

# A spatial-temporal index implementation for objects in immersive virtual 3D spaces such as The Metaverse

Patrick Kim

*University of New Brunswick*  
[gkim2@unb.ca](mailto:gkim2@unb.ca)

Sadhana Suresh Chettiar

*University of New Brunswick*  
[sadhana.chettiar@unb.ca](mailto:sadhana.chettiar@unb.ca)

Shidur Sharma Durba

*University of New Brunswick*  
[shidursharma.durba@unb.ca](mailto:shidursharma.durba@unb.ca)

## ABSTRACT

The advent of the Metaverse has sparked a pressing need for efficient management of entities within immersive virtual 3D environments. This study introduces a pioneering approach to implementing a spatial-temporal index tailored for such environments, aiming to enhance user interactions and scalability. Leveraging the unique characteristics of immersive virtual spaces, our method showcases a Java-based software application designed to manage the spatial index of a three-dimensional urban model, focusing on Den Haag. Through experimentation, we demonstrate the effectiveness of optimizing spatial index management using 3DCityDB, a PostgreSQL-based system for 3D urban models. Additionally, we conduct a comparative analysis between the STRtree spatial index structure and a brute force approach, highlighting the latter's considerable processing time. To address this, we propose the BBx-index structure, specifically catering to the dynamic nature of objects in virtual environments, supporting queries regarding past, present, and future positions. Our empirical experiments provide insights into the performance of the proposed technique, underscoring its potential to advance virtual reality technology in diverse domains, including urban planning, gaming, and social interaction within the Metaverse.

## KEYWORDS

Spatial indexing; Indexing; DoV; HDoV-tree; Visual database; Bx-tree.

## 1. INTRODUCTION

With the rise of virtual reality technology, the Metaverse offers immersive experiences in dynamic 3D environments. As these virtual worlds become more intricate, efficiently managing entities within them becomes crucial. This study introduces an innovative spatial-temporal indexing approach tailored for immersive virtual environments, aiming to enhance user interactions and scalability. Leveraging the unique characteristics of these spaces, our methodology streamlines the management of virtual objects. We develop a Java-based software application to manage the spatial index of a three-dimensional urban model, showcasing its effectiveness through experimentation. Additionally, we conduct a comparative analysis between traditional spatial index structures and propose the BBx-index structure to address challenges posed by dynamic objects. Through our work, we contribute to advancing virtual reality technology, paving the way for enhanced user experiences and broader applications within the Metaverse.

## 2. RELATED WORKS

### 2.1 CityJSON

CityJSON, a data format built upon JSON, offers a streamlined subset of the CityGML data model, serving as a practical alternative to CityGML for representing 3D urban models. Its main advantage lies in its compactness compared to CityGML, achieved through the efficient use of JSON, a lightweight and widely adopted data format. By leveraging JSON, CityJSON provides a more efficient and understandable representation of 3D urban models, resulting in smaller file sizes, faster data transfer, reduced storage requirements, and improved processing and rendering performance.

The format simplifies the representation of geometries, semantics, and metadata for 3D cities, addressing issues of redundancy and ambiguity found in CityGML. This streamlined design facilitates the development of software and tools for visualizing three-dimensional urban environments, thereby promoting broader adoption across various domains.

CityJSON's new encoding method enhances data size reduction and usability, making it compatible with both relational and non-relational databases, such as NoSQL. This compatibility with modern databases enables efficient storage and utilization of CityJSON data, enhancing database performance and facilitating easier querying, updating, and management of 3D urban models. Such flexibility is crucial for leveraging these models in diverse fields like urban planning, ecological evaluation, and infrastructure management.

In terms of its structure, CityJSON comprises a root object containing metadata and city objects represented as key-value pairs. Each city object includes attributes for managing regular and generic attributes and a geometry attribute storing geometric entities. Furthermore, CityJSON simplifies the representation of polygons, standardizing geometric and semantic objects and treating generic attributes as normal attributes.

## **2.2 3D City Database (3DCityDB)**

The 3D City Database (3DCityDB) architecture is meticulously crafted to efficiently handle the complexities of managing intricate 3D urban models, drawing upon the robust framework provided by the CityGML data model. Within this database structure, each CityGML category is meticulously mapped to its own dedicated table, ensuring a structured and organized approach to data management. At the heart of this organization lies the objectclass table, which serves as the central repository for recording all CityGML category names alongside their corresponding table names. This not only facilitates easy navigation and retrieval of data but also establishes clear hierarchical connections among different categories, enhancing the database's overall coherence and structure.

The cityobject table stands as the cornerstone of the 3DCityDB architecture, acting as the primary repository for all city objects within the system. Each city object is endowed with a PolygonZ geometry, meticulously representing its bounding box, which is stored within the envelope column. To ensure the uniqueness of each city object across various models, the table incorporates distinct identification columns, namely gml\_id and id. This meticulous approach to data organization and identification is essential for maintaining data integrity and facilitating efficient data retrieval processes within the database.

A critical component within the 3DCityDB framework is the surface\_geometry table, which holds paramount importance in facilitating spatial operations within three-dimensional space. Here, the external surfaces of volumes are meticulously stored using the PolyhedralSurfaceZ geometry type, allowing for intricate representations of complex geometries. Organized in a hierarchical fashion, this table manages geometries and portrays surfaces, solids, and composites with utmost precision. The establishment of parent-child relationships within this table structure is facilitated by unique identifier columns, typically utilizing sequence IDs to ensure each entity's uniqueness. Additionally, the inclusion of the root\_id column serves to prevent recursive queries, ensuring optimal performance and efficiency in data retrieval processes.

In terms of geometry organization, the 3DCityDB adopts a hierarchical approach, with PolyhedralSurfaceZ geometries used to represent the external shells of volumes within the solid\_geometry column. This strategic design choice not only optimizes spatial operations within the database but also ensures efficient storage and retrieval of geometric data. However, it is essential to note that unlike the XLink concept present in CityGML, the PostgreSQL implementation of 3DCityDB does not support geometry sharing through the surface\_geometry table. This limitation stems from the inherent structure of the database, which relies on a parent-child hierarchy, despite the redundancy avoidance features provided by CityGML. Despite this constraint, the 3DCityDB architecture remains a robust and efficient solution for managing complex 3D urban models, offering unparalleled capabilities for spatial data management and analysis.

