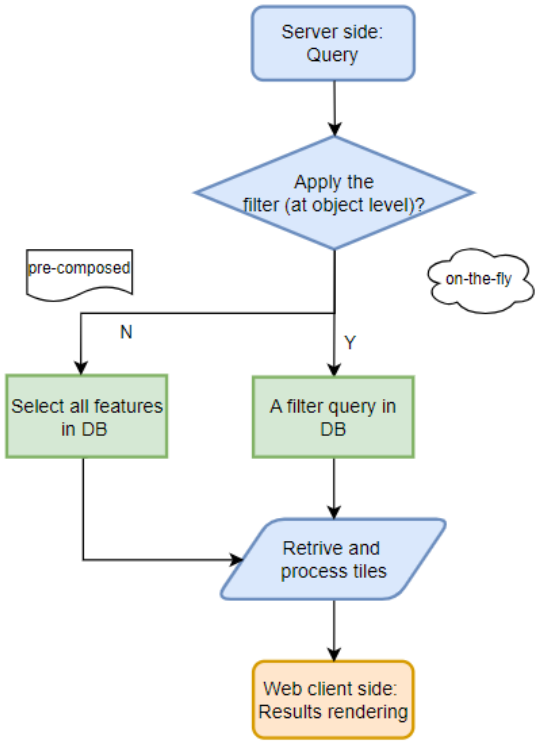


Figure 3.31.: The workflow of a filter query

Generic attributes have been created for objects, as explained in Section 3.4.1. We can apply an attribute filter at the object level and serve a subset of the dataset for visualization and analysis on the CesiumJS-based web client.

Spatial queries can be performed, and objects can be filtered within the search area. PostGIS and SFCGAL functions support spatial relationship checks, such as intersection, within, and contains. One of the frequently used functions is ST_Intersect. It returns true if two geometries have any points in common. A distance tolerance of 0.00001 metres is used so that points that are very close are considered to intersect.

Spatial indexing is critical in narrowing down the search region. Methods such as R-tree indexing and k-means clustering aid in organising spatial data into bounding volumes, facilitating efficient spatial filtering. Based on the hierarchical k-means clustering explained in Section 3.4.2, a method to help reduce the number of candidate objects is proposed, as shown in Algorithm 3.2.

3.5.3. Web client visualisation

This phase discusses how to present the model and attribute information to users. Declarative styling is used to change the appearance of the model at runtime flexibly.

Visualization of geometries helps people understand the environment surrounding them. Apart from strengthening the visible geometric attributes according to styling, 3D Tiles can

Algorithm 3.2: Query Procedure

```
Input  : geometry p
Output: Tiles intersecting p
if intersects(p, parent.MBR) then
    foreach child.MBR do
        if intersects(p, child.MBR) then
            report tiles that intersect p
        else
            report p is within total tile region but out of children tile region
else
    report p is out of total tile region
```

also support invisible attributes. There are some attributes important in human interaction with the city that cannot be perceived directly, such as the building age. These can all be considered as variables and styling the visualisation [Mao et al., 2020].

The appearance can follow the pre-configured appearance of transparency, colour, and texture in a binary glTF. For example, the baseColorFactor property in the binary glTF can be set differently, as shown in Figure 3.32. However, this is hardcoded and does not suit the needs of dynamic visualisation based on different properties.

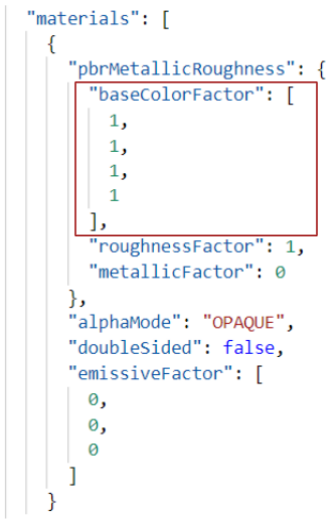


Figure 3.32.: Screenshot of JSON chunk in a glb

Compared with a configured appearance in the binary glTF, declarative styling is more flexible. It defines style rules in JavaScript (JS) code and styling appearance based on property values in Cesium at runtime. The model's appearance changes at runtime based on its properties. This includes changing colours, opacity, etc., to convey attribute information through visualisation.

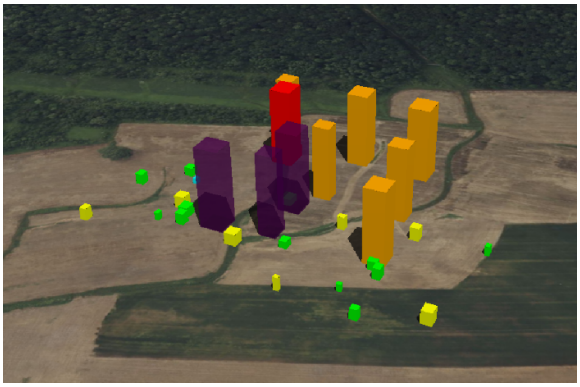


Figure 3.33.: b3dm styled based on height value in Cesium via Javascript

In addition to visualization, users can interact with content running on a web browser by clicking on a model, and the associated properties will be displayed.

4. Implementation and Experiments

This chapter describes the prototype implemented to show whether the 3D Tiles approach provides an effective DBMS solution to serve 3D Tiles directly. For this reason, certain tests and benchmarks have been performed. The chapter is organised as follows: In Section 4.1 the tools and datasets used for tests and benchmarks are presented. After that, Section 4.2 gives details about the database storage implementation and the methods for serving 3D Tiles on the fly from the implemented database and variants of the approach.

4.1. Tools and database used

The implementation is available on GitHub: 3dtiles, providing transparent access to the source code for collaboration and further development. The script takes (Multi)PolygonZ as geometry input, and implements the methodology of Chapter 3. See Appendix D for more details.

4.1.1. Software

A brief overview of tools, programming languages and necessary libraries and extensions used for the proposed methodology is shown in Figure 4.1.

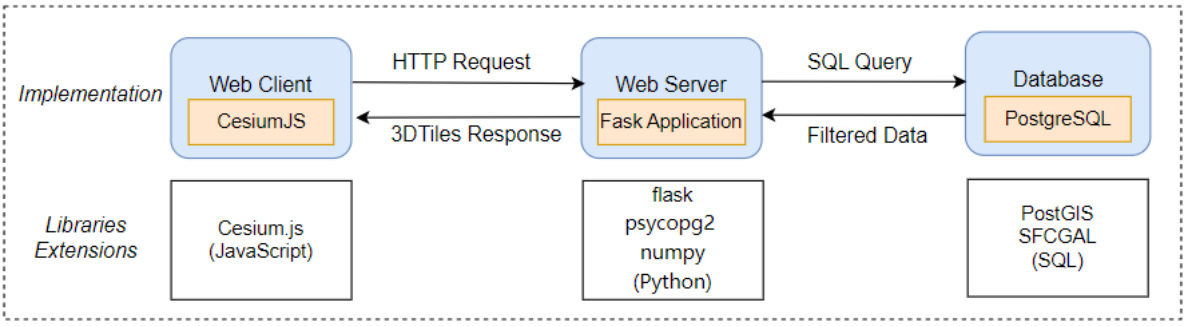


Figure 4.1.: The overview of tools used

- **ogr2ogr**

  GDAL ogr2ogr is used to load 3D data in the database. It is a command line tool of GDAL, which converts simple feature data between file formats [OSGeo, 2024].

- **PostgreSQL**

PostgreSQL Spatial is used because it is a powerful open-source relational database management system. It enhances the capability to process spatial data by enabling extensions such as PostGIS and SFCGAL.

- **Flask**

  Flask is chosen for its ability for easy and fast prototyping of a web server. However, note that the time efficiency of the code is limited.

- **CesiumJS**

  CesiumJS is an open-source JavaScript library for 3D globes and maps based on WebGL, for creating imagery and vector layers from various data sources.

In this setup, the Flask web server acts as an intermediary between the Cesium web client and PostgreSQL. It interacts with the database for data retrieval in response to the HTTP requests, and sends the data to the Cesium application in the format of 3D Tiles.

4.1.2. Datasets

We have tested the approach on primitives in the previous chapter. 3D digital models representing the physical world are also collected and preprocessed to test the 3D Tiles approach.

A brief overview of the experiments and reasons for dataset choosing is summarised as follows:

- To evaluate the 3D Tiles serving approaches, we use unclustered data, eliminating the effect of variation in the number and size of tiles. Datasets are collected from 3DBAG. The 3DBAG dataset has both LOD1 and LOD2 buildings. Datasets of different characteristics complement each other. The differences are relevant to evaluate how the proposed 3D Tiles methodology behaves according to the LOD1 and LOD2 of a building.
- To further evaluate the 3D Tiles tiling method, we use clustered data. The dataset is collected from PDOK. The differences are relevant to evaluate the overall scalability of the 3D Tiles methodology. Each PDOK dataset covers a larger area than 3D BAG, and fewer objects are split. Thus, it reduces the number of split objects, possibly affecting the tiling objects' evaluation. We use clustered data to test the tiling method.

As explained in Chapter 3, the project focuses on 3D volumetric geometries at a single level of detail. Tests on LOD1 and LOD2 models are implemented parallelly, which lays a basis for adapting the approach to multiple Levels of Detail in the future.

4.2. Implementation prototype

This section describes key implementation details of the methodology introduced in Chapter 3. For the complete implementation, please refer to the source code online and supplementary scripts in Appendix C.

4.2.1. Data preprocessing

Datasets are collected from 3DBAG and processed to examine the 3D Tiles Approach. The data is preprocessed to obtain harmonised valid geometries at a single level of detail. This includes three parts: data loading, geometry harmonisation and validity.

Data loading

Data loading into the PostgreSQL database is enabled using ogr2ogr, which is part of the GDAL suite. ogr2ogr enables converting and importing diverse formats like GeoJSON, KML, GML and more directly into PostGIS. The steps for extracting and generating volumetric geometry from the collected gpkg datasets are as follows:

LOD1/ LOD2 of 3D buildings To load data into the PostgreSQL database, we first convert the 3D layer (3D Multi Polygon) with LOD 1.2 and LOD 2.2 into PostGIS dump format, using ogr2ogr, and then load the dump file into the database. This results in a table in the database where the 3D geometry is stored as Multipolygon Z, with coordinate reference system EPSG:7415.

LOD1 of an extruded building For LOD1, the 3D model can be fully reconstructed from the 2D model by taking the 2D polygons and extruding each to one of their roof height values from their heights at the surface level. The 2D polygons represent the 2D projection of the 3D model's roof planes.

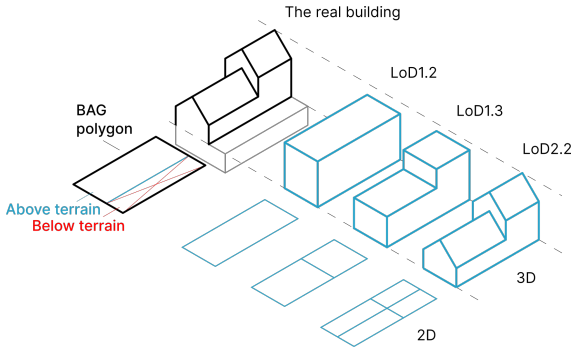


Figure 4.2.: The relation between a real-world building and 3D representation in the 3DBAG [Peters et al., 2021]

Similarly to 3D layer loading, first, we convert the 2D layer (Polygon) with LOD 1.2 and the layer with pand (where b3_h_maaiveld represents heights at surface level is stored) into PostGIS dump format using ogr2ogr and then load the dump files into the database. The attribute fid is used to join the two layers.

The 2D footprint is extruded using the height value to obtain LOD1.2 3D models represented by polyhedral surfaces. The maximum roof height was chosen (b3_h_dak_max), as it is associated with all of the geometries. This is executed by ST_Extrude(geometry g1,0,0, height). Then the 3D geometries can be translated to corresponding ground height if terrain is concerned in the use case. The corresponding SQL query is ST_Translate(geometry, 0, 0, b3_h_maaiveld).

Harmonisation

The harmonisation includes converting various input geometry types into Polygon Z and reprojecting to coordinate reference system EPSG4978.

Geometry type harmonisation To harmonise different geometry types (Mulyipolygon Z, Polyhedral surface), both the loaded and extruded volumetric geometries are expanded to polygons, with the query step (ST_Dump(geometry g1)).geom. This results in two tables in the public schema, object and face, where the table face stores geometries dumped from raw data as Polygon Z. In table face, object_id references the id in the table object. The object ID introduces a common administrative and a unique identifier (ID) for all the geometries in the database.

CRS transformation To maintain a consistent spatial reference system, if geometries come from multiple datasets with different CRS(Coordinate Reference System), reprojection to a unified CRS needs to be performed. In our case, the CRS of the source datasets are under the same CRS, EPSG:7415. To be precise, this is the combination of the Dutch national coordinate system EPSG:28992 (Amersfoort/RD New) and EPSG:5709 (NAP height). This means the coordinates of a point are in three dimensions, including its latitude, longitude, and elevation above or below the Normaal Amsterdams Peil (NAP) datum.

Geometries are transformed from the CRS of the source data (EPSG:7415) and target CRS (EPSG4978), as compatible with the Cesium web client. The transformation is implemented in the database using ST_Transform, which supports transforming spatial data from one Coordinate Reference System (CRS) to another.

Geometry validity

SQL queries are implemented to validate and inspect resulting 3D polygons. Examples of query results are as shown in Table 4.1, and it shows that ST_Transform introduce some errors.

It is expected to validate the geometry and fix invalid geometries to implement the 3D Tiles database approach further. However, functions supported by PostGIS or SFCGAL implement different definitions to determine a valid geometry. Obtaining a non-horizontal 3D geometry that meets both PostGIS and SFCGAL's validity rules is tricky. In addition, functions to fix 3D geometries are missing. For example, a feature that eliminates float error issues and results in watertight geometries is expected.

4.2.2. Feature generation

The feature creation is implemented by developing the database based on the proposed data model in Section 3.2 using PostgreSQL. This phase mainly contains two parts: geometry and attributes.

For the geometry, steps are followed to process geometric information for 3D Tiles and store the information based on the proposed data storage model. First, we compute the normals because they are used in generating b3dm and serve as a flag in the next step of triangulation. Second, triangulation is implemented. Next, the unique coordinates of the object and the topology of the meshes (triangulated faces) are stored.

Before ST_Transform

	ST_Delaunay Triangles	ST_Tesselate e	ST_IsValid	ST_IsPlanar	Total face number
lod22_3d	8: successful 46: geometry collection empty	47: successful 7: Error, Invalid polygon	7: true 47:false	47: true 7: false	54
lod12_3d	2: successful 6: geometry collection empty	All successful	2: true 6: false	All true	8
lod12_2d	Same as above				

After ST_Transform

	ST_Delaunay Triangles	ST_Tesselate	ST_IsValid	ST_IsPlanar	Total face number
lod22_3d	50: successful 4: geometry collection empty	32: true 22: Error, Invalid polygon	52: true 2: false	32: true 22: false	54
lod12_3d	All successful	2: successful 6: Error, Invalid polygon	All true	2: true 6: false	8
lod12_2d	Same as above				

Table 4.1.: Query results example of one building before and after ST_Transform

	planar	non-planar	vertical & non-planar	vertical	non-vertical
LOD1 of extruded buildings	1601	11592	0	0	13193
LOD1 of 3D buildings	2082	11372	0	0	13454
LOD2 of 3D buildings	11392	22082	5	15	33459

Table 4.2.: Face count for tested dataset of LOD1 of extruded buildings, and LOD1 and LOD2 of 3D buildings

In the intermediate step of developing the storage database model, geometric computation is implemented on the level of node and face primitives, i.e. Point geometry and Polygon geometry.

