Data Systems for 3D Geospatial Analytics in Collaborative Extended Reality (XR)

by

Vidit Sharma

Master of Computer Science, Guru Gobind Singh Indraprastha University, 2010

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

Master of Business Administration

in the Graduate Academic Unit of Management

Supervisor: Suprio Ray, PhD, Computer Science

Examining Board: Sampath Bemgal, PhD, Management

Dinesh Gajurel, PhD, Management, Chair

This report is accepted by the Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

March 2024

ABSTRACT

The rise in 3D spatial data necessitates efficient management. Extended Reality (XR) merges physical and digital realms, promising immersive visualization and revolutionizing geospatial systems. This project investigates scalable data systems for collaborative 3D spatial XR applications crucial for disaster readiness. It optimizes immersive virtual environments and object organization within the Metaverse using Java-based software for spatial indices. Scrutinizing Degree of Visibility (DoV) computation methods enhances urban planning, gaming, and social interaction in VR. The HDoV-tree structure improves visual realism and performance, outperforming R-tree-based systems. Additionally, Semantic Textual Similarity using BERT enhances spatial data analysis, integrating advanced NLP techniques. Integrating 3D City Database (3DCityDB) facilitates efficient storage, analysis, and visualization of urban environments. In conclusion, this project amalgamates spatial data management advancements, immersive visualization, and optimized data structures, propelling Spatial Extended Reality for enhanced collaborative experiences across domains.

KEYWORDS

spatial indexing; R-tree; HDoV-tree; DoV; metaverse; visual database; tree data structure

DEDICATION

This work is dedicated to:

- My loving parents, maternal uncle, aunts, and grandmother, and my elder sibling and sister-in-law, whose unwavering support and sacrifices have been instrumental in my educational journey.
- Dr. Suprio Ray, Professor of Computer Science at UNB, for inspiring and supporting my research endeavors.
- Ronal Kori, my childhood schoolmate, for introducing me to the realm of research and bridging the knowledge from his medical expertise to my field.
- Mrs. Raj Kaul, my middle school science teacher, for nurturing my early academic curiosity.
- Dr. Tripti Mishra, Dr. Malati, and my undergraduate classmate Jonathan Pandiaraj, who were pivotal in laying the foundation for my graduate education in Computer Science.
- Ankit Gupta and Gaurav Sharma, my classmates from the Graduate School of Computer Science, for their camaraderie and intellectual contributions.
- V.P Singh, Himansh Sharma, and Pavan Patil, my batchmates and residence mates, for their companionship and shared experiences.
- Amit Khurana and Tayyab Khan, my teachers/mentors, and Aditya Chaudhary, Chahat Bahl, Akshay Chamoli (AK6), who were crucial during my competitive exams and integral to my management education.

- Lastly, my mentor, Abhinav Garg, creator of SSVK, for his guidance and support, especially during recent challenging times in my life.

ACKNOWLEDGEMENTS

I express my profound gratitude to my supervisor, Dr. Suprio Ray, for his invaluable encouragement, guidance, and support, which were fundamental in developing my understanding of the subject throughout the initial and final stages of this project.

I extend my sincere thanks to Minh Duc Nguyen, pursuing M.S. in Computer Science at UNB, whose research on the Domain of Degree of Visibility (DoV) and the integration of 3D city databases has laid the foundation and backbone of this project.

Special thanks go to Sudip Chatterjee and Suvam Das, Ph.D. candidates in Computer Science department at UNB, for their assistance throughout the course, particularly in resolving issues related to coding environment settings. MBA programs officer Angeline Ng for all her support and help. Dr Devashis Mitra and Dr Jeff McNally for allowing and supporting me to take this research. I am also grateful to Suraj Pahadi, MBA from UNB(2023), for his backup support in finance and allied courses.

I would like to express my heartfelt appreciation to my family for their unwavering support. Special mention to my parents for their insightful comments and constant motivation towards the completion of my work. To Abhinav Garg, associated with SSVK, I am deeply thankful for all the support and encouraging words.

Table of Contents

List of Tables

List of Figures

ABBREVIATIONS

HDoV: Hierarchical Degree-of-Visibility

JSON: JavaScript Object Notations

SQL: Structured Query Language

PostgreSQL: Postgres Structured Query Language

GML: Geography Markup Language

VM: Virtual Machine

DB: DataBase

Xlink: eXtensible Linking Language

SRDBMS: Spatial Relational Database Management Systems

PL/SQL: Procedural Language/Structured Query Language (for ORACLE databases)

PL/pgSQL: Procedural Language/PostgreSQL Structured Query Language (for PostgreSQL databases)

GIS: Geographic Information System

ETL: Extract, Transform, Load

API: Application Programming Interface

xAL: eXtensible Address Language

JAXB: Java Architecture for XML Binding

JTS: Java Topology Suite

WKT: Well-Known Text

LOD: Level of Detail

PostGIS: PostgreSQL Geospatial Extension

STRtree: Sort-Tile-Recursive Tree

STRtree: Spatial Tree

MBR: Minimum Bounding Rectangle

η: Threshold for Detail Level(used in theory)

eta: Threshold(η)(used in code)

DoV: Degree of Visibility

VPage: Visibility Page

VPageIndex: Visibility Page Index

VSS: Vertical Storage Scheme

HSS: Horizontal Storage Scheme

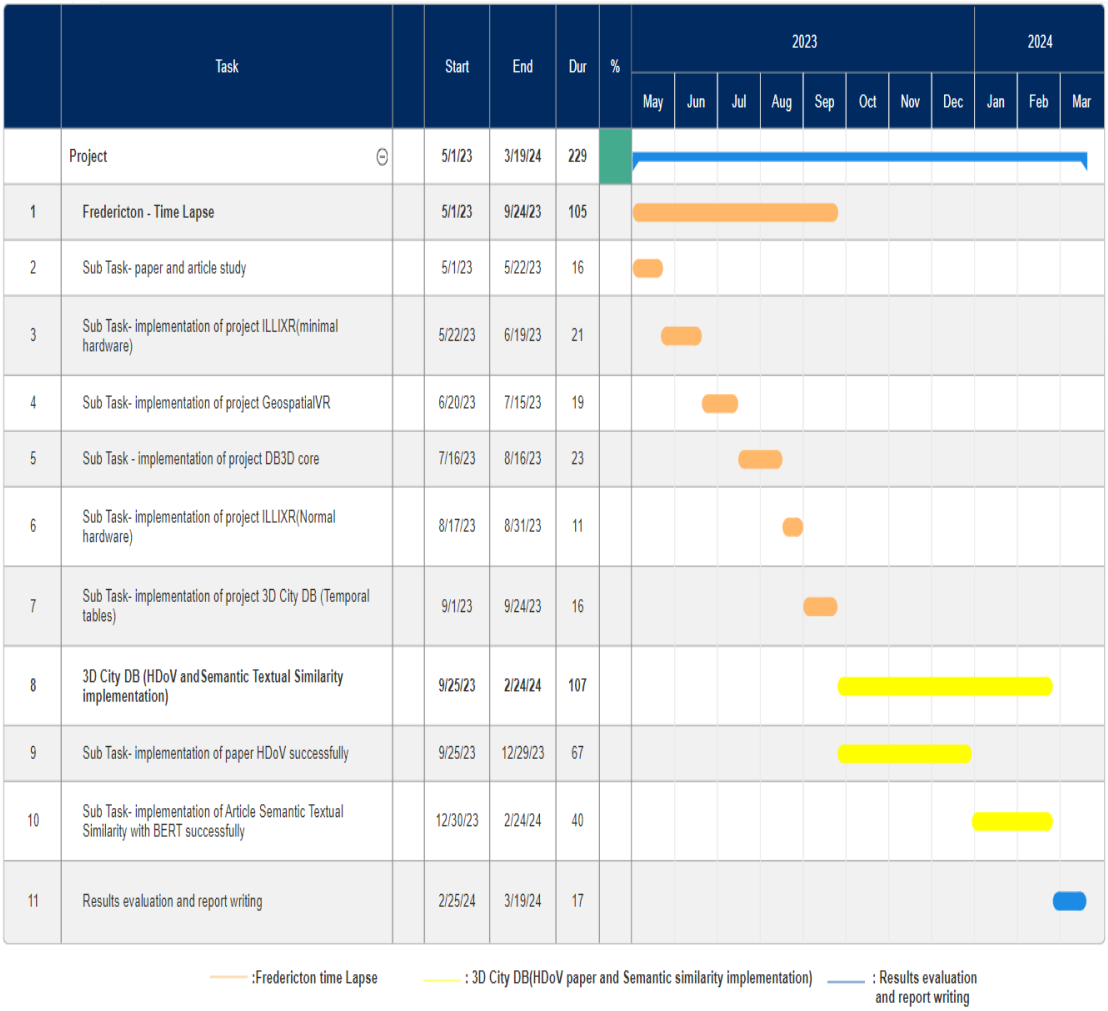
Chapter 1 INTRODUCTION

The growing popularity of the Metaverse, propelled by advancements in virtual reality, has facilitated deeply engaging user experiences in interactive 3D spaces. In these digitally constructed realms, users can engage with both virtual entities and one another, fostering rich and complex interactions. As the intricacy of these virtual worlds escalates, the need for effective organization and retrieval of virtual objects becomes increasingly paramount. (Abbas Al-Ghaili, 2022)

The integration of a spatial-temporal index in these immersive 3D virtual environments is crucial. Such an index would enhance the storage and access of objects within the Metaverse, thereby boosting both user experience and operational efficiency. This research implements an innovative spatial-temporal index, tailored to the distinct characteristics of immersive virtual worlds, which enhances both performance and scalability. The efficacy of this methodology is validated through a series of experiments conducted across diverse virtual environments.

1.1 Timeline and progression (of Novel work)

Table 1: Timeline of project and sub-projects



During the first 4-5 months in our project, we aimed to develop a time-lapse representation of a city, specifically using Fredericton as an example. During our research, we referred to several academic sources to guide our work:

An article from the ACM SIGMOD Blog discussing data management in the Metaverse. (Tan, 2022)

A study by Shou, Huang, and Tan, presented in 2003 at International Conference on data

engineering, which explores the HDoV-tree structure, focusing on its architecture, storage efficiency, and performance. (L. Shou, 2003)

A 2009 paper by Ooi, Tan, and Tung in the SIGMOD Record (Vol. 38, No. 3), examining the integration and management of physical and virtual spaces from a database perspective. (Beng Chin Ooi, 2010)

The 2022 paper presented at EDBT, titled '3DPro,' which delves into querying complex three-dimensional data using methods of progressive compression and refinement. (Dejun Teng, 2022)

We then explored the implementation of Augmented Reality (AR) using ILLIXR (Huzaifa, et al., 2021). Initially, we attempted to run it on a VM without success. Subsequently, we tried using minimal hardware, but found that ILLIXR requires a specific version of Ubuntu and certain hardware specifications. We were able to run only the most basic functionality. Additional hardware, like VR goggles, was required but unavailable to us. (Huzaifa, et al., 2021)

Our goal remained to create a time lapse of a city like Fredericton. We attempted to implement GeospatialVR for urban planning in Unity (Yusuf Sermet, 2021). Initially, we used a trial of Google's ARCore API but later sought a free alternative and approached ISRO. However, we learned that ISRO’s Navic system does not have an API

like ARCore, and we faced hardware limitations. (ISRO, 2023)

We then tried to implement the DB3D core project, based on the paper 'The Story of DB4Geo'. Our aim was to modify it for our purposes, but we encountered issues as only the executable file was accessible, and the source code was missing. Efforts to contact Professor Martin Breunig of Karlsruhe University in Germany for the code were unsuccessful. (geodb/ db3dcore, 2023)

We then explored the implementation of Augmented Reality (AR) using ILLIXR again. We tried to install it again and we prepared a list of required and requisite hardware, and tried to collaborate with faculty already working on AR for hardware but still it was not a success because we were unable to get specific hardware for it. (Huzaifa, et al., 2021)

Subsequently, we attempted to implement immersive time-based visualizations for the database of 3D City DB. This required adding temporal tables, but redesigning the entire schema of 3D City DB was an extensive task and not feasible within 2-3 months. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

Finally, we implemented an index based on the paper 'HDoV-tree: the structure, the storage, the speed' in the database and application of 3D City DB. We explored the concept of Degree of Visibility (DoV) using computational geometry, created an HDoV-tree, and implemented a search algorithm on it. We also tested storage schemes, both

horizontal and vertical, to evaluate their efficiency in storage." (L. Shou, 2003) (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

We compared and found that the type of data matters a lot and HDoV-tree (L. Shou, 2003) can improve the performance of spatio-visual queries significantly visibility queries makes a lot of difference, In our experimental evaluation, there are 3980 visibility queries and our evaluation indicates that the number of visibility queries (10,000) and type of spatial data makes a lot of sense. This also impacts the data storage schemes and time taken to search over them significantly changes.

Lastly, we implemented the semantic search capability based on “Semantic Textual Similarity with BERT” (Winastwan, 2024) note that though it has been implemented successfully but further improvements are required, which will be a future work.

1.2 Contributions

I implemented HDoV index and Semantic similarity in Java and Python on top of DoVs calculated on GML data extracted from Database/Software of 3D City DB.

First, I implemented HDoV logical structure and searching of nodes built and saved in HDoV logical structure mentioned in HDoV paper on DoVs calculated in by Minh Duc Nguyen.

Then, I implemented 2 storage schemes, mentioned in HDoV paper. I implemented Horizontal storage scheme and Vertical storage scheme to store data in form of HDoV logical structure and then implemented logic to search nodes saved in Horizontal and Vertical storage schemes respectively.

Finally, by extending HDoV logical structure I implemented Semantic textual similarity to search and find out all the nodes whose WKT (Well known Text), semantically match with the text entered (by user). This allows the extension of spatio-visual queries with semantic search capabilities.

1.3 Potential Business impact of this research

This research, at its core, explores the implementation of a spatial-temporal index in the context of 3D geo-spatial analytics, underpinned by the Hierarchical Degree-of-Visibility (HDoV) tree structure and augmented by the integration of Semantic Textual Similarity.

This novel approach has far-reaching implications in industries ranging from urban planning to virtual entertainment and e-commerce. By enhancing the organization and retrieval of virtual objects, our methodology not only improves user experience but also paves the way for innovative marketing strategies and customer engagement in digital spaces. The application of this technology in the virtual real estate sector, for example,

can revolutionize the way properties are showcased and explored remotely, offering a more immersive and intuitive experience for potential buyers or tenants. Similarly, in the gaming industry, the enhanced spatial indexing and semantic understanding can lead to more realistic and engaging virtual worlds, potentially increasing user retention and revenue.

To extend the applicability of this technology, we propose incorporating real-time map implementation for practical uses in fields like archaeology and emergency response. This integration can transform the way first responders, such as firefighters and paramedics, navigate and assess complex situations in real-time, offering improved situational awareness and operational efficiency.

For archaeologists, real-time mapping can aid in uncovering and documenting historical and cultural sites, providing a dynamic and interactive way to analyze spatial-temporal data. This application would enable archaeologists to visualize and interpret archaeological sites and findings in a more comprehensive and intuitive manner.

Furthermore, the gaming industry can benefit from improved spatial indexing and semantic understanding, leading to more realistic and engaging virtual worlds, potentially increasing user retention and revenue. The inclusion of semantic similarity search also broadens the scope for more intuitive and contextually relevant interactions within these environments, crucial for sectors like digital marketing and online retail.

The market potential of these advancements is significant, given the growing demand

for sophisticated and seamless virtual experiences, aligning with the increasing global investment in AR and VR technologies. This research, therefore, not only contributes to the academic understanding of spatial-temporal indexing in virtual environments but also offers tangible business applications, highlighting a market ripe for innovation and growth.

1.4 Organization of the thesis:

The rest of the thesis is organized as follows. In Chapter 2 we present the related work, which includes works of 3D city database software (Zhihang Yao, 3dcitydb-docs, latest version-2023.0). In Chapter 3 we introduced Problem Statement. Chapter 4 talks about the approach we used, which includes PostgreSQL database creation and installation of 3D City DB software to populate PostgreSQL database, HDoV-tree implementation on top of DoV (Degree of Visibility) generated from data retrieved from database in form of visibility code. Chapter 5 explains about the design and structure of our code, and shows schema of database and the end points in form of SQL query to extract data from database. This chapter also includes algorithms and implementation details. Chapter 6 elaborates on the algorithms mentioned in the last section and presents experimental results (outputs). Next comes Chapter 7 which presents the challenges we faced in setting up 3D city DB as a tool to calculate DoV. Chapter 8 concludes our findings and work. Chapter 9 includes further work and research which can be performed as future work.

Chapter 2 RELATED WORK

In this chapter we present previous work related to this thesis.

2.1 CityJSON

CityJSON is a data format built upon JSON, offering a subset of the CityGML data model. It serves as a practical alternative to the CityGML interchange format. One of CityJSON's main benefits lies in its compactness compared to CityGML. By leveraging JSON, a widely used and lightweight data format, it allows for a more efficient and easier-to-understand representation of 3D urban models. This results in smaller file sizes for CityJSON files compared to their CityGML counterparts, leading to faster data transfer, reduced storage needs, and improved processing and rendering performance.

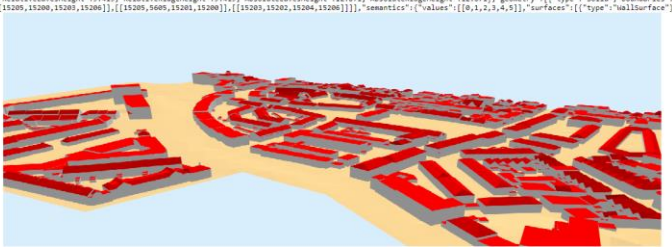
The CityJSON format provides a simplified way to represent geometries, semantics, and metadata for mockups of 3D cities, tackling problems of redundancy and ambiguity found in CityGML. Its efficient design allows creation of software and tools for depicting three-dimensional urban environments, promoting the broader adoption of such models across various domains.

The new encoding method reduces data size and improves usability. JSON integration is evident in both relational and non-relational databases, such as NoSQL. CityJSON can be efficiently stored and used in both types of databases. Furthermore, JSON integration into modern relational and NoSQL databases makes CityJSON highly compatible with

database storage and performance. CityJSON's compatibility with various database systems allows for easier querying, updating, and management of 3D urban models. This flexibility is essential for using such models in diverse fields like planning cities, evaluating ecologies, and infrastructure management. (Hugo Ledoux, 2019)

Regarding the structure of CityJSON, at the top level, there is the root CityJSON object. The metadata property includes a reference system that applies to the entire CityJSON file. Additionally, the cityobjects property contains city object's ID represented by key, and the city object represented by value, both of these i.e ID and Value are part of key-value pairs collection. Each city object includes an attributes property that manages regular and generic attributes. This entity also possesses a geometry attribute that stores an array of geometric entities. Every item within this array includes a semantics property, which oversees a semantics entity. The semantics entity contains a surfaces attribute, which in turn holds semantic surface entities in array format.

