BIM2Geo converter

van der Vaart, J.A.J.; Arroyo Ohori, G.A.K.; Stoter, J.E.

Publication date
2025

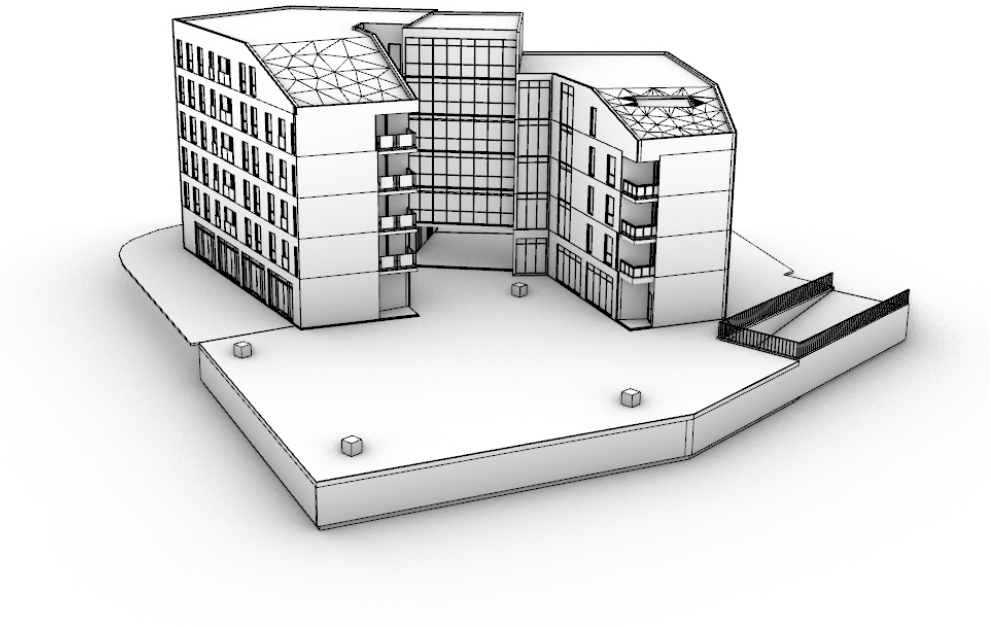
Document Version
Final published version

Citation (APA)
van der Vaart, J. A. J., Arroyo Ohori, G. A. K., & Stoter, J. E. (2025). *BIM2Geo converter*. Delft University of Technology, Faculteit Bouwkunde.

BIM2Geo converter

Final report

23 September, 2025



Jasper van der Vaart
Ken Arroyo Ohori
Jantien Stoter

This project has received funding from the European Union's Horizon Europe programme under Grant Agreement No.101058559

Contents

List of Figures

List of Tables

1 Introduction

Building information modelling (BIM) and geoinformation are widely recognised as complementary sources of data. Whereas a BIM model can represent a single building or infrastructure project in high detail, geoinformation-based sources can represent different types of features in a large region with less detail. Integrating geoinformation and BIM is very useful in practice and constitutes an active research field—often referred to as GeoBIM. A short list of GeoBIM applications include: performing checks for the issuance of building permits using buildings (BIM) and city regulations (Geo), navigation that combines outdoor (Geo) and indoor (BIM) portions, facility management for infrastructure sites (BIM) that include the regional connections between the sites (Geo), and risk management using regional simulations (Geo) that also takes into account the impact on specific sites (BIM).

Among the geoinformation datasets that are usually considered within a GeoBIM context, 3D city models stand out because they model buildings in three dimensions, making them analogous to BIM models of buildings. Because of this, there is a special interest in integrating building models of these two sources. Following the intuitive classification of Ma and Ren [2017], the integration can take place through conversions from BIM models to 3D city models, conversions from 3D city models to BIM models, or conversions of both to other data models or formats. For instance, such an integration can be used to provide neighbourhood context for the design of buildings (Geo to BIM) [Noardo et al., 2019] or input data for simulation software (joint conversion to another format) [Jusuf et al., 2017].

However, among these three types of conversions, the BIM to 3D city model case is particularly appealing since it has a wide range of potential applications, such as providing interior geometries for 3D city models and running spatial analyses to see the effect of planned—not yet constructed—buildings. Also, since BIM models are generally more geometrically and semantically detailed than 3D city models, it should be possible to compute the BIM-to-Geo conversion through an automatic abstraction process using the data that is already available in a typical BIM model. This conversion is therefore the focus of most research [Tan et al., 2023] and has substantial software support [Noardo et al., 2020]. This is unlike the opposite case (3D city model to BIM), which involves going from a less detailed to a more detailed representation—often making educated guesses about the shape and size of some elements [Salheb et al., 2020; Isailović et al., 2025].

There is a large body of research on the conversion of BIM models to 3D city models, but most methods in the literature tend to avoid applying complex geometric processing to the models—keeping the methods simpler but also limiting what they can do. For instance, it is difficult or impossible for such a method to output models that are geometrically valid (even from partially invalid input) or fully conform to a particular standard (such as a minimum distance between vertices).

In this paper, we apply geometric processing techniques to tackle the creation of building models at particular levels of detail (LoDs)—a key feature of 3D city modelling standards that enables efficient and scalable creation, processing and use in applications [Biljecki et al., 2015]. Our method starts from highly detailed BIM models in the open IFC data model and abstracts them to 9 different generalised output models that fit within the LoD frameworks in the open CityGML [Gröger et al., 2008, 2012; Kolbe et al., 2021] and CityJSON [Biljecki et al., 2016; Ledoux et al., 2019] standards.

1.1 CHEK project

This selection of output LoDs has been created based on the goals of the Horizon Europe funded CHEK (Change Toolkit for Digital Building Permits) project. In this project, a toolkit has been developed to check new and renovated buildings (modelled in BIM) against urban regulations (e.g. maximum building height) by integrating the BIM models into the 3D city models. For the integration, abstractions of the BIM models need to be derived containing those geometrical and semantic properties as needed in the regulations checking.

The users involved in the CHEK project, as other stakeholders, initially requested the output from the BIM to Geo conversion at LoD3 (either LoD3.2 or 3.3), since this is the most detailed and complete GIS output. However, LoD3 is challenging to generate from existing BIM models, since it requires extremely well structured IFC models. In addition, LoD3 models are geometrically complex. When this complex geometry is not required for downstream analysis or when the IFC model has issues, more abstracted LoD could be an alternative, more suitable, solution. To support this, the range of LoDs was expanded to the volumetric LoD2.2, 1.3, 1.2, and 1.0. These LoDs are often used in established GIS models and their abstracted geometry relies just on a selection of the input BIM model. The non-volumetric counterparts of LoD2.2, 1.3, 1.2, and 1.0 (LoD0.4, 0.3, 0.2, and 0.0 respectively) were added as well. These non-volumetric abstractions, except for LoD0.0, are created in a sub-step of the creation of the volumetric abstractions. They contain similar data when compared to their volumetric counterparts but their creation process is simpler. Therefore, these non-volumetric abstractions could function as either a file size compression method or a "fail-safe" output when the process to generate their volumetric counterparts encounters some issues.

1.2 Overview of this report

Section 2 covers the related work, including a description of the different LoD abstraction frameworks for 3D city models, a review of the BIM-to-Geo conversion methods available in the literature, and another review of different abstraction methods that can be used to generalise 3D city models.

Section 3 describes our method that creates buildings for 3D city models at different LoDs from BIM models.

Section 4 describes the implementation details of the method, which has been made into an open source prototype.

Section 5 analyses the results of testing the prototype with a variety of IFC files.

Section 6 contains the conclusions of this report.

1.3 Acknowledgements

This research has received funding from the European Union's Horizon Europe programme under Grant Agreement No.101058559.

2 Related work

2.1 3D city model LoD abstraction frameworks

The CityGML standard in its versions 1.0 [Gröger et al., 2008] and 2.0 [Gröger et al., 2012] introduced an abstraction framework consisting of five levels of detail (LoDs) ranging from LoD0 to LoD4. In the case of buildings, these LoDs correspond to a 2.5D representation of the building's footprint or roof area (roofprint) for LoD0, prismatic blocks with flat roofs for LoD1, relatively simple models with the shape of the building's roof structures and surfaces with semantics for LoD2, detailed architectural models of the building's exterior for LoD3, and detailed architectural models that also include the exterior for LoD4.

In CityGML version 3 [Kolbe et al., 2021], the abstraction framework was overhauled (Figure 1) in two main ways: (1) surface semantics can be included in any LoD, and (2) LoD4 was removed and interior geometries are now allowed in all LoDs, which includes floor plans for LoD0. As a result of the latter change, different LoDs can be used to represent the exterior and interior features of a building.