Normal computation

As mentioned in Section 3.4.1, normal calculation is implemented at the face level. For an arbitrary 3D polygon lying on the coordinate plane, each vertex in the same polygon is assigned the same face normal. This phase involves finding non-parallel edges, determining edge vectors, applying cross products, and normalising vectors.

After a preprocessing phase in the previous section, geometries are harmonised as Polygon Z. To find non-parallel edges, steps are implemented as follows. *ST_ExteriorRing(polygon)* is used to find the exterior ring that composes a polygon. The first three consecutive vertices of the linestring are extracted using *ST_PointN(linestring, N)*, where N = 1,2,3. ST_PointN returns the Nth point in a single linestring or circular linestring. Note that the index of the PostGIS function ST_PointN is 1-based.

To facilitate the subtract and cross-product operation on vectors, user-defined SQL functions ST_Subtract and ST_CrossProduct are created. For further details, please refer to SQL scripts in Appendix B.2.

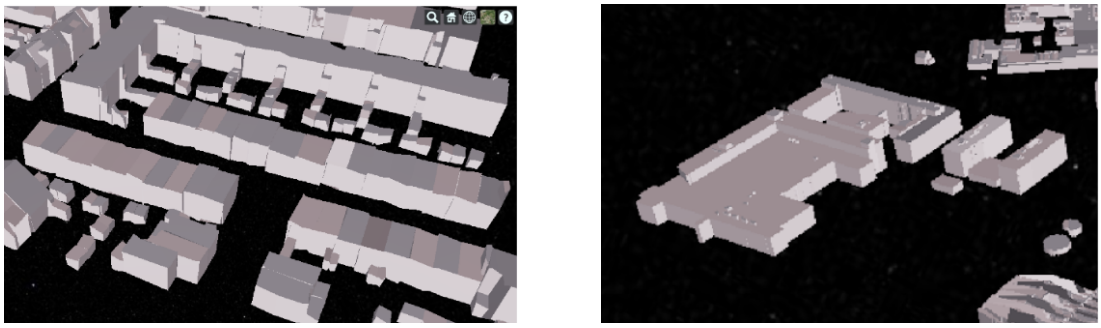
Triangulation

To evaluate how the triangulation method behaves on geometries, we implement tests on 3D buildings of LOD1 and LOD2.

As explained in Section 3.4.1, an adaptive approach from SFCGAL function ST_Tesselate that projects rotated geometry onto a 2D plane is performed to overcome the non-planar face issue. The rotation should be applied to vertical faces. A threshold (1e-6) is used to check if a face is vertical. We compare the z-axis of a normalised normal with 1e-6; if it is smaller, then apply a rotation.

To examine the cost of the adaptive methods added to the algorithm, we count the number of planar and non-planar faces, as well as vertical and non-vertical faces. The results provided in table 4.2 are drawn from 3D Building at LoD1/LOD2 and 3D extruded buildings collected from the 3DBAG portal [Peters et al., 2021]. The triangulation result is presented in Figure 4.3.

For LOD1 models, it is possible to triangulate the vertical faces with ST_DelaunayTriangles because it works with surfaces that are not strictly vertical. For non-vertical surfaces that are possibly roofs with concave shapes, we use the more sophisticated algorithm ST_Tesselate. However, sometimes this results in missing vertical faces.



(a) Buildings of LOD1 (b) Buildings of LOD2

Figure 4.3.: Triangulation result of ST_Tesselate

As explained in 3.2.1, the meshes are stored as coordinates in the table object and mesh topology in the table face. To store the topology from the triangulated geometries, indexing the triangles in an object is necessary.

The coordinates are extracted from geometry with ST_DumpPoints (geometry geom). This function returns a set of geometry_dump rows, each containing a geometry field and a path field (array of integers). These are formed into an array using array_agg[expression [ORDER BY]]. The order of expanding points(ST_Dump) before and after applying ST_Force2D is important. A wrong order can result in wrong triangles. The coordinates are extracted from geometry similarly. After obtaining the coordinates of triangles and the coordinates of the object, We compare and find matching or similar coordinates. In this comparison, a tiny threshold value is utilised to allow for floating-point error in the numerical values. This results in the tri_node_id column in the table face, and the nodes_coordniates in the table object.

Regarding finding the index for each triangle position by comparing object coordinates and face coordinates, I implemented this phase in Python using NumPy, which is a Python library convenient for working with arrays. Despite trying to implement the approach as much as possible in the database, it is not convenient to work with the nested multidimensional arrays in PostgreSQL. The drawback is that this greatly adds data exchange between DB and Python. Another issue that may arise from this is that rounding errors cause variation between the resulting coordinates and the original geometry. Because the computed geometric information mainly serves to compose a b3dm, this rounding error is considered acceptable for visualisation purposes.

Given the further need to encode a b3dm from coordinates and triangle topology, it is noted that there is a forward and reverse process between building a typology and constructing a geometry. During geometric computation, indices of triangulated faces are looked up from the coordinates of each object. When serving the b3dm file from the database, mesh geometry is reconstructed from the mesh topology by looking up its coordinates in the object table.

Attribute: Property enrichment

A table property was created. In our case, we focus on buildings at a city scale, where 3D volumetric geometry is considered a feature in b3dm. Thus, the property is recorded and

identified with an object ID, and a constraint is set. This is to enforce referential integrity, where the values in the object_id column of the property table must exist in the id column of the object table.

```
ALTER TABLE property
ADD CONSTRAINT fk_object_id FOREIGN KEY (object_id) REFERENCES object(id);
```

Property enrichment is implemented on the geometry stored in the database loaded from the dataset. The process of enriching properties with functions and operators is discussed in two cases: LOD1/LOD2 of 3D buildings and LOD1 of extruded buildings.

LOD1/LOD2 of 3D buildings
The property enrichment for LOD1/ LOD2 of 3D buildings is implemented at the face level. The reason for this is that the geometries are stored as Multipolygon Z from the dataset. It is problematic to create a polyhedron from these Multipolygon Z.

One of the examples is height calculation. First, compute the bounding box of the set of faces (Polygon Z) that bounds the same object using ST_3DExtent. Then, compute the difference between Z maxima and Z minima of the 3D bounding box. To update the property table with the height property. A query that updates the property name and value is as follows:

```
ALTER TABLE property ADD height float;
UPDATE property
SET height = (ST_ZMax(o.envelope) - ST_ZMin(o.envelope))
FROM object o WHERE object_id = id;
```

Another example is 2D area calculation. ST_Area is used and returns the area of a polygonal geometry. The SQL script is provided in Appendix B.4.1

LOD1 of an extruded building
Property enrichment for extruded 3D buildings can be implemented on both the solid(body) and the surface geometry because it is easy to get a valid Polyhedrasurface from a 2D footprint and height.

An example regarding implementation on the solid is volume calculation. First, a closed polyhedral surface is created by extruding the 2D polygon along the Z axis. Next, the closed surface is converted to a solid for volume calculation. Otherwise, it is treated as an area and returns 0. As explained in Section 3.4.1, PostGIS geometries are dropped when the geometric information required and properties expected in 3D Tiles are complete. Here, we provide an example of how to update the property table to add a new generic property volume from the raw geometry. In our implementation, the row data dumped from the GPKG file into the database is stored in the dbuser schema, while the 3D Tiles implementation is in the public schema. A cross-database query is executed as follows, where geom is a 2D polygon.

```
ALTER TABLE property DROP COLUMN IF EXISTS volume;
ALTER TABLE property ADD volume float;

UPDATE property
SET volume = subquery.volume
FROM (
    SELECT
        lod.fid AS id,
        ST_Volume(ST_MakeSolid(ST_Extrude(lod.geom, 0, 0, b3_h_max))) AS volume
    FROM dbuser.lod12_2d_9_284_556 lod
```

```
) AS subquery
WHERE property.object_id = subquery.id;
```

Note that the ST_Volume computation takes around 20000ms for only 10 objects. The execution time for ST_MakeSolid and ST_Extrude is only 10s.

An example of enrichment on surface geometry is 3D Surface area calculation with ST_3DArea, and SQL script is provided in Appendix B.4.2.

More properties are created to test multiple properties attached by executing SQL scripts as follows. A seed is set for the random number generator by SELECT SETSEED(0.5).

```
-- Randomly generate construction year
UPDATE property SET construction_year = 1950 + FLOOR(RANDOM() * (2021 - 1950));

-- Random building type classification
UPDATE property SET type =
    CASE
        WHEN RANDOM() < 0.25 THEN 'Residential'
        WHEN RANDOM() >= 0.25 AND RANDOM() < 0.5 THEN 'Commercial'
        WHEN RANDOM() >= 0.5 AND RANDOM() < 0.75 THEN 'Industrial'
        ELSE 'Infrastructural'
    END;
```

4.2.3. Tileset organisation and tile creation

The procedure follows the tileset organisational method defined in Chapter 3.

Creating hierarchy table

Hierarchical k-means clustering is implemented using ST_ClusterKMeans. The 3D city models are clustered into different subsets on hierarchical levels. This hierarchical clustering creates a tree structure, and results in a set of leaf nodes on the bottom level. For the complete implementation, please refer to the SQL scripts in Appendix B.5.

This results in a table hierarchy. It shows how models are divided into small parts. The tileset structure can be organised based on this.

Creating a tileset view

After creating the hierarchy table, a query needs to be executed to create a view for the tileset JSON (see Appendix B.6). The view for the tileset.json relies on table hierarchy and table property, named vw_tileset. This view provides a tileset JSON that can be served directly from the database to the web server.

The view vw_tilest contains two columns, tileset_id (data type: integer) and tileset (data type: JSON). The tileset_id is the identification of the unique tileset to which tiles belong. The 3D Tiles database can store multiple tilesets.

The following explains the organisation of the tileset. These include the information used for defining metadata, bounding volume, geometric error, and refinement type.

The names of properties in the b3dm are associated with the tileset as a key/value pair, i.e. an object. The key is "properties", and the value is a JSON object. Within this object, each property's value is also stored as an object. In the web client, this tileset.properties object is checked, and a declarative styling is executed at runtime. For example, the 3D tiles can be dynamically styled based on the "Height" property, with different colours representing different height ranges.

Apart from declarative styling based on the properties, this enhances user interaction with semantic information. The query to select all of the properties stored in the table property is as follows, where 'OFFSET 2' is to exclude the primary key and foreign key in the table property:

```
SELECT column_name AS properties
FROM information_schema.columns
WHERE table_name = 'property'
ORDER BY ordinal_position
OFFSET 2
```

By default, tile number 1 is put on the root level. To keep the tile on the root level displayed on Cesium when it intersects the view frustum, the bounding volume on the root level is set as the minimum bounding box for all the objects in the tileset because its bounding volume must encompass the entire scene. Other tiles are as children tiles pointing to the root tile.

The bounding volume for a child tile is computed from the 3D extent of the clusters in the table hierarchy. The corresponding SQL is below, where the envelope is computed using ST_3DExtent(geom).

```
SELECT
ARRAY[
(ST_XMin(h.envelope) + ST_XMax(h.envelope)) / 2, -- centerX
(ST_YMin(h.envelope) + ST_YMax(h.envelope)) / 2, -- centerY
(ST_ZMin(h.envelope) + ST_ZMax(h.envelope)) / 2, -- centerZ
(ST_XMax(h.envelope) - ST_XMin(h.envelope)) / 2, 0, 0, -- halfX
0, (ST_YMax(h.envelope) - ST_YMin(h.envelope)) / 2, 0, -- halfY
0, 0, (ST_ZMax(h.envelope) - ST_ZMin(h.envelope)) / 2 -- halfZ
]
FROM hierarchy h
```

Regarding the geometric error, the geometric error for the tileset and root tile of temp_tid 1 (by default) is set to approximately half of the diagonal length of the root bounding volume, and the children tile is set to 0. Note that this is not a standard calculation formula but one based on experience. A conditional statement in SQL is as follows:

```
CASE WHEN h.temp_tid = 1 THEN
--half of the diagonal_length
ROUND(
sqrt(
power(ST_XMax(h.envelope) - ST_XMin(h.envelope), 2) +
power(ST_YMax(h.envelope) - ST_YMin(h.envelope), 2) +
power(ST_ZMax(h.envelope) - ST_ZMin(h.envelope), 2)
)::numeric/2,
2)
```

```
ELSE
0
END AS geometric_error
```

In our case constrained to single LOD, the refinement strategy is set as "ADD" ensuring the root tile is not replaced when adding children tiles to the scene.

4.2.4. Encoding of geometry and property

As explained in Section 3.1.2, the methods for serving 3D Tiles on the fly from the implemented database and variants of the approach are proposed to serve the b3dm from the database. The differences lie in whether the binary gltf and the properties are composed beforehand.

- For method 1, it only computes compulsory geometric and topological information, and stores it in the database. The composition of the b3dm is implemented in the web server.
- Method 2 is to compose binary gltf in the database and store it as a blob. The binary glTF is then combined with the queried property to generate the b3dm in the web server. This allows different properties to be provided in b3dm without regenerating the entire tile.
- Method 3 is to compose b3dm in the database and store it as a blob.

In approach 3, an additional step in this phase is to pre-compose b3dm, a Binary Large Object (BLOB) for multiple 3D models grouped into larger batches. The reason for this is to store the fully pre-computed b3dm to serve 3D Tiles from the database. An additional b3dm column is added to the hierarchy table. A built-in function to pre-compute b3dm inside the database is expected in future development, and this leads to the pre-composed b3dm column in a view instead of a table.

Another thing to mention is the correct link between the attributes and the geometry model in the query results. The b3dm stores a table batch table that describes the property information of the model. In the batch table, the property value is associated with a unique batch ID, allowing for the identification of the properties of each model. During the composition phase, a query to join table object and property on object ID is executed. The batch ID in a b3dm is based on the object ID in the table, ensuring a unique identification for each model.

4.2.5. Web server query and visualisation

Web server set-up

A web server is set up to connect with the database and serve 3D Tiles to the web client. In this phase, we define a Flask application factory function create_app that takes theme as a parameter specified in a JSON file. Inside the function, a Flask application instance is created, and route handlers (ui, index, cesium_ui, tiles_tileset, tiles_one_tile) are defined within the function. For further details, please refer to Appendix B.1 and GitHub: 3dtiles.

An example of a JSON file is provided as follows:

Parameters in the input.json are explained as follows:

```
{
"theme": {
    "description": "This is for test, around TU Delft Aula",
    "lod": "lod12_2d",
    "mode": 1,
    "cluster_number": [1,1],
    "index_flag": 1,
    "b3dm_flag": -1,
    "glb_flag": 1,
    "property": ["height", "building_type", "construction_year", "city"],
    "filter": ""
}
}
```

Table 4.3.: Example of a theme JSON Object in the input JSON file

- theme: It is used to control customised input, and this parameter is specified in the Flask API.
- lod: LOD of the dataset, one can specify from lod12_2d, lod12_3d or lod22_3d. This value and theme collectively determine the dataset used. If the lod is not supported yet, an exception will be raised.
- mode: The Flask API includes two parts, creator and server. If the mode is set to 1, the tile creator is called first, and then the server. If 0, only the server is called, which is used in case 3D Tiles have been prepared in the database.
- cluster_number: number of clusters on the top level and bottom level used for the hierarchical k-means algorithm.
- index_flag: 0 for non-indexed b3dm or binary glTF, 1 for an indexed one.
- b3dm_flag and glb_flag: 1 for composed, -1 for no composed. For example, if both index_flag and glb_flag are set to 1, an indexed binary glTF will be composed beforehand in the database.
- property: specify the property to query, property names given in a list. If the property is not supported yet, an exception will be raised.
- filter: This is used for attribute and spatial filters. For example: "and height $>$ 10". By default, it is set to "", meaning no filter is specified.

