A Prototype Architecture for Interactive 3D Maps on the Web

by

Ting Liu
B.Eng., Nanjing University of Aeronautics and Astronautics, 2003
M.B.A, Huazhong University of Science and Technology, 2016

A Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Ting Liu, 2024
University of Victoria

All rights reserved. This project may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

A Prototype Architecture for Interactive 3D Maps on the Web

by

Ting Liu
B.Eng., Nanjing University of Aeronautics and Astronautics, 2003
M.B.A, Huazhong University of Science and Technology, 2016

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

ABSTRACT

Virtual 3D city models offer detailed 3D representations of urban space and serve in various fields, such as urban planning, architecture, navigation, and environmental simulation. With the advancement of technologies such as photogrammetry and laser scanning, the scale of 3D city models has increased significantly, making it a challenge to transmit and visualize such large datasets for sharing purposes. The development of advanced web technologies and the emergence of WebGL has made it possible to render and share large-scale 3D city models on the Internet. In addition, the introduction of game engines has further enhanced the simulation and interactive functions of 3D GIS applications.

In this project, the exploration focused on using and integrating WebGL-based rendering tools to visualize large 3D city models, providing a portal where users can navigate and interact with urban scenarios from different perspectives. The architecture utilized 3DCityDB for tiling and format conversion of 3D models, 3D Web Client/Cesium.js virtual globe for loading large-scale tiled data, and Babylon.js to achieve interactive functions and environmental simulation. A GridMap mechanism was proposed to solve the problem of loading a large number of models with geographic coordinates in the Babylon scene. Test results show that this mechanism can maintain effective loading efficiency. Especially when the size of the dataset grows significantly, loading time and memory consumption will not increase, and FPS can also be maintained at a high level to ensure smooth interaction. This study expands the feasibility of applying 3D GIS data in web-based game engines through enhanced interactivity and simulation.

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vi

List of Figures vii

Acknowledgements x

1 Introduction 1

1.1 Motivation and Challenge 1

1.2 Research Questions 3

1.3 Report Outline 3

2 Background and Related Work 4

2.1 Background 4

2.1.1 Foundations of the Web 4

2.1.2 Modern Web Application Technologies and Frameworks 5

2.1.3 3D File Formats 9

2.2 Related Work 13

2.2.1 Data Management and Transmission 13

2.2.2 Rendering Approach 16

2.2.3 User Interaction 20

3 Methodology 24

3.1 Architecture 24

3.2 Data Preprocessing 26

3.2.1 Data Source 26
3.2.2 Data Processing Steps 27
3.3 Server-side Solutions 36
3.4 Client-side Rendering 39
3.4.1 Bird's Eye View Rendering 40
3.4.2 First-Person View (FPV) Rendering 42
3.5 User Interactions 47
3.5.1 Navigation 47
3.5.2 Semantic Information Display 47
3.5.3 Location Marker 48
3.5.4 Mini-map Widget 49
3.5.5 Collision Detection 51
3.5.6 Environment Simulation 52

4 Performance Evaluation **55**

5 Conclusion and Future Work **61**

A Acknowledgment of AI Assistance **64**

References **65**

List of Tables

Table 2.1 Key hooks in React. 10
Table 2.2 The main components in glTF structure. 12
Table 2.3 Comparison of WebGL frameworks for use in geospatial applications 21

Table 3.1 The main components of the 3DCityDB schema. 30

Table 4.1 Test results of two datasets in conditions of GridMap enabled and disabled, separately, and each set of tests repeated 10 times. . . 57
Table 4.2 The average values of loading time, memory usage, and FPS for each dataset with GridMap enabled and disabled. 58

Table 5.1 Functionalities implemented for user interaction. 62

List of Figures

Figure 2.1 CityGML building object with aggregation hierarchy. 11
Figure 2.2 glTF object hierarchy. 13

Figure 3.1 Overview of the prototype architecture. 25
Figure 3.2 Key components of the 3DCityDB software suite. 27
Figure 3.3 User interface of the CityGML Import/Export tool. 28
Figure 3.4 The Schulungskurs area in 2D map. 28
Figure 3.5 The Charlottenburg-Wilmersdorf area in 2D map. 28
Figure 3.6 Screenshot of successfully setting up a 3D City Database instance in PostgreSQL to store CityGML data. 29
Figure 3.7 Types and numbers of the imported CityGML top-level features printing in the console window. 31
Figure 3.8 The database report displays all tables of the 3DCityDB with row numbers in the console window. 32
Figure 3.9 The VIS export tab of 3DCityDB Importer/Exporter. 33
Figure 3.10The textured LoD2 models. 34
Figure 3.11Coordinates calculation on the database tab. 35
Figure 3.12Preferences settings. 36
Figure 3.13Altitude and terrain settings. 37
Figure 3.14The New Column dialog window used to define the spreadsheet columns. 38
Figure 3.15Exported attribute data in XLSX format. 39
Figure 3.16An example of a hierarchical directory structure for export files. 40
Figure 3.17Screenshot of a 3D object's envelope attribute in the city object JSON file. 40

Figure 3.18 The GripMap mechanism implemented to load grids based on the user's position. The area is divided into equal-sized grids based on cell_size_lat and cell_size_lon. Nine adjacent grids are calculated and then loaded into the Babylon scene. 41
Figure 3.19 The initial page of the application. 42
Figure 3.20 The urban scene from a bird's eye view loaded through 3D Web Client/Cesium based on original latitude and longitude points. 43
Figure 3.21 The object is offset on the map when loaded into the scene combined with Babylon and Mapbox. 45
Figure 3.22 The object's texture becomes blurry and dark, and the shape becomes distorted and deformed. 45
Figure 3.23 3D buildings with texture displayed normally in Babylon without the combination of Mapbox. 45
Figure 3.24 A screenshot of the FPV scenario, in which the user can explore in first-person perspective using the mouse and arrow keys. The background environment is implemented using a skybox. In the upper right corner of the screen, a mini-map widget provides visual cues as a top-down map to assist in navigating the scene. In addition, the building's ID will be displayed in the upper left corner when the user approaches a building. 48
Figure 3.25 A screenshot of a pop-up window showing the selected building's information. 49
Figure 3.26 A screenshot shows a marker marking the location selected by the user. 50
Figure 3.27 The ellipsoid around the camera represents the player's dimensions for collision detection. When a mesh comes in contact with this ellipsoid, a collision event will be triggered to prevent the camera from getting too close to the mesh. 52
Figure 3.28 The CubeTexture images must have six textures corresponding to the six faces of the cube. 54

Figure 4.1 Screenshot of the page failed to load and displayed nothing during the tests of the Charlottenburg-Wilmersdorf dataset with GridMap disabled. 56

Figure 4.2 The console window shows the time, memory, and fps information when loading the Charlottenburg-Wilmersdorf dataset with GridMap disabled. 58
Figure 4.3 The failure messages of object loading during the same test. . . 58
Figure 4.4 The models near the origin location failed to add to the scene properly when loading a large dataset with GridMap not enabled. 59
Figure 4.5 Comparison of average loading time of two datasets with GridMap enabled and disabled. 59
Figure 4.6 Comparison of average memory usage of two datasets with GridMap enabled and disabled. 60
Figure 4.7 Comparison of average FPS of two datasets with GridMap enabled and disabled. 60

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Yvonne Coady, my supervisor, for giving me the opportunity to study with her. Her mentoring, support, encouragement, and patience enabled me to persevere.

Dr. Derek Jacoby, for guiding me and providing feedback throughout my research work.

My husband and family members, for their company and love, which have encouraged and inspired me throughout the journey. Without their support, I would never be where I am today.

My lab classmate, Xi Sun, and other friends, for their suggestions, encouragement, and support.

Chapter 1

Introduction

1.1 Motivation and Challenge

Virtual three-dimensional (3D) city models represent detailed, three-dimensional representations of urban spaces, combining both physical and spatial characteristics of real-world environments. A virtual 3D city model can range from simple block-like representations to highly detailed renderings with textures and details, showcasing buildings, roads, landscapes, etc. From urban planning and architecture to navigation systems and environmental simulations [1], [2], 3D city models serve in various applications, such as decision-making, simulation and analysis, public engagement, and educational tools, enhancing people's understanding and interaction with urban environments. For example, city planners can use these models for zoning, land use planning, and visualizing future developments. Environmentalists use it to monitor changes in land use, habitat loss, and the impacts of climate change [1]. Moreover, the widespread adoption of advanced 3D acquisition technologies, such as photogrammetry and laser scanning, the exponential growth of computing power, and the improvement of 3D software tools [3] have significantly increased the scale of 3D city models.

Despite the growing ease of acquiring models, transferring and visualizing such large 3D datasets for sharing purposes is a significant challenge. 3D models usually require more storage space than two-dimensional (2D) data and are combined to form model geometries complete with texture and semantic details [4]. Traditionally, working with these models required higher-end computer hardware, which has hindered their sharing, especially over the Internet [5], [6]. However, with the enhancement

of modern computer hardware and the development of network technology, it has become possible to share 3D models through the network. Of course, employing 3D models on the web presents specific technical difficulties, including issues such as network speed and cross-platform support [4]. Currently, using tiled 3D models is a common practice for efficient data processing and rendering. Tiled 3D models, which organize data into spatially coherent sections and offer multiple levels of detail, enable 3D viewing software to load individual tiles as needed [5]. Additionally, storing 3D models in the cloud and accessing the required models on demand through an online database can reduce the cost of transmitting large data sets between users and further improve the efficiency of sharing.

Another interesting trend nowadays is the use of game technology platforms to reproduce the simulation of "real" 3D scenes for areas such as urban management and planning [7]. The game engine creates a broad exploration environment for professional applications involving 3D GIS data through support for interactivity, animation, and simulation possibilities [4], [8]. Developers have long used game engines to visualize virtual worlds for players to explore. These immersive, dynamic, and engaging experiences can also help stakeholders, sometimes, who are often non-technical people affected by the project, better understand and explore natural 3D geographical spaces. Additionally, one of the most attractive developments in 3D urban modeling application platforms is the advancement of browser-based game engines. Not only is this expected to bring game engine-like functionality to browser-based 3D applications, but it also enables the integration of visualization capabilities, such as VR, with future online GIS applications [6]. However, using real-world data in game engines has always been a challenging task, as most game engines have little support for geospatial data. Therefore, a significant motivation for this research is integrating GIS platforms with browser-based game engines, enabling them to enhance one another.

Based on the above motivation and challenge, this project explored how to present large-scale 3D city models on the web. The purpose was to propose a prototype to realize rich interactions with the 3D city models in an urban scenario by integrating modern web frameworks and advanced WebGL-based rendering platforms. The prototype also implements a mechanism for solving the problem of efficiently loading large geospatial 3D city models in the browser-based game engine Babylon.js.

1.2 Research Questions

This project focuses on the main research question: How can large 3D city models be displayed on the web and achieve user interaction in near real-time?

This high-level question can be further broken down into several secondary questions:

1. How can large amounts of 3D city models be transferred efficiently from the back end to the front end?
2. How can large-scale 3D city models be visualized smoothly on the web?
3. Can users interact with objects in 3D city scenes without unacceptable latency?

1.3 Report Outline

The rest of the report is organized as follows:

Chapter 2 describes the background knowledge regarding web development, technologies, frameworks, and 3D file formats and reviews previous work from three aspects: data management and transmission, rendering methods, and user interactions according to the primary and secondary research questions.

Chapter 3 details the prototype development process, including architecture design, data preprocessing, server-side and client-side implementation, and user interactions.

Chapter 4 introduces the performance evaluation experiments and discusses the results.

Chapter 5 summarizes the contributions and limitations of the study and addresses future work.

Chapter 2

Background and Related Work

This chapter consists of two sections. The first section illustrates the concepts, terminologies, technologies, and framework of web applications, as well as 3D city data formats, particularly the format suitable for 3D city models and 3D rendering on the web. The second section reviews the related work in terms of processing large 3D city models for transmission, utilizing WebGL-based rendering tools for presenting 3D models, and integrating game engines into the visualization of geo 3D models.

2.1 Background

2.1.1 Foundations of the Web

A web application is a computer program that utilizes web browsers and web technology to perform tasks over the internet. Unlike traditional desktop applications installed on a local computer or device, web applications are cross-platform and easily accessible from virtually any device with internet connectivity. It generally follows a client-server architecture, on which the World Wide Web introduced three basic concepts: Uniform Resource Locators (URL), HyperText Markup Language (HTML), and Hypertext Transfer Protocol (HTTP) [9].

A URL is a reference or address used to access resources on the internet. It specifies the location of a resource (like a document or an image) on the web and provides a means to retrieve it. Web browsers use URLs to fetch and display content, as well as in countless other internet protocols and applications.

HTML is the primary language used to describe the structure and presentation of content on the web. With the development of the browser, HTML has also con-

tinuously displayed superior visual performance. The release of HTML5 has brought significant improvements, numerous enhancements, and new features over its predecessors, such as native video and audio support, the canvas element for drawing, and local storage, addressing the needs and demands of modern web applications. Regarding web-based 3D rendering, HTML5 provides the framework and foundational elements for structuring and interacting with web content, while another technology, the Web Graphics Library (WebGL), extends one of those elements, the `<canvas>`, to allow for advanced 3D rendering. Specifically, the `<canvas>` element essentially provides a blank drawing surface upon which graphics can be drawn using JavaScript. At the same time, WebGL operates the `<canvas>` element and the Document Object Model (DOM) interface to draw 3D graphics and provides a new standard for interactive 3D graphics by enabling web browsers to access Graphics Processing Units (GPUs). GPUs accelerate the imaging process of 3D graphics, reduce the time required for rendering, and support the use of real-time, photorealistic visualization techniques [10]. Therefore, WebGL technology makes it possible to execute large-scale 3D graphics on the web.

The primary protocol for web communication between the client and the server is the Hypertext Transfer Protocol (HTTP), which is an application layer protocol used for transmitting hypermedia documents, such as HTML, images, and other resources. It follows a request-response model where the client sends an HTTP request to the server, and the server returns an HTTP response. Requests and responses include headers that provide metadata about the communication (e.g., content type, status codes, cookies) and a body that contains the actual data. HTTP defines eight basic operations: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, and CONNECT, of which GET and POST are the two primary types used in the protocol [9]. The GET method requests data from a specific resource, while the POST method sends data to a server to create or update a resource.

2.1.2 Modern Web Application Technologies and Frameworks

Early web applications lacked a systematic structure as contents, formatting, and processing instructions (application or business logic) were combined in the same file. Later, they embraced and incorporated the concept of individual components that could be more easily integrated into build applications. As a result, web applications have used standard components or modules regularly [9].

In the beginning, web pages were just simple text files composed of static content and links. The web's client-server architecture inspired the idea of processing and dynamically generating pages on the server. The browser served as a universal user interface to render the page. This realization then led to a hybrid model, which is typically structured to create web pages containing a small amount of code directly in the page text. The code is executed, and its results are inserted into the current document when a request is made. Over time, the model was further refined and became typical in today's web applications, where a central component manages access to the data, while the code dispersed in the HTML page only displays what is needed [9].

In web development, the terms "client-side" and "server-side" are often referenced as the two components that combine to produce a functional web application. They differentiate between what the user interacts with directly and what operates behind the scenes. The client, often a web browser, is the web interface that interacts with the application server to retrieve and display information to the user. The client interacts with the user, with scripts embedded in HTML pages and running in the browser environment. Scripts react to user events, such as mouse movements and clicks, and their objects refer to user interface entities, such as windows, menus, or cookies [9]. Code executed within the browser primarily focuses on enhancing the visual and interactive elements of the displayed webpage. This encompasses tasks like customizing UI elements, crafting layouts, guiding navigation, verifying form inputs, and other related activities. The server-side scripts manage other types of objects, such as clients and files. Compared to the client, server-side code handles tasks such as validating submitted data and requests, using a database to store and retrieve data, and selecting what content to return to the browser in response to requests.

Nowadays, developers typically write code using web frameworks, which are comprehensive assemblages of functions, objects, protocols, and additional coding constructs designed to address common problems, expedite the developmental process, and streamline different types of tasks specific to a domain. A technology stack is a combination of software tools, programming languages, frameworks, and data storage technologies that developers use to build and run a single application.

For web applications, a tech stack often refers to the combination of tools and technologies used for both the client and server sides. Client-side development typically applies programming languages such as HTML, CSS, and JavaScript/TypeScript, with frameworks like React, AngularJS, or Vue.js. Server-side technologies typically

include languages and frameworks such as PHP, Java, Ruby, Python (Django, Flask), and JavaScript (Node.js) [9]. Client-oriented frameworks primarily facilitate layout design and content presentation. In contrast, server-side web frameworks afford a large number of typical web server functions without the need for manual implementation, such as session management, user authentication protocols, streamlined database connectivity, and templating libraries.

In this research, the application employs HTML, CSS, and JavaScript/TypeScript as the programming languages, with React as the framework on the client side. Additionally, JavaScript and Python are used as the programming languages, with Express and Flask serving as the frameworks to handle static file serving and calculation tasks on the server side.

Python [11] is a high-level programming language that is widely popular due to its simplicity, readability, and versatility. Its comprehensive standard library and support for multiple programming paradigms, including procedural, object-oriented, and functional programming, make Python the best choice for applications ranging from web development to data science and artificial intelligence. In web development, Python is known for its efficiency and scalability. It provides a solid foundation for various web frameworks and simplifies the development process of complex web applications. Python's diverse libraries and tools and its simple syntax ensure rapid development and deployment of web applications. Python provides a framework for almost all types of web applications. Some popular frameworks include Django, Flask, Pyramid, etc.

In this research, Flask [12] was chosen as the backend server framework mainly because it is lightweight and provides the essential elements for running web applications, allowing developers to use their favorite tools and libraries. Flask was created by Armin Ronacher and is based on the Werkzeug WSGI toolkit and the Jinja2 template engine. Flask is commonly used as a backend for web APIs, especially in conjunction with frontend frameworks like React or Angular.

JavaScript [13] is a high-level, interpreted scripting language that primarily enhances website interactivity, facilitates user interactions, manages animations, and updates content dynamically without reloading an entire page. JavaScript has continued to evolve since its inception. The language's specifications, managed by the ECMAScript (often abbreviated as ES) standard, have introduced many features, enhancing the language's capabilities and developer experience. Especially with the advent of environments like Node.js, JavaScript can run on various environments, not

just browsers, allowing developers to build full-fledged web applications both on the client and server sides.

Node.js [14] is a runtime environment that allows JavaScript to be executed outside the scope of a web browser. Initially conceived and created by Ryan Dahl in 2009, Node.js has fundamentally changed the way developers perceive and utilize JavaScript, shifting its functionality from merely scripting web browsers to building robust and scalable server-side applications. The core of Node.js is Google's V8 JavaScript engine, which compiles JavaScript directly into machine code to ensure optimized execution. Express.js [15] is a minimalist web application framework for Node.js designed to build web applications and APIs quickly and easily. It provides robust features for building web servers and handling HTTP requests and responses. It allows developers to define routes, which are mechanisms that set up rules to determine how the application responds to HTTP requests to handle different HTTP methods and URLs.

The Node Package Manager (NPM) [16] is the default package manager for Node.js, providing an extensive library of modules and packages to simplify the development process. With NPM, developers can easily share and reuse code and benefit from the vast Node community. The development community has produced numerous libraries and frameworks based on JavaScript to address recurring issues and streamline the developmental workflow.

TypeScript [17] is a superset of JavaScript that was first made public in 2012 and has since gained widespread popularity in the web development community. TypeScript enhances JavaScript by adding static typing and advanced type features, making the code more predictable and allowing for better analysis during development. It also provides better completion prompts when writing code. Many potential problems can be discovered before running, improving maintainability and reducing errors. TypeScript code can be transpiled to JavaScript, making it executable in any JavaScript environment, including browsers and Node.js. TypeScript also has a strong community and ecosystem, with many popular libraries and frameworks offering TypeScript typings, which allows developers to use these libraries with the full benefits of TypeScript's type system. In addition, Babylon.js, the 3D rendering tool adopted for this project, is written in TypeScript. Therefore, TypeScript was chosen as the programming language to implement the Babylon rendering scene.

