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

# Contents

# List of Figures



# List of Tables

# List of Listings

# Short abstract

This is my short abstract.

# Abstract

This is my abstract

# 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 full 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 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. This volume 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. This

[1] W3C, 2023.

makes visual querying, or simply put, 3D exploration of models, an increasingly difficult task. Especially when looking at newer devices used in the industry such as mobile phones, and tablets, which are becoming more and more powerful, but still have limited computational resources in comparison to office computers.

To bring this volume of geometric data in perspective, Table 1.1 shows the size of the test-models used in Johansson et al., 2015 , a study from 2015 on the performance of BIM viewers for large models with the following description:

" 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 for 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. 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 Figures 1.1 and 1.2.

Figure 1.1 illustrates the basic idea of this thesis, presenting an extra step in the communication between a user, represented here by a Hand Held Device (HHD), and a database storing 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. In addition to the culling, Web of Data technologies behind LDBIM allow for a more flexible approach to data management. The expressive capabilities of SPARQL Protocol and RDF Query Language (SPARQL) enable complex and fine-grained queries, in contrast to current BIM approaches, for data retrieval. This offers a broad range of

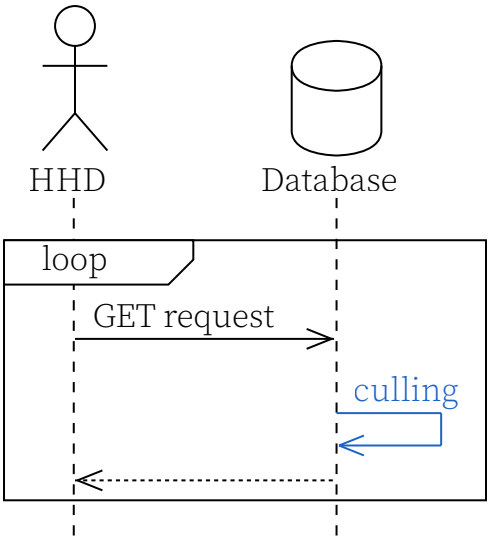


Figure 1.1: Sequence diagram - basic concept

end-use cases tailored to multiple stakeholders. In this thesis, this translates to user or application-specific query adaptation capabilities.

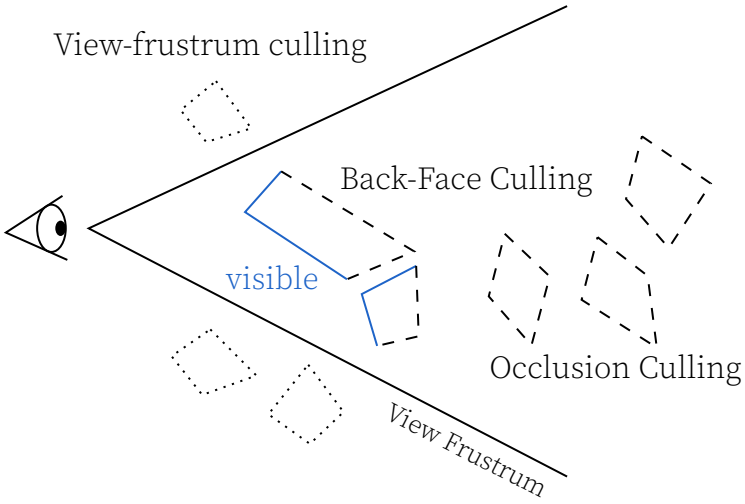


Figure 1.2: Illustration of culling principle, based on Cohen-Or et al., 2003

Figure 1.2 showcases multiple culling techniques to showcase 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, and not whole objects, are facing away from the user.

## 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 described in section 3.3, the research questions in this thesis concentrate on assessing the feasibility of introducing such algorithms in LDBIM. It aims to propose a set of possible solutions tailored to this specific problem, 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?

This thesis focuses on computing with data snippets or triples inside a LDBIM model, rather than within. This means that the smallest unit of data that can be culled is the one described in a single triple, which is, in the most likely scenario, a single Level of Detail (LOD) of a single element. It implies that geometry is defined and separated at the object level. Furthermore, this thesis will not address culling techniques such as back-face culling, as these will be left to the viewer itself, not the database.

Determining the necessary data snippets for the viewer is a key question. The fundamental requirements for the viewer include, first, the geometry itself, which involves selecting the appropriate geometry format for the application as well as the additional visual information such as color, texture, etc. Second, the identifier of each element is of crucial importance in order to maintain the link to other semantic resources in the graph. This enables the viewer to retrieve those resources for a multitude of use cases, transforming it into a user-friendly visual query tool.

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

Unlike the computer graphics industry, this interconnected context already contains both explicit and implicit relationships within the graph, the latter being derived through inferencing. This is similar to Johansson and Roupé, 2009 and their paper where they utilized the semantics of a BIM model in Industry Foundation Classes (IFC) format to develop culling techniques. However, this thesis will concentrate 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 is an evolution of the data *management* layer. "Linked Data", 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.

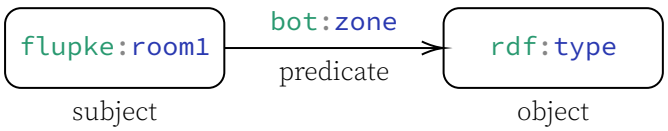


Figure 2.1: Triple structure

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 predicate, which is referred to as an edge. Multiple triples can thus create and link

[2] W3C, 2015b.

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/AR-Linked-BIM-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/AR-Linked-BIM-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.[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, each with their own database.