The Flask application loads this JSON file named input.json. Based on the provided theme, the application extracts relevant data from the JSON file and utilises necessary data type conversion. The extracted and processed customised values are then passed within the Flask routes to generate responses, allowing dynamic 3D Tiles selection based on custom requirements.

In this project, the database schema designed for 3D Tiles is emphasised. The web application is mainly used for visualisation and analysis. Thus, the prototype does not support

interactive spatial or attribute querying from the web client user interface. Custom queries are only defined in JSON files within the web server and parsed to the Flask API.

Attribute and spatial query

An attribute query can be performed. For example, the corresponding SQL for the above height filter is below, where maxz is the value of the parameter.

```
SELECT * FROM {obj_record}
WHERE height > {maxz}
```

Another example is volume property. The corresponding SQL to query the top 10 volume geometries with the largest volumes is as follows:

```
SELECT * FROM {obj_record}
ORDER BY volume ASC Limit 10
```

Spatial querying is enhanced by bounding box filtering. Firstly, a search region is created around a specific point. Next, a spatial filter is applied to check the tiles in the domain intersecting the region. During the experiment, a box buffer is created by extending 1 metre from the coordinate point (3921335 299904). Two methods are implemented. One directly checks all the bounding boxes on the bottom level, and the other is a hierarchical search. It first checks the top level, filtering the parent bounding box that intersects the specified region, and then checks its children. For details, please refer to Appendix B.7.1.

Web client visualisation

We implement declarative stying by defining a set of visualisation conditions based on the property in the Javascript code in the HTML. The colour and transparency of buildings vary according to their property values.

A 3D visualisation platform based on Cesium Sandcastle is utilised. It has an operation interface that supplements the 3D Tiles visualisation. An interactive editing box to set JS scripts for declarative styling is provided. The styling criteria can also be adjusted and recompiled on the Cesium application interface, allowing interactive visualisation on the web client side.

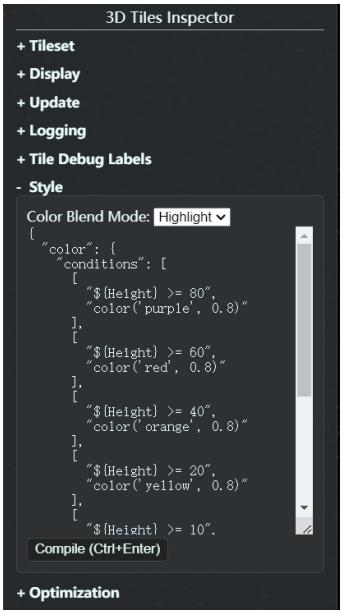
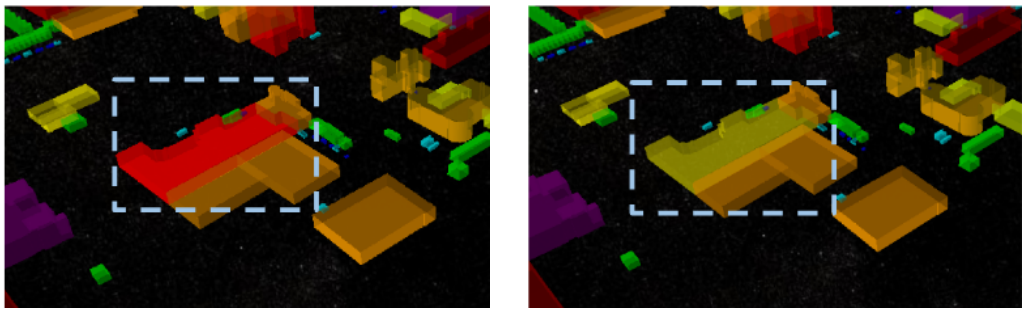


Figure 4.4.: JavaScript code box with online compiler support

Users can interactively select models on the web client with a mouseover. Each model is interactive in the scene and can be highlighted in yellow, as shown in Figure 4.5. When the object is clicked, all the properties that are stored in the header of the b3dm are displayed.



(a) model visualised based on property (b) highlight 3d model on mouse hover

Figure 4.5.: Models displayed with and without a mouseover

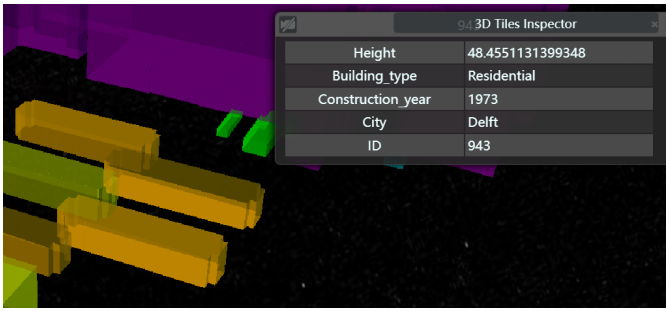
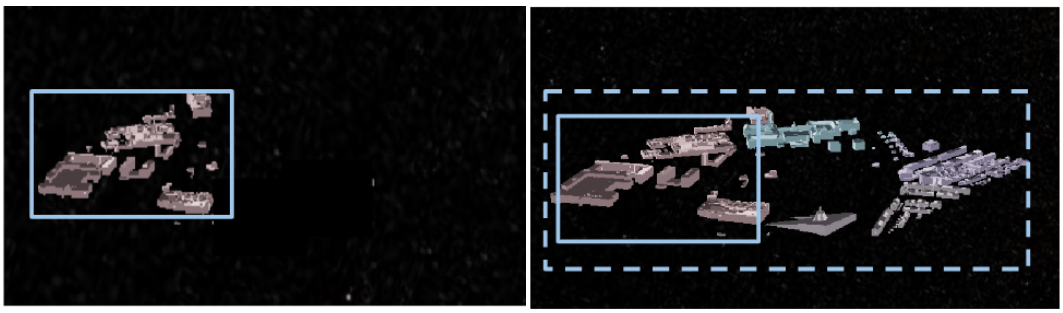


Figure 4.6.: Properties displayed as the user clicks the model

Another thing worth mentioning is that appearance colours are pre-configured in the binary glTF during the prototyping stage. This is for better debugging and result comparison, making the development process straightforward and controllable. The entire scene is streamed to the scene tile by tile. The construction of b3dm and the definition of the tileset JSON hierarchy are crucial factors that influence rendering behaviour and web performance. For example, at the bottom-left corner of Figure 4.7a, there are a few buildings loaded, and in Figure 4.7b, more buildings are loaded and the entire scene is displayed. These buildings are not rendered at the same time since they are encapsulated in different b3dm files. These b3dm files are referenced in different hierarchies in the tileset JSON file with different geometric errors.



(a) zoom out (b) zoom in

Figure 4.7.: The model progressively displayed when zoomed in

5. Results and Analysis

This chapter evaluates the design of the database storage and the performance of serving 3D Tiles to the web client. Tools and datasets used are presented in Section 5.1. Benchmarking concerning the storage system and web retrieval has been performed, and the results are discussed in Section 5.2.1 and Section 5.2.2. The tiling method is evaluated in Section 5.3. A case study regarding 3D Tiles creation enhanced by the DBMS approach is given in Section 5.4.

5.1. Tools and datasets

5.1.1. Test environment

Table 5.1 gives the system details used for the tests and benchmarks described in this chapter. The web server performance is particularly affected by GPU and memory.

5.1.2. Datasets

In Chapter 4, we implemented the 3D Tiles method using a small spatial extent dataset. As explained in 4.1.2, we use datasets from 3DBAG to test 3D Tiles serving from the perspective of the storage system and web retrieval, and datasets from PDOK to test the tiling method. The dataset description is provided in table 5.2.

Datasets of different sizes and characteristics are used. The differences are relevant to evaluate how the proposed 3D Tiles approach behaves in these use cases. An overview is given for these datasets in Figure 5.1. The Orange area represents Aula_LOD; the red area represents BK_LOD; orange and red together represent Campus_LOD; the blue area is for Delft_NE, the yellow area is for Delft_NW and the green area is for Delft.

OS	Windows 11 64bit
Processor	Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
Memory	16G
Graphics Card	Intel(R) UHD Graphics NVIDIA GeForce RTX 2060 with Max-Q Design
Browser	Chrome
Screen Resolution	1920*1080

Table 5.1.: System specifications

Name	Files	LOD	Geometry representation	Disk size (MB)	Description
Delft	4	LOD1.2	2D polygon	67.4	City Delft and surroundings
Delft_NE	1	LOD1.2	2D polygon	15.4	Northeastern part of City Delft
Delft_NW	1	LOD1.2	2D polygon	39.4	Northwestern part of City Delft
Campus_LOD1	3	LOD1.2	MultiPolygonZ	32.4	TU Delft Campus
Campus_LOD2	3	LOD2.2	MultiPolygonZ		
BK_LOD1	2	LOD1.2	MultiPolygon Z	15.13	BK and surrounding areas
BK_LOD2	2	LOD2.2	MultiPolygon Z		
Aula_LOD1	1	LOD1.2	MultiPolygon Z	17.3	Aula and surrounding areas
Aula_LOD2	1	LOD2.2	MultiPolygon Z		
Aula_Extrusion	1	LOD1.2	2D polygon		

Table 5.2.: Dataset description

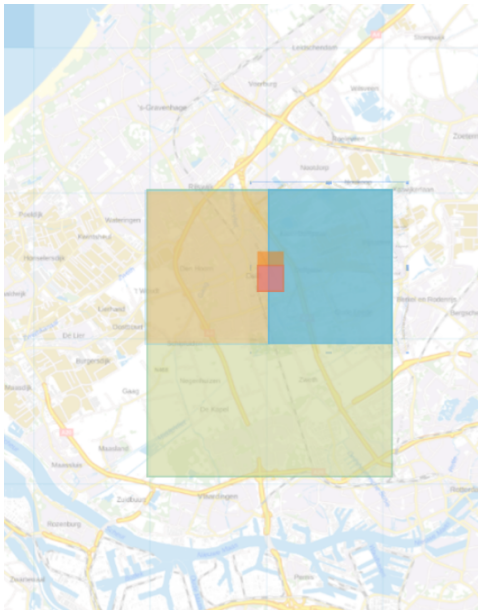
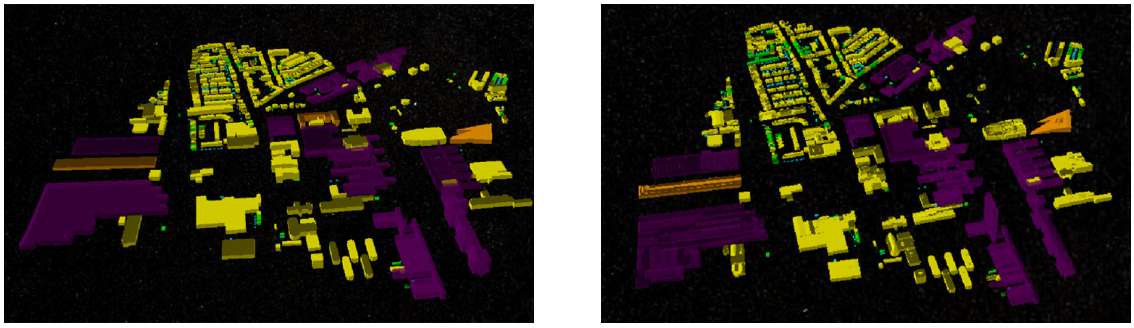


Figure 5.1.: Approximated projection of the extent of the used datasets

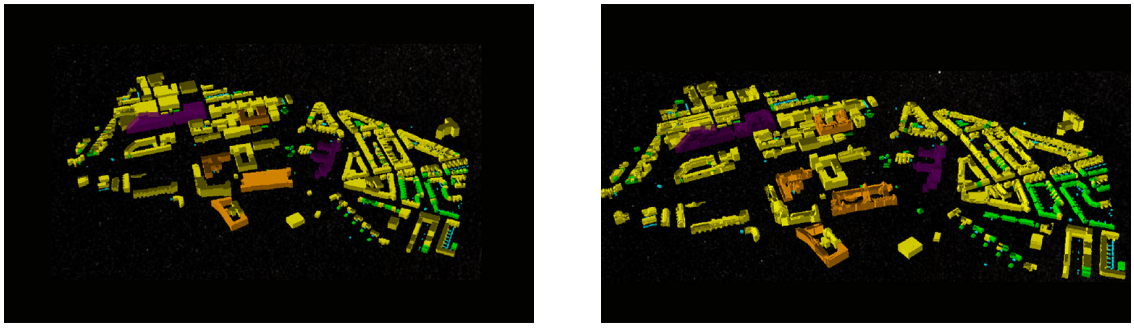
5.2. 3D Tiles Serving approaches

Benchmarking is implemented on the LOD1 and LOD2 3D building datasets collected from 3DBAG. To evaluate 3D Tiles serving approaches, we use unclustered data. This eliminates the effect of variation in the number and size of tiles. The b3dm contains height property. The visualization of the datasets based on height property is provided.



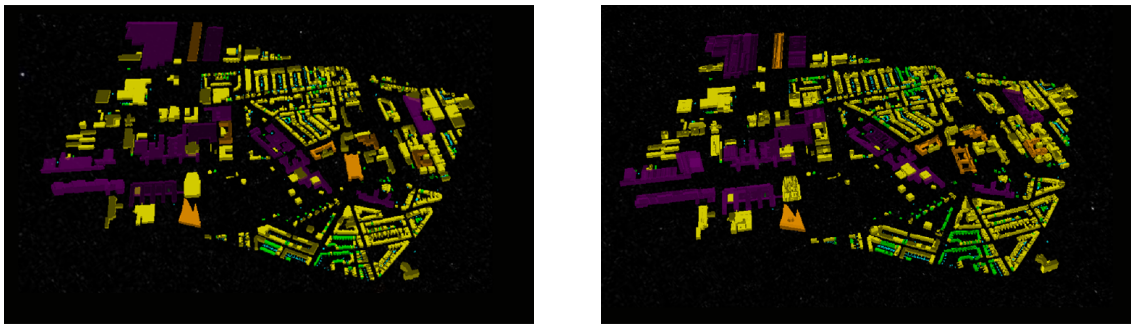
(a) Buildings of LOD1 (b) Buildings of LOD2

Figure 5.2.: Visualisation of dataset Aula_LOD



(a) Buildings of LOD1 (b) Buildings of LOD2

Figure 5.3.: Visualisation of dataset BK_LOD



(a) Buildings of LOD1 (b) Buildings of LOD2

Figure 5.4.: Visualisation of dataset Campus_LOD

5.2.1. Storage system

The database storage needed for each of the three proposed DBMS methods is computed to compare the three proposed approaches with the file-based approach. As explained in Section 5.1, we use unclustered data to compare and focus on the content(b3dm) in 3D Tiles. Thus, the tileset and hierarchy storage are not included. The table 5.3 presents the storage results for the datasets collected from 3DBAG. The file generated for comparison is a non-indexed b3dm. In a non-indexed B3DM file, each vertex of the 3D geometry is stored individually without any optimisation through indexing. This means that vertices are repeated for each triangle that uses them.