Although GML3 (Rouault, 2014) provides 26 ways to represent a polygon, CityJSON restricts this to just one method for representing both geometric and semantic objects. Essentially, CityJSON simplifies the data model of CityGML by standardizing the representation of geometric and semantic objects and treating generic attributes as normal attributes. (Hugo Ledoux, 2019)




```
{"type":"CityJSON","version":"2.0","metadata":{"geographicalExtent
":[78248.66,457604.591,2.463,79036.024,458276.439,37.481],"referen
ceSystem":"https://www.opengis.net/def/crs/EPSG/0/7415"},"CityObje
cts":{"GUID_DBDABF53-7DD5-4C2F-BE7F-
51F29A0CBA16_1":{"type":"BuildingPart","attributes":{"roofType":"1
000","RelativeEavesHeight":3.133,"RelativeRidgeHeight":3.133,"Abso
luteEavesHeight":7.717,"AbsoluteRidgeHeight":7.717},"geometry":[{"
type":"Solid","boundaries":[[[[10915,10991,10992,10916]],[[10991,1
0993,10994,10992]],[[10993,10995,10996,10994]],[[10995,10915,10916
,10996]],[[10995,10993,10991,10915]],[[10916,10992,10994,10996]]]]
,"semantics":{"values":[[0,1,2,3,4,5]],"surfaces":[{"type":"WallSu
rface"},{"type":"WallSurface"},{"type":"WallSurface"},{"type":"Wal
lSurface"},{"type":"RoofSurface","Direction":0,"Slope":90},{"type"
:"GroundSurface"}]},"material":{"":{"values":[[2313,2313,2313,2313
,2314,2315]]}},"lod":"2"}],"parents":["GUID_DBDABF53-7DD5-4C2F-
BE7F-51F29A0CBA16"]},"GUID_8CE54418-E2F7-49A7-9A8D-
C3D172BA62C4_2":{"type":"BuildingPart","attributes":{"roofType":"1
000","RelativeEavesHeight":6.392,"RelativeRidgeHeight":6.392,"Abso
luteEavesHeight":10.075,"AbsoluteRidgeHeight":10.075},"geometry":[
{"type":"Solid","boundaries":[[[[15207,15208,15209,15210]],[[15208
,15211,15212,15209]],[[15211,15213,15214,15212]],[[15213,15215,152
16,15214]],[[15215,15217,15218,15216]],[[15217,15219,15220,15218]]
,[[15219,15207,15210,15220]],[[15219,15217,15215,15213,15211,15208
,15207]],[[15210,15209,15212,15214,15216,15218,15220]]]],"semantic
s":{"values":[[0,1,2,3,4,5,6,7,8]],"surfaces":[{"type":"WallSurfac
e"},{"type":"WallSurface"},{"type":"WallSurface"},{"type":"WallSur
face"},{"type":"WallSurface"},{"type":"WallSurface"},{"type":"Wall
Surface"},{"type":"RoofSurface","Direction":0,"Slope":90},{"type":
"GroundSurface"}]},"material":{"":{"values":[[3612,3612,3612,3612,
3612,3612,3612,3613,3614]]}},"lod":"2"}],"parents":["GUID_8CE54418
-E2F7-49A7-9A8D-C3D172BA62C4"]},
```


Figure 1: Sample of Den Haag data set

Explanation of Sample example CityJson from Den Haag Data set (Figure 1):

1. Metadata: The metadata section provides information about the geographical extent and reference system of the 3D urban model. In this dataset, the geographical extent is defined by the coordinates [78248.66, 457604.591, 2.463] to [79036.024, 458276.439,

37.481], and the reference system is specified as EPSG:7415.

2. CityObjects: CityObjects represent individual entities within the urban model. Each CityObject is identified by a unique identifier (GUID), and in this dataset, we have several building parts represented. For example, "GUID_DBDABF53-7DD5-4C2F-BE7F-51F29A0CBA16_1" represents a building part with attributes such as roof type and heights. Similarly, other building parts are represented with their unique identifiers.

3. Geometry: The geometry section within each CityObject defines the spatial representation of the entity. It consists of solid boundaries defining the shape of the object. Semantics are provided for each surface, specifying whether it represents a wall, roof, or ground surface.

4. Material: Material information is also provided, indicating the material properties associated with each surface. In this dataset, material values are represented numerically.

5. Level of Detail (LoD): Each CityObject includes a level of detail (LoD) attribute specifying the detail level of the geometric representation. In this dataset, LoD is set to "2," indicating a certain level of detail in the representation.

Based on above information the cityJson format of part highlighted in Figure 1:


```
{
  "type": "CityJSON",
  "version": "2.0",
  "metadata": {
    "geographicalExtent": [78248.66, 457604.591, 2.463, 79036.024, 458276.439, 37.481],
    "referenceSystem": "https://www.opengis.net/def/crs/EPSG/0/7415"
  },
  "CityObjects": {
    "GUID_DBDABF53-7DD5-4C2F-BE7F-51F29A0CBA16_1": {
      "type": "BuildingPart",
      "attributes": {
        "roofType": "1000",
        "RelativeEavesHeight": 3.133,
        "RelativeRidgeHeight": 3.133,
        "AbsoluteEavesHeight": 7.717,
        "AbsoluteRidgeHeight": 7.717
      },
      "geometry": [
        {
          "type": "Solid",
          "boundaries": [
            [
              [
                [10915, 10991, 10992, 10916],
                [10991, 10993, 10994, 10992],
                [10993, 10995, 10996, 10994],
                [10995, 10915, 10916, 10996],
                [10995, 10993, 10991, 10915],
                [10916, 10992, 10994, 10996]
              ]
            ]
          ],
          "semantics": {
            "values": [[0, 1, 2, 3, 4, 5]],
            "surfaces": [
              {"type": "WallSurface"},
              {"type": "WallSurface"},
              {"type": "WallSurface"},
              {"type": "WallSurface"},
              {"type": "RoofSurface", "Direction": 0, "Slope": 90},
              {"type": "GroundSurface"}
            ]
          },
          "material": {"": {"values": [[2313, 2313, 2313, 2313, 2314, 2315]]}},
          "lod": "2"
        }
      ],
      "parents": ["GUID_DBDABF53-7DD5-4C2F-BE7F-51F29A0CBA16"]
    }
  },
  "vertices": []
}
```


2.2 3DCityDB

The architecture has been designed to efficiently manage complex 3D urban models by

Table 2: Object class

id	is_ade_class	is_toplevel	classname	tablename	superclass_id	baseclass_id
0	0	0	Undefined	NULL	NULL	NULL
1	0	0	_GML	cityobject	NULL	NULL
2	0	0	_Feature	cityobject	1	1
3	0	0	_CityObject	cityobject	2	2
4	0	1	LandUse	land_use	3	3
5	0	1	GenericCityObject	generic_cityobject	3	3
6	0	0	_VegetationObject	cityobject	3	3
7	0	1	SolitaryVegetationObject	solitary_vegetat_object	6	3

Table 3: City Object

id	objectclass_id	gmlid	envelope	creation_date	last_modification_date	updating_person
1	26	GUID_999_0179	01030000A0E9	40:55.2	40:55.2	citydb_user
2	26	GUID_3A16BB46-96FE-4F19-9127-68DF0C5213FC	01030000A0E9	40:55.2	40:55.2	citydb_user
3	26	GUID_D558D97E-0822-4BC0-8FB6-9CF15255B81A	01030000A0E9	40:55.4	40:55.4	citydb_user
4	34	UUID_a635cfdc-164e-4a43-b12e-8dc23b462fe1	01030000A0E9	40:55.4	40:55.4	citydb_user
5	34	UUID_2f24b9da-72b6-408d-b36a-d2524509fc11	01030000A0E9	40:55.4	40:55.4	citydb_user
6	34	UUID_4b237947-faa4-4f1e-813b-76e36c780242	01030000A0E9	40:55.5	40:55.5	citydb_user
7	34	UUID_05e695d2-d5a3-48a2-bb8d-06cfbf206353	01030000A0E9	40:55.5	40:55.5	citydb_user

leveraging the model of CityGML. Within the database, each CityGML category corresponds to a separate table. The objectclass table (Table 2) is responsible for recording all CityGML category names and their corresponding table names, while also establishing hierarchical connections among these categories. The cityobject table (Table 3) serves as the main repository for city objects, each equipped with a PolygonZ geometry representing its bounding box, stored in the envelope column. This table includes two distinct columns, gml_id and id, designated for storing the identification of each city object, ensuring their uniqueness across different city models.

Table 4: surface_geometry

id	gmlid	gmlid_codespace	parent_id	root_id	is_solid	is_composite	is_triangulate	is_xlink	is_reverse	geometry	cityobject_id
1	ID_5223b8d6-3fb7-420b-a1ae-4c9e10712918	NULL	NULL	1	1	0	0	0	0	NULL	1
2	ID_45318303-f35e-4464-9421-10a248491726	NULL	1	1	0	1	0	0	0	NULL	1
3	UUID_f0adcd6e-9006-4815-87ff-7e65934a0d33	NULL	2	1	0	0	0	2	0	01030000A0E9	1
4	UUID_b22be7c4-3b94-4e97-8e9b-54564c91b02b	NULL	2	1	0	0	0	2	0	01030000A0E9	1
5	UUID_a57fed12-28ec-452c-8cf4-6805cb0f217d	NULL	2	1	0	0	0	2	0	01030000A0E9	1
6	UUID_825400cc-a200-47d2-99ef-a66a23bbe885	NULL	2	1	0	0	0	2	0	01030000A0E9	1
7	UUID_2b4d3ba3-c55c-47b6-aaf0-6b2ce144ed6d	NULL	2	1	0	0	0	2	0	01030000A0E9	1

The surface_geometry table (Table 4) holds significant importance in the 3DCityDB framework. It stores the external surface of a given volume, represented by the PolyhedralSurfaceZ geometry type, enabling spatial operations in three dimensions. This table is organized hierarchically, with parent and child entities, to manage geometries and depict surfaces, solids, and composites. Establishing the parent/child structure requires a unique identifier column, typically using a sequence ID to ensure uniqueness. The addition of the root_id column helps prevent recursive queries. (Karin Staring, 2020)

The structure of 3DCityDB employs a hierarchical organization for geometries, where PolyhedralSurfaceZ geometries are used to keep the external covering of volumes in the solid_geometry column. This design choice aims to optimize 3D spatial operations within the database. However, it is important to note that unlike the XLink concept in CityGML, the implementation of 3DCityDB in PostgreSQL does not support geometry sharing. This constraint arises from the inability of the 3DCityDB PostgreSQL implementation to enable geometry sharing through the surface_geometry table due to its parent-child structure, despite the redundancy avoidance feature provided by CityGML. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

2.3 Implementation of Database of 3D City DB and its tools

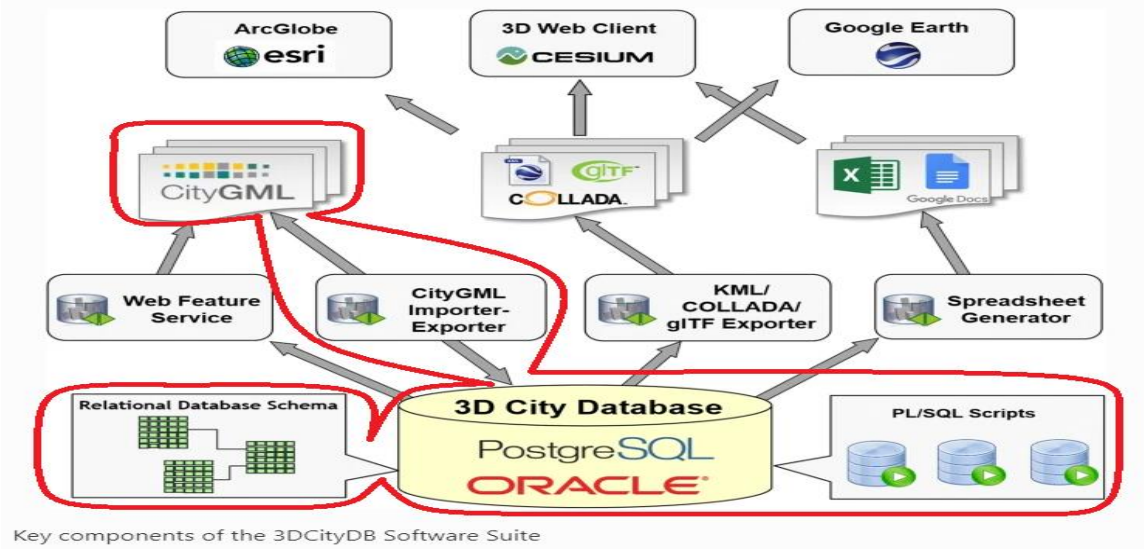


Figure 2: 3D Citydb Software Suite (Zhihang Yao, 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML, 2018)

The focus of the referenced diagram is on the red-bordered area, which encompasses various software and tools from the 3D City DB suite. The core architecture of Database of 3D City (3DCityDB) is based on its relational database schema. This schema incorporates mapping rules that aid in manually converting data from CityGML, an object-oriented format, into this relational structure. The design of the schema emphasizes the use of spatial data types to precisely depict the CityGML objects spatial characteristics. 3DCityDB accomodates two types of spatial relational database management systems (SRDBMS) presently: the proprietary ORACLE Spatial/Locator and the open-source PostgreSQL enhanced with PostGIS. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

This schema of database is more efficient than those generated automatically, requiring

fewer tables. The manual mapping process ensures that the names of tables and attributes in the schema are closely aligned with those in the CityGML model. This alignment simplifies user interactions with the database, enabling effective management, analysis, and querying of extensive and intricate CityGML datasets through SQL. It also provides straightforward integration with external GIS and ETL tools, facilitating the enrichment of augmenting 3D city models by incorporating data into the database.

Additionally, the database schema includes stored procedures, written in PL/SQL for ORACLE or PL/pgSQL for PostgreSQL. These procedures are essential for basic operations such as calculating the bounding volumes of specific 3D entities and the complete city model, deleting objects, and managing spatial indexes. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

2.4 CityGML import/export tool

Within the software package of 3DCityDB, an essential element is the Importer/Exporter tool for CityGML, developed in Java. This tool acts as a bridge to database of 3DCityDB and has a GUI (as shown in the Figure below). The Importer/Exporter is designed to efficiently handle the parsing (reading, writing) of CityGML datasets, regardless of their size. It leverages a specialized low-level Java API, citygml4j, for processing and validating CityGML datasets in accordance with schema definition files. The CityGML, GML, and OASIS xAL formats are transformed into Java classes by employing Java XML Schema binding compiler (xjc) within the JAXB framework (Java Architecture for XML Binding). These classes provide a way to handle CityGML features and attributes

in an object-oriented manner in Java runtime. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

During the importing phase, CityGML datasets are segmented and processed in parts. Each primary feature element, like a Building, WaterBody, or Street, is individually parsed and converted into a Java object as per its designated class in the citygml4j library. These objects of java are subsequently placed in a queue which is buffered and uploaded into the database in parallel, employing a multi-threaded strategy to boost processing efficiency. To optimize the usage of multiple cores of CPU and reduce the overhead associated with thread lifecycles, the system utilizes a thread pool. This pool dynamically adjusts the thread count based on the processing power available from the hardware in use.

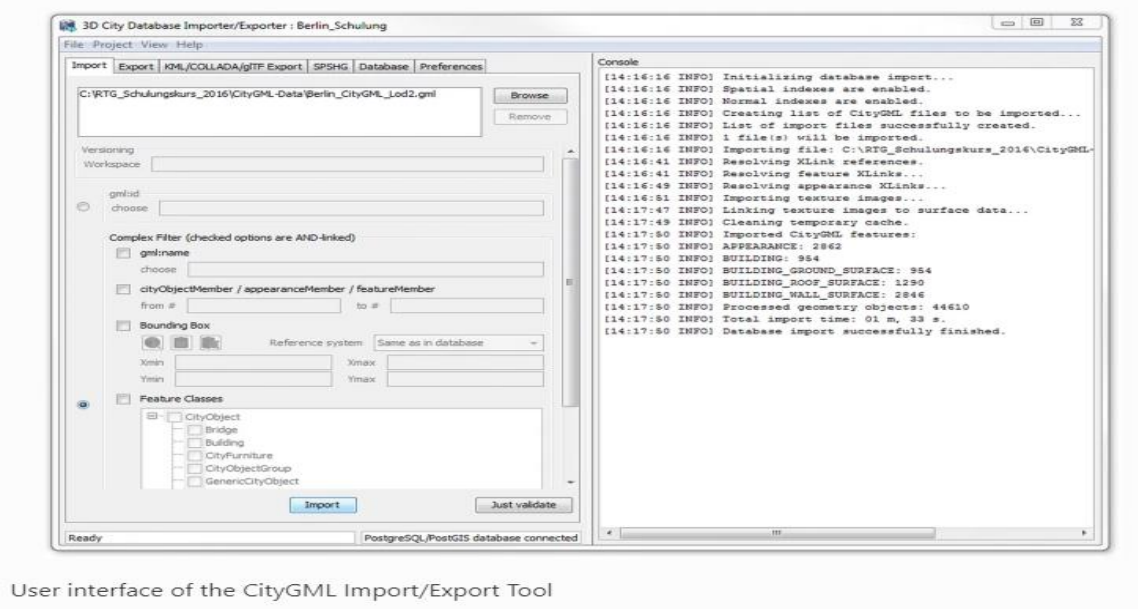


Figure 3: 3D Citydb Import/Export Tool (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

An important part of the importing procedure is dealing with XLinks present in CityGML

datasets. This step is vital because other features or geometries in CityGML might refer to a particular feature or geometry using the GML XLink mechanism. Addressing these XLinks can be a complex task, particularly in scenarios where elements of CityGML file's start refer to those at the end. This situation necessitates loading the complete dataset into the main memory, which can cause memory overflows, a significant concern with the increasingly large CityGML datasets of today.

To tackle this challenge, the initial import of spatial representations and CityGML features is performed without integrating XLink reference details. These references are stored temporarily in the database. After the primary import is finished, the system then processes and incorporates the stored XLink references into the appropriate CityGML data tables. This step marks finishing the import procedure. The depiction of the process is in the Figure below. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

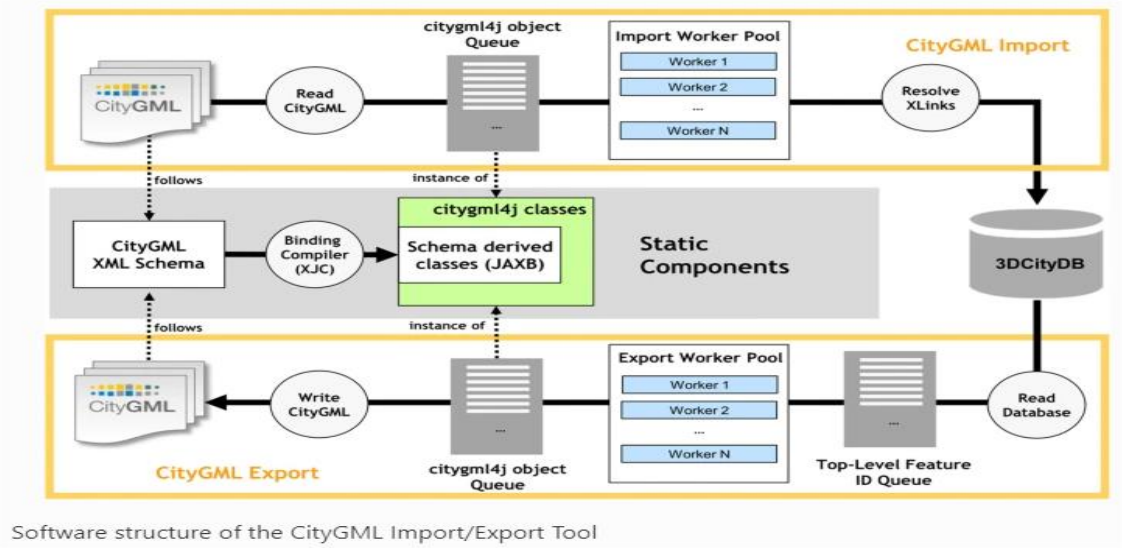


Figure 4: 3D Citydb Import/Export Tool Structure (Zhihang Yao, 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML, 2018)

In the export phase, the initial action is to extract GMLIDs representing top-level

features. This selection is based on criteria set by the user, such as the type of feature class and the specified geographic area, from the database. Following this, a group of multiple worker threads, known as a worker pool, is formed. Each thread in this pool is tasked with handling a specific GMLID. It does so by citygml4j object creation from the data of the CityGML feature obtained from relevant database tables. In the concluding step of the process, these citygml4j objects are transformed into XML elements and then compiled into instance document of CityGML. The above Figure illustrates this process.