React.js, commonly referred to as React, is an open-source JavaScript library for building user interfaces, bringing a paradigm shift in how web applications are devel-

oped [18]. It emphasizes component-based architecture and reactive updates, encouraging developers to create scalable and interactive user interfaces. Each component in React maintains its state and logic, making the codebase more maintainable and scalable. One of its standout features is the implementation of a virtual DOM, which optimizes the rendering process. Also, React does not update the entire page but detects changes and updates only the necessary parts, ensuring efficient and speedy rendering. In addition, React employs a unidirectional data flow, which means that data changes and updates originate from a single source (typically the top-level component or state) and propagate downwards through the component hierarchy. This feature makes the state predictable, enhances the debugging process, and makes the application behave more consistently. Its flexibility, ease of combination with other frameworks, and multiplatform integration make React a perfect choice for developers from all fields.

Introduced in React 16.8, React Hooks represent a significant shift in how developers can work with React. Before that, the state and other React functions could only be accessed in the class component. With the introduction of Hooks, functional components gained the ability to manage local state and access React features like lifecycle methods (useEffect), context (useContext), memoization (useMemo, useCallback), and more, promoting cleaner and more concise code. Hooks simplify logic, allowing more direct logic and state management without switching between class and function components. With custom hooks, logic can be extracted and reused across different components. They also eliminate the need for class boilerplate and make components simpler. Related logic can be grouped together using multiple effects, leading to more organized and readable components. Table 2.1 lists some of the most commonly used hooks in React [19]. In this application, the useState hook was used to manage the update of object IDs in the Babylon scene, and the useEffect hook to perform various side effects, such as the scene initialization, camera control, model loading, etc., related to setting up and managing the Babylon scene.

2.1.3 3D File Formats

A 3D file format is a standardized specification that defines how three-dimensional data is structured and stored, typically in plain text or binary form. These formats encompass a broad spectrum and accommodate different aspects such as geometry, appearance, scene, and animation of 3D objects. As the 3D graphics industry has

Table 2.1: Key hooks in React.

Key Hooks	**Features**
useState	useState lets developers add state management to functional components. The useState function returns the current state and a function to update it.
useEffect	useEffect represents lifecycle methods, such as component DidMount. It is used for side effects in function components, like data fetching, setting up subscriptions, or manually updating the DOM.
useRef	useRef returns a mutable ref object. It is useful for accessing and interacting with DOM elements directly.
useContext	useContext provides a way to pass data through the component tree without having to pass props manually at every level.

evolved over the years, a variety of 3D formats have emerged for describing, storing, and exchanging three-dimensional representations of any type of object, each with its own advantages and fit for specific applications or software ecosystems [20]. Therefore, one of the challenges in this project was choosing an open 3D format suitable for efficient transmission over the network while also maintaining long-term scalability and interoperability with other software tools.

Within the realm of 3D city models, the City Geography Markup Language (CityGML [21]) format is the most prevalent storage and management technology, owing to its broad support across various professional and open-source geospatial products. It stands as the most comprehensive tool utilized by the academic community for digitizing entire cities or specific sections, integrating both semantic and geographical data [22]. CityGML is an Open Geospatial Consortium (OGC) standard whose goal is to ensure that geospatial content and services, sensor webs, and the Internet of Things (IoT) can be efficiently shared and integrated.

CityGML supports various Levels of Detail (LoD), from simple building blocks (LoD0) to detailed architectural models (LoD4). The format can express topological relationships between different structures, which can be crucial for some simulation and analysis tasks. CityGML models are typically structured hierarchically. The City Model is the topmost level, representing an entire city or region. City Objects are individual entities, like buildings, roads, water bodies, vegetation, etc. Components are parts of city objects; for example, a building object might have components like

walls, roofs, and windows (Figure 2.1 [23]).

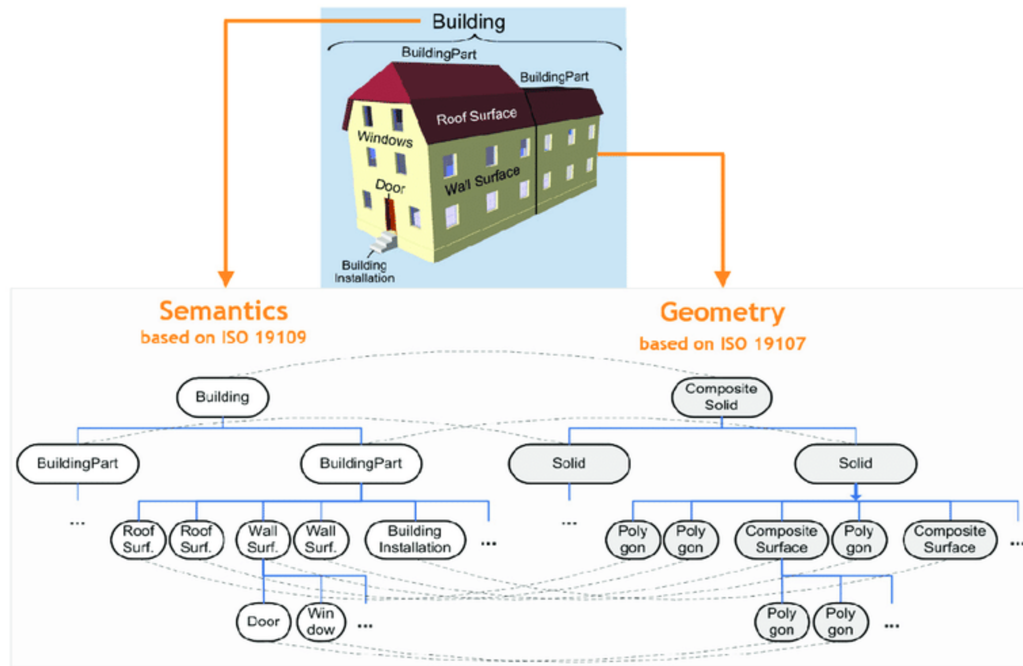

Figure 2.1: CityGML building object with aggregation hierarchy.

In addition to the 3D geometry of urban structures, CityGML also stores semantic information. City objects and their components can have various attributes, like materials, usage (e.g., residential, commercial), or other metadata. CityGML is based on the Geography Markup Language (GML), an Extensible Markup Language (XML) grammar specified by the OGC for expressing geographical features. Consequently, owing to GML's inherent features, such as the strict use of coordinates in Spatial Reference Systems and the allowance of complex 3D polygons for object modeling, CityGML is typically not employed directly for streaming geospatial datasets in server/client configurations for visualization purposes. When streaming extensive 3D city models, the data packets transmitted over the network and processed by viewing components must be immediately accessible to prevent delays, compact, simple to parse, and in a coordinate system readily suitable for rendering [20].

To publish 3D assets online, it is necessary to utilize open standards and integrate browser technologies like WebGL. A significant format in the realm of web graphics is the Graphics Library Transmission Format (glTF) [24], developed by the Khronos Group and supported by many web 3D presentation tools such as Babylon. glTF, often referred to as the "JPEG of 3D", is designed for minimal file size and fast

loading. It provides a compact and self-contained format optimized for real-time rendering, ensuring efficient transfer through fast loading and reduced computational overhead. glTF can represent various types of 3D assets and support complex 3D scenes by distinguishing between the scene hierarchy and a binary block containing vertex, index, and animation keyframe data in the form of raw byte arrays [20]. A glTF asset is described by a JSON file, which may accompany external binary data (.bin) for geometry and other heavy data, as well as external image files for textures. A JSON file outlines the node hierarchy, meshes, materials, and methods for accessing the binary block. Within this file, BufferViews and Accessors guide the parser in reading and interpreting the binary data, as well as in extracting necessary information for rendering, like a triangle mesh. This approach is very close to the OpenGL specification and the process of preparing and transmitting data to the graphics card via an OpenGL API like WebGL. Additionally, it facilitates the creation of larger data segments and enables optimization strategies that significantly enhance rendering performance, such as employing interleaved data blocks for coordinates, normals, and texture coordinates [20]. Table 2.2 illustrates the main components and their features of the glTF format and Figure 2.2 [25] demonstrates relations between top-level arrays in a glTF asset.

Table 2.2: The main components in glTF structure.

Components	**Features**
Asset	Provides metadata about the glTF and the tools that generated it.
Scenes	Describe the visual hierarchy.
Nodes	Represent objects in the scene.
Buffers	Binary data for the asset.
Buffer Views	Represent a segment of a buffer, indicating how to use the binary data.
Accessors	Explain how to access data in a buffer view.
Meshes	Define the geometry to be rendered.
Materials	Describe the appearance of objects.
Textures	Define a source image and how it's sampled.
Cameras	Define a projection (orthographic or perspective) to view the scene.
Animations	Define sequences of values for node properties, allowing for animated transformations and other effects.

As a royalty-free specification, glTF can efficiently transfer and load 3D scenes

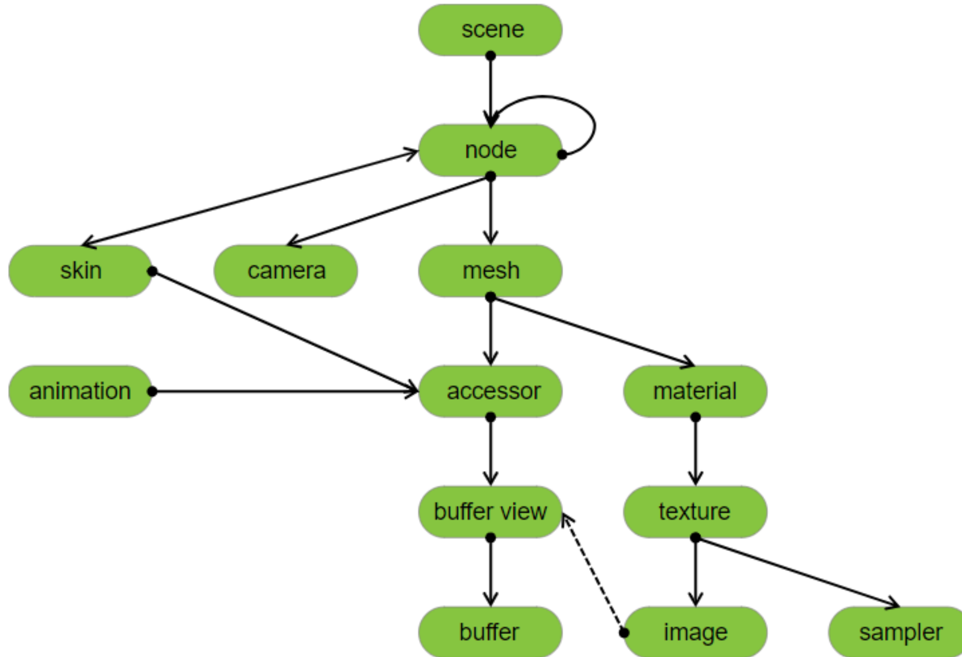

Figure 2.2: glTF object hierarchy.

and models on the web [26]. Hence, all 3D models in this project are encoded in glTF format. However, glTF itself does not contain geographic coordinate information, unlike formats explicitly designed for geospatial data. Therefore, some methods are needed to associate geographical coordinate information with the glTF model so that the glTF model can be loaded based on location information. Some commonly used methods include custom extensions, external metadata, integration with mapping APIs, and conversion tools. Chapter 3 describes in detail how to use 3DCityDB to convert the original data from CityGML format to glTF format with additional geographical coordinate information.

2.2 Related Work

2.2.1 Data Management and Transmission

While the visualization of digital 3D models has been feasible for over twenty years, initially, such applications were confined to specialized geo-visualization laboratories and relied on specific software [27]. Nowadays, with affordable hardware such as dedicated GPUs, standard computers can efficiently render large 3D models. Although

modern computing hardware is adequate, challenges such as accessible visualization software and the requirement to transfer large files persist. Web-based distribution may be a highly promising and practical approach for quickly streaming 3D digital datasets without the need to transmit raw data, particularly for broad users who need information in an understandable form instead of raw data [28].

Integrating web technologies into geo-visualization presents significant opportunities and benefits. Web technologies make visualization more accessible to a broader audience as it can be accessed from any device with an internet connection and a web browser. Web-based platforms enable real-time collaboration, allowing multiple stakeholders to interact with the same data simultaneously and make collective decisions [29]. Web technologies also support interactive visualizations, enabling users to explore data more user-driven, which can lead to discovering new insights [30]. Benefiting from the continuous expansion of the internet environment, many contents and research are being combined with the network, and the development of browser-based 3D rendering has enabled the transition of interactive 3D applications from desktop systems to browsers. The web represents a common denominator for developers, designers, and users of all types and is increasingly playing a potentially important role in creating, editing, and presenting 3D models [31].

However, challenges still remain. On the one hand, the transmission is often limited by bandwidth and server performance due to the heterogeneity and large scale of 3D geo-data. On the other hand, it is difficult to implement it on the web because of the unique requirements of geospatial applications for data manipulation and management technologies [32]. Currently, the most common approach for management and transmission is utilizing the tiling mechanism, which splits the entire data into multiple tiles with smaller file sizes compared to a single file and then stores the tile data on the cloud. This allows the employment of streaming techniques, such as loading and displaying only the tiles visible from the current camera viewpoint and unloading tiles as the camera moves. This strategy is effective in minimizing the memory footprint and maximizing visualization performance, particularly in web-based applications. Cesium.js 3D Tiles, a specification for streaming massive heterogeneous 3D geospatial datasets, has become a promising approach for the online rendering of extensive 3D city models. The primary concept behind 3D Tiles is to spatially partition and hierarchically simplify massive 3D datasets. Bringing techniques from the field of 3D graphics and built on glTF, it defines a hierarchical data structure and a set of tile formats that deliver renderable content, allowing efficient streaming

of visible tiles based on the viewer's position.

In [33], Mao et al. implemented an online 3D city model visualization framework based on 3D Tiles for energy simulation. They merged the 3D geometric data of urban objects with its simulation results as attributes or with object ID information to generate batch 3D models (B3DM) in 3D Tiles for visualization. This process began with extracting 3D geometry data from the 3D City Database software package (simply as 3DCityDB) and converting it into the OBJ format suitable for 3D models. Concurrently, a batch table for 3D Tiles was created for each 3D object in the OBJ file, utilizing their attributes from the CityGML database. Subsequently, the tool obj23d-tiles was employed to generate the 3D Tiles data in the B3DM format, where a metadata file named tileset.json was created to outline the visualization configuration, including geolocation and geometric error, for the B3DM files. In the case of extensive urban areas, models were initially divided into smaller segments, with B3DM files being produced for each segment. Moreover, models across various regions were adapted to support multiple LoDs.

In [20], Schilling et al. described an approach for streaming large and detailed 3D city models, focusing on open standards and open-source development and overcoming the challenges associated with large datasets and the constraints of web-based environments. They utilized CityGML datasets, which is the most widely used technology for storing and managing 3D city models, converted them to web-friendly formats (glTF, B3DM, 3D Tiles), and embedded them in the Cesium.js virtual globe. They introduced a series of steps for processing 3D data, including subdivision/parallelization, LoD filtering, appearance filtering, 3D polygon triangulation, coordinate transformation, texture atlas generation, and mapping of attributes and semantics. For example, the subdivision approach decomposed massive spatial data sets into minimal packets, which are non-overlapping rectangular areas of 3D geo objects in the form of an R-Tree structure, by defining a minimum tile size and/or a maximum number of objects contained. When doing format conversion, one problem to solve was how to configure georeferencing in glTF. In order to accurately represent geographic data in a global reference system, they introduced the CESIUM_RTC extension during conversion, which allows global reference points to be combined with local vertex coordinates to improve accuracy. Meanwhile, they merged glTF assets and attributes into the B3DM format to transfer objects and include 3D geographic data in Cesium.js. However, since the limitations of the batch table in B3DM cause the hierarchical relationship between objects to be lost, the semantic attributes were processed as separate B3DM

batches.

Another tiling strategy is to leverage 3DCityDB [34] to generate tile data with formats suitable for web rendering, such as KML, COLLADA, and glTF. In [23], Chaturvedi et al. exported the CityGML format data, which comprises spatial, semantic, and topological information of the city objects, through the 3DCityDB Importer/Exporter into spatial data containing a large number of tiles for efficient visualization. However, since these formats (KML/COLLADA) have no specifying locations to store additional object information, the structure and attribute information were lost during processing. Their approach was to derive semantic and structural information along with spatial information. All this information is logically linked through a shared identifier, where the semantic information is a simplified data structure with attribute-level information, while the structure information contains different aggregation hierarchies and the meta information of the derived city objects. The export results contain a visualization model and additional JSON-encoded data detailing the decomposition of all complex city objects and thematic information in a tabular format.

The initial consideration for the data tiling approach in this project was to utilize Cesium 3D Tiles, which related to the data format conversion from CityGML to 3D Tiles. However, there is limited availability of open-source tools for CityGML to 3D Tiles conversion. Based on the previous research, one approach is to upload data into Cesium Ion for hosting and conversion. The problem is that the Cesium ion account has storage space limitations and requires the consumption and purchase of Cesium ion credits when processing and storing large-scale datasets. The same situation exists on another solution, which is to use FME [35], a commercial tool that requires a license to use beyond a trial period for converting spatial data formats. Hence, these two methods were not adopted for the project. Finally, 3DCityDB was utilized to realize the tiling process and the data conversion from CityGML to glTF. The benefits of 3DCityDB rely on its open-source availability as well as the direct conversion result to glTF format, which is the format that can be loaded into a web rendering tool, such as Babylon.js, without any further conversion.

2.2.2 Rendering Approach

The advent of WebGL, eliminating the need for additional plugins or extensions, enables modern web browsers to tap into the local GPU for enhanced rendering of 2D

and 3D graphics and brings new opportunities to areas that utilize 3D technology [32], [36]. WebGL has become the predominant standard for 3D visualization on the web. Currently, all modern browsers, such as Google Chrome, Opera, Safari, and Firefox, support WebGL technology. WebGL has been employed in the growth of various web applications, especially heavy graphics applications. However, WebGL is a low-level 3D graphics API, which means developing 3D web applications using this technology can be time-consuming and, in some instances, extremely complicated. For this reason, many WebGL-based JavaScript libraries and frameworks have been developed to ease and accelerate the development of 3D web applications. These frameworks offer powerful tools and features to help developers achieve higher performance and simplify the development process. Some of these frameworks have been used to develop web applications for visualizing 3D city models based on differences in their feature sets, methods of defining 3D content, and their intended use in various applications [32].

In [32], Kramer and Gutbell compared three WebGL frameworks (X3DOM, Three.js, and Cesium.js) for creating 3D geospatial applications on the web. They defined several criteria for evaluating these frameworks based on their capacity to fulfill the essential needs of geospatial applications. They evaluated the frameworks by conducting five case studies, including "a prototype developed in the urbanAPI project," "web-based visualizations for the cities Mainz, Villingen-Schwenningen, and Wiesbaden," and "an application for visualizing large landscapes."

The first case was implemented with X3DOM, a framework that allows developers to define a 3D scene declaratively, helping structure the scene semantically. They utilized the features of X3DOM for the development of the urbanAPI project. For example, the standardized X3D format allows the importing of 3D data to the scene directly and defines and controls the scene without the need to work with low-level WebGL APIs. In addition, X3D allows metadata to be attached to objects.

The second case investigated the capability of X3DOM for visualizing large-scale geospatial data, compared to the first case, which only visualized a small part of the city. The Mainz dataset contains 147,048 buildings in an area of about 240 km2. However, for a number of reasons, declarative X3D offers few advantages over imperative APIs when working with large amounts of geospatial data. The most severe drawback was that X3DOM did not support memory management then. Even though the data was optimized before loading and the streaming algorithm was implemented, the browser crashed after a few minutes of running. Other major drawbacks include

the lack of geospatial coordinate support in X3DOM and the inability to identify individual objects and attribute displays in the scene due to the loss of structure and metadata during data optimization.

In the third case study, they developed a 3D visualization based on Three.js with a dataset similar to that of Mainz for the purpose of comparing the performance between X3DOM and Three.js. Three.js also did not support geographic coordinates and could not identify individual objects. However, it could access WebGL directly to implement memory management, so the browser no longer crashed when visualizing such a large dataset.

The fourth case was the visualization of a city model combined with a point cloud in Three.js and Potree. Potree is an open-source extension based on Three.js that renders vast point clouds and allows the combination of point clouds and 3D geometries in Three.js.