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

## 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.

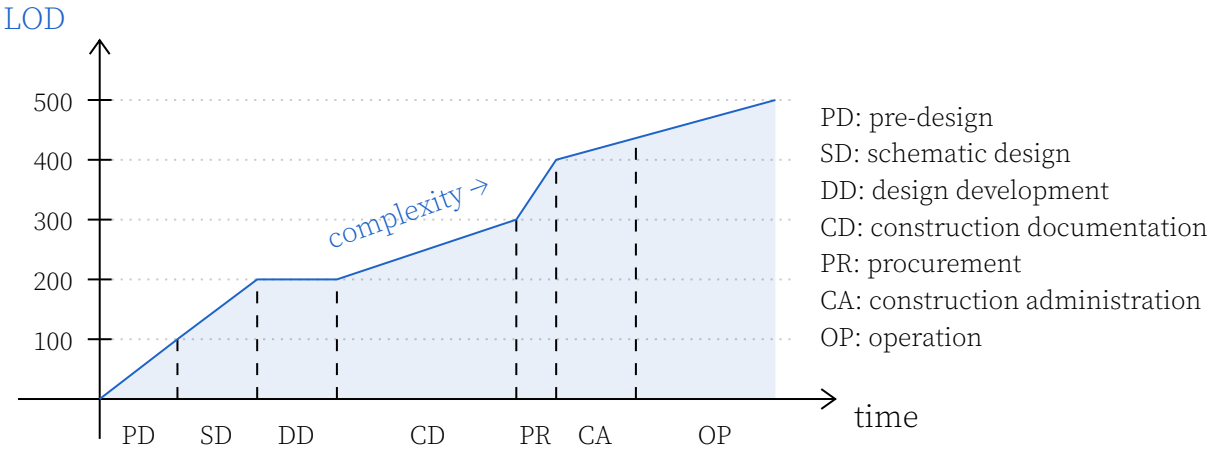


Figure 2.2: Evolution of LOD during the life-cycle of a building.
Based upon the Macleamy Curve (Ilozor & Kelly, 2012)
randow values at the moment

### 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 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 serves as this thesis's goal, 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:Storey`, 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).

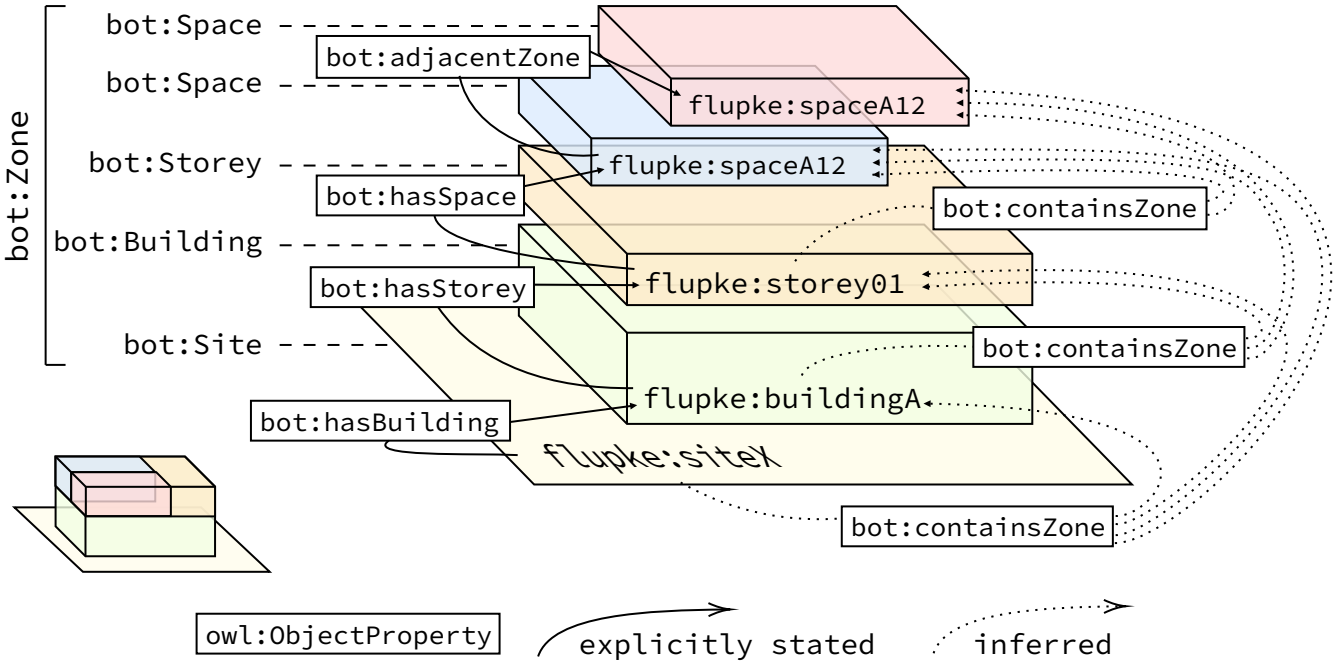


Figure 3.1: Illustration of the BOT ontology, based on Rasmussen et al., 2020.

#### 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 Extensible Markup Language (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 1.1, the external database, from which some files are expected to be fetched. This is illustrated in Figure 3.2.

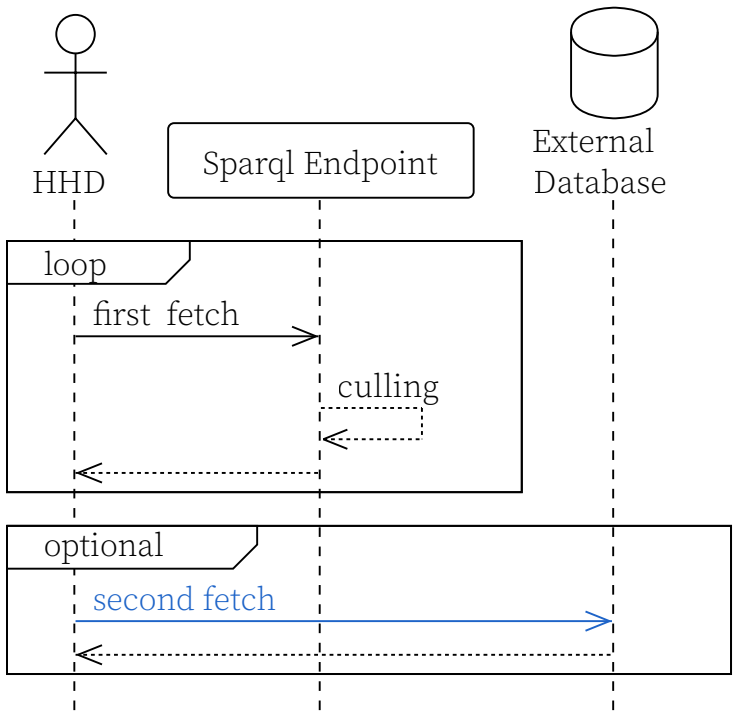


Figure 3.2: Sequence diagram with external database - concept

[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.