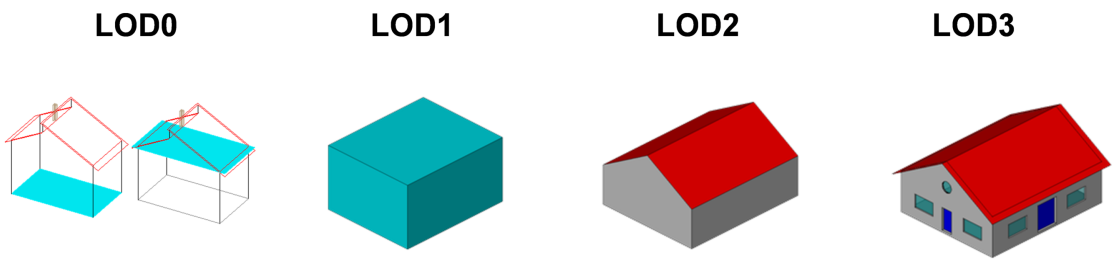


Figure 1: The four LoDs available in the CityGML 3.0 standard

Considering the large gaps between LoDs, as well as the possibility of different interpretations of single LoDs in the standard, Biljecki et al. [2016] proposed a finer-grained abstraction framework in which each LoD in the CityGML standard is split into four refined LoDs (Figure 2). By doing so, an LoDx (where $0 \leq x \leq 3$) in the original standard becomes four different levels of the form LoDx.y, where $0 \leq y \leq 3$. This LoD framework is explicitly supported in the CityJSON standard [Ledoux et al., 2019].

In this scheme, LoD0.x and 1.x follow a similar pattern, with 0.x modelling footprints and sometimes roof areas, and 1.x modelling volumes. LoD0.0 and 1.0 allow multiple buildings to be aggregated into a single geometry, LoD0.1 and 1.1 model buildings individually (without aggregation) with all their large parts, LoD0.2 and 1.2 also model smaller parts (e.g. alcoves) with roofs modelled/extruded at a single height, and LoD0.3 and 1.3 are similar but allow multiple roof surfaces at different heights. Note how LoD0.2 and 1.2, as well as 0.3 and 1.3 essentially encode the same information, as the missing walls in LoD0.x could be derived by extruding the roof surfaces down to ground level.

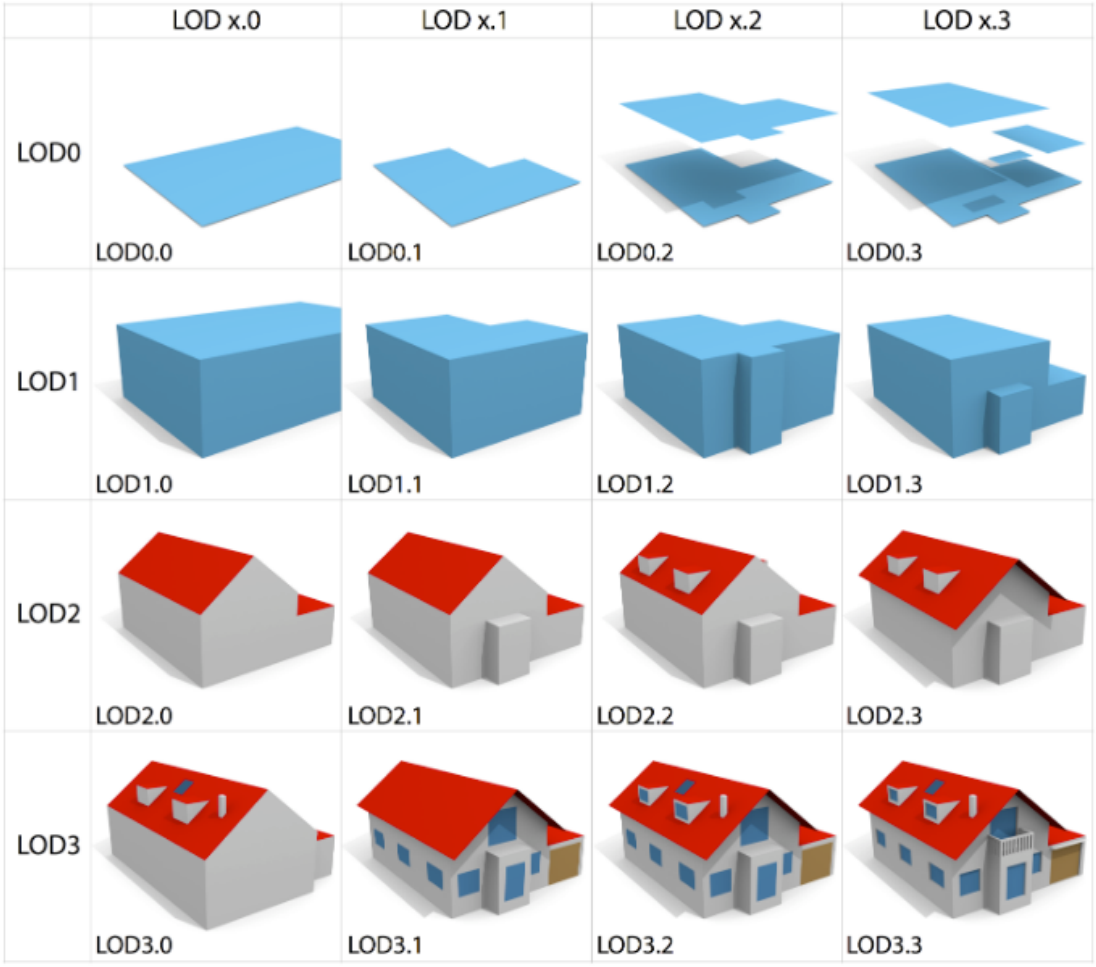


Figure 2: The 16 LoDs proposed by Biljecki et al. [2016]

In LoD2.x, the different LoDs progressively add smaller elements that can be modelled. Starting from LoD2.0, which only models large building parts, LoD2.1 adds smaller building parts, LoD2.2 adds roof superstructures (e.g. dormers), and LoD2.3 adds explicitly modelled roof overhangs.

LoD3.x is characterised by explicitly modelled openings, e.g. windows and doors, but abstraction scheme follows a different logic where the different LoDs do not have a strict increase in detail. LoD3.0 models detailed roof superstructures and their possible openings (e.g. skylights) but less detailed walls, whereas LoD3.1 is the opposite (detailed walls with openings but less detailed roofs) and models roof overhangs. These opposing modelling approaches reflect the elements that can be discerned from the two common data capture locations: from above (e.g. using satellite or aerial imagery and LiDAR) and from the ground (e.g. using terrestrial or car-mounted scanners). Finally, LoD3.2 models both detailed walls and roofs, and LoD3.3 incorporates finer details (e.g. window embrasures and awnings).

2.2 BIM to Geo conversion

The problem of automatic BIM to Geo conversion has long been studied. Most early academic efforts focused on defining a mapping between equivalent BIM and Geo classes [Isikdag and Zlatanova, 2009; El-Mekawy et al., 2012], or did a conversion of some basic geometry types [Wu and Hsieh, 2007]. At roughly the same time, closed source software (e.g. FME and IFC Explorer) introduced basic functionality to convert from IFC to CityGML, but the methods used were not documented in literature and the resulting files did not comply with the conventions and expectations of a typical CityGML file (e.g. containing non-overlapping and correctly classified semantic surfaces).

To our knowledge, the first well-documented effort to perform a BIM to Geo conversion was around the open source building information server BIMserver. Discussed in de Laat and van Berlo [2011], it introduced two related pieces of work: the CityGML LoD4 export functionality of the software and a GeoBIM ADE of CityGML. Since BIMserver can import IFC files, the export functionality could be used to perform an IFC to CityGML conversion, whereas the ADE allowed the storage of IFC semantics and properties in CityGML.

Later on, a few different methods which attempted to extract specific LoDs from BIM models were developed. For example, Donkers et al. [2016] specifically targets the creation of CityGML LoD3 models. It uses a combination of semantic mapping of equivalent CityGML and IFC classes, geometric transformations based on constructive solid geometry to merge elements, and a process of geometric and semantic refinement to obtain missing surface semantics. There is an open implementation based on the Computational Geometry Algorithms Library (CGAL).

Similarly, Stouffs et al. [2018] uses a triple graph grammar that defines a correspondence between equivalent elements in IFC and CityGML. By doing so, object graphs for both

IFC and CityGML are created and related to each other. The end result is a direct conversion of elements that should be geometrically equivalent to the input IFC model. The authors also define a CityGML Application Domain Extension (ADE) to improve the compatibility of CityGML with IFC, as well as the specificities of the Singaporean context of the study. The latter is described in more detail in Biljecki et al. [2021].

A more recent representative example is described by Lam et al. [2024], who target CityGML LoD4 specifically. Their method leverages the capabilities of existing software (FME and 3dcitydb) to do the initial conversion, validates the results using the OWL/RDF semantic web technologies and visualises the results using Cesium and Unreal Engine.

A recent review of the scientific literature of IFC to CityGML conversion is contained in Liang and Tan [2024], whereas Bello Pérez [2015] has a comprehensive review of the earliest attempts at this conversion, including descriptions of sources that apparently are no longer available online. Apart from the methods in the literature, Noardo et al. [2020] provides a summary of the capabilities and limitations of other software for this purpose. In summary, while a few different pieces of software offer this functionality, the results are not ideal: output models are generally invalid, semantic mapping is difficult and targetting specific LoDs is not realistic.

2.3 Abstraction methods

In order to actually obtain 3D city models at specific LoDs from more detailed BIM models, applying an abstraction or generalisation method is necessary. Fan et al. [2009] is an early example of an abstraction method to simplify building models in CityGML. They describe a few different techniques: extracting the outer envelope of approximately convex buildings, simplifying ground plans (useful for the bottom of building models), generalising façades, and substituting detailed windows for typified templates.