In the last case, the researchers implemented an application integrating the visualization of a high-resolution terrain, 3D geometries, and abstract information based on Cesium.js. They applied Cesium Terrain Builder/Server to process and host terrain data and combined it with georeferenced imagery to create a complete landscape. In addition, they also visualized 3D geometries, such as buildings, and geospatial objects, such as points, polygons, etc., utilizing the direct support of geospatial coordinates in Cesium.js. These features of Cesium.js, such as support for streaming loading and geospatial coordinates, improve the accuracy of geographic data rendering and maintain the high performance of the application, which is a significant advantage over previous solutions.

Finally, they evaluated their work based on a number of criteria qualitatively and concluded that each framework offers distinct advantages: Cesium is specialized for geospatial applications with streaming capabilities; Three.js offers flexibility through direct WebGL access for a broader range of applications; and X3DOM simplifies 3D scene description with its high-level approach and long-term storage compatibility. Each framework has its unique approach, goals, and target audience, with no solution fitting all needs. The choice of the proper framework depends significantly on the specific use case, considering each tool's pros and cons.

In recent years, Cesium has progressively gained traction for its capability in geospatial 3D visualizations. Cesium.js [37] is an open-source JavaScript library for creating 3D globes and 2D maps in the web browser. It applies WebGL to accelerate hardware, supports cross-platform operations, and is extensible to integrate with

other GIS and web development tools. Cesium has distinguishing features that differentiate it from other WebGL frameworks. It can stream global-scale terrain and imagery and handle a multitude of data types, from simple markers and polygons to complex 3D models and point clouds. It allows data imports from KML, ESRI Shapefiles, and JSON. It supports geospatial coordinates out of the box and allows users to build 3D scenarios by adding and managing various entities, such as earth, buildings, etc. Cesium also provides a wide range of APIs and plugins to assist developers in producing diverse custom functions and extensions, which makes it ideal for building professional applications that embed existing GIS data. Moreover, one of the core features of Cesium is that it supports real-time streaming through the 3D Tiles specification.

Many studies have utilized or extended Cesium as the platform for their web applications. Dimitrov and Petrova-Antonova used Cesium to display a user-interactive 3D city model in LoD1 that integrates the building and terrain of Sofia's Lozenets district [38]. Their 3D content was hosted by Cesium ion and accessed through its asset identifier. In [39], Mete et al. developed a 3D web application based on Cesium to visualize vectors and grid geographical space data types. In addition to the above data, they used Cesium to visualize the 3D city mode, which contains 1.1 million OSM buildings, showing the journey of the New York 3D city model. Buyukdemircioglu and Kocaman used Cesium and other tools to present a smart city project in Sahinbey District, Gaziantep City, Turkey [4]. As a part of the project, they reconstructed the data from multiple sources into the integrated city model. Utilizing Cesium as a visualization platform, they rendered multiple 3D GIS scenes, integrating existing digital city and planning models to facilitate participatory planning and decision-making by providing an accurate 3D representation of the design within its context.

Furthermore, some researchers utilized 3DCityDB Web-Map-Client (simply as 3D Web Client), a component of the 3DCityDB, as the platform for their applications. 3DCityDB is an open-source software for importing, managing, analyzing, visualizing, and exporting CityGML data [34]. With the release of version 3.3.0, the 3DCityDB suite includes the 3D Web Client, a web-based client for 3D visualization and interaction of large-scale semantic 3D city models. The 3D Web Client extends the Cesium.js WebGL virtual globe, with its primary functionality supporting and managing configurable data layers. This feature allows for the development of visualization mashups, incorporating digital terrain models, imagery data, and possibly extensive, tiled 3D

visualization models in various formats, including KML, COLLADA, and glTF [40]. It leverages the multi-threading capabilities of HTML5 to delegate time-consuming operations, such as parsing multiple 3D objects to background threads that run in parallel. Meanwhile, a separate thread tracks the interaction with the virtual camera. It is responsible for loading and unloading data blocks according to their visibility, thus enabling dynamic visualization of tiled 3D models [23], [41].

In [42], Kilsedar et al. created a tool to visualize the earth and buildings in three dimensions on the web, aimed at simulating and evaluating the effects of floods. Their data is in CityGML format; however, this format cannot be directly used with Cesium. They utilized the 3DCityDB for data storage, representation, and management, the 3DCityDB Importer/Exporter for converting data from CityGML to KML/COLLADA/glTF, and the 3D Web Client for visualization. In another study [22], Cantatore et al. constructed a simplified digital building model database with geo-referenced based on the CityGML standard, which was developed to create a virtual restoration plan for historic districts. They used the 3D Web Client platform as a visual technical support to achieve energy resilience management of historic districts, allowing all stakeholders involved in urban management to share information, strategies, solutions, and results on the network.

After researching and comparing various rendering platforms, the 3D Web Client was chosen to visualize the entire city scenario in this project. As an extension of Cesium, the 3D Web Client realizes the LoD concept and tiling management in glTF format. It is also open-source and can be combined with other platforms conveniently. Table 2.3 summarizes the frameworks that were reviewed in this section.

2.2.3 User Interaction

Besides the feature of facilitating the visualization of large and tiled data in KML / COLLADA / glTF format, the 3D Web Client also features interactivity enhancements with city models, such as object highlighting upon mouse-over and click actions [43]. It supports the explicit linking of the 3D visualization models with their thematic data, which can be achieved using (1) a Google Spreadsheet stored in the cloud or (2) PostgREST, a RESTful API for PostgreSQL [44]. For example, it employs the Google Fusion Tables API to retrieve thematic information on 3D objects. Clicking on a 3D object triggers a query to the corresponding Google Fusion Table, displaying its attributes in a dialogue box. Meanwhile, the 3D Web Client inherits abundant

Table 2.3: Comparison of WebGL frameworks for use in geospatial applications

	X3DOM	**Three.js**	**Cesium**	**3D Web Client**
Supported file formats	X3D	JSON, glTF, OBJ, FBX, COLLADA, etc.	3D Tiles, glTF, GeoJSON, KML, COLLADA	CityGML, KML, COLLADA, glTF
Georeferencing support	Limited support in X3D	Not directly support	Support	Support
Data streaming support	Not support streaming	No built-in streaming capability	Strong support for streaming large-scale geospatial datasets	Moderate support for streaming city models
Interactive capabilities	Integrates interactive 3D content directly into web pages	Provides tools for creating interactive 3D visualizations	Provides tools for interacting with geospatial data on a global scale	Specializes in visualizing and interacting with city models

functions of the Cesium virtual globe. Basically, it extends and customizes the "Cesium Viewer" composite widget shipped with Cesium, which contains a number of beautiful widgets and plugins, like switching between different imagery and terrain layers [41]. These common interactions are helpful for GIS applications and familiar to the public. However, a more realistic and natural urban exploration environment is needed as stakeholders want to participate in and experience immersively.

The combination of game engines and geographical applications has attracted widespread research interest because of the characteristics of simulating natural scenes and rich interactive functions [45]. Game engines offer broader functionalities than virtual globes, simplify the process of developing models and environments, and create richer graphical user interfaces to enhance user interactions and experiences. Its ability to process 3D scenes, such as the physics engine functions that digitally replicate the gravity, collision, and reflection of the real world, as well as high-fidelity interactive simulation of urban scenes, brings huge potential to the development of multi-functional 3D city model applications. Another vital built-in feature of game engines is the first-person view (FPV), which allows video game users to experience the scene and control their movement from a specific visual perspective [45].

The paper [46] explores the visualization of realistic city models using various OGC standards within game engines (Unreal and Unity), applied to real-world scenarios

across diverse themes like Augmented Reality, urban development, acoustic urban analysis, and open-world navigation. Notably, the third scenario highlights urban development, showcasing the potential of using a 3D gaming environment to visualize various building projects, offering an alternative to the more common 3D web globes. This setup features a controllable character that navigates through different urban development scenes. Users can alternate between first-person and third-person views and employ various modes of transportation, including walking, driving, or drone flying, to provide varied perspectives for exploring the targeted area.

In [45], Laksono and Aditya present the exploration of the Unity3D game engine for developing a 3D visualization for user interactions and environment simulations based on an integrated topographic map and 3D models. They employed Mapbox for Unity to enable georeferencing terrain data and 3D models in the game environments. They also developed custom C# scripts for user interface and camera navigation in three modes: birds-eye, first-person, and drone. The birds-eye view was obtained using the raycasting method, in which users control the camera movements, such as panning, rotating, and zooming, for navigation. The first-person view (FPV) leveraged the Unity Standard Asset, which incorporates game components that facilitate user navigation techniques and is designed for first-person shooter (FPS) controls. In this mode, users use the keyboard (W-A-S-D, shift, and arrow keys) for movement and the mouse to control the camera's viewing direction. A comparable setup was employed for the drone controller. Based on Unity's multiplatform nature, they built different versions, of which an Apache web server deployed the WebGL version, which could be accessed by users easily.

Beyond this, one of the most attractive developments in 3D city modeling application platforms is the advancement of browser-based game engines. Not only does this promise to bring game-engine-like capabilities to browser-based 3D applications, but it could also integrate visualization capabilities with future online GIS applications. Browser-based game engines developed exclusively using JavaScript infrastructure, providing the right environment for performance, ease of development, and abstraction [31]. The most representative game engine among them is Babylon.js. Babylon.js [47] (simply as Babylon) is a popular real-time engine for creating and displaying 3D content on the web. It originated from Microsoft and has highly comprehensive official documentation and consistent updates. Babylon can run on all web browsers that support WebGL and are designed to simplify the intricacies of WebGL, making 3D graphics more accessible to developers. It offers many technical

features such as effects, lights, various shaders, etc., and a bunch of built-in features. For example, Babylon has a comprehensive materials library, an integrated physics engine, advanced particle systems, a shadow generator, and virtual and augmented reality support.

However, there are limited studies on developing geospatial applications based on online game engines, especially applying real geographic data. In [6], Jurado et al. developed a web application based on Babylon for visualizing, interacting, and exploring underground infrastructures through a 3D perspective. They proposed an effective method for 3D reconstruction of underground features and data model design, and a mobile platform for real-time underground exploration. The features they applied in the application include height map generation, solid particle system, collision detection, and virtual reality. These works have shown the potential of a web-based 3D game engine in facilitating access to and the visualization and analysis of geospatial data.

In recent years, various interfaces, such as the Unreal Engine plugins for Cesium's web globe and the Unreal and Unity plugins for Esri's web globe, have been developed that open up new doors to combining the 3D geospatial capabilities with the rendering power of game engines. However, a limited use case exists for integrating browser-based game engines and virtual globes to represent large 3D city models on the web.

This project explored the integration of the 3D Web Client/Cesium with Babylon for rendering 3D city models. The 3D Web Client/Cesium was adopted as the rendering tool for large data scenarios, while Babylon was adopted to achieve scene simulation and user interaction in a selected area. The 3D Web Client matches the needs of this application as it significantly facilitates the visualization of extensive, tiled city model data in glTF format. With Babylon's rich built-in features and web-based game engine background, the application can take advantage of the game engine's visual quality and interactivity and ensure accessibility. In addition, Babylon has some key elements that differentiate it from commonly used web libraries, such as collision recognition, simulation of gravity, game-oriented cameras, etc. One of the project's goals focused on how to load real-world 3D objects based on their geospatial locations in the Babylon scene.

Chapter 3

Methodology

This project aims to visualize large 3D city models and provide dynamic interactions to users in near real-time on a web-based application. Due to the possibility of a large amount of data, efficient management of such data is a major challenge. In addition, learning how to load models dynamically to achieve smooth and acceptable performance is another goal that needs to be achieved. Providing rich interactions with city objects, such as object operation, first-person control, etc., is also essential. Due to the large size of the city model and the large number of tiles, combining consistent performance and rich interaction is a significant challenge. This chapter tries to answer these questions regarding data streaming methods, the selection of visualizing approaches, and near real-time interaction implementation.

3.1 Architecture

Based on previous studies on related work and various web technologies, the prototype architecture shown in Figure 3.1 was sketched. The architecture, which consists of the server and client sides, is designed on modern web standards, open specifications, and open-source tools and guarantees scalability. From the bottom to the top, the figure describes the workflow of the developed process. Data preprocessing is necessary because it is the prerequisite for implementing data streaming loading and format conversion. The processing of the semantic 3D city models, which are in CityGML format, is conducted in the 3DCityDB using the relational database based on PostgreSQL/PostGIS, which realizes data import and export. Then, the output data, which consists of the 3D models organized in a tiling structure and converted

as the KML/COLLADA/glTF format and spreadsheet files of thematic features, are saved online in cloud services.

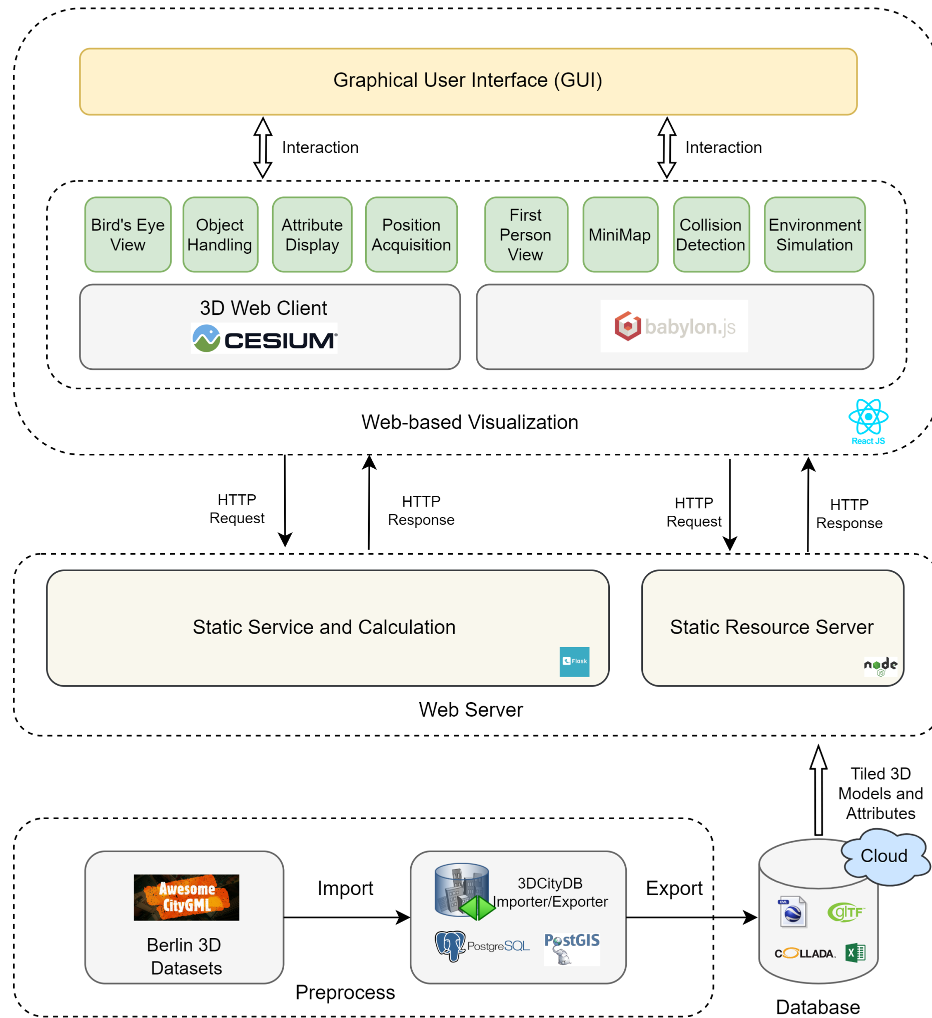

Figure 3.1: Overview of the prototype architecture.

The server layer, which stands on top of Flask and Node.js, provides flexible and scalable network services. A simple Node.js server using the Express framework was set up to serve static files for the Babylon scene. In addition, a Python server was built to process the static resources for the Cesium scene and handle calculation tasks to provide dynamic services for the Cesium and Babylon scenes.

Regarding the client side, the application stands on the shoulder of WebGL-based platforms and libraries (3D Web Client/Cesium and Babylon), which handle different rendering scenarios and multiple interactive components. Generally, the city

panorama rendering (the bird's eye view) is based on the 3D Web Client/Cesium, while the First Person View (FPV) rendering is on top of Babylon. Users can interact with the 3D content in these two scenarios, like browsing the urban environment in different modes, querying building information, moving around, etc.

3.2 Data Preprocessing

The main challenge in visualizing 3D city models on the web is efficiently transmitting the data from the back end to the front end. Due to the network's characteristics, bandwidth often limits data transmission. The scale of a 3D city model is usually massive, particularly coupled with the texture images. If all the data is transferred at one time, it will cause problems such as slow loading, page freezes, and even crashes. Segmenting one large dataset into small tiled files is an effective way of solving the issues. On the other hand, file formats of geospatial data sources are not standardized for web graphics, and Babylon only supports a limited number of file formats. Therefore, data tiling processing and format transformation are usually required to apply geospatial data to virtual 3D environments prior to web application development.

3.2.1 Data Source

The project's data source came from Awesome CityGML datasets [48] during the development and experiment. 3DCityDB [49] was utilized as the tool for data importing, managing, and exporting, delivering the data in an efficient and web-friendly format for WebGL rendering.

3DCityDB is an open-source package consisting of a database schema and a set of software tools for storing, portraying, and overseeing virtual 3D urban models on top of a standard spatial relational database [40]. It implements the CityGML standard to support semantically detailed and scalable urban objects. This database is equipped with several tools (Figure 3.2 [40]) that enable the flexible extraction of 3D city model data, accommodating thematic and spatial filtering criteria. The data can be extracted in various formats, including CityGML, 3D visualization formats (KML/COLLADA/glTF), and spreadsheet formats (CSV).

A vital component of the 3DCityDB software suite is the 3DCityDB Importer / Exporter, which was also the primary tool for processing the data in this project.

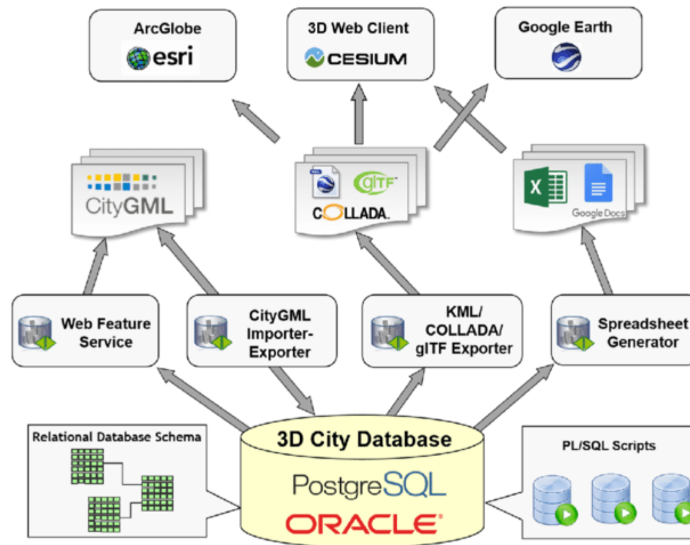

Figure 3.2: Key components of the 3DCityDB software suite.

This Java-based desktop application is a user-friendly front-end for the 3DCityDB database, complete with a graphical user interface (Figure 3.3).

Awesome CityGML lists a comprehensive collection of Open CityGML datasets from all over the world that can be imported into and used with 3DCityDB. This project chose Berlin 3D [50] as the data source because it is the most mature dataset, one of the driving factors and primary test cases for database development in the past. The Berlin City Model currently contains more than 560,000 fully textured building models in LoD2. The original dataset is an object-oriented LoD2 model for the individual districts of Berlin in CityGML format.