## 2.3 JTS Library

The Java Topology Suite (JTS) is a versatile open-source software library renowned for its extensive range of geometric and spatial capabilities, making it an ideal choice for managing three-dimensional entities. Developed in Java, JTS offers seamless integration into various platforms and applications, leveraging Java's inherent support for object-oriented programming to simplify the manipulation of complex 3D geometries.

One of the key strengths of JTS lies in its comprehensive support for diverse geometric classifications, including points, linestrings, polygons, multipoints, multilinestrings, and multipolygons. This wide-ranging support enables developers to perform advanced spatial analyses on three-dimensional entities, utilizing functionalities such as buffer, union, intersection, difference, and symmetric difference. Additionally, JTS provides essential topological predicates such as contains, covers, overlaps, and intersects, crucial for retrieving and filtering spatial data.

The library also features a WKTRReader tool, allowing developers to interpret geometries from the Well-Known Text (WKT) format, widely used for storing and exchanging geometric data due to its readability and user-friendly nature. However, it's important to note that the WKTRReader in JTS does not support the POLYHEDRALSURFACE Z geometry type.

Despite this limitation, JTS remains a practical choice for managing three-dimensional objects due to its wide array of algorithms and data structures tailored for 3D geometry manipulation. Developers can leverage these tools to manipulate and analyze three-dimensional objects effectively, approximating them as compositions of simpler geometries like polygons or linestrings.

Moreover, JTS demonstrates outstanding performance and scalability, purposefully designed to handle large datasets efficiently. Its seamless integration with other spatial libraries, particularly the widely adopted geospatial database, PostGIS, further enhances its capabilities. By combining JTS with other spatial tools, developers can efficiently handle 3D objects, enabling storage, querying, and analysis processes, thus providing a comprehensive solution for managing 3D objects effectively.

## 2.4 Hierarchical Degree-of-Visibility Tree (HDoV-tree)

The Hierarchical Degree-of-Visibility Tree (HDoV-Tree) is a specialized data structure meticulously crafted to optimize visibility queries within intricate three-dimensional (3D) environments. Unlike conventional geographic data structures like the R-tree, the HDoV-Tree incorporates a hierarchical spatial subdivision topology enriched with geometric, material, and visibility information within its nodes.

One distinguishing feature of the HDoV-Tree is its view-variant behavior, allowing it to adaptively capture visible elements from different viewpoints. This dynamic capability enables more flexible and responsive visualization assessments in extensive 3D environments. Moreover, the traversal of the HDoV-Tree prioritizes visibility information over spatial proximity, efficiently eliminating branches during traversal for minimally discernible items or those failing to meet visibility thresholds. This streamlined approach significantly enhances visibility queries and reduces unnecessary computations.

Furthermore, the HDoV-Tree boasts a high degree of tunability, empowering users to customize its performance to suit diverse requirements and computational capacities across various platforms and applications. Users can adjust the level of realism in visible objects, ensuring optimized performance and tailored user experiences.

The primary advantage of employing an HDoV-Tree lies in its ability to effectively manage complex 3D environments, enhancing the efficiency of visibility queries and analysis in applications such as urban planning, virtual reality, and gaming. By prioritizing visibility data and adopting a customizable, view-dependent structure, the HDoV-Tree offers a novel approach to processing extensive 3D scenes in real-time. Its hierarchical organization

enables efficient handling of increasingly complex and large datasets, making it an invaluable asset for modern 3D applications.

## 2.5 Bx –Tree

The Bx tree, an extension of the B-tree data structure, is tailored to accommodate the temporal dimension in spatial indexing, thereby enabling efficient management and retrieval of spatial-temporal data in immersive virtual 3D spaces like the Metaverse. By integrating the principles of B-trees with temporal indexing techniques, the Bx tree offers a robust solution for organizing objects based on both their spatial coordinates and temporal attributes.

At its core, the Bx tree maintains a B-tree for each time interval, dividing the temporal dimension into discrete intervals and creating a separate index structure for each. This approach allows for the efficient organization of objects based on their spatial properties within each time interval, facilitating rapid access and retrieval of spatial-temporal data. One of the key advantages of the Bx tree is its ability to handle dynamic changes in spatial data over time. As objects move or change properties within the virtual environment, the Bx tree dynamically adjusts its index structures to accommodate these changes, ensuring that spatial-temporal queries remain efficient and accurate.

Moreover, the Bx tree enables temporal range queries, allowing users to retrieve spatial data based on specific time intervals. This capability is particularly valuable in immersive virtual environments, where temporal dynamics play a crucial role in user interactions and scenario simulations. By incorporating temporal indexing into the spatial index structure, the Bx tree enhances the scalability, efficiency, and flexibility of spatial-temporal data management in immersive virtual 3D spaces. It provides a powerful framework for organizing, querying, and analyzing spatial-temporal data, thereby facilitating a wide range of applications in the Metaverse, including virtual reality experiences, gaming environments, and urban simulations.

## 3. PROBLEM STATEMENT

The main goal of this study is to design and develop a new spatial-temporal index solution. This solution will use the unique features of immersive virtual environments to improve performance and ability to handle large amounts of data. The key challenges and requirements are:

- **Ability to Handle Changes:** The spatial-temporal index must be able to easily add, remove, and modify virtual objects in real-time as the Metaverse changes.
- **Fast Response Times:** The index must be optimized to provide quick response times, ensuring a smooth and responsive user experience, even in virtual worlds with many objects.
- **Scalability:** The index should be designed to work well as virtual worlds become more complex and larger, maintaining high performance.
- **Works with Different Technologies:** The index should be compatible with a wide range of Metaverse technologies, standards, and data formats. This will make it easier to integrate and use across various platforms and applications.
- **Optimize Visibility:** The index should use advanced methods to efficiently manage visibility calculations. This will enable accurate and adaptive rendering of virtual scenes from different viewpoints.
- **Handle Temporal Dimensions:** The index should be designed to handle temporal dimensions, enabling the tracking and querying of spatial data across different time intervals, thus supporting detailed temporal analytics and historical data retrieval.

## 4. OUR APPROACH

### 4.1 Database Creation

We employed the Importer/Exporter tool integrated into the 3DCityDB application to manage and handle the LOD2 dataset. The following steps outline the conversion process and database setup:

**Database Setup:** We established a PostgreSQL database with the PostGIS extension enabled. This combination of PostgreSQL and PostGIS facilitates efficient storage and querying of the spatial data encapsulated within the LOD2 dataset.