Deng et al. [2016] proposed an instance-based method to create mapping rules between equivalent IFC and CityGML classes, which are formally stored in a reference ontology called the Semantic City Model. The study focusses on the transformation of coordinates and the creation of explicit geometries. Multiple LoDs can be generated from a BIM model by selecting or removing particular types, as well as by extracting the outer envelope of a building using ray casting.

Kang and Hong [2018] uses a method that combines exterior envelope extraction and custom LoD mapping rules to extract multiple CityGML LoDs from a BIM model. The exterior envelope extraction uses a screen buffer to sample the surfaces that are visible from the exterior quickly, which can be implemented in a GPU.

Similarly, Zhou et al. [2019] use a number of different observation points known to be in the exterior of a building to extract its outer envelope. Since only the exterior surfaces

of a building are visible from these observation points, this can be used to efficiently obtain these surfaces.

Ji et al. [2024] use an ontology-based method to extract specific CityGML LoDs from an IFC file. Rule maps are used to convert coordinates (local to global), filter or extract geometries and map the semantics of the different objects.

3 Method

The methods developed for the abstraction of BIM (IFC) models to Geo (CityJSON) models at different LoDs can be split into different steps.

Before the computation of the desired LoD abstraction output, our method always starts with a preprocessing step, which reduces the data's complexity to ease the next steps. In it, the model's placement is optimised, the objects required for the processes are selected, and complex objects (doors and windows) are simplified.

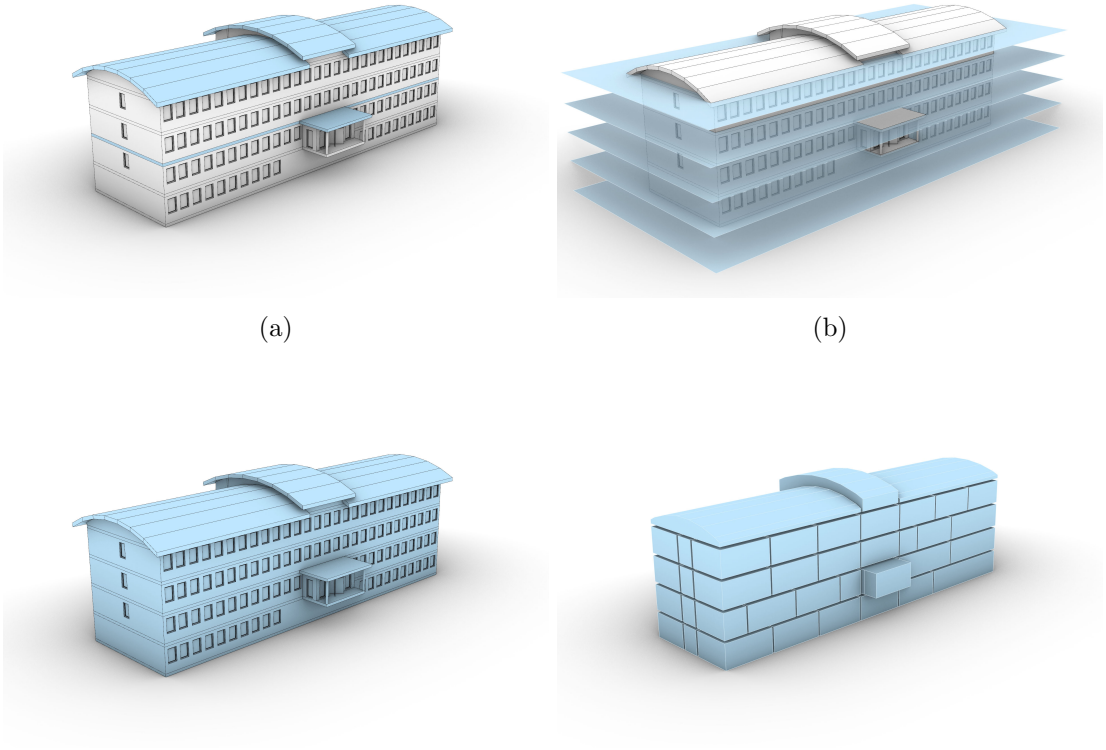
Afterwards, a voxelisation step is required for every LoD except for LoD0.0 and 1.0. Here, the voxels that are used for the coarse filtering of data in most of the downstream abstraction processes are computed.

This is followed by a series of abstraction-specific processing steps. These are the processes where the actual abstracted shapes (i.e. geometries) are created. Depending on the required output LoD geometries, certain processing steps are executed while others are omitted. Roughly, the LoD abstraction methods can be grouped into five different levels. These levels are based on how much of the geometry of the input IFC model the methods rely on. The levels and the individual LoDs generated are presented in Table 1 and the IFC elements they use are shown in Figure 3. In addition to the standard LoDs described by Biljecki et al. [2015], we consider one more useful LoD consisting of the roofing structure of LoD2.2, which we hereafter refer to as LoD0.4.

level	output	based on BIM model's
low	LoD0.0 and LoD1.0 exterior	vertices
mid	LoD0.2, 0.3, and 0.4* roof structure LoD1.2, 1.3 and 2.2 exterior	roofing geometry
high	LoD0.2 and 0.3 storeys	selective outer shell geometry
very high	LoD3.2 exterior	complete outer shell geometry
interior	LoD0.2, 1.2, 2.2, and 3.2 interior	*IfcSpaces*

Table 1: A summary of the five levels of geometry dependence of the different LoD abstractions. *Non-standard LoD

A schematic overview of the methods and their relationships can be found in Figure 4. As this overview shows, some of the output LoDs are obtained by a mix of the five methods. For example, for a full LoD0.3 model, the LoD0.3 roof structure would be created via the mid geometric dependent abstraction methods, the LoD0.3 storeys and footprint would be created via the high geometric dependent abstractions and the LoD0.3 rooms would be created via the interior geometric dependent abstractions. A more detailed description of the output LoDs that can be derived can be found in Appendix 7.1.



(a) (b)

(c) (d)

Figure 3: The elements relevant for the different abstraction methods. Low geometric dependent abstraction utilizes the vertices of the model (not displayed in this figure). a) Mid geometric dependent abstraction utilizes the roofing structure of the model. b) High geometric dependent abstraction utilizes the geometry at horizontal internals. c) Very high geometric dependent abstraction utilizes the complete outer shell. d) Interior geometric dependent abstraction utilizes the *IfcSpace* objects.

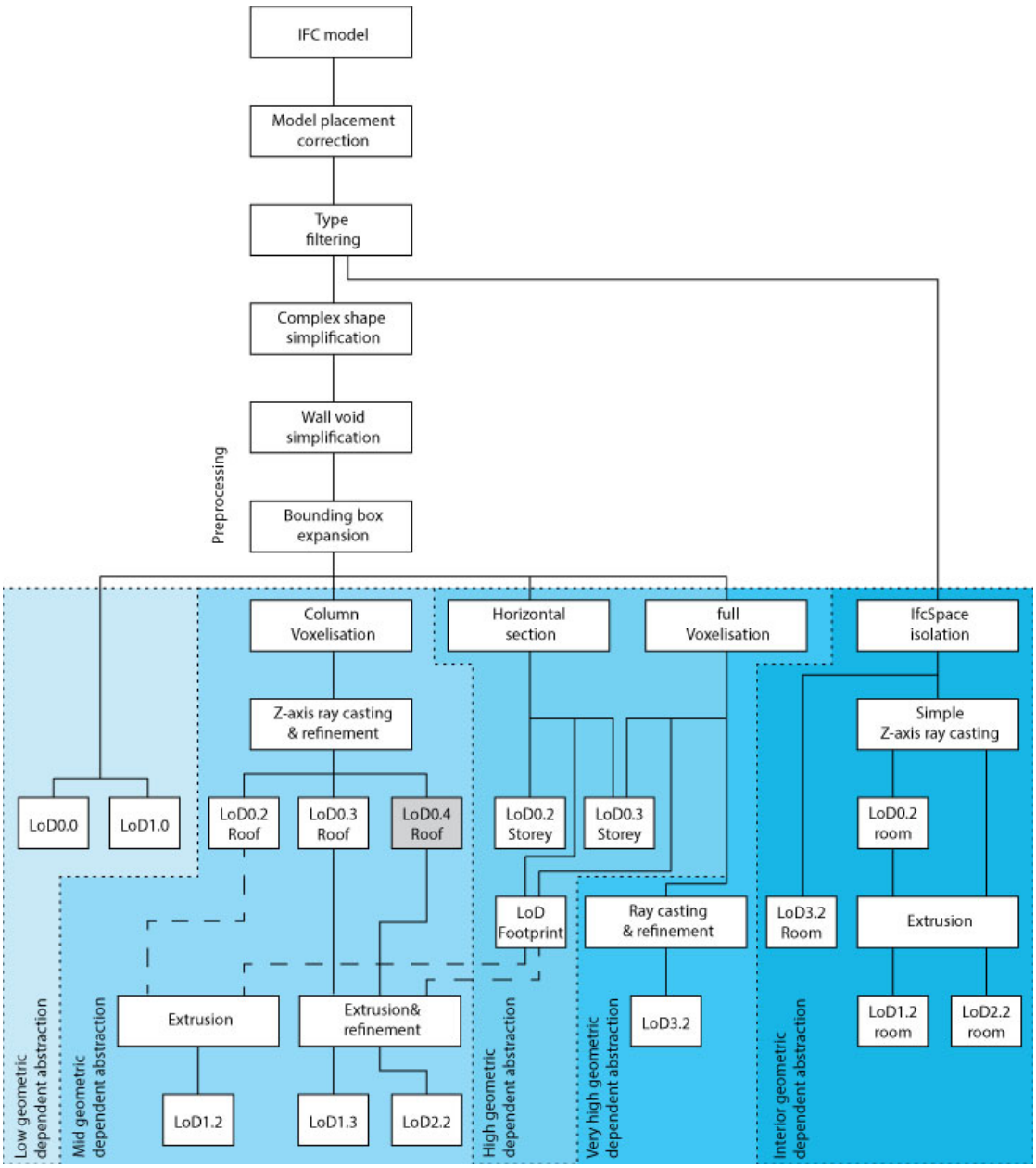


Figure 4: Flowchart to show the relationships of the different steps in the process. Non-standard LoD are shown in gray. Optional connections based on the desired output refinement are shown in dashed lines. This is a simplified representation of these steps. In practice, and in the descriptions in the following sections, these are more complex.