Dataset	Object & face tables	Property table	GLB in DB	B3DM in DB	B3DM File
Campus_LOD1	3.55MB	0.04MB	2.07MB	2.10MB	6.73MB
Campus_LOD2	8.88MB	0.04MB	5.36MB	5.39MB	17.30MB
BK_LOD1	1.62MB	0.019MB	0.93MB	0.95MB	3.18MB
BK_LOD2	4.12MB	0.019MB	2.45MB	2.47MB	8.04MB
Aula_LOD1	1.93MB	0.021MB	1.13MB	1.15MB	3.67MB
Aula_LOD2	4.76MB	0.021MB	2.90MB	2.92MB	9.25MB

Table 5.3.: Content size performance (non-indexed b3dm)

We also generate an indexed b3dm and make comparisons. In an indexed B3DM file, the vertices of the 3D geometry are stored once, and then the triangles are defined using indices that reference those vertices. The table 5.4 presents the storage results. The storage comparison is provided in Figures 5.5, 5.6, and 5.7. The four methods are summarised as follows:

Method 1: For the on-the-fly approach, the compulsory data are from object, face, and property tables. The geometric and property data and their associated identifiers are used to compute the database storage.

Method 2: For the precomposed geometry approach the compulsory data is glb and property.

Method 3: For the precomposed b3dm approach, the compulsory data is the binary b3dm.

Method 4: The b3dm file on the disk.

Dataset	Object & face tables	Property table	GLB in DB	B3DM in DB	B3DM File
Campus_LOD1	3.55MB	0.04MB	1.46MB	1.49MB	4.15MB
Campus_LOD2	8.88MB	0.04MB	3.78MB	3.82MB	10.50MB
BK_LOD1	1.62MB	0.019MB	0.66MB	0.68MB	1.88MB
BK_LOD2	4.12MB	0.019MB	1.73MB	1.74MB	4.91MB
Aula_LOD1	1.93MB	0.021MB	0.80MB	0.82MB	2.26MB
Aula_LOD2	4.76MB	0.021MB	2.06MB	2.07MB	5.65MB

Table 5.4.: Content size performance (indexed b3dm)

The results show that on-the-fly generation takes up less disk storage than the file approach. The pre-composed approaches take up less space. Geometry representation is reduced with a binary glTF representation. Another thing worth noticing is that the b3dm blob stored

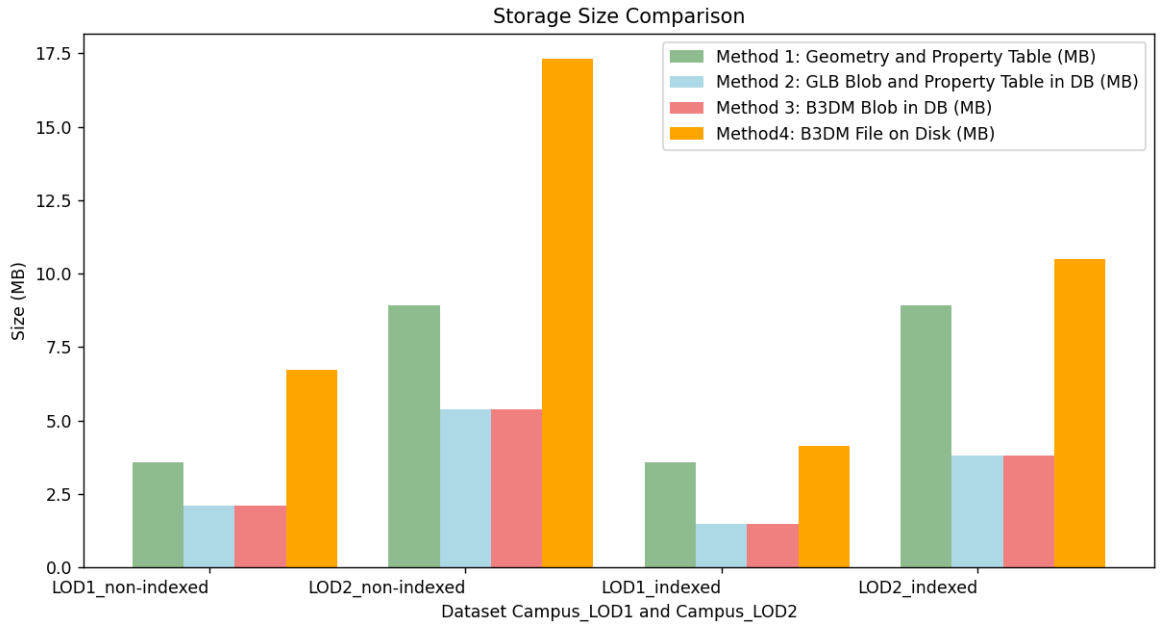


Figure 5.5.: Content size performance (Dataset Campus)

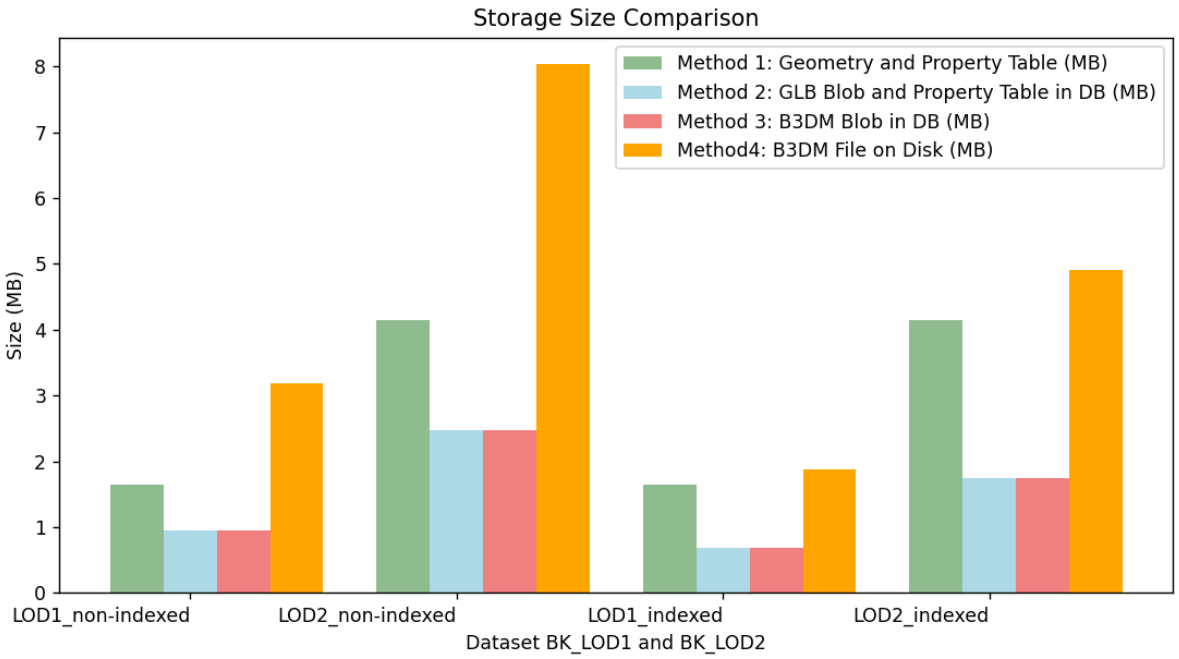


Figure 5.6.: Content size performance benchmark (Dataset BK)

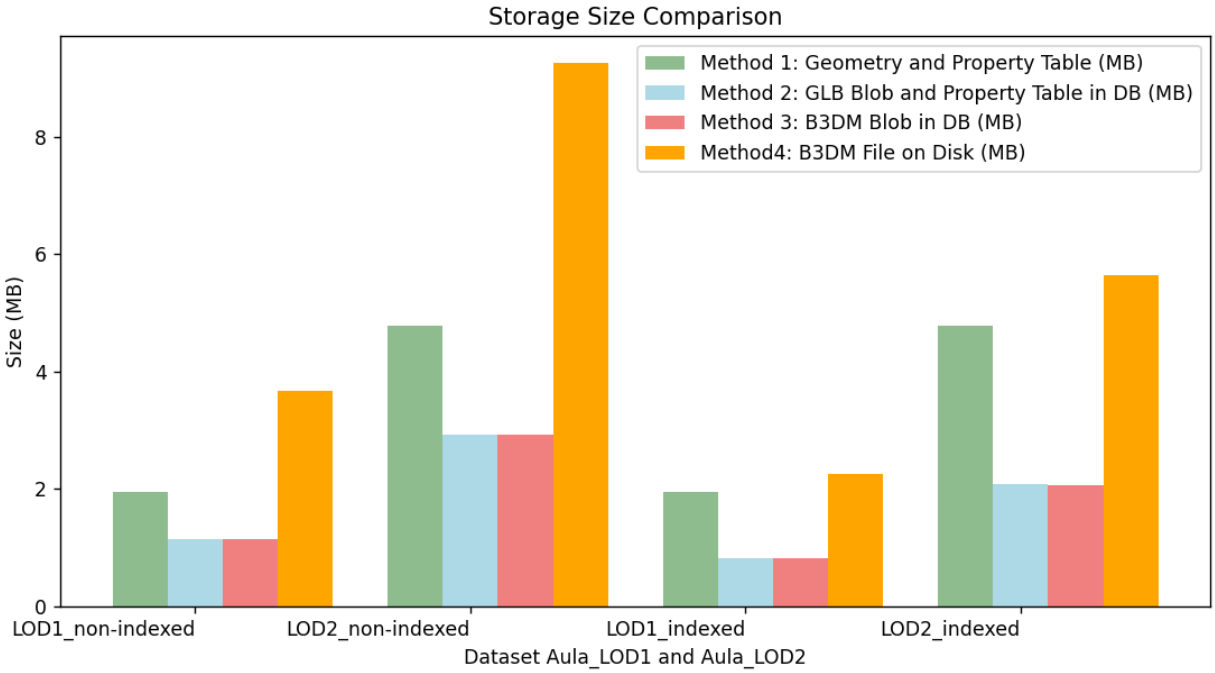


Figure 5.7.: Content size performance benchmark (Dataset Aula)

in the database is compressed and is approximately one third of the b3dm file representation.

5.2.2. Web retrieval

We compare precomposed methods with on-the-fly methods from the perspective of web retrieval efficiency and compare with a file-based system regarding efficient data access. The used datasets are Aula_LOD1 and Aula_LOD2.

We compare the DBMS approaches enhanced by a Flask API with a file system. The serving time is split into creating time and fetching time. The triangulation method is an adaptive ST_Tesselate. The property in the b3dm is height. The results are provided in Table 5.5.

For the on-the-fly approach (Method 1), the creating time is mainly calculated from the total time spent on computing properties, normals, positions, and indices. For the precomposed geometry approach (Method 2), the increased creating time is composing a glb. For the precomposed b3dm approach (Method 3), the increased creating time is composing a b3dm. Method 4 is serving the generated b3dm file from the disk.

In comparing the fetching time of the three DBMS methods, the precomposed methods (method 2 and method 3) outweigh the on-the-fly method (method 1). The precomposed b3dm method (method 3) is the fastest. The total execution time for the three DBMS methods is similar.

The file system's retrieval efficiency performs better than that of a DBMS, as shown in Fig-

	Method	Property(s)	Normal(s)	Triang-ulation(s)	Indices(s)	Blob	Total creating time(s)	Total fetching time(s)	Total serving time(s)
LOD1	1	0.009	2.336	20.929	12.871	0	38.350	4.686	43.036
	2					3.039	41.389	0.147	41.536
	3					3.637	41.987	0.043	42.03
	4	file stored on the disk						0.002	0.002
LOD2	1	0.009	5.932	69.991	80.672	0	161.731	10.250	171.981
	2					8.520	170.251	0.375	170.626
	3					8.758	170.489	0.101	170.59
	4	file stored on the disk						0.006	0.006

Table 5.5.: Time performance for serving one tile

ure 5.8. However, the file system is impacted by the file size, which indicates the burdensome on the disk in the case of a larger dataset. To further prove this, more datasets need to be collected.

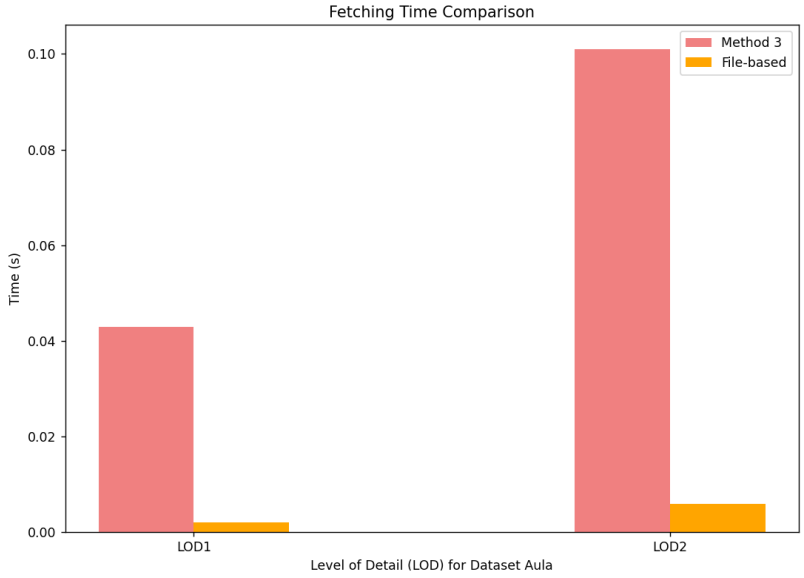


Figure 5.8.: Time performance for fetching one tile

Regarding the time from 3D Tiles fetching completion in the web server to the display on the web application, it is tricky and out of expectation. The two precomposed methods show differences. Displaying precomposed b3dm on the web application takes an unexpectedly longer time, while precomposed binary glTF does not face this problem. Although the glTF and b3dm are stored compressed as blobs, the glTF is uncompressed during the b3dm composing phase in the Flask prototype while b3dm is not.

5.3. Tiling method

5.3.1. Bounding box filtering time performance

We use clustered data to test the tiling method, and the dataset is Delft_NW and Delft collected from PDOK. The cluster number settings and corresponding query timing are

provided in the table. Note that because of the limited performance of the adaptive triangulation approach in the prototype, the triangulation method is simplified to the combination of ST_Tesselate for the bottom and top face while ST_Delaunay for the others.

The table shows the performance result of filtering the tile that intersects with a search region (ST_Buffer(ST_GeomFromText('POINT(3921335 299904)'), 1)), with two methods. The first method is to check all the bounding boxes directly on the bottom level. The second method is hierarchical search, as explained in 4.2.5.

Dataset	Subdivision		Total tile number	Method I timing (s)	Method II timing(s)
	level1	level2			
Delft_NW	8	8	64	0.121	0.159
	8	32	256	0.273	0.317
Delft	8	8	64	0.125	0.181
	8	32	256	0.289	0.392

Table 5.6.: Bounding box filtering time performance

From tests with other search regions, it is noticed that there are also scenarios where the area of interest is located in more than one tile. It is also possible that the query returns none, as the search region lies within the bounding volume on level 1 but out of the bounding volume on level 2.

Method I show an advantage over method II from the dataset tested. An analysis of the possibility of applying method II to a larger dataset is presented in 6.2.