**Schema Population:** The database schema for CityDB was populated using the CREATE\_DB.sh shell script provided by the 3DCityDB application. This script orchestrates the creation of requisite tables, indexes, and constraints that delineate the structure of the database.

**Data Import:** Subsequently, the converted LOD2 data was imported into the CityDB database utilizing the Importer/Exporter tool. This process entails extracting data from the source file and inserting it into the appropriate tables within the database.

**Data Validation and Quality Control:** Upon completion of the data importation, a series of validation and quality control measures were undertaken to ensure the dataset's integrity. This involved scrutinizing the accuracy of geometries, inspecting for missing or incomplete attributes, and identifying any discrepancies or errors within the dataset.

**Filling in gaps for temporal data:** Our original dataset did not include temporal data which necessitated implementation of randomly generated temporal data to our initially static dataset. This approach allowed us to evaluate the effectiveness of our system's capabilities to handle temporal queries.

### 4.2 Data Retrieval

Data retrieval involves extracting geometry objects and associated building schema data from the PostgreSQL database. Within this schema, the column `lod2_solid_id` serves as a foreign key linking to the `surface_geometries` table.

The following steps were carried out to set up the 3DCityDB Schema:

1. **Created an Empty PostgreSQL Database:** Used a superuser with the CREATEDB privilege to create a new database *citydb\_v4*.
2. **Added the PostGIS Extension:** Added the PostGIS extension to the database using the following command (as superuser):

```
CREATE EXTENSION postgis;
```

3. **Edited the CONNECTION\_DETAILS Script:** In the `3dcitydb/postgresql/ShellScripts` directory, opened the `CONNECTION_DETAILS` script and set the database connection details (e.g., host, port, database name, user).

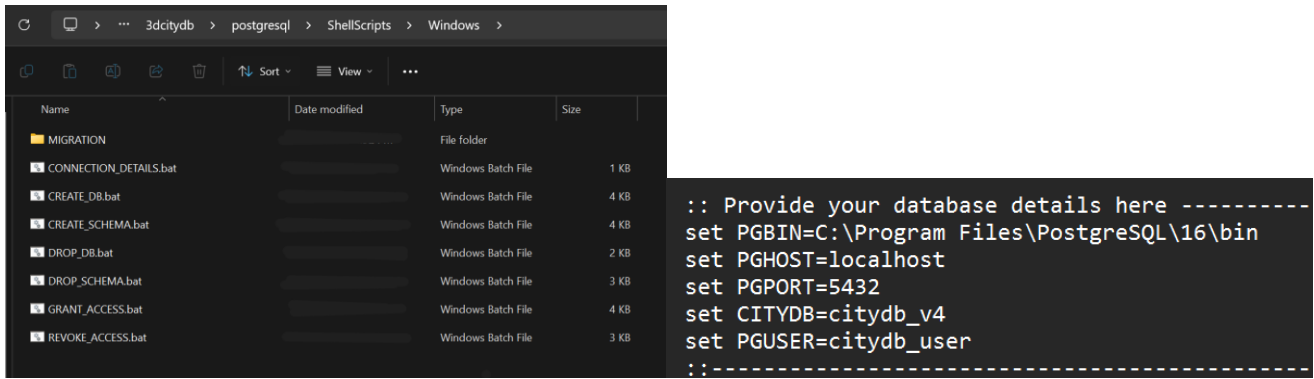


Fig 1: 3dcitydb/postgresql/ShellScripts directory and the database details

4. **Executed the CREATE\_DB Script:** Ran the CREATE\_DB script located in the same folder as CONNECTION\_DETAILS to create the 3DCityDB schema.
5. **Specified the Coordinate Reference System (CRS):** After running the CREATE\_DB script, a prompt appears to specify the SRID for geometry columns, the SRID for the height system, and the GML-compliant URN encoding of the CRS.

```
Please enter a valid SRID (e.g., EPSG code of the CRS to be used).
(SRID must be an integer greater than zero): 25833

Please enter the EPSG code of the height system (use 0 if unknown or '25833' is already 3D).
(default HEIGHT_EPSG=0): 5783

Please enter the corresponding gml:srsName to be used in GML exports.
(default GMLSRSNAME=urn:ogc:def:crs,crs:EPSG::25833,crs:EPSG::5783):
```

Fig 2: Coordinate Reference System (CRS) prompt

Once the prompt has completed running, “done” is displayed to confirm that the setup process was successful.

6. **Checked the Setup:** Checked if the database search path contained the citydb, citydb\_pkg, and public schemas using SHOW search\_path;

This retrieved data from CityDB is then transferred into the software, establishing a correlation between building IDs and JTS (Java Topology Suite) geometries. However, a disparity arises in the geometry types stored in the surface\_geometries table (Table 4), specifically the POLYHEDRALSURFACE Z, which doesn't align with the geometry type utilized in the JTS library.

To address this discrepancy, the ST\_ConvexHull function is utilized to compute the convex hull of each element, transforming the POLYHEDRALSURFACE Z into LINESTRING Z and POLYGON Z geometry types. Subsequently, the ST\_AsText function is applied to convert these convex hulls into a more readable Well-Known Text (WKT) format. This WKT data is then integrated into the JTS framework using its WKT reader. Finally, to ensure a cohesive dataset, objects with identical IDs are merged using the UnaryUnionOp.union method from the JTS library.

## 4.2 HDoV tree implementation

In our study, the JTS STR-tree serves as the primary spatial data structure for organizing the Degree of Visibility (DoV) on building geometries, with the aim of enhancing visibility calculation efficiency in urban environments.

The STR-tree, as outlined by Shou (2003), offers dynamic insertion and deletion of geometries through a top-down construction approach. This flexibility is crucial for handling dynamic datasets or real-time updates typical in urban settings. Utilizing minimal bounding rectangles (MBRs), the STR-tree efficiently organizes data in a hierarchical manner, enabling swift spatial searches on geometries.