In the remainder of this section, the developed abstraction methods are described. First, the preprocessing and the voxelisation methods are described. After that, the five abstraction methods are described.

3.1 Preprocessing

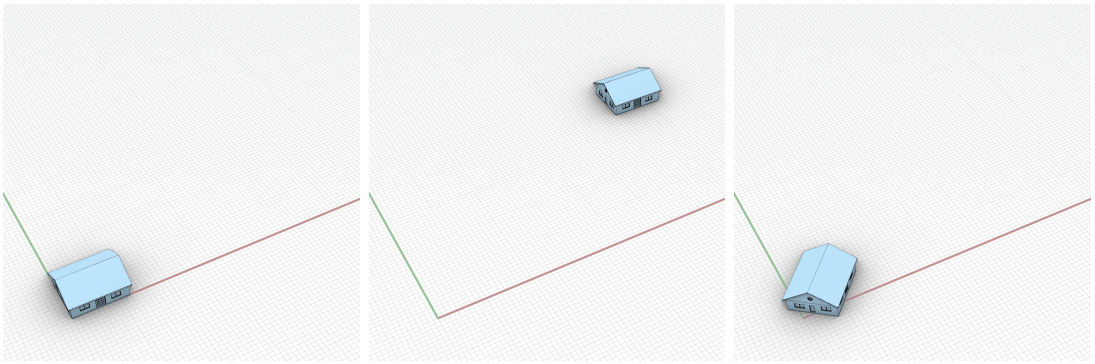
The preprocessing encompasses three different simplifications steps: the model placement is corrected, objects are filtered based on their *IfcClass*, and objects with complex geometries are simplified. As mentioned above, these simplifications are done before every abstraction method. An exception is the interior geometric dependent abstraction. This abstraction process does not require the complex geometry simplification step, as can be seen in Figure 4.

3.1.1 Correction of model placement

An IFC model can be placed in sub-optimal ways in its local coordinate system, see Figure 5. Mostly, three unwanted situations can occur: a model is placed far away from its local coordinate system's origin, a model is rotated in a way where it is not axis aligned to its local coordinate system's axis, or both. These erroneous local placements are often used to compensate for partial, incomplete, or missing referencing data in the *IfcMapConversion* class.

Geometry that is located far away from the local coordinate system's origin $(0, 0, 0)$ can be affected by floating point precision issues. This would result in a loss of significance which limits the accuracy of subsequent processing, possibly rendering their outcomes unusable. An easy solution for this issue is moving the geometry of the model closer to the local coordinate system's origin. To do this, a translation vector is computed. This vector is constructed from the lower left bottom corner of a bounding box around one of the model's *IfcSlab* objects to the origin of the local coordinate system. All other objects that are to be processed in later steps are translated according to this vector. This results in a coarse location correction of the model which is sufficient to avoid issues in further processing. The inverse of this vector is stored to be added to the georeferencing data of the output GIS file. It is presumed that all building models include at least a single *IfcSlab* object. If this is not the case, a fallback can be made to other classes of objects. However, the methods described in this report are developed to function on models representing buildings. Most models representing buildings have *IfcSlab* objects present, so if those are missing the model is possibly not suitable for the downstream processing methods.

The computed translation vector is based on only a single object to improve the computing speed and keep the complexity of the implementation simple. To prevent floating point issues, the model is not required to be placed precisely at the $(0, 0, 0)$ point with the lower left bottom corner of its total bounding box. A more refined approach would



(a) (b) (c)

Figure 5: An IFC model can be placed in suboptimal ways. a) Shows the optimal placement of the model. It is placed close to the local coordinate system's origin and aligned to its axis. b) Shows an often occurring error where the local coordinate system is used as an absolute coordinate system resulting in the model being placed away from the origin. In practice this placement is often very far away from the origin. c) Shows another often occurring error where the model's location is correctly stored in its georeferencing data, but the rotation is not. The rotation is incorrectly applied by rotating the model in its local coordinate system.

be to use all *IfcSlab* objects or even all *IfcObjects*. However, this will be an unnecessarily slow process if the input model is large, such as would be for a high-rise building.

Although the rotation of the model does not introduce the same issues as its location in the local coordinate system can, it is still important to correct it. As was mentioned before, and as will be covered by Section 3.2, a large part of the methods rely on voxelisation. Voxelisation, and especially spatial querying during that process, can be done more effectively if the model is local coordinate system axis aligned. To correct the rotation, a smallest orientated bounding box around the model's *IfcSlab* object's vertices is constructed in the XY plane. To construct this bounding box, the box is not rotated around the model but the model is rotated around the Z-axis. This will ensure that the bounding box will always be aligned to the axis of the local coordinate system. Since most buildings have floor plans that are fairly rectangular, this rotation will often result in the building model being aligned to the local coordinate system axis. The inverse of this rotation is stored to be applied to the abstracted output geometry after their creation.

3.1.2 Class based selection

The first and most coarse filtering step is the filtering based on object class. For the abstraction of BIM models to GIS models, only space dividing objects are required [van der

Vaart, 2022]. These objects represent physical/tangible geometries that play a role in the division of space. Examples are walls, floors, and roofs that encapsulate specific spaces and isolate the interior of the building from the exterior. The inclusion of objects which do not fulfil this role will unnecessarily increase the computation complexity of the processing step.

This filtering can be achieved because every object in an IFC file is assigned to a class. In most cases, the space dividing object, described by van der Vaart [2022] suffices. These are: *IfcWall*, *IfcCurtainWall*, *IfcWallStandardCase*, *IfcRoof*, *IfcSlab*, *IfcWindow*, *IfcColumn*, *IfcBeam*, *IfcDoor*, *IfcCovering*, *IfcMember* and *IfcPlate* objects. However, in practice the use of classes in IFC files is not consistent and often has errors [Noardo et al., 2021]. This means that relying only on this selection of classes will not always yield the desired filtering. To address this, the class filter should be made flexible. If it is known that a certain class is used in an unexpected way, this class can be included or excluded from the process. Despite this flexibility in the approach, it is always better to resolve these issues in the BIM models when possible. Correct IFC class use in BIM models is a good practice and makes the models more suitable for processing by other applications.

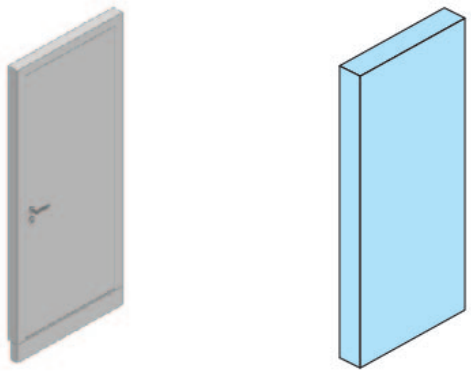
The classes in an IFC file can also be used as abstract/non-geometric containers that include other objects that represent actual geometry. For example the *IfcRoof* class is often used as a container "super" object that contains *IfcSlab* class "sub" objects, which are then used to represent the roof's geometry. It is important that these "sub" objects are also evaluated even if their *IfcClass* falls outside of the used class filter. If not, important geometry could be missed and the final result could be unusable.

Interior space abstractions could in theory also be based on these space diving objects. This would be a complex process similar to, and possibly more complex than, the exterior abstraction processes. Due to time constraints, a simpler approach was chosen.

The interior abstractions are based on the *IfcSpace* representations. So, if interior spaces are to be abstracted and exported to CityJSON, the *IfcSpace* objects are also filtered. These objects are not part of the potential space dividing object list. They are filtered and stored in their own unique container to be used as is, see Section 3.7. The class use of *IfcSpaces* in IFC models is often correct [Noardo et al., 2021]. So, the filtering process relies on this IFC class only and no flexibility in the class use is required.

3.1.3 Complex object abstraction

After the initial filtering step, the geometry of complex objects, i.e. *IfcDoor* and *IfcWindow* is simplified. These objects often include many small details, such as hinges and grips, which are not relevant at a GIS scale and would unnecessarily increase the complexity of subsequent abstraction steps. Because doors and windows are often box-like in nature, they can be abstracted by replacing their geometry with an orientated smallest bounding box in full 3D space (using X, Y and Z rotation), see Figure 6.