3.2.2 Data Processing Steps

Requirements and Preparation

The first step was setting up a 3DCityDB environment, which involves installing several relevant software components and minimum system requirements, including a Java Running Environment (JRE), PostgreSQL with PostGIS extension, pgAdmin (user interface for PostgreSQL), 3DCityDB Scripts, and 3D City Database Importer/Exporter. Meanwhile, the data was downloaded in CityGML format from two areas of Berlin: one in Schulungskurs and the other in Charlottenburg-Wilmersdorf. Their original sizes are 192 MB and 4.38 GB, respectively, and the corresponding city areas are $1km^2$ and $90km^2$, respectively, as shown in the following 2D maps (Figures

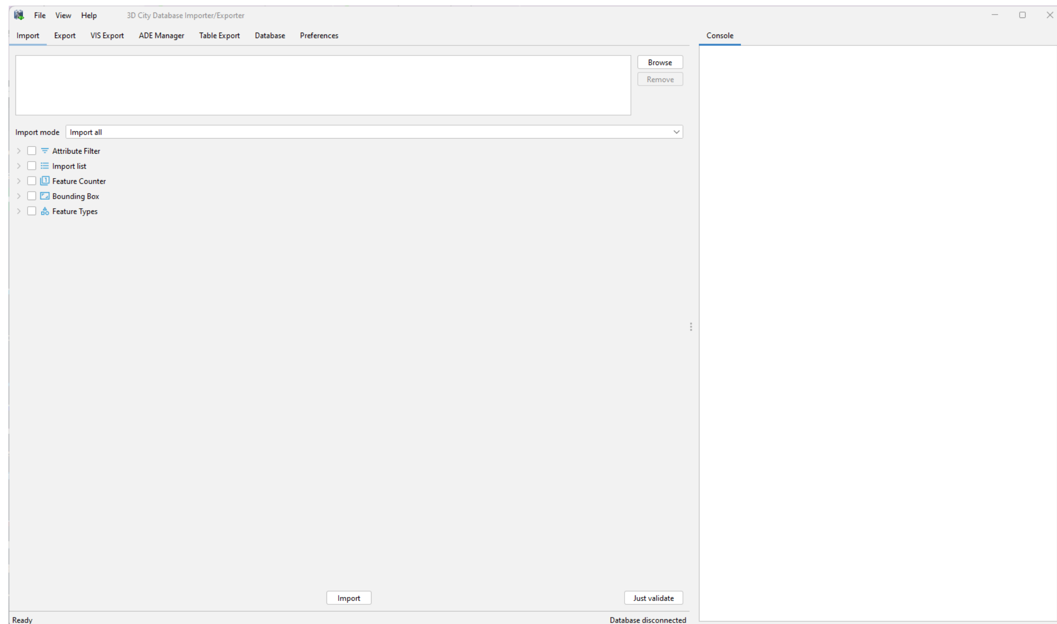

Figure 3.3: User interface of the CityGML Import/Export tool.

3.4 and 3.5).

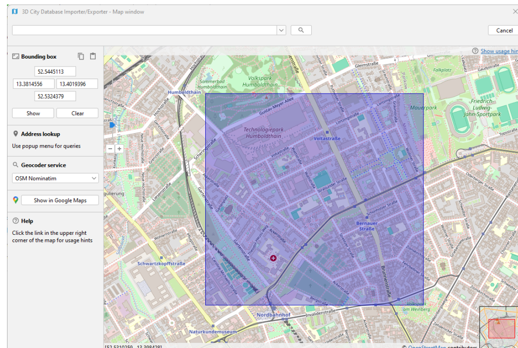

Figure 3.4: The Schulungskurs area in 2D map.

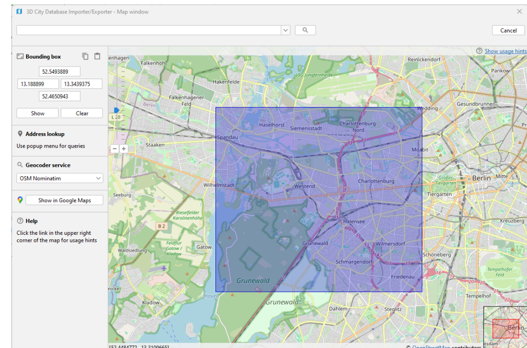

Figure 3.5: The Charlottenburg-Wilmersdorf area in 2D map.

Applying Database Schema

After installing all the required tools, the next step was to set up the database schema. The 3DCityDB database schema is a sophisticated relational database schema that organizes city model data, such as buildings, vegetation, transportation infrastructure, etc., into various interrelated tables with features and attributes. The 3DCityDB software package includes a database schema designed for spatially enhanced relational database management systems (ORACLE Spatial or PostgreSQL / PostGIS), capable of managing the geographical and geometric data essential to city

models [40]. The shell scripts shipped with 3DCityDB allow for the creation of a new 3DCityDB instance (Figure 3.6) on the spatial database system, including tables, constraints, data types, and indexes. Table 3.1 describes the main components of the 3DCityDB schema.

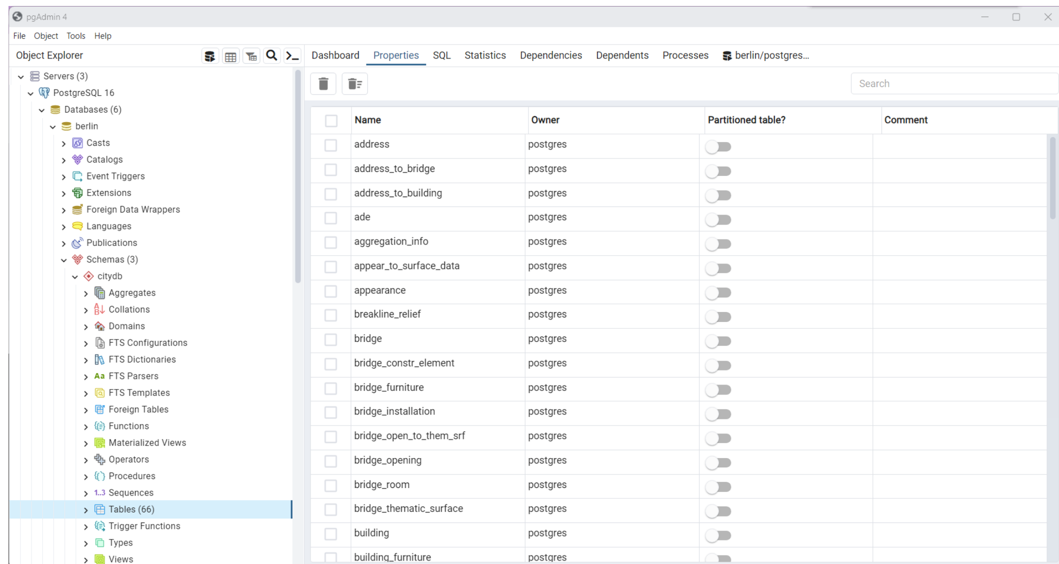

Figure 3.6: Screenshot of successfully setting up a 3D City Database instance in PostgreSQL to store CityGML data.

Import Processing

The Importer/Exporter tool supports importing CityGML, CityJSON, GZIP, and ZIP files to load 3D city model content into the 3D City Database instance. An important step before import is to validate the input files against the official CityGML XML and CityJSON schemas. Only files that successfully pass the validation can be imported into the database. Otherwise, errors will occur, and the import process will terminate. Once the import is finished, the console window will display a summary detailing the types and quantities of the top-level CityGML features that were imported (Figure 3.7). Additionally, clicking the "Generate database report" button in the Database tab will output a list of all the tables in the 3D City Database, including their total row counts, directly to the console (Figure 3.8).

3D Model Export Processing

The 3DCityDB comes with a plugin called KML/COLLADA/glTF Exporter. The spatial contents of CityGML features can be directly exported to the 3D visualization formats KML, COLLADA, and glTF on the VIS Export tab (shown in Figure 3.9) by using this plugin. The function of the visualization export allows for the thematic

Table 3.1: The main components of the 3DCityDB schema.

Component	**Description**
City Object Table	The central table stores information about the different city objects (buildings, roads, vegetation, etc.). Each city object has a unique ID and is associated with its corresponding CityGML feature type.
Geometry and Appearance Tables	These tables store the geometric representations and visual appearances of city objects. Geometry data can include points, lines, polygons, and more complex 3D shapes. Appearance data includes textures, materials, and colors used to represent city objects visually.
Relationship Tables	These tables define the relationships between different city objects, such as the connection between a building and its roof. Relationships are crucial for reconstructing the full 3D model of the city.
Metadata and Attributes Tables	These tables store additional information about city objects, such as their names, types, and other descriptive attributes. Metadata can also include information about the data source, date of creation, and more.

and spatial filters, enabling the limitation of the export to specific sections of the 3D city model content held in the database. This filtering process can be based on various criteria, such as spatial regions (bounding box), object IDs, feature types, names, levels of detail, etc.

In order to export a proper glTF format file with attached textures for realistic rendering of the building objects, some configurations need to be set up before exporting:

(1) Tiling Filter

Exporting 3D city model content into a single file can result in excessively large file sizes that are difficult to process. To enable interactive exploration with acceptable frame rates, it is necessary to geographically divide larger datasets into smaller, more manageable tiles. Tiling effectively breaks down the visualization export into multiple segments, each with a significantly reduced file size compared to a single output file. This approach facilitates streaming techniques and maintains a low memory footprint while ensuring high visualization performance, particularly in web clients.

3DCityDB accommodates tiling and can generate master files (in KML or JSON format) that dynamically load the relevant tiles based on the current camera view.

```
[10:48:46 INFO] Initializing database import...
[10:48:46 INFO] Spatial indexes are enabled.
[10:48:46 INFO] Normal indexes are enabled.
[10:48:46 INFO] Creating list of files to be imported...
[10:48:46 INFO] List of import files successfully created.
[10:48:46 INFO] 1 file(s) will be imported.
[10:48:46 INFO] Importing file: C:\Ting\Thesis\thesisproject\3dcitydb\RTG_Schulungskurs_2016\CityGML-Data\Berlin_CityGML
[10:48:50 INFO] Resolving XLink references.
[10:48:50 INFO] Resolving feature XLinks...
[10:48:51 INFO] Resolving appearance XLinks...
[10:48:51 INFO] Importing texture images...
[10:48:54 INFO] Linking texture images to surface data...
[10:48:55 INFO] Cleaning temporary cache.
[10:48:55 INFO] Imported city objects:
[10:48:55 INFO] app:Appearance: 2862
[10:48:55 INFO] bldg:Building: 954
[10:48:55 INFO] bldg:GroundSurface: 954
[10:48:55 INFO] bldg:RoofSurface: 1290
[10:48:55 INFO] bldg:WallSurface: 2846
[10:48:55 INFO] Processed geometry objects: 44610
[10:48:55 INFO] Total import time: 09 s.
[10:48:55 INFO] Database import successfully finished.
```

Figure 3.7: Types and numbers of the imported CityGML top-level features printing in the console window.

With tiling activated, the designated export area is segmented into a regular grid, creating distinct visualization datasets for each tile and form of display. Since each tile comprises only a fraction of the total city objects for export, the resulting file sizes are much smaller, allowing quicker loading times [51].

The division of tiles can be defined in two ways: automatic tiling or manual tiling (Figure 3.9). The automatic chunking was selected, and the preferred side length for each tile was set to the default value (125m) within the Fixed side length option. The entire export area was then divided by this value to derive the number of rows and columns of the final grid. The exported tile files are arranged in a hierarchical tree format within the file system, with the names of the folders corresponding to the row and column numbers of the grid used in the tiling process.

(2) LoD Filters

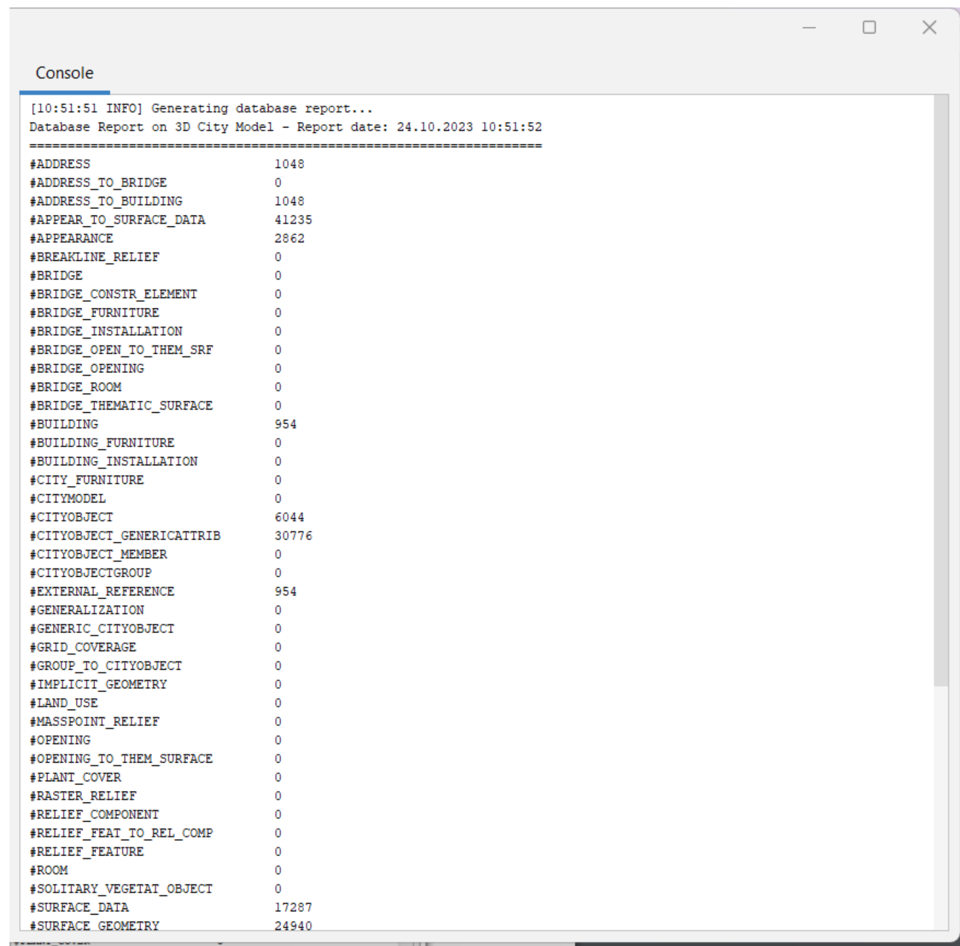

Figure 3.8: The database report displays all tables of the 3DCityDB with row numbers in the console window.

CityGML supports different levels of detail (LoD), ranging from LoD0 to LoD4, typically generated with different authoring tools [52]. Each LoD is designed for specific use cases, enhancing visualization efficiency and data processing analysis. For example, LoD0 represents a building with a 3D horizontal surface depicting its footprint and roof. At LoD1, the model includes a geometric representation of the building's volume. In contrast, the textured LoD2 model, as seen in Figure 3.10, is utilized for rendering the complete 3D city model. The Berlin CityGML dataset downloaded is in LoD2. However, since glTF lacks a LoD group node for distance-dependent switching between multiple representations, the specific LoD to be extracted from CityGML needs to be determined during the export process. Users can choose which LoD should be exported and which appearance should be used for creating the visualization models in the export dialog (Figure 3.9). Higher LoD exports result in

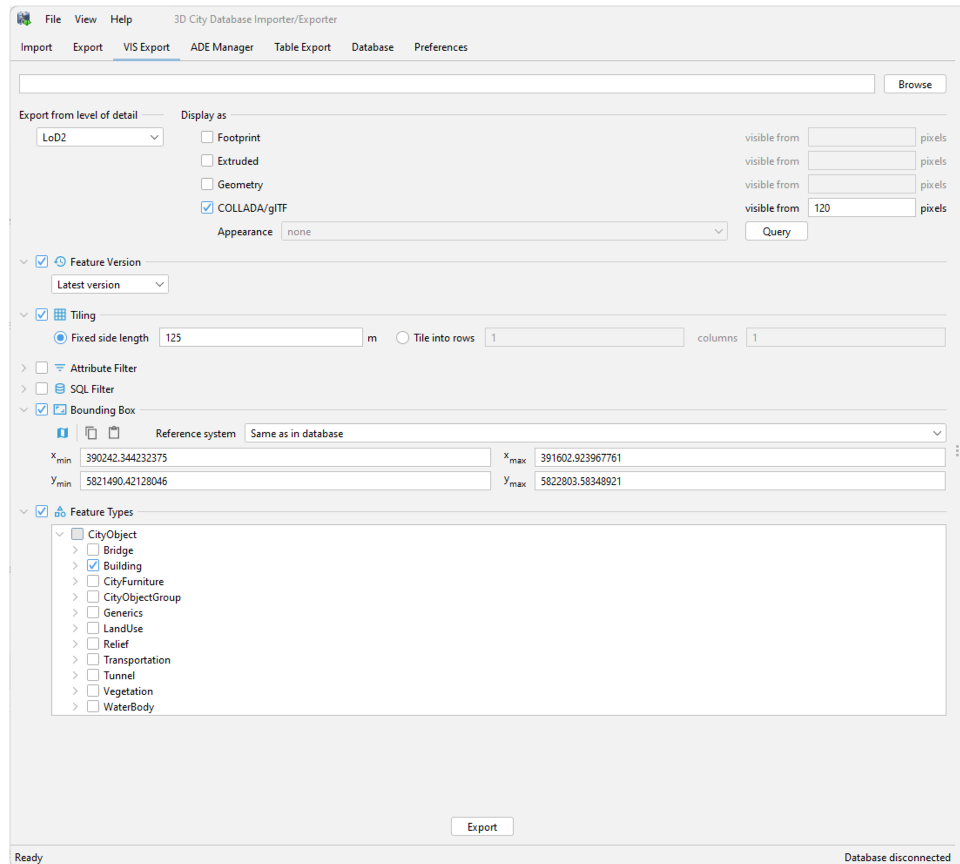

Figure 3.9: The VIS export tab of 3DCityDB Importer/Exporter.

larger export files due to their increased complexity and greatly impact viewing performance. LoD2 was chosen as the export option in this project.

In addition to specifying the LoD, the desired display form for the models must also be defined. Users have the option to select one or multiple display forms. Each chosen display form is produced based on the geometry of the city object at the specified LoD, and it also dictates the output format for the respective visualization models. This project chose COLLADA/glTF as the display form, which represents objects with full geometry and supports textures as well [53].

(3) Bounding Box Filter

The bounding box filter (Figure 3.9) operates by using a 2D bounding box, defined by the coordinates of its lower left and upper right corners, as its parameter. This filter is applied to the "envelope" column of the "cityobject" table for evaluation. The coordinate values of the bounding box filter can either be entered manually or chosen interactively in a 2D map window [54]. With tiling enabled, this project exported

Figure 3.10: The textured LoD2 models.

all 3D city models stored in the 3DCityDB instance. The bounding box values are calculated on the Database tab (Figure 3.11) and pasted into the export dialog.

(4) Feature type filter

The feature types filter allows users to limit the export to specific feature types by selecting the appropriate checkboxes. This results in the export including only features of the chosen type(s) [55]. This filter displays only top-level feature types, such as bridges, buildings, vegetation, etc (Figure 3.9). In this project, the "Building" type was selected.

(5) Other Preference Configurations

Additional settings for COLLADA/glTF export include General settings (Figure 3.12) and Altitude and Terrain settings (Figure 3.13). In General settings, an important operation is scaling texture images. Reducing the size of texture images decreases the file size of exports and enhances loading speed, although it may also lower the image quality. A scale factor of 0.4 frequently provides an effective compromise between file size reduction and image quality preservation [56].

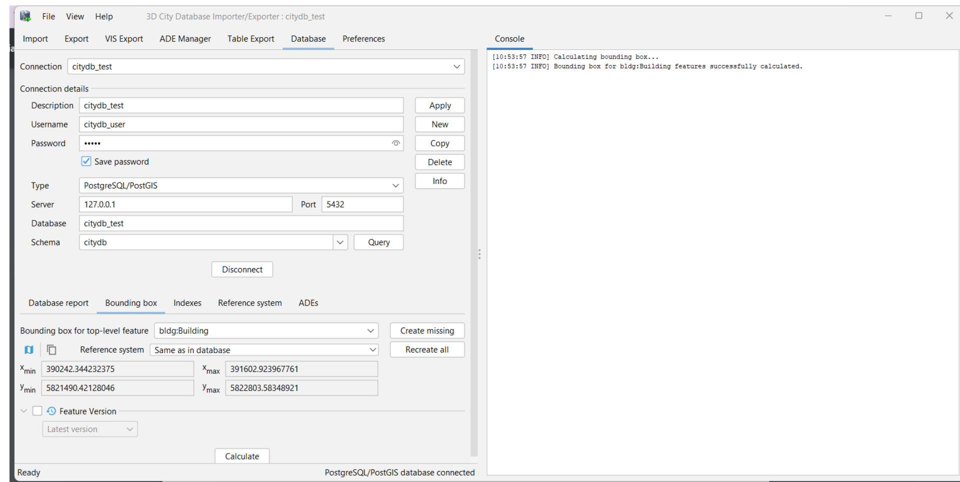

Figure 3.11: Coordinates calculation on the database tab.