It is important to note that the 3DCityDB Importer/Exporter is exclusively designed for checking the compliance of CityGML datasets with their respective XML schemas. However, for validating datasets in terms of semantic and geometrical precision, or for assessing geometric-topological consistency, external applications such as CityDoctor are required. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

2.5 JTS Library

The Java Topology Suite (JTS) is a freely available software library providing a wide array of geometric and spatial capabilities, making it well-suited for managing three-dimensional entities. Developed in Java, JTS allows developers to easily integrate the library into various platforms and applications. Additionally, Java's built-in support for object-oriented programming simplifies the manipulation of complex 3D geometries.

JTS has a significant advantage in its ability to support various geometric classifications, including points, linestrings, polygons, multipoints, multilinestrings, and multipolygons.

It provides essential functionalities like buffer, union, intersection, difference, and symmetric difference, enabling developers to perform advanced spatial analyses on three-dimensional entities. Additionally, JTS supports a variety of topological predicates such as contains, covers, overlaps, and intersects, which are crucial for retrieving and filtering spatial data. (Zhihang Yao, 3dcitydb-docs, latest version-2023.0) (Mikael Johansson, 2002)

The JTS library includes a WKTReader tool, which allows software developers to interpret geometries from the Well-Known Text (WKT) format. The WKT format is widely used for storing and exchanging geometric data because of its readability and user-friendly nature, making it a preferred choice for sharing spatial information. With the WKTReader tool provided by JTS, developers can process various geometry types such as POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, and MULTIPOLYGON. However, it is important to note that the WKTReader in JTS does not support the POLYHEDRALSURFACE Z geometry type.

Despite its inability to directly interpret the POLYHEDRALSURFACE Z geometry type, JTS remains a practical choice for managing three-dimensional objects for several reasons. Firstly, JTS provides a wide range of algorithms and data structures specifically designed for manipulating 3D geometries. Developers can leverage these tools to manipulate and analyze three-dimensional objects by approximating them as compositions of simpler geometries, such as polygons or linestrings. This approach offers flexibility and adaptability in handling complex three-dimensional entities.

Moreover, JTS demonstrates outstanding performance and scalability. It is purposefully crafted to handle large datasets efficiently, making it suitable for managing and analyzing complex three-dimensional entities across various application fields. JTS's performance is further enhanced by its seamless integration with other spatial libraries, notably the widely adopted geospatial database, PostGIS. By combining JTS with other spatial tools, developers can efficiently handle 3D objects, enabling storage, querying, and analysis processes. This interoperability provides a comprehensive solution for managing 3D objects effectively.

2.6 Hierarchical Degree-of-Visibility Tree (HDoV-tree)

The Hierarchical Degree-of-Visibility Tree (HDoV-tree) is a specialized data structure designed to streamline visibility queries in complex three-dimensional (3D) environments. It utilizes a hierarchical spatial subdivision topology that includes geometric, material, and visibility details within its nodes. The HDoV-tree differs significantly from conventional geographic data structures like the R-tree in several key aspects.

The HDoV-tree exhibits a feature called view-variant behavior, meaning that the objects visible to an observer can change depending on their viewing position. In other words, the tree and its nodes can capture different visible elements from various viewpoints, allowing for more adaptable and dynamic visualization evaluations in large 3D

environments. (L. Shou, 2003)

The traversal of the HDoV-tree primarily depends on visibility details rather than spatial closeness. This approach allows for the efficient elimination of branches during traversal when items are barely visible or fail to meet a set visibility threshold. As a result, the HDoV-tree effectively manages visibility queries and reduces unnecessary computations. (L. Shou, 2003)

Lastly, the HDoV-tree demonstrates a significant level of adjustability, allowing it to meet diverse user needs and accommodate different computational capabilities of users and devices. Users can fine-tune the HDoV-tree to achieve optimized performance and tailored user experiences across various platforms and applications, adjusting the level of realism in visible objects as needed. (Hugo Ledoux, 2019)

The main advantage of using an HDoV-tree lies in its ability to effectively and adaptively manage complex three-dimensional environments. This innovative approach improves the efficiency of visibility queries and analysis in a variety of applications, including urban planning, virtual reality, and gaming, by giving priority to visibility data and employing a customizable, view-dependent structure. This method is especially valuable for processing large 3D scenes in real-time. The hierarchical organization of the HDoV-tree enables it to efficiently handle the increasing complexity and size of datasets, making it a valuable asset for modern 3D applications.

Chapter 3 PROBLEM STATEMENT

The objective of this initiative is to conduct an in-depth investigation and enhancement of a high-capacity Hierarchical Degree of Visibility Tree (HDoV-tree). The project involves the implementation of both horizontal and vertical storage mechanisms within the HDoV-tree framework to optimize storage efficiency for search operations, utilizing the Java programming language. The fundamental goal is to develop a tree architecture that excels in essential operations such as adding, removing, segmenting, combining, exploring, and navigating, thereby maximizing its performance and effectiveness for diverse applications.

A critical component of this project is to refine the tree's capacity for accommodating and managing intricate models at varying Levels of Detail (LOD) with high precision. The intention is to boost the tree's applicability and relevance in practical scenarios, particularly in horizontal and vertical storage contexts. This involves dynamically adjusting the model's resolution to meet the demands of real-time data transmission. This adaptive capability is vital for the HDoV-tree to efficiently process extensive datasets, offering significant contributions to fields like urban development, architecture, and environmental impact analysis.

Furthermore, this research will evaluate the scalability, dependability, and memory efficiency of the HDoV-tree in its application to horizontal and vertical storage solutions. The study aims to benchmark the performance of the HDoV-tree against other tree-based

structures. Through comprehensive analysis, we aim to pinpoint areas for enhancement and optimization, enabling the HDoV-tree to be more effective in managing various types of datasets and in large-scale application environments.

Figure 5(a), 5(b): Target problem set of city Den Haag (Fig 5(a): Source: https://ninja.cityjson.org/#, Fig 5(b): Source: https://www.shutterstock.com/search/panoramic-den-haag)

Our aim is to store DoV (Degree of visibility) from different viewpoints of different buildings in Figure 5, which is visualization of cityJson data set of Den Haag City. Viewpoint and bounding box are explained in Figure 9 and Figure 10 in subsequent chapters.

Chapter 4 OUR APPROACH

Here we outline methodology we have employed to address the goal of enhancing the efficiency and accuracy of urban visibility analysis. Our approach is structured around four key steps, each detailed in the following sections. First, in Section 4.1, we focus on the establishment of a robust and scalable database using the Importer/Exporter tool of the 3DCityDB application. This process includes setting up a PostgreSQL database with PostGIS extension (4.1.1), populating the 3DCityDB database schema (4.1.2), importing the LOD2 dataset (4.1.3), and conducting thorough data validation and quality control (4.1.4). These steps ensure the integrity and accessibility of spatial data for further analysis. Next, Section 4.2 delves into the methods used to retrieve and process geometry objects from the database, adapting them to be compatible with the JTS (Java Topology Suite) library. This involves converting POLYHEDRALSURFACE Z geometries into more suitable formats for JTS processing and merging objects with identical IDs for dataset consistency. In Section 4.3 we introduce the utilization of the JTS STR-tree, a dynamic spatial data structure, to arrange the Degree of Visibility (DoV) on building geometries. This section elaborates on the advantages of using the STR-tree for spatial queries in urban settings, emphasizing its efficiency and flexibility in handling dynamic datasets. Finally, Section 4.4 presents our comprehensive approach to evaluating urban visibility. We explain the process of generating a cell matrix, establishing line-of-sight vectors, and conducting intersection tests to compute the visibility levels of various geometries from specific observation points.

4.1 Database creation

We utilized the Importer/Exporter tool included in the 3DCityDB application to manage and process the LOD2 dataset. The following steps outline the conversion process and database setup.

4.1.1 Database Setup: A PostgreSQL database is set up with the PostGIS extension enabled. This combination of PostgreSQL and PostGIS allows for efficient storage and querying of the spatial data contained in the LOD2 dataset.

4.1.2 Schema Population: The database schema for CityDB is populated using the CREATE_DB.sh shell script provided by the 3DCityDB application. This script creates the necessary tables, indexes, and constraints that define the structure of the database.

4.1.3 Data Import: The converted LOD2 data is then imported into the CityDB database using the Importer/Exporter tool. This process involves reading the data from the source file and inserting it into the appropriate tables within the database.

4.1.4 Data Validation and Quality Control: Once the data is imported, we conducted a series of validation and quality control checks to ensure the integrity of the data. This may include verifying the accuracy of the geometries, checking for missing or incomplete attributes, and identifying any inconsistencies or errors within the dataset.

(Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

4.2 Data retrieval

Geometry objects and associated building schema data are retrieved from the PostgreSQL database. In this schema, the column lod2_solid_id holds the IDs of these geometry objects, acting as a foreign key linking to the surface_geometries table (Table 4). This data from citydb is then transferred into the software, establishing a correlation between building IDs and JTS (Java Topology Suite) geometries. However, the type of geometry stored in the surface_geometries table (Table 4), known as POLYHEDRALSURFACE Z, does not match the geometry type used in the JTS library.

To resolve this incompatibility, the ST_ConvexHull function is used to calculate the convex hull of each element. This process transforms the POLYHEDRALSURFACE Z into a series of LINESTRING Z and POLYGON Z geometry types. Following this, the ST_AsText function is applied to convert these convex hulls into a WKT (Well-Known Text) format that is more readable. This WKT data is then processed into the JTS framework using its WKT reader. Finally, to create a unified and organized dataset, objects with identical IDs are merged using the UnaryUnionOp.union method from the JTS library.

4.3 HDoV-tree

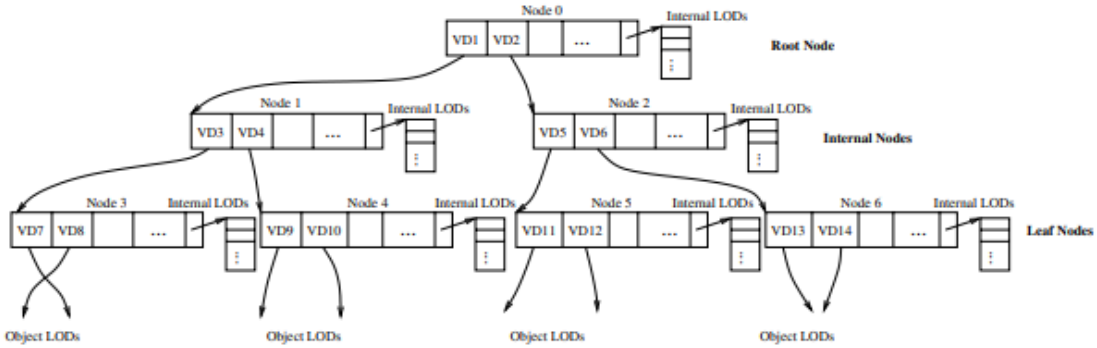


Figure 6: HDoV (Hierarchical degree of visibility)-tree ((L. Shou, 2003))

Figure 6 illustrates the conceptual framework of the HDoV-tree. Within the HDoV-tree, the data in the leaf nodes are formatted as (V D, MBR, Ptr) (L. Shou, 2003).

1. **V D (View-Dependent Value):** This is the Degree of Visibility value of an object. It changes depending on the viewer's perspective, indicating how visible an object is from a specific viewpoint. V D represents DoV (Degree of visibility).
2. **MBR (Minimum Bounding Rectangle):** This is the smallest rectangular area that fully encloses an object. It's a constant value, not affected by the viewer's location or perspective.
3. **Ptr (Pointer):** This indicates the address or reference to the object's Level of Detail (LoD). It's a static value that doesn't change with the viewer's perspective.
4. **LoD (Level of Detail):** The detail level of an object's representation, which can vary based on distance or importance.
5. **Internal LoDs:** Simplified versions of objects grouped within a node, representing an aggregated view.

6. **Object LoDs:** The specific detail levels of individual objects within the HDoV-tree.
7. **Node:** A basic unit in the HDoV-tree structure, which can be either a leaf node or an internal node.
8. **Leaf Nodes:** Terminal nodes in the HDoV-tree that contain direct information about objects.
9. **Internal Nodes:** Nodes in the HDoV-tree that aggregate information from leaf nodes and other internal nodes.
10. **Root Node:** The topmost node in the HDoV-tree, from which all other nodes descend.
11. **DoV (Degree of Visibility):** VD represents DoV. This is a measure of how visible an object is from a specific viewpoint. It's a dynamic value that changes depending on the viewing region. In leaf nodes, it represents the visibility of a single object. In internal nodes, it aggregates the visibility values of all objects encompassed by the node. It's instrumental in determining the visibility of nodes and objects within the HDoV-tree structure.
12. **Understanding DoV and Its Relation:** The Degree of Visibility (DoV) is a key metric in the HDoV-tree, used to quantify the visibility of an object from a particular viewpoint. It's fundamental in determining whether a node or object in the tree is considered visible or not. In leaf nodes, DoV represents the visibility of individual objects, while in internal nodes, it's the sum of the visibility of all objects within that node. This aggregation principle allows for a hierarchical understanding of visibility in the tree, with each node's visibility influencing its overall structure. DoV values

are dynamic, and change based on the viewer's position, making them essential for rendering view-dependent scenes or images.

4.3.1 HDoV-tree implementation

The JTS STR-tree is utilized as the primary spatial data structure in our study to arrange the Degree of Visibility (DoV) on building geometries. Our objective is to improve the efficiency of visibility calculations in urban settings.

The STR-tree (L. Shou, 2003) can facilitate the dynamic insertion and deletion of geometries by employing a top-down construction approach. The ability to be flexible is of utmost importance when dealing with dynamic datasets or real-time updates within urban settings. The STRtree achieves expedient query performance for spatial searches on geometries by utilizing minimal bounding rectangles (MBRs) to hierarchically organize data.

The utilization of the STR-tree's inherent support in the JTS library streamlines the process of development and facilitates the integration with other JTS functionalities. The advantageous aspect of this feature is especially valuable in urban visibility research conducted using Java, as the smooth incorporation with pre-existing spatial software is a crucial consideration. Moreover, the STR-tree has been devised to optimize memory usage, reducing the additional memory required to sustain the index. The consideration of memory consumption is of utmost importance while dealing with extensive datasets,

as it can significantly affect the overall performance and resource utilization of the application. Apart from the intersection queries that constitute the primary focus of our visibility calculations, we investigate the efficacy of STR-tree in facilitating other categories of spatial queries, including range and nearest neighbor queries. The STR-tree's ability to accommodate various spatial relationships makes it a useful instrument in a range of applications, including urban planning, architecture, and environmental impact assessment.

4.4 Visibility analysis

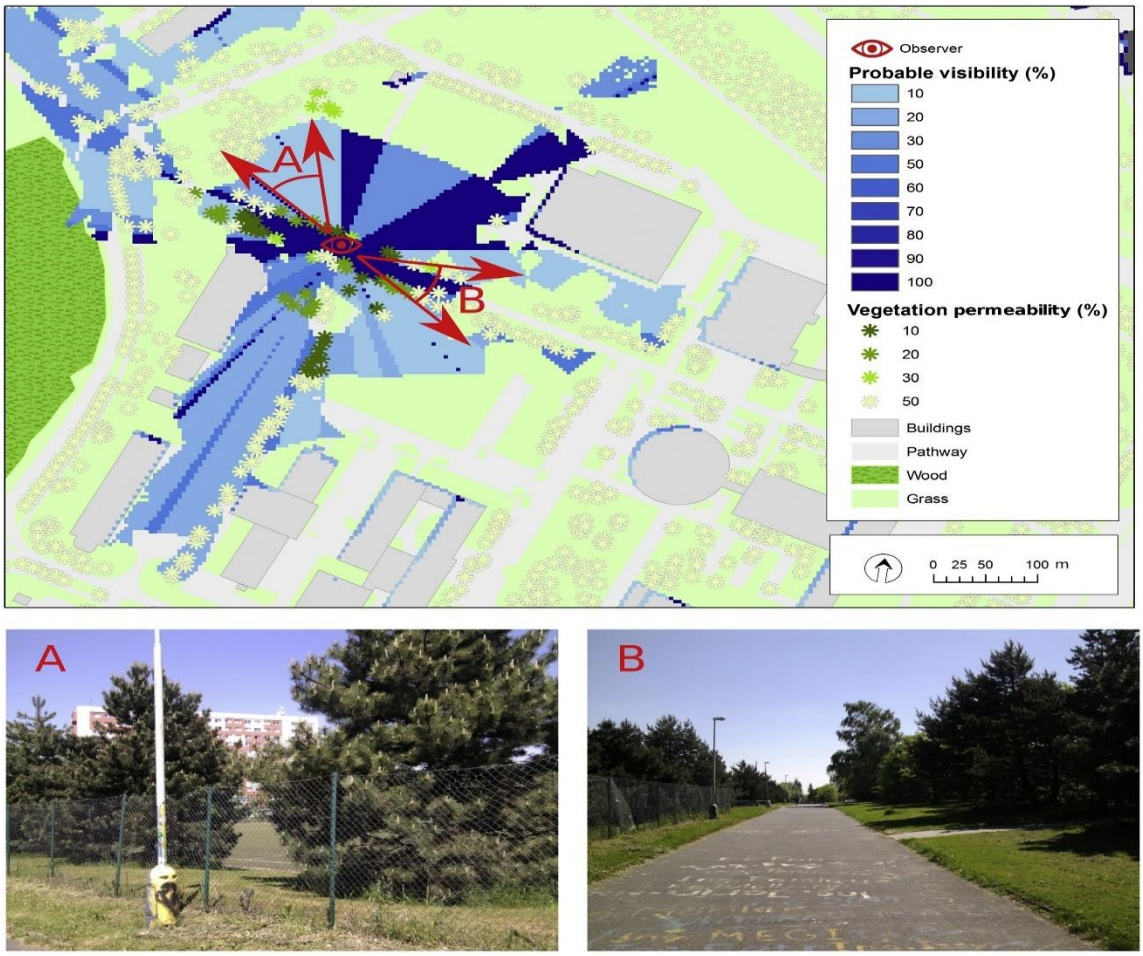


Figure 7: Visibility Analysis Example ((Katerina Ruzickova, 2021))

Considering Figure 7 as an example, it includes two photos captured from the observer's location. The red arrows on the map indicate the viewing direction from where the photos were taken. The image labeled as photograph B illustrates a clear view above the pathway, contrasting with photograph A, where the visibility is partially obstructed by trees surrounding the pathway. Despite the trees, photograph A still reveals the building behind them (Katerina Ruzickova, 2021). We extrapolate this visibility calculation of Figure 7 to our visibility calculation. The primary objective of the visibility analysis is to ascertain the level of visibility of a particular geometry from a designated observation point. To attain this objective, a methodical approach is adopted, commencing with the generation of a cell matrix that encompasses the input geometry, utilizing a designated grid magnitude. Subsequently, these cells are incorporated into a distinct STR-tree, denominated as cellTree, to enable expedient querying at a later stage. After the grid cells have been established, the next step involves the computation of line-of-sight (LOS) vectors connecting the observation point to the centroid of each cell. Through this approach, a direct channel of observation is established to evaluate possible impediments in the urban setting. (L. Shou, 2003)

To achieve precise visibility analysis, it is imperative to detect any potential impediments that could impede the line of sight between the observation point and the grid cells. The process of conducting intersection tests involves querying the main STR-tree, which encompasses all geometries, as well as the cellTree, which encompasses grid cells. The objective is to identify the geometries and cells that intersect with the envelope of the LOS vector. Through the analysis of these intersections, it is possible to accurately

differentiate between observable cells and those that are concealed, while considering the intricacies of the metropolitan terrain. (L. Shou, 2003)

After identifying the cells that are visible and occluded, the subsequent step involves computing the level of visibility for the specified geometry. The said computation is executed by dividing the count of observable grid cells by the overall count of grid cells that enclose the identical geometry. The proportion that ensues is subsequently articulated as a floating-point index that ranges from 0 to 1, thereby offering a lucid indication of the extent of perceptibility of the candidate geometry from the vantage point of observation.