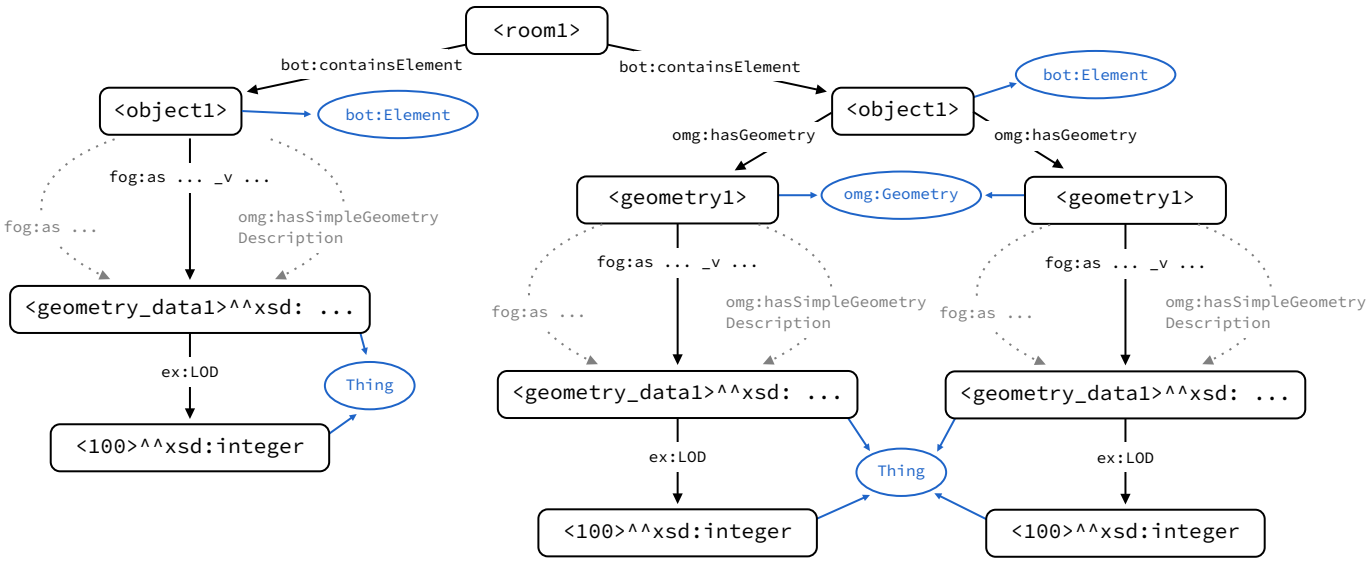


Figure 3.3: Illustration of Level 1 (left) and Level 2 (right) of the FOG and OMG ontologies, based on Bonduel et al., 2019. LOD can't be assigned to literal, needs to be changed

### 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]

As multiple triplestores and SPARQL endpoints support the GeoSPARQL extension, it is a viable candidate for spatial and LOD culling algorithms. Such algorithms require spa-

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

tial 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)<br>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 Random Access Memory (RAM), as discussed in 1.2.

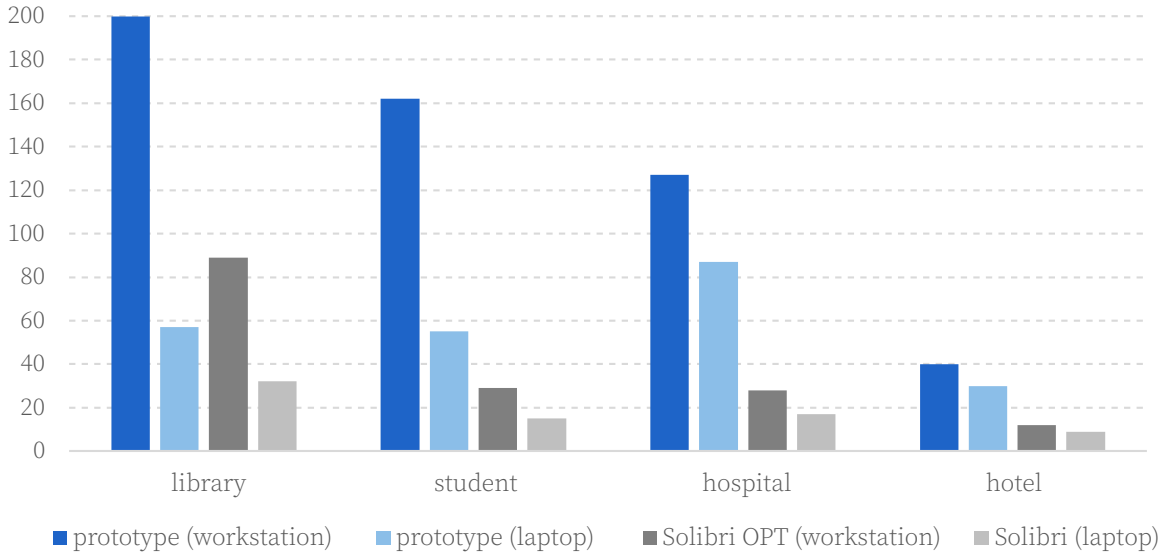


Figure 3.4: Comparison in rendering performance.
from Johansson et al., 2015

## 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] "BIM Collaboration Format (BCF) - buildingSMART Technical", n.d.
[16] "What is BCF - BIMcollab", n.d.

### 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 `?location` 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 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.