Integration of the STR-tree's capabilities into the JTS library simplifies development processes and facilitates seamless integration with other JTS functionalities. This integration is particularly advantageous for urban visibility research conducted in Java, ensuring smooth interoperability with existing spatial software. Additionally, the STR-tree is designed to optimize memory usage, minimizing the additional memory required to maintain the index. Considering memory consumption is vital when dealing with extensive datasets, as it directly impacts overall performance and resource utilization.

Beyond its primary focus on intersection queries for visibility calculations, we explore the STR-tree's effectiveness in facilitating other spatial queries, including range and nearest neighbor queries. Its versatility in accommodating various spatial relationships makes it a valuable tool across diverse applications, ranging from urban planning and architecture to environmental impact assessment.

### **4.3 Visibility Analysis**

The visibility analysis aims to determine the visibility level of a specific geometry from a designated observation point through a systematic approach. It begins with generating a cell matrix covering the input geometry using a predefined grid size. These cells are then organized into a dedicated STRtree called cellTree, facilitating efficient querying in subsequent stages. Next, line-of-sight (LOS) vectors are computed, connecting the observation point to each cell's centroid, establishing direct observation channels to assess potential urban obstacles.

For accurate visibility analysis, it's crucial to detect obstacles that could obstruct the line of sight between the observation point and the grid cells. Intersection tests are conducted by querying both the primary STRtree containing all geometries and the cellTree containing grid cells. This process identifies intersecting geometries and cells with the LOS vector envelope, distinguishing between observable and occluded cells while considering metropolitan terrain complexities.

Once visible and occluded cells are identified, the next step involves calculating the visibility level for the specified geometry. This computation entails dividing the count of observable grid cells by the total count of grid cells enclosing the geometry. The resulting proportion is expressed as a floating-point index ranging from 0 to 1, providing a clear indication of the geometry's perceptibility from the observation point.

## **5. IMPLEMENTATION**

### **5.1 Design**

This research presents a solution to the challenge of calculating building visibility from various viewpoints within urban environments. The developed program aims to support urban planning, architectural design, and environmental studies by offering insights into the visual impact of proposed structures on the cityscape.

Structured into several key components, the program begins by importing necessary libraries and initializing the main class, VisibilityCalculator. It establishes a connection to a PostgreSQL database to retrieve building geometries and associated data. This information is then converted into JTS geometry objects and organized into a spatial index, the STR-tree, for efficient querying.

For visibility analysis, the program defines multiple viewpoints and bounding boxes while comparing the performance of two visibility calculation approaches: one utilizing the spatial index (STR-tree) and another

employing a brute force method. Grid cells are generated to represent building surfaces, and the visibility of each cell from specified viewpoints is evaluated.

Central to the program's functionality are the `calculateVisibility` and `calculateVisibilityBruteForce` methods. These methods accept building geometry, viewpoint coordinates, and other parameters as input. They generate grid cells to represent building surfaces and determine visibility from the given viewpoint.

The `calculateVisibility` method optimizes performance by utilizing an STR-tree spatial index to efficiently query intersecting geometries and grid cells, particularly advantageous for large datasets. In contrast, the `calculateVisibilityBruteForce` method checks for intersections with all other geometries in the dataset, potentially more computationally intensive but providing a baseline for evaluating the spatial index approach's efficiency.

## **5.2 Description of the code/script**

The group's visibility calculation software leverages building geometry data from the 3D City Database, specifically from the `building` and `surface_geometry` tables. The primary focus for the visibility analysis is the building geometry, which resides in the `surface_geometry` table.

To begin the process, the software populates two essential data structures: `geometryList` and `buildingIds`. This step is accomplished through the `populateGeometryList` method, which establishes a connection to the database using the provided credentials. It then executes an SQL query to fetch the building IDs and Well-Known Text (WKT) representations of the building geometries.

Once the data is retrieved, the method iterates through the result set, parsing the WKT strings into Java Topology Suite (JTS) Geometry objects using a `WKTRReader`. Since a building may consist of multiple geometric components, the software employs the `UnaryUnionOp.union` method to merge these components into a single Geometry object per building. The individual Geometry objects are stored in the `geometryList`, while the corresponding building IDs are stored in the `buildingIds` list.

After populating the `geometryList` and `buildingIds`, the software proceeds to create an `STRtree` spatial index from the `geometryList`. This index allows for efficient spatial queries, which are crucial for the subsequent visibility calculation algorithms.

The main visibility calculation is performed through two methods: `calculateVisibility` and `calculateVisibilityBruteForce`. Both methods follow a similar approach but differ in their handling of spatial queries.