Table Export

The Importer/Exporter tool enables the export of attribute data of the city objects stored in the 3D City Database into a tabular format with the help of the Spreadsheet Generator Plugin [57]. The tabular data, which can be either comma-separated values (CSV) or a Microsoft Excel (XLSX) file, can be further uploaded to services and accessed to show the information of the city objects. The Table Export tab operates the export process of the spreadsheet, in which users can create columns and define the template file to customise the attributes that they want to export (Figure 3.14). Figure 3.15 shows the exported tabular file of the building attributes in XLSX format. For each city object, the corresponding object identifier, which is stored in the GMLID column of the "cityobject" table, will consistently be exported as the first column in every output file record.

Finally, the Schulungskurs and Charlottenburg-Wilmersdorf data was exported in glTF format. The Schulungskurs file contains 954 building objects with textures and is 39.3 MB. The Charlottenburg-Wilmersdorf file contains 22,771 building objects with textures and is 926 MB. The Schulungskurs file was used during development, and both files were for performance testing. The completed exported data was saved in the file system as tile files arranged in a hierarchical tree-like structure (Figure 3.16 [53]). In this arrangement, the folder names correspond to the row and column numbers from the grid applied in the tiling procedure. The entire dataset consists of numerous tiles, with each tile file containing a set number of buildings, complete with thematic and texture details. In addition to the tile files, KML files are also

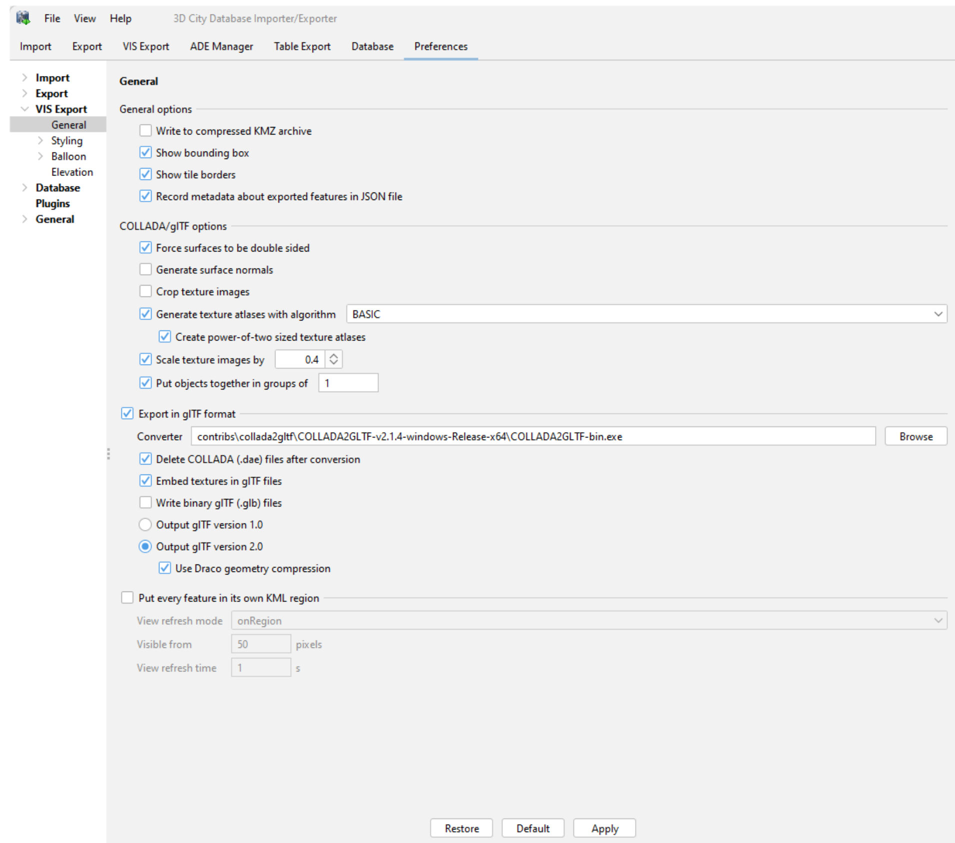

Figure 3.12: Preferences settings.

generated to point to the tile files and can be added as layers to the 3D Web Client for visualization [58]. Another vital file generated in the home directory is the city object JSON file, which comprises a list of GMLIDs for all the 3D objects that have been exported. For each 3D object, the file records the specific tile it resides in, along with its envelope attribute that delimits a bounding box in WGS84 latitude/longitude coordinates (Figure 3.17), which is very useful for determining a particular object when needed [59].

3.3 Server-side Solutions

After the processing, the data was tiled and ready to be streamed into the 3D Web Client/Cesium platform. The advantage is that the data can be completed through static resource transmission and does not need to be calculated on the server. However, loading large city-scale size 3D models in the Babylon scene could be challenging because Babylon does not have built-in support for data streaming. The application

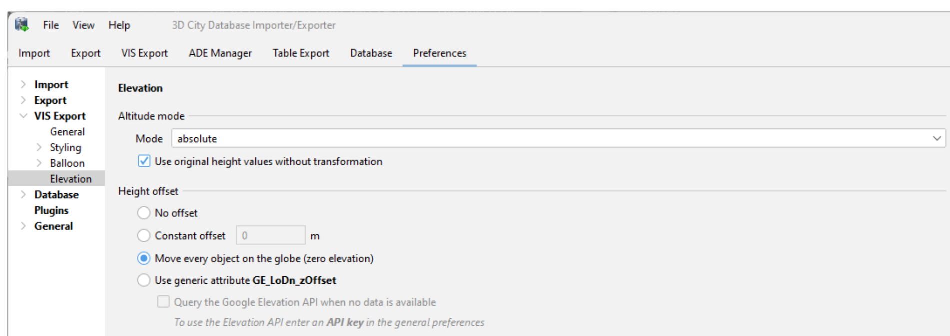

Figure 3.13: Altitude and terrain settings.

performance will significantly reduce if all the data are loaded simultaneously. A strategy was developed to load only the 3D objects surrounding the user's selected area instead of the entire city to address the issue. This mechanism (GridMap) partitions the map into smaller grids and then loads related grids depending on the position selected by the user, as illustrated in Figure 3.18. This method can avoid the slow rendering problem caused by loading too large 3D models at one time in Babylon and can also meet the user's need to explore in enough space. Loading can be achieved relatively quickly because all content is already preprocessed on the server side. Consequently, a server with convenient computing capabilities was established to handle the partitioning and calculations. Python was chosen as the programming language for building this calculation server, benefiting from its extensive libraries that offer computational advantages over JavaScript.

Specifically, a GridMap class was created, with functions that include calculating the map bounds and grid dimensions and constructing a grid map that organizes the data into grids. The first step was calculating the map bounds implemented by the calculate_bounds method. This method computes the maximum and minimum latitude and longitude values from the exported city object JSON file. It iterates over all building models, updating the maximum and minimum latitude and longitude values based on the envelope attribute of each object. The four extreme values obtained after iteration can contain all models in a large rectangular area. Next, the number of rows and columns in the grid was calculated based on the total geographic distance (in metres) covered by the data and the specified cell sizes. The two cell_sizes parametres, cell_size_lat and cell_size_lon, are introduced to define the length and width of each grid area, both set to 200 meters in this instance but adjustable according to needs. The distance between two points in latitude and longitude is calculated us-

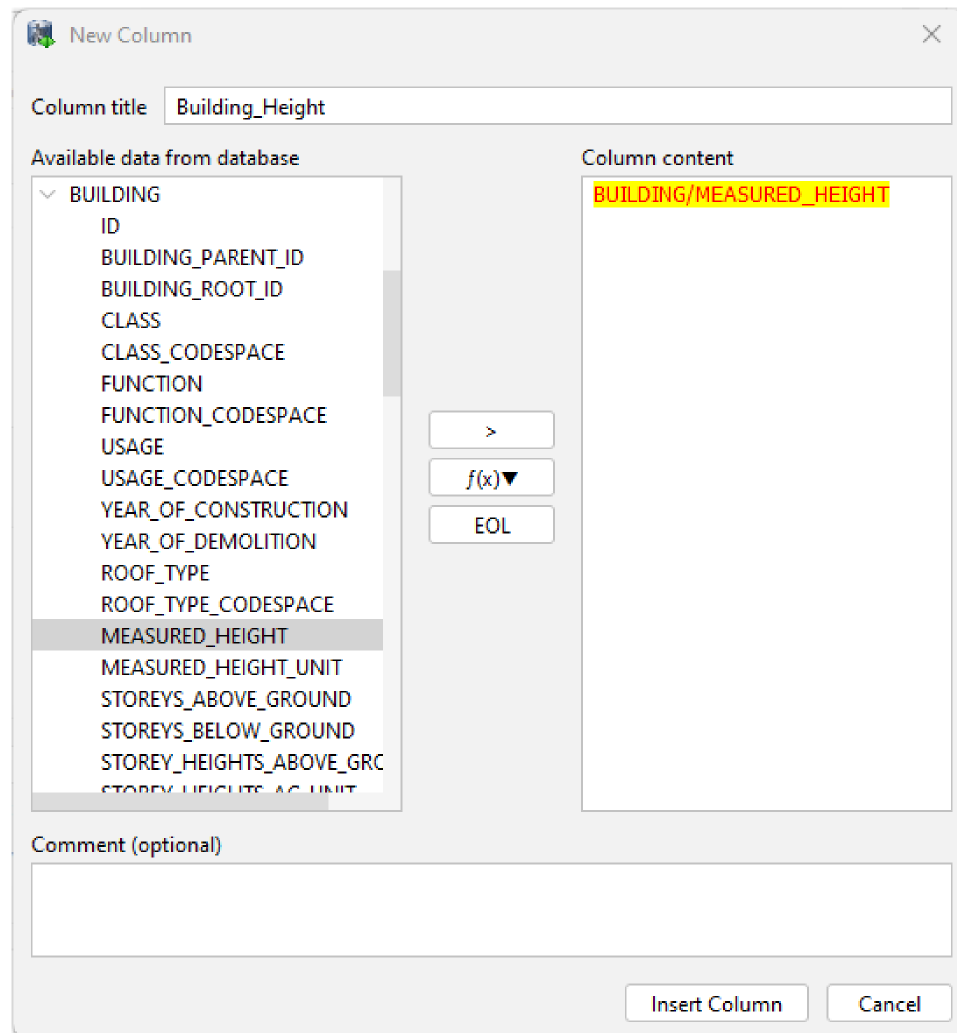

Figure 3.14: The New Column dialog window used to define the spreadsheet columns.

ing the geodesic method from the geopy library, converting these distances to a grid dimension in rows and columns.

When the user selects a location on the map and switches to a first-person scene, the client returns the location information to the server to calculate which grid should be acquired according to the selected location. Then, the server queries this grid and eight neighboring grids simultaneously (Figure 3.18) and returns the data to the front end for loading. This step was implemented using two methods: create_grid_map and get_items_for_location. The create_grid_map method organizes data into a grid structure (grid_map). It computes each model's average latitude and longitude based on its envelope and then assigns each model to the appropriate grid cell based on these average coordinates. The method iteratively checks each grid cell to see if the

GMLID	Building_Height	Building_Height_Unit	Street_Name	House_Number	City
BLDG_00030009007eef6d	17.40264	urn:ogc:def:uom:UCUM::m	Wolgaster Str.	9	Berlin
BLDG_00030009007eeee4	22.64977	urn:ogc:def:uom:UCUM::m	Rheinsberger Str.	66	Berlin
BLDG_0003000a002be2d4	24.84075	urn:ogc:def:uom:UCUM::m	Strelitzer Str., Strelitzer Str., Strelitzer Str.	43, 44, 45	Berlin, Berlin, Berlin
DEB_LOD2_UUID_8d6848d1-63dd-4aca-a2b4-54dcf3aa782c	21.885	urn:ogc:def:uom:UCUM::m			Berlin
DEB_LOD2_UUID_a6fc7b08-6dfa-49f8-81f6-9fe2e6fb94e0	13.531	urn:ogc:def:uom:UCUM::m			Berlin
BLDG_00030002004da912	22.96477	urn:ogc:def:uom:UCUM::m	Strelitzer Str.	24	Berlin
BLDG_0003000a0022366c	22.92685	urn:ogc:def:uom:UCUM::m	Bernauer Str.	86	Berlin
BLDG_0003000a00295b75	22.38171	urn:ogc:def:uom:UCUM::m	Stralsunder Str.	27	Berlin
BLDG_000300000018f9d9	24.5482	urn:ogc:def:uom:UCUM::m	Rheinsberger Str.	79	Berlin
BLDG_0003000000204893	5.05711	urn:ogc:def:uom:UCUM::m	Brunnenstr.	64	Berlin
BLDG_0003000a00223f6c	17.14926	urn:ogc:def:uom:UCUM::m	Graunstr.	7	Berlin
BLDG_0003000000204356	21.19865	urn:ogc:def:uom:UCUM::m	Brunnenstr.	73	Berlin
BLDG_000300000020b7dc	6.75036	urn:ogc:def:uom:UCUM::m	Lortzingstr.	32	Berlin
BLDG_00030009007eef9a	16.10926	urn:ogc:def:uom:UCUM::m	Wolliner Str.	45	Berlin
BLDG_00030009007eef82	17.72268	urn:ogc:def:uom:UCUM::m	Swinemünder Str.	105	Berlin
BLDG_00030009007eefb1	5.17681	urn:ogc:def:uom:UCUM::m	Brunnenstr.	119	Berlin
DEB_LOD2_UUID_f15f6e8a-4600-4c13-b110-4a0c3824c351	22.446	urn:ogc:def:uom:UCUM::m			Berlin
BLDG_00030009007eef86	17.91969	urn:ogc:def:uom:UCUM::m	Swinemünder Str.	103	Berlin
BLDG_0003000f004136e9	13.26942	urn:ogc:def:uom:UCUM::m	Usedomer Str.	11	Berlin
DEB_LOD2_UUID_4dfefd4c-373e-4049-9794-3b570cab1044	24.156	urn:ogc:def:uom:UCUM::m			Berlin
BLDG_00030009007eef89	16.29147	urn:ogc:def:uom:UCUM::m	Bernauer Str., Swinemünder Str.	25, 68	Berlin, Berlin
BLDG_00030000001901d8	6.72053	urn:ogc:def:uom:UCUM::m	Brunnenstr.	143	Berlin
BLDG_0003000e0080eccd	19.83827	urn:ogc:def:uom:UCUM::m	Schönholzer Str.	18	Berlin
BLDG_0003000e0080ecc3	19.3775	urn:ogc:def:uom:UCUM::m	Schönholzer Str.	21	Berlin
BLDG_00030009007eef2a	20.59013	urn:ogc:def:uom:UCUM::m	Rheinsberger Str., Rheinsberger Str., Rheinsberger Str.	21, 22, 23	Berlin, Berlin, Berlin
BLDG_000300000019113b	3.85313	urn:ogc:def:uom:UCUM::m	Rheinsberger Str.	15	Berlin

Figure 3.15: Exported attribute data in XLSX format.

midpoint of a model's envelope falls within that cell. If it does, the grid map adds the model to that cell. Their row and column numbers index the grid cells, and each cell can contain multiple building objects. The get_items_for_location method retrieves data in and around a specified latitude and longitude. It first determines which grid the location falls into and then checks the adjacent cells to capture data close to the boundaries of the primary grid.

In addition, this server also implements the processing of the attribute display. When the user clicks on an object in the 3D Web Client/Cesium scene, the server obtains the ID parameter in the request and checks if the ID exists in the parsed data. If yes, the server returns the attribute data of the city object to the front end, which is parsed from the CSV files exported in the data processing step.

3.4 Client-side Rendering

The second research question is how to visualize 3D city models on the client side. This question is related to the selection of open-source rendering tools that have the ability to effectively render large amounts of 3D content and realistically simulate virtual environments. Based on previous research, a solution that combines 3D Web

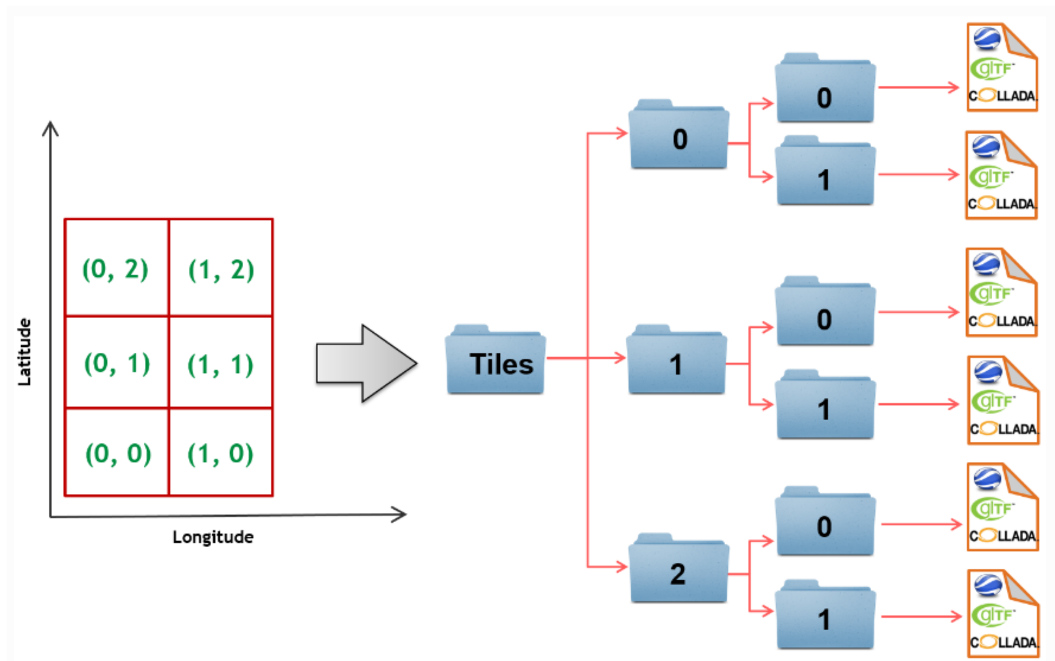

Figure 3.16: An example of a hierarchical directory structure for export files.


```
"BLDG_0003000e0000ff77": {"envelope": [13.314742345383747,52.46727969419985,13.314995100829977,52.46741055381104],
"tile": [1,67]},
```


Figure 3.17: Screenshot of a 3D object's envelope attribute in the city object JSON file.

Client/Cesium and Babylon was chosen. The application incorporates multiple approaches and implements two core rendering scenarios: one is based on the 3D Web Client/Cesium to realize the tile presentation of large-scale city models from a bird's eye view, and the other is based on Babylon to realize the presentation of a small and specific area scene for first-person simulation and exploration.

3.4.1 Bird's Eye View Rendering

The initial page (Figure 3.19) displays an earth image, a default page in Cesium. On this page, three buttons were added to control the loading of 3D objects and the changing to the first-person scene. When the user clicks the button to add glTF Data Layers, it navigates to the area based on the initial longitude and latitude values and presents the urban area from the air (Figure 3.20). Data loading for this scene is implemented through the 3D Web Client/Cesium. The 3D Web Client extends the functionality of Cesium to support the dynamic loading of large 3D city models

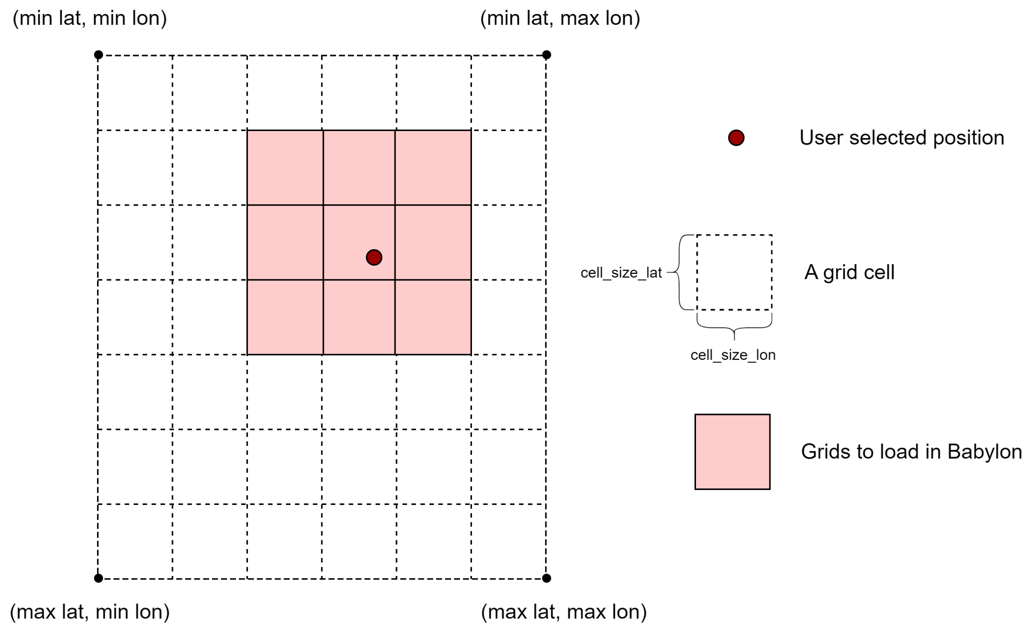

Figure 3.18: The GripMap mechanism implemented to load grids based on the user's position. The area is divided into equal-sized grids based on cell_size_lat and cell_size_lon. Nine adjacent grids are calculated and then loaded into the Babylon scene.

in the KML/COLLADA/glTF format. This method loads or unloads data onto the screen according to camera position changes, improving the overall performance and transmission efficiency. Specifically, the implementation of dynamic visualization of the 3D Web Client is dependent on the exported KML files. These files are organized within each tile file in a hierarchical directory structure that contains information about the buildings that are geographically located within the area defined by that tile [58].