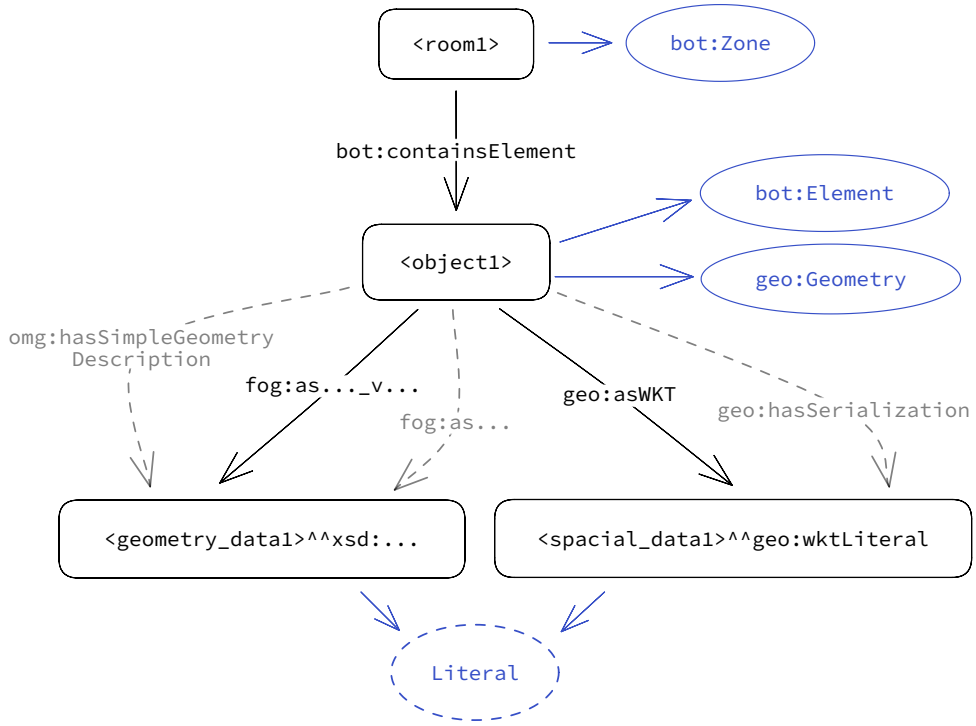


Figure 4.1: Serialization of a geometry using the OMG level 1 pattern.

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.

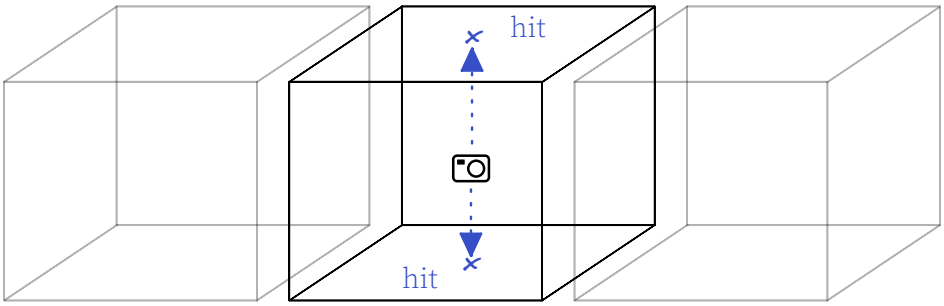


Figure 4.2: In viewer ”bot:Space” identification, with raytracing

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("(PDF) Cache Replacement Algorithm", n.d.). 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 ("(PDF) Cache Replacement Algorithm", n.d.).

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.

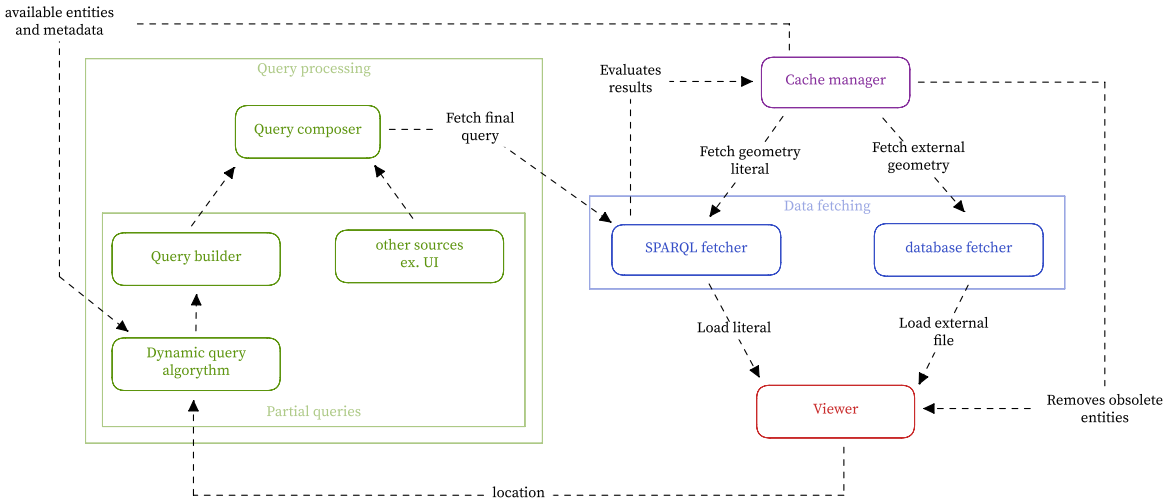


Figure 5.1: Conceptual diagram of the interactions between the modules.