Chapter 5 IMPLEMENTATION

This chapter focuses on explaining how we turn our ideas and methods into a working software program. It is designed to guide readers step by step through every crucial aspect of how our software was created.

In Section 5.1, we talk about the overall structure of our program. We show the main parts of the program and how they work together to figure out how well buildings can be seen in cities. This part covers how we start the program, connect to a database, and handle data with the JTS (Java Topology Suite). We also explain how the program decides where to look from, the areas it covers, and how it compares different ways to calculate visibility. This helps to understand how the program works and what it can do.

Next, Section 5.2, goes into the details of the program's code. Here, we explain the different settings one can adjust, how the program talks to a database, gets data, and processes it. We talk about how the PostgreSQL database is set up, the SQL commands used to get data, and how this data is changed into JTS formats. This part is important if one is interested in the technical details of the program and gives a clear picture of how the software works inside.

Lastly, Section 5.3 gives a thorough explanation of the different algorithms the program uses. It describes each algorithm, including what it does, how it works, and its role in the software. These algorithms are for figuring out visibility, building and using the HDoV-

tree, and storing data in different ways. This part is key to understanding the smart methods and calculations we use in our program for analyzing visibility.

5.1 Design

In this research, we have designed a solution to address the problem of calculating the visibility of buildings from various viewpoints within a city. The program is aimed at assisting urban planning, architectural design, and environmental studies by providing insights into the visual impact of proposed structures on the cityscape.

The program is structured to include several key components that work in unison to achieve the desired outcome. Initially, the program imports the required libraries and sets up the main class, VisibilityCalculator. It then connects to a PostgreSQL database to fetch building geometries and associated information. The retrieved data is converted into JTS geometry objects and organized into a spatial index, STR-tree, for efficient querying.

To analyze the visibility, the program defines multiple viewpoints and bounding boxes. It also compares the performance of two different visibility calculation approaches: one that uses the spatial index (STR-tree) and another that uses a brute force method. The program generates grid cells to represent the building surfaces and assesses the visibility of each cell from the specified viewpoints.

The calculateVisibility and calculateVisibilityBruteForce methods play a vital role in the program's functionality. Both methods accept a geometry representing a building, a viewpoint coordinate, and additional parameters such as grid size as input. They create grid cells to represent the building's surfaces and determine the visibility of each cell from the given viewpoint.

The calculateVisibility method utilizes an STR-tree spatial index to efficiently query intersecting geometries and grid cells. This approach enhances performance when working with large datasets. Conversely, the calculateVisibilityBruteForce method checks for intersections with all other geometries in the dataset, which may be computationally expensive but provides a baseline for evaluating the efficiency of the spatial index approach.

5.2 Description of the code/script

The software employs multiple input parameters, including the dimensions of the bounding box, the location of the viewpoint, and the specifics of the database connection. In this study, the positioning of the viewpoint and the configuration of the bounding box are carefully planned to maximize data point collection in the visibility assessment. This approach offers an extensive overview of the urban setting. To establish the viewpoint, the central coordinates of each geometry in the dataset are first identified. The viewpoint's x, y, and z coordinates are then set at a fixed distance and angle from this central location. This method considers various elements such as terrain elevation,

building heights, and viewing angles to ensure the viewpoint has a clear line of sight to a large number of buildings. The bounding box is created by setting two corners around this central point, with dimensions that are proportionate to the set distance, covering a substantial area around the viewpoint. This broad inclusion ensures a significant number of building geometries are analyzed for visibility. This precise setup of the viewpoint and bounding box is key to gathering an optimal level of visibility data while maintaining computational efficiency. These techniques enhance the understanding of urban environments and are valuable for applications in urban planning, architectural design, and environmental impact assessments.

For the application to function, it must establish a connection with a PostgreSQL database that has been augmented with the PostGIS extension and contains the 3D City Database. The necessary database connection parameters include the server's address, the port number, the username, and the password. Within this database, the building geometries are stored in a table named 'building' (Table 5). The 'building' table (Table 5) in a citydb database is structured as follows: (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

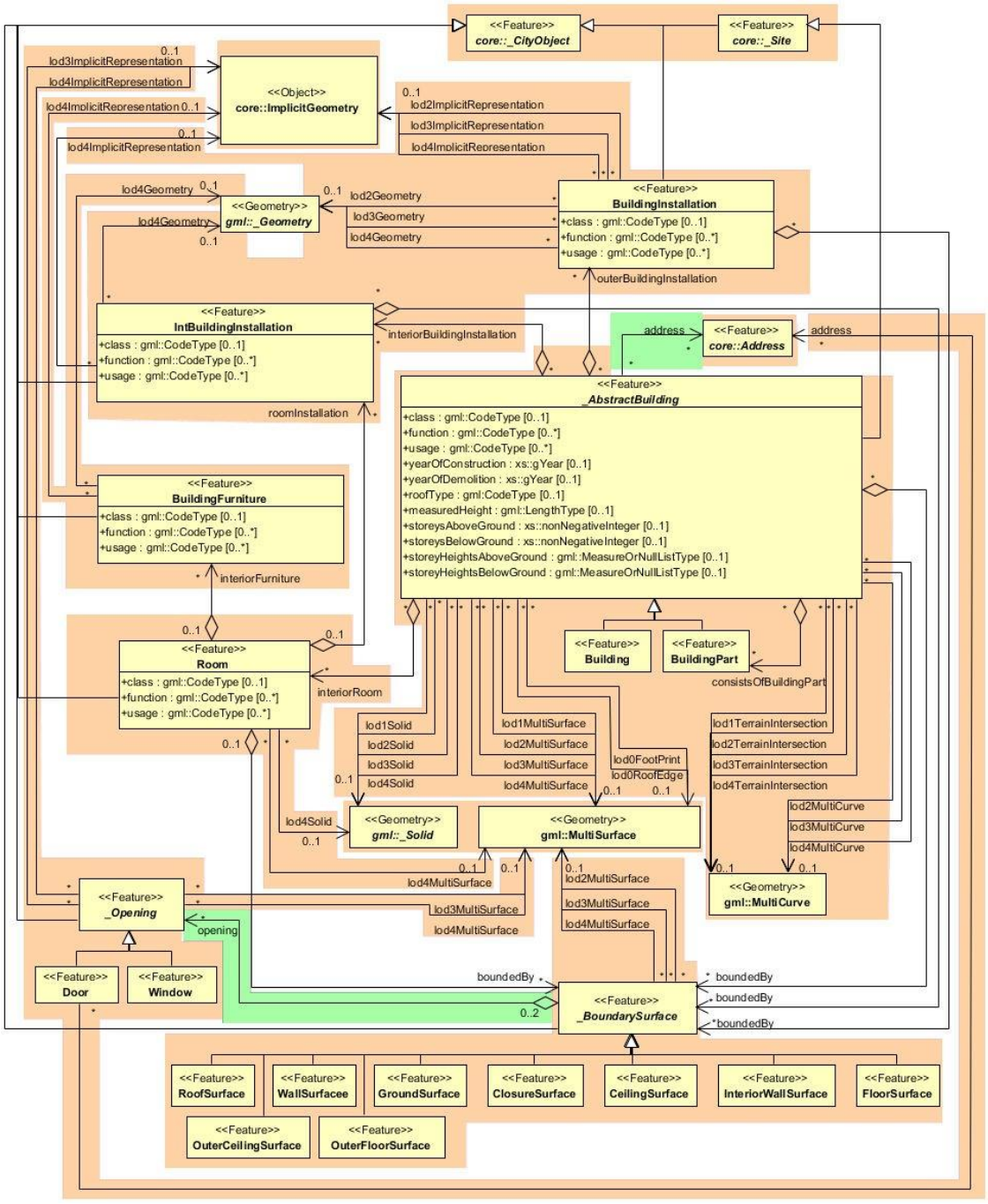


Figure 8: Citydb building schema (Zhihang Yao, 3dcitydb-docs, latest version-2023.0)

In Level of Detail 2 (LOD2) data, the 'lod2_solid_id' column acts as a foreign key linked to the 'surface_geometries' table (Table 4). This particular table stores the identifier for the building's geometric object. The objective is to extract this specific geometry object and process it as a Java Topology Suite (JTS) geometry object to assess the Degree of

Visibility. To accomplish this, the 'populateGeometryList()' function is utilized for fetching the data from the database. Our software primarily focuses on the data pertaining to buildings and surface_geometries from the 3D City Database.

Table 5: Building

id	objectclass_id	building_parent_id	building_root_id	roof_type	lod2_solid_id
1	26	NULL	1	1030	1
2	26	NULL	2	NULL	NULL
3	26	NULL	3	1030	21
22	25	2	2	1000	95
31	25	2	2	1000	137
38	25	2	2	1000	169
45	26	NULL	45	1000	201
52	26	NULL	52	1000	233
53	26	NULL	53	1000	251
56	26	NULL	56	NULL	NULL
59	26	NULL	59	1030	280
61	25	56	56	1000	310

The 'building' table (Table 5) is designated for storing the geometrical representations of buildings, while the 'surface_geometries' table (Table 4) is responsible for holding the 3D surface geometries of these buildings. These tables play an essential role in the analysis of building visibility within urban landscapes. The geometry object is obtained through the following query:

```
SELECT b.id,
ST_AsText(ST_ConvexHull((ST_Dump(ms.solid_geometry)).geom))
as wkt  FROM building b
JOIN surface_geometry ms ON b.lod2_solid_id = ms.id;
```

Figure 9: SQL query to retrieve WKT data

This SQL command is designed to extract the convex hull of the solid geometry for each building in a specific dataset. Analyzing the command, the initial focus is on the SELECT clause, which indicates the intention to retrieve the unique identifier of each building and the convex hull of its solid geometry in the Well-Known Text (WKT) format. WKT is a widely accepted format for textual representation of geometric shapes. The execution of this query involves two tables: 'building' (abbreviated as 'b') and

'surface_geometry' (abbreviated as 'ms'). The JOIN operation links these tables by correlating the 'lod2_solid_id' field in the 'building' table with the 'id' field in the 'surface_geometry' table. This linkage allows for the extraction of solid geometries corresponding to each building in the set. The function ‘ST_Dump’ is applied to decompose the solid geometry into its separate geometric elements. Then, the ‘ST_ConvexHull’ function is employed to streamline the building's geometry, generating a convex hull around it. This convex hull represents the minimal convex shape that encompasses all points of the original geometry. In the context of citydb data, ‘ST_ConvexHull’ divides the POLYHEDRALSURFACE Z into various LINESTRING Z and POLYGON Z geometries, which are interpretable by the JTS library. To conclude, ‘ST_AsText’ is used to transform the convex hull into a human-readable WKT format.

```
61 | LINESTRING Z (78677.255 457975.18100000004 9.51,78672.971 457978.162 9.51)
61 | LINESTRING Z (78672.971 457978.162 9.51,78678.46 457986.014 9.51)
61 | LINESTRING Z (78678.46 457986.014 9.51,78682.748 457982.933 9.51)
61 | LINESTRING Z (78682.748 457982.933 9.51,78677.255 457975.18100000004 9.51)
61 | POLYGON Z ((78672.971 457978.162 9.51,78678.46 457986.014 9.51,78675.708
457982.077
12.676000000000002,78672.971 457978.162 9.51))
61 | POLYGON Z ((78677.255 457975.18100000004 9.51,78679.971 457979.014
12.676000000000002,78682.748 457982.933 9.51,78677.255 457975.18100000004 9.51))
61 | POLYGON Z ((78677.255 457975.18100000004 9.51,78672.971 457978.162
9.51,78675.708 457982.077
12.676000000000002,78679.971 457979.014 12.676000000000002,78677.255
457975.18100000004 9.51))
```

Figure 10: WKT output

The process involves iterating over the ResultSet to extract the building ID and its WKT geometry for each record. The WKT text is then converted into a JTS 'Geometry' object using a JTS WKTReader. This converted geometry is appended to a list of geometries associated with the respective building ID, updating the buildingGeometries map. In

cases of parsing errors, an error message is outputted. After iterating through the ResultSet, the buildingGeometries map contains building IDs along with a list of their corresponding geometries. The JTS library's UnaryUnionOp.union method is then utilized to merge the geometries in each list into a single geometry object. These merged geometries, along with their respective building IDs, are added to the geometryList and buildingIds lists, respectively. This function populates these lists with building geometries and their corresponding IDs from the 'building' and 'surface_geometries' tables, setting the stage for further visibility calculations and analyses.

The application calculates each building's visibility ratio by employing the calculateVisibility() and calculateVisibilityBruteForce() methods, based on a predefined viewpoint and the building's geometry. The calculateVisibility() method starts by creating an STR-tree spatial index, incorporating all building geometries and their IDs, which facilitates efficient spatial queries. For each building, a grid cell set is generated from its geometry. A line of sight is then drawn from the fixed viewpoint to the center of each grid cell. The STR-tree is queried to identify any geometries that might intersect this line of sight, discarding irrelevant geometries based on their minimum bounding rectangles. The intersecting geometries are then examined for actual intersections with the line of sight. Grid cells without intersections are deemed visible, while those with intersections are considered occluded.

The visibility ratio is calculated as the proportion of visible grid cells to the total grid cells. The calculateVisibilityBruteForce() method, though similar in process, does not

utilize the STR-tree. It instead iterates over all dataset geometries to check for line-of-sight intersections. The visibility ratio is calculated using the same formula, but this method is computationally more intensive due to its lack of spatial indexing, particularly in large datasets.

The program's output includes a visibility index for each building within the defined bounding box around the viewpoint, indicating the proportion of the building's grid cells visible from the viewpoint. This data, often structured as a hash map with building IDs and visibility ratios, can be visually represented through color-coded maps in various formats like GeoJSON, KML, or raster images. These maps show buildings with varying color intensities based on their visibility ratios, facilitating easier interpretation and integration into GIS tools or visualization software.

5.3 Algorithms

Algorithm 1: DoV(Degree of visibility) **calculation** (Calculate visibility function based on computational geometry)

```
1: Algorithm calculatevisibility(geometry, viewpoint, STRtree, gridsize):

2:    gridCells <- createGridCells(geometry, gridsize);

3:    STRtree cellTree;

4:    for each cell in gridCells:
5:        cellTree.insert(cell); //create a STRtree for grid cell

6:    for each cell in gridCells:
7:        center cell.getCentroid().getCoordinate(); //get center getCoordinate
```

```
8:      createLine0fSight (viewpoint, center) //raycast

9:      intersectingGeometries <- STRtree.query() // find intersecting geometries with the
                                                                    LineOfSight
10:     intersectingCells <- cellTree.query() // find intersecting cell with line of sight


11:       for each intersectingGeometry in intersectingGeometries; //check if geometry is
                                                                    occluded or not
12:           if intersectingGeometry is geometry: //check if current intersecting geometry is the
                                        same as the target Geometry we're calculating visibility
13:               continue;


14:           if intersectingGeometry.intersects (lineofSight): // check if the current intersecting
                                geometry actually intersects (crosses) the lineOfSight
15:               occluded = true;
16:               break;

17:       If not occluded:,
18:           for each intersectingCell in intersectingCells:
19:               if intersectingCell is cell:
20:                   continue;


21:           if intersectingCell. intersects(lineOfSight):
22:               occluded = true;
23:               break;

24:       If still not occluded:
25:           visibleCount++

26:   return (number of cell intersect) / (total number of cell)
```

```
27: Algorithm createGridCells(geometry, gridsize):
28:     geometry.getBoundingBox();
29:     create grid cell in the bounding box;
30:     for each coordinate x, y in the boundary:
31:         cell <- createPolygon(x,y, gridSize);
32:         if cell intersect with geometry:
33:             intersect <- cell.intersect(geometry);
34:             intersectCell.add(intersect);
35:     return intersectCell;
```

Algo - Calculate Visibility

This algorithm (Calculate Visibility) calculates the visibility of a given 3D object (geometry) from a specific viewpoint using a spatial index structure (STR-tree) and a specified grid size.

Explanation:

1. Grid Cell Creation: The createGridCells function divides the object's bounding box into smaller grid cells. Each cell is checked for intersection with the object, and intersecting parts are added to a list.

2. Visibility Check: For each grid cell, a line of sight is drawn from the viewpoint to the cell's center. The algorithm then checks if any other object in the scene (represented in the STR-tree) intersects this line of sight. If no intersection is found, the cell is counted as visible.

3. Visibility Ratio: The ratio of visible cells to the total number of cells is calculated, giving an estimate of the object's visibility from the viewpoint.

This algorithm is useful in scenarios like urban planning, 3D mapping, and virtual reality,

where understanding visibility from various viewpoints is crucial.

Calculation of DoVs by R-Tree (STR-tree) and Bruteforce is contributed by Minh Duc Nguyen (MS Computer Science, UNB) in form of algorithm 1-Calculate visibility and its Java code.

Algorithm 2: Build HDoV (Hierarchichal degree of visibility) **tree**

```
1: clusteredNodes <- clusterLeafNodes(leafNodes);

2: rootNode <- null;

3: while clusteredNodes is not empty:
4:     newInternalNodes <- [];
5:     for each cluster in clusteredNodes:
6:         internalNode <- createInternalNode(cluster); // Create an internal node for each cluster
7:         newInternalNodes.add(internalNode);
8:     if newInternalNodes.size() is 1:
9:         rootNode <- newInternalNodes.get(0); // Set root node
10:        break;
11:    clusteredNodes <- clusterInternalNodes(newInternalNodes); // Cluster the newly created
                                                                    internal nodes
12: return rootNode;
```

Algo – HDoV-tree Logical Structure

This algo pertains to the construction and utilization of an HDoV-tree (Hierarchical Degree-of-Visibility tree), that integrates the concepts of Level of Detail (LoD), spatial indexing, and Degree of Visibility (DoV) for use in 3D virtual environments.

Explanation:

1. Initial Clustering of Leaf Nodes: The algorithm starts by clustering the leaf nodes of the HDoV-tree. Each leaf node is likely to contain data about an object in a 3D

environment, including its DoV and spatial information.

2. Creation of Internal Nodes: The clustered leaf nodes are then used to create internal nodes. Each internal node represents a cluster and is responsible for aggregating the DoV and spatial information of the leaf nodes it encompasses.

3. Building the Tree Structure: The process of clustering and creating new internal nodes is repeated until there is only one node left, which becomes the root of the HDoV-tree. This iterative process ensures that the tree is built from the bottom up, grouping nodes in a way that respects their spatial and visibility relationships.

Algorithm 3: Search HDoV-tree

```
1: results <- empty list;

2: function traverseNode(node):
3:     if node's DoV is 0:
4:         return;
5:     if node is a leaf:
6:         if node's DoV > η:
7:             results.add(node.MBR);
8:     else:
9:         heuristicValue <- calculateHeuristic(node);
10:        if heuristicValue suggests high visibility:
11:            for each child in node.children:
12:                traverseNode(child);

13: traverseNode(rootNode);
14: return results;
```

Algo-HDoV-tree-Search

The algorithm, "Search HDoV-tree," defines a method for traversing and querying the Hierarchical Degree-of-Visibility (HDoV) tree structure. This algorithm is designed to

efficiently search for visible objects in a virtual environment, taking into account their Degree of Visibility (DoV).

Explanation:

1. Initialization: An empty list, results, is prepared to store the results of the search.

2. Function traverseNode(node): This function is the core of the algorithm. It recursively traverses the nodes of the HDoV-tree starting from the root.

3. DoV Check: If the node's DoV is 0, it means the node (and its children, if any) are not visible from the current viewpoint, so the function returns immediately without further processing this branch.

