FACULTY OF ENGINEERING AND ARCHITECTURE

# Pre-culling geometric linked building data for lightweight viewers

Philippe Soubrier
Student number: 01702837

Supervisor: Prof. ir.-arch. Paulus Present
Counsellors: Ir.-arch. Jeroen Werbrouck, Prof. dr. ir. arch. Ruben Verstraeten

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de ingenieurswetenschappen: architectuur

Academic year 2022-2023

GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

# Pre-culling geometric linked building data for lightweight viewers

## Linked Data

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de ingenieurswetenschappen: architectuur

Supervisor: Prof. ir.-arch. Paulus Present

Counselors: Ir.-arch. Jeroen Werbrouck
Prof. dr. ir. arch. Ruben Verstraeten

Philippe Soubrier 01702837 philippe.soubrier@ugent.be
Academic year: 2022–2023

GHENT UNIVERSITY

Deze masterproef vormt een onderdeel van een examen. Eventuele opmerkingen die door de beoordelingscommissie tijdens de mondelinge uiteenzetting van de masterproef werden geformuleerd, werden niet verwerkt in deze tekst.

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

# Preface

This thesis is the result of research about Augmented Reality (AR) in the Architecture, Engineering and Construction (AEC) sector and the use of linked data to improve the performance of AR applications. It quickly evolved into broader research about lightweight viewers with the same need for performance improvements. It would not have been possible without the help and guidance of several people, whom I would like to thank.

Firstly, I would like to thank my supervisor Paulus Present and counselors Jeroen Werbrouck and Prof. Ruben Verstraeten for their guidance, feedback, and discussions we had throughout the entire process. In particular, I would like to thank Jeroen Werbrouck for his previous courses that sparked my interest in the domain of coding and web development and for his valuable feedback on the technical aspects of this thesis.

Secondly, I would like to thank Prof. Tiemen Strobbe for taking the time to explain the workings of the Qonic viewer, which inspired the developed process of this thesis. And finally, my friends Hanne den Biesen and Bram van Rooijen for thoroughly proofreading this thesis. Their feedback and comments were invaluable and I am very grateful for their help.

# Pre-culling geometric linked building data for lightweight viewers

Philippe Soubrier

Supervisor: Prof. ir.-arch. Paulus Present
Counselors: Ir.-arch. Jeroen Werbrouck
Prof. dr. ir. arch. Ruben Verstraeten

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de ingenieurswetenschappen: architectuur

Academic year 2022-2023

## Abstract

A significant interaction with a building's digital twin in the Architecture, Engineering and Construction (AEC) sector is the 3D visualisation of a Building Information Modelling (BIM) model. As the industry evolves in this digital era, the increasing amount of digital data exponentially amplifies the complexity of a building's semantic and geometrical definition. This geometric complexity poses a challenge for the visualisation of BIM models on newer, resource-limited devices used in the industry, such as smartphones and tablets.

Proposing a solution to this issue, this thesis introduces the concept of initial filtering, or pre-culling in computer graphics terminology, the geometrical data of a BIM model on the device where it is stored before transmitting it to the resource-limited device. Thus leaving this device with a minimal, relevant 3D scene confined to its view frustum.

The focus of this thesis is placed on the utilization of Semantic Web technologies,such as the Resource Description Framework (RDF) and the SPARQL Protocol and RDF Query Language (SPARQL), to store and query BIM models as databases. This enables the datasource to be divided into smaller, more manageable chunks containing only the data required for the visualisation purpose.

This concept is brought to life through the proposal of a web-based 3D viewer prototype together with its modular and extendable structure. This enables not only the demonstration of the pre-culling concept, but also establishes a robust groundwork for further research and alternative implementations.

*Keywords* - **Linked Data, lightweight viewers, culling, BIM, SPARQL**

# Extended Abstract


## Philippe Soubrier

Supervisor: Prof. ir.-arch. Paulus Present

Counselors: Ir.-arch. Jeroen Werbrouck, Prof. dr. ir. arch. Ruben Verstraeten



*Abstract* - **This thesis tackles the issue of visualizing complex Building Information Modelling (BIM) models on resource-limited devices in the Architecture, Engineering and Construction (AEC) sector, introducing "pre-culling" to reduce the 3D scene to the viewer's view frustum. By utilizing Semantic Web technologies like the Resource Description Framework (RDF) and the SPARQL Protocol and RDF Query Language (SPARQL), BIM models are stored and queried as databases, thereby facilitating their segmentation.**



*Keywords* - **Linked Data, lightweight viewers, culling, BIM, SPARQL**


The AEC industry has undergone significant technological transformations over the past few decades. The advent of 3D modeling constituted a major shift in the industry, creating a new era of geometric data representation. The subsequent introduction of BIM has functioned as a versatile repository for semantics derived from various applications throughout the design and construction processes, including but not limited to cost estimation, energy analysis, and production planning. However, as Werbrouck, 2018 highlighted, the next challenge for the AEC industry is related to the domain-specific nature of current BIM software solutions, which remain inaccessible to other disciplines. This data **management** issue is currently being tackled by entities such as the Linked Building Data Community Group (LBD-CG) and academic institutions like the University of Ghent, leveraging the potential of Semantic Web technologies for more integrated and efficient data handling, thereby enabling interdisciplinary collaboration. The term Linked Data BIM (LDBIM) is used in this thesis to denote this emerging milestone.

Alongside this evolution, in tandem with the increasing amount of semantic data, the geometric data describing a BIM model is also growing in size and complexity. This makes the task of visualization increasingly complex on newer, resource-limited devices used in the industry. Such devices, for example, smartphones and tablets, are becoming increasingly popular on construction sites. Existing solutions use a filtering step in the visualization process, known as culling. This computational step necessitates the processing of the entire 3D file or scene to cull it to the viewer's view frustum (Johansson et al., 2015). This operation produces a minimal 3D scene, which is then utilized for the significantly more resource-intensive process of rendering.

A bottleneck arises when the initial 3D scene, upon which the in-viewer culling process is performed, becomes too large for the device to handle. As a solution to this problem, this thesis proposes the use of a **pre-culling** step. In this process, and with the aid of Semantic Web technologies, a BIM model is stored and queried as a partitioned database. By segregating the storage of the BIM model and thus offloading the computationally intensive task of culling to a storage server in a preliminary culling step, the amount of data that needs to be processed by the visualization device is minimized.

This thesis will therefore investigate the extent to which LDBIM geometry can be pre-culled. It will examine the minimum size of geometry that can be defined in a RDF database. For this purpose, existing ontologies in the field of AEC or related fields such as Geographic Information System (GIS) will be explored. These ontologies will also be scrutinized for their potential utilization within culling algorithms, similar to the work of Johansson and Roupé, 2009. In their paper, they exploited the semantics of a BIM model in an Industry Foundation Classes (IFC) format to feed culling algorithms. As

BIM models possess an inherent underlying hierarchy of Bounding Volume Hierarchy (BVH), these can be employed in modern in-viewer culling algorithms such as the Coherent Hierarchical Culling algorithm (CHC)++ (Johansson et al., 2015).

## Outline

**Chapter 1 -** This introduction chapter presents the motivation for this thesis and the research questions. It also introduces the concept of culling and provides an overview of the structure of the thesis.

**Chapter 2 -** As an introduction to the Semantic Web, this chapter presents the Linked Data principles used in this thesis. Furthermore, it explains the complexity and size of BIM models within Semantic Web databases, which are represented as RDF graphs.

**Chapter 3 -** Johansson et al., 2015 pointed out that there is a scarcity of research exploring the performance of current BIM viewers. Thus, this state-of-the-art research aims to concentrate on the overall features of some promising newer viewers and the ontologies/tools that will be used in this thesis.

**Chapter 4 -** The concept of dynamic queries is introduced in this chapter. These represent the automatic generation of queries responsible for feeding culling algorithms, to obtain the data needed to visualize the scene visible to the viewer. Therefore, firstly, the needs of an LDBIM viewer are explored, which lies at the base of the process. Secondly, three approaches where the culling action is performed are presented: in the query, in the viewer, and in situ. Each approach showcases the possibilities of culling by constructing SPARQL queries.

**Chapter 5 -** In addition to the actual culling, this chapter proposes a modular approach to implement the culling process in an LDBIM viewer. This robust foundation serves as a framework for future research. Informed by this thesis's prototype, it identifies the different components needed to create an LDBIM viewer and integrate the culling process into it. It dissects the viewer's framework into four main components: the viewer itself, the cache manager, the query processor, and the data fetcher.

**Chapter 6 -** To demonstrate the feasibility of the proposed culling process, a prototype is developed in this chapter. This prototype is a proof of concept that implements the modular approach presented in the previous chapter within a web-based viewer. It showcases the feasibility of the proposed dynamic queries and also demonstrates the potential for extension to implement the semantic visualisation of related semantics in the RDF database.

## Conclusion

**Chapter 7 -** This thesis has shown that culling LDBIM graphs to reduce the size of the scene a lightweight viewer has to manage in order to visualise a building model stored in a database using Semantic Web technologies is possible. It also presented multiple approaches to perform the culling, each with its own advantages and disadvantages, as well as a modular approach to implement the whole process in a web viewer. This technology has proven to be a viable solution to the demanding needs of the AEC industry when visualizing large BIM models. And it presents a strong foundation to expand upon in a diverse set of use-cases and scenarios in need of a 3D visual representation.

## References


Johansson, M., & Roupé, M. (2009). *Efficient real-time rendering of building information models.* https://www.researchgate.net/publication/220758081

Johansson, M., Roupé, M., & Bosch-Sijtsema, P. (2015). Real-time visualization of building information models (bim). *Automation in Construction, 54*, 69–82. https://doi.org/10.1016/j.autcon.2015.03.018

Werbrouck, J. (2018). Linking data : Semantic enrichment of the existing building geometry. http://lib.ugent.be/catalog/rug01:002494740


**Demonstration:**
https://github.com/flol3622/Pre-culling_LDBIM#demo
**Prototype:**
https://github.com/flol3622/LDBIM-viewer

# Uitgebreid Abstract


## Philippe Soubrier

Promotor: Prof. ir.-arch. Paulus Present

Begeleiders: Ir.-arch. Jeroen Werbrouck, Prof. dr. ir. arch. Ruben Verstraeten



*Abstract* - **Deze thesis pakt het probleem aan van het visualiseren van complexe Building Information Modelling (BIM) modellen op apparaten met beperkte vermogen in de Architecture, Engineering and Construction (AEC) sector, waarbij "pre-culling" wordt geïntroduceerd om de 3D-scène te reduceren tot het zichtveld van de kijker. Door gebruik te maken van Semantic Web technologieën zoals Resource Description Framework (RDF) en SPARQL Protocol and RDF Query Language (SPARQL), worden BIM modellen opgeslagen en opgevraagd als databases, wat hun segmentatie vergemakkelijkt.**



*Trefwoorden* - **Linked Data, minimale viewers, culling, BIM, SPARQL**


De AEC industrie heeft de afgelopen decennia belangrijke technologische transformaties ondergaan. De komst van 3D-modellering heeft een grote verschuiving veroorzaakt in de industrie, waardoor een nieuw tijdperk van geometrische gegevensrepresentatie is begonnen. De daaropvolgende introductie van BIM heeft gewerkt als een veelzijdige opslagplaats voor semantiek afkomstig van verschillende toepassingen gedurende het ontwerp- en bouwproces, inclusief maar niet beperkt tot kostenraming, energieanalyse en productieplanning. Echter, zoals Werbrouck, 2018 benadrukte, ligt de volgende uitdaging voor de AEC industrie in de domeinspecifieke aard van de huidige BIM softwareoplossingen, die ontoegankelijk blijven voor andere disciplines. Dit gegevensbeheerprobleem wordt momenteel aangepakt door de Linked Building Data Community Group (LBD-CG) en academische instellingen zoals de Universiteit van Gent, die het potentieel van *Semantic Web*-technologieën benutten voor een meer geïntegreerde en efficiënte gegevensbehandeling, waardoor interdisciplinaire samenwerking mogelijk wordt. De term Linked Data BIM (LDBIM) wordt in deze thesis gebruikt om deze opkomende mijlpaal aan te duiden.

Naast deze evolutie, en gelijktijdig met de toenemende hoeveelheid semantische gegevens, groeit ook de geometrische data die een BIM model beschrijft in grootte en complexiteit. Dit maakt de taak van visualisatie steeds complexer op nieuwere apparaten met beperkte vermogen, die in de industrie worden gebruikt. Dergelijke apparaten, bijvoorbeeld smartphones en tablets, worden steeds populairder op bouwplaatsen. Bestaande oplossingen gebruiken een filteringstap in het visualisatieproces, bekend als *culling*. Deze computationele stap vereist de verwerking van het volledige 3D-bestand of de scène om het te reduceren tot het zichtveld van de kijker (Johansson et al., 2015). Deze operatie produceert een minimale 3D-scène, die vervolgens wordt gebruikt voor het aanzienlijk meer rekenintensieve renderproces.

Er ontstaat een probleem wanneer de initiële 3D-scène, waarop het *in-viewer culling* proces wordt uitgevoerd, te groot wordt voor het apparaat om te verwerken. Als oplossing voor dit probleem stelt deze thesis het gebruik van een **pre-culling** stap voor. In dit proces, en met de hulp van *Semantic Web* technologieën, wordt een BIM model opgeslagen en opgevraagd als een gepartitioneerde database. Door de opslag van het BIM model te scheiden en zo de rekenintensieve taak van *culling* naar een opslagserver te verplaatsen in een voorlopige *culling* stap, wordt de hoeveelheid data die moet worden verwerkt door het visualisatieapparaat geminimaliseerd.

Deze thesis zal daarom onderzoeken in hoeverre de geometrie van LDBIM op voorhand kan worden geculled. Het zal onderzoeken wat de minimale grootte van geometrie kan zijn die kan worden gedefinieerd in een RDF database. Voor dit doel

zullen bestaande ontologieën van de AEC industrie of gerelateerde industrieën zoals Geographic Information System (GIS) worden verkend. Deze ontologieën zullen ook worden onderzocht voor hun potentiële gebruik binnen *culling* algoritmen, vergelijkbaar met het werk van Johansson and Roupé, 2009. In hun paper maakten ze gebruik van de semantiek van een BIM model in een Industry Foundation Classes (IFC) formaat om *culling* algoritmen te voeden. Aangezien BIM modellen een inherent onderliggende hiërarchie van Bounding Volume Hierarchy (BVH) bevatten, kunnen deze worden gebruikt in moderne *in-viewer culling* algoritmen zoals de Coherent Hierarchical Culling algorithm (CHC)++ (Johansson et al., 2015).

## Overzicht

**Hoofdstuk 1 -** Dit inleidende hoofdstuk presenteert de motivatie voor deze thesis en de onderzoeksvraagen. Het introduceert ook het concept van *culling* en geeft een overzicht van de structuur van de thesis.

**Hoofdstuk 2 -** Als inleiding tot het *Semantic Web* presenteert dit hoofdstuk de Linked Data principes die in deze thesis worden gebruikt. Verder legt het de complexiteit en grootte uit van BIM modellen binnen *Semantic Web* databases (RDF).

**Hoofdstuk 3 -** Johansson et al., 2015 vermeld in zijn paper dat er een gebrek is aan onderzoek naar de prestaties van huidige BIM viewers. Daarom richt dit state-of-the-art onderzoek zich op de algemene kenmerken van enkele veelbelovende nieuwe viewers en de ontologieën/tools die in deze thesis worden gebruikt.

**Hoofdstuk 4 -** Het concept van dynamische queries wordt in dit hoofdstuk geïntroduceerd. Deze representeren de automatische generatie van queries die verantwoordelijk zijn voor het voeden van *culling* algoritmen, om de data te verkrijgen die nodig is om de scène die voor de kijker zichtbaar is te visualiseren. Daarom worden eerst de behoeften van een LDBIM viewer onderzocht, die aan de basis van het proces ligt. Ten tweede worden drie benaderingen gepresenteerd waar de *culling* actie wordt uitgevoerd: in de query, in de viewer, en in situ. Elke benadering toont de mogelijkheden van *culling* door het construeren van SPARQL queries.

**Hoofdstuk 5 -** Naast de daadwerkelijke *culling*, stelt dit hoofdstuk een modulaire benadering voor om het *culling* proces te implementeren in een LDBIM viewer. Deze robuuste basis dient als een *framework* voor toekomstig onderzoek. Gebaseerd op het prototype van deze thesis, identificeert het de verschillende componenten die nodig zijn om een LDBIM viewer te creëren en het *culling* proces daarin te integreren. Het ontleedt het *framework* van de viewer in vier hoofdcomponenten: de viewer zelf, de cache manager, de query processor, en de data fetcher.

**Hoofdstuk 6 -** Om de haalbaarheid van het voorgestelde *culling* proces te demonstreren, wordt in dit hoofdstuk een prototype ontwikkeld. Dit prototype is een proof of concept dat de modulaire aanpak die in het vorige hoofdstuk is gepresenteerd, implementeert binnen een web-based viewer. Het toont de haalbaarheid van de voorgestelde dynamische queries aan en demonstreert ook de potentie voor uitbreiding om de semantische visualisatie van gerelateerde semantiek in de RDF database te implementeren.

## Conclusie

**Hoofdstuk 7 -** Deze thesis heeft aangetoond dat het mogelijk is om LDBIM grafieken te snoeien om de grootte van de scène die een lichte viewer moet beheren te verminderen, om zo een gebouwmodel opgeslagen in een database met behulp van *Semantic Web* technologieën te visualiseren. Het presenteerde ook meerdere benaderingen om het snoeien uit te voeren, elk met zijn eigen voordelen en nadelen, evenals een modulaire benadering om het gehele proces in een web viewer te implementeren. Deze technologie heeft bewezen een haalbare oplossing te zijn voor de veeleisende behoeften van de AEC industrie bij het visualiseren van grote BIM modellen. En het presenteert een sterke basis om uit te breiden op een diverse set van use-cases en scenario's die behoefte hebben aan een 3D visuele representatie.