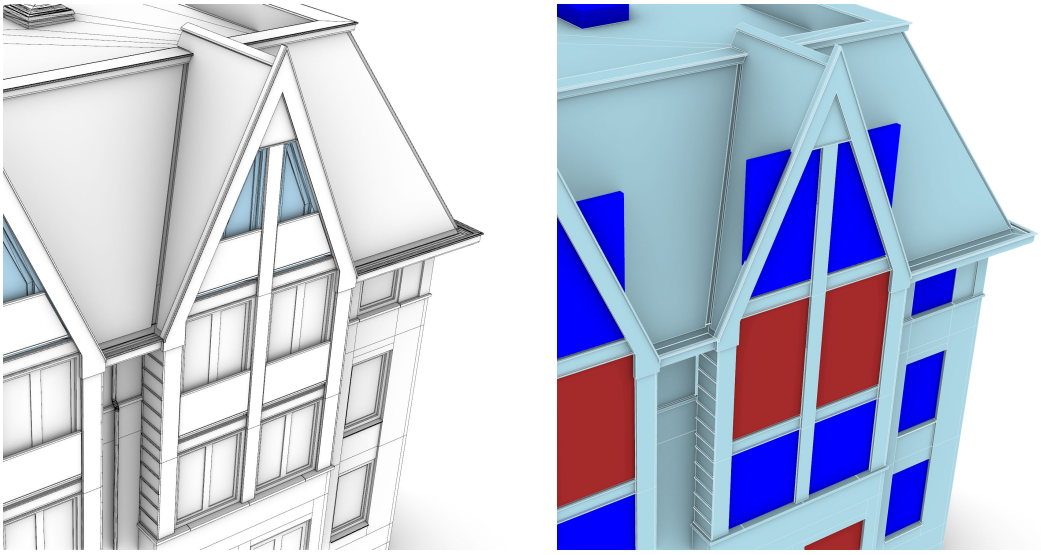


(a) (b)

Figure 6: An example of simplification by bounding box creation. a) Shows a complex *IfcDoor* object. b) Shows the resulting bounding box that is used to replace the *IfcDoor* geometry. The reduction of detail can be clearly seen, every side of the door is constructed out of a single surface and the door handle has been internalized as well. Image from van der Vaart [2022].

Not all *IfcDoor* and *IfcWindow* objects can be simplified using this box simplification. For example, if a window has a triangular frame a rectangular bounding box will not accurately represent the window, see Figure 7. If the representation error is considered too large, there should be an option to bypass this simplification process. This can be done for either all windows and doors, so that none of them are simplified, or selectively, so that only windows and doors that have a certain GUID are not simplified. The non-simplified objects will reduce the robustness and time effectiveness of the subsequent steps due to the increased geometric complexity.

When faster processing is required, a more drastic abstraction of windows and doors can be achieved by selectively applying subtractive elements. Doors and windows are often placed in openings inside of *IfcWall*, *IfcSlab*, or *IfcRoof* objects. These openings can be modelled with *IfcOpeningElement* or *IfcFeatureElementSubtraction* objects. The geometry of these objects is used to subtract geometry of other objects and create openings. For example, if these subtractive elements related to a wall are not applied, the wall will have no opening. The windows and doors that were placed inside of these openings can now be discarded without creating a hole between the interior and exterior. However, not every *IfcOpeningElement* and *IfcFeatureElementSubtraction* object can be discarded blindly. They can also be used to model openings that are not filled by doors and windows. Discarding those subtractive elements will close openings that are supposed to be open. It helps that each unique opening is created by one, or more, unique subtractive element(s). This means that we can selectively apply subtractive



(a) (b)

Figure 7: Example of the incorrect results that can occur when box simplifying windows that have a triangular frame. a) Input IFC file with triangular windows highlighted in blue. b) Output LoD3.2 abstraction when box simplification is used, resulting in windows that are incorrectly simplified. This incorrect simplification also clips the roofing structure, which will have an effect on the mid geometric dependent abstraction methods.

elements to the *IfcWall*, *IfcSlab*, or *IfcRoof* objects only if they are not occupied with an *IfcDoor* or *IfcWindow* object, see Figure 8. Both the simplification of the *IfcWall*, *IfcSlab*, and *IfcRoof* objects and the discarding of a set of *IfcDoor* and *IfcWindow* objects reduces the geometric complexity. However, this will also mean that windows and doors cannot be detected accurately during the following processes. In most cases this will have no notable effect on most LoD1.2, 1.3 and 2.2 abstraction processes because most of these abstractions have no representation of windows and doors. However, in the case of LoD3.2, windows and doors will often not be properly detectable.

This simplification cannot always be applied. In certain cases, the voids in *IfcWall*, *IfcSlab*, or *IfcRoof* objects are modelled without the use of *IfcOpeningElement* and *IfcFeatureElementSubtraction*. In these cases selectively applying subtractive elements has no effect on the objects.

In all normal processing cases, it is recommended to only use the bounding box simplification described earlier. As mentioned, most of the intricate detail of windows and doors is not required for GIS. It is recommended that the outliers that cannot be properly simplified by this method are filtered out by their GUID. If no LoD3.2 is required, selectively applying voids (i.e. subtractive elements) can improve the computing speed and robustness of the following processes even further. If LoD3.2 is required but no win-

dow/door data is necessary, applying this can also be a solution. In addition, if LoD3.2 is required but the IFC model is too complex to be processed, applying this can be a solution to allow for further processing. Depending on the desired goal of the abstracted shape, this is an optimisation choice. However, if this simplification is applied, the output GIS model should have information related to this choice included to inform users. This is especially important if the opening objects in the IFC model are modelled in different ways and some windows and doors are eliminated while others are not. With clear information on the abstraction process followed, the user will know to what extent to trust the data related to the doors and windows.

3.2 Voxelisation

For mid, high, or very high geometric dependent abstraction processes, a voxel grid will be created. The bounding box used for the rotation of the IFC model (see Section 3.1.1) is expanded so that it encompasses all the vertices of the selected space dividing objects (see Section 3.1.2). This tightly encapsulating bounding box is later used to create the LoD0.0 and 1.0 abstraction shapes as described in Section 3.3. The dimensions of this box are rescaled so that there is at least a single layer/shell of non-intersecting voxels surrounding the entire input model.

The voxelisation approach is dependent on the desired output abstractions. If mid geometric dependent abstraction is to be executed, the voxel grid would be populated with voxel columns. If high or very-high geometric dependent abstraction is selected to be executed, the voxel grid will be populated with full voxelisation. The two approaches are not interchangeable: if both voxelisation approaches are required, both are executed to be utilized for their respective abstraction processes.

Column voxelisation is the result of "growing" voxels vertically in a column-like shape, see Figure 9. All the voxels in the top "slice" of the voxel grid are used as starting voxels for this growing process. Each voxel will be elongated downwards (negative Z-direction) with the voxel size dimension until the created column intersects with one or multiple mesh representations of the space dividing objects. If the column encounters no objects, the growing of the column ends if it surpasses the lower Z bound of the total voxel domain. For example, if a voxel size of 1 metre is used, the column starts as a $1 \times 1 \times 1$ metre cube. If this intersects with no objects, it will grow to a column of $1 \times 1 \times 2$ metre. If again no object is intersected, it grows into a column of $1 \times 1 \times 3$ metre, and so on.

For the full voxelisation, each voxel is independently tested for intersection with a mesh representation of any of the space dividing objects. After the intersection test has been executed for all voxels, the exterior space can be grown from one of the voxels that are at the edge of the domain. Because the domain was expanded so that it includes a shell of non-intersecting voxels, it is known that any voxel in this shell represents the external void and consequently is not intersecting with any of the space dividing objects.

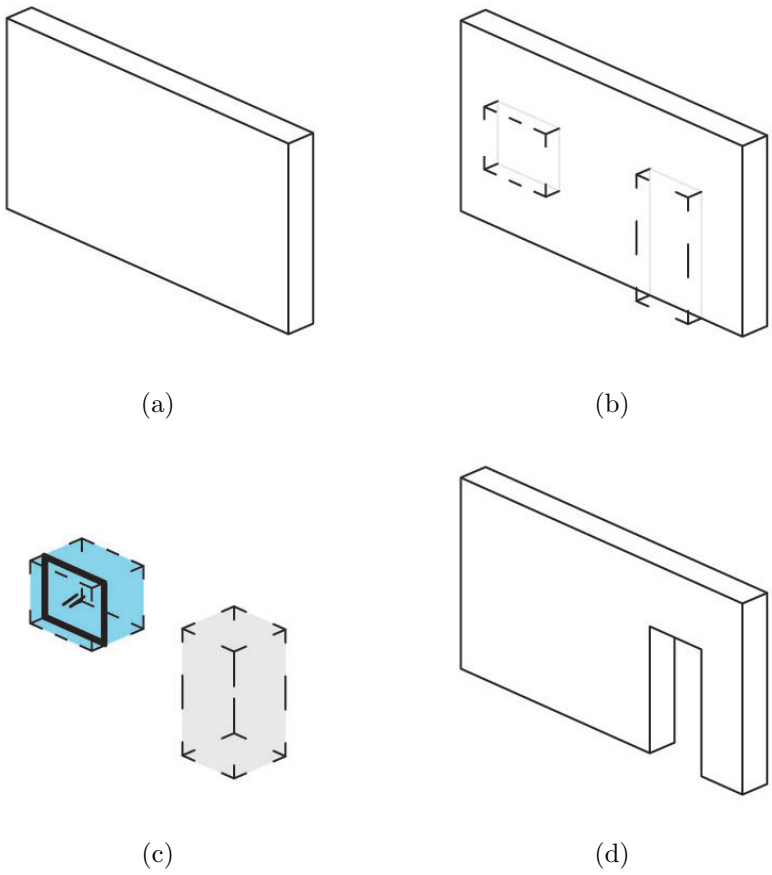


Figure 8: An example of selectively applying subtractive shapes in an *IfcWall* object. a) A wall with no subtractive objects applied. b) The original wall with the subtractive objects superimposed on top of it. c) The objects encapsulated by these subtractive objects. The gray void has no nested objects while the blue void has a nested *IfcWindow* object. d) Only the subtractive objects with no internal objects is applied, resulting in the simplified wall geometry. Image from van der Vaart [2022].