4. Leaf Node Handling: If the node is a leaf (i.e., it does not have any children), and its DoV is greater than a certain threshold η, its MBR (Minimum Bounding Rectangle) is added to the results. The threshold η is used to filter out objects with low visibility, ensuring that only objects with sufficient visibility are included.

5. Internal Node Handling: For internal nodes, a heuristic value is calculated (the method for this is not detailed in the algorithm but is based on the node's properties such as DoV, possibly its children's DoV, and other factors). If this heuristic suggests high visibility, the algorithm recursively traverses each child of the node.

6. Execution: The traverseNode function is called with the rootNode of the HDoV-tree, starting the recursive traversal.

7. Results: The results list, which now contains the MBRs of all sufficiently visible objects, is returned.

Algorithm 4: Horizontal Storage Scheme

```
1: function setupHorizontalStorage(rootNode, visibilityData, cellId):
2:     if rootNode is null:
3:         return;

    // Initialize VPage and set visibility data
4:     vPage <- new VPage();
5:     for each visibilityEntry in visibilityData:
6:         visibilityValue <- visibilityEntry.getValue();
7:         vPage.setVisibility(cellId, visibilityValue);

    // Set the VPage to the rootNode
8:     rootNode.setVPage(vPage);

    // Recursively set visibility data for child nodes
9:     for each child in rootNode.children:
10:        setupHorizontalStorage(child, visibilityData, cellId);
```

Algo-Horizontal Storage Scheme

"Horizontal Storage Scheme" algorithm is a method for storing and accessing visibility data in a Hierarchical Degree-of-Visibility (HDoV) tree. This approach is designed to handle the dynamic nature of object visibility in a virtual environment, where visibility depends on the viewer's position.

Explanation:

1. Initialization: The function setupHorizontalStorage initializes the horizontal storage scheme for a given node in the HDoV-tree. It takes the rootNode, a collection of visibilityData, and a specific cellId as parameters.

2. VPage Creation: A new VPage is created for each node. The VPage is a data structure

designed to store visibility data for different cells.

3. Setting Visibility Data: The visibility data for the given cellId is set in the VPage. This step involves iterating over visibilityData and setting the visibility values for the corresponding cell.

4. Recursive Assignment: The algorithm then recursively applies this setup to all child nodes of the rootNode, ensuring that each node in the subtree has its visibility data correctly set for the given cellId.

Algorithm 5: Horizontal Storage Search

```
1: function searchHorizontalStorage(node, queryPoint, eta, cellId):
2:     results <- empty list;

3:     function traverseNode(node):
4:         vPage <- node.getVPage();
5:         nodeDoV <- vPage.getVisibility(cellId);

6:         if nodeDoV is 0:
7:             return; // Skip if DoV is 0

8:         if node is a leaf and nodeDoV > eta:
9:             results.add(node.MBR); // Add MBR of the leaf node to results
10:        else if node is not a leaf:
11:            for each child in node.children:
12:                traverseNode(child); // Recurse into children

13:    traverseNode(node);
14:    return results;
```

Algo-Horizontal Storage Search

Explanation:

1. Initial Setup: The searchHorizontalStorage function begins by initializing an empty list result to store the search outcomes.

2. Recursive Traversal Function: A nested function traverseNode is defined to recursively traverse the HDoV-tree, starting from a given node.

3. Visibility Check: For each node, the algorithm retrieves the Degree-of-Visibility (DoV) value from its associated VPage using the cellId. If the DoV is 0, the node (and its subtree) is skipped, as it is not visible.

4. Leaf Node Handling: If a leaf node has a DoV greater than a threshold value eta, its Minimum Bounding Rectangle (MBR) is added to the results list. This indicates that the object is visible and should be included in the query results.

5. Internal Node Handling: If an internal node is encountered, the function recursively calls traverseNode for each of the node's children.

6. Results Return: Finally, the searchHorizontalStorage function returns the results list, containing the MBRs of all visible objects from the query point.

Algorithm 6: Vertical Storage Scheme

```
1: function setupVerticalStorage(rootNode, leafNodes, cellId, visibilityData):
2:     vPageIndex <- new VPageIndex(Nnode); // Assuming Nnode is the number of nodes in
                                                the tree
3:     offset <- 0;

4:     for each node in leafNodes:
5:         if offset >= vPageIndex.getSize():
6:             vPageIndex.resize(vPageIndex.getSize() + 100); // Resize if necessary

7:         node.setVPageIndexOffset(offset);
```

```
8:        vPage <- new VPage();

      // Populate vPage with visibility data for this node
9:        visibilityValue <- visibilityData.getOrDefault(node.getId(), 0.0);
10:         vPage.addVisibilityData(node.getId(), visibilityValue);

11:         vPageIndex.setVPage(offset, vPage);
12:         offset++;

13:    return vPageIndex;
```

Algo-Vertical Storage Scheme

Explanation:

1. Setup Function: The function setupVerticalStorage initializes the vertical storage structure for an HDoV-tree. It takes parameters like the root node, leaf nodes, a cell ID, and visibility data.

2. VPageIndex and VPage:

i)VPageIndex is a list of VPage objects. Each VPage contains visibility data for a specific node.

ii)The algorithm assigns a VPage to each node in the tree, storing the visibility data relevant to that node.

3. Offset Management: Each node in the tree is assigned an offset in the VPageIndex. This offset points to the VPage that contains the visibility data for that node.

4. Visibility Data Storage: The visibility data (DoV values) for each node is stored in its corresponding VPage.

5. Dynamic Resizing: The VPageIndex can dynamically resize if the initial size is insufficient to accommodate all nodes.

Algorithm 7: Vertical Storage Search

```
1: function searchVerticalStorage(node, queryPoint, eta, vPageIndex):
2:     results <- empty list;

3:     function traverseNode(node):
4:         offset <- node.getVPageIndexOffset();
5:         vPage <- vPageIndex.getVPage(offset);

6:         if vPage is null:
7:             return; // Skip if no V-page

8:         nodeDoV <- vPage.getVisibility(node.getId());

9:         if nodeDoV is 0:
10:            return; // Skip if DoV is 0

11:        if node is a leaf and nodeDoV > eta:
12:            results.add(node.MBR); // Add MBR of the leaf node to results
13:        else if node is not a leaf:
14:            heuristicValue <- calculateHeuristic(node);
15:            if heuristicValue suggests high visibility or nodeDoV > eta:
16:                for each child in node.children:
17:                    traverseNode(child);

18:    traverseNode(node);
19:    return results;
```

Algo-Vertical Storage Search

Explanation:

1. Purpose: The algorithm searches the HDoV-tree to find objects that are visible from a given query point using visibility data stored in a Vertical Storage Scheme (VSS).

2. Traversal Method: It recursively traverses the HDoV-tree. At each node, it checks visibility using the Vertical Storage Scheme, which associates nodes with their visibility data stored in V-pages.

3. Visibility Check:

i)Each node in the tree has an offset in the V-page-index pointing to its visibility data.

ii)The algorithm fetches this data to determine if the node is visible from the current viewpoint.

4. Decision Making:

i)For leaf nodes, if the Degree-of-Visibility (DoV) is greater than a threshold (η), the node's Minimum Bounding Rectangle (MBR) is added to the results.

ii)For internal nodes, the algorithm uses a heuristic to decide whether to continue traversing the tree or to add the node's MBR to the results based on the node's DoV value and other criteria.

Algorithm 8: Semantic Textual Similarity Calculation

Java end point:

1: Algorithm getSemanticSimilarity(sentence1, sentence2):

2: Initialize an HTTP client (CloseableHttpClient).

3: Create an HTTP POST request to the Flask server URL (e.g., "http://localhost:5000/similarity").

4: Construct a JSON object with the key "sentences" and the value as an array containing sentence1 and sentence2.

5: Set the JSON object as the entity of the HTTP request.

6: Set the appropriate headers for the HTTP request (Content-Type as application/json).

7: Send the HTTP request and receive the response.

8: Extract the response content as a string.

9: Parse the response string into a JSON object.

10: Retrieve the "similarity" value from the JSON object.

11: Return the similarity score (a floating-point number).

12: End Algorithm

13: Algorithm processNodeBasedOnSimilarity(userInputWKT, similarityThreshold):

14: For each node in the HDoV Tree:

15: Retrieve the textual description of the node.

16: Call getSemanticSimilarity with the node description and userInputWKT.

17: Obtain the similarity score.

18: If similarityScore > similarityThreshold:

19: Process the node as relevant (e.g., mark it visible, add to a list, etc.).

20: End Algorithm

Python end point:

Algorithm for Semantic Textual Similarity Calculation Using Sentence Transformers in Python:

1: Algorithm initializeSentenceTransformerModel:

2: Import the SentenceTransformer class from the sentence_transformers library.

3: Load the Sentence Transformer model (e.g., 'bert-base-uncased') using SentenceTransformer.

4: Return the loaded model.

5: Algorithm encodeSentencesUsingModel(model, sentences):

6: For each sentence in the list of sentences:

7: Use the model to encode the sentence, obtaining its embedding vector.

8: Return the list of embedding vectors.

9: Algorithm calculateCosineSimilarity(embedding1, embedding2):

10: Convert the embedding vectors to PyTorch tensors.

11: Calculate the cosine similarity between the two tensors using torch.nn.functional.cosine_similarity.

12: Return the cosine similarity score as a floating-point number.

13: Algorithm flask Server Setup:

14: Import necessary modules from Flask and initialize the Flask app.

15: Define a route (e.g., '/similarity') to handle POST requests.
16: Endpoint /similarity (POST):
17: Retrieve the JSON data from the incoming request.
18: Extract the 'sentences' array from the JSON data.
19: Call encodeSentencesUsingModel with the loaded model and extracted sentences.
20: Call calculateCosineSimilarity with the two sentence embeddings.
21: Return the similarity score in a JSON response format.

Algo- Semantic Textual Similarity Calculation

The algorithm represents a sophisticated integration of Java and Python to perform Semantic Textual Similarity (STS) calculations. It leverages the advanced natural language processing capabilities of the BERT model, implemented via the Python-based Sentence Transformers library, and interfaces with a Java application through a RESTful API.

At Java's End:

1. **Semantic Similarity Retrieval:**
    - The Java application acts as a client, initiating an HTTP request to a Python-based Flask server.
    - It sends two sentences in JSON format, requesting their semantic similarity assessment.
2. **Node Processing Based on Similarity:**
    - In a hierarchical data structure, each node's textual content is evaluated.
    - Nodes are compared with user input for semantic similarity.
    - Nodes surpassing a similarity threshold are marked for further processing,

recognizing their relevance to the input query.

At Python's End:

1. **Model Initialization:**
    - The Flask server, upon initialization, loads the BERT-based Sentence Transformer model.
    - This pre-trained model is renowned for its efficacy in understanding textual semantics.
2. **Sentence Encoding:**
    - Sentences received in the request are converted into numerical embedding vectors.
    - These embeddings capture the nuanced semantic features of the text.
3. **Cosine Similarity Calculation:**
    - The core of the algorithm lies in calculating the cosine similarity between sentence embeddings.
    - This step quantifies the semantic closeness of the two sentences.
4. **Flask Server as an API Endpoint:**
    - The Flask server is configured to expose an endpoint that handles POST requests for similarity calculations.
    - It receives sentence pairs, processes them through the model, and returns similarity scores.

5. **Response Generation:**

   - The server's response, containing the similarity score, is sent back to the Java client.
   - This score, a floating-point number, is a quantitative representation of how similar the two sentences are, semantically.

Chapter 6 EVALUATION

This chapter is dedicated to assessing the effectiveness of our software in real-world scenarios. In this chapter, we systematically analyze and compare different methodologies implemented in our software, using practical experiments to understand their strengths and weaknesses. In Section 6.1, we introduce the Den Haag dataset, which forms the basis of our testing. This dataset, which includes detailed representations of buildings in The Hague, Netherlands, allows us to simulate realistic urban environments for our visibility analysis. We explain how we've set up the experiments, including choosing viewpoints, establishing bounding boxes, and defining the criteria for our analysis. This section is crucial as it lays the foundation for the subsequent experiments, ensuring that they are carried out in a controlled and consistent manner.

Following this, Sections 6.2 and 6.2.1, present the outcomes of our experiments. Here, we reveal the performance differences between the STR-tree method and the brute force approach, two techniques used in our software for visibility calculation, and we also compare results of HDoV (Naive), Horizontal and Vertical storage schemes in terms of search efficiency and storage efficiency and then we share our findings of similarity of input text of user with the text of existing nodes. We discuss the efficiency of these methods in terms of speed and accuracy, providing insights into their practical applicability in various scenarios. This section is pivotal as it not only showcases the results of our tests but also offers an analysis of why certain methods performed better than others under specific conditions.

Finally, Section 6.2.2 presents a more quantitative analysis of our findings. Here, we compare the execution times and storage sizes of different methods, such as the Naïve

(HDoV), Horizontal Storage, and Vertical Storage approaches, when applied to various data structures. This section is pivotal for a data-driven understanding of our software's performance, allowing us to draw concrete conclusions based on empirical evidence.

This chapter includes a thorough discussion of unexpected findings, like the surprising efficiency of the brute force method in certain scenarios, and an analysis of the effectiveness of different storage schemes used in our software which also includes surprising efficiency of horizontal scheme over vertical scheme. By the end of this chapter, readers will have a clear understanding of how our software performs in practical situations, the implications of using different methodologies, and the factors that influence their effectiveness. This comprehensive evaluation is essential for validating the reliability and utility of our software in real-world urban visibility analysis.

6.1 Experimental setup

The experimental framework for this study utilizes the Den Haag dataset, comprising 3D rooftop Configurations and a 2.5D terrain layout of the Hague in Netherlands and its adjacent areas. Originating from aerial imagery captured in 2010 and earlier versions of BAG data from the same period, the dataset encompasses around 100,000 building representations within The Hague and roughly 12,500 in nearby municipalities. It includes Detail Level (LOD2) data, crucial for computing the Degree of Visibility (DoV) in subsequent stages of the experiment.

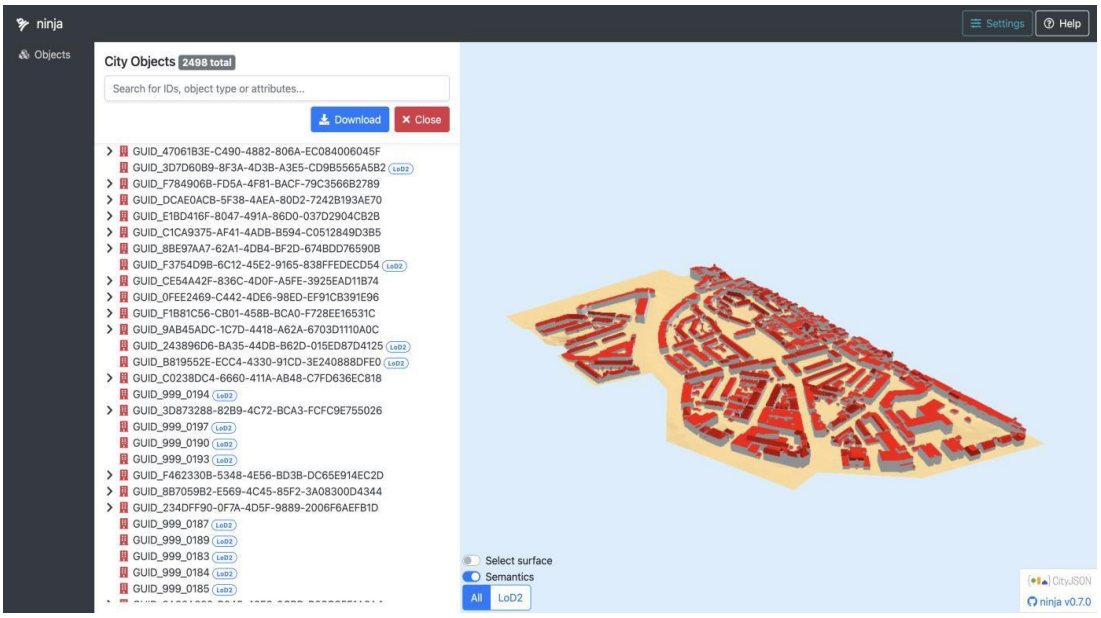


Figure 11: The city of Den Haag on cityJson visualization

The main goal of this experiment is to evaluate and contrast the effectiveness of the STR-tree method against the brute force approach in determining visibility values under specific conditions. To this end, six analogous viewpoints are established for both methodologies. At each viewpoint, a random coordinate is chosen as the focus point for an observer's gaze. To maintain consistency in the visual field across these six experimental viewpoints, six bounding boxes are generated, each delineating the visual range for a viewpoint. Following this setup, the software calculates the degree of visibility from every building within these ranges for both the STR-tree and brute force techniques. Additionally, the time taken by each method is meticulously recorded and documented. Leveraging the DenHaag dataset facilitates a large-scale experiment, involving a substantial number of building models and a variety of viewpoints. Through this comparative analysis of the STR-tree and brute force methods, the experiment aims to shed light on the practical effectiveness of these techniques in real-life settings. The inclusion of execution time as a measurement criterion adds an extra layer of insight,

highlighting the operational feasibility and efficiency of each method.

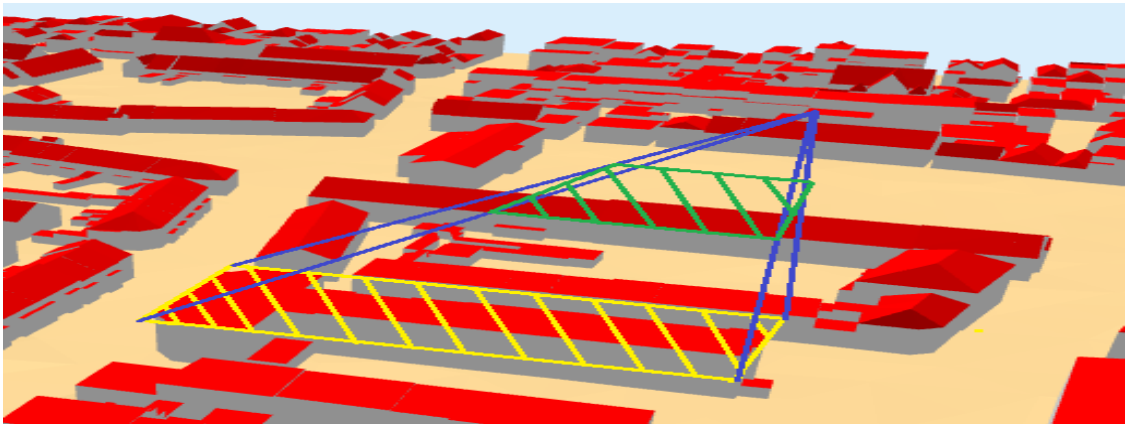


Figure 12: Viewpoint and bounding box

In summary, the experimental design of this project is methodically orchestrated to ensure the reliability and comparability of the findings. Employing the DenHaag dataset and establishing six comparable viewpoints with defined bounding boxes enable a thorough examination of the STR-tree and brute force methods. The process of calculating the degree of visibility and noting the execution times offers critical perspectives on the performance of each approach.

6.2 Experimental results

The experimental findings reveal that the brute force technique was able to achieve query times that were up to 50% quicker than those using the STR-tree method.