## Referenties

Johansson, M., & Roupé, M. (2009). *Efficient real-time rendering of building information models*. https://www.researchgate.net/publication/220758081

Johansson, M., Roupé, M., & Bosch-Sijtsema, P. (2015). Real-time visualization of building information models (bim). *Automation in Construction, 54*, 69–82. https://doi.org/10.1016/j.autcon.2015.03.018

Werbrouck, J. (2018). Linking data : Semantic enrichment of the existing building geometry. http://lib.ugent.be/catalog/rug01:002494740

Demonstratie: https://github.com/flol3622/Pre-culling_LDBIM#demo

Prototype: https://github.com/flol3622/LDBIM-viewer

# Résumé détaillé


## Philippe Soubrier

Promoteur : Prof. Ing.-arch. Paulus Present Superviseurs : Ing.-arch. Jeroen Werbrouck, Prof. dr. Ing. arch. Ruben Verstraeten



*Résumé* - **Cette thèse aborde le problème de la visualisation de modèles Building Information Modelling (BIM) complexes sur des appareils à capacités limitées dans le secteur de l'Architecture, Engineering and Construction (AEC), en introduisant le "pre-culling" pour réduire la scène 3D au champ de vision de l'observateur. En utilisant des technologies du Semantic Web comme Resource Description Framework (RDF) et SPARQL Protocol and RDF Query Language (SPARQL), les modèles BIM sont stockés et demandés en tant que bases de données, ce qui facilite leur segmentation.**



*Mots clés* - **Linked Data, visualiseurs minimalistes, culling, BIM, SPARQL**


L'industrie de l'AEC a subi d'importantes transformations technologiques ces dernières décennies. L'arrivée de la modélisation 3D a provoqué un grand changement dans l'industrie, marquant une nouvelle ère de représentation des données géométriques. L'introduction subséquente du BIM a servi de dépôt polyvalent pour la sémantique provenant de différentes applications tout au long du processus de conception et de construction, y compris mais sans s'y limiter, l'estimation des coûts, l'analyse énergétique et la planification de la production. Cependant, comme Werbrouck, 2018 l'a souligné, le prochain défi pour l'industrie de l'AEC réside dans la nature spécifique du domaine des solutions logicielles BIM actuelles, qui restent inaccessibles à d'autres disciplines. Ce problème de gestion des données est actuellement abordé par le Linked Building Data Community Group (LBD-CG) et des institutions académiques comme l'Université de Gand, qui exploitent le potentiel des technologies du *Semantic Web* pour une gestion des données plus intégrée et efficace, permettant une collaboration interdisciplinaire. Le terme Linked Data BIM (LDBIM) est utilisé dans cette thèse pour désigner ce jalon émergent.

Parallèlement à cette évolution, en tandem avec la quantité croissante de données sémantiques, les données géométriques décrivant un modèle BIM augmentent également en taille et en complexité. Cela rend la tâche de visualisation de plus en plus complexe sur les nouveaux appareils à ressources limitées utilisés dans l'industrie. De tels appareils, par exemple, les smartphones et les tablettes, deviennent de plus en plus populaires sur les chantiers de construction. Les solutions existantes utilisent une étape de filtrage dans le processus de visualisation, connue sous le nom de culling. Cette étape de calcul nécessite le traitement de l'ensemble du fichier 3D ou de la scène pour la rogner jusqu'au frustum de vue de l'observateur (Johansson et al., 2015). Cette opération produit une scène 3D minimale, qui est ensuite utilisée pour le processus de rendu nettement plus gourmand en ressources.

Un goulot d'étranglement se produit lorsque la scène 3D initiale, sur laquelle le processus de culling in-viewer est effectué, devient trop volumineuse pour que l'appareil puisse la gérer. En solution à ce problème, cette thèse propose l'utilisation d'une étape de **pré-culling**. Dans ce processus, et avec l'aide des technologies du Web sémantique, un modèle BIM est stocké et interrogé comme une base de données partitionnée. En séparant le stockage du modèle BIM et donc en déchargeant la tâche computationnellement intensive de culling vers un serveur de stockage dans une étape préliminaire de culling, la quantité de données qui doit être traitée par l'appareil de visualisation est minimisée.

Cette thèse examinera donc dans quelle mesure la géométrie LDBIM peut être pré-élaguée. Elle étudiera la taille minimale de la géométrie qui peut être définie dans une base de données RDF.

À cette fin, les ontologies existantes dans le domaine de l'AEC ou des domaines connexes tels que l'Geographic Information System (GIS) seront explorées. Ces ontologies seront également examinées pour leur potentiel d'utilisation au sein des algorithmes de culling, à l'instar des travaux de Johansson and Roupé, 2009. Dans leur article, ils ont exploité la sémantique d'un modèle BIM au format Industry Foundation Classes (IFC) pour alimenter les algorithmes de culling. Comme les modèles BIM possèdent une hiérarchie sous-jacente inhérente de Bounding Volume Hierarchy (BVH), ceux-ci peuvent être utilisés dans les algorithmes modernes de culling in-viewer tels que le Coherent Hierarchical Culling algorithm (CHC)++ (Johansson et al., 2015).

## Plan

**Chapitre 1 -** Ce chapitre d'introduction présente la motivation de cette thèse et les questions de recherche. Il introduit également le concept de culling et donne un aperçu de la structure de la thèse.

**Chapitre 2 -** En guise d'introduction au Web sémantique, ce chapitre présente les principes des données liées utilisées dans cette thèse. De plus, il explique la complexité et la taille des modèles BIM au sein des bases de données du Web sémantique, qui sont représentés comme des graphes RDF.

**Chapitre 3 -** Johansson et al., 2015 a souligné qu'il y a une pénurie de recherches explorant les performances des visualiseurs BIM actuels. Ainsi, cette recherche sur l'état de l'art vise à se concentrer sur les caractéristiques globales de certains visualiseurs plus récents et prometteurs ainsi que sur les ontologies/outils qui seront utilisés dans cette thèse.

**Chapitre 4 -** Le concept de requêtes dynamiques est introduit dans ce chapitre. Ces dernières représentent la génération automatique de requêtes responsables de l'alimentation des algorithmes de culling, afin d'obtenir les données nécessaires pour visualiser la scène visible pour l'observateur. Par conséquent, tout d'abord, les besoins d'un visualiseur LDBIM sont explorés, ce qui se trouve à la base du processus. Deuxièmement, trois approches où l'action de culling est effectuée sont présentées : dans la requête, dans le visualiseur, et in situ. Chaque approche met en évidence les possibilités de culling en construisant des requêtes SPARQL.

**Chapitre 5 -** En plus du culling proprement dit, ce chapitre propose une approche modulaire pour mettre en œuvre le processus de culling dans un visualiseur LDBIM. Cette base solide sert de cadre pour la recherche future. Informé par le prototype de cette thèse, il identifie les différents composants nécessaires pour créer un visualiseur LDBIM et intégrer le processus de culling en son sein. Il dissèque le cadre du visualiseur en quatre composants principaux : le visualiseur lui-même, le gestionnaire de cache, le processeur de requêtes, et le récupérateur de données.

**Chapitre 6 -** Pour démontrer la faisabilité du processus de culling proposé, un prototype est développé dans ce chapitre. Ce prototype est une preuve de concept qui met en œuvre l'approche modulaire présentée dans le chapitre précédent au sein d'un visualiseur basé sur le web. Il démontre la faisabilité des requêtes dynamiques proposées et démontre également le potentiel d'extension pour mettre en œuvre la visualisation sémantique des sémantiques associées dans la base de données RDF.

## Conclusion

**Chapitre 7 -** Cette thèse a montré qu'il est possible d'élaguer les graphes LDBIM pour réduire la taille de la scène qu'un visualiseur léger doit gérer afin de visualiser un modèle de bâtiment stocké dans une base de données utilisant les technologies du Web sémantique. Elle a également présenté plusieurs approches pour effectuer le culling, chacune avec ses propres avantages et inconvénients, ainsi qu'une approche modulaire pour mettre en œuvre l'ensemble du processus dans un visualiseur web. Cette technologie s'est avérée être une solution viable pour répondre aux besoins exigeants de l'industrie de l'AEC lors de la visualisation de grands modèles BIM. Et elle présente une base solide pour se développer dans un ensemble diver-

sifié de cas d'utilisation et de scénarios nécessitant une représentation visuelle 3D.

## Références


Johansson, M., & Roupé, M. (2009). *Efficient real-time rendering of building information models.* https://www.researchgate.net/publication/220758081

Johansson, M., Roupé, M., & Bosch-Sijtsema, P. (2015). Real-time visualization of building information models (bim). *Automation in Construction, 54*, 69–82. https://doi.org/10.1016/j.autcon.2015.03.018

Werbrouck, J. (2018). Linking data : Semantic enrichment of the existing building geometry. http://lib.ugent.be/catalog/rug01:002494740


Démonstration : https://github.com/flol3622/Pre-culling_LDBIM#demo

Prototype : https://github.com/flol3622/LDBIM-viewer

# Contents

# List of Figures



# List of Tables

# List of Listings

# List of Acronyms

**LRU** Least Recently Used . . . . . 30
**MEP** Mechanical, Electrical and Plumbing . . . . . 9
**MRU** Most Recently Used . . . . . 30
**NOHC** Near Optimal Hierarchical Culling . . . . . 16
**OC** Occlusion Culling . . . . . 15
**OGC** Open Geospatial Consortium . . . . . 14
**OMG** Ontology for Managing Geometry . . . . . 12
**OWL** Web Ontology Language . . . . . 7
**RAM** Random Access Memory . . . . . 2
**RDF** Resource Description Framework . . . . . 5
**RDFS** Resource Description Framework Schema . . . . . 7
**SDK** Software Development Kit . . . . . 38
**SPARQL** SPARQL Protocol and RDF Query Language . . . . . 7
**SQL** Structured Query Language . . . . . 7
**UI** User Interface . . . . . 11
**URI** Uniform Resource Identifier . . . . . 6
**VFC** View Frustum Culling . . . . . 15
**VRAM** Video Random Access Memory . . . . . 16
**W3C** World Wide Web Consortium . . . . . 5
**WKT** Well-Known Text . . . . . 15
**XML** Extensible Markup Language . . . . . q
**XSD** Extensible Markup Language (XML) Schema Definition . . . . . 13

# Chapter 1

# Introduction

From 2D, to 3D and now to BIM. The evolution of the Architecture, Engineering and Construction (AEC) industry has been a long and complex one. The introduction of 3D modeling was the first major step in the industry's evolution, as it allowed for more accurate representations of buildings. No longer solely relying on 2D drawings, a 3D model of a building can be used to create various representations, from a simple 2D floor plan to a complete 3D model. Following the adoption of 3D modeling, the implementation of Building Information Modelling (BIM) emerged as another significant milestone. BIM adds an extra layer of information on top of the standard 3D model. As the digital representation of a building's physical and functional characteristics, BIM serves as a repository for semantics originating from various applications throughout the design and construction processes, including cost estimation, energy analysis, and production planning.

However, as mentioned in Werbrouck, 2018, the next challenge for the AEC industry is related to the domain-specific nature of current BIM softwares, which remains closed off to other disciplines. This data **management** challenge is currently being adressed by the Linked Building Data Community Group (LBD-CG) and other research entities, such as the University of Ghent, through the use of Web of Data technologies[1]. This emerging milestone will be discussed in this thesis under the term Linked Data BIM (LDBIM).

## 1.1 Proposal

Each of these evolutions has brought, and will continue to bring, a significant amount of data together. The volume of data is expected to grow exponentially in the future as the industry shifts towards a more digital approach, and opens up to other stakeholders. The data graphs will not only expand in terms of semantics, but also in geometry.

[1] W3C, 2023.

The growing size of the graph makes visual querying, or simply put, 3D exploration of models, an increasingly difficult task. In particular the newer devices used in the industry, such as phones and tablets. Even though these devices have become more powerful over the years, they still lack the computational resources of traditional office computers.

To visualize the amount of geometric data, Table 1.1 shows the size of the test-models used in Johansson et al., 2015. This study, conducted in 2015, focused on the performance of BIM viewers for large models. It used the following description in regards to the large models:

" Although the Hotel model contains some structural elements they are primarily architectural models. As such, no Mechanical, Electrical or Plumbing (MEP) data is present. However, all models except the Hospital contain furniture and other interior equipment. " (Johansson et al., 2015)

| Model | # of triangles | # of objects | # of geometry batches |
|---|---|---|---|
| Library | 3 685 748 | 7318 | 11 195 |
| Student House | 11 737 251 | 17 674 | 33 455 |
| Hospital | 2 344 968 | 18627 | 22 265 |
| Hotel | 7 200 901 | 41 893 | 62 624 |

Table 1.1: Size of test-models in Johansson et al., 2015

These models demonstrate how basic BIM models can already contain a significant amount of data. LDBIM will not only bring together new stakeholders, but also be able to keep track of multiple geometry versions of each object, should they occur. Therefore, this thesis proposes a new approach to the visual querying of LDBIM models, wherein viewers will not have to load the entire model into memory. As existing visualisation techniques encounter processing limitations when the scene continues to expand in size, particularly when managing non-active parts of the scene, which are stored in the Random Access Memory (RAM). Instead, after filtering at the source, only the geometry needed for the visual tasks at hand will be loaded, while maintaining the original link to each resource for further processing and use cases. This filtering step is commonly referred to as culling in the computer graphics industry and is illustrated in Figure1.1.

Figure 1.1 showcases multiple culling techniques to elaborate on some culling principles. The first technique, **frustum culling**, is used to determine which objects are visible to the user. The second technique, **occlusion culling**, is used to determine which objects are occluded by or behind other objects. And lastly, **back-face culling** is used to determine which faces, instead of whole objects, are facing away from the user.

Figure 1.2 illustrates the basic idea of this thesis, presenting an extra step in the com-

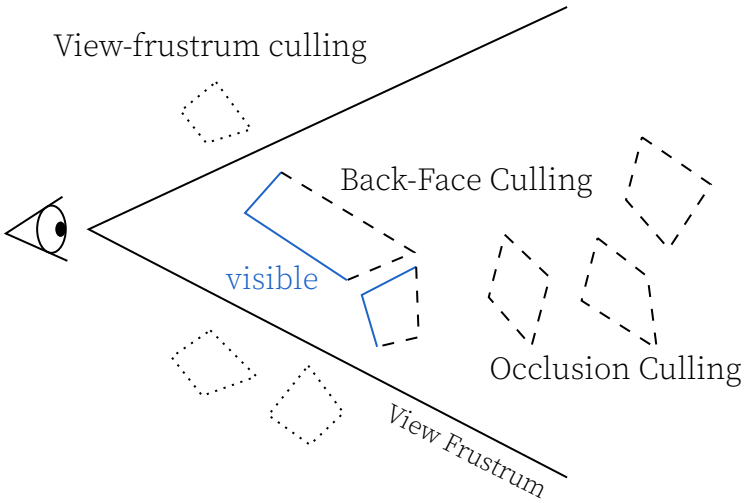



Figure 1.1: Illustration of culling principle, based on Cohen-Or et al., 2003

munication between a user, represented here by a Hand Held Device (HHD), and a database that houses the model. An HHD has been chosen to exemplify a low-powered device used in the field, which requires a lightweight 3D viewer to visualize and explore the digital twin of the building. The HHD is assumed to have no knowledge of the LDBIM model and only receives the geometry that needs to be displayed from the database. On the other hand, the database is assumed to possess, or have access to, all the knowledge of the model and the necessary semantics to perform the culling.

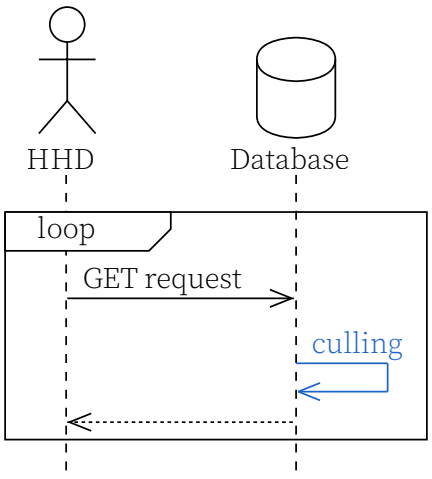



Figure 1.2: Sequence diagram - basic concept

## 1.2 Research questions

This thesis proposes the introduction of culling algorithm technology within the context of LDBIM, to address the previously mentioned issue of the scene's size, by culling the scene at its source prior to sending it to the viewer. As culling algorithms have

been extensively researched and continue to evolve, as is described in section 3.3, the research questions in this thesis concentrate on assessing the feasibility of introducing such algorithms to LDBIM. It aims to propose a set of possible solutions tailored to the problems a 3D viewer would encounter upon receiving and handling a complete 3D model, while highlighting possibilities for future research and specific use cases.

### 1.2.1 To what extent can LDBIM geometry be culled to be streamed to lightweight viewers?

In other words, what constitutes the smallest triples (Section 2.1) or snippets of data required by the viewer and culling algorithms? This question will be addressed by examining possible culling algorithms, their applicability to LDBIM models, and a hands-on approach to the development of a prototype. This exploration will extend this research question to the implementation of secondary parameters, both from a culling perspective, such as special filters, and from a visual standpoint, such as the visualization of semantics related to each object.

### 1.2.2 Can existing semantics and ontologies be used to feed possible culling algorithms?

Unlike the computer graphics industry, this interconnected context already includes both explicit and implicit relationships within the data graph, the latter being derived through inferencing (Section 2.2). This is similar to the work of Johansson and Roupé, 2009. In their paper, they utilized the semantics of a BIM model in an Industry Foundation Classes (IFC) format to develop culling techniques. In contrast, this thesis will focus on the use of Semantic Web resources. As such, it will examine both AEC-specific and AEC-related ontologies, such as those related to Geographic Information System (GIS), to determine if they can be employed to feed culling algorithms.