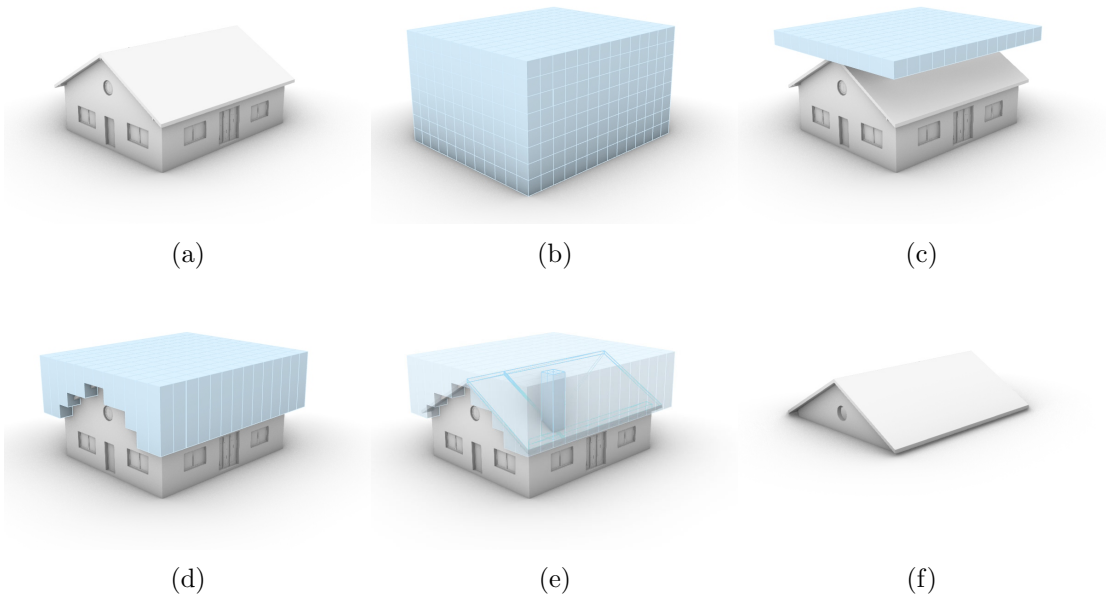
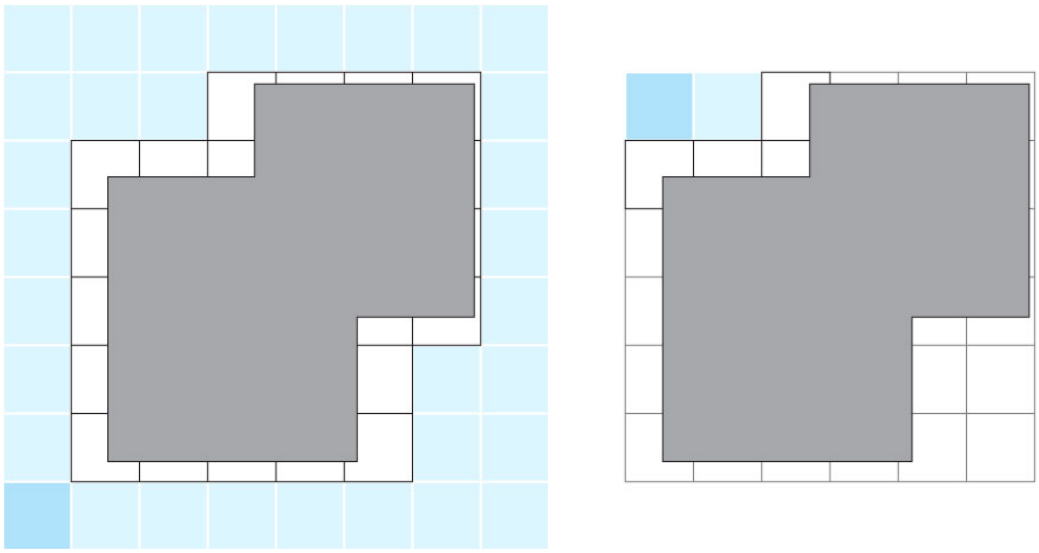


Figure 9: Example of column voxelisation. a) Shows the starting IFC model. b) Shows the full voxel grid. c) Shows the top slice of voxels from which the growing is started. d) Shows the elongated columns that are grown up until they hit geometry. e) Shows an isolated voxel with the wireframe of the geometry it intersects with highlighted. f) Shows the isolated geometry of the model that intersect with the voxel columns. Note that for c) onward the outer ring of columns has been hidden for clarity.



(a) (b)

Figure 10: Comparison of voxelisation with and without the expansion "ring" domain. This is a 2D plan view where the outline of a building is shown in gray. Subfigure a) and b) show the resulting non-intersecting void voxels highlighted in blue that were grown from the starting voxel highlighted in dark blue. a) Shows the correct void found due to the expansion ring. b) Shows that without this, the growing gets stuck and only detects a small area.

Additionally, this shell guarantees that the growing process can completely surround the building, see Figure 10.

If information related to interior voxels is required, see Section 3.5, the voxels representing the interior spaces can be grown after the growing of the exterior has been finished. This can be started from any unprocessed non-intersecting voxel. This growing process can be repeated until no unprocessed non-intersecting voxels are left.

The voxels are not just geometry. They also store data. For example, each of the voxels stores the *IfcProduct* objects that it is intersecting, part of which space it is, and if it is located in the interior or exterior of the model.

As noted by van der Vaart et al. [2024] the "ideal" voxel size depends on the size and nature of the model that has to be voxelised. A too large voxel size might miss detail required for any of the downstream applications, while a too small voxel size will slow down the subsequent processes and reduce robustness. Therefore, the voxel size can be selected by the user so that it can be fine-tuned to fit the model that is to be processed.

3.3 Low geometric dependent abstraction

The low geometric dependent abstraction processing extracts the LoD envelopes that are primarily based on the model's vertices (LoD 0.0 and 1.0 exterior). Due to this low dependency on the model's geometry and the coarseness of the abstraction shapes, large errors in the input IFC models can be present without having a notable impact on the output.

The LoD0.0 representation can consist out of two faces: a face representing the roof outline and a face representing footprint outline. If the LoD0.0 roof outline is desired, the face representing the roof outline is created based on the tightly encapsulating bounding box surrounding all the space dividing objects. The top surface of this bounding box is isolated and stored as an independent object in the output file. The surface type is set to *RoofSurface*. If no footprint outline is desired the roof outline surface is placed at the footprint height instead of at the top height. Instead of the *RoofSurface* surface type this surface gets the custom surface type *+ProjectedRoofOutline*. This custom type is used to alert the user that this is not a footprint based surface.

If the LoD0.0 footprint outline is desired, a new bounding box has to be created. The tightly encapsulating bounding box surrounds all objects, while the footprint in theory can be smaller. The footprint outline LoD0.0 surface is based on the bounding box of all the objects that are placed at, or intersect with, the footprint level ± 0.15 m. A bounding box is created from these isolated objects' vertices. The lower vertical surface of this bounding box is isolated, placed at the footprint height, and stored as an independent object in the output file. The surface type is set to *GroundSurface*.

The model is not rotated when creating the bounding surface that represents the LoD0.0 footprint outline. This means that this bounding surface might not be the smallest bounding surface around the footprint area.

The LoD1.0 representation is a copy of the shape of the tightly encapsulating bounding box surrounding all the space dividing objects, see Section 3.2. The type of the four surfaces of this box that have a normal with a Z-component of close to 0 are set to *WallSurface*. Of the two remaining unclassified surfaces, the type of the surface with the highest Z-value is set as *RoofSurface*. The type of the other surface is set as *GroundSurface*.

3.4 Mid geometric dependent abstraction

The mid geometric dependent abstraction extracts LoD envelopes and projections that are primarily based on the model's roofing structure (LoD 0.2 and 0.3 roof structures and 1.2, 1.3, and 2.2 exterior). These are the LoDs that are most used in practice since they are relatively easy to generate from measurements. These abstractions are usually made from airborne measurements such as airborne LiDAR scans. To isolate the

roof structures from a BIM model these in field approaches are "simulated". They are, however, altered to fit and optimised for a BIM source.