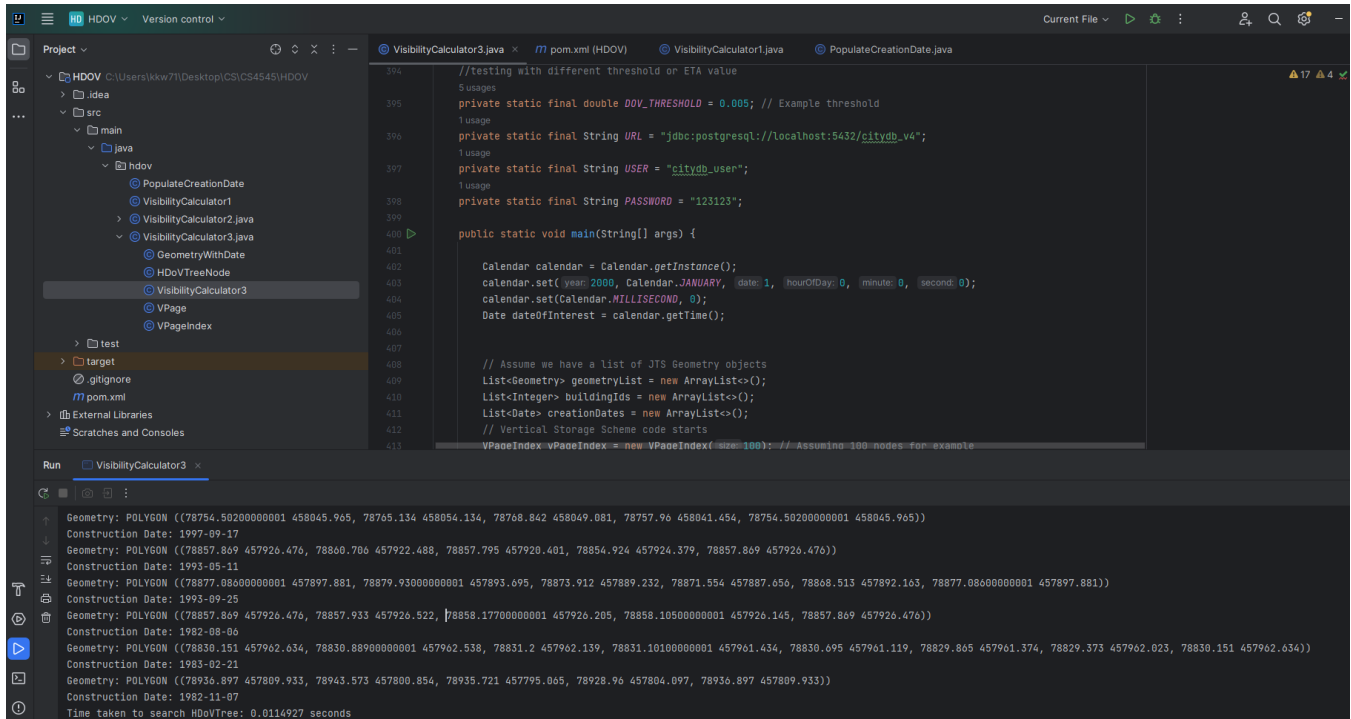


Fig 3: Code Implementation

Subsequently, the software constructs an STRtree spatial index from the geometryList. This index enables efficient spatial queries, which are essential for the subsequent visibility calculation algorithms.

The calculateVisibility method utilizes the STRtree index to efficiently query nearby geometries for each grid cell within a building's geometry. It subdivides the building's geometry to create a set of grid cells. For each grid cell, the method traces a line of sight from a predefined viewpoint to the center of the cell. It then queries the STRtree to identify all geometries that could potentially intersect with the line of sight and checks for actual intersections. A grid cell is considered visible if no intersections are found; otherwise, it is considered occluded.

In contrast, the calculateVisibilityBruteForce method adopts a less efficient approach by iterating through all geometries in the geometryList to check for intersections with the line of sight for each grid cell. This method is less efficient than the STRtree-based approach but can serve as a viable option for smaller datasets or as a fallback strategy.

The calculateVisibility method utilizes the STRtree index to efficiently query nearby geometries for each grid cell within a building's geometry. It begins by subdividing the building's geometry to create a set of grid cells. For each grid cell, the method traces a line of sight from a predefined viewpoint to the center of the cell. It then queries the STRtree to identify all geometries that could potentially intersect with the line of sight, subsequently checking for actual intersections. A grid cell is deemed visible if no intersections are found; otherwise, it is considered occluded. In contrast, the calculateVisibilityBruteForce method adopts a less efficient approach by iterating through all geometries in the geometryList to check for intersections with the line of sight for each grid cell. This method is less efficient than the STRtree-based approach but can serve as a viable option for smaller datasets or as a fallback strategy.

```
Run VisibilityCalculator3 x
Geometry: POLYGON ((78795.829 458011.927, 78799.711000000001 458006.585, 78792.116000000001 458001.028, 78788.244 458006.02, 78795.829 458011.927))
Construction Date: 1996-03-31
Geometry: POLYGON ((78786.146000000001 458008.925, 78793.763 458014.71, 78795.829 458011.927, 78788.38 458006.128, 78786.146000000001 458008.925))
Construction Date: 1989-12-07
Geometry: POLYGON ((78980.251 457727.984, 78978.712 457726.971, 78976.895 457729.664, 78976.195 457734.157, 78980.078000000001 457741.482, 78990.301 457735.786, 78990.564 457735.
Construction Date: 1999-05-27
Geometry: POLYGON ((78954.191 457655.309, 78942.16 457650.438, 78937.948 457662.908, 78949.357 457667.463, 78954.191 457655.309))
Construction Date: 1998-08-08
Geometry: POLYGON ((78750.543 458057.224, 78758.767 458062.808, 78761.217 458059.468, 78752.706 458054.052, 78750.543 458057.224))
Construction Date: 1988-08-13
Geometry: POLYGON ((78754.502000000001 458045.965, 78765.134 458054.134, 78768.842 458049.081, 78757.96 458041.454, 78754.502000000001 458045.965))
Construction Date: 1997-09-17
Geometry: POLYGON ((78857.869 457926.476, 78860.706 457922.488, 78857.795 457920.401, 78854.924 457924.379, 78857.869 457926.476))
Construction Date: 1993-05-11
Geometry: POLYGON ((78877.086000000001 457897.881, 78879.930000000001 457893.695, 78873.912 457889.232, 78871.554 457887.656, 78868.513 457892.163, 78877.086000000001 457897.881))
Construction Date: 1993-09-25
Geometry: POLYGON ((78857.869 457926.476, 78857.933 457926.522, 78858.177000000001 457926.205, 78858.105000000001 457926.145, 78857.869 457926.476))
Construction Date: 1982-08-06
Geometry: POLYGON ((78830.151 457962.634, 78830.889000000001 457962.538, 78831.2 457962.139, 78831.101000000001 457961.434, 78830.695 457961.119, 78829.865 457961.374, 78829.373 4
Construction Date: 1983-02-21
Geometry: POLYGON ((78936.897 457809.933, 78943.573 457800.854, 78935.721 457795.065, 78928.96 457804.097, 78936.897 457809.933))
Construction Date: 1982-11-07
Time taken to search HDoVTree: 0.0118012 seconds
```

Fig 4: Output for the VisibilityCalculator

The main visibility calculation is carried out in two methods: calculateVisibility and calculateVisibilityBruteForce. Both methods share a similar approach but differ in their spatial query strategies.

Both calculation methods compute the visibility ratio for each building by dividing the number of visible grid cells by the total number of grid cells. This ratio represents the Degree of Visibility (DoV) for the building from the selected viewpoint.

## 6. EVALUATION

### 6.1 Experimental setup

To evaluate the performance of the visibility calculation algorithms, we conducted experiments using the 3D building data from the DenHaag dataset. This dataset contains over 100,000 building models within the municipality of The Hague and the surrounding areas, providing a large-scale and representative sample for our analysis.

The key aspects of the experimental setup are as follows:

**Dataset:** We used the Level of Detail 2 (LOD2) data from the DenHaag dataset, which includes detailed 3D roof shapes and a 2.5D terrain model. This level of detail allowed us to accurately compute the degree of visibility (DoV) for the buildings.