# Chapter 2

# Linked Data

As mentioned in the Introduction, the evolution from BIM to LDBIM represents an evolution of the data **management** layer. This layer utilizes *Linked Data* which, as stated by the World Wide Web Consortium (W3C), is a collection of interrelated datasets on the Web, formatted in a standard way that is accessible and manageable by Semantic Web tools. The same applies to the relationships among them.[2] The following collection of Semantic Web technologies explores the required environment to achieve this goal.

## 2.1 RDF and triples

At the core of the Semantic Web is the Resource Description Framework (RDF), a data model for describing resources on the Web. RDF is a graph data model that consists of **triples**, which are statements about resources. A triple consists of a subject, a predicate, and an object. The subject is the resource that is being described, the predicate is the property of the subject, and the object is the value of the property. Both the predicate and the object can, in turn, become the subjects of other triples. Listing 2.1 shows an example of an RDF database described in the Turtle format.

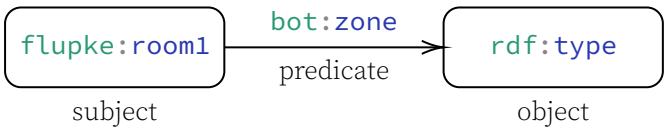



Figure 2.1: Triple structure

The basic, yet versatile, structure of a triple is illustrated in Figure 2.1. Both the subject and object are considered as nodes in the data graph, and they are linked by the

[2] W3C, 2015b.

predicate, which is referred to as an edge. Multiple triples can thus create and link multiple nodes or enrich a connection between two nodes by creating new edges between them. Each element contains a single resource that can be one of the three types: a URI, a literal, or a blank node. A Uniform Resource Identifier (URI) identifies the name and/or location of a resource on the web and, as its name states, is unique and unambiguous, thus enabling queries and reasoning of the same nature. A literal is a value, and a blank node is an anonymous resource, sometimes used as a placeholder when the exact resource is not known or not necessary to specify. Due to their nature, a subject must be either a URI or a blank node, a predicate exclusively a URI, and the object may be any of the three types. As URI descriptions can be very long, a prefix can be used to shorten them. This is illustrated in Listing 2.1 with the `@prefix bot: <https://w3id.org/bot#>`, which declares that `bot:Zone` refers, in its full length, to the address `<https://w3id.org/bot#Zone>`.

```
@prefix fog: <https://w3id.org/fog#> .
@prefix omg: <https://w3id.org/omg#> .
@prefix bot: <https://w3id.org/bot#> .
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .
@prefix flupke:   <http://flupke.archi#> .

flupke:room1 rdf:type bot:Zone ;
    bot:containsElement flupke:coneOBJ ;
    bot:containsElement flupke:cubeGLTF .

flupke:coneOBJ omg:hasGeometry flupke:coneOBJ_geometry-1 ;
    rdf:type bot:Element .

flupke:cubeGLTF omg:hasGeometry flupke:cubeGLTF_geometry-1 ;
    rdf:type bot:Element .

flupke:coneOBJ_geometry-1 rdf:type omg:Geometry ;
    fog:asObj_v3.0-obj "https://raw.githubusercontent.com/flol3622/LDBIM-
    ↪  viewer/main/public/assets/database_1/coneOBJ.obj"^^xsd:anyURI
    ↪  .

flupke:cubeGLTF_geometry-1 rdf:type omg:Geometry ;
    fog:asGltf_v1.0-gltf "https://raw.githubusercontent.com/flol3622/LDBI
    ↪  M-viewer/main/public/assets/database_1/cubeGLTF.gltf"^^xsd:anyURI
    ↪  .
```

Listing 2.1: Example of an RDF database in turtle format

This basic concept can be extrapolated to describe and store any kind of data. The advantage for the AEC industry would be to allow any stakeholders to describe and enrich the knowledge base of a building.

## 2.2 Ontologies and reasoning

When looking at Listing 2.1, a distinction can be made between two types of statements: some refer to classes or properties, such as `bot:Zone` or `bot:containsElement`, while others refer to instances such as `flupke:room1`. The former is referred to as the TBox for "terminology", and the latter is referred to as the ABox for "assertions". The TBox, the ontology layer, is used to describe instances in the ABox and their relationships.

By developing an ontology, the domain of interest and the relationships between the classes and properties can be described. This is achieved by defining the classes and properties of the domain and their relationships. The ontology is then used to reason about the domain, inferring new facts based on the ontology and the existing facts within the domain. This is done by a reasoner, which is software capable of performing the reasoning itself on the ontology and associated data. As mentioned, the reasoner can be used to infer new facts, check if created facts are consistent with the ontology, and check if the ontology itself is consistent.[3] It is often integrated with RDF databases, also known as triplestores or graph databases.

Classes, properties, and their relationships can be defined using Resource Description Framework Schema (RDFS), which is a vocabulary for describing RDF schemas using a basic set of constructs. As an extension of RDFS, Web Ontology Language (OWL) is a vocabulary for describing ontologies using a more expressive set of constructs tailored to the needs of ontologies. Both RDFS and OWL are considered to be formal ontologies themselves, as they describe the classes and properties of the domain of RDF.

## 2.3 Triplestores and SPARQL

As briefly discussed in 2.2, triplestores are RDF databases that store data in the form of a graph. They are used to store and query Linked Data and are often integrated with a reasoner. The data itself is retrieved and modified using the SPARQL Protocol and RDF Query Language (SPARQL).[4] In contrast to Structured Query Language (SQL), SPARQL queries are able to work across multiple triplestores, called SPARQL endpoints. These are known as federated queries, and their results are combined into a single result set. This is useful when the data is distributed across multiple triplestores in a decentralized manner.[5] For example, multiple stakeholders participating in a project,

[3] W3C, 2015a.
[4] W3C, 2015c.
[5] Ontotext, 2022b.

each with their own database.

## 2.4 Complexity of the data graph

The complexity of the data graph is a major concern when working with LDBIM. This section discusses the origins of the different sources of geometric data that enrich it. Firstly, by looking at the different Level of Detail (LOD)'s that can be used to describe a building, which already exists in the BIM domain. Secondly, by looking at LDBIM and the exponential growth it imposes on the data graph.

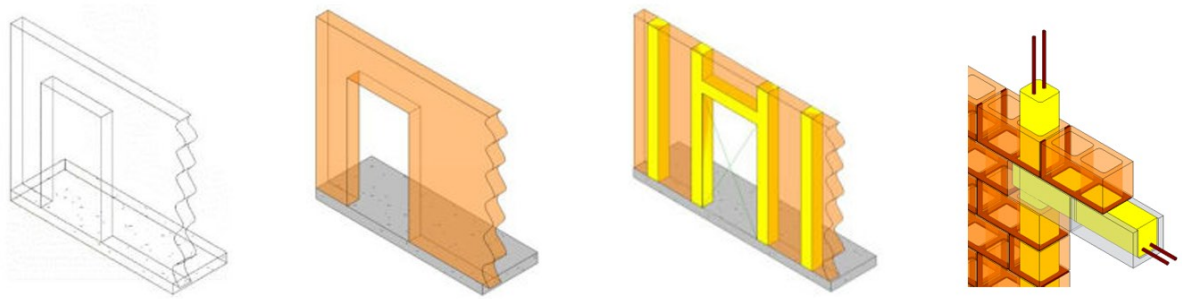


Figure 2.2: LOD evolution
from BIMForum, 2015

### 2.4.1 BIM geometry

The 3D model of a building consists of a multitude of sub-models, describing objects for all the different stakeholders participating in the project. Some describe very large objects, and some very small parts. Both can be defined in their most simple and abstract form or have an intricate and complex geometry. For instance, a door can simply be defined as a box, or up to the level of the screw-thread for the hinge system. The level of abstraction is here described as the LOD, which is most of the time pre-selected for the needs of a BIM model, and is applied throughout a single model.

As shown in Figure 2.2, a standard BIM workflow goes through multiple phases, each with their associated model and LOD. This makes it an important concept in the AEC industry, as it allows for a very efficient workflow. The modeling step is approached from a top-down perspective, starting with rougher geometries describing the broader ideas of a concept model and evolving to a more refined model for the construction documentation phase. As the last and longest-standing model, a higher LOD can be used to describe subtle changes in the evolution of a building during the operation phase.

### 2.4.2 LDBIM geometry

The interconnectivity of semantics can also be applied to geometry descriptions. This could allow the co-existence of multiple LODs in a single model database. Besides storing the evolution of a single element's geometry, it enables the linking of the different LODs, described in 2.4.1, to each other. Not only that, but extending onto the size of the models described in Table 1.1, already existing Mechanical, Electrical and Plumbing (MEP), structural, alongside many other stakeholders' geometry can be added.

# Chapter 3

# State of the art

As mentioned in Johansson et al., 2015, existing research on the performance of currently used BIM viewers is quite limited. This state-of-the-art research will, therefore, focus on the overall features of some promising newer viewers and the ontologies/tools that will be used in this thesis.

## 3.1 Existing BIM viewers and ontologies

### 3.1.1 Qoniq and LOD Streaming for BIM

Qonic focuses on developing an open platform BIM viewer. With the use of Unity to enable cross-platform compatibility, they focused on two main aspects: performance and aesthetics. The latter refers to the visual quality of the viewer, offering both a seamless experience for the viewer as well as a pleasant one, with, for example, the implementation of ambient lighting and shadow castings. The first and most researched aspect of their viewer, the performance, is mainly focusing on a LOD culling algorithm.
(T. Strobbe, personal communication, November 25, 2022)

#### 3.1.1.1 Qoniq's approach to LOD streaming

Their core research is developing a dynamic LOD streaming model. Starting from the geometry and semantics of an IFC file, they compute an LOD hierarchy tree of the model. Through multiple mesh decimation algorithms, they reduce the number of triangles of each object's mesh, regardless of the semantics associated with that object. On top of that, a filtering algorithm is implemented in the streaming model to filter out objects, regarding their semantics, that are not relevant to the current camera position. In doing so, they both reduce the size of models far from the viewpoint and evaluate the need to show certain objects based on their nature, extracted from semantics in the IFC file, and their distance to the camera. The resulting dynamic LOD streaming

model is reevaluated at each camera move in Unity.
(T. Strobbe, personal communication, November 25, 2022)

Unity was chosen as it allows for writing once and deploying everywhere. This means that the viewer can be used on any platform, including mobile devices and browsers. The performance results are thus related to the hardware capabilities of each device, with the exception of the browser, where the performance of Unity's WebGL build is limited to a scene size of 2Gb.[6]

#### 3.1.1.2 Advantages and trade-offs

Being able to run on many platforms, offering a smooth viewer experience and a pleasing aesthetic makes it an ideal candidate for lightweight viewers on the job site. However, the LOD library has to be computed on every model update. The decimation algorithms are furthermore computational results that are not humanly reviewed. This means that the quality of the resulting meshes is not guaranteed for the lower LODs, which are, as illustrated in Figure 2.2, already modeled in previous design phases. LDBIM could, by interconnection, recall previous LODs in the viewer's scene. Without the need for computational remodeling. Nevertheless, Qonic's approach is already versioning-proof as later LOD's may vary from the last design choices. They therefore serve as the goal of this thesis, outside the LDBIM context.

### 3.1.2 ld-bim.web.app

"The purpose of the app is to showcase our LBD toolset and to demonstrate the capabilities of Linked Building Data to newcomers."[7]

https://ld-bim.web.app/ demonstrates a viewer built around an RDF database. It separates the data from an IFC file into semantics, stored in the previously mentioned graph, and a glTF model, together with a JavaScript Object Notation (JSON) file containing a reference table. Extra local or remote graphs can be added to the User Interface (UI). As it contains a SPARQL engine to query and visualize, in the form of highlighting, the results of the query in a 3D viewer. The viewer is based on the ifc.js project, which is itself based on the three.js 3D JavaScript library.

### 3.1.3 AEC related ontologies

As mentioned in the second research question 1.2.2, this section will discuss AEC-related technologies that are actively researched by the LBD-CG[8].

[6] Unity, 2023.
[7] Rasmussen and Schlachter, n.d.
[8] LBD-CG, 2022.

### 3.1.3.1 BOT

The Building Topology Ontology (BOT) proposes a set of classes and properties, "which provides a high-level description of the topology of buildings including storeys and spaces, the building elements they may contain, and the 3D mesh geometry of these spaces and elements." (Rasmussen et al., 2020), as illustrated in Figure 3.1. This high-level description could be fed to portal-culling algorithms in a situation where the visibility is contained within one `bot:Space` or `bot:adjacentZone`, or it could extend the scope to `bot:adjacentZone`[9]. Additionally, it could play a part in the construction of the Bounding Volume Hierarchy (BVH) needed for other occlusion culling algorithms, such as the Coherent Hierarchical Culling algorithm (CHC)++ (Johansson et al., 2015).

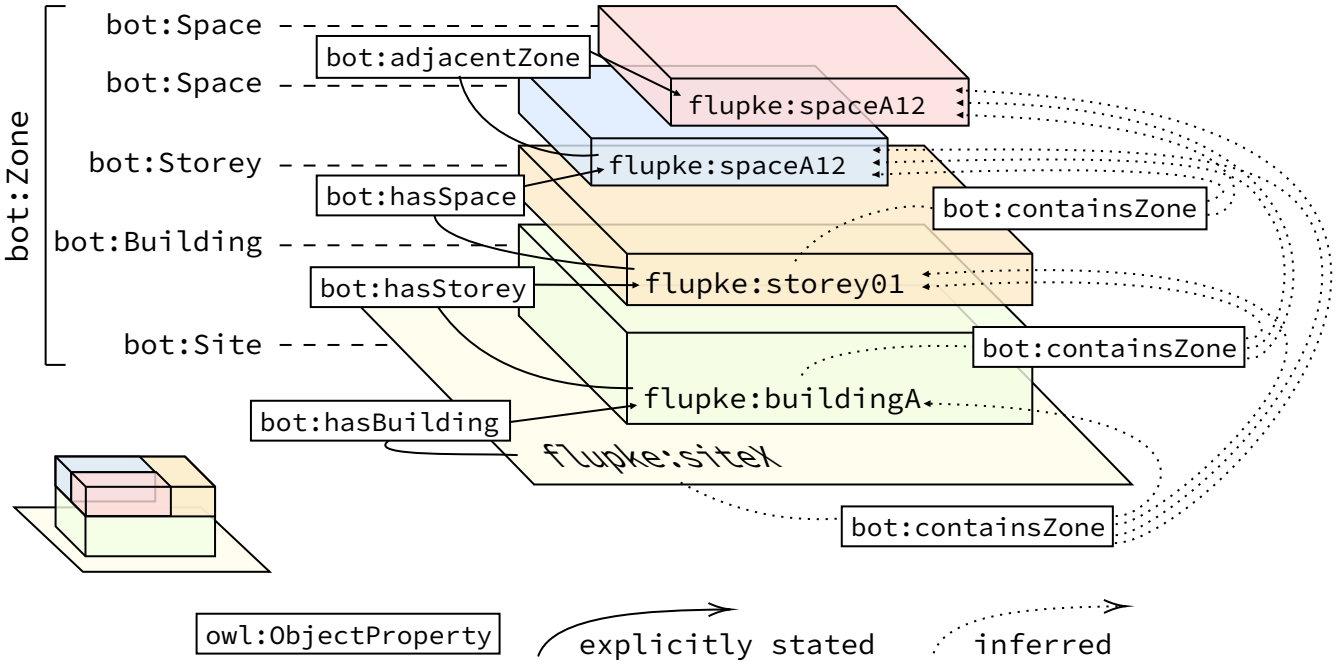



Figure 3.1: Illustration of the BOT ontology, based on Rasmussen et al., 2020.

### 3.1.3.2 FOG and OMG

With the help of File Ontology for Geometry formats (FOG) and Ontology for Managing Geometry (OMG), geometry descriptions can be linked in the data graph. The innovation lies in the choice to store it either inside or outside the graph, by means of one triple referring to a literal or an URI. Listing 2.1 showcases multiple examples of objects assigned with a geometry description using an URI(Bonduel et al., 2019).

[9] Linietsky et al., 2023.

```
flupke:coneOBJ_geometry-1 fog:asObj_v3.0-obj "https://..."^^xsd:anyURI .
```

Listing 3.1: Example of FOG usage

Listing 3.1 describes a subject of datatype `xsd:anyURI` from the XML Schema Definition (XSD)[10]. The versatile approach of Bonduel et al., 2019 also proposes the following datatypes: `xsd:string` for American Standard Code for Information Interchange (ASCII)-based geometry descriptions or `xsd:base64Binary` for binary geometry descriptions.

The format of the geometry is assigned directly by the predicate in Listing 3.1, which is `fog:asObj_v3.0-obj`. This further infers the statements in Listing 3.2.[11]

```
flupke:coneOBJ_geometry-1 fog:asObj "https://..."^^xsd:anyURI ;
    ex:LOD "100"^^xsd:integer .
```

Listing 3.2: FOG inference examples

The possibility of introducing an external geometry location adds a new participant to Figure 1.2, the external database, from which some files are expected to be fetched. This is illustrated in Figure 3.2.