The roof related elements are isolated in two steps: coarse filtering with column voxelisation and fine filtering via ray casting in Z-direction, see Figures 9 and 11. Due to the high dependency on the input model's roof structure, it is important that this structure is modelled properly in the IFC file. However, the low dependency on the rest of the model's geometry means that large errors can be present in those other areas without having an impact on the output. In theory, an input model that only includes the roof structure with all other geometry missing could still create a satisfactory result.

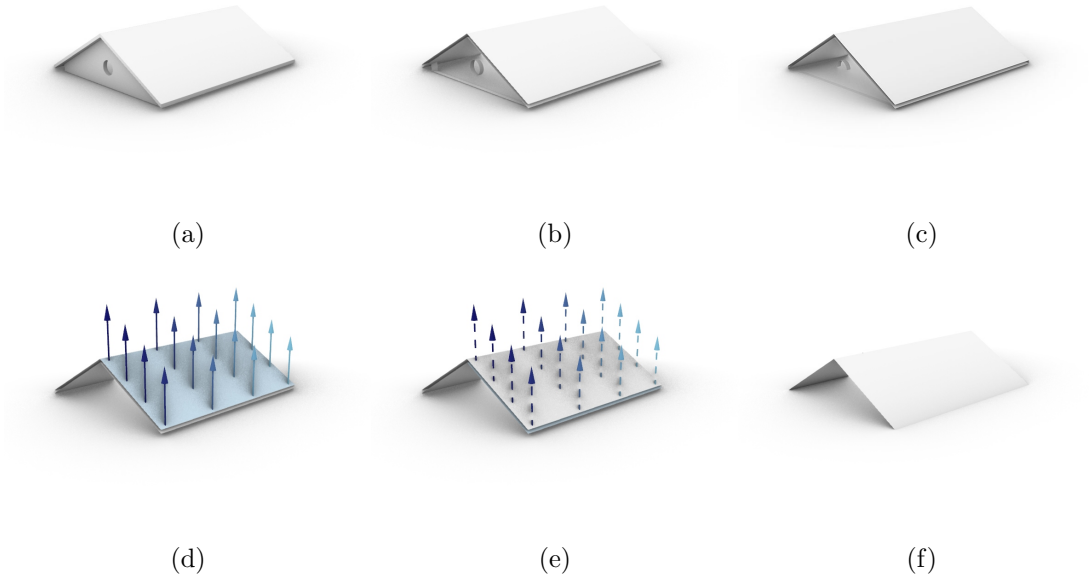


Figure 11: Example of the surface filtering and ray casting to isolate the surfaces that are part of the roof structure. a) The resulting surface collection from the column voxelisation described in Section 3.2. b) The collection of surfaces is filtered by eliminating the surfaces with a normal direction with a Z-component of 0. c) The collection of surfaces is filtered by eliminating the similar faces. d) and e) show an isolated situation of the ray casting process of the two viewer facing roof surfaces. The top surface in d) has at least one non intersecting ray (solid colour arrow). The bottom surface in e) has no non intersecting ray (dotted arrow). f) The results from the complete ray intersecting process.

3.4.1 Filtering of surfaces

The first step in the isolation of the roof objects is the column voxelisation, see Section 3.2. The objects with which the column intersects are assumed to have at least one

surface that could play a role in the construction of LoD0.2, 0.3, 1.2, 1.3, and 2.2. The unique objects that are intersected and their surfaces are selected, see Figure 11a.

The selected objects' surfaces are further filtered via a ray casting process where the rays are cast from each surface upwards parallel to the Z-axis, see Figure 11. The rays are cast from the surfaces themselves instead of from the "air" (like is done for in field LiDAR measurements) due to the possible presence of small surfaces that could be missed if a uniform grid were be used. By using the surfaces as a casting source the density of the rays can be refined depending on the properties of the surface that has its visibility tested.

Ray casting can be a resource intensive process, so it is important to avoid unneeded casts. This is accomplished by filtering the surfaces with some less resource intensive approaches prior to the ray casting. Firstly, surfaces that have a normal with a Z-component of close to 0 are removed from the surface list. These surfaces will not be visible from a Z-direction only cast. Secondly, surfaces that are part of the same object are tested if they are equal when disregarding the Z-component. For example, a concrete floor slab often has a top surface representation and bottom surface representation that are identical except for their location in the Z-direction. Thus, the top surface perfectly covers the lower surface, this means that only the top surface will be visible from a Z-direction only cast. The lower surface can be discarded without the need of any ray cast. This similarity test has been kept lightweight to retain computing performance. So it only tests for three differences: area, vertex count and vertex pairing by location. False positives that are incorrectly included due to this coarse filtering will be caught in the subsequent steps.

The remaining surfaces are populated with a point grid that will function as the rays' origins to cast from. Depending on the shape of the surface, different approaches can be applied. If the shape of the surface is a thin triangle, a triangular grid is created by using the lines between the centre point and the three corners as axis. Other surfaces are populated by a rectangular grid along the U- and V-coordinates of the surface. Depending on the shape of the surface, this point grid might not represent the surface properly, see Figure 12. To improve this, a method utilising the surface's wires is also applied. This method offsets the edges of the surface slightly inwards and places points at a regular interval along these offset edges. The offset edges are only used for the point population and can be discarded after the process is finished.

The density of the grid from which the rays are cast is at least 0.5×0.5m. This is fairly coarse as a base and is refined on smaller surfaces. If the surface is covered by less than 5 points, the grid size is split in 2 recursively. This is done until the surface is covered by more than 5 points or if the grid density is smaller than 0.05×0.05m.

From each of the grid points, rays are cast along the Z-axis vertically upwards. First, it is tested if any of the rays intersect with another surface of the same object. If all rays intersect, it means that the surface is completely covered by other surfaces of the same object. So, this surface wont be visible from above and can be discarded. This reduces

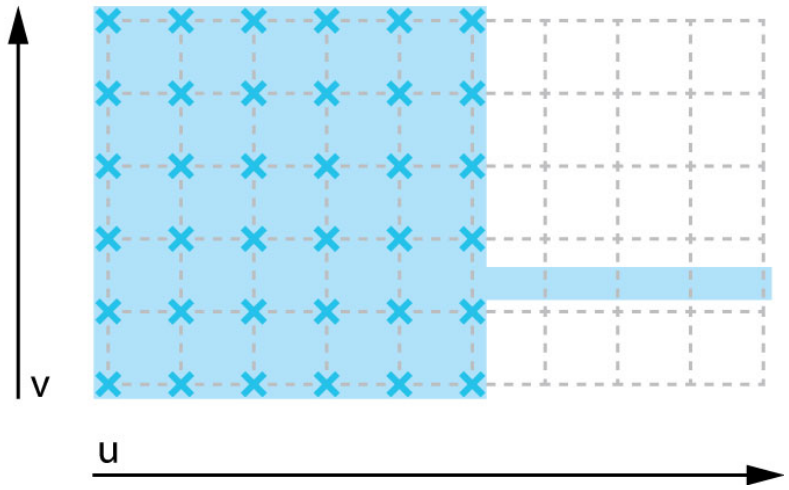


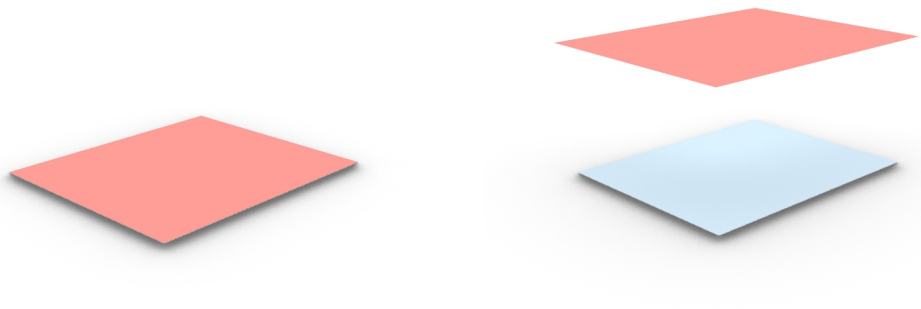
Figure 12: Some surfaces have shapes that are not properly represented by a point grid that is based on the UV domain. For example, this surface (in blue) has a thin extension that passes through the grid (in gray) without points being placed upon it (the crosses).

the pool of surfaces further and also catches the surfaces that might have slipped through the similarity test in the earlier filtering step. The non-intersecting rays are tested for intersection again, but this time against the surfaces of the complete pool. If the surface has at least a single cast ray that does not intersect with any of the surfaces, the surface is stored as part of the roof structure.

This is followed by a polyhedral surface approximation of the collected surfaces. An IFC file supports both implicit en explicit geometry while CityJSON only supports explicit geometry. Doing the simplification step this early in the process improves processing speed and robustness due to the simplicity of the new surfaces.

3.4.2 Creating surface based abstractions

To construct the LoD0.2 roof abstraction, the surfaces identified as part of the visible roof structure are all projected to the XY-plane ($Z = 0$) and merged into a single surface: the projected roof outline. If footprint extraction is to be executed/exported, which can be generated by the high dependent abstraction method (see Section 3.5), the roof outline surface is translated to the max height of the building. The surface type is set to *RoofSurface*. If no footprint extraction is to be executed/exported the projected roof outline is stored at the ground level of the model. This can function as an alternative representation of the footprint at LoD0.2, See Figure 13. In this case the surface type of the surface is set to *+ProjectedRoofOutline*. To alert the user that this is not the actual footprint representation.