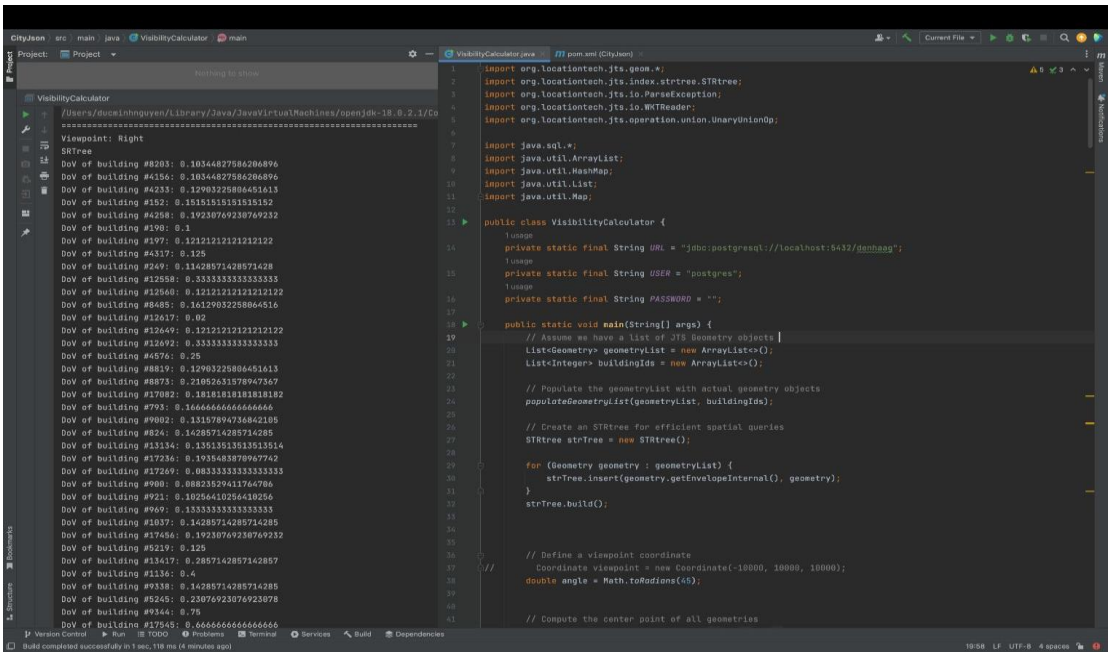


Figure 13: Program result

A plausible reason for this unexpected outcome could be attributed to the small size of the dataset utilized in the experiment. The limited size of the dataset may have amplified the overhead costs associated with the STR-tree method, outweighing its potential advantages. Additionally, the experiment's reliance on static, pre-established fields of view could have further escalated the overhead involved in constructing the STR-tree index. On the other hand, the brute force method may have leveraged these conditions to its advantage, resulting in more streamlined and efficient computations.

These results underscore the critical importance of rigorously assessing the specific needs and limitations inherent to a task when deciding on an algorithmic approach. Although the STR-tree indexing method is typically better suited for handling complex and resource-intensive spatial queries, the brute force method appears to be more efficient for simpler queries. However, due to the limited scale of the dataset in this

experiment, further extensive testing is required to definitively confirm this observation.

Additionally, the experiment also demonstrated that searches conducted using the horizontal scheme were, on average, 66.5% faster compared to those conducted with the Vertical Scheme. This unexpected and notable finding could be influenced by several factors:

1. Data Characteristics: The distinctive features of the data in our 3980 queries may differ from the data used in the referenced paper's experiments. Variations in the distribution, density, and type of geometry can significantly affect the efficacy of spatial queries and visibility calculations. It is conceivable that our data set possessed certain qualities that were more conducive to the horizontal scheme, thereby improving its performance.

2. Hardware and Environmental Variations: The outcomes of the tests might also be impacted by the specific hardware Configurations and environmental settings in which they were conducted. Variables such as CPU efficiency, memory size, disk speed, and Java Virtual Machine (JVM) Configurations could all play a role in influencing execution times.

3. Dataset Size and Complexity: The disparity in the size and complexity of the dataset used in our 3980 queries compared to the 10,000 queries in the original paper may also be a contributing factor. Datasets that are smaller or less complex may be more effectively processed using certain storage schemes.

These findings stress the importance of a thorough and detailed analysis of the specific

demands and constraints of a task when selecting an algorithmic strategy. While Vertical Storage is generally preferred for spatial queries involving more than 10,000 queries or complex geometries, the horizontal approach appears to be more suitable for less complex geometries and a smaller number of queries. However, extensive testing beyond the confines of this project is necessary to substantiate this observation.

6.2.1 Result Output Explanation:

STR-tree:

1-Right:

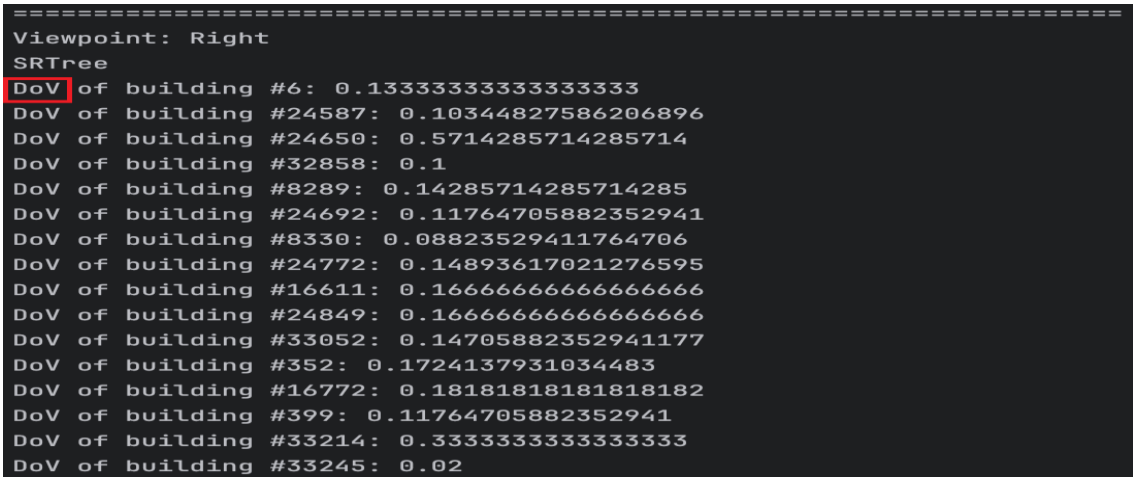


Figure 14: DoV STR-tree

DoV STR-tree Indicates that the Spatial R-Tree data structure is used for spatial querying. DoV of Building# ID: Each line shows the Degree of Visibility (DoV) of a specific building identified by its ID. The DoV value (a number between 0 and 1) represents the visibility of the building from the specified viewpoint. A higher DoV indicates greater visibility. Calculate Visibility algorithm is used. This algorithm calculates the visibility of a geometry (like a building) from a specific viewpoint. It uses

a grid-based approach and spatial querying to determine how much of the geometry is visible.

1 Grid Cell Creation: The createGridCells method divides the geometry's bounding box into smaller grid cells. It checks which of these cells intersect with the geometry, indicating the parts of the geometry that are potentially visible.

2 Line of Sight Calculation: For each intersecting cell, a line of sight is drawn from the viewpoint to the cell's center. This line is used to determine if the cell is visible or occluded by other geometries.

3 Occlusion Checking:

i) Intersecting Geometries: The algorithm queries the STR-tree to find geometries that intersect with the line of sight. If any intersecting geometry (other than the target geometry) crosses the line, the cell is considered occluded.

ii) Intersecting Cells: Additionally, it checks for intersections with other grid cells. If any other cell intersects the line of sight, occlusion is marked.

4 Visibility Counting: If a cell is not occluded, it's counted as visible. The algorithm sums up the number of visible cells.

5 DoV Calculation: The Degree of Visibility is calculated as the ratio of visible cells to the total number of cells in the grid. A higher ratio indicates a higher visibility.

The output shows the calculated visibility (DoV) for each building from a specific viewpoint. The calculateVisibility algorithm effectively determines how much of each building is visible by analyzing the intersections of lines of sight with both the target geometry and other geometries in the environment. This approach allows for an accurate

representation of visibility in complex 3D environments.

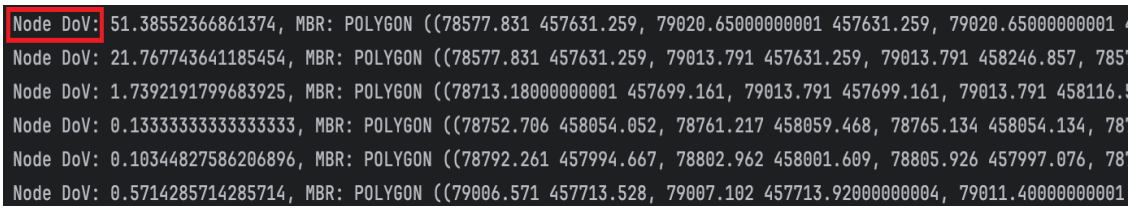


Figure 15: HDOV node DoV, MBR (Minimum Bounding Rectangle), LoD(Level of details) details

1. Node DoV (Degree of Visibility): This appears to be a measure of how visible a particular node (or object) is from a certain viewpoint. The value is double, and higher values indicate greater visibility. For example, a node with DoV: 51.38552366861374 is more visible than one with DoV: 0.10344827586206896.

2. MBR (Minimum Bounding Rectangle): This is a common concept in spatial indexing. Each node has an associated MBR defined as a polygon. The MBR is the smallest rectangle that completely encloses the geometry of the node. It's used for spatial queries and operations, like finding out if a point lies within a node or if two nodes overlap.

3. LoD (Level of Detail): This typically refers to the level of detail in which a node is represented. A lower LoD (e.g., 0.0) might mean a more general, less detailed representation, whereas a higher LoD (e.g., 2.0) indicates a more detailed representation. In 3D modeling and GIS, this is often used to optimize rendering or analysis by using simpler models where less detail is acceptable.

The operations involve:

- Constructing tree structure (R-tree) to manage spatial data efficiently.
- Calculating the visibility of different nodes (geometries) from a given viewpoint.

- Handling different levels of detail for rendering or analysis purposes.

```
Traversing HDoV-tree for viewpoint: Right
Node distance from viewpoint: 99672.54442888952, DoV: 51.38552366861374
Node distance from viewpoint: 99672.54442888952, DoV: 23.916925078967157
Node distance from viewpoint: 99672.54442888952, DoV: 2.0353160827694543
Node distance from viewpoint: 99672.73725702336, DoV: 0.3333333333333333
Node distance from viewpoint: 99679.57455452136, DoV: 0.11764705882352941
Node distance from viewpoint: 99681.96335253392, DoV: 0.5714285714285714
Node distance from viewpoint: 99682.54714778565, DoV: 0.024390243902439025
Node distance from viewpoint: 99778.57580674907, DoV: 0.15151515151515152
Node distance from viewpoint: 99801.75732560595, DoV: 0.23076923076923078
Node distance from viewpoint: 99806.44965167389, DoV: 0.125
Node distance from viewpoint: 99844.55274215655, DoV: 0.23809523809523808
Node distance from viewpoint: 99857.8325970349, DoV: 0.06666666666666667
Node distance from viewpoint: 99897.43789059726, DoV: 0.17647058823529413
Node distance from viewpoint: 99676.16642888954, DoV: 3.4075533661740556
```

Figure 16: HDoV-tree traversal

i) Node distance from viewpoint: This represents the distance from the viewpoint (camera or observer's location) to a particular node in the HDoV-tree. A node here could be an internal node or a leaf node in the tree.

ii) DoV (Degree of Visibility): This value indicates the degree of visibility of the node from the given viewpoint. A higher DoV means more visibility, while a lower DoV suggests less visibility or more obstruction from other objects in the scene.

Relating Output to HDoV-tree:

1. Spatial Structure and Visibility: The HDoV-tree combines spatial data structure (R-tree) with visibility information. Each node in this tree represents a spatial region (with its MBR - Minimum Bounding Rectangle) and contains information about the visibility (DoV) of objects in that region from a certain viewpoint.

2. Traversal Based on DoV: The HDoV-tree is traversed based on the DoV values rather than purely spatial content. This means the tree traversal prioritizes nodes (or regions) with higher visibility from the current viewpoint.

3. Dynamic Visibility: The visibility of nodes is dynamic, depending on the position of the viewpoint. In your output, each line corresponds to a different node's visibility from the specified viewpoint. The same spatial structure (node layout) is used, but DoV values change as the viewpoint changes.

4. Threshold for Detail Level: Threshold DoV value (η) balances visual fidelity and performance. Nodes with a DoV higher than this threshold can be loaded in greater detail, while those with lower DoV might be represented with coarser details.

5 Efficiency in Data Loading: By focusing on nodes with higher DoV, the HDoV-tree approach minimizes the amount of data to be loaded for rendering, improving performance. This is evident in the output, where each node's DoV guides the level of detail needed for rendering.

This output is a representation of how an HDoV-tree functions in assessing and rendering 3D environments based on visibility from different viewpoints. Each line in the output corresponds to a node in the HDoV-tree, providing insights into the node's distance from the viewpoint and its visibility (DoV), which directly influences rendering decisions.

Figure 11, 12 Conclusion:

It details how the tree combines LoD, spatial indexing, and DoV into a single structure, which is dynamic and view-dependent. The HDoV-tree is used to optimize the retrieval and display of 3D objects based on their visibility from a particular viewpoint. This optimization involves using different LoDs and aggregating DoV values to minimize

data load and improve performance. The paper also discusses different storage schemes for the HDoV-tree, including horizontal and vertical storage schemes, each with its trade-offs in terms of storage cost and performance.

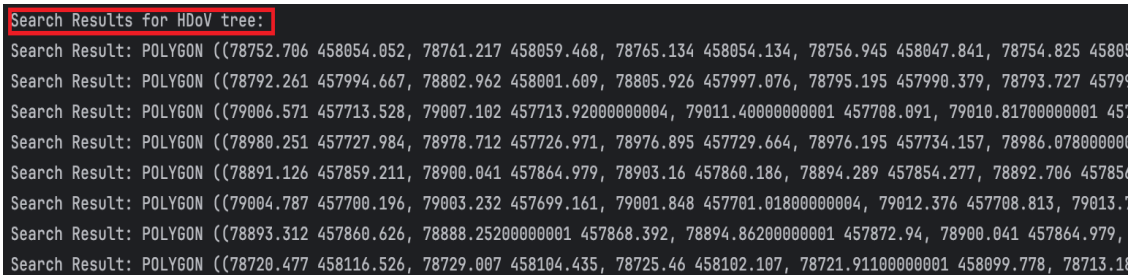


Figure 17: It details how the tree combines LoD

Output involves searching through hierarchical structure HDoV-tree for visible geometrical entities. The output includes a series of "POLYGON" objects represented in Well-Known Text (WKT) format, which are likely the results of the visibility search.

1 HDoV-tree: HDoV-tree integrates Level of Detail (LoD), Degree-of-Visibility (DoV), and spatial indexing. It's used to efficiently determine which objects in a virtual environment are visible from a given viewpoint. This structure is dynamic, as the visibility of objects changes with the viewpoint.

2 Node Structure: Each node (both leaf and internal) in the HDoV-tree contains a DoV, an MBR, and possibly a pointer to child nodes or object LoDs. The DoV is a measure of visibility, and the MBR represents the spatial extent of the node.

3 Analysis: Each "POLYGON" in output represents a geometrical entity (building or part of a building) that has been determined as visible based on the viewpoint and DoV criteria. The polygons are likely the Minimum Bounding Rectangles (MBRs) of these

entities.

4 Visibility Calculation: The visibility of each object is calculated based on its DoV value. Objects with a DoV higher than a certain threshold (η) are included in the search results. This process helps balance visual fidelity and computational performance by fetching objects with higher visibility in greater detail.

5 Search Algorithm: The search algorithm traverses the HDoV-tree. When it encounters a node, it checks the DoV value. If the DoV is below a certain threshold, the branch may not be explored further, reducing the computational load. The algorithm prioritizes nodes closer to the current viewpoint.

6 Storage and Speed Considerations: There are various methods for storing the HDoV-tree, considering the dynamic nature of visibility data. Efficient storage schemes are crucial for quick access and updates as the viewpoint changes. Storage Schemes- Horizontal Storage, Vertical Storage, Indexed Vertical Storage.

7 Logic Relevance: Logic implements the construction and traversal of the HDoV-tree. It includes classes for representing nodes of the tree, storing visibility data, and the logic for traversing the tree based on visibility and LoD criteria.

The output is the result of a visibility query executed on an HDoV-tree, which returns the visible geometrical entities (as polygons) from a certain viewpoint in a virtual environment. The process optimizes performance by considering the DoV of objects and their spatial arrangement.

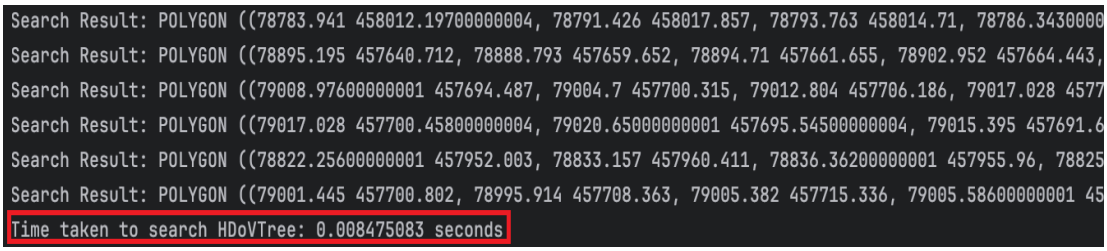


Figure 18: Time taken for Searching HDoV-tree

Horizontal:

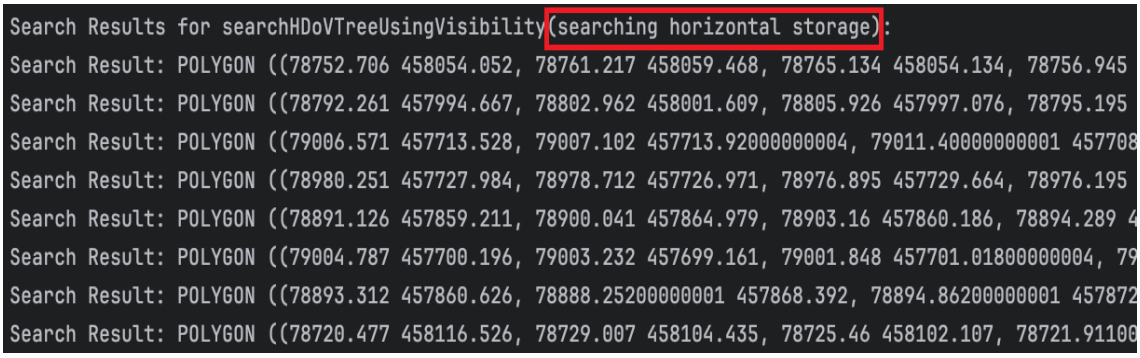


Figure 19: Searching- Horizontal Storage

Logical Structure of HDoV-tree: This is a spatial data structure combining Level of Details (LoDs), spatial indexing, and Degree of Visibility (DoV). It differs from a typical spatial structure because it traverses based on DoV values, and its visibility data is dynamic, changing with the position of viewpoints.

Implementation: The paper discusses implementing this structure with an R-tree for spatial indexing.

Visibility Calculation: The provided Java code calculates visibility based on the HDoV-tree. It uses a grid-based approach where each 3D object is divided into smaller cells (grid), and visibility is calculated for each cell.

Horizontal Storage Scheme: The code includes a horizontal storage scheme to manage visibility data efficiently. This involves storing visibility data (DoV values) for objects in a specialized structure (VPageIndex and VPage) to facilitate fast retrieval.

Search Algorithm: The system supports visibility queries from a point in the virtual environment. It leverages the HDoV-tree's DoV values to determine which objects or object parts are visible from a given viewpoint.