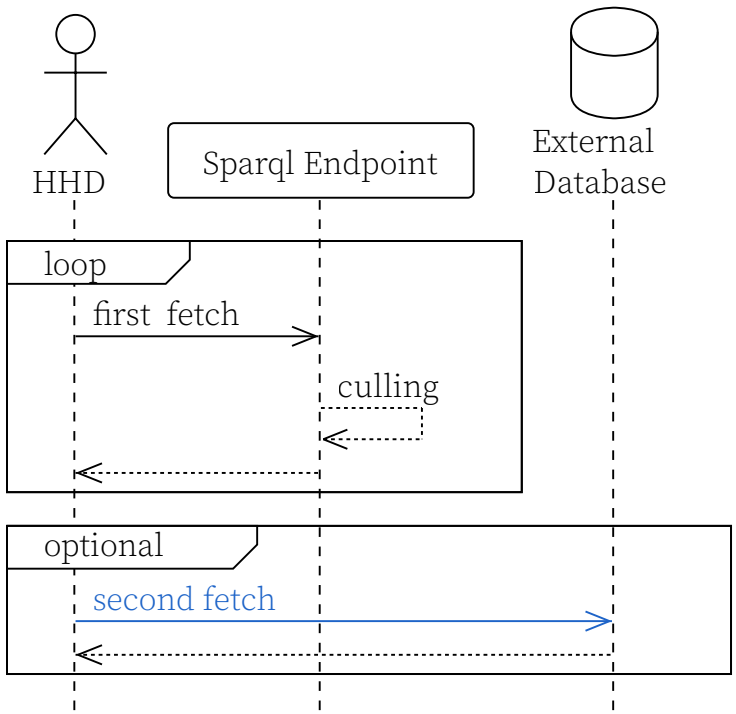



Figure 3.2: Sequence diagram with external database - concept

[10] Carrol and Pan, 2006.
[11] Bonduel et al., 2020.

Bonduel et al., 2019 refers to the proposal of the LBD-CG stated in Holten Rasmussen et al., 2018 to write Linked Data patterns on three possible levels, "each having a different degree of complexity". The first and second levels are illustrated in Figure 3.3. Level 2 allows assigning multiple geometry descriptions to a single object, each with, for example, a different LOD.

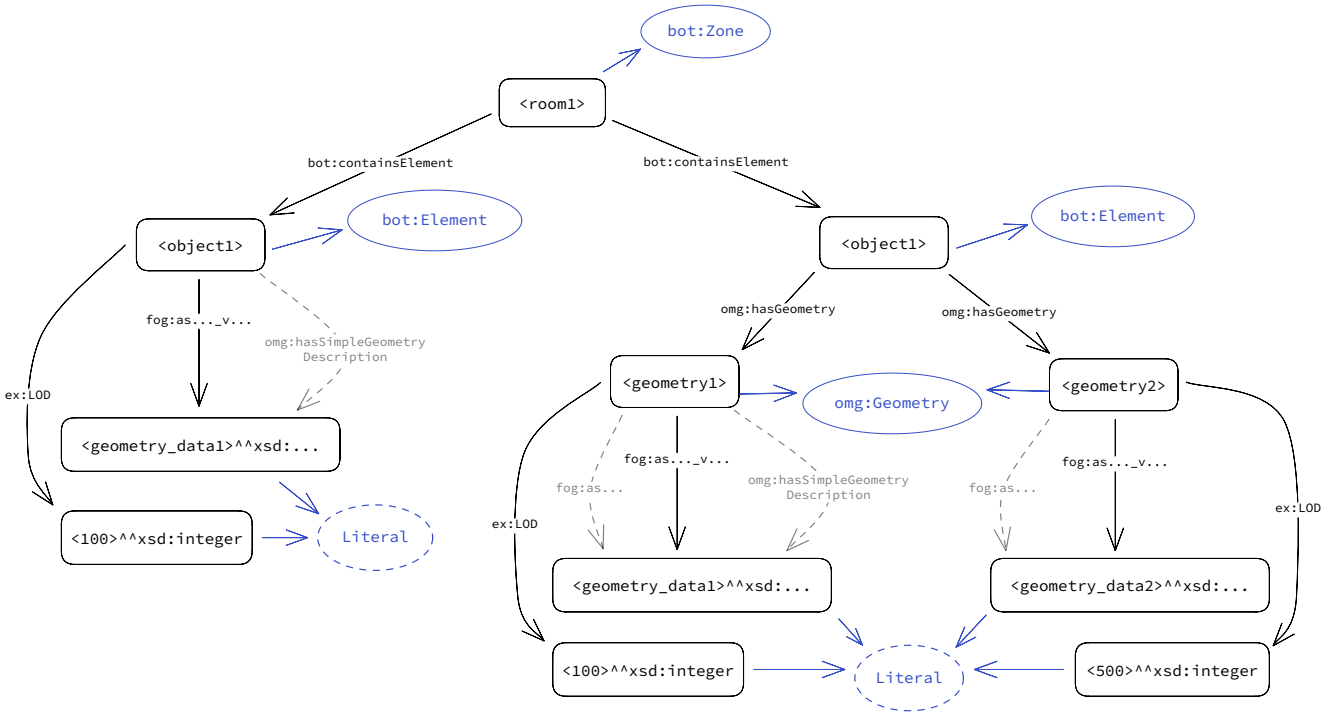



Figure 3.3: Illustration of Level 1 (left) and Level 2 (right) of the FOG and OMG ontologies, based on Bonduel et al., 2019.

### 3.1.4 GIS related ontologies

The technological field of study, Geographic Information System (GIS), is closely related to the BIM domain. The central standards organization, Open Geospatial Consortium (OGC), which actively maintains the GIS standards, is also prominent in the Semantic Web scene. They recognize widely adopted standards such as GeoSPARQL.[12]

#### 3.1.4.1 GeoSPARQL

"The OGC GeoSPARQL standard supports representing and querying geospatial data on the Semantic Web. GeoSPARQL defines a vocabulary for representing geospatial data in RDF, and it defines an extension to the SPARQL query language for processing geospatial data. In addition, GeoSPARQL is designed to accommodate systems based on qualitative spatial reasoning and systems based on quantitative spatial computations."[13]

[12] OGC, 2023a.
[13] OGC, 2023b.

As multiple triplestores and SPARQL endpoints support the GeoSPARQL extension, it is a viable candidate for spatial and LOD culling algorithms. Such algorithms require spatial data, such as the distance from the viewpoint to the object. Spatial query functions proposed in this extension are needed for this purpose. The functions can compute on nodes of geospatial geometry as if they are expressed using Well-Known Text (WKT) or the Geography Markup Language (GML). These expressions can be assigned by using the predicates `geo:asWKT` or `geo:asGML`. However, GeoSPARQL comes with some limitations that are less prevalent in the GIS domain, which mostly requires 2D data (Perry & Herring, 2012), in contrast to BIM where 3D distance functions would be needed. Despite such limitations, GeoSPARQL remains a viable solution for spatial querying, and workarounds could be employed to address them.

## 3.2 On the market viewers comparison

Johansson et al., 2015 mentioned in their paper the lack of research about objective BIM viewers comparison and made one as a result. The size of the model they tested can be found in Table 1.1. They evaluted the following viewers:

- DDS CAD Viewer
- Tekla BIMsight
- Autodesk Navisworks
- Solibri Model Viewer

### 3.2.1 General Features

Their study had two main goals. Firstly, evaluating existing viewers and their capabilities, they identified the acceleration techniques used, which are presented in Table 3.1.

| BIM viewer | Acceleration technique |
|---|---|
| Solibri 9.0 | VFC \| DC (optional) \| HAGI (optional) |
| Naviswork 2015 | VFC \| DC (optional) \| CPU OC (optional) GPU OC (optional) |
| BIMSight 1.9.1 | VFC |
| DDS 8.0 | VFC \| DC (optional) |
| DDS 10.0 | VFC \| DC |

Table 3.1: Acceleration techniques used by tested viewers from Johansson et al., 2015. (View Frustum Culling (VFC), Drop Culling (DC), Hardware Accelerated Geometry Instancing (HAGI), Central Processing Unit (CPU), Occlusion Culling (OC))

Secondly, they implemented modern culling algorithms and strategies such as CHC++. The worst-case scenarios are shown in Figure 3.4 against the Solibri viewer. The results are quite promising, but as concluded by the authors, the gains are limited to the capacities of the Graphics Processing Unit (GPU), Video Random Access Memory (VRAM), and RAM.

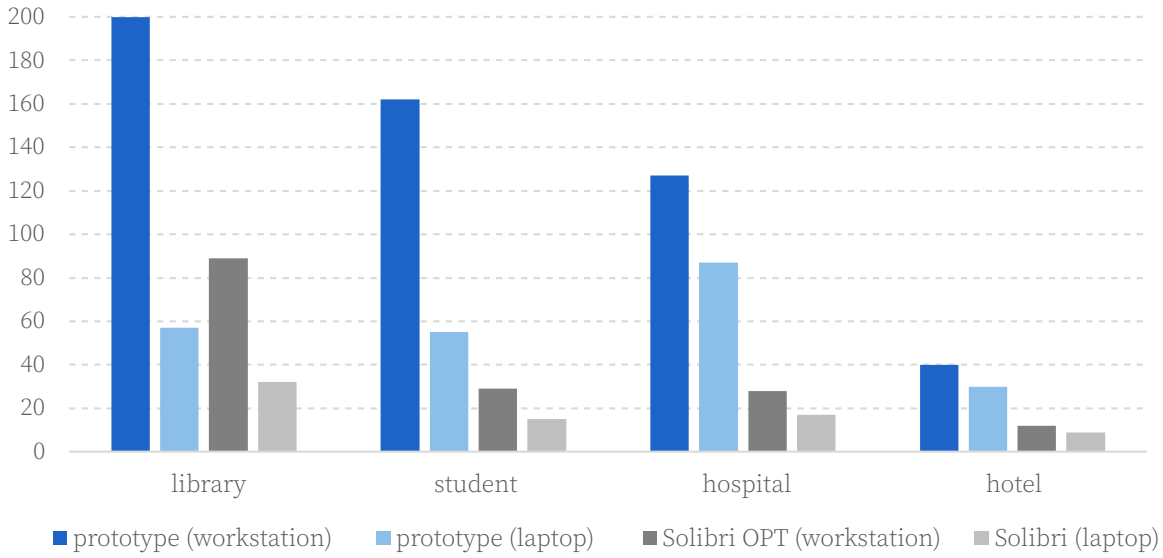



Figure 3.4: Comparison in rendering performance.
from Johansson et al., 2015

## 3.3 Culling algorithms

Johansson et al., 2015 presented in their paper a new BIM viewer equipped with the powerful CHC++. This is a third-generation occlusion culling algorithm developed by Mattausch et al., 2008a, the first being the CHC (Bittner et al., 2004), followed by the Near Optimal Hierarchical Culling (NOHC) (Mattausch et al., 2008b). Their conclusion stated that although occlusion culling is very efficient, it is still bound to the scene size, which is limited by hardware capabilities. More precisely, the GPU, VRAM, and RAM capacities.

# Chapter 4

# Dynamic Queries

This chapter introduces the concept of dynamic querying. In this thesis, it refers to the automatic generation of queries responsible for obtaining the data needed to visualize building elements from a BIM model within an RDF graph. The examples are presented as static SPARQL queries, since the automation itself depends on the implementation or framework used. A link to the appendix is provided, where the implementation within the prototype of this thesis is explained.

The structure is as follows: first, the requirements for the viewer are researched, as its functioning will dictate the output of the queries. Second, the capabilities of the viewer are explored in relation to the visualization of semantic data, thus emphasizing the added value of working with Linked Data. Third, three types of dynamic queries for culling are presented, each with its own advantages and disadvantages.

## 4.1 Requirements

A viewer designed to visualize data stored in an RDF graph is required to understand the data stored within it. Therefore, the requirements for the viewer align with those of its source, the RDF graph. Section 3.1.3.2, which discusses the use of both the FOG and OMG ontologies, offers an overview of the available options in terms of file format and file source. The FOG ontology supports the description of a wide range of geometry formats, as illustrated in Table 4.1. In conjunction with the OMG ontology, which allows for the description of the file source using the datatype of the literal, it can be concluded that the viewer should be able to handle a broad spectrum of file formats, preferably described in the FOG ontology, and accept both remote files and literal values.

| | | | |
|---|---|---|---|
| COLLADA | Compressed LAS | Compressed Nexus | DWG |
| E57 | GeoJSON | Well Known Text SFA | GML |
| IFC | IGES | LAS Point Cloud | Nexus |
| OBJ | PCD Point Cloud | Uncompressed LAS | Revit |
| Rhino | Shapefile | Simple Feature Access | SketchUp |
| SPFF | STEP SPFF | Uncompressed Nexus | SVG |
| PLY | STL | Well Known Binary SFA | X3D |
| glTF | | | |

Table 4.1: List of geometry formats that can be assigned with the FOG ontology.[14]

## 4.2 Beyond geometry

This section highlights the adavantages a viewer based on Linked Data has over a viewer based on traditional file-based systems, by extenting the thought of a 3D viewer to its ability to visualise non geometric data from its source. LDBIM links geometrical entities to their corresponding semantic data, which can be visualised in the viewer. This allows for the visualisation of data that is not directly related to the geometry of the building elements, such as the physical properties of a wall or the cost of a door. The possibilities are endless, as long as the data is available in the RDF graph. The following sections will discuss different possible implementations.

### 4.2.1 BCF integration

As a first possible implementation of non-geometric data, this section examines the BIM Collaboration Format (BCF) buildingSMART standard. BCF is an open file format that enables the creation and communication of issues about BIM models[15]. Both it and its translation in the Semantic Web as BIM Collaboration Format Ontology (bcfOWL) (Schulz et al., 2021) link a screenshot, a camera angle, and a list of concerned entities to form a specific issue[16].

This type of semantic offers two types of implementations. The first is the positioning of a screenshot, together with its camera position and orientation, within the 3D scene. This allows for the visualization of issues, which can be linked to the screenshot, in the viewer, offering communication integration. The second implementation involves the metadata surrounding issues that can be used as visual properties to feed specific queries. This type of implementation is discussed in the next section, Visualising semantic.

[14] Bonduel et al., 2020
[15] BuildingSMART International, 2023.
[16] BIMcollab, 2023.

### 4.2.2 Visualising semantic

When examining semantics such as physical properties of entities, free from geometric and spatial data (see BCF integration), a visual interpretation superimposed on the viewer offers powerful rendering possibilities. By coloring elements based on their properties, both physical and non-physical, the viewer can be used to detect anomalies or insights in the model, in a feature-rich output medium.

Physical properties such as thermal, acoustic, structural, and others, or non-physical properties like cost, time, and pathologies, can all be described and linked in the RDF graph. The expressive capabilities of SPARQL enable complex and fine-grained queries, offering application-specific query creation about these properties. By selecting a specific subset or combining them, a user or developer can transform the viewer into a powerful tool.

As such, the viewer is expected to comply with a multitude of requirements, which are not all covered in this thesis. Chapter Modular Approach thus proposes a modular approach, allowing for a step-by-step implementation of the viewer, starting with the most basic requirements, which are adopted in Chapter Prototype, while leaving room for future extensions.

## 4.3 In situ WKT location

The first type of dynamic query identifies the room in which the observer/camera is located using the WKT serialization of `bot:Space` entities. It proposes to base its culling algorithm on super-elements such as rooms, thus grouping the scene into a limited number of elements within meaningful boundaries. As entities are primarily viewed within their allocated rooms, this approach takes advantage of the spatial organization of buildings using the BOT ontology. Moreover, not all building elements have a meaningful WKT serialization (e.g., a door), which makes the use of super-elements necessary.

This approach is limited to 2D, as the widely adopted GeoSPARQL functions used to achieve this are constrained to 2D. Nonetheless, the approach can be extended to 3D if the GeoSPARQL functions are extended to 3D in the future or the needed introduction of a 3D SPARQL is adopted.

Listing 4.1 proposes a static query in which a location is first assigned to the variable `flupke:room1` to create an easily reusable query. This location is the WKT serialization of a point, the coordinates of which can be updated at each move of the camera in the viewer. It therefore represents the location of the observer both on-site, “in situ” for Augmented Reality (AR) use cases, and in the 3D scene.

The query then combines two sets of so-called “entities”, represented in the graph as `bot:Element` or `bot:Space`, both of which are linked in the graph to their correspond-

ing WKT and geometric serialization using an OMG level 1 pattern explained in Figure 3.3. This pattern level links the literal directly to the entity, without the need for an intermediate node, but without the possibility to assign multiple serializations of the same type. The geometry literal is assigned using the FOG ontology, while the WKT literal is assigned using the GeoSPARQL ontology, as illustrated in Figure 4.1. As this last serialization requires a WKT format, a literal assigned with `geo:asWKT` is required, and queried for in the query.

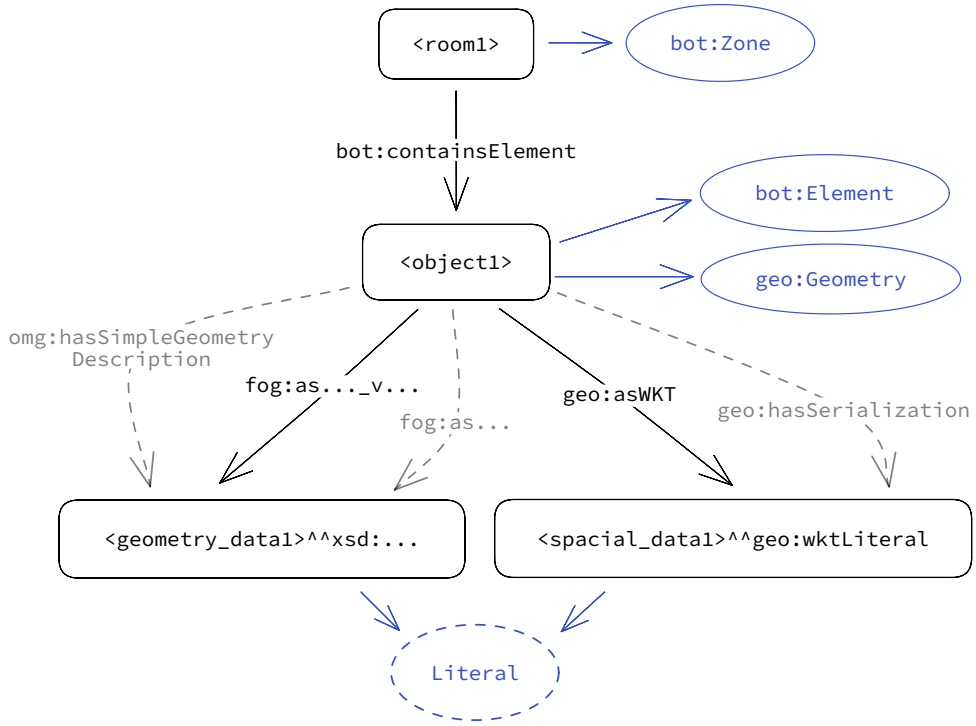



Figure 4.1: Serialization of a geometry using the OMG level 1 pattern.