## 5.5 Sequences

# Chapter 6

# Prototype

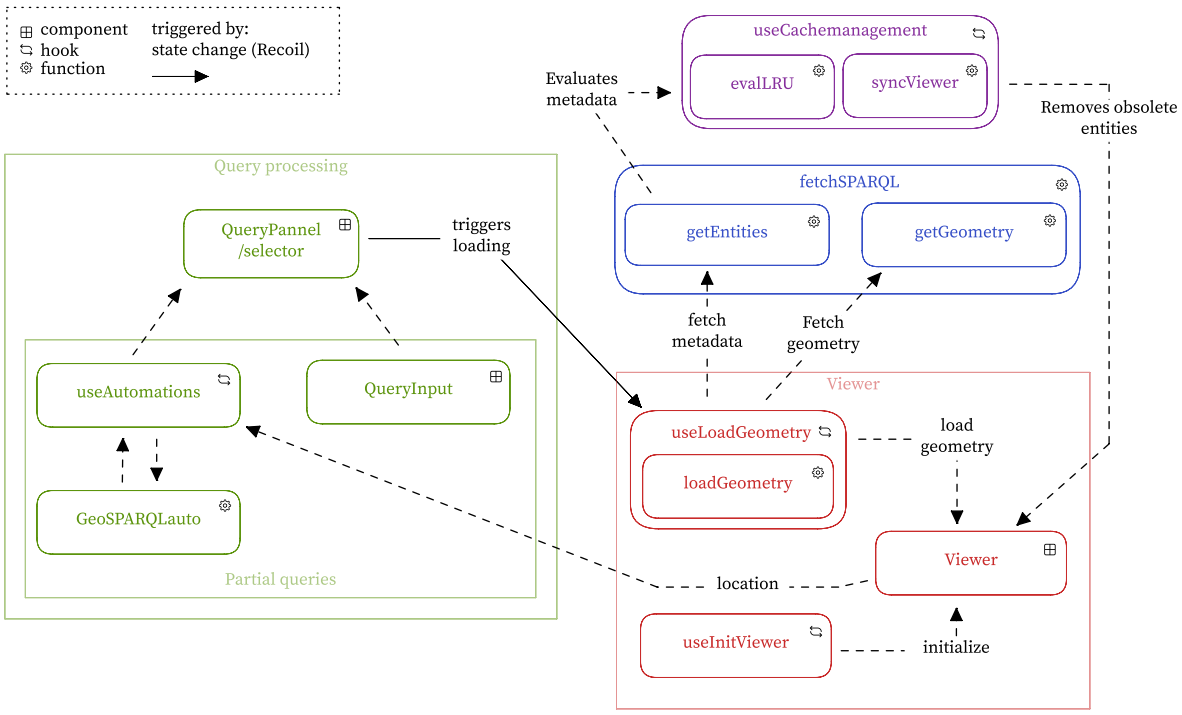


Figure 6.1: Conceptual diagram of the interactions between the moduwithin the prototype, based on Figure 5.1.

# Chapter 7

# Conclusion

# List of Acronyms

# References


*(pdf) cache replacement algorithm*. (n.d.). https://www.researchgate.net/publication/353635256_Cache_Replacement_Algorithm

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.

Ilozor, B. D., & Kelly, D. J. (2012). Building information modeling and integrated project delivery in the commercial construction industry: A conceptual study. *Journal of Engineering, Project, and Production Management*, *2*, 23–36. https://doi.org/10.32738/JEPPM.201201.0004

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

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.

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

# Referenced webistes


*Bim collaboration format (bcf) - buildingsmart technical*. (n.d.). https://technical.buildingsmart.org/standards/bcf/

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

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. (2022). *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/

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/

*What is bcf - bimcollab*. (n.d.). https://www.bimcollab.com/en/resources/openbim/about-bcf/

# 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 Components

### A.1.1 Viewer

```tsx
import { Viewer } from "@xeokit/xeokit-sdk";
import { LRUMap } from "lru_map";
import { useRef } from "react";
import { useAutomations } from "~/modules/automations";
import { MetadataLRU } from "~/modules/useCacheManagement";
import { LoaderType, useInitViewer, useLoadGeometry } from
↪  "~/modules/viewer";

export default function ARViewer() {
  const viewerRef = useRef<Viewer>();
  const loaderTypesRef = useRef<LoaderType>();
  const LRU = useRef<LRUMap<string, MetadataLRU>>();

  // initialize the setup
  useInitViewer(viewerRef, loaderTypesRef);

  // fetch the input query
  useLoadGeometry(viewerRef, loaderTypesRef, LRU);

  // automatic querries
  useAutomations(viewerRef, LRU);

  function test() {
    if (viewerRef.current) {
      viewerRef.current.scene.camera.eye = [5, 5, -5];
    }
  }

  return (
    <>
      <canvas id="myCanvas" className="mt-2 h-full w-full"></canvas>
      <canvas
        className="fixed right-0 bottom-0 h-40 w-40"
        id="myNavCubeCanvas"
      ></canvas>
      <button className="fixed top-24 left-0 z-10" onClick={test}>
        test
      </button>
    </>
  );
}
```