5.3.2. Cluster distribution performance

The following experiments are based on the dataset Delft_NE of LOD1 extrusion and aim to quantify the clustering distribution of hierarchical k-means and find how tiling methods affect the overall streaming performance.

Utilising an HTTP request to stream multiple models within the same tile improves efficiency because each tile issues one draw call. The dataset used is Delft_NE, and the objects are clustered into different cluster numbers. This aims to evaluate the tiling method and find how tile size and number affect the overall performance. The table 5.7 shows the tile size, tile number(cluster number) and fetching time.

Total fetch timing	Tile number	Tile Size			
		Min	Max	Medium	Avg
18 s	15	7 KB	1333 KB	512 KB	581 KB
11 s	10	202 KB	1592 KB	890 KB	870 KB
8 s	5	694 KB	2798 KB	1576 KB	1738 KB
19*s*	3	1475 KB	4848 KB	2363 KB	2895 KB

Table 5.7.: Relationship between the tile size and serving time

The variation between tile sizes is big, resulting in different transmission times per tile, thus affecting the overall transmission performance. It can be seen from Figure 5.9 that as the tile number increases, the overall loading time is reduced. However, a turning point

occurs when the tile number continues increasing. The tiles that intersect with the bounding volume and meet the maximumScreenSpaceError will issue draw calls. The tests suggest that, for the experiment tileset, the optimal number of tiles is around 5, and the single tile size is around 2 MB. This test is not conducted by directly measuring or monitoring memory, CPU, or GPU usage.

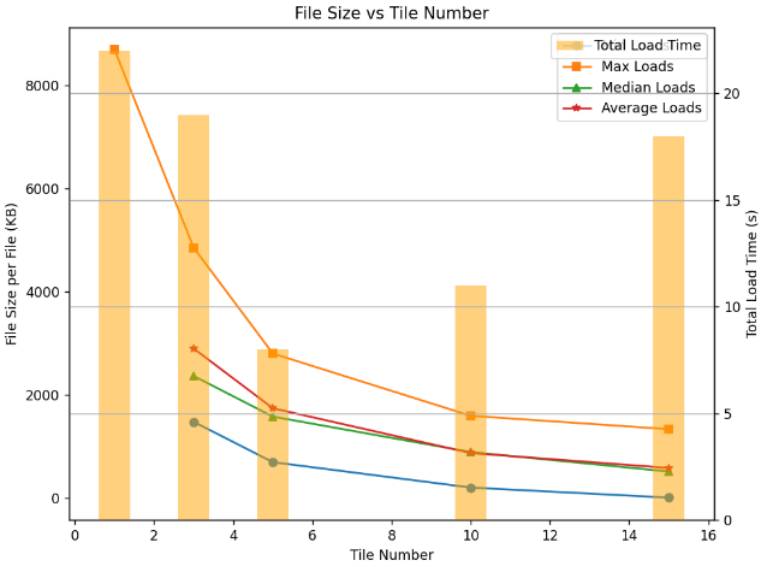


Figure 5.9.: Relationship between file size and fetching time

5.4. Case study

A case study is provided to show how the 3D Tiles approach can be applied in a specific use case as proof of the applicability of the on-the-fly approach for dynamic serving 3D Tiles.

5.4.1. Campus Emergency Evacuation—Sea Level Rise

Extreme events are becoming increasingly frequent. In this situation, rapid decision-making is critical to reduce risk and ensure the safety of individuals and communities. In this case study, we explore how the 3D Tiles approach facilitates emergency evacuation decision-making during a hypothetical sea level rise event. We want to answer the question of where would be the ideal temporary emergency evacuation centre when the sea level rises to 10m.

The goal is to identify and visualise temporary safe destinations on the campus in the event of rising sea levels. The property building height is relevant to this event. An attribute query is executed to find the ideal place for a temporal evacuation centre. The 3D Tiles database for the Campus has been created and is ready to serve. The implementation is on the Aula_Extrusion dataset.

Attribute query

The following visualisations are served from the database with a SQL attribute filter in each case, the returned buildings are considered as potential safe destinations.

The attribute query is explained together with the usage of input.json in Flask API.

First, we retrieve buildings whose height is less than 10 meters. The settings in input.json are displayed in table 5.8.

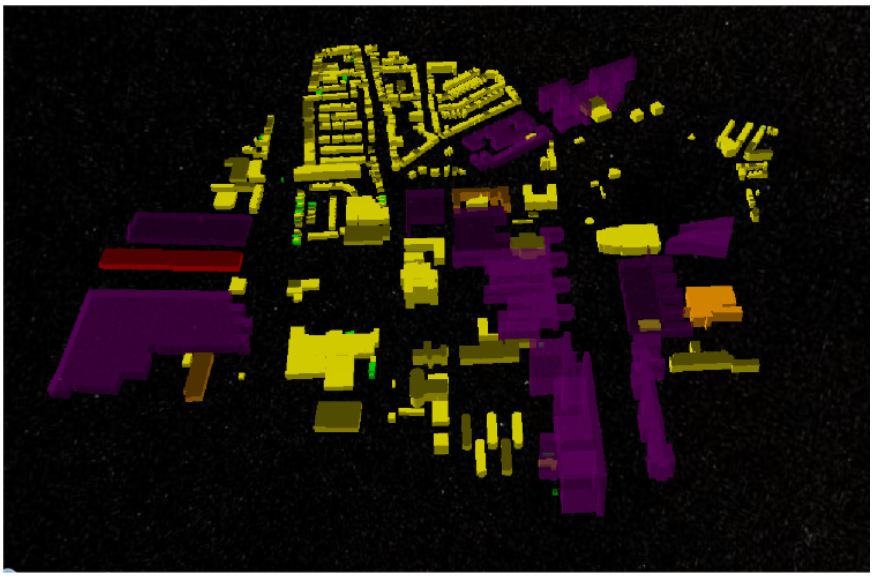
```
{
"theme": {
    "description": "Dataset round TU Delft Aula",
    "lod": "lod12_2d",
    "mode": 1,
    "cluster_number": [1,1],
    "index_flag": 0,
    "b3dm_flag": -1,
    "glb_flag": -1,
    "property": ["height"],
    "filter": "and height > 10"
}
}
```

Table 5.8.: input.json file settings

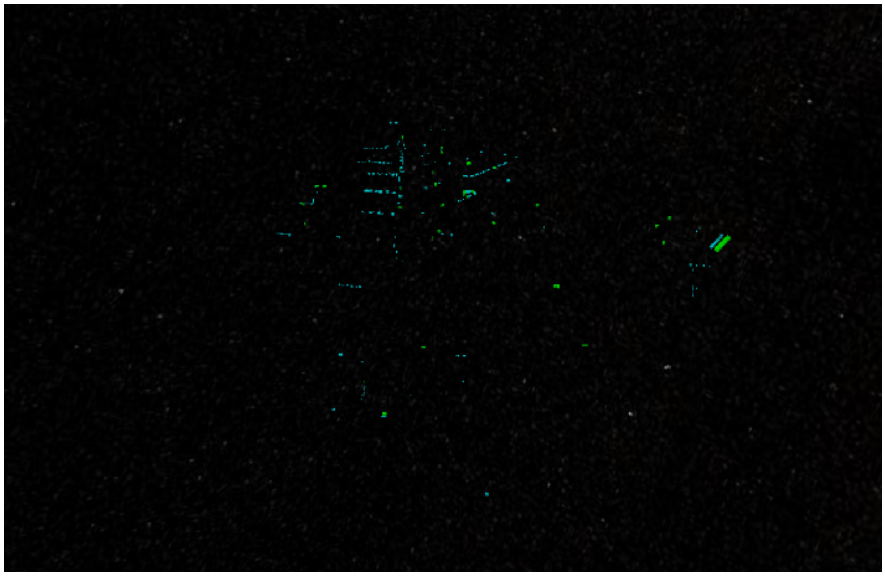
Next, we retrieve buildings whose height is more than 10 meters. The filter in the input.json is changed to:

```
"filter": "and height <= 10"
```

Visualisation of the case study is presented in Figure 5.10.



(a) Buildings above the water



(b) Buildings under the water

Figure 5.10.: Visualisation of campus underwater(a) and above water(b)

6. Conclusion and Future Work

Chapter 6 concludes the research performed. The Section 6.1 gives discussion and conclusions derived from the analysis of the obtained results, and the future work is discussed in 6.2.

6.1. Conclusions and Discussion

Section 6.1 answers the research questions introduced in Section 1.3 and gives conclusions. Section 6.1.2 summarises the overall contribution. In Section 6.1.3, limitations of the proposed methodology are presented.

6.1.1. Research Questions

Firstly, answers to sub-questions are provided. Subsequently, the section focuses on addressing the main research questions. Finally, the main findings and conclusions drawn from addressing these research questions are summarised.

1. **How to organise the database storing raw data, for example, storing raw geometries as polygons, multi polygons or polyhedrons?**

   The geometry type of raw data is harmonised, with expanding the geometry types (multi polygon, polyhedral surface) to polygons. Geometric operation on face level reduces complexity compared with volume level. Adhered to a Cesium WGS84 globe, all geometries are reprojected to a unified coordinate reference system (CRS). This allows managing data from multiple sources with different CRS using a standard and consistent workflow.

   The polygons are further harmonised into positions and normals, which fit into the final deliverable data format b3dm.

2. **How to define the mapping rules for storing 3D Tiles in a relational database?**

   - 3D coordinates of each object body are stored as an array of coordinates (nodes). References to faces at the lower level are maintained through object IDs. Properties are linked to the corresponding object.
   - The hierarchy supports object clustering based on minimum bounding regions on different hierarchical levels. Thus forming different tile sizes.
   - Regarding storing the precomposed glb and b3dm, there is a balance between database storage and web retrieval. The results show that precomposed b3dm or binary glTF largely reduces the fetching time.

3. **How to derive meshes (triangulated geometries) from raw data?**

   The algorithm adapted from ST_Tesselate triangulates both convex and concave surfaces correctly. This method fits both LOD1 and LOD2 models.

   Whether to maintain or drop raw geometries depends on the use case. It is suggested that the geometry be dropped if there is no need to compute new generic properties from it or if geometry rarely needs to be reconstructed from the topology.

4. **How to avoid potential problems such as data redundancy, and data inconsistency?**

   Regarding data redundancy:

   - Tile organisation is stored in view, relying on the table hierarchy.
   - Topology models avoid redundancy and maintain consistency. Triangulated face topology is maintained on face level. Indices of the triangulated faces are stored as an array of indices, referencing vertices coordinates in an object.
   - A POSTGIS geometry can be reconstructed from nodes and faces. For visualization purposes, this method provides a compact storage model.

   Regarding data inconsistency:

   - The table face and property are linked with the object table with a foreign key referencing the object ID.
   - Furthermore, objects assigned to the tile are associated with the table object. This establishes a one-to-one association between each tile content and each object. This is insufficient for multiple representations, where a feature identifier must be introduced.

5. **How to define the tiling rules?**

   - Tiling rules can be defined based on spatial relationships or properties. These rules organize objects into different tiles.
   - We can organise tiles with a cluster algorithm. The tested algorithm is hierarchical k-means with user-defined cluster numbers. The 3D models are partitioned using a "top-bottom" clustering. The tiling process organises the data in a hierarchical structure based on a multi-scale partition of the dataset. Objects assigned to the cluster on the same level are combined into one tile.
   - The tile number and tile size should be balanced. If there are too many tiles, it will issue many draw calls. If the tile size is too large, there is a problem with the 'uncertain' web transmission. This suggests improving web performance by incorporating appropriate spatial index tuning. Delivering a tile with a super large size is not suggested.

6. **What kind of spatial and attribute queries could be performed based on the proposed data model?**

   - 2D and 3D computation can be implemented on geometries and stored as properties. These properties can then be used to generate 3D tiles.
   - Validation inside the database is tricky for functions supported by PostGIS and SFCGAL.

- It is easy to implement 3D computation with SFCGAL on LOD1 geometries because a valid polyhedron is guaranteed.
- However, this is problematic for LOD2 buildings because of the failure to build a polyhedral surface from Multipolygon Z.
- Precision should be paid attention to. ST_transform is not accurate. CRS transformation affects precision.

7. **What are the advantages and disadvantages of generating 3D Tiles on the fly compared to a file-based approach?**

   The on-the-fly approach is more flexible than the file-based approach. Geometric information and attributes are only stored in the table and fetched to compose a b3dm together when needed. However, creating 3D Tiles on the fly takes a long computation time.

After answering the sub-question above, the main research question can also be answered: ***How to compactly store both geometries and attributes in a database and efficiently serve data that complies with 3D Web standards to the client?***

The database tailored for 3D Tiles ensures compatibility and adherence to standardised formats. The database is further connected to the web server, facilitating the retrieval and streaming of 3D geospatial data directly to the web application. Tiling objects into clusters and streaming multiple models in several tiles improves efficiency.

In the database, a face-object approach stores the triangulated topology and unique coordinates of the object. Compared with a file-based approach, the on-the-fly and precomposed approaches take up less disk storage. Instead of producing a b3dm file on the disk, it calls memory and generates content on the server side.

Regarding fetching time, pre-composed approaches outweigh the on-the-fly approach. Both are much slower than serving from disk. However, the DBMS approach maintains data consistency and offers more flexibility to users. It is fair to believe that on-the-fly is a promising approach with the generation process improved.

6.1.2. Contribution

The 3D Tiles approach provides a way to serve 3D Tiles directly from the database. It reduces the generation of data copies and helps ensure spatiotemporal consistency.

It enhances the dynamic generation of a subset of the data model stored in the database with spatial and attribute filters. This solves the issue of 3D Tiles, which contains many fixed-divided files, making it difficult to deliver a subset of the data.

Method 2 (precomposed geometry) can serve b3dm with different properties on the fly, ensuring its flexibility in dynamically serving different attributes. The geometric information in the database is stored as a binary glTF and generated b3dm dynamically. The binary geometry representation (glTF) is computed once but can be used many times.

6.1.3. Reflection and discussion

The main criticisms of the prototype developed are as follows:

Processing raw geometry to obtain triangle topology is slow:

- ST_Tesselate is chosen to handle concave faces. However, this is sometimes a waste of CPU computation because handling convex surfaces with a sophisticated algorithm is unnecessary.
- A flag for concave surfaces should be introduced, which can help save CPU time and result in fast triangulation.

Data transfer between Python and Database:

- The prototype is lacking of native database functionality. Data transfer back and forth between Python and PostgreSQL occurs (see Section 6.2)

Property enrichment and filter only at the object level:

- The generic attribute in the database is limited to object property, and so is the filter implementation. The filter should also be applied to the face level or vertex level.

The tiling methodology does not ensure an even distribution and results in large variations in different clusters:

- Hierarchical k-means tiling does not ensure an even distribution. An improved k-means or other clustering method is expected. The indexing and clustering method should be improved (see Section 6.2).

The BLOB storage in the table is redundant.

- As discussed in Section 6.1.1, However, b3dm or binary glTF stored in the table is redundant because geometric information is stored twice.
- An alternative is to represent this binary content in a view instead of storing the content column in the hierarchy table, where a fully native database approach is needed (see Section 6.2).

The validity of a polyhedron is not ensured:

- Faces are expected to keep a counterclockwise orientation when viewed from the outside of a three-dimensional enclosure.
- At the face level, the orientation is correct. The orientation test shows a consistency between the polygon and the triangulated geometries. However, there are still issues with orientation at the body level.
- It is also suggested that triangulating geometries with external libraries and importing triangulated geometries to databases be considered (see Section 6.2).