The separation into two sets allows for querying entities that have a WKT serialization but are not located in a `bot:Space` (such as a floor, which would not be linked to a room) and entities that are located in a `bot:Space` that has a WKT serialization. The latter variant uses the `bot:containsElement` or `bot:adjacentElement` properties to select entities within a room.

After filtering out spaces, the geometry of which is not needed by the viewer, it filters the entities' geometry based on a list of implemented/accepted formats by the receiving viewer. These last consist of a list of FOG super-properties inferred by their assignment. In other words: a triple in the RDF graph of the form `?entity fog:asObj_v3.0 ?geometry_data` infers `?entity fog:asObj ?geometry_data` (as shown in Figure 4.1). This allows for the use of a single property to query related formats while still permitting the declaration of more specific formats if needed.

In the end, the query returns a list of entities that are located in the same room as the

observer, or are not located in a room but have a WKT serialization that is validated by the GeoSPARQL filter function. Together with the needed metadata, which is further explained in Section 5.4.

```
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX inst: <https://172.16.10.122:8080/projects/1001/>

SELECT DISTINCT ?entity ?fog_geometry ?datatype
WHERE {
  BIND(STRDT("POINT(5000 -5000)", geo:wktLiteral) AS ?location)

  {
    ?entity geo:asWKT ?entityWKT .
    FILTER(geof:sfWithin(?location, ?entityWKT))
  }
  UNION
  {
    ?space rdf:type bot:Space .
    ?space geo:asWKT ?spaceWKT .
    FILTER(geof:sfWithin(?location, ?spaceWKT))
    ?space bot:containsElement|bot:adjacentElement ?entity .
  }

  FILTER NOT EXISTS { ?entity rdf:type bot:Space }
  ?entity ?fog_geometry ?geometryData .
  FILTER(?fog_geometry IN (fog:asObj, fog:asStl, fog:asGltf, FOG:asIfc))
  BIND(DATATYPE(?geometryData) AS ?datatype)
  FILTER(?datatype = xsd:anyURI)
}
```

Listing 4.1: Dynamic culling query using GeoSPARQL

In the case of Listing 4.1, the GeoSPARQL `FILTER` uses the function `geof:sfWithin` to evaluate if the given point coordinates are within the WKT serialization, which in most cases represents a WKT polygon. For example:
`"POLYGON ((40 10, 30 10, 30 40, 40 40, 40 10))"^^geo:wktLiteral`

The use of GeoSPARQL results in a very efficient query, as the filtering is done directly in the query and does not require any post-processing. Although the lack of 3D capabilities is a limitation, it allows, in 2D to easily implement more complex algorithms leveraging the use of different functions. With which, together with a level 2 OMG pattern, LOD culling based on the distance to the observer can be implemented.

Selecting rooms and entities outside the room in which the observer is present is also conceivable when leveraging the use of the `bot:adjacentZone` property. Or using the GeoSPARQL distance to a room.

## 4.4 In viewer “bot:Space” identification

In this culling approach, ray tracing is employed to assess the room where the observer is situated. As illustrated in Figure 3.2, out of the three participants, only one utilizes a 3D engine—a component capable of handling geometric modeling operations, such as ray tracing—referred to as the 3D viewer. In essence, both the SPARQL endpoint and the external database lack the ability to perform 3D operations or evaluations. Consequently, this approach capitalizes on the 3D viewer to pinpoint the room in which the observer is positioned.

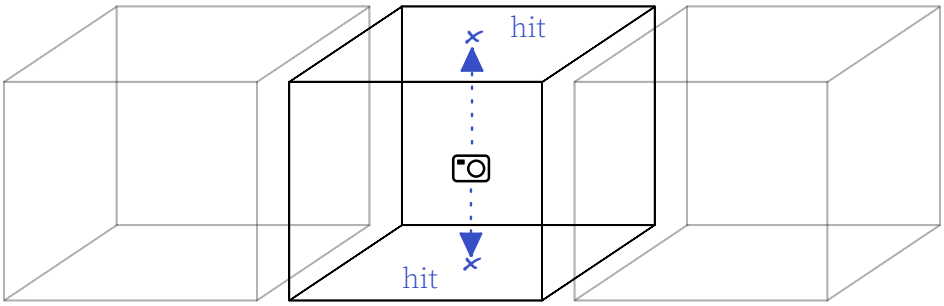



Figure 4.2: In viewer “bot:Space” identification, with raytracing

This process involves casting a ray from the observer’s position both upwards and downwards, and evaluating the first intersection with a `bot:Space` entity. Unlike the previous approach, the geometry of the rooms themselves is required by the viewer, necessitating an initial query to load the rooms’ geometry. By utilizing multiple layers in the viewer, the rooms’ geometry can be concealed from the user while still being employed for ray tracing. Listing 4.2 demonstrates the function used in this thesis prototype to assess the location. The scope of the picking is limited to the rooms using the cache’s metadata, in this case `lru: RefLRU`. Subsequently, two picking operations are carried out and compared, as depicted in Figure 4.2; if they match, the intersected room’s URI is returned.

Once the room is identified, the query in Listing 4.3 is executed to retrieve the entities located within the room. This is achieved by querying the `bot:containsElement` property of the room. Similar to the previous method, it necessitates the combination of two sets of entities in this case: those located in the room and in the adjacent rooms themselves. This enables the optimization of the viewer by evaluating only the adjacent rooms, rather than all the rooms in the database.

```
function findRoom(viewer: RefViewer, lru: RefLRU, position: number[]):
↪  string | undefined {
    const down = [0, -1, 0];
    const up = [0, 1, 0];
    const rooms: string[] = [];

    lru.current?.forEach((value, key) => {
      if (value.botType === "Space") {
        rooms.push(key);
      }
    });

    function pick(direction: number[]) {
      return viewer.current?.scene.pick({
        origin: position,
        direction: direction,
        pickSurface: true,
        includeEntities: rooms,
      });
    }

    const resultDown = pick(down);
    const resultUp = pick(up);

    if (resultDown?.entity === resultUp?.entity) {
      return resultDown?.entity?.id;
    }
  }
```

Listing 4.2: Typescript code for raytracing in viewer

The remainder of the query closely mirrors the previous one (Listing 4.1), with the distinction that it necessitates the additional storage of the `?botType` property to keep track of BOT classes and to identify `bot:Space` elements locally.

Due to the absence of `bot:adjacentZone` relations in the database used for the prototype, the query is unable to directly query the adjacent rooms. Instead, the property `bot:adjacentElement` is employed to query related room's adjacent walls and, consequently, the corresponding rooms. The lack of BOT relations in the utilized database was identified as a recurring issue in this thesis. Chapter **??** delves into this problem in greater detail.

This approach takes advantage of the 3D viewer and its engine to contribute to the culling process, although the first step requires downloading all the rooms' geometry. However, this step can be replaced by an initial manual localization by the user. By utilizing the adjacent rooms during subsequent queries, the impact on performance and

resource usage remains minimal throughout the operation. In contrast to the In situ WKT location, this method is not limited to 2D but can be employed in 3D. However, it requires the geometry format of the room to be compatible with the viewer.

```
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX inst: <https://172.16.10.122:8080/projects/1001/>

SELECT DISTINCT ?entity ?fog_geometry ?datatype ?botType
WHERE {
    BIND (inst:room_1xS3BCk291UvhgP2dvNvkw AS ?room) .

    {
        ?room bot:adjacentElement ?adjacentWall .
        ?adjacentWall rdf:type bot:Element .
        ?entity bot:adjacentElement ?adjacentWall .
        ?entity rdf:type bot:Space .
    } UNION {
        ?room bot:containsElement|bot:adjacentElement ?entity .
    }
    FILTER (?botType != bot:Space)

    ?entity ?fog_geometry ?geometryData .
    ?entity rdf:type ?botType .
    FILTER(STRSTARTS(str(?botType), "https://w3id.org/bot#"))
    FILTER(?fog_geometry IN (fog:asObj, fog:asStl, fog:asGltf, FOG:asIfc))
    BIND(DATATYPE(?geometryData) AS ?datatype)
    FILTER(?datatype = xsd:anyURI)

}
```

Listing 4.3: Querying in viewer ”bot:Space” identification

## 4.5 In query OBJ geometry filtering

This last approach is similar to the previous one, in that it also relies on the existing 3D geometry of the rooms, not their additional WKT serialization. However, in contrast to In viewer “bot:Space” identification, the computation is once again offloaded to the SPARQL endpoint. Since this endpoint is unable to perform geometric operations, a string analysis is performed instead.

The idea proposed by this thesis is to evaluate a string value, the literal representing the geometry, within a single SPARQL query. As complicated string operations are dif-

ficult to perform with native SPARQL functions, a custom JavaScript function is added to the SPARQL endpoint. This function is then called within the query, as illustrated in Listing 4.4. The prototype of this thesis uses GraphDB as its SPARQL endpoint and RDF database, which allows the addition of user-defined functions. Listing 4.5 demonstrates the addition of the function to the endpoint.

```
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX inst: <https://172.16.10.122:8080/projects/1001/>
PREFIX jsfn: <http://www.ontotext.com/js#>

SELECT ?entity ?fog_geometry ?datatype ?botType
WHERE {
    BIND("5000,-15000,2000" as ?position)

    # select the room
    ?room a bot:Space .
    ?room fog:asObj ?obj .
    BIND(DATATYPE(?obj) AS ?type)
    FILTER(?type = xsd:string)
    FILTER(jsfn:insideObjAABBox(?obj, ?position))

    # select the entities in the room
    ?room bot:containsElement|bot:adjacentElement ?entity .
    ?entity ?fog_geometry ?geometryData .
    FILTER(?fog_geometry IN (fog:asObj, fog:asStl, fog:asGltf, FOG:asIfc))
    BIND(DATATYPE(?geometryData) AS ?datatype)
    FILTER(?datatype = xsd:anyURI)
    ?entity rdf:type ?botType .
    FILTER(STRSTARTS(str(?botType), "https://w3id.org/bot#"))
}
```

Listing 4.4: Querying in query OBJ geometry filtering

The function itself (Listing 4.6) evaluates whether a given point is inside a geometry by extracting the vertex coordinates of the geometry literal to build an Axis Aligned Bounding Box (AABB) in the form of a $3 \times 2$ matrix representing both a minimum and maximum value for each of the three axes. It then evaluates each point's coordinates against the given interval, returning a boolean value. The returned boolean can be interpreted by the `FILTER` clause inside the query, as illustrated in Listing 4.4.

```
PREFIX extfn:<http://www.ontotext.com/js#>

INSERT DATA {
    [] <http://www.ontotext.com/js#register> '''
function insideObjAABBox(objString, point) { ... }
'''
}
```

Listing 4.5: Inserting new javascript function in GraphDB

The advantages of this technique are that, similar to the previous proposal, 3D culling is achievable, there is no need for an extra serialization such as WKT, and the computation is offloaded to the SPARQL endpoint. However, the possibility to evaluate multiple geometry formats is on one side more flexible, as newer functions can interpret newer formats, not restricted by the viewer capabilities. This builds upon the concept of an all-knowing database serving a knowledge-free viewer (Section 1.1). Although the possibility of allowing the JavaScript functions to fetch external files was not tested in this thesis, it could be a potential extension.

On the other hand, the function was tailored specifically for the SPARQL endpoint used in this thesis, GraphDB, and would need to be adapted to other endpoints since no standard for user-defined functions was found during the research. Additionally, the JavaScript function was limited during the development of the prototype to a restricted scope of the JavaScript language, resembling the ECMAScript 2009 (ES5) version, which reduces the possibilities and optimization of the function.

```
function insideObjAABBox(objString, point) {
  var lines = objString.split("v ");
  var vertices = [];
  for (var i = 0; i < lines.length; i++) {
    if (lines[i].trim() !== "") {
      var coords = lines[i].split(" ");
      var vertex = [
        parseFloat(coords[0]),
        parseFloat(coords[1]),
        parseFloat(coords[2]),
      ];
      vertices.push(vertex);
    }
  }
  var verticesT = transpose(vertices);
  var AABOX = [];
  for (var i = 0; i < 3; i++) {
    var minVal = Math.min.apply(Math, verticesT[i]);
    var maxVal = Math.max.apply(Math, verticesT[i]);
    AABOX.push([minVal, maxVal]);
  }

  var location = point.split(",");
  console.log(AABOX);

  for (var i = 0; i < 3; i++) {
    var position = parseFloat(location[i]);
    if (position < AABOX[i][0] || position > AABOX[i][1]) {
      return false;
    }
  }
  return true;
}

function transpose(matrix) {
  var transposedMatrix = [];
  for (var i = 0; i < matrix[0].length; i++) {
    var newRow = [];
    for (var j = 0; j < matrix.length; j++) {
      newRow.push(matrix[j][i]);
    }
    transposedMatrix.push(newRow);
  }
  return transposedMatrix;
}
```

Listing 4.6: Querying in situ WKT location

# Chapter 5

# Modular Approach

This chapter describes a theoretical modular approach for the implementation of this thesis' proposed LDBIM viewer and associated culling techniques. This design principle consists of separate, independent modules, with each module responsible for a specific functionality of the framework. They are designed to be compatible with any web development framework. The modular approach is important for the following reasons:

- **Extendability**: The modular approach makes it easy to extend the framework with new functionalities, allowing for the addition of new features without altering existing ones.
- **Maintainability**: Modules form meaningful units, making it easy to maintain and update the framework. The structure is easily readable, enabling targeted updates or maintenance efforts.
- **Reusability**: The reusability of modules allows them to be implemented in other projects. This section proposes the role each module should have in the framework, facilitating extrapolation to other web development frameworks.

The modules are designed to be specific to both the culling and the LDBIM viewer. The theoretical model proposed in this thesis is informed by practical experiences and observations gained through the development of the prototype.

The main module, the viewer, and its requirements (Section 4.1) have already been discussed in Chapter 4.

## 5.1 Data fetching

This section discusses the modules responsible for retrieving external data. Two submodules can be identified: the SPARQL fetcher and the database fetcher. The SPARQL

fetcher handles communication with the RDF database and SPARQL endpoint, while the database fetcher manages communication with external databases where geometry files may be stored. The primary function of these modules, with respect to data fetching, involves authentication and error handling for requests.

Authentication and error handling are crucial when facilitating communication between multiple web instances, such as the website, database, and SPARQL endpoint. By incorporating these functionalities into separate modules, the framework can be adapted to suit the requirements of each use case. This allows for different authentication methods and error handling procedures to be employed, depending on the technology used by each web instance.

### 5.1.1 SPARQL fetcher

This module handles the back and forth communication with the RDF database and SPARQL endpoint. It should do this on two occasions:

- When the SPARQL query is updated, fetching the results of the query from the SPARQL endpoint. This includes the metadata of every given entity provided by the SPARQL endpoint for the cache manager.
- When the entities are approved by the cache manager, fetching the geometry literals from the RDF database, via the SPARQL endpoint.

As mentioned above, the SPARQL fetcher retrieves data from the SPARQL endpoint and is not an endpoint itself. This is done to offload the querying from the viewer, which is a resource-intensive task. It also implies that this module is responsible for the authentication of the connection, error handling, and result interpretation. Metadata about entities is dispatched from this module to the cache manager, waiting for approval before a new query is sent for the retrieval of geometry literals. Once loaded, these literals are dispatched to the viewer for rendering.

Serving two main functions but only communicating with one web instance, the choice was made to combine the two functions into one module. This allows for sharing the same authentication and error handling procedures.

### 5.1.2 Database fetcher

This module, triggered by the cache manager, retrieves the geometry files from an external file server/database. Similar to the SPARQL fetcher, it is responsible for authentication and error handling. It is also responsible for the interpretation of the results, dispatching the geometry files to the viewer for rendering.

## 5.2 Cache manager

The cache manager is responsible for managing the cache. It is the module that decides which entities are to be cached and which are to be removed from the cache and subsequently from the viewer. This module is crucial for the performance of the framework, as it determines which entities are to be added and which are to be removed from the viewer. It is also responsible for maintaining the integrity of the metadata about entities in the viewer, ensuring that the cache is not filled with outdated metadata.

It therefore evaluates if results from the SPARQL fetcher already exist in the viewer/-cache, and if so, whether they are up to date. If the results are not up to date or propose a new entity, the cache manager will trigger the fetching of the new geometry by the appropriate fetcher. The cache manager is also responsible for keeping track of the entities in the viewer and removing them when they are no longer relevant or outdated.

This last functionality requires the use of a cache replacement algorithm. The algorithm determines which entities are to be removed from the viewer, based on a set of rules such as: the number of entities in the viewer, the time since the entity was last viewed, the number of times the entity was viewed, etc. To illustrate the functionality, the following section will discuss the Least Recently Used (LRU) algorithm. This algorithm was chosen because of its simplicity and ease of implementation. It is also a popular algorithm for cache replacement and is therefore a good starting point for the development of the framework. Some other possible algorithms are, for example, the Least Frequently Used (LFU), First In First Out (FIFO), Last In First Out (LIFO), and Most Recently Used (MRU) algorithms(Ali, 2021). These algorithms are not discussed in this thesis but are possible candidates for future research, as they may be better suited for the framework.