### A.1.2 Navbar

```tsx
import { TextField, Tooltip } from "@mui/material";
import Autocomplete from "@mui/material/Autocomplete";
import {
  CheckIcon,
  GitHubLogoIcon,
  RocketIcon,
  TrashIcon,
} from "@radix-ui/react-icons";
import { useState, useEffect } from "react";
import { useRecoilState, useRecoilValue } from "recoil";
import {
  cleanStart,
  defaultEndpoints,
  endpoint,
  freezing,
  lruLimit,
} from "~/modules/atoms";
import Divider from "./Divider";
import Button from "./Button";

export default function Navbar() {
  const [clean, setClean] = useRecoilState(cleanStart);
  const [endpointValue, setEndpoint] = useRecoilState(endpoint);
  const [limit, setLimit] = useRecoilState(lruLimit);
  const dftEndpoints = useRecoilValue(defaultEndpoints);
  const freezingValue = useRecoilValue(freezing);
  const [tempEndpoint, setTempEndpoint] = useState("");
  const [tempLimit, setTempLimit] = useState<number>(0);

  useEffect(() => {
    setTempLimit(limit);
  }, []);

  const handleTempLimit = (event: React.ChangeEvent<HTMLInputElement>) =>
  ↪  {
    setTempLimit(parseInt(event.target.value));
  };

  const updateEndpoint = () => {
    setEndpoint(tempEndpoint);
    console.log("updated endpoint:", endpointValue);
  };

  const updateLimit = () => {
    setLimit(tempLimit);
  };

  const handleKeyDownEndpoint = (event: React.KeyboardEvent) => {
```

```
    if (event.key === "Enter") {
      updateEndpoint();
    }
  };

  const handleKeyDownLRU = (event: React.KeyboardEvent) => {
    if (event.key === "Enter") {
      updateLimit();
    }
  };

  return (
    <div className="border-black absolute z-10 flex h-16 w-full
↪   items-center justify-between gap-2 border-b bg-white p-2 shadow">
      <div className="flex h-full flex-grow items-center gap-2">
        <Button tooltip="Clean viewer" onClick={() => setClean(!clean)}>
          <TrashIcon />
        </Button>
        <Divider />
        <Autocomplete
          key={clean.toString()}
          disabled={freezingValue}
          size="small"
          sx={{ flexGrow: 1, maxWidth: 600 }}
          freeSolo
          options={dftEndpoints}
          onInputChange={(_, input) => {
            setTempEndpoint(input);
          }}
          onKeyDown={handleKeyDownEndpoint}
          renderInput={(params) => (
            <TextField {...params} label="Type endpoint" />
          )}
        />
        <Button
          onClick={updateEndpoint}
          tooltip="Go to database"
          disabled={freezingValue}
        >
          <RocketIcon />
        </Button>
      </div>
      <Divider />
      <TextField
        type="number"
        size="small"
        label="Max. objects"
        sx={{ width: 100 }}
        value={tempLimit}
        onKeyDown={handleKeyDownLRU}
```

```
          onChange={(e) => setTempLimit(parseInt(e.target.value))}
        />
        <Button onClick={updateLimit} tooltip="Update LRU limit">
          <CheckIcon />
        </Button>
        <Divider />
        <h1 className="text-right">
          Pre-culling geometric linked building data
          <br />
          for lightweight viewers
          <br />
        </h1>
        <Button
          tooltip="info"
          href="https://github.com/flol3622/AR-Linked-BIM-viewer"
        >
          <GitHubLogoIcon height={20} width={20} />
        </Button>
      </div>
    );
}
```

### A.1.3 QueryPannel

```
import { ToggleButton, ToggleButtonGroup } from "@mui/material";
import {
  ArrowTopRightIcon,
  Cross1Icon,
  PaperPlaneIcon,
} from "@radix-ui/react-icons";
import { useRecoilState, useRecoilValue, useSetRecoilState } from
↪  "recoil";
import { QueryMode, autoMode, closeQuery, freezing, query, uiQuery } from
↪  "~/modules/atoms";
import Button from "./Button";
import Divider from "./Divider";
import QueryInput from "./Queryinput";

export default function Querypannel() {
  const [close, setClose] = useRecoilState(closeQuery);
  const [mode, setMode] = useRecoilState(autoMode);
  const freezingValue = useRecoilValue(freezing);
  const tempUiQuery = useRecoilValue(uiQuery);
  const setQuery = useSetRecoilState(query);

  const updateQuery = () => {
    setQuery(tempUiQuery);
```

```
  };

  const handleMode = (_: React.MouseEvent, newMode: QueryMode) => {
    setMode(newMode);
  };

  const changeClose = () => {
    setClose(!close);
  };

  return (
    <>
      {!close && (
        <div className="fixed bottom-4 left-4 flex w-[400px] flex-col gap-2 rounded border bg-white p-2 shadow-lg">
          <div className="flex items-center gap-2">
            <h3 className="flex-grow">Query</h3>
            <Divider />
            <div className="flex items-center gap-2">
              <p className="text-sm opacity-50">auto modes:</p>
              <ToggleButtonGroup
                color="primary"
                value={mode}
                exclusive
                onChange={handleMode}
                aria-label="Platform"
              >
                <ToggleButton size="small" value="OBJ">
                  OBJ
                </ToggleButton>
                <ToggleButton size="small" value="BOT">
                  BOT
                </ToggleButton>
                <ToggleButton size="small" value="GEO">
                  GEO
                </ToggleButton>
              </ToggleButtonGroup>
            </div>
            <Divider />
            <Button tooltip="close editor" onClick={changeClose}>
              <Cross1Icon />
            </Button>
          </div>
          <hr />
          <QueryInput />
          {freezingValue && (
            <div className="absolute flex h-full w-full items-center justify-center ">
              <p className="bg-white p-2 ">
                Query is frozen while the viewer is initializing
```

```
              </p>
            </div>
          )}
          <div className="flex w-full items-center justify-between gap-2
↪  px-2 ">
            <Button
              tooltip="update query"
              disabled={mode != null || freezingValue}
              onClick={updateQuery}
            >
              <PaperPlaneIcon />
            </Button>
            <p className="text-sm opacity-50">shortcut: Shift + Enter</p>
          </div>
        </div>
      )}
      {close && (
        <Button
          className="fixed bottom-4 left-4 rounded border bg-white
↪  shadow-lg"
          tooltip="open editor"
          onClick={changeClose}
        >
          <ArrowTopRightIcon />
        </Button>
      )}
    </>
  );
}
```

### A.1.4 Queryinput

```
import { TextareaAutosize } from "@mui/base";
import { useEffect } from "react";
import { useRecoilState, useRecoilValue } from "recoil";
import { autoMode, freezing, query, uiQuery } from "~/modules/atoms";

export default function QueryInput() {
  const [tempUiQuery, setTempUiQuery] = useRecoilState(uiQuery);
  const [queryValue, setQuery] = useRecoilState(query);
  const mode = useRecoilValue(autoMode);
  const freeze = useRecoilValue(freezing);

  useEffect(() => {
    setTempUiQuery(queryValue);
  }, [queryValue]);

```

```
  const updateQuery = () => {
    setQuery(tempUiQuery);
  };

  const handleKeyDown = (e: React.KeyboardEvent) => {
    console.log(e.key);
    if (e.key === "Enter" && e.shiftKey) {
      e.preventDefault();
      updateQuery();
    }
  };

  if (mode === null) {
    return (
      <TextareaAutosize
        minRows={4}
        aria-label="maximum height"
        placeholder="Write query"
        defaultValue={tempUiQuery}
        spellCheck="false"
        style={{
          width: "100%",
          padding: "0.5rem",
          fontFamily: '"Fira code", "Fira Mono", monospace',
          fontSize: "0.7rem",
          opacity: freeze ? 0.3 : 1,
        }}
        disabled={freeze}
        onChange={(e) => setTempUiQuery(e.target.value)}
        onKeyDown={handleKeyDown}
      />
    );
  } else {
    return (
      <p
        className="text-blue whitespace-pre-wrap p-2 text-[0.7rem]
↪ opacity-50"
        style={{ fontFamily: '"Fira code", "Fira Mono", monospace' }}
      >
        {queryValue}
      </p>
    );
  }
}
```