Floating-point error issue is not solved.

- Precision is not confirmed. A set of geometric operations introduced floating errors, leading to inconsistency between the resulting coordinates and the raw geometry. Thus, the geometry is less accurate than the raw geometries. Use cases relying highly on geometric accuracy cannot be implemented.
- During the CRS conversion process, floating point errors are introduced. The transformation can be implemented at run time (see 6.2).

6.2. Future work

We would recommend the following directions for future exploration.

6.2.1. Native database functionality

In my prototype, geometric processing has been done with the use of SFCGAL and PostGIS functions, including normal computation, triangulation and some property calculations. The b3dm composition is still performed outside the database. The prototype first queries from the database to Python and then composes the queried information into a binary b3dm in Python; finally, it transfers the binary data from Python to the database and updates the b3dm field in the table hierarchy.

In addition, during the 3D Tiles generation process in the prototype, data transfer back and forth between Python and PostgreSQL occurs. The prototype first queries the triangulated meshes from PostgreSQL to Python, and then finds unique coordinates and triangulation topology of the object in Python, finally transfers the data from Python to PostgreSQL to update the nodes field in the table object and tri_node_id in the table face.

This causes expensive input/output (I/O) operations, which can minimise unnecessary data transfer. A b3dm composition function within the database is to be developed, supporting the creation of a set of views representing 3D Tiles.

A postgresql-plpython extension can be a solution. With enabling "plpython3u", the NumPy module is expected to work properly. Otherwise, implementing it in C++ would be ideal, considering there are more existing works around implementing C++ with PostgreSQL that lay a foundation, such as SFCGAL.

In addition, storing node coordinates, triangle indices, and normals as bytes instead of arrays in the database can be further explored. If the native database functionality is realised, it is also suggested to further explore the possibility of loading views into memory as a cache database.

6.2.2. Improving indexing and clustering method

The bounding box filtering (hierarchical search) results presented in Section 5.3 show the potential of the hierarchical tiling method when applied to larger data sets, such as national datasets. However, K-means supported by Postgres does not ensure an even distribution of objects in the cluster, resulting in tile size variations. It is suggested that a native database function, such as R-tree, be further implemented. In addition, SFC clustering supports access to the available clusters(tiles) regardless of hierarchical levels. Mapping R-tree to 1D with a Hilbert curve is worth exploring.

Based on this, a larger dataset can be organised as tilesets refer to another tileset. If a national dataset visualisation is needed, one can organise a tileset containing smaller tilesets.

In addition, because the binary glTF asset in 3D Tiles is a mesh. The b3dm size is not only affected by the total number of objects in a batch, but also related to the number of triangles. A complex shape can result in a larger data size. A clustering based on triangle numbers instead of object numbers can be considered.

6.2.3. Investigating refined LODs

Currently, the design of 3D data representation typically involves a single level of detail (LOD). This also lays a foundation for future development. Future iterations can be built upon this simplified framework, and the schema can be extended and adapted to incorporate models at multiple LODs.

One approach is maintaining data at different LODs directly in the database. Another approach is parsing the storage LODs from the view of topology, as depicted in Figure 6.1.

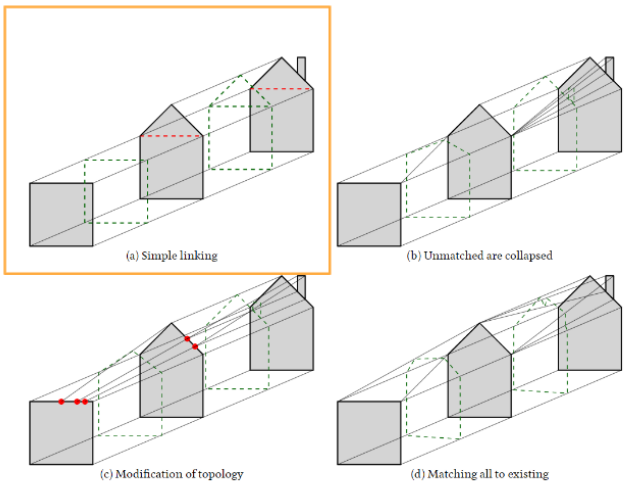


Figure 6.1.: The four linking schemes for three LODs of a house, here depicted in 2D. The objects that would obtained by slicing between the LODs can be seen in dashed green contours; the red dashed lines reflect the cells that need to be added and split in order to ensure a valid 3D (2D+LOD) cell complex [Arroyo Ohori, 2016]

A starting point can be investigating simple linking and matching between LODs and utilising sparse storage in the database, as indicated in Figure 6.1(a). This approach not only decreases storage requirements but also enhances retrieval efficiency.

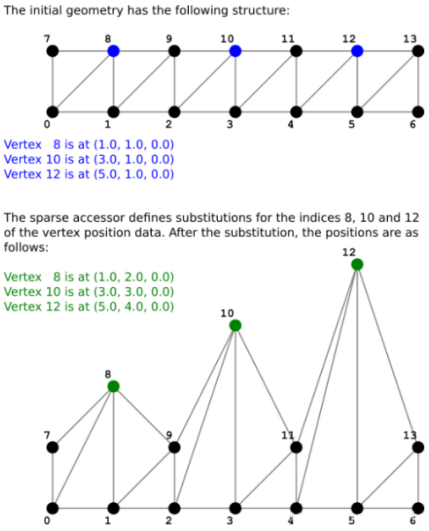


Figure 6.2.: sparse indexing in glTF [KhronosGroup, 2021]

By establishing hierarchical connections between LODs within the database, detailed LODs can reference coarse LODs where common data is stored. This is also aligned with sparse indexing in glTF. This hierarchical structure enables shared information storage at coarse LODs, reducing storage space.

This can be incorporated with the 'ADD' and 'REPLACE' refine strategy in 3D Tiles. Users can seamlessly navigate between different LODs as needed. Note that if the building is divided into multiple parts, there are potential texture seam issues with the UV maps of adjacent faces in the textured model.

6.2.4. Collaborating with more 3D data formats

Apart from 3D city models, more data formats are used to represent the physical world. One of the examples is Industry Foundation Classes (IFC) formats, an open file format used by BIM programs in the architecture, engineering, and construction (AEC) industry. The BIM model is usually big. Thus, a typical way is to decompose it into pieces to make this large asset possible to stream.

A typical conversion approach from IFC to 3D Tiles consists of four key steps, as depicted in Figure 6.3. They are IFC decomposition, IFC to OBJ conversion, OBJ to glTF conversion, and glTF to b3dm conversion. The binary glTF can be directly stored in the 3D Tiles database.

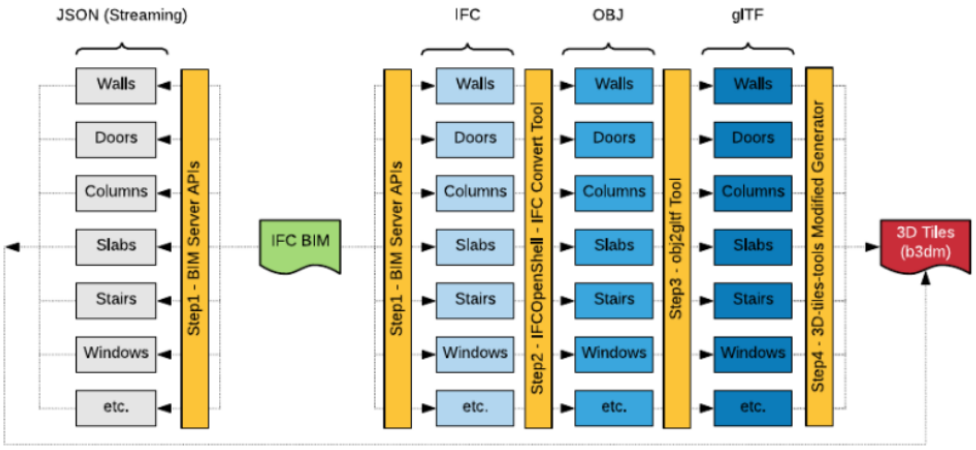


Figure 6.3.: An example of IFC to 3DTiles conversion workflow [Chen et al., 2018]

Each of the BIM model components can be stored in the database as a small binary glTF file. When merging glTF files to a single constituent or multiple b3dm files is needed, such as merging all the walls on the ground floor. SQL operations easily support this with a GROUP BY Statement based on the properties.

Regarding BIM, it is an asset with owners, which can be government utilities, private owners or a sharing ownership. The ownership problem should be handled during the BIM dissemination, and this can be done by managing roles and access permissions. For example, the view represents the agreed-to-share data, and the tables containing the whole information can not be accessed without permission.

6.2.5. Generating standard 3D Tiles on the web application

The loading of massive 3D data is limited by browsers and the network bandwidth. The smaller the page, the faster it loads. In web transmission, the binary glTF takes a large portion of the data size of 3D Tiles. In the built environment, ideally, there are primitives in LOD1, which are irregular but flat surfaces in LOD2 where many Delaunay triangles share the same normal. These result in many duplicated normals in the file.

Directly delivering many 3D geometries in the format designed based on graphics programming would be burdensome for network transmission in some cases. Instead of standard glb, a possible way to deliver the geometries is to remove normals and reuse positions to reduce the file size. Khronos supports the above format, but Cesium has not yet supported it.

An alternative and direct way is that the system delivers tileset JSON sent from DB, buffered geometric data will be sent to the web application when tiles are called, and generates glb data on the client side. This requires more processing time on the web client side. However, techniques such as WebAssembly may help improve the processing on the web side. The bottleneck of web transmission requires more effort to overcome than processing on the web side.

6.2.6. Coordinates transformation

In the prototype, the vertices in the binary glTF are stored as global coordinates. Alternatively, 3D global coordinates can be placed in local object space, and the transformation matrix can be stored, as shown below. In addition, coordinate reference system transformations can be applied at the run time.

```
Structure:          local transform     global transform
root                R                   R
 +- nodeA           A                   R*A
     +- nodeB       B                   R*A*B
     +- nodeC       C                   R*A*C
```

6.2.7. Adapting to 3D Tiles 1.1

In 3D Tiles 1.1, the content is a glb instead of b3dm. Existing creation or processing tools (e.g. 3D modelling software, validators, optimizers) have better compatibility with glTF assets. For example, converting IFC to glTF assets. This makes the 3D Tiles DBMS approach comply with the newly involved standards 3D Tiles 1.1 and can more easily provide support for directly serving 3D Tiles for use cases that rely on glTF.

6.2.8. Interoperability with other existing databases and web clients

It is worth exploring extensions that support the generation of views representing 3D Tiles that rely on the developed databases, such as 3DCityDB and multiple digital twin databases. In addition, further integration with more web clients, such as the Unity Web platform, is recommended. This extends support for spatial information access. The overview is depicted in Figure 6.4.

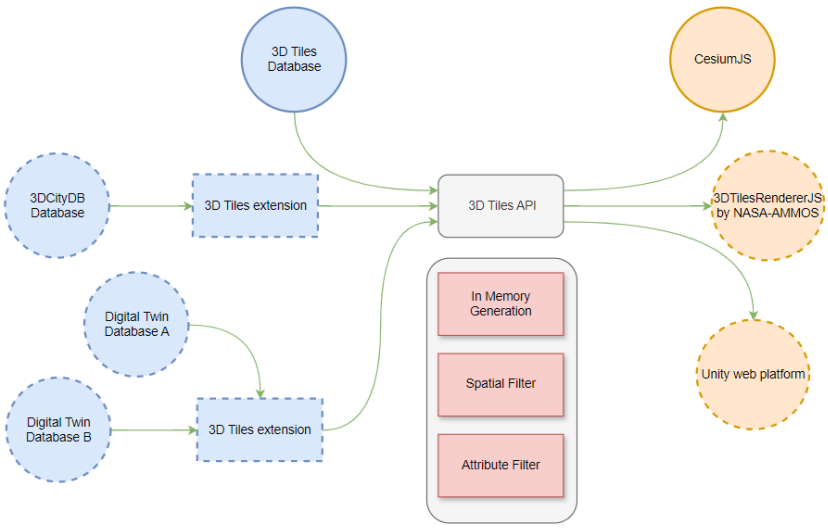


Figure 6.4.: System Outlook Overview

A. 3D Tiles example

A.1. Bounding volume example

```
boundingVolume : {
    "box" : [
        0 , 0 , 10, // Center
        20 , 0 , 0, // x-axis
        0 , 30 , 0, // y-axis
        0 , 0 , 10 // z-axis
    ]
}

boundingVolume : {
"region" : [
-1.319700, // West
0.6. // South
-1.33, // East
0.6, // North
0.0, // Min height
20.0 // Max height
]
}

boundingVolume : {
"sphere" : [
1 , 1 ,1 , // Centre
100 // Radius
]
}
```

A.2. Tileset JSON example

```
{
    "asset": {
      "version": "1.0",
      "tilesetVersion": "1.2.3"
    },
    "extras": {
      "name": "Sample Tileset"
```

```
      },
      "properties": {
        "id": {},
        "Height": {}
      },
      "geometricError": 200, //This is tileset Geometric Error
      "root": {
        "boundingVolume": {
          "region": [
            -1.3197209591796106,
            0.6988424218,
            -1.3196390408203893,
            0.6989055782,
            0,
            88
          ]
        },
        "geometricError": 200,//This is root Geometric Error
        "refine": "ADD",
        "content": {
          "uri": "parent.b3dm",
          "boundingVolume": {
            "region": [
              -1.3197004795898053,
              0.6988582109,
              -1.3196595204101946,
              0.6988897891,
              0,
              88
            ]
          }
        },
        "children": [
          {
            "boundingVolume": {
              "region": [
                -1.3197209591796106, //West
                0.6988424218, //South
                -1.31968, //East
                0.698874, //North
                0, //Min height
                20 // Max height
              ]
            },
            "geometricError": 0,//This is leaf node Geometric Error
            "content": {
              "uri": "ll.b3dm"
            }
          }
        ]
```

```
58        }
59    }
```

B. Code description and SQL statements

B.1. Flask code structure

The Flask application factory function named create_app with a parameter theme is provided as follows:

```
def create_app(theme):
    app = Flask(__name__)

    @app.route("/Cesium-1.110/<path:name>")
    def ui(name):
        print("ui route")
        return send_from_directory("Cesium-1.110", name, as_attachment=False)

    @app.route("/")
    def index():
        print("Index route")
        return send_file("static/index.html")

    @app.route("/ui/")
    def cesium_ui():
        print("cesium_ui route")
        return send_file("static/cesium_ui_server_map.html")

    @app.route("/tiles/tileset.json")
    def tiles_tileset():

        # Fetch tileset JSON from the database

        contents = tileset_json
        return jsonify(contents)

    @app.route("/tiles/<string:tile_name>.b3dm")
    def tiles_one_tile(tile_name):

        # Fetch and compose b3dm from the database

        response = Response(b3dm_bytes, mimetype="application/octet-stream")
        return response

    return app
```

This Flask application can be created with the specified dataset theme (eg: ”9_284_556”) and called as follows:

```
theme = "9_284_556"
app = create_app(theme)
app.run(debug=True)
```