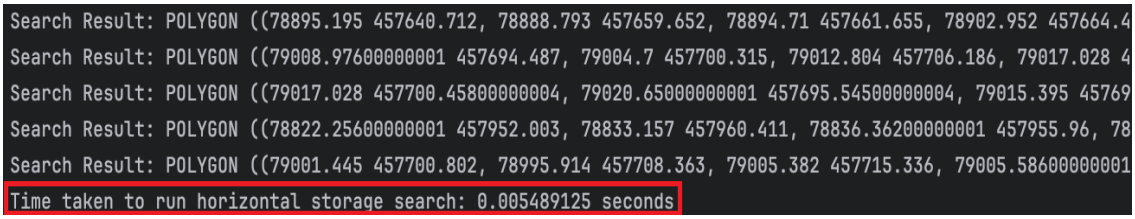


Figure 20: Time taken for Searching- Horizontal Storage

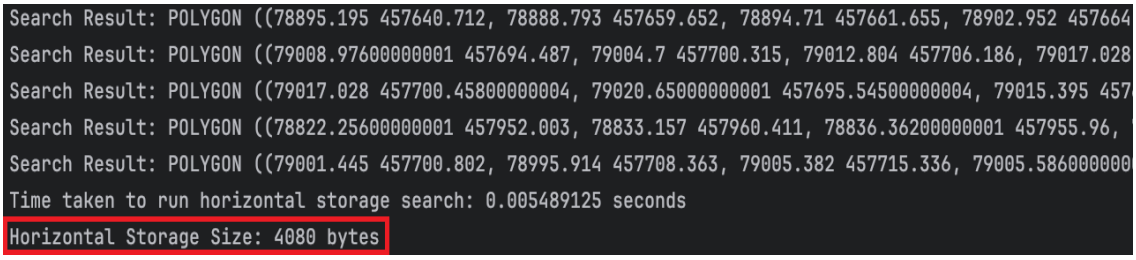


Figure 21: Storage Size- Horizontal Storage

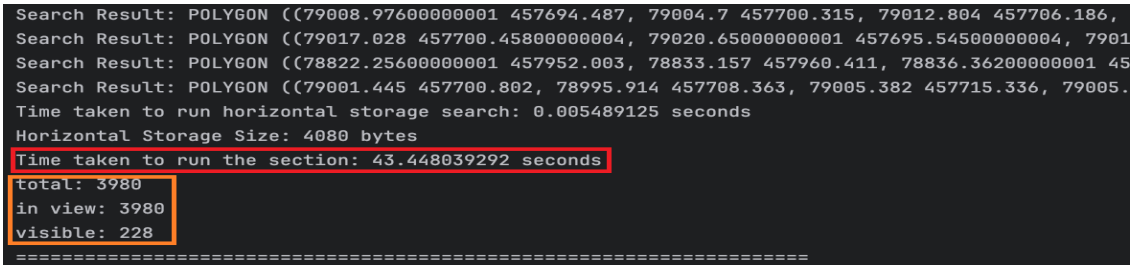


Figure 22: Time taken to run section(right): DoV+HDoV+ Horizontal Storage

Output Explanation:

- **Viewpoint:** Right: Indicates that the current viewpoint or perspective being considered is 'Right'.
- **STR-tree:** Refers to the use of a Spatial R-Tree, which is a data structure used for indexing spatial data. It's used here to efficiently manage and query spatial objects in 3D space.
- **Time Taken:** The computation for this section took 43.448039292 seconds.

- **Total:** 3980 objects were processed. This number represents the total count of geometries processed in this particular run. In our case these are individual buildings.
- **In view:** All 3980 objects are within the view frustum (the visible area from this viewpoint). This could mean that the bounding box or area of interest defined for the calculation encompasses all the geometries. Visibility is likely calculated based on whether the line of sight from the viewpoint to each geometry is obstructed or not.
- **Visible:** Out of the 3980 objects, 228 are visible from this viewpoint, considering obstructions and visibility calculations. Visibility is calculated based on whether the line of sight from the viewpoint to each geometry is obstructed or not.

Code Breakdown:

Classes and Data Structures:

VPage & VPageIndex: VPage stores visibility data (mapping object IDs to DoV values), while VPageIndex is an index structure for VPages. VPage and VPageIndex handle the storage of visibility data, mapping object IDs to their Degree of Visibility (DoV) values.

HDoVTreeNode: Represents nodes in the HDoV-tree. It stores aggregated DoV, geometry (MBR - Minimum Bounding Rectangle), Level of Detail (LoD), and visibility data.

Core Concepts:

Degree of Visibility (DoV): A metric that quantifies how visible an object is from a

certain viewpoint.

Level of Detail (LoD): Determines the detail level at which an object is rendered, based on its distance from the viewpoint and its visibility.

Visibility Calculation: Involves determining which parts of objects are visible from a given viewpoint, considering occlusions and spatial distribution.

Visibility Calculation Process:

i) The code calculates visibility from different viewpoints, considering the spatial distribution of objects and their visibility (DoV).

ii) It uses both the STR-tree (Spatial R-Tree) for efficient spatial querying and the HDoV-tree structure for managing visibility and detail levels.

iii) The process involves creating HDoV-tree nodes, calculating visibility for each object, and aggregating this data in the tree structure.

Search Algorithm:

1. It defines how to traverse the HDoV-tree to determine which objects are visible and at what level of detail they should be rendered.

2. The search algorithm uses the DoV threshold (η) to decide whether to render an object in high detail or to use a coarser representation.

Storage Schemes:

1. The code implements a vertical storage scheme for managing the visibility data efficiently.

2. It employs a VPageIndex to map the visibility data to the corresponding nodes in the

HDoV-tree.

Vertical:

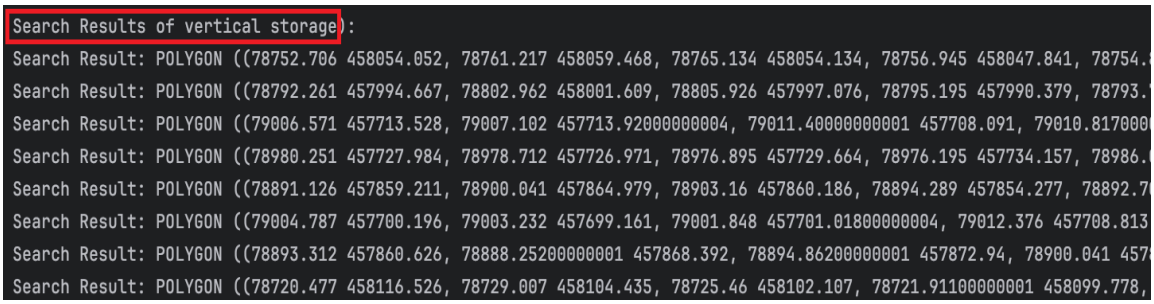


Figure 23: Searching- Vertical Storage

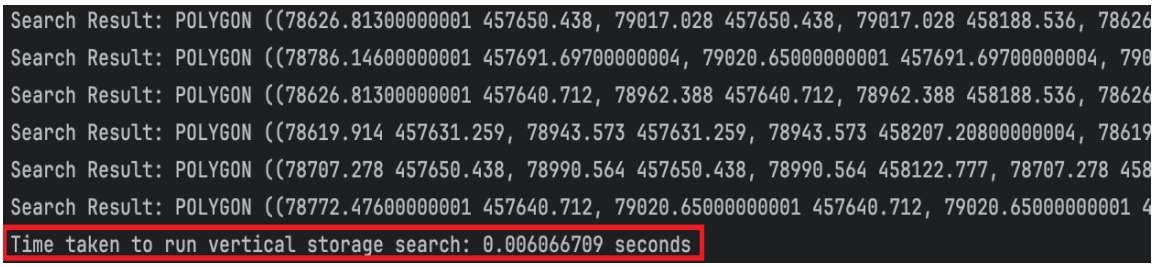


Figure 24: Time taken for Searching- Vertical Storage

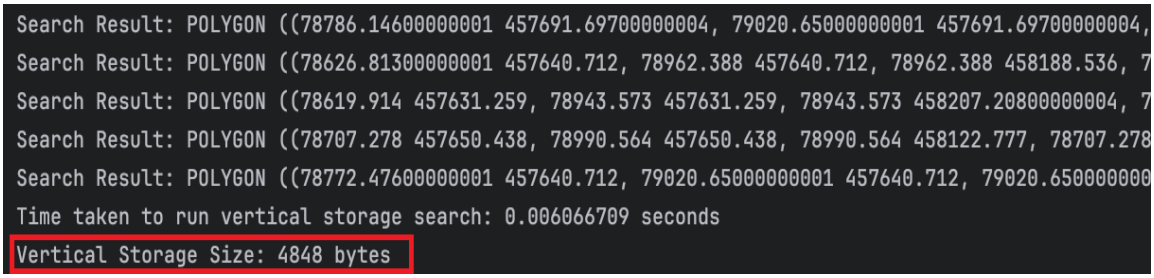


Figure 25: Storage Size- Vertical Storage

Output explanation:

- **HDoV Tree:** It is a spatial data structure that incorporates Level-of-Detail (LoD), spatial indexing, and Degree of-Visibility (DoV). This structure allows for efficient querying of visible objects from a specific viewpoint, making it highly dynamic and viewpoint-dependent.
- **Vertical Storage Scheme (VPage and VPageIndex classes):** This approach optimizes storage and query efficiency. Each node in the HDoV-tree has an associated V-page

(visibility page) that stores visibility data (DoV values) for objects at that node (VPage is like a page in a database storing visibility information of various objects (mapped by their IDs).). The VPageIndex manages these V-pages (VPageIndex is an index to these pages, allowing quick access to the visibility data).

- **Search Algorithm:** It utilizes the HDoV-tree's structure to efficiently query visible objects. The search algorithm takes into account the aggregated DoV of nodes and the threshold η (eta) to determine the level of detail needed for objects or groups of objects. This helps in balancing performance and visual fidelity in the virtual environment.

Output Explanation:

- The output lists several "POLYGON" results. Each POLYGON represents a spatial object's Minimum Bounding Rectangle (MBR) that is deemed visible from the given viewpoint according to the HDoV-tree's structure and the search algorithm's criteria.
- The search algorithm, considering the Degree-of-Visibility (DoV) and the threshold η, has determined these polygons as significant for the query viewpoint. This means they are visible or relevant based on their DoV values compared to the set threshold.
- The **time "0.006066709 seconds"** indicates the efficiency of the vertical storage scheme in retrieving these results. This is a significant aspect as it demonstrates the performance of the system in handling complex spatial queries quickly.
- The **Vertical Storage Size: 4848 bytes** suggests that the vertical storage scheme is being used to manage the visibility data in a compact and efficient manner.

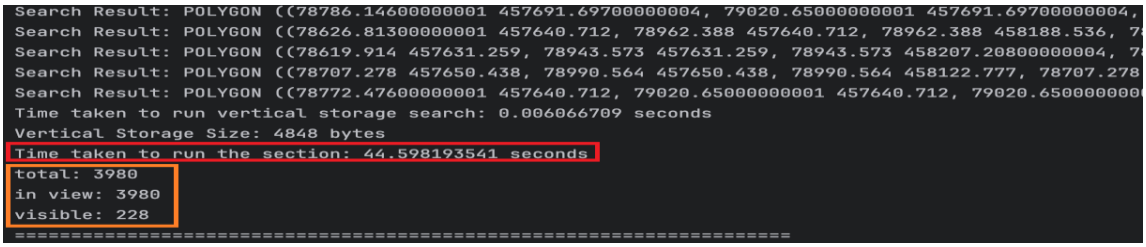


Figure 26: Time taken to run section(right): DoV+HDoV+ Vertical Storage

Output explanation:

Same as Figure 22. Here it is for vertical.

Brute Force:

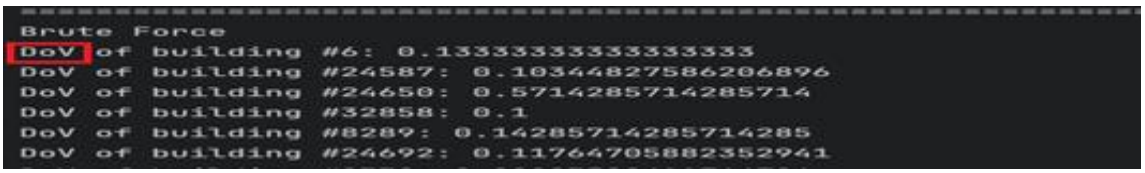


Figure 27: DoV Brute Force

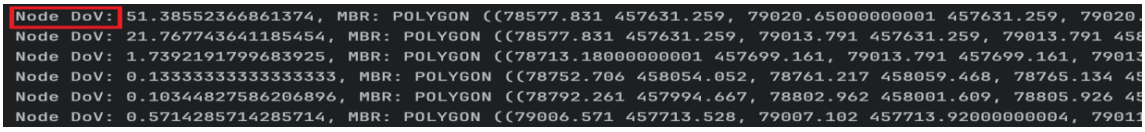


Figure 28:HDOV node DoV, MBR (Minimum Bounding Rectangle), LoD (Level of details) details (Brute force)

Figure 29: HDoV-tree traversal (Brute Force)

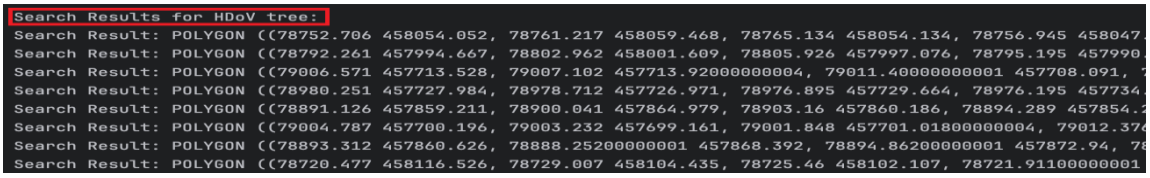


Figure 30: Searching HDoV-tree (Brute Force)

Figure 31: Time taken for Searching HDoV-tree (Brute Force)

Horizontal (Bruteforce):

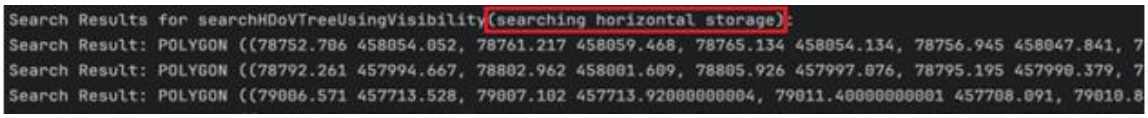


Figure 32:Searching- Horizontal Storage (Brute Force)

Figure 33: Time taken for Searching- Horizontal Storage (Brute Force)

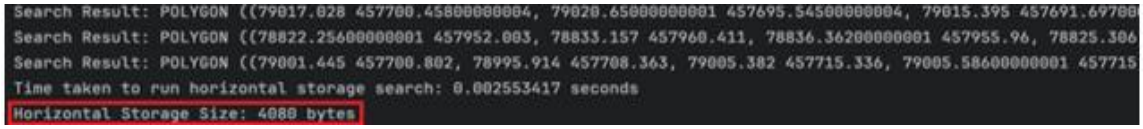


Figure 34: Storage Size- Horizonal Storage (Brute Force)

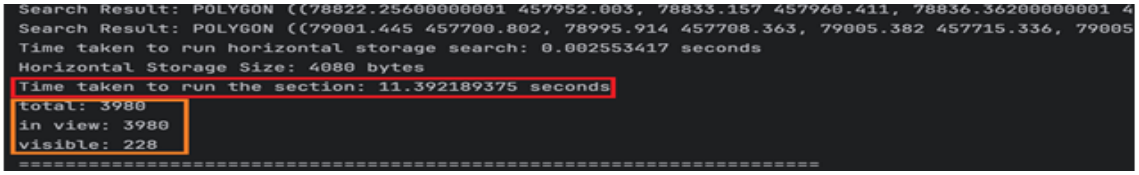


Figure 35: Time taken to run section Brute Force(right): DoV+HDoV+ Horizontal Storage

Vertical (Brute Force):

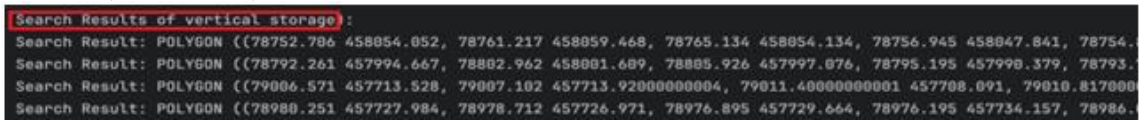


Figure 36:Searching- Vertical Storage (Brute Force)

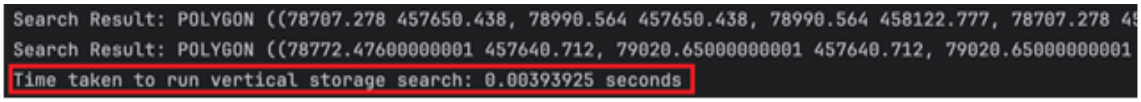


Figure 37:Time taken for Searching- Vertical Storage (Brute Force)

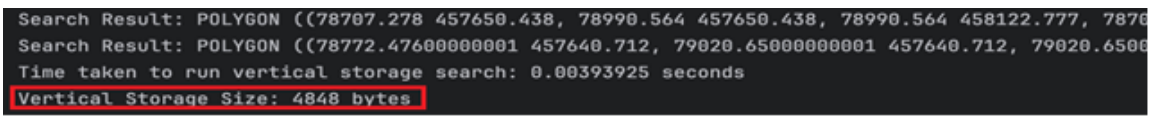


Figure 38: Storage Size- Vertical Storage (Brute Force)

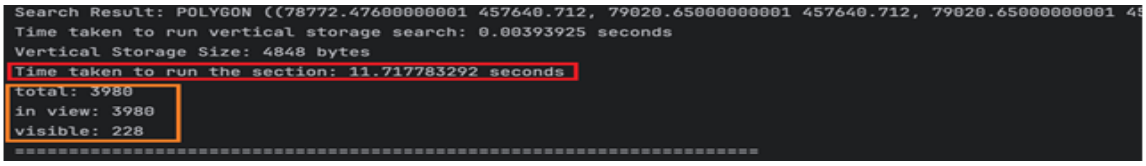


Figure 39:Time taken to run section Brute Force(right): DoV+HDoV+ Vertical Storage

Figure 27-39 explanation:

Same as Figure 14 - 31. Here it is for Brute Force. Brute force has no special tree implementation like STR-tree.

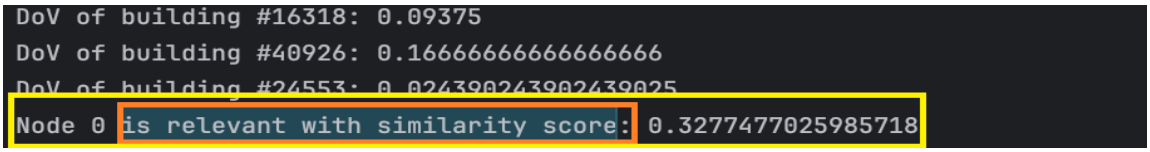


Figure 40: Node which matches the text we entered

Figure 40 explanation:

1. **Computing Similarity Score:** The similarity score is obtained by calling getSemanticSimilarity of the SemanticAnalysis class, which sends a request to an external service (likely an API hosted at "http://localhost:5000/similarity"). The similarity score is a numerical value representing how similar two texts are, on a scale typically from 0 to 1, where 1 means exactly similar. In our case 2 texts are: 1) user input text which is: "POLYGON ((78449.89 457831.617, 78446.79000000001 457834.392, 78447.95300000001 457837.047, 78452.126 457836.689, 78451.009 457834.153, 78449.89 457831.617))" and 2) WKT Texts of nodes with which we are matching it with.
2. **Checking Against Threshold:** If the calculated similarity score exceeds the provided threshold, it means that the node's textual description is sufficiently similar to the user input. In this case, the node is considered "relevant.". Threshold in our case is 0.2.