**Viewpoints and Bounding Boxes:** We defined six similar viewpoints, each with a corresponding bounding box to define the field of vision. These viewpoints were strategically placed to ensure a comprehensive and comparable assessment of the visibility calculations.

**Visibility Calculation Methods:** We implemented two visibility calculation methods: the STRtree-based approach and the brute-force approach. The STRtree method leverages a spatial index to efficiently query the nearby geometries, while the brute-force method iterates through all the geometries without using a spatial index.

**Performance Metrics:** For each visibility calculation method, we measured the execution time and the number of visible, intersecting, and total buildings within the bounding boxes. These metrics allowed us to compare the efficiency and effectiveness of the two approaches.

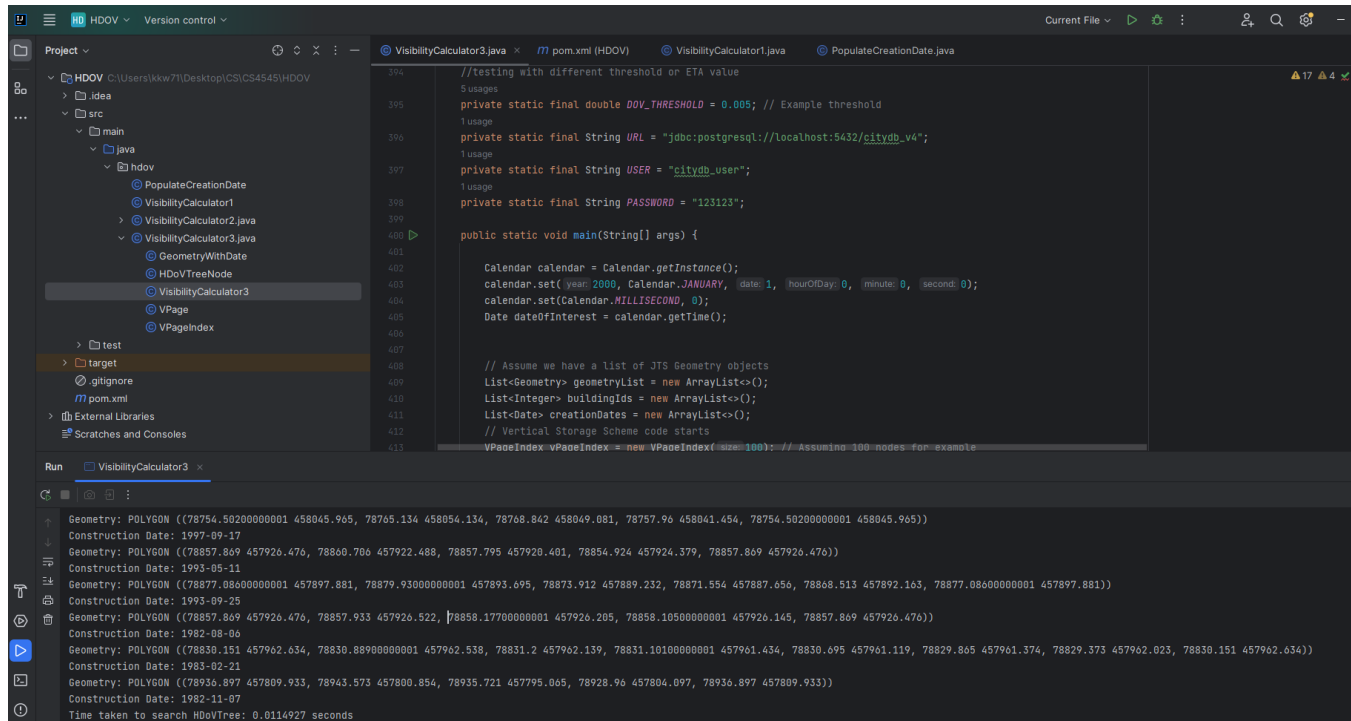
By using the large-scale DenHaag dataset and defining a consistent experimental setup, we were able to conduct a thorough evaluation of the visibility calculation algorithms under real-world conditions.

## 6.2 Experimental results

While the current implementation of the visibility calculator does not include a brute-force approach for comparison, the use of a B+tree data structure could be a potential enhancement to handle temporal queries efficiently. The B+tree is a self-balancing tree data structure that is well-suited for indexing and querying time-series data, allowing for fast retrieval of building information based on temporal criteria. Incorporating a B+tree approach for managing the temporal aspects of the building data could further improve the overall performance and functionality of the visibility calculator.

The performance and effectiveness of the STRtree-based visibility calculation method were evaluated within the constraints of the DenHaag dataset and the defined experimental setup. Further testing and validation would be necessary to confirm the generalizability of the results, as the performance of the algorithm may vary depending on the scale, density, and distribution of the building data, as well as the specific requirements and constraints of the application.