### 5.2.1 LRU algorithm

As a possible cache replacement algorithm, the LRU algorithm limits the number of entities allowed in the viewer. It can be interpreted as a list in which new entities are added at the front of the list, and existing entries are also moved back to the front. The addition of a new entity thus shifts the existing entities back in the list. When an existing entity is viewed again, it moves back to the front of the list without shifting the tail of the list. When the list is full, the entity at the end of the list is removed from the viewer, as it is the least recently used entity (Ali, 2021).

To implement this algorithm, the cache manager needs to store the metadata of each new entity, or if the entity already exists, compare the new metadata about it and move it to the front of the list, triggering the fetching modules.

## 5.3 Query processing

The query processing module proposed by this thesis comprises multiple sub-modules aiming to assemble the SPARQL query from different sources, such as the UI, culling algorithm, and other user-specific settings.

### 5.3.1 Query builder

To facilitate the construction of the SPARQL query, the query builder module is responsible for assembling the query from the different requirements stated by the culling algorithms, thereby constructing a query that complies with the various algorithms in one. It concatenates the abstractions stated by these algorithms into one coherent query, reducing the load on the SPARQL fetcher.

The sub-functions or culling algorithms of this module need access to the rest of the framework. As illustrated in Sections 4.3, 4.4, and 4.5, it will require communication with the viewer to retrieve the camera position and execute 3D operations.

### 5.3.2 Query composer

As the UI interactions can override the query generated by the building module, the query composer module is responsible for combining the query from the query builder with the query parameters from the UI. Performing a final check on the query, it ensures that the query is valid and ready to be sent to the SPARQL fetcher.

By separating this module from the builder, the builder can focus on the construction of the culling query, while the composer can concentrate on the combination of the query with the UI parameters apart from the culling query.

## 5.4 Interactions

All the modules come together in one flexible framework illustrated in Figure 5.1. Four main modules create the proposed framework. The query processing module, once fed with data from both the cache manager and viewer, concatenates the different sub-modules to the query composer, creating a SPARQL query. This query creation triggers a first request to the SPARQL endpoint, which returns metadata about the needed entities. Once approved by the cache manager, to avoid constantly reloading the geometry data, it gets dispatched to the SPARQL or database fetcher. The retrieved geometry data is then sent to the viewer for rendering. Afterward, the cycle starts again, feeding viewer and cache manager data to the dynamic culling query algorithms.

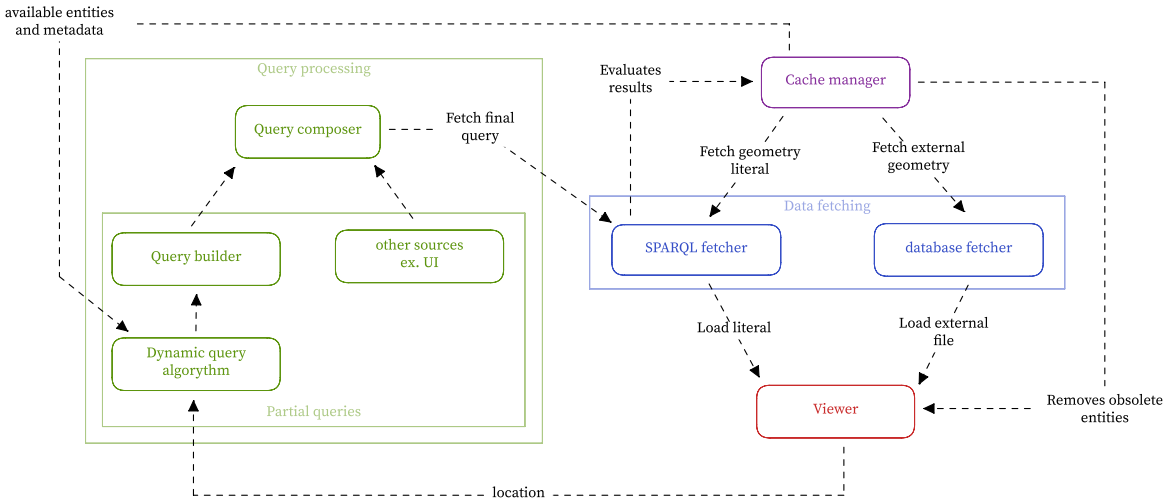



Figure 5.1: Conceptual diagram of the interactions between the modules.

# Chapter 6

# Prototype

The code for the prototype developed in this thesis is available at:
https://github.com/flol3622/LDBIM-viewer.

## 6.1 Database design

The database used in this thesis is based on the database developed by Mads Holten Rasmussen and made available at:
https://github.com/MadsHolten/BOT-Duplex-house/tree/master/Model%20files/LBD

It was chosen as a starting point, as it firstly contained BOT descriptions and relations between entities which are heavily utilized by this thesis' algorithm. Secondly, it was selected because it already contained geometry descriptions of those entities. And lastly, because it also contained WKT serializations of walls, rooms, and floor entities. Although the size of the model it describes is small, too small to evaluate culling performance, no other suitable database meeting the requirements was found.

Nevertheless, some changes had to be made to accommodate the needs of this thesis. This section will describe these changes.

### 6.1.1 GraphDB

To host the database, and exercise the role of the SPARQL endpoint, a GraphDB instance was run locally to store, query, modify, and extract the semantic data. "GraphDB is an enterprise ready Semantic Graph Database, compliant with W3C Standards."[17] It was chosen for its ease of use and installation, its support for the GeoSPARQL extension (Section 4.3), as well as the possibility to create custom functions in JavaScript (Section 4.5).

[17] Ontotext, 2022a.

To comply with the requirements of the algorithm, the structure of the database had to be altered to resemble the one illustrated in Figure 4.1. Figure 6.1 illustrates the changes made to the database, where the existing data was used to describe new relations needed for this thesis.

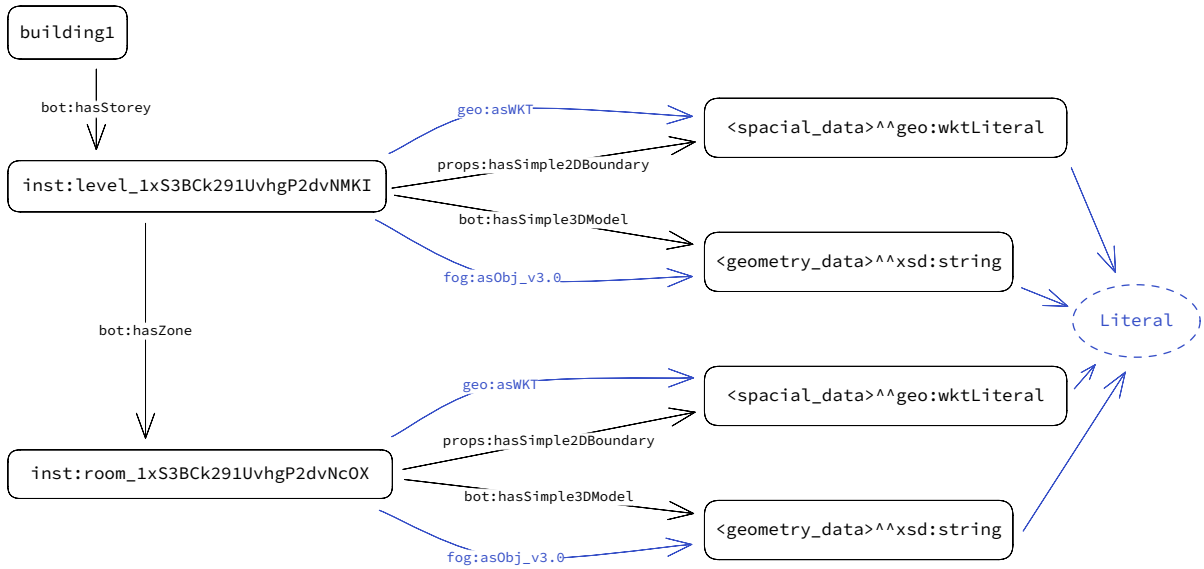



Figure 6.1: Database adaptation to Figure 4.1.

### 6.1.2 3D geometries

Listing 6.1 demonstrates how the FOG ontology was implemented in the RDF database. The following subsections detail the additional steps taken to adapt the database specifically to the requirements of the viewer used.

```
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>

INSERT { ?s fog:asObj_v3.0 ?o . }
WHERE { ?s bot:hasSimple3DModel ?o . }
```

Listing 6.1: Inserting FOG relations into the database.

#### 6.1.2.1 Extracting geometries from RDF database

Due to the inability to display OBJ literals in the utilized viewer, the geometries had to be extracted from the database and stored as separate files. This was accomplished using a Python script (Appendix B.2), which extracted the geometries from the database using Listing 6.2 and stored them as separate files.

```
SELECT ?s ?o
WHERE {
    SERVICE <http://localhost:7200/repositories/duplex-v1> {
    ?s fog:asObj ?o .
    filter(datatype(?o)=xsd:string)
    }
}
```

Listing 6.2: SPARQL query to extract geometries.

#### 6.1.2.2 Rescaling geometries

Rescaling the entities to fit the viewer's coordinate system was performed using the open-source 3D modeling software, Blender. This software allowed for the batch import, rescale, and export of the different entities. However, batch export into multiple files was restricted to the STL file format. The individual files were then uploaded to the GitHub repository of the prototype, which functions as an external database.

#### 6.1.2.3 Inserting data

The new external files were then referenced in the database using the SPARQL query in Listing 6.3. This query designated the literal's datatype as `xsd:anyURI` (Section 3.1.3.2), which the prototype would later use to treat the resource as a link to an external file (Section 5.1.2).

```
PREFIX fog: <https://w3id.org/fog#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

INSERT { ?s fog:asStl_v1.0-ascii ?newURI }
WHERE {
    ?s fog:asObj_v3.0 ?o .
    FILTER (datatype(?o) = xsd:anyURI)
    BIND (REPLACE(STR(?s), "https://172.16.10.122:8080/projects/1001/",
    ↪  "") AS ?localName)
    BIND (REPLACE(STR(?localName), "%", "-") AS ?newLocalName)
    BIND (CONCAT("https://raw.githubusercontent.com/flol3622/LDBIM-viewer
    ↪  /main/public/assets/duplex-v1/stl_m/", ?newLocalName, ".stl") AS
    ↪  ?uriString)

    BIND (STRDT(?uriString, xsd:anyURI) AS ?newURI)
}
```

Listing 6.3: Inserting external links with FOG

### 6.1.3 WKT serialisation

Similarly to the geometries, the WKT literals were implemented using the GeoSPARQL ontology, as shown in Listing 6.4.

```
PREFIX props: <https://w3id.org/props#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

INSERT { ?s geo:asWKT ?o . }
WHERE{ ?s props:hasSimple2DBoundary ?o . }
```

Listing 6.4: Inserting WKT literals using GeoSPARQL.

#### 6.1.3.1 Syntax correction

A Python script was created (see Appendix B.1) to experiment with the WKT literals. This script visualized the WKT literals using the matplotlib package. This revealed syntax errors in the WKT literals, which were corrected using the query in Listing 6.5 to enable the use of GeoSPARQL functions. This query modifies the WKT literals from the format: `"POLYGON (x1 y1, x2 y2 ,...)"^^geo:wktLiteral` to: `"POLYGON ((x1 y1, x2 y2 ,...))"^^geo:wktLiteral`. The adjustment adds an extra pair of parentheses to the POLYGON definition, adhering to the correct syntax for a WKT Polygon literal as defined by the GeoSPARQL standard.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

DELETE { ?room geo:asWKT ?old_wkt . }
INSERT { ?room geo:asWKT ?new_wkt . }
WHERE {
    ?room geo:asWKT ?old_wkt .
    BIND(str(?old_wkt) as ?old_lit)
    FILTER (
        STRSTARTS(?old_lit, "POLYGON (") &&
        !STRSTARTS(?old_lit, "POLYGON ((")
    )
    BIND (STRDT(CONCAT("POLYGON (", SUBSTR(?old_lit, 9), ")"),
    ↪  geo:wktLiteral) AS ?new_wkt
    )
}
```

Listing 6.5: Fixing WKT syntax.

## 6.2 Web Development Stack

The prototype was developed using the hrefhttps://nextjs.org/Next.js framework, which is built on top of React, a popular JavaScript library for building component-based web applications. This choice adheres to the modular approach explained in Chapter 5. Next.js and the following tools or libraries were chosen for their popularity and ease of use within the React ecosystem:

- Recoil: As a state management library, Recoil was used to manage and exchange data between the different modules. It provides a simple and efficient way to share state across components.
- Tailwind CSS: Adopted for styling the UI components, Tailwind CSS facilitates rapid prototyping and easy customization of the UI.
- Radix Icons: The Radix Icons library provided the icons used in the UI components.
- Material UI: This popular React UI library was employed for the more complex UI components. Material UI offers a large library of pre-built components that are easy to use and customize, aligning with the best practices for web UI design.

### 6.2.1 Specialized packages

#### 6.2.1.1 lru_map

The efficient LRU algorithm used in the caching algorithm is the lru_map package by Rasmus Andersson. This package was chosen for its simplicity and efficiency, as it implements a double-linked list which prevents the need for shifting the list when an item is accessed. This reduces the load on the lightweight viewer, ensuring optimal performance.

#### 6.2.1.2 fetch-sparql-endpoint

The fetch-sparql-endpoint package, developed by Ruben Taelman, a Web postdoctoral researcher at IDLab, Ghent University – imec[18], is used in the prototype's SPARQL fetcher module to manage communication with the SPARQL endpoint. This package facilitates the process of sending queries to and receiving responses from the endpoint, providing a simple and effective way to interact with the RDF database.

### 6.2.2 Xeokit-SDK

Following Malcolm et al., 2021 and their Linked Building Data (LBD) Server, the prototype of this thesis utilizes the Xeokit SDK, described as an "open source 3D graphics

[18]Taelman, 2023.

SDK from xeolabs for BIM and AEC"[19]. This Software Development Kit (SDK) supports an extensive array of 3D formats employed in the AEC industry, as detailed in Table 6.1. It also supplies the necessary tools to construct the viewer module in accordance with the requirements described in Section 4.1.

| | | |
|---|---|---|
| BCF Viewpoints | IFC | GLTF |
| CityJSON | OBJ | STL |
| 3DXML | XKT | LAS |

Table 6.1: Supported formats in Xeokit SDK.[20]

## 6.3 Structure

The prototype's design, translated into the conceptual modular approach discussed in Chapter 5, is depicted in Figure 6.2. The four main modules are further dissected into components developed using the previously mentioned tools and libraries. These fall into one of three categories:

- **React components:** These are the UI components that are rendered in the browser. They display output elements and/or receive user input.
- **React hooks:** These are functions used to manage the state of the application. They store and update data, which is then passed to the React components.
- **JavaScript functions:** These are standalone functions used to perform specific tasks, such as fetching or computing data.

In the case of the GeoSPARQLauto module, which executes a culling technique based on the WKT serialization as proposed in Section 4.3, the query processing module receives localisation data from the viewer module. This data passes through the useAutomations hook, which manages the different culling algorithms. The QueryPanel component allows selection between the culling algorithms and the user-defined query from the QueryInput component. This selector stores the final query, which triggers the useLoadGeometry hook from the viewer module. This hook uses the getEntities function to fetch data from the SPARQL endpoint, which then communicates with the evalLRU function from the cache manager for entity approval. Upon approval, the useLoadGeometry hook fetches the geometry data using the getGeometry function, retrieving a geometry literal or an URI. It then sends it to the viewer for loading into the scene. The loading occurs within Xeokit's loading plugins, which accept literal values or URIs and fetch the external files independently. Once all the entities are loaded, the

[19] xeolabs, 2021.
[20] xeolabs, 2021

viewer's scene is compared to the updated LRU list using the syncViewer function to remove obsolete entities.

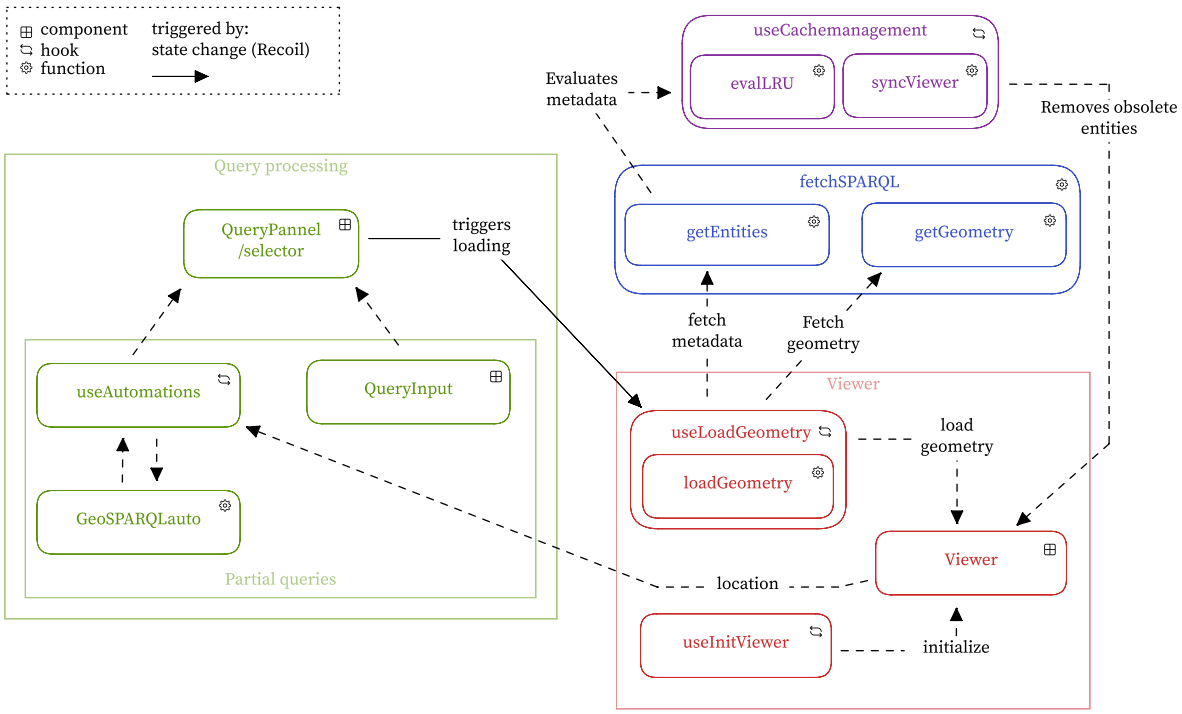



Figure 6.2: Conceptual diagram of the interactions between the modules within the prototype when using a dynamic query algorithm, such as GeoSPARQLauto, based on Figure 5.1.

## 6.4 Semantic visualisation

The cache management module is constructed such that the LRU list stores metadata about the entities, with the key of each entry representing its URI and the value being a JavaScript object containing, in the current state of the prototype, the datatype and format of the geometry. Therefore, this object can be extended to accommodate properties such as the non-geometrical ones described in Section 4.2, as illustrated in its type description in Listing 6.6.