B.2. Normal computation

```
    CREATE OR REPLACE FUNCTION ST_Subtract(p1 geometry, p2 geometry)
    RETURNS geometry AS
    $$
    DECLARE
        x1 numeric := ST_X(p1);
        y1 numeric := ST_Y(p1);
        z1 numeric := COALESCE(ST_Z(p1), 0.0);
        x2 numeric := ST_X(p2);
        y2 numeric := ST_Y(p2);
        z2 numeric := COALESCE(ST_Z(p2), 0.0);
    BEGIN
        RETURN ST_SetSRID(ST_MakePoint(
            x1 - x2,
            y1 - y2,
            z1 - z2
        ), ST_SRID(p1));
    END;
    $$
    LANGUAGE plpgsql IMMUTABLE;


    CREATE OR REPLACE FUNCTION ST_CrossProduct(p1 geometry, p2 geometry)
    RETURNS geometry AS
    $$
    DECLARE
        a1 numeric := ST_X(p1);
        a2 numeric := ST_Y(p1);
        a3 numeric := COALESCE(ST_Z(p1), 0.0);
        b1 numeric := ST_X(p2);
        b2 numeric := ST_Y(p2);
        b3 numeric := COALESCE(ST_Z(p2), 0.0);
    BEGIN
        RETURN ST_SetSRID(ST_MakePoint(
            a2 * b3 - a3 * b2,
            a3 * b1 - a1 * b3,
            a1 * b2 - a2 * b1
        ), ST_SRID(p1));
    END;
    $$
```

```
40      LANGUAGE plpgsql IMMUTABLE;
41
42       -- Drop the table if it exists
43      DROP TABLE IF EXISTS temp_nn;
44
45      CREATE TEMP TABLE temp_nn AS
46      (with points AS (
47      SELECT id, poly, ST_AsText(linestr) AS linestr,
48      ST_AsText(ST_Subtract(ST_PointN(t.linestr, 2), ST_PointN(t.linestr, 3))) AS p1,
49      ST_AsText(ST_Subtract(ST_PointN(t.linestr, 2), ST_PointN(t.linestr, 1))) AS p2
50      FROM (
51      SELECT id, ST_AsText(polygon) AS poly,
52
53      --ST_ExteriorRing(st_transform(polygon, 4978)) AS linestr
54      ST_ExteriorRing(polygon) AS linestr
55
56      FROM face
57      )AS t
58      )
59      SELECT id, poly, ARRAY[x,
60          y,
61          z ] as nn
62      FROM
63      (SELECT id, poly, ST_X(n) as x, ST_Y(n) as y, ST_Z(n) as z  FROM
64      (SELECT id, poly, linestr, ST_AsText(ST_CrossProduct(
65      p1, p2
66      )) AS n from points) as tn) as tnn)
67      ;
68
69      -- Update the 'face' table with normalised normal
70      UPDATE face
71      SET normal = CASE
72          WHEN (temp_nn.nn[1]*temp_nn.nn[1] + temp_nn.nn[2]*temp_nn.nn[2] +
            ↪  temp_nn.nn[3]*temp_nn.nn[3]) != 0 THEN
73              ARRAY[
74                  temp_nn.nn[1] / sqrt(temp_nn.nn[1]*temp_nn.nn[1] + temp_nn.nn[2]*temp_nn.nn[2] +
                    ↪  temp_nn.nn[3]*temp_nn.nn[3]),
75                  temp_nn.nn[2] / sqrt(temp_nn.nn[1]*temp_nn.nn[1] + temp_nn.nn[2]*temp_nn.nn[2] +
                    ↪  temp_nn.nn[3]*temp_nn.nn[3]),
76                  temp_nn.nn[3] / sqrt(temp_nn.nn[1]*temp_nn.nn[1] + temp_nn.nn[2]*temp_nn.nn[2] +
                    ↪  temp_nn.nn[3]*temp_nn.nn[3])
77              ]
78          ELSE ARRAY[0, 0, 0]
79      END
80      FROM temp_nn
81      WHERE face.id = temp_nn.id;
```

B.3. Triangulation

B.3.1. Triangulation on convex geometries

```
SELECT
ST_AsText(ST_DelaunayTriangles((ST_Dump(ST_GeomFromText(
    'POLYHEDRALSURFACE Z(
        ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
    ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
    ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
    ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
    ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
    ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)))')
)).geom))
AS Delaunay_result;

SELECT
ST_AsText(ST_Tesselate((ST_Dump(ST_GeomFromText(
    'POLYHEDRALSURFACE Z(
        ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
    ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
    ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
    ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
    ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
    ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)))')
)).geom))
AS Tesselate_result;
```

B.3.2. Triangulation on concave geometries

```
SELECT
ST_AsText(ST_DelaunayTriangles((ST_Dump(ST_GeomFromText(
        'POLYHEDRALSURFACE Z(
        ((2 0 0, 2 2 0, 0 2 0, 0 3 0, 3 3 0, 3 0 0, 2 0 0)),
        ((2 0 0, 2 0 1, 2 2 1, 2 2 0, 2 0 0)),
        ((2 2 0, 2 2 1, 0 2 1, 0 2 0, 2 2 0)),
        ((0 2 0, 0 2 1, 0 3 1, 0 3 0, 0 2 0)),
        ((0 3 0, 0 3 1, 3 3 1, 3 3 0, 0 3 0)),
        ((2 0 0, 3 0 0, 3 0 1, 2 0 1, 2 0 0)),
        ((3 3 0, 3 3 1, 3 0 1, 3 0 0, 3 3 0)),
        ((3 0 1, 3 3 1, 0 3 1, 0 2 1, 2 2 1, 2 0 1, 3 0 1)))')
)).geom))
AS Delaunay_result;

SELECT
ST_AsText(ST_Tesselate((ST_Dump(ST_GeomFromText(
        'POLYHEDRALSURFACE Z(
        ((2 0 0, 2 2 0, 0 2 0, 0 3 0, 3 3 0, 3 0 0, 2 0 0)),
        ((2 0 0, 2 0 1, 2 2 1, 2 2 0, 2 0 0)),
```

```
        ((2 2 0, 2 2 1, 0 2 1, 0 2 0, 2 2 0)),
        ((0 2 0, 0 2 1, 0 3 1, 0 3 0, 0 2 0)),
        ((0 3 0, 0 3 1, 3 3 1, 3 3 0, 0 3 0)),
        ((2 0 0, 3 0 0, 3 0 1, 2 0 1, 2 0 0)),
        ((3 3 0, 3 3 1, 3 0 1, 3 0 0, 3 3 0)),
        ((3 0 1, 3 3 1, 0 3 1, 0 2 1, 2 2 1, 2 0 1, 3 0 1)))')
)).geom))
AS Tesselate_result;
```

B.3.3. Triangulation on titled geometries

```
EXPLAIN ANALYZE
WITH cube AS (
    SELECT
        (ST_GeomFromText(
            'POLYHEDRALSURFACE Z(
            ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
            ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
            ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
            ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
            ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
            ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1))
            )'
        )) AS geom
)
SELECT
    ST_AsText(ST_DelaunayTriangles((
        ST_Dump(ST_RotateY(ST_RotateX(cube.geom, radians(30)), radians(30)))).geom)
        )
    AS Delaunay_result
FROM cube;

EXPLAIN ANALYZE
WITH cube AS (
    SELECT
        (ST_GeomFromText(
            'POLYHEDRALSURFACE Z(
            ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
            ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
            ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
            ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
            ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
            ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1))
            )'
        )) AS geom
)
SELECT
    ST_AsText(ST_Tesselate((
        ST_Dump(ST_RotateY(ST_RotateX(cube.geom, radians(30)), radians(30)))).geom)
        )
```

```
    AS Tesselate_result
FROM cube;
```

B.4. Attribute enrichment

B.4.1. 2D Area

```
ALTER TABLE property DROP COLUMN IF EXISTS area2d;
ALTER TABLE property ADD area2d float;

UPDATE property
SET area2d = subquery.area2d
FROM (
    SELECT
        lod.fid AS id,
        --ST_Volume(ST_MakeSolid(ST_Extrude(lod.geom, 0, 0, b3_h_max))) AS volume
        ST_Area(lod.geom) AS area2d
    FROM dbuser.lod12_2d_9_284_556 lod
) AS subquery
WHERE property.object_id = subquery.id;
```

B.4.2. 3D Area

```
ALTER TABLE property DROP COLUMN IF EXISTS area3d;
ALTER TABLE property ADD area3d float;

UPDATE property
SET area3d = subquery.area3d
FROM (
    SELECT
        lod.fid AS id,
        --ST_Volume(ST_MakeSolid(ST_Extrude(lod.geom, 0, 0, b3_h_max))) AS volume
        ST_3DArea(ST_Extrude(lod.geom, 0, 0, b3_h_max)) AS area3d
    FROM dbuser.lod12_2d_9_284_556 lod
    limit 50
) AS subquery
WHERE property.object_id = subquery.id;
```

B.5. Hierarchy

```
DROP TABLE IF EXISTS temp_centroids;

CREATE TEMP TABLE temp_centroids AS
```

```
With p
AS
(
SELECT id, (ST_Dump(envelope)).geom AS polygon from object
)
SELECT id,
ST_AsText(ST_Force2D(ST_Centroid(ST_Collect(polygon)))) AS cc
FROM p GROUP BY id;

SELECT tmpc.id, cc, envelope FROM temp_centroids tmpc
JOIN object
ON tmpc.id = object.id;


DROP TABLE IF EXISTS hierarchical_clusters;

-- Create a table to store hierarchical cluster information
CREATE TEMP TABLE hierarchical_clusters(
    object_id INTEGER, -- Assuming this column references the object's unique identifier
    level INTEGER, -- Level of clustering (e.g., 0 for initial clustering, 1 for sub-clustering)
    cluster_id INTEGER, -- Cluster ID at this level
    parent_cluster_id INTEGER, -- Cluster ID at the previous level
    name TEXT --unique name for all clusters
);

-- Perform k-means clustering for the first level (level 0)
INSERT INTO hierarchical_clusters (object_id, level, cluster_id, parent_cluster_id, name)
SELECT object_id, 1 AS level, cid AS cluster_id, NULL AS parent_cluster_id, 'Level_0_Cluster_'
↪  || cid AS name
FROM (
    SELECT ST_ClusterKMeans(cc, {0})  OVER()  AS cid, id as object_id, cc
    FROM temp_centroids AS obj
) level_0_clusters;


-- Subsequent levels of clustering (if required)
-- Example: Second level clustering
INSERT INTO hierarchical_clusters(object_id, level, cluster_id, parent_cluster_id, name)
SELECT object_id, 2 AS level, cid AS cluster_id, parent_cluster_id, 'Level_1_Cluster_' ||
↪  parent_cluster_id || '_SubCluster_' || cid AS name
FROM (
    SELECT
        ST_ClusterKMeans(o.cc, {1}) OVER(PARTITION BY t.cluster_id ORDER BY t.cluster_id) AS
        ↪  cid,
        id AS object_id,
        t.cluster_id AS parent_cluster_id
    FROM hierarchical_clusters t
    JOIN temp_centroids o ON t.object_id = o.id
    WHERE t.level = 1 -- Consider removing specific Cluster_id filter here
) level_1_clusters;

DROP TABLE IF EXISTS hierarchy CASCADE;
```

```
53
54 CREATE TABLE hierarchy (
55 hid SERIAL PRIMARY KEY,
56 level int,
57 object_id int[],
58 cluster_id int,
59 parent_cluster_id int,
60 envelope box3d
61 );
62
63 INSERT INTO hierarchy (level, object_id, cluster_id, parent_cluster_id, envelope)
64 --CREATE TABLE hierarchy AS
65 SELECT
66 (ARRAY_AGG(DISTINCT level))[1] AS level,
67 ARRAY_AGG(object_id) AS object_id,
68 (ARRAY_AGG(DISTINCT cluster_id))[1] AS cluster_id,
69 (ARRAY_AGG(DISTINCT parent_cluster_id))[1] AS parent_cluster_id,
70 ST_3DExtent(envelope) AS envelope
71 FROM hierarchical_clusters
72 JOIN object
73 on object.id = object_id
74 GROUP BY name;
75
76
77 ALTER TABLE hierarchy
78 ADD COLUMN temp_tid INTEGER;
79 --ADD COLUMN hid SERIAL PRIMARY KEY;
80
81 --Update temp_tid column with values generated by ROW_NUMBER() window function
82 WITH n AS (
83     SELECT hid,
84         ROW_NUMBER() OVER (PARTITION BY level ORDER BY cluster_id) AS rn
85     FROM hierarchy
86 )
87 UPDATE hierarchy AS h
88 SET temp_tid = n.rn
89 FROM n
90 WHERE h.hid = n.hid;
```

B.6. Tileset JSON

```
1 CREATE OR REPLACE FUNCTION create_vw_tileset() RETURNS VOID AS $$
2 BEGIN
3     -- Drop the existing view if it exists
4     EXECUTE 'DROP VIEW IF EXISTS vw_tileset;';
5
6     -- Create the view vw_tileset
7     EXECUTE '
8         CREATE VIEW vw_tileset AS
```

```
        WITH property AS (
            SELECT json_object_agg(initcap(properties), json_build_object()) AS property_json
            FROM (
                SELECT column_name AS properties
                FROM information_schema.columns
                WHERE table_name = ''property''
                ORDER BY ordinal_position
                OFFSET 2
            ) AS property_data
        ),
    tile
    AS (
    WITH
    e AS (
    SELECT ST_3DExtent(envelope) AS envelope FROM hierarchy WHERE level = 2
)
    SELECT
        h.temp_tid AS id,
        1 AS tileset_id,

        CASE WHEN h.temp_tid != 1 THEN
            1
        ELSE
            NULL
        END AS parent_id,

        CASE WHEN h.temp_tid != 1 THEN
        ARRAY[
        (ST_XMin(h.envelope) + ST_XMax(h.envelope)) / 2, -- centerX
        (ST_YMin(h.envelope) + ST_YMax(h.envelope)) / 2, -- centerY
        (ST_ZMin(h.envelope) + ST_ZMax(h.envelope)) / 2, -- centerZ
        (ST_XMax(h.envelope) - ST_XMin(h.envelope)) / 2, 0, 0, -- halfX
        0, (ST_YMax(h.envelope) - ST_YMin(h.envelope)) / 2, 0, -- halfY
        0, 0, (ST_ZMax(h.envelope) - ST_ZMin(h.envelope)) / 2 -- halfZ
    ]
        ELSE
            ARRAY[
        (ST_XMin(e.envelope) + ST_XMax(e.envelope)) / 2, -- centerX
        (ST_YMin(e.envelope) + ST_YMax(e.envelope)) / 2, -- centerY
        (ST_ZMin(e.envelope) + ST_ZMax(e.envelope)) / 2, -- centerZ
        (ST_XMax(e.envelope) - ST_XMin(e.envelope)) / 2, 0, 0, -- halfX
        0, (ST_YMax(e.envelope) - ST_YMin(e.envelope)) / 2, 0, -- halfY
        0, 0, (ST_ZMax(e.envelope) - ST_ZMin(e.envelope)) / 2 -- halfZ
    ]
        END AS bounding_volume,

        CASE WHEN h.temp_tid = 1 THEN
            ROUND(
            sqrt(
                power(ST_XMax(h.envelope) - ST_XMin(h.envelope), 2) +
                power(ST_YMax(h.envelope) - ST_YMin(h.envelope), 2) +
                power(ST_ZMax(h.envelope) - ST_ZMin(h.envelope), 2)
```

```
            )::numeric/2,
            2)
            --diagonal_length
        ELSE
            0
        END AS geometric_error,

        CASE WHEN h.temp_tid = 1 THEN
            ''ADD''
        ELSE
            NULL
        END AS refine,

        h.temp_tid AS content
    FROM hierarchy h, e
    WHERE h.level = 2
    ),

        children AS (
            SELECT
                array_agg(json_build_object(
                    ''boundingVolume'', json_build_object(
                        ''box'', tile_data.bounding_volume
                    ),
                    ''geometricError'', tile_data.geometric_error,
                    ''content'', json_build_object(''uri'', CONCAT(''/tiles/'',
                    ↪  tile_data.content, ''.b3dm''))
                )) AS children_json
            FROM

                (
                SELECT *
                FROM tile
                WHERE tileset_id = 1 --AND tile_data.parent_id IS NOT NULL
                 --AND tile_data.parent_id IS NOT NULL
                ORDER BY id -- Order by tile_id
                ) AS tile_data
             GROUP BY tile_data.tileset_id
            )

        SELECT
            1 AS id,
            json_build_object(
                ''asset'', json_build_object(
                    ''version'', ''1.0'',
                    ''tilesetVersion'', ''1.2.3''
                ),
                ''properties'', property.property_json,
                ''geometricError'', tile_data.geometric_error,
                ''root'', json_build_object(
                    ''boundingVolume'', json_build_object(
                        ''box'', tile_data.bounding_volume
```

```
                    ),
                    ''geometricError'', tile_data.geometric_error,
                    ''refine'', tile_data.refine,
                    ''content'', json_build_object(
                        ''boundingVolume'', json_build_object(
                            ''box'', tile_data.bounding_volume
                        ),
                        ''uri'', CONCAT(''/tiles/'', tile_data.content, ''.b3dm'')
                    ),
                    ''children'', (children.children_json)[2:],
                    ''transform'', ARRAY[1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                    ↪  0.0, 1, 1, 1, 1.0]
                )
            ) AS tileset_json
        FROM (
            SELECT *
            FROM tile
            WHERE tileset_id = 1 AND parent_id IS NULL
        ) AS tile_data
        ,property
        ,children;';
    RETURN;
END;
$$ LANGUAGE plpgsql;

-- Call the function to create the view
SELECT create_vw_tileset();

-- Retrieve data from the created view
SELECT * FROM vw_tileset;
```