Additionally, the combination of two parameters (minLodPixels and maxLodPixels), specified in the MasterJSON file, establishes the visible range's minimum and maximum limit for each data layer, managing the dynamic loading and unloading of data tiles. Through these two parameters, the 3D Web Client implements the LoD concept, loading higher-resolution data tiles for closer observations and lower-resolution tiles otherwise. This method is also widely used in 3D computer graphics and GIS to efficiently stream and render tiled datasets [59].

The above dynamic loading can be achieved through the CitydbKmlLayer class, a JavaScript class in the 3D Web Client that implements the interface Layer3DCityDB

Figure 3.19: The initial page of the application.

for processing KML/KMZ/glTF data. The CitydbKmlLayer.js file was introduced into the HTML, and a button was created to listen to the addLayers event. When the user clicks the “add glTF Data Layers” button, the data URL will be loaded to the CitydbKmlLayer as its parameter, and the glTF data will be added to the given Cesium viewer. Similarly, clicking the “remove glTF Data Layers” button can remove the loaded data layer. This function is also implemented through CitydbKmlLayer.

3.4.2 First-Person View (FPV) Rendering

In addition to the bird’s eye view rendering, which is a simple viewing offered in many applications, there was also a desire to provide multi-view points for users. Thus, another scenario is the first-person view, where users can use a first-person perspective to explore and move around the city. As a game engine, Babylon provides abundant interaction APIs to achieve game-like functionalities in 3D rendering, which is the main reason that was chosen as the first-person exploration scenario rendering library in this project. However, Babylon does not support georeferencing, and its coordinate system differs from the Cesium coordinate system. As mentioned before, when the user selects a position in the Cesium scene and switches to the Babylon scene, the front end will send this position to the back end for grid calculation. At the same time, this location will also be sent to Babylon through the URL. Therefore, converting the original geographical location of the model into the location identified by the Babylon coordinate system was the first problem to be solved in this step.

Figure 3.20: The urban scene from a bird's eye view loaded through 3D Web Client/Cesium based on original latitude and longitude points.

Furthermore, positioning multiple models to their corresponding locations in Babylon was a secondary issue that needed to be considered.

The first problem that needed to be dealt with was the coordinate conversion. Cesium uses a 3D geographic coordinate system based on the World Geodetic System 1984 (WGS84) ellipsoid. The fundamental coordinates in Cesium are Cartographic Coordinates and Cartesian Coordinates. In Cartographic Coordinates, the latitude and longitude are specified in degrees, where latitude values range from -90 to 90 degrees and longitude values range from -180 to 180 degrees. In addition, Cesium also utilizes Cartesian Coordinates in the form of 3D world coordinates (X, Y, Z) relative to the center of the earth for precise positioning and to accommodate 3D rendering needs. Cesium provides built-in functions to easily convert between these coordinate systems, such as from Cartographic (latitude, longitude, altitude) to Cartesian coordinates and vice versa, enabling straightforward integration of geographic data in a 3D scene. Whereas Babylon uses a right-handed Cartesian coordinate system adapted explicitly for web graphics within the WebGL context.

The approaches to deal with coordinate transformation can be broken down into several steps. The first was to extract geographic coordinates (latitude and longitude) from mouse clicks within the Cesium scene. The Camera.pickEllipsoid method in the Cesium API was utilized, which estimates where the ray from the camera through the mouse point intersects the ellipsoid and returns a Cartesian3 coordinate (X, Y, Z). This method converts the 2D mouse click coordinates (e.clientX and e.clientY) from

the screen into 3D Cartesian coordinates on the surface of the ellipsoid, a mathematical representation of the earth's shape and used for precise geographic calculations. Then, the ellipsoid.cartesianToCartograph method in the Cesium API was used to convert the Cartesian coordinates into Cartographic coordinates (latitude and longitude). This method returns a cartographic object containing the latitude, longitude (in radians), and altitude (in meters) derived from the input Cartesian coordinates. Next, the resulting longitude and latitude were converted from radians to degrees by using Cesium.Math.toDegrees and formatted to 10 decimal places. At this point, the processing in Cesium has been completed. The conversion results will be sent to the server side for calculation. Meanwhile, this position will also be sent to Babylon as the original point for loading 3D models. When Babylon receives this URL, it performs a simple parsing to extract the longitude and latitude values for coordinate conversion.

Next, the mapboxgl.MercatorCoordinate.fromLngLat and meterInMercatorCoordinateUnits methods from the Mapbox GL JS library were introduced to convert the latitude and longitude into Babylon coordinate values. These two methods are useful for handling coordinate and unit conversions in the Web Mercator projection. The mapboxgl.MercatorCoordinate.fromLngLat method was used to convert geographic coordinates expressed in longitude and latitude to the Mercator projection coordinate system. The Mercator projection, commonly used in web mapping, allows the Earth to be visualized on a flat surface while preserving angles and shape, making it ideal for navigation maps but distorting size and area, especially near the poles. This function returns a MercatorCoordinate object representing a point on a projected plane. The meterInMercatorCoordinateUnits method scales objects to represent real-world distances accurately on a map that uses the Mercator projection to maintain accurate scaling across different latitudes on the map.

This project used Mapbox APIs for the coordinate conversions because the initial idea was to integrate a 2D map into the Babylon scene and project the 3D objects on the map. Mapbox is a widely used tool for developers to create dynamic maps in web applications. It uses Mercator coordinates and provides convenient APIs to project real-world geographical locations on a two-dimensional plane. Combining Babylon and Mapbox within a single scene to project 3D objects according to their positions was investigated, but two issues arose. The first issue was that the 3D objects could not be perfectly aligned with the projection on the map (Figure 3.21). Various rotation methods were tested; however, adjusting the rotation for one object caused

a further deviation in the positions of other objects. The second issue involved the texture of the building objects becoming blurry and dark after loading, with some objects even distorting and deforming (Figure 3.22). These issues remain unresolved, leading to the decision to temporarily abandon the integration of Babylon and Mapbox. However, the coordinate transformation results by using the Mapbox APIs can still be retained, and the 3D objects can be directly loaded in the Babylon scene based on the converted positions. As shown in Figure 3.23, the textures are displayed normally in the scene without the combination of Mapbox. Moreover, because aligning the 3D objects with the 2D map was not a requirement, doing operations such as rotation on the 3D objects after loading was unnecessary.

Figure 3.21: The object is offset on the map when loaded into the scene combined with Babylon and Mapbox.

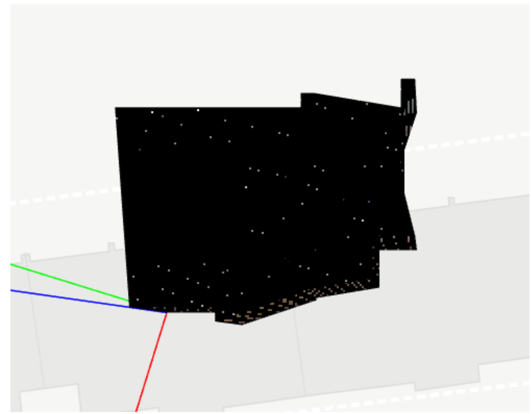

Figure 3.22: The object's texture becomes blurry and dark, and the shape becomes distorted and deformed.

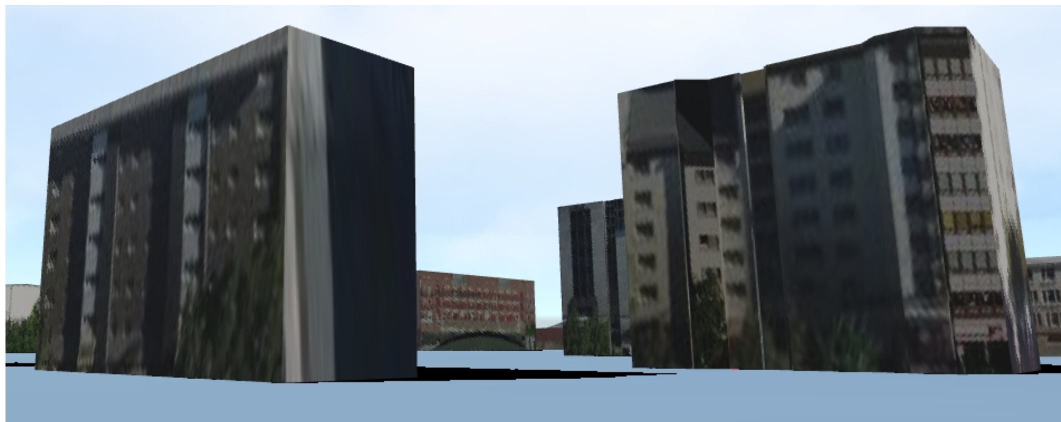

Figure 3.23: 3D buildings with texture displayed normally in Babylon without the combination of Mapbox.

The second problem when realizing FPV rendering was ensuring the meshes were correctly placed in a virtual space representing a scaled version of a real-world location. Babylon provides several methods for loading meshes, such as the SceneLoader class, AssetsManger class, etc [60]. The SceneLoader class is one of the most commonly used mechanisms for importing meshes and entire scenes into Babylon. It supports various 3D file formats and provides some common functions, such as ImportMesh, Load, Append, etc., which allows to handle different aspects of the loading process, such as asynchronous callbacks for success and error handling. Another commonly used loading method is the AssetsManger, which allows more granular control over asset loading, especially in large projects. AssetsManager allows the creation of specific tasks for loading meshes. Each mesh loading task can be managed individually, and extensive callback and error-handling mechanisms can be set up. AssetsManager was chosen to realize the loading because it ensures a smooth and responsive user experience, which is especially useful in complex 3D applications. It also enables asynchronous loading and efficient management of multiple assets, including setting tasks for each asset, defining success and error handling routines, and adjusting model properties such as shadows, collision detection, and position.

Based on AssetsManager, an asynchronous loadModels function was implemented to fetch and process 3D models according to geographic coordinates. This function realizes multiple operations. First, it retrieves query parameters from the URL to determine latitude and longitude. It makes an HTTP GET request to a server endpoint that provides 3D object data based on the specified geographic coordinates. Then, it uses a custom parseKmlData function to extract filenames (ID) and geographic coordinates from the returned data and convert it into a structured format suitable for the Babylon 3D scene. Next, it uses AssetsManager to manage loading 3D model files in glTF format into the scene. Each model is associated with a mesh task. The idea for calculating the position of each mesh was to convert the coordinates of each mesh from longitude and latitude to scene coordinates (modelCoords) and adjust the coordinates to the appropriate scale in the 3D environment using Mapbox conversion methods. Then, the point originally selected by the user's mouse is used as the reference origin coordinate (origin_coord), which is the starting point of the FPV scene. The proportional difference between modelCoords and origin_coord is used to position each mesh relative to the scaled origin point.

3.5 User Interactions

The application aims to not only visualize the simple viewing of the 3D city models but also provide rich interactions and interesting experiences for users to navigate in the 3D urban environment. The Graphical User Interface (GUI) provides buttons and instructions to access each function. Besides the scenarios switching between the bird's eye and first-person view, other rich features include the semantic information display of city objects, the location marker, the mini-map widget, collision detection, and environment simulation.

3.5.1 Navigation

Due to the advantages of 3D maps in self-orientation and spatial navigation, navigation is a core feature designed in the application, which can help users better explore the urban environment from different perspectives. The combination of two navigation modes (bird's eye view and first-person view) builds an interactive visualization that provides users with free exploration and better perception. Users can explore the urban scene from the air in a bird's-eye view, such as zooming, rotating, and panning, the same as those provided by most web 3D presenters. Different buttons in the mouse and keyboard are assigned different functions to provide user interaction. For example, panning is achieved by holding the left mouse button and moving the mouse, zooming by rolling the mouse wheel, and rotation by holding down the Ctrl key and the left mouse button.

FPV allows users to gain a walking experience and navigate their movements freely in an environment surrounded by 3D buildings (Figure 3.24). Babylon provides a wide range of cameras, among which the Free Camera creates a First-Person Shooter (FPS)-like control in the scene with the arrow keys and mouse to realize FPV. The default operations that Free Camera can achieve include moving the camera left and right with the left and right arrow keys, moving forward and backward with the up and down arrow keys, and rotating the camera around its axis with the camera as the origin by using the mouse.

3.5.2 Semantic Information Display

One of the features implemented in the application is displaying the semantic information of 3D objects in the 3D urban visualization. However, a major issue with

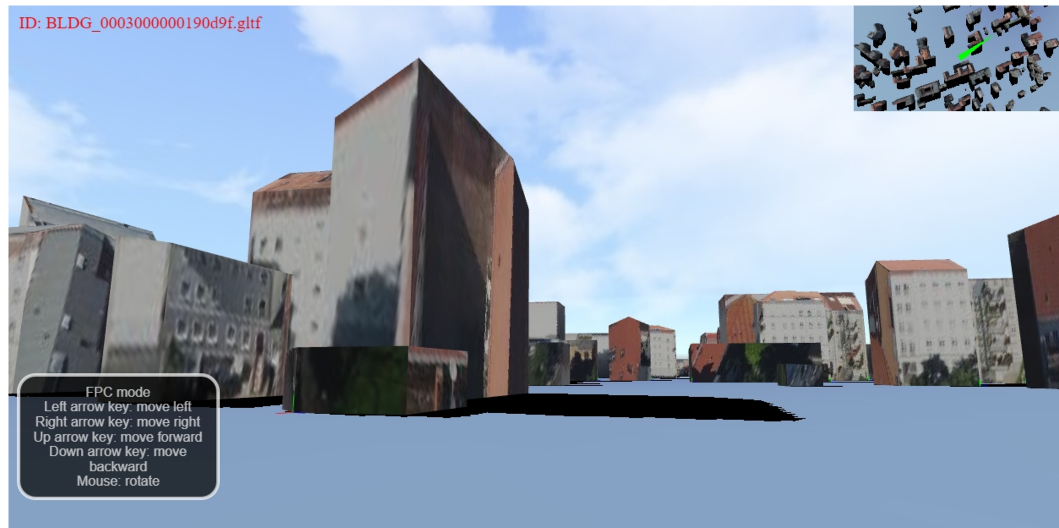

Figure 3.24: A screenshot of the FPV scenario, in which the user can explore in first-person perspective using the mouse and arrow keys. The background environment is implemented using a skybox. In the upper right corner of the screen, a mini-map widget provides visual cues as a top-down map to assist in navigating the scene. In addition, the building's ID will be displayed in the upper left corner when the user approaches a building.

the graphical visualization of semantic 3D city models is that their attribute information is either partially or completely lost in the 3D graphics formats during the conversion of data formats [23]. The 3DCity Importer/Exporter was used to export the semantic information as a separate CSV file linked with the 3D objects through the ID. Additionally, a function was implemented to request and display information about a specific entity in a user interface card. When a user clicks on a 3D object, the function initiates by fetching an HTTP GET request to a server endpoint. The server then returns data based on the provided ID. After receiving the response, the function uses ReactDOM to render a React component into a DOM container. The related semantic data for the selected building are shown in a pop window (Figure 3.25).

3.5.3 Location Marker

As mentioned before, when the user clicks a certain location on the bird's eye view interface, an event listener is registered to listen to the acquisition of the current position. Meanwhile, a function is added to this event to handle adding or updating a marker in the Cesium environment. This function aims to manage a marker high-

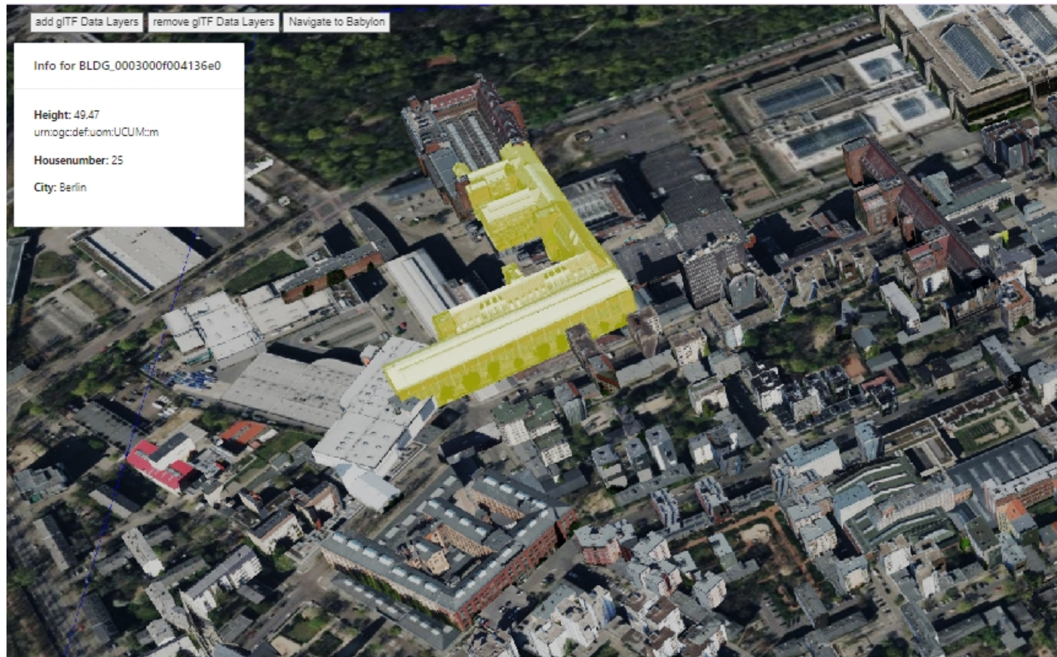

Figure 3.25: A screenshot of a pop-up window showing the selected building's information.

lighting a specific geographical location and dynamically updating the position while selecting a location for observation or switching to another navigation mode visually (Figure 3.26). Specifically, if the marker exists, it is updated to the new cartesian position without creating a new entity each time. If no marker exists, it uses the method viewer.entities.add to add the new entity to the viewer's entities collection, in which a billboard is used for loading a marker image. Billboard is an entity in Cesium that can display an image that always faces the camera at a specified location on the Cesium globe.

3.5.4 Mini-map Widget

In the upper right corner of the first-person interface, a mini-map widget (Figure 3.24) was implemented to provide visual cues as a top-down map to assist in navigating the scene during the first-person view. This mini-map was realized in Babylon by creating and manipulating a 3D mesh (a polygon) and setting up interactive camera behaviors. First, an additional observe_camera was defined for the mini-map rendering scene. Different from the primary camera, the observe_camera is a high-altitude camera that specifically represents the viewpoint of rendering the minimap scene. Then,

Figure 3.26: A screenshot shows a marker marking the location selected by the user.

a green triangular geometry was created by defining a shape array with Babylon MeshBuilder.CreatePolygon, referring to the position and orientation of the user. The interaction between the triangle and the two cameras can be controlled by configuring their layer masks during rendering. In the Babylon scene, each object has a layerMask property that specifies the layer or layers to which the object belongs [61]. During rendering, Babylon compares the layerMask property of each object to determine whether to include or exclude it from the rendering process. If the layer masks of two objects have at least one common bit set, they will interact with each other during rendering. Otherwise, they will be rendered independently. This feature is useful for various scenarios, such as rendering specific objects independently of others or applying post-processing effects selectively to certain objects. The triangle and the primary camera were set on two separate layers so that the triangle would not be rendered in the primary camera scene. Meanwhile, the observe_camera was set on the same layers as the triangle and the primary camera so that the triangle could be rendered in the observe_camera scene, and the observe_camera could also follow the primary camera.