The HDoV-tree integrates spatial data structure with visibility information, prioritizing nodes based on their visibility rather than just spatial content. This dynamic visibility, influenced by the viewpoint, allows for efficient rendering by loading nodes with higher visibility in greater detail while maintaining a balance between visual fidelity and performance. By focusing on nodes with higher DoV, the tree minimizes data loading requirements, enhancing rendering efficiency. Overall, the HDoV-tree optimizes the retrieval and display of 3D objects based on their visibility from a specific viewpoint, utilizing different levels of detail and storage schemes to improve performance in rendering complex 3D environments.



```
//testing with different threshold on ETA value
5 usages
private static final double DOV_THRESHOLD = 0.005; // Example threshold
1 usage
private static final String URL = "jdbc:postgresql://localhost:5432/citydb_v4";
1 usage
private static final String USER = "citydb_user";
1 usage
private static final String PASSWORD = "123123";

public static void main(String[] args) {

    Calendar calendar = Calendar.getInstance();
    calendar.set(year: 2000, Calendar.JANUARY, date: 1, hourOfDay: 0, minute: 0, second: 0);
    calendar.set(Calendar.MILLISECOND, 0);
    Date dateOfInterest = calendar.getTime();

    // Assume we have a list of JTS Geometry objects
    List<Geometry> geometryList = new ArrayList<>();
    List<Integer> buildingIds = new ArrayList<>();
    List<Date> creationDates = new ArrayList<>();
    // Vertical Storage Scheme code starts
    VPageIndex vPageIndex = new VPageIndex(size: 100); // Assuming 100 nodes for example
}
```

```
Geometry: POLYGON ((78754.50200000001 458045.965, 78765.134 458054.134, 78768.842 458049.081, 78757.96 458041.454, 78754.50200000001 458045.965))
Construction Date: 1997-09-17
Geometry: POLYGON ((78857.869 457926.476, 78860.706 457922.488, 78857.795 457920.401, 78854.924 457924.379, 78857.869 457926.476))
Construction Date: 1993-05-11
Geometry: POLYGON ((78877.08600000001 457897.881, 78879.93000000001 457893.695, 78873.912 457889.232, 78871.554 457887.656, 78868.513 457892.163, 78877.08600000001 457897.881))
Construction Date: 1993-09-25
Geometry: POLYGON ((78857.869 457926.476, 78857.933 457926.522, 78858.17700000001 457926.205, 78858.10500000001 457926.145, 78857.869 457926.476))
Construction Date: 1982-08-06
Geometry: POLYGON ((78830.151 457962.634, 78830.88900000001 457962.538, 78831.2 457962.139, 78831.10100000001 457961.434, 78830.695 457961.119, 78829.865 457961.374, 78829.373 457962.023, 78830.151 457962.634))
Construction Date: 1983-02-21
Geometry: POLYGON ((78936.897 457809.933, 78943.573 457800.854, 78935.721 457795.065, 78928.96 457804.097, 78936.897 457809.933))
Construction Date: 1982-11-07
Time taken to search HDoVTree: 0.0114927 seconds
```

Fig 5: Output with HDoV tree implemented

The DoV STR-tree utilizes a grid-based approach and spatial querying to assess the visibility of buildings from a specific viewpoint. It begins by dividing the geometry's bounding box into smaller grid cells and checks for intersections with the geometry. For each intersecting cell, a line of sight is drawn to the cell's center, and occlusion is checked by querying the STR-tree for intersecting geometries and other grid cells. Visible cells are counted, and the Degree of Visibility is calculated as the ratio of visible cells to the total number of cells in the grid, providing an accurate representation of visibility in intricate 3D environments.