```
export type MetadataLRU = {
    format: Format;
    datatype: Datatype;
    botType?: string;
    [key: string]: string | undefined;
  };
```

Listing 6.6: Type of the LRU entries, extendible to include additional metadata.

```
{ "key": "https://172.16.10.122:8080/projects/1001/wall_1xS3BCk291UvhgP2d
↪  vNrGN",
  "value": {
    "format": "https://w3id.org/fog#asStl",
    "datatype": "http://www.w3.org/2001/XMLSchema#anyURI",
    "botType": "https://w3id.org/bot#Element",
    "color": "[0.0, 1.0, 0.0]" }}
```

Listing 6.7: Example of an entry in the LRU list.

To illustrate this proposal, a triple containing color data was inserted into the database, with the query used shown in Listing 6.8. The entry in Listing 6.7 displays the metadata stored in the LRU list for this entity. The color data is then applied to the geometry using the Xeokit Application Programming Interface (API). Since the latter does not allow color assignment during entity loading, the color is applied after the entity is loaded, within the syncViewer function (see Appendix A.2).

```
PREFIX inst: <https://172.16.10.122:8080/projects/1001/>
PREFIX flupke: <https://flupke.archi/thesis/#>

INSERT DATA {
        inst:wall_1xS3BCk291UvhgP2dvNrGN flupke:color "[0.0, 1.0, 0.0]"
}
```

Listing 6.8: Insertion of color data into the database.

# Chapter 7

# Conclusion and future work

This thesis has explored the feasibility of culling geometry from an LDBIM graph for streaming to lightweight viewers. The goal was to minimize the volume of data within the viewer itself, as existing in-viewer culling techniques encounter processing limitations when the scene continues to expand in size, particularly when managing non-active parts of the scene. Two primary questions were therefore posed:

1. To what extent can LDBIM geometry be culled to be streamed to lightweight viewers? (Section 1.2.1)
2. Can existing semantics and ontologies be utilized to inform potential culling algorithms? (Section 1.2.2)

Both questions were initially explored by reviewing the current state of the art in the field of Semantic Web and culling algorithms. This evolved into a hands-on approach, where the development of a prototype led to new insights.

## 7.1 Object-level Culling

Employing existing ontologies such as FOG and OMG enabled the definition of geometry at the object level, which identified the smallest units that can be culled . As a result, this thesis did not address culling techniques such as back-face culling, as these are handled by the viewer itself, not the database. These ontologies, although not specifically explored in this thesis, also allow the linking of auxiliary geometry files, such as texture maps, to the entities. Future research could explore the possibilities of implementing these auxiliary geometry data in the viewer, utilizing the OMG level 2 data pattern, thereby enhancing the visual quality of the resulting scene.

## 7.2 Culling Algorithms

Three culling algorithms were proposed, each performing culling operations in radically different ways, showcasing the possibilities of culling by constructing SPARQL queries.

The first algorithm (Section 4.3) utilizes the WKT serialization of BOT rooms to evaluate GeoSPARQL functions. This approach highlighted the lack of 3D operations in GeoSPARQL, indicating a crucial need in the AEC industry for a standard for 3D spatial operations in SPARQL, which, as of now, does not exist.

The second algorithm (Section 4.4) leveraged the viewer's 3D engine capabilities to perform 3D operations not feasible with the first algorithm. It utilized raytracing to determine the room in which the observer was located. Although this algorithm offloaded 3D operations from the database to the viewer, which appears counter-intuitive to the thesis's objective, the operations were optimized to use as little computing power as possible by performing the raytracing only on a subset of rooms in the scene.

The third algorithm (Section 4.5) proposes a method to implement 3D operations in the form of string operations within the SPARQL query itself. As SPARQL string operations are limited, a custom JavaScript function was added to the SPARQL endpoint. However, the implementation is specific to the endpoint used in this thesis, GraphDB, as no standard exists for custom functions in SPARQL endpoints. The developed function is also limited to the analysis of the OBJ geometry format, but similar functions could be developed for other formats.

All options utilized the BOT ontology to zoom out from the object-level to cull at the room-level, leveraging the inherent underlying hierarchy of BIM models which is described in the graph using the BOT ontology. It was found to produce coherent results and reduce the computational resources needed for the culling. However, this hierarchy is only important when inside the building and becomes irrelevant when outside the building. Further research could explore the potential of culling algorithms for scenarios outside the building.

## 7.3 Modular approach and prototype

The modular approach (Chapter 5) presented in this thesis' prototype (Chapter 6) was found to be successful in showcasing the feasibility of culling geometry from an LDBIM graph. It also highlighted the need for further research in each step of the process, from the cache management to the moment when the query has to run again. But showcases a strong foundation for future development and research.

The main features of the prototype are showcased at:
https://github.com/flol3622/Pre-culling_LDBIM#demo
The following features are presented:

- **Basic use case:** The basic use case is presented where the entire building is loaded with a maximum number of entities set to 20. The user can navigate through the building and entities are loaded on demand. The user can also change the maximum number of entities to be loaded at once.
- **Different sources and formats:** The prototype can load different sources and formats. A database of abstract shapes is selected which holds STL, OBJ and GLTF geometry files as literals or as links to external files on GitHub. (Section 4.1)

  To achieve this goal, the existing ontologies FOG and OMG were used. However, these were stretched to their limits, as newer ontologies and semantics are needed to fully leverage the potential and versatility of Linked Data. The developed prototype highlighted the need for additional data about each geometry description, such as orientation and scale, which can differ between file formats.
- **Semantic-driven filtering:** With the aid of semantics, the user can filter the entities to load in the viewer. This example filters the entities to only show the ones with `rdf:type prod:Window`.
- **Semantic-driven colorization:** Semantics associated with each entity can be used to colorize the entities in the viewer. This example colorizes the entities based on their `flupke:color` property which holds a color code. (Sections 4.2.2, 6.4)
- **In-query, OBJ-string analytics** A first culling algorithm is showcased where the `bot:Space` entities are filtered based on their OBJ definition, which is analyzed by the endpoint using a string operation. The cache management mechanism is also visualized as the viewer's scene does not exceed an entity count of 20. (Section 4.5)
- **In situ, GeoSPARQL** This second culling algorithm uses GeoSPARQL functions. As the GeoSPARQL functions are limited to 2D space, the viewer hovering over a `bot:Space` triggers the loading of this last. (Section 4.3)

The source code is available at: https://github.com/flol3622/LDBIM-viewer

## 7.4 Database

The database used in this thesis was based on the work of Mads Holten Rasmussen, which was found to be relatively easy to adapt to the needs of this thesis as it already contained geometrical (obj geometry), spatial (WKT serialisation) and realtaional (BOT relations) data. Despite its usefullness, the limited size prevented the evaluation of the performance of the culling algorithms on larger datasets. The lack of larger datasets

was also found to be a general problem in the field of LDBIM, as no other was found during this research. Besides its size, the dataset was also found to be missing some relations, such as the adjacent spaces and the floors associated with the rooms. These relations are important for the culling algorithms as they are used to zoom out from the object-level to the room-level.

## 7.5 Conclusion

This thesis has shown that culling LDBIM graphs to reduce the size of the scene a lightweight viewer has to manage in order to visualise a building model stored in a database using Semantic Web technologies is possible. It also presented multiple approaches to perform the culling, each with its own advantages and disadvantages, as well as a modular approach to implement the whole process in a web viewer. This technology has proven to be a viable solution to the demanding needs of the AEC industry when visualizing large BIM models. And it presents a strong foundation to expand upon in a diverse set of use-cases and scenarios in need of a 3D visual representation.

# References


Ali, S. (2021). *Cache replacement algorithm*. https://arxiv.org/abs/2107.14646

BIMForum. (2015). *Level of development specification*. https://bimforum.org/lod

Bittner, J., Wimmer, M., Piringer, H., & Purgathofer, W. (2004). Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, *23*, 615–624. https://doi.org/10.1111/J.1467-86

Bonduel, M., Wagner, A., Pauwels, P., Vergauwen, M., & Klein, R. (2019). Including widespread geometry formats in semantic graphs using rdf literals. http://lib.ugent.be/catalog/pug01:8633665

Cohen-Or, D., Chrysanthou, Y., Silva, C., & Durand, F. (2003). A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, *9*, 412–431. https://doi.org/10.1109/TVCG.2003.1207447

Holten Rasmussen, M., Lefrançois, M., Bonduel, M., Anker Hviid, C., & Karlshøj, J. (2018). Opm: An ontology for describing properties that evolve over time. *CEUR Workshop Proceedings*, *2159*, 24–33.

Johansson, M., & Roupé, M. (2009). *Efficient real-time rendering of building information models*. https://www.researchgate.net/publication/220758081

Johansson, M., Roupé, M., & Bosch-Sijtsema, P. (2015). Real-time visualization of building information models (bim). *Automation in Construction*, *54*, 69–82. https://doi.org/10.1016/j.autcon.2015.03.018

Malcolm, A., Werbrouck, J., & Pauwels, P. (2021). Lbd server : Visualising building graphs in web-based environments using semantic graphs and gltf-models. http://lib.ugent.be/catalog/pug01:8692815

Mattausch, O., Bittner, J., & Wimmer, M. (2008a). Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum*, *27*, 221–230. https://www.academia.edu/14388994

Mattausch, O., Bittner, J., & Wimmer, M. (2008b). Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum*, *27*, 221–230. https://www.academia.edu/14388994

Perry, M., & Herring, J. (2012). *Ogc geosparql - a geographic query language for rdf data* (OGC Implementation Standard 11-052r4). Open Geospatial Consortium. http://www.opengis.net/doc/IS/geosparql/1.0

Rasmussen, M. H., Lefrançois, M., Schneider, G., & Pauwels, P. (2020). Bot: The building topology ontology of the w3c linked building data group. *Semantic Web*, *11*, 1–20. https://www.researchgate.net/publication/342802332

Schulz, O., Oraskari, J., & Beetz, J. (2021). Bcfowl: A bim collaboration ontology. https://technical.buildingsmart.org/projects/opencde-api/

Werbrouck, J. (2018). Linking data : Semantic enrichment of the existing building geometry. http://lib.ugent.be/catalog/rug01:002494740

# Referenced websites


BIMcollab. (2023). *What is bcf?* https://www.bimcollab.com/en/resources/openbim/about-bcf/

Bonduel, M., Wagner, A., & Pauwels, P. (2020). *Fog: File ontology for geometry formats*. https://mathib.github.io/fog-ontology/

buildingSMART International. (2023). *Bim collaboration format (bcf)*. https://technical.buildingsmart.org/standards/bcf/

Carrol, J. J., & Pan, J. Z. (2006). *Xml schema datatypes in rdf and owl*. https://www.w3.org/TR/swbp-xsch-datatypes/#sec-xmls-dt

LBD-CG. (2022). *A list of ontologies related to linked building data*. https://github.com/w3c-lbd-cg/ontologies

Linietsky, J., Manzur, A., & the Godot community. (2023). *Portal — documentation de godot engine (stable) en français*. https://docs.godotengine.org/fr/stable/classes/class_portal.html

OGC. (2023a). *Geosparql - a geographic query language for rdf data*. https://www.ogc.org/standard/geosparql/

OGC. (2023b). *Open geospatial consortium*. https://www.ogc.org/

Ontotext. (2022a). *Ontotext graphdb - get the best rdf database for knowledge graphs*. https://www.ontotext.com/products/graphdb/

Ontotext. (2022b). *What is sparql?* https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/

Rasmussen, M. H., & Schlachter, A. (n.d.). *Ld-bim - bim meets linked data*. https://ld-bim.web.app/

Taelman, R. (2023). *Ruben taelman*. https://www.rubensworks.net/

Unity. (2023). *Manual: Memory in unity webgl*. https://docs.unity3d.com/2023.2/Documentation/Manual/webgl-memory.html

W3C. (2015a). *Semantic web: Inference*. https://www.w3.org/standards/semanticweb/inference

W3C. (2015b). *Semantic web: Linked data*. https://www.w3.org/standards/semanticweb/data

W3C. (2015c). *Semantic web: Query*. https://www.w3.org/standards/semanticweb/query

W3C. (2023). *Linked building data community group*. https://www.w3.org/community/lbd/

xeolabs. (2021). *Xeokit: Web programming toolkit for aec graphics*. http://xeokit.io

# Appendix A

# Prototype

```
src/
├── components/
│   ├── index.ts
│   ├── Viewer.tsx
│   ├── Navbar.tsx
│   ├── QueryPannel.tsx
│   ├── Queryinput.tsx
│   ├── Divider.tsx
│   └── Button.tsx
├── modules/
│   ├── atoms/
│   │   ├── index.ts
│   │   ├── cleanStart.ts
│   │   ├── endpoint.ts
│   │   ├── query.ts
│   │   ├── lru.ts
│   │   └── ui.ts
│   ├── automations/
│   │   ├── index.ts
│   │   ├── useAutomations.ts
│   │   ├── BOT.ts
│   │   ├── GeoSPARQL.ts
│   │   └── OBJ.ts
│   ├── viewer/
│   │   ├── index.ts
│   │   ├── types.ts
│   │   ├── useInitViewer.ts
│   │   └── useLoadGeometry.ts
│   ├── fetchSPARQL.ts
│   ├── useCacheManagement.ts
│   └── refTypes.ts
└── pages/
    └── index.tsx
```

## A.1 automations

### A.1.1 useAutomations

```
import { useEffect } from "react";
import { useRecoilValue, useSetRecoilState } from "recoil";
import { autoMode, query } from "~/modules/atoms";
import { RefLRU, RefViewer } from "../refTypes";
import GeoSPARQLauto from "./GeoSPARQL";
import OBJauto from "./OBJ";

export default function useAutomations(viewer: RefViewer, LRU: RefLRU):
↪  void {
  const mode = useRecoilValue(autoMode);
  const setQuery = useSetRecoilState(query);

  useEffect(() => {
    let automation: NodeJS.Timer | undefined;

    switch (mode) {
      case "GEO":
        automation = GeoSPARQLauto(viewer, setQuery);
        break;
      case "BOT":
        // BOTauto(viewer, setQuery);
        break;
      case "OBJ":
        automation = OBJauto(viewer, setQuery);
        break;
      default:
        console.log("manual mode");
    }

    return () => {
      if (automation) clearInterval(automation);
    };
  }, [mode]);
}

export {};
```

### A.1.2 BOT

```
import { Viewer } from "@xeokit/xeokit-sdk";
import { RefLRU, RefViewer } from "../refTypes";

function findRoom(viewer: RefViewer, lru: RefLRU, position: number[]) {
  const down = [0, -1, 0];
  const up = [0, 1, 0];

  const rooms = <string[]>[];
  lru.current?.forEach((value, key) => {
    if (value.botType === "Space") {
      rooms.push(key);
    }
  });

  function pick(direction: number[]) {
    return viewer.current?.scene.pick({
      origin: position,
      direction: direction,
      pickSurface: true,
      includeEntities: rooms,
    });
  }

  const resultDown = pick(down);
  const resultUp = pick(up);

  if (resultDown?.entity === resultUp?.entity) {
    return resultDown?.entity?.id;
  }
}

export default function BOTauto(
  viewer: React.MutableRefObject<Viewer | undefined>,
  setQuery: (query: string) => void
): NodeJS.Timer {
  return setInterval(() => {
    const eye = viewer.current?.scene.camera.eye;
    if (eye && eye[0] && eye[2]) {
      const xcoord = Math.round(eye[0] * 1000).toString();
      const ycoord = Math.round(-eye[2] * 1000).toString();
      console.log(xcoord, ycoord);
      setQuery(`
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
PREFIX inst:<https://172.16.10.122:8080/projects/1001/>

SELECT ?entity ?fog_geometry ?datatype
WHERE {
  {
    ?entity geo:asWKT ?entityWKT .
    FILTER(geof:sfWithin("POINT(${xcoord} ${ycoord})", ?entityWKT))
  }
  UNION
  {
    # elements in the room
    ?room rdf:type bot:Space .
    ?room geo:asWKT ?roomWKT .
    FILTER(geof:sfWithin("POINT(${xcoord} ${ycoord})", ?roomWKT))

    # get elements in the room
    ?room bot:containsElement|bot:adjacentElement ?entity .
  }
  FILTER NOT EXISTS { ?entity rdf:type bot:Space }
  ?entity ?fog_geometry ?geometryData .
  FILTER(?fog_geometry IN (fog:asStl))
  BIND(DATATYPE(?geometryData) AS ?datatype)
  FILTER(?datatype = xsd:anyURI)
}
LIMIT 20
          `);
    }
  }, 1000);
}
```

### A.1.3 GeoSPARQL