Next, in order to allow the green triangle, which represents the user, to move with the primary camera in real-time, a callback function was implemented to update the

position of the triangle to ensure that any changes can be immediately visually reflected on the screen. The scene.onBeforeRenderObservable.add() method realizes the execution of the function before each frame is rendered, guaranteeing dynamic updates of object properties (such as position, rotation, etc.). The triangle's position was first synchronized with the primary camera to ensure it matched the camera's current position. Subsequently, the triangle was always oriented to face the direction in which the primary camera was pointing. This can be achieved by setting a directional vector that matches the quaternion extracted and created from the camera rotation, aligning it with the camera's current viewing direction. The triangle is oriented to face a point directly ahead based on the updated directional vector by calling the Mesh.lookAt method. In Babylon, the Mesh.lookAt method is used to orient a mesh (object) to face a specified target point or direction in 3D space. This method is commonly used to make an object, such as a camera or a model, look towards another object, a position, or a direction.

Through the above settings, the mini-map feature offers a secondary perspective for the first-person scene. The scene remains interactive and engaging by updating the user's position and orientation in sync with the camera and employing an overhead observational camera, ensuring a seamless and immersive experience for users.

3.5.5 Collision Detection

Collision detection is one of the essential features to achieve an interactive experience. Babylon provides built-in functionality to simulate camera collisions with meshes. The first step is to define the gravity vector, a Vector3 object used to specify the direction and speed of the camera, and then apply the scene's gravity to the camera. Babylon follows a simple gravitational model that scene.gravity represents a constant velocity, and it is measured in units/frames. The next step is to define the ellipsoid around the camera (Figure 3.27 [62]). The ellipsoid properties have a default size of (0.5, 1, 0.5), which can be changed to resize the ellipsoid as needed. The final step is to declare which meshes could collide with the camera [62]. In this project, the ellipsoid size was adjusted to (1, 1, 1) to avoid adhering to the ground and declared that all 3D buildings and the ground could collide with the camera. After all the steps are set, a collision event is triggered whenever the camera's ellipsoid comes into contact with another mesh to prevent the camera from penetrating this mesh.

In addition, another feature was implemented to extract the ID information for

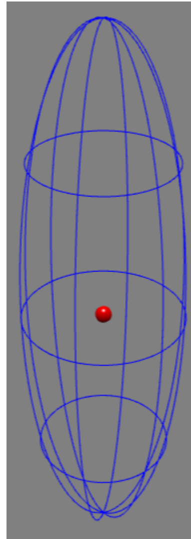

Figure 3.27: The ellipsoid around the camera represents the player's dimensions for collision detection. When a mesh comes in contact with this ellipsoid, a collision event will be triggered to prevent the camera from getting too close to the mesh.

each building by leveraging intersecting collision detection with the buildings. When the user moves and approaches a building in the first-person scene, the information about the building will be displayed in the upper left corner of the screen while triggering collision detection (Figure 3.24). This feature gives users a better awareness of their location and the surrounding environment when exploring first-person scenes. The initial idea was to display the name of the building, but since the dataset lacks this attribute, the building's ID attribute was used instead. Should a dataset include the building's name attribute in the future, the ID attribute can be easily replaced by the name attribute. This feature was implemented using the "useState", a React Hook that allows functional components to manage state. When the collision is detected, the state variable "currentFilename" will be updated using its corresponding setter function "setCurrentFilename".

3.5.6 Environment Simulation

To enhance the realism of the FPV environment, shadows were generated for objects, and the sky was simulated by adding a skybox. In Babylon, shadows are dynamically produced based on the light defined in the scene. A shadowGenerator object is required to generate shadow effects. This process involves creating a shadow map, a representation of the scene from the light's point of view. The shadowGenerator has

two parameters: the size of the shadow map and the light used for computing the shadow map [63]. The helper function, addShadowCaster, adds each mesh and its descendants to the list of shadow casters, defining whose shadows will be rendered. Finally, the ground and mesh parameters were set to true to define where the shadows will be displayed.

In addition to shadows, the skybox was utilized in the FPV scene (Figure 3.24). A skybox is a large cube that surrounds the entire scene, onto which a panoramic texture can be applied to simulate distant scenery such as the sky, mountains, or other environments. Using skyboxes for backgrounds makes rendering easier and faster than using 3D objects. In Babylon, skyboxes commonly employ CubeTexture to simulate a pseudo-reflection texture on a large cube. By default, the CubeTexture constructor accepts a base URL and appends "_px.jpg", "_nx.jpg", "_py.jpg", "_ny.jpg", "_pz.jpg", and "_nz.jpg" to load the six sides of the cube, corresponding to the positive and negative x, y, and z directions (Figure 3.28 [64]). Creating a skybox needs to follow several steps. The first step is to create a box and disable the backface culling. Backface culling is a technique used in computer graphics to improve rendering performance and efficiency by not rendering the faces of 3D objects that are not visible to the camera. Afterward, the infiniteDistance property of the skybox is set to true to ensure the skybox follows the position of the camera. In addition, all light reflections on the box must be removed so that the sun will not reflect on the sky. Finally, the sky texture is applied, and the coordinatesMode is adjusted to SKYBOX_MODE, enabling the texture to be directly applied to the cube rather than simulating the reflection [64]. Besides the manual method, Babylon also provides a Skybox Helper, the createDefaultSkybox method, to help create a skybox quickly when an entire environment is not needed.

From version 6.27.0, Babylon introduces a ground projection feature that allows simulating a ground surface within the skybox. This feature helps provide a smooth transition from the ground to the environment, grounding models without the need for extra meshes or textures. This new feature was adopted in the project to create the skybox, while the steps are similar to creating a skybox. First, a box was created below the objects and ensured the bottom face was coplanar with the fake ground to support shadows without distortions. This box is oriented to allow viewing from within, simplifying the setup. Next, a "BackgroundMaterial" was created to support ground projection, setting the "enableGroundProjection" property to true. Adjusting the "projectedGroundRadius" and "projectedGroundHeight" properties can control

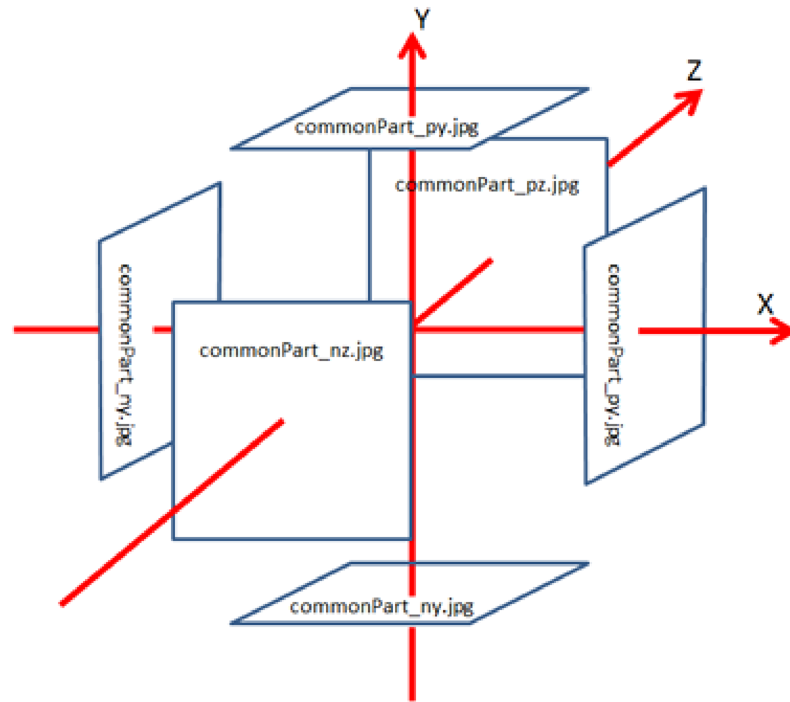

Figure 3.28: The CubeTexture images must have six textures corresponding to the six faces of the cube.

the appearance of the ground simulation within the skybox. Finally, a unique sky texture was applied to the "BackgroundMaterial." This texture should be prepared as a skybox in a dedicated "skybox" directory. It provides the environment projection within the skybox, completing the ground projection setup.

Chapter 4

Performance Evaluation

To investigate the application's performance, the Schulungskurs and Charlottenburg-Wilmersdorf, Berlin datasets were utilized as a test. The smaller dataset contains 954 buildings with textures, and the data size is 39.3 MB. The larger dataset contains 22,771 buildings with textures, and the data size is 926 MB. Experiments were performed on a personal laptop with an Intel® Core (TM) i7-10750H CPU, 16GB RAM, an Intel® UHD GPU, and an NVIDIA GeForce GTX 1650 GPU. The web browsers used in the experiment were Google Chrome (version 124.0.6367.203).

One of the research contributions was implementing a GridMap mechanism for efficiently loading 3D objects in Babylon. When GridMap is enabled, only nine neighbouring grids with 3D objects, the coordinates of which are selected depending on the user's position, are loaded in Babylon. By contrast, when GridMap is disabled, all 3D objects will be loaded in Babylon, which may cause latency when the dataset is large. The performance was compared in terms of loading time, memory usage, and frames per second (FPS) in the condition that GridMap was enabled or disabled. Loading time represents the time required to complete the loading of the entire Babylon scene, and memory usage represents the amount of memory consumed during the loading process. Fast load times lead to a seamless and enjoyable experience, which can result in higher user satisfaction. Efficient memory usage ensures the application runs smoothly without causing the browser to slow down or crash. In addition, FPS represents how many frames are displayed on the screen in one second and affects the smoothness and responsiveness of the visual experience, which is commonly used as a benchmark in testing the efficiency of software applications as well. Generally speaking, 30 FPS is a threshold [65]. The higher the FPS value, the smoother the rendering. When the FPS is above 30, the human eye can see a smooth picture, while

if it is below 30, the screen could be choppy or stuttered, which can be noticeable and potentially uncomfortable for users.

Table 4.1 records the test results. Both datasets were tested with GridMap enabled and disabled and each case ten times. It was observed that when GridMap was enabled, the loading time, memory usage, and FPS were maintained within a relatively good range regardless of the size of the dataset. When GridMap was disabled, as the dataset increased, the loading time and memory usage increased significantly while the FPS decreased significantly. Several issues arose when testing a large dataset. The first issue was that the page failed to load and displayed nothing (Figure 4.1), which happened in four out of ten tests. Although the other six tests showed successful loading (Figure 4.2), errors also occurred during the loading process (Figure 4.3), indicating that not all models were appropriately loaded. For example, in some tests, no models appeared around the user origin location, with only a few models in the distance (Figure 4.4).

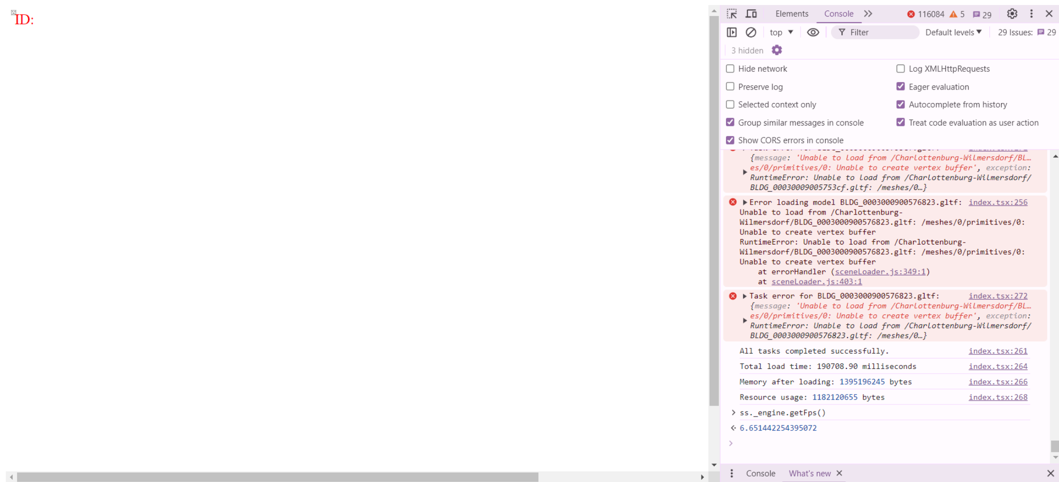

Figure 4.1: Screenshot of the page failed to load and displayed nothing during the tests of the Charlottenburg-Wilmersdorf dataset with GridMap disabled.

Table 4.2 presents the average loading time, memory usage, and FPS values, which are further compared in Figures 4.5, 4.6, and 4.7. The comparison is stark: when GridMap was enabled, the average loading times of the two datasets were 8.71s and 6.27s, respectively, which is close to the upper limit of the acceptable loading time of web applications [66]. However, when GridMap was disabled, the loading time was far beyond the acceptable range for users, reaching 209.87s for the large dataset. In terms of memory usage, application consumption with GridMap enabled was kept

Table 4.1: Test results of two datasets in conditions of GridMap enabled and disabled, separately, and each set of tests repeated 10 times.

Dataset	**No. of Tests**	**GridMap Enabled**			**GridMap Disabled**		
		Loading Time (S)	**Memory Usage (MB)**	**FPS**	**Loading Time (S)**	**Memory Usage (MB)**	**FPS**
Schulungskurs (954 buildings, 39.3 MB)	1	4.67	50.27	60	26.51	272.36	20
	2	5.45	71.51	52	29.62	287.65	16
	3	6.79	19.02	43	24.9	226.42	22
	4	10.49	116.15	28	35.02	260.62	16
	5	11.97	99.89	38	36.3	273.79	13
	6	11.09	77.47	29	35.63	212.62	13
	7	5.85	73.61	43	40.62	232.62	11
	8	8.02	82.39	36	47.66	239.61	9
	9	6.76	82.48	38	37.64	221.99	14
	10	16.02	134.97	20	37.5	228.54	12
Charlottenburg-Wilmersdorf (22,771 buildings, 926 MB)	1	6.53	65.59	45	failed	failed	failed
	2	2.11	74.25	60	139.22	526.55	9
	3	4.46	57.49	50	155.99	346.22	8
	4	6.28	126.79	41	failed	failed	failed
	5	4.92	77.47	55	failed	failed	failed
	6	6.62	68.46	36	230.12	530.19	5
	7	7.79	96.53	24	259.2	483.4	5
	8	8.77	18.62	28	259.71	550.93	5
	9	9.65	80.72	26	214.96	509.61	6
	10	5.56	71.9	48	failed	failed	failed

within 100MB, more in line with the memory usage of lightweight web applications. When GridMap was not enabled, the two average memory usages reached 245.62MB and 491.15MB, respectively, and the resources occupied were relatively large. The FPS remained above 30 when the GridMap was enabled and was far lower than 30 when the GridMap was disabled. It can be concluded that applying GridMap can significantly reduce loading time and memory consumption and increase FPS, ensuring standard rendering and maintaining a better user experience.

Table 4.2: The average values of loading time, memory usage, and FPS for each dataset with GridMap enabled and disabled.

Average Value of	**Loading Time (S)**		**Memory Usage (MB)**		**FPS**	
GridMap Enabled	Yes	No	Yes	No	Yes	No
Schulungskurs	8.71	35.14	80.78	245.62	39	15
Charlottenburg-Wilmersdorf	6.27	209.87	73.78	491.15	41	6

Figure 4.2: The console window shows the time, memory, and fps information when loading the Charlottenburg-Wilmersdorf dataset with GridMap disabled.

Figure 4.3: The failure messages of object loading during the same test.

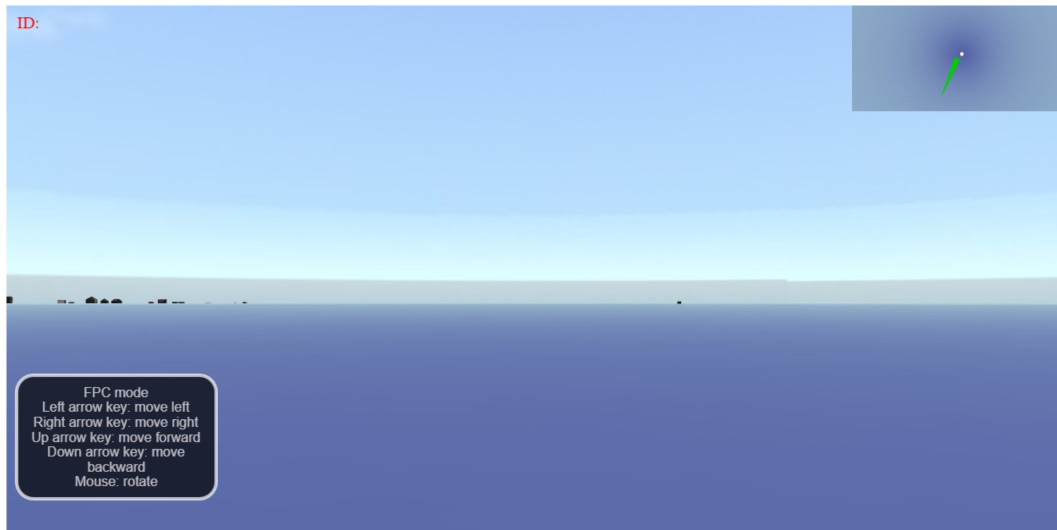

Figure 4.4: The models near the origin location failed to add to the scene properly when loading a large dataset with GridMap not enabled.

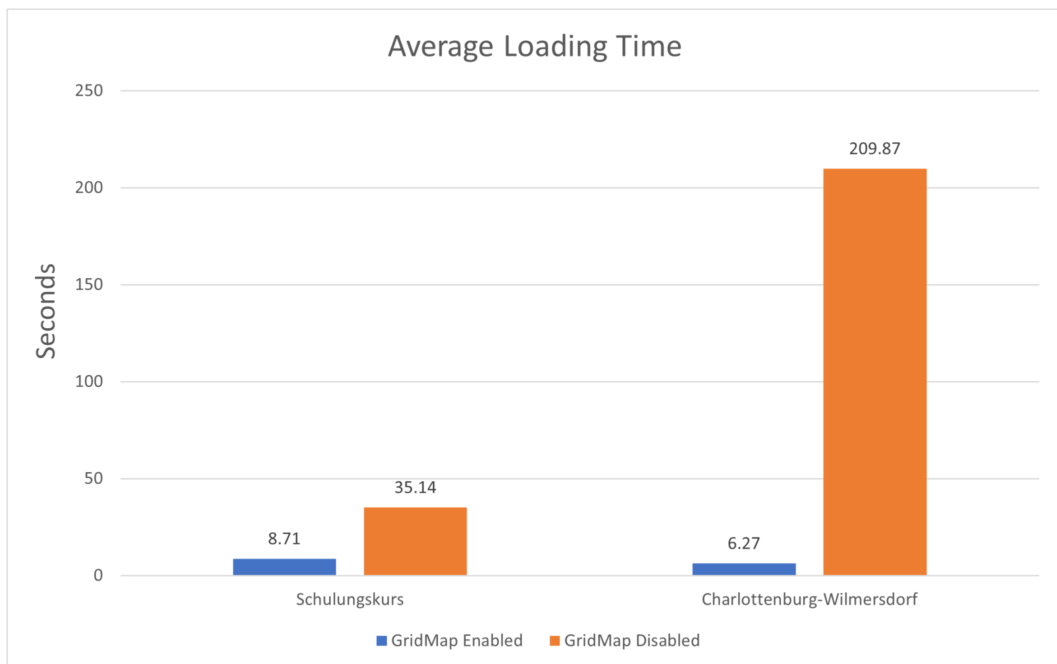

Figure 4.5: Comparison of average loading time of two datasets with GridMap enabled and disabled.