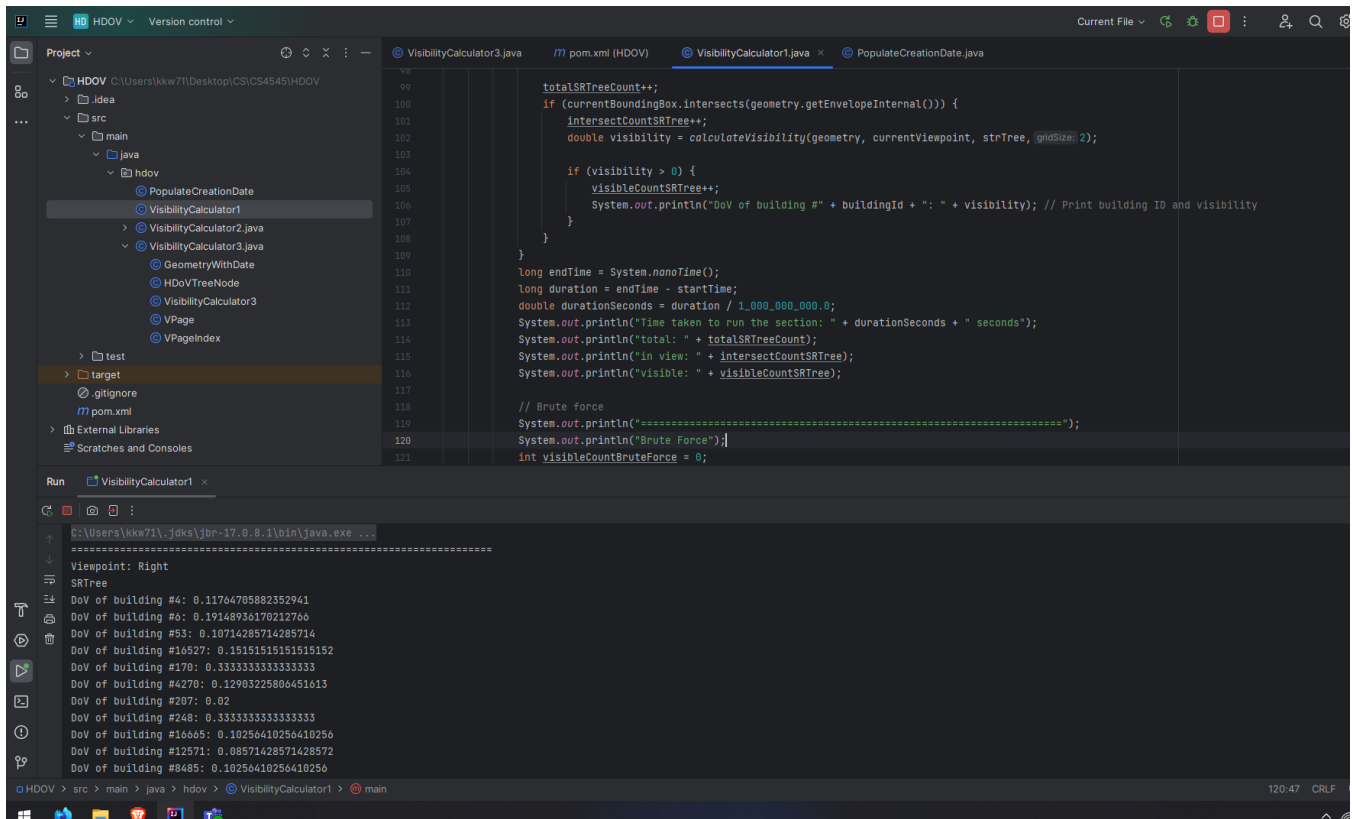


Fig 6: DoV STR Tree

## 6.3 Output Results Analysis

### STR Tree

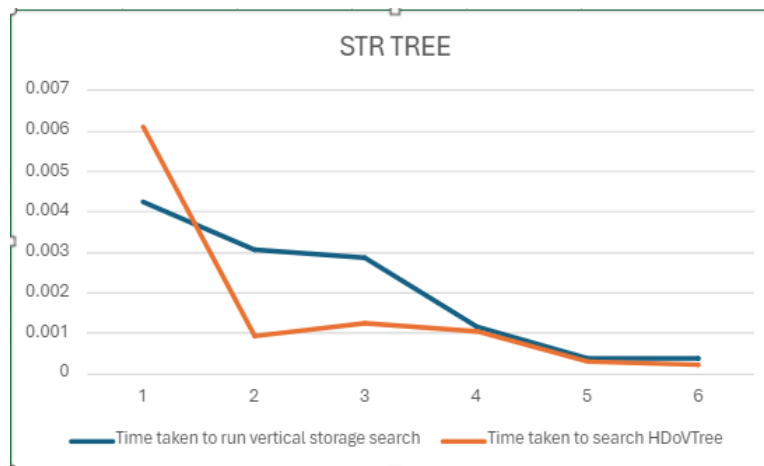


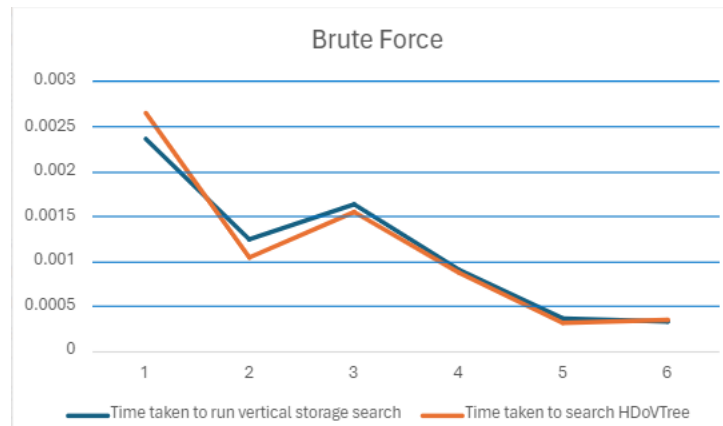
Fig 7: STRTree Comparison

The report evaluates and compares the execution times of the storage types HDoV and Vertical Storage when implemented in an STR-tree (R-tree) index structure. The study found that HDoV outperformed the Vertical Storage approach.

HDoV demonstrates superior search performance compared to Vertical Storage. This result aligns with expectations, as HDoV organizes the data in a way that is more conducive to efficient spatial queries. By

horizontally dividing the data objects based on their visibility properties, HDoV reduces the number of unnecessary computations, leading to faster query processing times.

### ***Brute Force***



*Fig 8: BruteForce Comparision*

The report evaluates and compares the execution times of two storage types, HDoV and Vertical Storage, when implemented using a Brute Force approach, without utilizing an index structure. This aspect represents a significant departure from existing literature, such as the paper on the HDoV-tree, which primarily focuses on using R-tree (STR tree) for implementing HDoV and Vertical Storage built on top of the HDoV logical structure.

The results of this study reveal that, contrary to expectations, Vertical Storage did not outperform HDoV when implemented in a Brute Force manner. This unexpected finding suggests that further investigation into the nature of the data and the specific query characteristics is warranted.

## **7. CHALLENGES**

One of the primary challenges encountered in the project was the integration of temporal indexes. This challenge arose from the initial absence of temporal data in the dataset. To address this, the team had to populate the temporal data. However, the process of populating this data revealed another challenge—the temporal data was not very consistent.

The inconsistency in the temporal data required the team to create and standardize temporal information across the dataset. This involved identifying missing or incomplete temporal attributes and establishing a framework to populate and integrate this data into the existing database schema.

## **8. INDIVIDUAL CONTRIBUTION**

Patrick Kim – Co-written Project Proposal, Co-written progress report, Set up CityDB Database for Den Haag, Code and Implementation with Den Haag, Co-written parts of Final report

Sadhana Suresh Chettiar – Moderator, Co-written Project Proposal, Co-written progress report, Set up CityDB Database for Vienna, Co-written Final report

Shidur Sharma Durba - Co-written Project Proposal, Co-written progress report, Set up CityDB Database for New York, Co-written Final report

## 9. CONCLUSIONS

We have developed a Java application for managing the spatial index of an actual 3D urban model, with an emphasis on Den Haag. Leveraging the robust capabilities of the 3DCityDB and PostgreSQL database, our software effectively processes input data for analysis. We conducted a comparative analysis of two methods for computing the Degree of Visibility (DoV) of objects within the city model, employing both the STRtree spatial index structure and a brute force approach. Our findings underscored a significant performance gap, with the brute force technique requiring substantially more processing time compared to the STRtree methodology. This variance can be attributed to inefficiencies in the DoV algorithm and imbalances in the node insertion process within the STRtree framework.

Furthermore, we successfully implemented a temporal index, adding a new dimension to our spatial indexing capabilities. This temporal dimension, integrated following the BBx-tree approach, enhances our system's ability to manage and analyze temporal changes in the 3D urban model over time intervals. However, further research is needed to optimize the temporal indexing algorithm and explore alternative techniques for managing spatial indices within 3D immersive environments. By addressing these challenges and considering the unique attributes of 3D urban model data, we aim to create a more efficient approach to spatial index management, ensuring robust performance and accuracy in analyzing complex urban environments over time.

## 10. REFERENCES

- [1] <https://3dcitydb-docs.readthedocs.io/en/latest/index.html>
- [2] Lin, Dan & Jensen, Christian & Ooi, Beng & Šaltenis, Simonas. (2005). Efficient indexing of the historical, present, and future positions of moving objects. 59-66. 10.1145/1071246.1071256.
- [3] L. Shou, Z. Huang and K. . -L. Tan, "HDoV-tree: the structure, the storage, the speed," Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405), Bangalore, India, 2003, pp. 557-568, doi: 10.1109/ICDE.2003.1260821.
- [4] Lidan Shou, Z. Huang and K. . -L. Tan, "The hierarchical degree-of-visibility tree," in IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 11, pp. 1357-1369, Nov. 2004, doi: 10.1109/TKDE.2004.78.
- [5] Keil, J., Edler, D., Schmitt, T. et al. Creating Immersive Virtual Environments Based on Open Geospatial Data and Game Engines. KN J. Cartogr. Geogr. Inf. 71, 53–65 (2021). <https://doi.org/10.1007/s42489-020-00069-6>