B.7. Spatial query

B.7.1. Bounding box query

```
-- Approach1 intersection check directly on the bottom level
--EXPLAIN ANALYZE
SELECT  level,
    h.temp_tid,
    h.level,
    h.cluster_id,
    h.parent_cluster_id,
ST_Intersects(ST_Force2D(envelope),
              ST_Buffer(ST_GeomFromText('POINT(3921335 299904)'), 1)
             ) AS s
FROM hierarchy h
WHERE level = 2
AND ST_Intersects(ST_Force2D(envelope), ST_Buffer(ST_GeomFromText('POINT(3921335 299904)'), 1))
ORDER BY level;
```

```
15
16  -- Approach2, intersection check from top level to bottom level
17  --EXPLAIN ANALYZE
18  WITH Level1 AS (
19      SELECT cluster_id
20      FROM hierarchy
21      WHERE level = 1
22  )
23  SELECT
24      h.temp_tid,
25      h.level,
26      h.cluster_id,
27      h.parent_cluster_id,
28      ST_Intersects(ST_Force2D(h.envelope), ST_Buffer(ST_GeomFromText('POINT(3921335 299904)'),
        ↪  1)) AS s
29  FROM
30      hierarchy h
31  WHERE
32      h.level = 2
33      AND h.parent_cluster_id IN (SELECT cluster_id FROM Level1) -- Filtering multiple cluster IDs
34      AND ST_Intersects(ST_Force2D(h.envelope), ST_Buffer(ST_GeomFromText('POINT(3921335.5566243
        ↪  299904.402694535)'), 1))
35  ORDER BY
36      h.level;
```

B.8. Database storage system benchmarks

```
1   SELECT table_name,
2          calculation_type,
3          total_size / (1024.0 * 1024.0) AS total_size_MB
4   FROM (
5   SELECT 'face' AS table_name,
6          'total_size1' AS calculation_type,
7          SUM(pg_column_size(id) +
8              pg_column_size(tri_node_id) +
9              pg_column_size(object_id) +
10             pg_column_size(normal)) AS total_size
11  FROM face
12  UNION ALL
13  SELECT 'face' AS table_name,
14         'total_size2' AS calculation_type,
15         SUM(pg_column_size(tri_node_id) +
16             pg_column_size(normal)) AS total_size
17  FROM face
18  UNION ALL
19  SELECT 'object' AS table_name,
20         'total_size1' AS calculation_type,
21         SUM(pg_column_size(id) +
22             pg_column_size(nodes)) AS total_size
```

```
FROM object
UNION ALL
SELECT 'object' AS table_name,
       'total_size2' AS calculation_type,
       SUM(pg_column_size(nodes)) AS total_size
FROM object
UNION ALL
SELECT 'property' AS table_name,
       'total_size1' AS calculation_type,
       SUM(pg_column_size(pid) +
           pg_column_size(object_id) +
           pg_column_size(height)) AS total_size
FROM property
UNION ALL
SELECT 'property' AS table_name,
       'total_size2' AS calculation_type,
       SUM(pg_column_size(height)) AS total_size
FROM property

UNION ALL
SELECT 'hierarchy' AS table_name,
       'glb' AS calculation_type,
       SUM(pg_column_size(glb)) AS total_size
FROM hierarchy

UNION ALL
SELECT 'hierarchy' AS table_name,
       'b3dm' AS calculation_type,
       SUM(pg_column_size(b3dm)) AS total_size
FROM hierarchy

UNION ALL
SELECT 'vw_tileset' AS table_name,
       'total_size1' AS calculation_type,
       SUM(pg_column_size(id) +
           pg_column_size(tileset_json)) AS total_size
FROM vw_tileset
UNION ALL
SELECT 'vw_tileset' AS table_name,
       'total_size2' AS calculation_type,
       SUM(pg_column_size(tileset_json)) AS total_size
FROM vw_tileset
) AS subquery;
```

C. Github link

C.1. Software usage

The source code is stored and maintained in GitHub repository: https://github.com/yangyzoey/3dtiles.

Step:

1. Load data into Postgres

First, download gpkg dataset:

Downloaded a tile from 3dbag.nl in geopackage format (tile around TU Delft Aula).

Then, convert the 3D layer (3D Multi Polygon) with LOD 1.2 into PostGIS dump format, using ogr2ogr:

```
$ ogr2ogr --config PG_USE_COPY YES -f PGDump test_9-284-556.dmp 9-284-556.gpkg -sql
"SELECT * FROM lod12_3d" -nln "test_lod12_3d" -lco SCHEMA=dbuser
```

The command helps check the information of the dataset:

```
$ ogrinfo -so 9-284-556.gpkg
```

Next, load the dump file into Postgres. Make sure PostGIS is enabled in the target database. This results in a table 'test_lod12_3d', where the 3D geometry is stored as multipolygonz, with coordinate reference system EPSG:7415:

```
CREATE EXTENSION IF NOT EXISTS postgis;

$ psql -d [database_name] -U [database_host] -h localhost -f test_9-284-556.dmp
```

2. Run the prototype:

Preparation

Set up for your local database in database.ini

Download Cesium-1.110.zip from Cesium GitHub releases, and put in the project root directory.

Set up python environment (See requirements.txt)

Serve 3D Tiles on-the-fly

- Perform a coordinate transformation from EPSG:7415 (RD+NAP) to EPSG:4978, and harmonise geometries to valid polygonz.
- Compute and prepare 3D Tiles information (normal, position, triangulated topology, and tileset structure)
- Run the webservice and complete the tiles creation. Then visualise on Cesium.

```
$ python server.py
```

Now connect with a web browser to the service running on your own laptop: http://127.0.0.1:5000

Additional Notes:

Feel free to customise configuration for the application.
In the input.json file, you can modify the parameters according to your requirements.

D. Reflection

During this journey, I have encountered numerous challenges and opportunities. When I reflect on my path, I am grateful for the valuable gains I have gained in the learning process

Taking notes during implementation tests is important. The implementation and validation of the proposed methods lasted for several months. After some time, some key points may be missed. This is different from what we did in assignments or labs for master courses, where I can clearly remember all the steps. In addition, it is important to explain the research story and convey ideas to audiences from different backgrounds. Writing things down and drawing figures is a good way to organize your ideas and research results.

Time management is another aspect. I am still learning and making progress. When I was studying architecture for my bachelor's, people always said design never ends. I feel it is similar to prototype refinement. However, I learned how to set milestones for each phase and keep the progress on track.

At the end of March, I had the opportunity to attend the OGC meeting on the topic of Geo-BIM for the Built Environment. I was always wondering if the 3D Tiles database could work as a translator to bridge multiple databases. Thanks to insights from the discussion and presentation in OGC meetings, I understand the gap in the domain and find the exact point where the 3D Tiles database should work to bridge the gap.

To conclude, I learned a lot from this journey. I don't think I have reached the destination of this project, but it is a milestone for me, and I would like to share this work.

Bibliography

Alattas, A., de Vries, M., Meijers, B., Zlatanova, S., and van Oosterom, P. (2021). 3D pgRouting and visualization in Cesium JS using the integrated model of LADM and IndoorGML.

Arroyo Ohori, K. (2016). Higher-dimensional modelling of geographic information. https://3d.bk.tudelft.nl/ken/files/16_thesis_lowres.pdf.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., and Çöltekin, A. (2015). Applications of 3D city models: State of the art review. *ISPRS Int. J. Geo Inf.*, 4:2842–2889. https://api.semanticscholar.org/CorpusID:21082988.

Boissonnat, J., Devillers, O., Teillaud, M., and Yvinec, M. (2000). Triangulations in CGAL. https://dl.acm.org/doi/10.1145/336154.336165.

Broilo, M., Piotto, N., Boato, G., Conci, N., and De Natale, F. G. B. (2010). Object trajectory analysis in video indexing and retrieval applications. https://api.semanticscholar.org/CorpusID:12979404.

Brunel, R. (2017). Enhancing relational database systems for managing hierarchical data. https://mediatum.ub.tum.de/doc/1369976/1369976.pdf.

Cesium and OGC (2019). 3D Tiles Specification 1.0. https://docs.ogc.org/cs/18-053r2/18-053r2.html.

CesiumGS (2021). 3d-tiles-reference-card. https://github.com/CesiumGS/3d-tiles/blob/main/3d-tiles-reference-card.pdf.

CesiumGS (2022). 3d-tiles-reference-card-1.1. https://github.com/CesiumGS/3d-tiles/blob/main/3d-tiles-reference-card-1.1.pdf.

Chen, Y., Shooraj, E., Rajabifard, A., and Sabri, S. (2018). From ifc to 3d tiles: An integrated open-source solution for visualising bims on cesium. https://api.semanticscholar.org/CorpusID:53237119.

Coors, V. (2003). 3D-GIS in networking environments. *Comput. Environ. Urban Syst.*, 27:345–357. https://api.semanticscholar.org/CorpusID:206048790.

Indrajit, A. (2021). 4d open spatial information infrastructure. https://repository.tudelft.nl/islandora/object/uuid:cd993e69-6310-4e64-87ab-1ac1a1fcb149?collection=research.

Khater, I., Nabi, I. R., and Hamarneh, G. (2020). A review of super-resolution single-molecule localization microscopy cluster analysis and quantification methods. *Patterns*, 1. https://api.semanticscholar.org/CorpusID:225724448.

Khemani, C., Doshi, J., Duseja, J., and Shah, K. (2019). Solving rubik's cube using graph theory. https://www.researchgate.net/publication/326749335_Solving_Rubik's_Cube_Using_Graph_Theory_ICCI-2017.

KhronosGroup (2021). glTF 2.0 Specification. https://github.com/KhronosGroup/glTF.

Koukofikis, A., Coors, V., and Gutbell, R. (2018). Interoperable visualization of 3D city models using OGC's standard 3D portrayal service. *ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci.*, IV(4):113–118.

Kragler, R. (2016). Solid modeling with boolean operations in mathematica. part 1: Definition of solid objects. https://www.researchgate.net/publication/319902549_Solid_Modeling_with_Boolean_Operations_in_Mathematica_Part_1_Definition_of_Solid_Objects.

Mao, B., Ban, Y., and Laumert, B. (2020). Dynamic Online 3D Visualization Framework for Real-Time Energy Simulation Based on 3D Tiles. *ISPRS International Journal of Geo-Information*, 9(3):166.

Mei, G., Tipper, J. C., and Xu, N. (2012). Ear-clipping based algorithms of generating high-quality polygon triangulation. https://link.springer.com/chapter/10.1007/978-3-642-34531-9_105.

Molenaar, M. (1992). A topology for 3D vector maps. *International Journal of Applied Earth Observation and Geoinformation*, pages 25–34. https://api.semanticscholar.org/CorpusID:130039734.

NASA AMMOS (2020). 3dtilesrendererjs. https://github.com/NASA-AMMOS/3DTilesRendererJS/.

OSGeo (2024). Gdal. https://gdal.org/programs/ogr2ogr.html.

Oslandia and IGN (2022). Sfcgal. https://oslandia.gitlab.io/SFCGAL/authors.html.

Peters, R. Y., Dukai, B., Vitalis, S., van Liempt, J., and Stoter, J. E. (2021). Automated 3d reconstruction of lod2 and lod1 models for all 10 million buildings of the netherlands. *ArXiv*, abs/2201.01191. https://api.semanticscholar.org/CorpusID:245668761.

Pilouk, M. (1996). Integrated modelling for 3D GIS. https://api.semanticscholar.org/CorpusID:117660282.

Renxin, Y., Qinghuang, Y., Tianrong, Z., Wei, L., and Ying, M. (2019). Visualized panoramic display platform for transmission cable based on space-time big data. https://link.springer.com/chapter/10.1007/978-981-15-1304-6_25.

Teunissen, W. and van Oosterom, P. (1988). The creation and display of arbitrary polyhedra in hirasp. https://gdmc.nl/oosterom/rul_cs88-20.pdf.

Treumer, J., Neumann, L., Lorenz, B., and Pfleging, B. (2023). Fast triangle strip generation and tunneling for different cost metrics. https://link.springer.com/chapter/10.1007/978-3-031-22025-8_13.

Van Oosterom, P., Stoter, J., Quak, W., and Zlatanova, S. (2002). The balance between geometry and topology. https://api.semanticscholar.org/CorpusID:8454374.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., and Kolbe, T. (2018). 3DCityDB - A 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3:1–26. https://api.semanticscholar.org/CorpusID:44088436.

Zlatanova, S. and Gruber, M. (2001). 3D urban GIS on the web: Data structuring and visualization. https://www.isprs.org/proceedings/xxxii/part4/zlatan82neu.pdf.

Zlatanova, S., Holweg, D., and Coors, V. (2004). Geometrical and topological models for real-time GIS. https://api.semanticscholar.org/CorpusID:61851191.

Zlatanova, S., Pilouk, M., and Tempfli, K. (2009). Building reconstruction from aerial images and creation of 3D topologic data structure. https://api.semanticscholar.org/CorpusID:5174597.

Zlatanovaa, S. (2000). 3D GIS for urban development. https://api.semanticscholar.org/CorpusID:130381637.

Zlatanovaa, S., A.A., R., and Shi, W. (2003). Topological models and frameworks for 3D spatial objects. *ComputersGeosciences*, 30:419–428.

Colophon

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