```
import { Viewer } from "@xeokit/xeokit-sdk";
import { RefViewer } from "../refTypes";

export default function GeoSPARQLauto(
  viewer: RefViewer,
  setQuery: (query: string) => void
): NodeJS.Timer {
  return setInterval(() => {
    const eye = viewer.current?.scene.camera.eye;
    if (eye && eye[0] && eye[2]) {
      const xcoord = Math.round(eye[0] * 1000).toString();
      const ycoord = Math.round(-eye[2] * 1000).toString();
      console.log(xcoord, ycoord);
      setQuery(`
PREFIX bot: <https://w3id.org/bot#>
```

```
PREFIX fog: <https://w3id.org/fog#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX inst:<https://172.16.10.122:8080/projects/1001/>

SELECT ?entity ?fog_geometry ?datatype
WHERE {
  {
    ?entity geo:asWKT ?entityWKT .
    FILTER(geof:sfWithin("POINT(${xcoord} ${ycoord})", ?entityWKT))
  }
  UNION
  {
    # elements in the room
    ?room rdf:type bot:Space .
    ?room geo:asWKT ?roomWKT .
    FILTER(geof:sfWithin("POINT(${xcoord} ${ycoord})", ?roomWKT))

    # get elements in the room
    ?room bot:containsElement|bot:adjacentElement ?entity .
  }
  FILTER NOT EXISTS { ?entity rdf:type bot:Space }
  ?entity ?fog_geometry ?geometryData .
  FILTER(?fog_geometry IN (fog:asStl))
  BIND(DATATYPE(?geometryData) AS ?datatype)
  FILTER(?datatype = xsd:anyURI)
}
LIMIT 20
          `);
    }
  }, 1000);
}
```

### A.1.4 OBJ

```
import { RefViewer } from "../refTypes";

export default function OBJauto(
  viewer: RefViewer,
  setQuery: (query: string) => void
): NodeJS.Timer {
  return setInterval(() => {
    const eye = viewer.current?.scene.camera.eye;
    if (eye && eye[0] && eye[2] && eye[1]) {
      const xcoord = Math.round(eye[0] * 1000).toString();
      const ycoord = Math.round(-eye[2] * 1000).toString();
      const zcoord = Math.round(eye[1] * 1000).toString();
      setQuery(`
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX inst: <https://172.16.10.122:8080/projects/1001/>
PREFIX jsfn: <http://www.ontotext.com/js#>

SELECT ?entity ?fog_geometry ?datatype ?botType
WHERE {
    BIND("${xcoord},${ycoord},${zcoord}" as ?position)

    # select the room
    ?room a bot:Space .
    ?room fog:asObj ?obj .
    BIND(DATATYPE(?obj) AS ?type)
    FILTER(?type = xsd:string)
    FILTER(jsfn:insideObjAABBox(?obj, ?position))

    # select the entities in the room
    ?room bot:containsElement|bot:adjacentElement ?entity .
    ?entity ?fog_geometry ?geometryData .
    FILTER(?fog_geometry IN (fog:asStl))
    BIND(DATATYPE(?geometryData) AS ?datatype)
    FILTER(?datatype = xsd:anyURI)
    ?entity rdf:type ?botType .
    FILTER(STRSTARTS(str(?botType), "https://w3id.org/bot#"))
}
LIMIT 20
        `);
    }
  }, 1000);
}
```

## A.2 useCacheManagement

```
import { LRUMap } from "lru_map";
import { useEffect } from "react";
import { useRecoilValue } from "recoil";
import { cleanStart, lruLimit } from "./atoms";
import { RefLRU, RefViewer } from "./refTypes";
import { Datatype, Format } from "./viewer/types";

export type MetadataLRU = {
  format: Format;
  datatype: Datatype;
  botType?: string;
};
export type EntryLRU = {
  id: string;
  metadata: MetadataLRU;
};

export default function useCacheManagement(viewer: RefViewer, LRU:
↪  RefLRU) {
  const clean = useRecoilValue(cleanStart);
  const limit = useRecoilValue(lruLimit);

  // initialize the LRU cache
  useEffect(() => {
    console.log("cache initialized");
    LRU.current = new LRUMap(limit);
    LRU.current.clear();
  }, [limit]);

  // clear the cache when the clean prop changes
  useEffect(() => {
    LRU.current?.clear();
    console.log("cache cleared");
  }, [clean]);

  // evaluate need to add to Viewer, move entity to head of LRU
  function evalLRU(entity: EntryLRU): boolean {
    const lruValue = LRU.current?.get(entity.id);
    if (lruValue != undefined) {
      if (JSON.stringify(lruValue) != JSON.stringify(entity.metadata)) {
        viewer.current?.scene.models[entity.id]?.destroy();
        return true;
      } else {
        return false;
      }
    }
    LRU.current?.set(entity.id, entity.metadata);
```

```
    return true;
  }

  function syncViewer(): void {
    const modelIds = viewer.current?.scene.modelIds;
    if (!modelIds) return; // if no models
    for (const id of modelIds) {
      if (!LRU.current?.has(id))
↪   viewer.current?.scene.models[id]?.destroy();
    }
  }

  return { evalLRU, syncViewer }; // to use when loading a new model
}
```

# Appendix B

# Python scripts

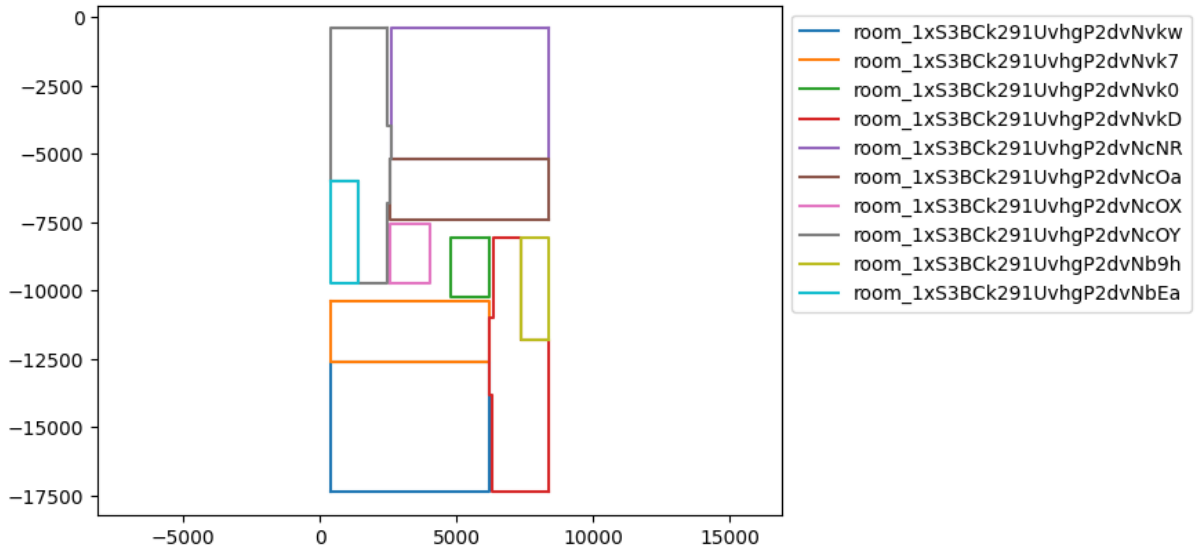
## B.1 WKT visualisation

```
import matplotlib.pyplot as plt
import rdflib

levels = [
    "inst:level_1xS3BCk291UvhgP2dvNMKI",
    "inst:level_1xS3BCk291UvhgP2dvNMQJ",
    "inst:level_1xS3BCk291UvhgP2dvNsgp",
    "inst:level_1xS3BCk291UvhgP2dvNtSE"
]

data = {}

for level in levels:
    g = rdflib.Graph()
    qres = g.query("""
      PREFIX props: <https://w3id.org/props#>
      PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
      PREFIX bot: <https://w3id.org/bot#>
      PREFIX inst: <https://172.16.10.122:8080/projects/1001/>
      SELECT ?s ?o
      WHERE {
        SERVICE <http://localhost:7200/repositories/duplex-v1> {
          ?s props:hasSimple2DBoundary ?o .
          ?s rdf:type bot:Space .
          """ + str(level) + """ bot:hasSpace ?s.
        }
      }
      LIMIT 100
      """)
    rooms = {}
```

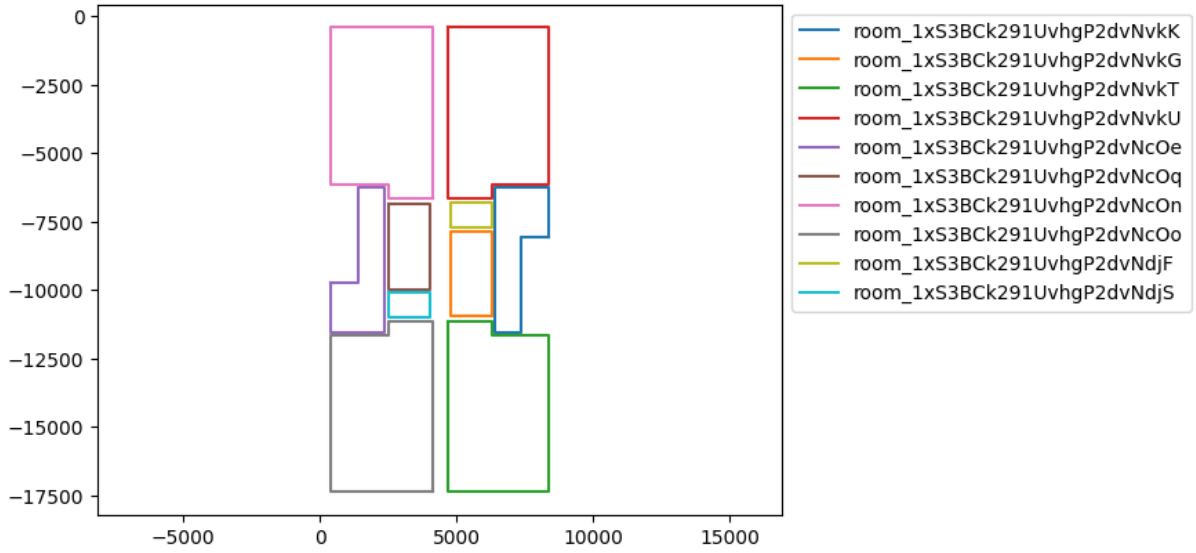
```
    for row in qres:
        polygon_string = row.o.n3()
        polygon_string = polygon_string.replace('POLYGON (',
        ↪  '').replace('()', '').replace(')', '').replace(
            '"', '').replace('^^<http://www.opengis.net/ont/geosparql#wkt
            ↪  Literal>',
            ↪  '')
        coordinate_pairs = polygon_string.split(', ')
        coordinates = [(int(pair.split(' ')[0]), int(pair.split(' ')[1]))
                       for pair in coordinate_pairs]
        row.o.n3()
        room = row.s.n3().replace(
            '<https://172.16.10.122:8080/projects/1001/',
            ↪  '').replace('>', '')
        rooms[room] = coordinates
    data[level] = rooms


def showLevel(level):
    level = data[level]
    plt.figure()
    for room, polygon in level.items():
        xs, ys = zip(*polygon)
        plt.plot(xs, ys, label=room)
    plt.axis('equal')
    plt.legend(bbox_to_anchor=(1, 1))
    plt.show()
```

```
for level in levels:
    print(level)
    print(data[level].keys())
    showLevel(level)
```

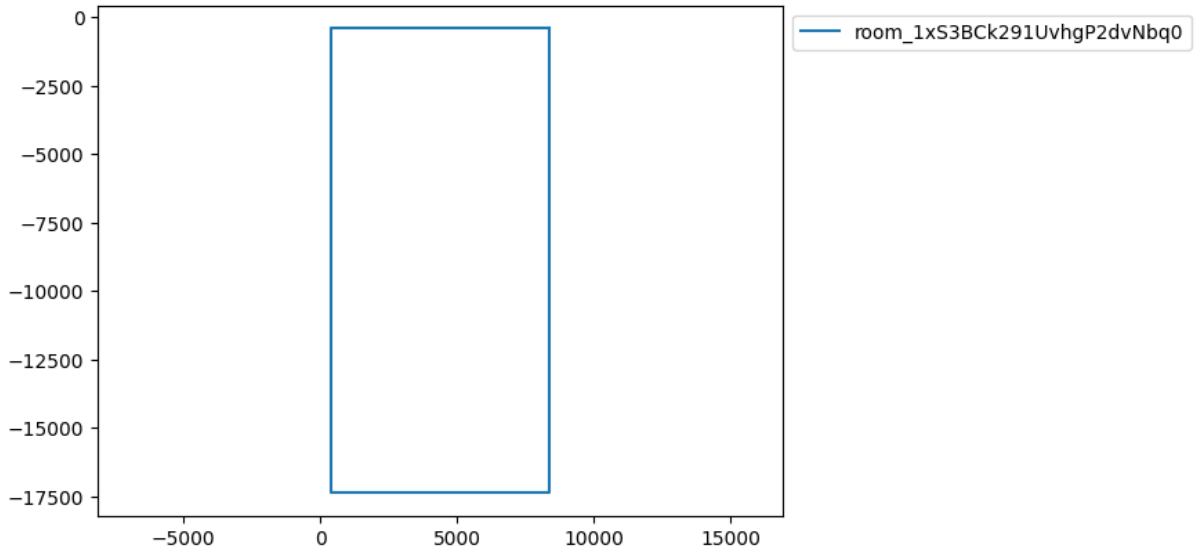
```
texttt{inst:level_1xS3BCk291UvhgP2dvNMKI
dict_keys(['room_1xS3BCk291UvhgP2dvNvkw', 'room_1xS3BCk291UvhgP2dvNvk7',
↪  'room_1xS3BCk291UvhgP2dvNvk0', 'room_1xS3BCk291UvhgP2dvNvkD',
↪  'room_1xS3BCk291UvhgP2dvNcNR', 'room_1xS3BCk291UvhgP2dvNcOa',
↪  'room_1xS3BCk291UvhgP2dvNcOX', 'room_1xS3BCk291UvhgP2dvNcOY',
↪  'room_1xS3BCk291UvhgP2dvNb9h', 'room_1xS3BCk291UvhgP2dvNbEa'])}
```



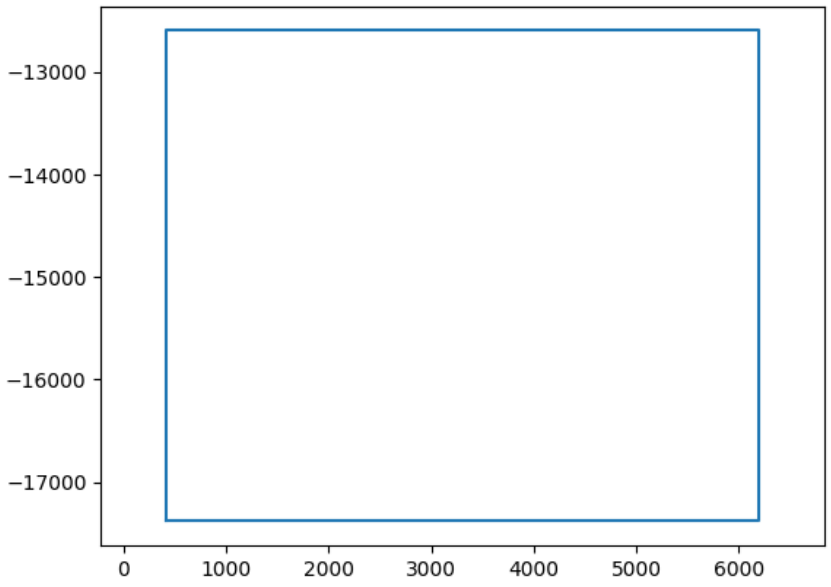

```
inst:level_1xS3BCk291UvhgP2dvNMQJ
dict_keys(['room_1xS3BCk291UvhgP2dvNvkK', 'room_1xS3BCk291UvhgP2dvNvkG',
↪  'room_1xS3BCk291UvhgP2dvNvkT', 'room_1xS3BCk291UvhgP2dvNvkU',
↪  'room_1xS3BCk291UvhgP2dvNcOe', 'room_1xS3BCk291UvhgP2dvNcOq',
↪  'room_1xS3BCk291UvhgP2dvNcOn', 'room_1xS3BCk291UvhgP2dvNcOo',
↪  'room_1xS3BCk291UvhgP2dvNdjF', 'room_1xS3BCk291UvhgP2dvNdjS'])
```




```
inst:level_1xS3BCk291UvhgP2dvNtSE
dict_keys(['room_1xS3BCk291UvhgP2dvNbq0'])
```




```
def showRoom(level, room):
    polygon = data[level][room]
    plt.figure()
    xs, ys = zip(*polygon)
    plt.plot(xs, ys)
    plt.axis('equal')
    plt.show()
showRoom("inst:level_1xS3BCk291UvhgP2dvNMKI",
↪  "room_1xS3BCk291UvhgP2dvNvkw")
```



## B.2 Geometry extraction

```
import rdflib

g = rdflib.Graph()
qres = g.query("""
PREFIX props: <https://w3id.org/props#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?s ?o
WHERE {
  SERVICE <http://localhost:7200/repositories/duplex-v1> {
    ?s fog:asObj ?o .
    filter(datatype(?o)=xsd:string)
  }
}
# LIMIT 10
""")

for row in qres:
    name = row.s.replace("https://172.16.10.122:8080/projects/1001/",
    ↪  "")
    name = name.replace("%", "-") # for github user content
    extention = ".obj"
    directory = "out/"
    fileName = directory + name + extention
    file = open(fileName, "w")
    file.write(row.o)
    file.close()
```