3. **Output Generation:** When a node's similarity score exceeds the threshold, the code prints the message with the node's ID and its similarity score. In our specific output, "Node 0 is relevant with similarity score: 0.3277477025985718" means that the node with ID 0 has a similarity score of approximately 0.328 with the user-provided WKT string, surpassing the set threshold (0.2 in the current context).

6.2.2 Result Evaluation (Tables):

Table 6: Storage - Execution time (milli seconds)

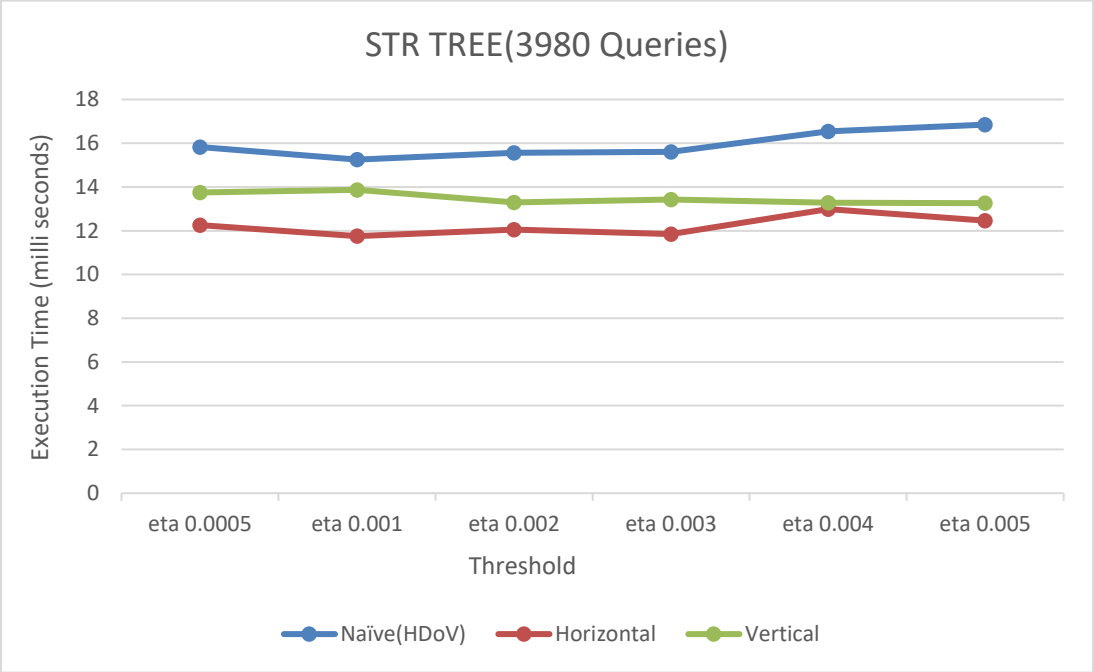


Table 6: It represents and compares the execution times of three storage types: Naïve (HDoV), Horizontal Storage, and Vertical Storage when implemented in STR-tree (R tree) index structure. Although both Horizontal and Vertical Storage outperformed Naïve, ideally, Vertical Storage should exhibit better search performance than Horizontal Storage, which is not observed in this case. This deviation can be attributed to the type of data and the number of queries. In our case, the number of queries was significantly less

than 10,000, totaling only 3,980.

Table 7: Storage - Execution time (milli seconds)

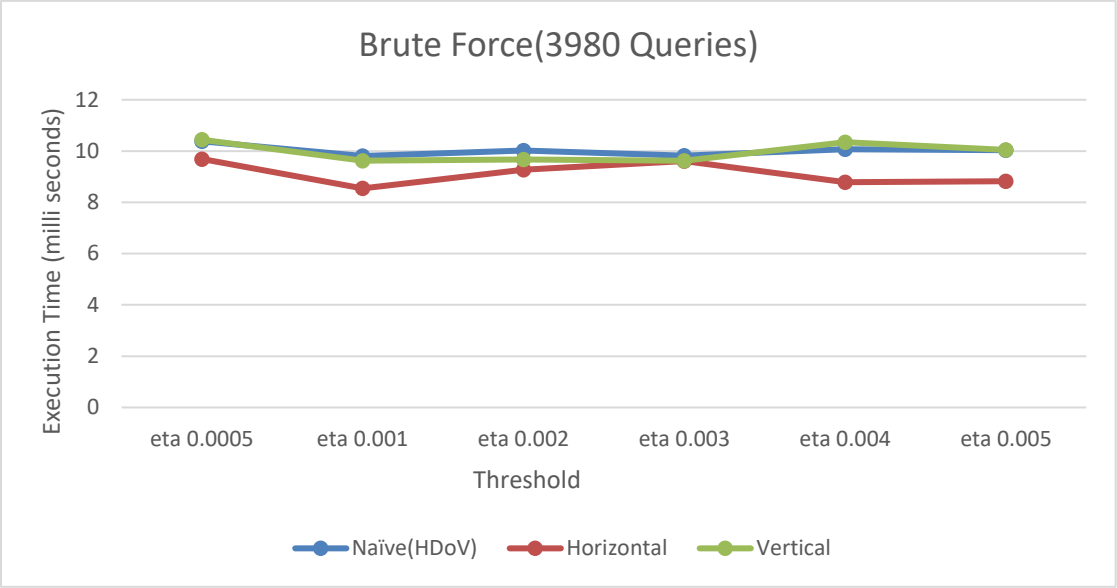


Table 7: It represents and compares the execution times of three storage types: Naïve (HDoV), Horizontal Storage, and Vertical Storage when implemented in Brute Force i.e no index structure. This is the most surprising and un-investigated area since the paper HDoV-tree: the structure, the storage, the speed, talks about using R-tree (STR tree) for implementing HDoV and further Horizontal and Vertical Storage are built on top of HDoV logical structure (L. Shou, 2003) so the results in our case reveals that though Horizontal performed better than vertical and Naïve(HDoV) but still further investigation in type of data and number of queries is required. In our case, the number of queries was significantly less than 10,000, totaling only 3,980. As Authors of HDoV-tree: the structure, the storage, has used 10,000 queries.

Table 8: Storage - size (bytes)

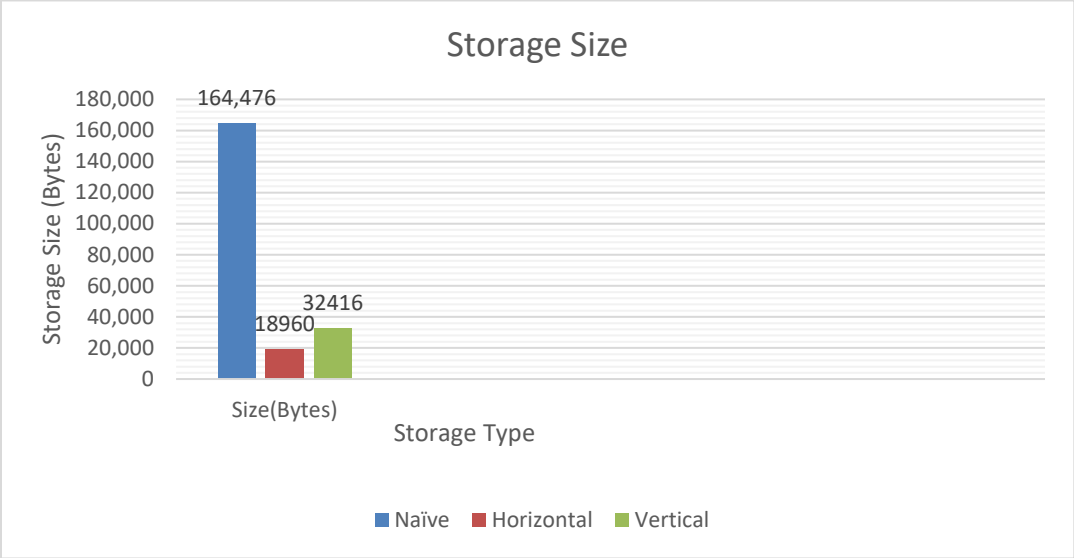


Table 8: It represents and compares the storage space of three storage types: Naïve (HDoV), Horizontal Storage, and Vertical Storage when implemented. Although both Horizontal and Vertical Storage outperformed Naïve, ideally, Vertical Storage should exhibit better storage than Horizontal Storage, which is not observed in this case. This deviation can be attributed to the type of data and the number of queries which can attribute to the way indexes are created.

Chapter 7 CHALLENGES

In this project, a significant challenge encountered was the discrepancy between the 3DCityDB geometries and those supported by JTS. We endeavored to convert POLYHEDRALSURFACE Z geometries into types compatible with JTS, yet this remains a notable constraint when using the JTS library alongside Java for implementation. As a temporary solution, the UnaryUnionOp.union method was employed for combining geometries within a single structure. While this approach may be suitable in some cases, its applicability is limited in complex scenarios, especially at higher Levels of Detail (LOD). Future studies should focus on this issue to ensure the robustness and accuracy of spatial operations in visibility analysis.

Another area requiring improvement is the algorithm used for calculating the Degree of Visibility. The current implementation is not as efficient as desired, evidenced by query execution times being 20 to 50% slower compared to those obtained using the brute force method. To enhance the overall process's performance, exploring more effective techniques for determining the Domain of Validity is crucial. A promising direction could be to explore the work of Shou and colleagues, who proposed a more detailed approach for computing the DoV. By incorporating their insights and refining the algorithm, the efficiency and reliability of the visibility analysis could be significantly improved. This enhancement would enable more accurate assessments of urban landscapes and their impact on visibility.

Chapter 8 CONCLUSION

We successfully developed a Java application designed to manage the spatial index of a real 3D city model, focusing specifically on the city of Den Haag. This application harnesses the power of 3DCityDB, a robust database system compatible with PostgreSQL databases, to process 3D urban model data. A key part of their research involved comparing two methods for calculating the Degree of Visibility (DoV) in urban models: one using the STR-tree spatial index and the other employing a brute force technique.

Our analysis revealed that the brute force method demands significantly more processing time compared to the STR-tree method. This performance discrepancy is influenced by various factors, including inefficiencies in the DoV calculation algorithm and differences in the node insertion process within the STR-tree structure.

The inefficiency in the DoV algorithm might stem from its design or implementation, leading to unnecessary calculations or repetitive processes. Optimizing this algorithm could substantially improve system performance. We are considering exploring alternative DoV computation methods that might be more efficient.

Regarding the STR-tree structure, the manner in which nodes are inserted can adversely affect query performance due to potential imbalances. We propose investigating different strategies for maintaining balance in the tree during the insertion phase or

considering other spatial indexing structures better suited to their specific needs. The unique characteristics of the 3D urban model data should be taken into account, as they influence the effectiveness of the chosen spatial index method.

We compared and found that the type of data matters a lot and the number of visibility queries on which paper HDoV (L. Shou, 2003) has been tested i.e 10,000 visibility queries makes a lot of difference, because in our case the data on which we tested had 3980 visibility queries and that is the reason we are able to conclude that min number of visibility queries(10,000) and type of spatial data makes a lot of sense and hence the data storage schemes and time taken to search over them significantly changes (180 degree shift), i.e against the norm the horizontal scheme performs better than vertical one.

Further research is needed to develop a more efficient algorithm for managing spatial indices in 3D immersive systems. This involves a deeper analysis of the current algorithm's performance, exploring other algorithms and data structures, and experimenting with various techniques to balance the spatial index tree. Additionally, we must navigate the specific challenges of working with real 3D urban models, such as varying complexity levels, data diversity, and the complexities of urban environments. Addressing these factors will help us refine the approach to spatial index management in 3D immersive settings.

We implemented the article Semantic Textual similarity with BERT (Winastwan, 2024)

but the results are not satisfactory, the WKT data which we are working on for textual similarity are not yielding better results. We may need to investigate and try to implement spatial similarity instead.

We underscore the significant business implications and transformative potential of our spatial-temporal indexing research, anchored in the Hierarchical Degree-of-Visibility (HDoV) tree structure and augmented by Semantic Textual Similarity. These innovations present a multifaceted toolkit for several key industries.

Archaeology and First Response Operations: The implementation of real-time maps, powered by our indexing method, can revolutionize the field of archaeology and the operations of first responders. In archaeology, this technology allows for a dynamic, nuanced examination of historical sites, enhancing the visibility and analysis of spatial-temporal data. For first responders, such as firefighters and paramedics, our approach significantly elevates situational awareness and operational effectiveness, enabling faster, more informed decision-making in crisis situations.

Virtual Real Estate: In the virtual real estate sector, the application of our spatial indexing combined with semantic similarity search ushers in a new era of remote property exploration and marketing. Potential buyers and tenants can enjoy highly intuitive, immersive experiences, exploring properties in fine detail from anywhere in the world. This digital transformation not only elevates the customer experience but also opens new

avenues for real estate marketing and sales.

Gaming Industry: The gaming industry stands to benefit immensely from our research. By integrating enhanced spatial indexing and semantic understanding, game developers can create more realistic, engaging virtual worlds. This not only improves the player experience but also opens up new possibilities for game design, potentially boosting user retention and revenue. Furthermore, the inclusion of semantic similarity search widens the scope for more intuitive and contextually relevant interactions within these virtual environments.

Chapter 9 Further innovation, implementation and refinement

1. Further refinement and improvement is required for Vertical and Horizontal storage code.
2. Paper: “Efficient Semantic Similarity Search over Spatio-textual Data.", can be further implemented over the existing code of paper HDoV-tree: the structure, the storage, the speed, which will then can show Semantic similarity over spatio data which we are using to implement HDoV paper over DoV from data set of paper: 3D City Database. (L. Shou, 2003) (George S. Theodoropoulos, 2023)
3. Hybrid indexing of paper “Efficient Semantic Similarity Search over Spatio-textual Data” could potentially incorporate visibility aspects from “HDoV-tree: the structure, the storage, the speed” to enhance its functionality. (L. Shou, 2003) (George S. Theodoropoulos, 2023)
4. Paper “HDoV-tree: the structure, the storage, the speed” focus on dynamic visibility data based on viewer positions can be an inspiration for handling dynamic textual data in Paper “Efficient Semantic Similarity Search over Spatio-textual Data”. This can be especially relevant if textual data changes over time or based on user context. (L. Shou, 2003) (George S. Theodoropoulos, 2023)
5. Paper “HDoV-tree: the structure, the storage, the speed” strategy for loading data based on DoV values could inform how Paper “Efficient Semantic Similarity Search over Spatio-textual Data” manages the loading of spatio-textual data, especially in scenarios where only a subset of the data is relevant to a query. (L. Shou, 2003) (George S. Theodoropoulos, 2023)

6. The primary challenge would be in reconciling the different focuses of the two papers. Paper “HDoV-tree: the structure, the storage, the speed” is more about visual data and its visibility, while Paper “Efficient Semantic Similarity Search over Spatio-textual Data” is about textual and spatial data. Integrating these aspects would require a careful consideration of how visibility factors into semantic and spatial relevance. (L. Shou, 2003) (George S. Theodoropoulos, 2023)
7. Further refinement is needed in the code of the article 'Semantic Textual Similarity with BERT', as well as in its results, to enhance the search capabilities of HDoV, Horizontal, and Vertical Storage. This refinement is essential to fully realize the potential of these storage methods.
8. A further challenge involves investigating the viability of two approaches on our current dataset: textual similarity and spatial similarity. Notably, the implementation of textual similarity (Winastwan, 2024) has not yielded viable results in our case, prompting the need for this exploration.

Bibliography

Abbas Al-Ghaili, H. K.-H. (2022). A Review of Metaverse's Definitions, Architecture, Applications, Challenges, Issues, Solutions, and Future Trends. IEEE.

Beng Chin Ooi, K. L. (2010). Sense The Physical, Walkthrough The Virtual, Manage The Co (existing) Spaces: A Database Perspective. ACM SIGMOD .

Dejun Teng, Y. L. (2022). 3DPro: Querying Complex Three-Dimensional Data with Progressive Compression and Refinement. National library of medicine.

geodb/ db3dcore. (2023, August). Retrieved from https://github.com/geodb/db3dcore

George S. Theodoropoulos, K. N. (2023). Efficient Semantic Similarity Search over Spatio-textual Data. OpenProceedings.

Hugo Ledoux, K. A. (2019). CityJSON: a compact and easy-to-use encoding of the CityGML data model. Springer open.

Huzaifa, M., Desai, R., Grayson, S., Jiang, X., Jing, Y., Lee, J., . . . Adve, S. V. (2021). ILLIXR: Enabling End-to-End Extended Reality Research. IEEE.

ISRO. (2023, july). *Satellite Navigation Services*. Retrieved from Navigation with Indian Constellation (NavIC): https://www.isro.gov.in/SatelliteNavigationServices.html

Karin Staring, S. V. (2020). CityJSON in combination with MongoDB, PostgreSQL and GraphQL. *Delft University of Technology*.

Katerina Ruzickova, J. R. (2021). A new GIS-compatible methodology for visibility analysis in digital surface models of earth sites. Geoscience Frontiers.

L. Shou, Z. H.-L. (2003). HDoV-tree: the structure, the storage, the speed. *IEEE*.

Mikael Johansson, L. H. (2002). Using Java Topology Suite for real-time data generalisation and integration. *National Land Survey of Sweden*.

Rajput, A. (2024, February). *Semantic search using NLP*. Retrieved from Semantic search using NLP An starter guide on building intelligent search engine using semantic understanding of search queries: https://medium.com/analytics-vidhya/semantic-search-engine-using-nlp-cec19e8cfa7e

Rouault, E. (2014, April 6). *Geo tips & tricks(OSGeo, GDAL and other GIS fun.)*. Retrieved from https://erouault.blogspot.com/2014/04/gml-madness.html

Tan, K.-L. (2022). DATA MANAGEMENT FOR THE METAVERSE. *ACM SIGMOD Blog*.

Winastwan, R. (2024, March). *Semantic Textual Similarity with BERT*. Retrieved from

Semantic Textual Similarity with BERT How to use BERT to calculate the semantic similarity between two texts: https://towardsdatascience.com/semantic-textual-similarity-with-bert-fc800656e7a3

Yusuf Sermet, I. D. (2021). GeospatialVR: A Web-based Virtual Reality Framework for Collaborative Environmental Simulations. ELSEVIER.

Zhihang Yao, C. N. (2018). 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards.* Springer Open.

Zhihang Yao, C. N. (latest version-2023.0). *3dcitydb-docs.* Retrieved from https://3dcitydb-docs.readthedocs.io/en/latest/

Curriculum Vitae

Vidit Sharma

University Attended:

- Master of Computer Science, Guru Gobind Singh Indraprastha University, 2010
- Master of Business Administration, University of New Brunswick, Canada, Expected May 2024

Research Papers:

- Data Systems for 3D Geospatial Analytics in Collaborative Extended Reality (XR), March 2024

Review Papers:

- Self-Supervised learning and SimCLR V2: A Review, December 2022

Articles:

- Advantages and Disadvantages of performance metrices in the different learning approaches of machine learning, April 2023

Critiques:

- Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins, March 2023.
- The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins, March 2023.
- An Intermediate Representation for Optimizing Machine Learning Pipelines, April 2023

Honours:

- Dr. Fransisco Arcelus Memorial Scholarship, December 2023.
- Dean’s List, March 2024.

Professional Experience:

- Acumen Infinite Solutions: Developer Front End Technologies. Jan 2020 – Aug 2021.
- IBS: Routing and Switching. Jan 2017 – Sep 2019.
- Aricent CISCO: Routing and Switching Protocols. May 2015 – Jul 2016.
- Digital Power: Routing and Switching. Apr 2014 – May 2015.
- Adobe Systems: Consultant, Flash Media Servers/Adobe Media servers (RTMP, RTMFP protocols). Nov 2012 – May 2013.
- Wipro Technologies: Protocol Developer – Ipsec and routing protocols, Ericsson ISER(Integrated Sited edge Router). Feb 2011 – Nov 2012.