### A.1.5 Divider

```
export default function Divider() {
  return <div className="border-black self-stretch border-l
↪  border-dashed" />;
}
```

### A.1.6 Button

```
import { Tooltip } from "@mui/material";

type ButtonProps = {
  children: React.ReactElement;
  tooltip: string;
  className?: string;
  onClick?: () => void;
  disabled?: boolean;
  href?: string;
};

export default function Button(props: ButtonProps) {
  function base(props: ButtonProps) {
    return (
      <button
        onClick={props.onClick}
        className={`p-2 hover:bg-ugent hover:bg-opacity-20
↪  ${props.className}`}
        disabled={props.disabled}
        style={{ opacity: props.disabled ? "0.5" : "1" }}
      >
        <Tooltip title={props.tooltip}>{props.children}</Tooltip>
      </button>
    );
  }

  if (props.href) {
    return (
      <a href={props.href} target="_blank">
        {base(props)}
      </a>
    );
  } else {
    return base(props);
  }
}
```

## A.2 Modules

### A.2.1 Atoms

#### A.2.1.1 cleanStart

```
import { atom } from "recoil";

const cleanStart = atom<boolean>({
  key: "cleanStart",
  default: false,
});

export default cleanStart
```

#### A.2.1.2 endpoint

```
import { atom } from "recoil";

const endpoint = atom<string>({
  key: "endpoint",
  default: "",
});

const defaultEndpoints = atom<string[]>({
  key: "defaultEndpoints",
  default: [
    "http://localhost:7200/repositories/duplex-v1",
    "http://localhost:7200/repositories/test2",
    "http://localhost:7200/repositories/test3",
  ],
});

export { endpoint, defaultEndpoints };
```

#### A.2.1.3 query

```
import { atom } from "recoil";

const query = atom<string>({
  key: "query",
  default: `PREFIX bot: <https://w3id.org/bot#>
PREFIX fog: <https://w3id.org/fog#>
PREFIX omg: <https://w3id.org/omg#>
```

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?entity ?fog_geometry ?datatype ?botType
where {
    ?entity ?fog_geometry ?geometryData .
    BIND(datatype(?geometryData) AS ?datatype)
    FILTER NOT EXISTS { ?entity rdf:type bot:Space }
    FILTER(?fog_geometry IN (fog:asStl))
    FILTER(?datatype = xsd:anyURI)
    ?entity rdf:type ?botType .
    FILTER(STRSTARTS(str(?botType), "https://w3id.org/bot#"))
}
#ORDER BY (?element) (?fog_geometry)
LIMIT 20`,
});

const uiQuery = atom<string>({
  key: "uiQuery",
  default: "",
});

export type QueryMode = null | "BOT" | "GEO" | "OBJ";

const autoMode = atom<QueryMode>({
  key: "autoMode",
  default: null,
});

export { query, autoMode, uiQuery };
```

#### A.2.1.4 lru

```
import { atom } from "recoil";

const lruLimit = atom<number>({
  key: "lruLimit",
  default: 20,
});

export default lruLimit;
```

#### A.2.1.5 ui

```
import { atom } from "recoil";

```

```
const freezing = atom<boolean>({
  key: "freezing",
  default: true,
});

const closeQuery = atom<boolean>({
  key: "closeQuery",
  default: false,
});

export { freezing, closeQuery };
```

### A.2.2 Automations

#### A.2.2.1 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.2.2.2 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.2.2.3 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.2.2.4 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.2.3 Viewer

#### A.2.3.1 types

```ts
// for entities
export type Format =
  | "https://w3id.org/fog#asGltf"
  | "https://w3id.org/fog#asObj"
  | "https://w3id.org/fog#asStl";

export type Datatype =
  | "http://www.w3.org/2001/XMLSchema#string"
  | "http://www.w3.org/2001/XMLSchema#anyURI";
```

#### A.2.3.2 useInitViewer

```ts
import {
  GLTFLoaderPlugin,
  NavCubePlugin,
  OBJLoaderPlugin,
  STLLoaderPlugin,
  Viewer,
} from "@xeokit/xeokit-sdk";
import { useEffect } from "react";
import { useRecoilValue, useSetRecoilState } from "recoil";
import { cleanStart, freezing } from "../atoms";
import { RefLoaderTypes, RefViewer } from "../refTypes";

export type LoaderType = {
  [key: string]: {
    loader: any; // could not find a general type for loaders
    params?: any;
    litParam?: string;
    uriParam?: "src";
  };
};

export default function useInitViewer(
  viewer: RefViewer,
  loaderTypes: RefLoaderTypes
) {
  const clean = useRecoilValue(cleanStart);
  const setFreeze = useSetRecoilState(freezing);

  useEffect(() => {
    // freeze the query and endpoint inputs
    setFreeze(true);
```

```
    // clear the existing canvas
    viewer.current?.scene.clear();

    // initialize the viewer
    viewer.current = new Viewer({
      canvasId: "myCanvas",
      transparent: true,
    });

    // initialize the navcube
    new NavCubePlugin(viewer.current, {
      canvasId: "myNavCubeCanvas",
      visible: true,
    });

    // identify the scene and camera
    const scene = viewer.current?.scene;
    const camera = scene.camera;
    camera.projection = "perspective";

    // initialize the loaders
    const gltfLoader = new GLTFLoaderPlugin(viewer.current);
    const objLoader = new OBJLoaderPlugin(viewer.current, {});
    const stlLoader = new STLLoaderPlugin(viewer.current);