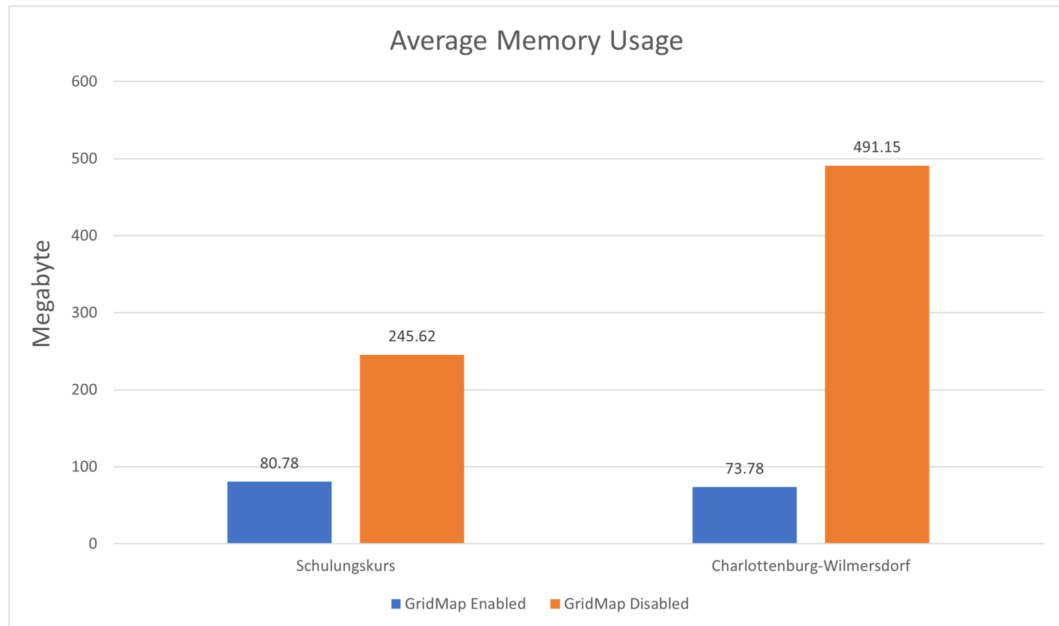

Figure 4.6: Comparison of average memory usage of two datasets with GridMap enabled and disabled.

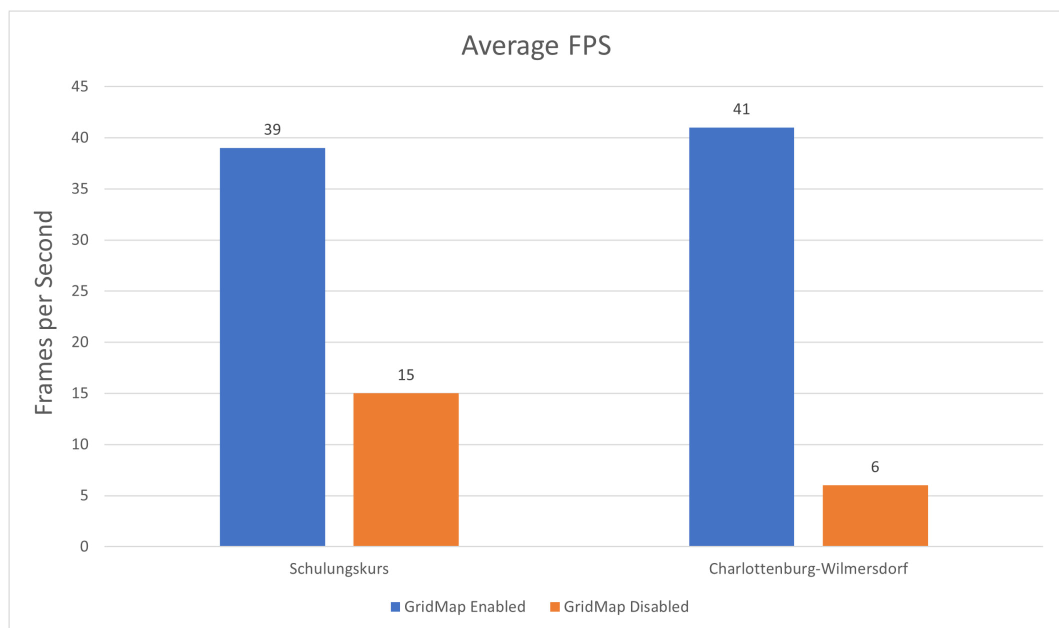

Figure 4.7: Comparison of average FPS of two datasets with GridMap enabled and disabled.

Chapter 5

Conclusion and Future Work

3D city model representations have been widely adopted in many fields, where they are used to analyze, plan, and communicate information about urban areas and their numerous components. Compared with traditional desktop applications, the built-in capabilities of WebGL in modern web browsers provide a viable platform for publishing 3D city models online.

This project proposed a prototype architecture for the visualization and interaction of urban spaces in 3D. The application was designed around web standards, open specifications, and open-source tools. In addition, the application combined two rendering tools based on WebGL. The 3D Web Client/Cesium was used to guarantee robust streaming transmission of data on the web. Babylon was used to realize environment simulation and dynamic interaction in a selected area. The data was delivered in glTF format and underwent a series of processing and conversions before delivery to avoid rendering bottlenecks.

Several functionalities (summarised in Table 5.1) were integrated into the application, and a GUI was built to allow users to explore and interact with the 3D urban scenarios. This study demonstrated that a rich 3D urban virtual experience can be achieved by adopting advanced web technologies and combining two WebGL-based rendering tools. Moreover, a central contribution of this study is to propose a GridMap mechanism that not only solves the problem of loading 3D objects with accurate geographical location information in the Babylon scene but also avoids inefficiency issues caused by loading too many redundant 3D objects, such as lagging, long time loading, and excessive memory usage. Public datasets were used to evaluate the application's performance. When enabling GridMap, 3D models can be loaded into the scene efficiently. The loading time and memory usage are maintained in an

efficient range, and the FPS value is maintained within a range that ensures smooth interaction. Especially when loading large datasets, the loading time, memory usage, and FPS of using GridMap significantly differ from those without GridMap. The results for GridMap enabled and not enabled are 6.27 seconds vs. 209.87 seconds (average load time), 73.78 MB vs. 491.15 MB (average memory usage), and 41 vs. 6 (average FPS). The results showed that the application can perform effectively with the GridMap mechanism.

Table 5.1: Functionalities implemented for user interaction.

Functionality	**Description**
Navigation	Provide two navigation modes: the bird's eye view and FPV. In the bird's eye view, users can explore the entire scene from the air by zooming, panning, and rotating. In the FPV, users can gain a walking experience around the buildings by using the arrow keys and mouse.
Semantic information display	Upon selection, highlight a building and show its semantic information, such as ID, height, and city, in a pop-up information box.
Location marker	Mark and indicate a location on the map upon the user's selection.
Mini-map widget	Provide visual cues as a top-down map to assist in navigation during the FPV mode.
Collision detection	Prevent intersections with objects when exploring in FPV mode. Meanwhile, this function triggers an event to display the current object's ID.
Environment simulation	Produce a realistic environment by generating objects' shadows and adding a skybox to simulate a sky.

However, as a web application, a potential issue may be the performance of presenting a 3D environment depending on the user's computer hardware and internet capabilities. For example, loading for 3D visualization could be challenging due to a poor internet connection. In particular, improvements in rendering quality are often accompanied by performance degradation. Hence, achieving a balance between picture quality and loading efficiency is an exciting study in the future. Another limitation of the application is that the problem of accurately projecting 3D objects on a 2D map in Babylon has not been solved. It could be considered a research direction for future work on improving the effect of scene presentation. Future works could also focus on extending and integrating the application with other domains, such as

traffic patterns and economic activities, to simulate and provide a holistic view of urban dynamics. Finally, leveraging the game engine's capabilities, the urban scenario could further serve as a background for storytelling and narrative cartography, allowing stakeholders to place stories in actual locations or map real-life narratives.

Appendix A

Acknowledgment of AI Assistance

This report acknowledges the use of OpenAI's ChatGPT to improve the clarity and structure of the writing. ChatGPT provided suggestions for sentence rephrasing, grammar corrections, and organizational improvements, which were reviewed and incorporated by the author.

References

[1] F. Biljecki, J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin, "Applications of 3d city models: State of the art review," *ISPRS International Journal of Geo-Information*, vol. 4, no. 4, pp. 2842–2889, 2015.

[2] S. Saran, K. Oberai, P. Wate, A. Konde, A. Dutta, K. Kumar, and A. Senthil Kumar, "Utilities of virtual 3d city models based on citygml: Various use cases," *Journal of the indian society of remote sensing*, vol. 46, pp. 957–972, 2018.

[3] J. Stoter and S. Zlatanova, "3d gis, where are we standing," in *Proceedings on the ISPRS Joint Workshop on Spatial, Temporal and Multi-Dimensional Data Modeling and Analysis, Cardiff, UK*, pp. 5–8, 2003.

[4] M. Buyukdemircioglu and S. Kocaman, "Reconstruction and efficient visualization of heterogeneous 3d city models," *Remote Sensing*, vol. 12, no. 13, p. 2128, 2020.

[5] S. J. Buckley, J. A. Howell, N. Naumann, C. Lewis, M. Chmielewska, K. Ringdal, J. Vanbiervliet, B. Tong, O. S. Mulelid-Tynes, D. Foster, *et al.*, "V3geo: A cloud-based repository for virtual 3d models in geoscience," *Geoscience Communication Discussions*, vol. 2021, pp. 1–27, 2021.

[6] J. M. Jurado, L. Ortega Alvarado, and F. R. Feito, "3d underground reconstruction for real-time and collaborative virtual reality environment," 2018.

[7] P. Greenwood, J. Sago, S. Richmond, and V. Chau, "Using game engine technology to create real-time interactive environments to assist in planning and visual assessment for infrastructure," in *18th World IMACS/MODSIM Congress*, pp. 2229–2235, 2009.

[8] J. Keil, D. Edler, T. Schmitt, and F. Dickmann, "Creating immersive virtual environments based on open geospatial data and game engines," *KN-Journal of Cartography and Geographic Information*, vol. 71, no. 1, pp. 53–65, 2021.

[9] M. Jazayeri, "Some trends in web application development," in *Future of Software Engineering (FOSE'07)*, pp. 199–213, IEEE, 2007.

[10] A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Blat, "3d graphics on the web: A survey," *Computers & graphics*, vol. 41, pp. 43–61, 2014.

[11] "Python documentation." [Online]. Accessed: May-04-2024. Available: https://docs.python.org/3/.

[12] "Welcome to flask — flask documentation (3.0.x)." [Online]. Accessed: May-29-2024. Available: https://flask.palletsprojects.com/en/3.0.x/.

[13] "Javascript — mdn." [Online]. Accessed: Oct.-12-2023. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[14] "Node.js." [Online]. Accessed: Oct.-04-2023. Available: https://nodejs.org/en.

[15] "Express - node.js web application framework." [Online]. Accessed: May-04-2024. Available: https://expressjs.com/.

[16] "About npm — npm docs." [Online]. Accessed: Oct.-04-2023. Available: https://docs.npmjs.com/about-npm.

[17] "Typescript is javascript with syntax for types." [Online]. Accessed: Jan.-14-2024. Available: https://www.typescriptlang.org/.

[18] "React: The library for web and native user interfaces." [Online]. Accessed: Oct.-04-2023. Available: https://react.dev/.

[19] "Built-in react hooks – react." [Online]. Accessed: Oct.-05-2023. Available: https://react.dev/reference/react.

[20] A. Schilling, J. Bolling, and C. Nagel, "Using gltf for streaming citygml 3d city models," in *Proceedings of the 21st International Conference on Web3D Technology*, pp. 109–116, 2016.

[21] "Citygml." [Online]. Accessed: Oct.-12-2023. Available: https://www.ogc.org/standard/citygml/.

[22] E. Cantatore, M. Lasorella, and F. Fatiguso, "Resilient improvement of historic districts via digital tools. the virtualization of energy retrofit actions using simplified citygml-based plans," in *International Conference on Computational Science and Its Applications*, pp. 155–172, Springer, 2021.

[23] K. Chaturvedi, Z. Yao, and T. H. Kolbe, "Web-based exploration of and interaction with large and deeply structured semantic 3d city models using html5 and webgl," in *Bridging Scales-Skalenübergreifende Nah-und Fernerkundungsmethoden, 35. Wissenschaftlich-Technische Jahrestagung der DGPF*, 2015.

[24] "gltf - runtime 3d asset delivery." [Online]. Accessed: Sep.-21-2023. Available: https://www.khronos.org/gltf/.

[25] "gltf 2.0 specification." [Online]. Accessed: Oct.-02-2023. Available: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html.

[26] G.-h. Lee, P.-h. Choi, J.-h. Nam, H.-s. Han, S.-h. Lee, and S.-c. Kwon, "A study on the performance comparison of 3d file formats on the web," *International journal of advanced smart convergence*, vol. 8, no. 1, pp. 65–74, 2019.

[27] L. Bilke, T. Fischer, C. Helbig, C. Krawczyk, T. Nagel, D. Naumov, S. Paulick, K. Rink, A. Sachse, S. Schelenz, *et al.*, "Tessin vislab—laboratory for scientific visualization," *Environmental Earth Sciences*, vol. 72, pp. 3881–3899, 2014.

[28] A. K. Turner, "Challenges and trends for geological modelling and visualisation," *Bulletin of Engineering Geology and the Environment*, vol. 65, pp. 109–127, 2006.

[29] I. Nishanbaev, "A web repository for geo-located 3d digital cultural heritage models," *Digital Applications in Archaeology and Cultural Heritage*, vol. 16, p. e00139, 2020.

[30] J. Kahkonen, L. Lehto, T. Kilpelainen, and T. Sarjakoski, "Interactive visualisation of geographical objects on the internet," *International Journal of Geographical Information Science*, vol. 13, no. 4, pp. 429–438, 1999.

[31] F. N. K. Özgün, "Web browsers as a 3d visualization environment," *Bilişim Teknolojileri Dergisi*, vol. 15, no. 3, pp. 251–259, 2022.

[32] M. Krämer and R. Gutbell, "A case study on 3d geospatial applications in the web using state-of-the-art webgl frameworks," in *Proceedings of the 20th international conference on 3d web technology*, pp. 189–197, 2015.

[33] B. Mao, Y. Ban, and B. Laumert, "Dynamic online 3d visualization framework for real-time energy simulation based on 3d tiles," *ISPRS International Journal of Geo-Information*, vol. 9, no. 3, p. 166, 2020.

[34] "3dcitydb-docs." [Online]. Accessed: Oct.-26-2023. Available: https://3dcitydb-docs.readthedocs.io/en/latest/index.html.

[35] "Fme." [Online]. Accessed: Jun.-17-2024. Available: https://fme.safe.com/.

[36] A.-M. Boutsi, C. Ioannidis, and S. Soile, "Interactive online visualization of complex 3d geometries," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 42, pp. 173–180, 2019.

[37] "Cesium: The platform for 3d geospatial." [Online]. Accessed: Sep.-24-2023. Available: https://cesium.com/.

[38] H. Dimitrov and D. Petrova-Antonova, "3d city model as a first step towards digital twin of sofia city," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 43, pp. 23–30, 2021.

[39] M. O. Mete, D. Guler, and T. Yomralıoglu, "Development of 3d web gis application with open source library," *Selçuk Üniversitesi Mühendislik, Bilim Ve Teknoloji Dergisi*, vol. 6, pp. 818–824, 2018.

[40] Z. Yao, C. Nagel, F. Kunde, G. Hudra, P. Willkomm, A. Donaubauer, T. Adolphi, and T. H. Kolbe, "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*, vol. 3, no. 1, pp. 1–26, 2018.

[41] "3d web map client feature overview." [Online]. Accessed: Mar.-02-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/webmap/features.html.

[42] C. E. Kilsedar, F. Fissore, F. Pirotti, and M. A. Brovelli, "Extraction and visualization of 3d building models in urban areas for flood simulation," *The international archives of the photogrammetry, remote sensing and spatial information sciences*, vol. 42, pp. 669–673, 2019.

[43] "3d web map client interaction with 3d objects." [Online]. Accessed: Mar.-02-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/webmap/selection.html.

[44] "3d web map client enriching kml/gltf models with thematic data." [Online]. Accessed: Mar.-02-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/webmap/online-spreadsheet.html.

[45] D. Laksono and T. Aditya, "Utilizing a game engine for interactive 3d topographic data visualization," *ISPRS International Journal of Geo-Information*, vol. 8, no. 8, p. 361, 2019.

[46] P. Würstle, R. Padsala, T. Santhanavanich, and V. Coors, "Viability testing of game engine usage for visualization of 3d geospatial data with ocg standards: 17th 3d geoinfo conference, 19-21 october 2022, sydney, australia," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 10, pp. 281–288, 2022.

[47] "Babylon.js: Powerful, beautiful, simple, and open web rendering engine." [Online]. Accessed: Sep.-25-2023. Available: https://www.babylonjs.com.

[48] O. Wysocki, "Awesome citygml." [Online]. Accessed: OCt.-20-2023. Available: https://github.com/OloOcki/awesome-citygml.

[49] "3dcitydb – the citygml database." [Online]. Accessed: OCt.-16-2023. Available: https://www.3dcitydb.org/3dcitydb/.

[50] "Berlin 3d - downloadportal." [Online]. Accessed: Oct.-29-2023. Available: https://www.businesslocationcenter.de/berlin3d-downloadportal/?lang=en/export.

[51] "3dcitydb-docs visualiztion export tiling filter." [Online]. Accessed: May-10-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/impexp/export-vis-filters/tiling.html.

[52] G. Gröger, T. H. Kolbe, C. Nagel, and K.-H. Häfele, "Ogc city geography markup language (citygml) encoding standard," 2012.

[53] "3dcitydb-docs visualiztion export." [Online]. Accessed: May-10-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/impexp/export-vis.html.

[54] “3dcitydb-docs bounding box filter.” [Online]. Accessed: May-10-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/impexp/export-vis-filters/bbox.html.

[55] “3dcitydb-docs feature type filter.” [Online]. Accessed: May-10-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/impexp/export-vis-filters/feature-type.html.

[56] Z. Yao, S. Nguyen, M. Sindram, K. Chaturvedi, and T. Kolbe, “3d city database for citygml - a hands-on tutorial for beginners.” [Online], 2016. Available: https://www.3dcitydb.org/3dcitydb/fileadmin/TUM$_{W}orkshop/Documents/Tutorial.pdf$.

[57] “3dcitydb-docs spreadsheet generator plugin.” [Online]. Accessed: May-10-2024. Available: https://3dcitydb-docs.readthedocs.io/en/latest/plugins/spreadsheet/index.html.

[58] “3dcitydb-docs exporting to kml/collada/gltf.” [Online]. Accessed: Mar.-16-2024. Available: https://3dcitydb-docs.readthedocs.io/en/release-v4.2.3/impexp/kml-collada-gltf-export.htmlpic-kml-collada-gltf-export-hierarchical-directory.

[59] “3dcitydb-docs handling kml/gltf models with online spreadsheet.” [Online]. Accessed: Mar.-16-2024. Available: https://3dcitydb-docs.readthedocs.io/en/release-v4.2.3/webmap/online-spreadsheet.html?highlight=city

[60] “Importing assets — babylon.js documentation.” [Online]. Accessed: May-23-2024. Available: https://doc.babylonjs.com/features/featuresDeepDive/importers.

[61] “Layer masks and multi-cam textures — babylon.js documentation.” [Online]. Accessed: May-24-2024. Available: https://doc.babylonjs.com/features/featuresDeepDive/cameras/layerMasksAndMultiCam.

[62] “Babylon.js documentation camera collisions.” [Online]. Accessed: Mar.-28-2024. Available: https://doc.babylonjs.com/features/featuresDeepDive/cameras/camera_collisions.

[63] "Shadows — babylon.js documentation." [Online]. Accessed: Mar.-27-2024. Available: https://doc.babylonjs.com/features/featuresDeepDive/lights/shadows.

[64] "Skyboxes, babylon.js documentation." [Online]. Accessed: Mar.-28-2024. Available: https://doc.babylonjs.com/features/featuresDeepDive/environment/skybox.

[65] S. Wöllmann, R. Zink, and M. Piser, "3d mapping to collect volunteered geographic information," in *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*, pp. 509–513, IEEE, 2020.

[66] A.-M. Boutsi, C. Ioannidis, and S. Soile, "An integrated approach to 3d web visualization of cultural heritage heterogeneous datasets," *Remote Sensing*, vol. 11, no. 21, p. 2508, 2019.