    // store the loaders in a ref
    loaderTypes.current = {
      "https://w3id.org/fog#asGltf": {
        loader: gltfLoader,
        params: { edges: true },
        litParam: "gltf",
        uriParam: "src",
      },
      "https://w3id.org/fog#asStl": {
        loader: stlLoader,
        params: { edges: true, rotation: [180, 0, 0] },
        litParam: "stl",
        uriParam: "src",
      },
      "https://w3id.org/fog#asObj": {
        loader: objLoader,
        uriParam: "src",
      },
    };
    console.log("viewer initialized");
    setFreeze(false);
  }, [clean]);
}
```

### A.2.3.3 useLoadGeometry

```
import { useEffect } from "react";
import { useRecoilValue } from "recoil";
import { endpoint, lruLimit, query } from "../atoms";
import { getEntities, getGeometry } from "../fetchSPARQL";
import { RefLRU, RefLoaderTypes, RefViewer } from "../refTypes";
import useCacheManagement, { EntryLRU } from "../useCacheManagement";
import { LoaderType } from "./useInitViewer";

async function loadGeometry(
  query: string,
  endpoint: string,
  loaderTypes: React.MutableRefObject<LoaderType | undefined>,
  evalLRU: (entry: EntryLRU) => boolean,
  syncViewer: () => void
) {
  await getEntities(query, endpoint, (bindings: any) => {
    try {
      const entry = {
        id: bindings.entity.value,
        metadata: {
          format: bindings.fog_geometry.value,
          datatype: bindings.datatype.value,
          botType: bindings.bot_type?.value,
        },
      } as EntryLRU;

      // cache management, stop if needed
      if (evalLRU(entry)) {
        const loaderType = loaderTypes.current?.[entry.metadata.format];
        getGeometry(entry.id, entry.metadata.format, endpoint)
          .then((data) => {
            // if the data is a literal, and is supported
            if (
              entry.metadata.datatype ===
                "http://www.w3.org/2001/XMLSchema#string" &&
              loaderType?.litParam
            ) {
              loaderType.loader.load({
                ...loaderType.params,
                id: entry.id,
                [loaderType.litParam]: data,
              });
            }
            // if the data is a uri, and is supported
            else if (
              entry.metadata.datatype ===
                "http://www.w3.org/2001/XMLSchema#anyURI" &&
              loaderType?.uriParam
```

```
            ) {
              loaderType.loader.load({
                ...loaderType.params,
                id: entry.id,
                [loaderType.uriParam]: data,
              });
            }
            // if the data source is not supported
            else console.log("unsupported / undefined data source",
 ↪  entry.id);
          })
          .catch((error) => {
            console.error("Error fetching geometry data:", error);
          });
      } else {
        console.log("Already in cache");
      }
    } catch (error) {
      console.error("Error loading geometry:", error);
    }
  });

  syncViewer();
}

export default function useLoadGeometry(
  viewer: RefViewer,
  loaderTypes: RefLoaderTypes,
  LRU: RefLRU
) {
  const limit = useRecoilValue(lruLimit);
  const queryValue = useRecoilValue(query);
  const endpointValue = useRecoilValue(endpoint);

  const { evalLRU, syncViewer } = useCacheManagement(viewer, LRU);

  useEffect(() => {
    if (endpointValue) {
      // ensures no loading on first render
      loadGeometry(queryValue, endpointValue, loaderTypes, evalLRU,
 ↪  syncViewer);
    }
  }, [endpointValue, queryValue, limit]);
}
```

### A.2.4 fetchSPARQL

```
import { SparqlEndpointFetcher } from "fetch-sparql-endpoint";
import { Format } from "./viewer/types";

const myFetcher = new SparqlEndpointFetcher();

// ----------------------------------------
// Fetch entities for a given query
// ----------------------------------------
async function getEntities(
  query: string,
  endpoint: string,
  forEachEntry: (bindings: any) => void
): Promise<void> {
  const bindingsStream = await myFetcher.fetchBindings(endpoint, query);

  bindingsStream.on("data", (bindings: any) => {
    forEachEntry(bindings);
  });

  bindingsStream.on("error", (error: any) => {
    console.error("Error fetching entities:", error);
  });
}

// ----------------------------------------
// Fetch geometry data of a given entity
// ----------------------------------------

function getGeometryQuery(id: any, format: Format): string {
  return `SELECT ?geometryData
    WHERE {
      <${id}> <${format}> ?geometryData .
    }
    LIMIT 1`;
}

async function getGeometry(id: any, format: Format, endpoint: string) {
  return new Promise(async (resolve, reject) => {
    const bindingsStream = await myFetcher.fetchBindings(
      endpoint,
      getGeometryQuery(id, format)
    );

    bindingsStream.on("data", (bindings: any): void => {
      const data = bindings.geometryData.value;
      resolve(data);
    });

```

```
    bindingsStream.on("error", (error: any): void => {
      reject(error);
    });

    bindingsStream.on("end", (): void => {
      reject(new Error("No data found"));
    });
  });
}

export { getEntities, getGeometry };
```

### A.2.5 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
}
```

### A.2.6 refTypes

```
import { Viewer } from "@xeokit/xeokit-sdk";
import { LoaderType } from "./viewer";
import { LRUMap } from "lru_map";

export type RefViewer = React.MutableRefObject<Viewer | undefined>;
export type RefLoaderTypes = React.MutableRefObject<LoaderType |
↪  undefined>;
export type RefLRU = React.MutableRefObject<LRUMap<string, any> |
↪  undefined>;
```

## A.3 Pages

### A.3.1 index

```
import dynamic from "next/dynamic";
import Head from "next/head";
import { Navbar, Querypannel } from "~/components";

const Viewer = dynamic(() => import("~/components/Viewer"), {
  ssr: false,
});

export default function Home() {
  return (
    <>
      <Head>
        <title>Create T3 App</title>
        <meta name="description" content="Generated by create-t3-app" />
        <link rel="icon" href="/favicon.ico" />
      </Head>
      <main>
        <div className="absolute top-14 h-[calc(100vh-3.5rem)] w-full
↪  overflow-hidden">
          <Viewer />
        </div>
        <Navbar />
        <Querypannel/>
      </main>
    </>
  );
}
```