(a) (b)

Figure 13: The different placement of the roof outline surface based on the desired output. a) The roof outline is placed at the building's footprint height if no footprint output is desired. b) The roof outline is placed at the building's max Z height if footprint output is desired.

To construct the LoD0.3 roof abstraction (i.e. with varying height in case of significant height jumps), the surfaces identified as part of the visible roof structure are grouped and flattened, see Figure 14b and 14c. A surface is grouped together with another surface if the two surfaces have parallel edges that completely or partially rest against each other. Each group of surfaces is then flattened so that the horizontal component of the normal has a magnitude of 0. Because surfaces with a normal Z-component of close to 0 were excluded during the filtering process (see Section 3.4.1), no checks for zero area cases are required. The flattened surfaces of a group are merged into a singular surface and placed at a Z height which is the highest Z-value of the original group.

This group of flattened roof surfaces is not yet a valid LoD0.3 roof model. The roof surfaces of LoD0.3 are not allowed to overlap with each other, according to the LoD framework of Biljecki et al. [2016]. The LoD0.3 roof structure is in most cases identical to the LoD1.3 roof structure.

To create a valid LoD0.3 roof structure that does not overhang over itself in the XY-plane the surfaces have to be trimmed, see Figure 14c, 14d, 14e, and 14f. This is done by extruding the roof surfaces downwards to the lowest height of the input model to form solids. These solids are used to split the flattened isolated roof surfaces. Each of these flattened surfaces are split by the solids that were created by all the other flattened surfaces, except itself. To test which of these surfaces model the roofing structure, another ray cast in the Z-direction is done. In this case a single ray cast per surface suffices. If the ray intersects with one of the flattened surfaces, the split surface it originates from is ignored. The remaining sub-surfaces are collected and represent the LoD0.3 roof surface collection. The types of these surfaces are all set to *RoofSurface*.

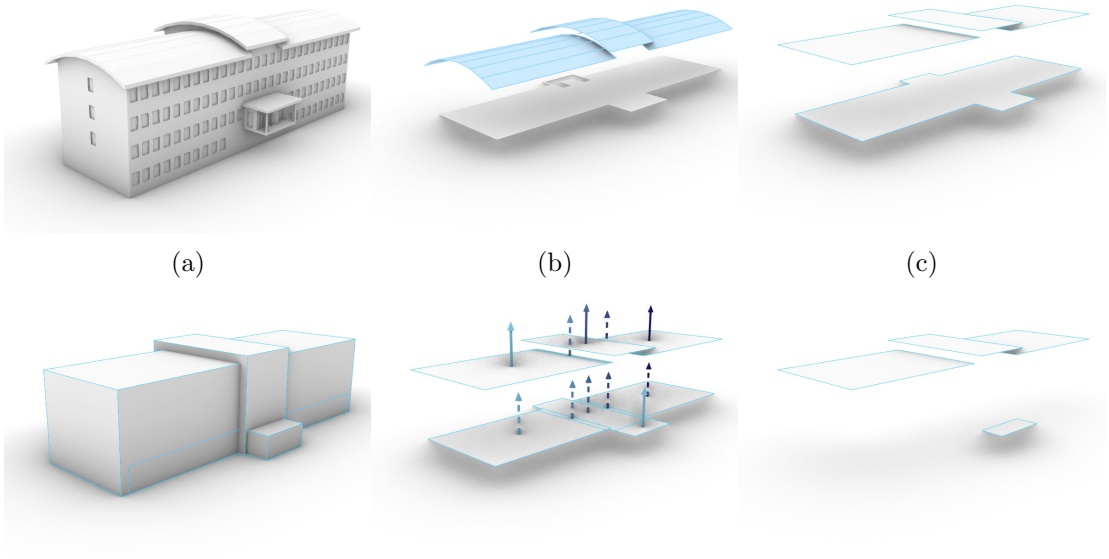
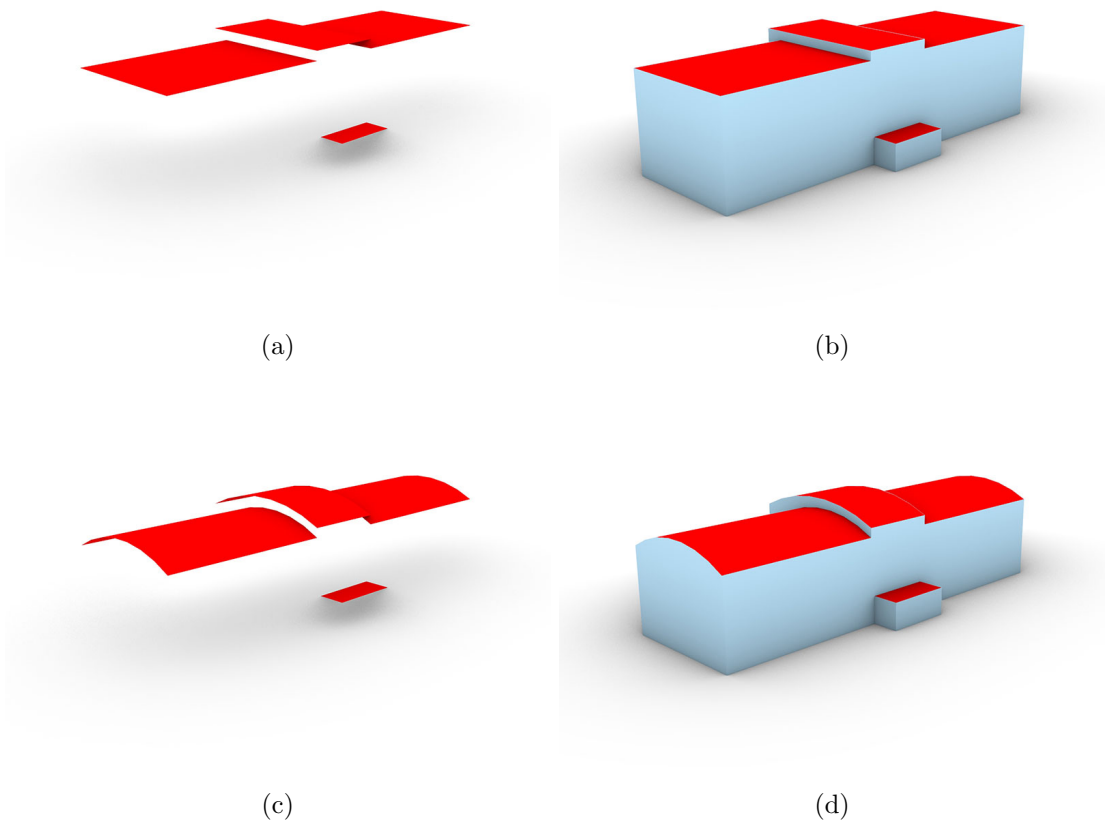


Figure 14: The steps of the LoD0.3 roof abstraction. a) For this example, another model than in the former figures was picked. b) The fine filtered roof surfaces that were isolated with the help of column voxelisation and ray casting. Three different groups of roofing surfaces are highlighted in blue. c) These groups are flattened. d) The surfaces are extruded to ground floor level and each original un-extruded surface is split with the extruded shapes that intersect it. e) The split surfaces are filtered with the help of a ray casting process, solid arrows do not intersect, dotted ones do. f) The resulting LoD0.3 surfaces.

Although not included in existing LoD frameworks, the non-flattened surfaces identified as part of the visible roof structure can also go through the same process to create the non-flattened roof structure. This becomes the isolated roof structure of the LoD2.2 abstraction in the same way LoD0.3 contains the roof structure of LoD1.3 abstractions, see Figure 15. We call this the roof structure of LoD0.4. To create the LoD0.4 roof surfaces, all the steps of LoD0.3 are followed, see Figure 14, except for the grouping and flattening of the shapes.



(a) (b)

(c) (d)

Figure 15: Comparison of LoD0.3 and the non-standard LoD0.4. LoD0.3 (a) contains the roofing structure of LoD1.3 (b). LoD0.4 (c) contains the roofing structure of LoD2.2 (d).

3.4.3 Creating volumetric abstractions

To construct the LoD1.2 exterior abstraction, the LoD0.2 roof outline surfaces are projected to the XY plane ($Z = 0$), if not already located there. These projected surfaces are then extruded upwards to the max height of the building. The same logic for the

surface semantics that is applied for the LoD1.0 shape (see Section 3.3) is applied here. The type of the surfaces that have a normal with a Z-component of close to 0 are set to *WallSurface*. Of the two remaining surfaces, the type of the top surface is set as *RoofSurface* and the type of the lower surface as *GroundSurface*.

The construction of the LoD1.3 and 2.2 exterior abstraction is closely related to each other. The only difference is that the LoD1.3 reconstruction use the LoD0.3 roof structure as starting point and the LoD2.2 processes uses the LoD0.4 roof structure as starting point, see Figure 15. The rest of the executed processes are identical.

Each of the starting surfaces (either the LoD0.3 or LoD0.4) are extruded downwards to the ground level and merged into a single solid. Depending on the input model, this can be a complex process consisting of many surfaces. Due to the nature of the input surfaces, a couple of steps can be taken to make the merging process more lightweight. As described in Section 3.4.2, the input LoD0.3 and LoD0.4 roof surfaces have no spatial overhang in the XY-plane. So, the extrusions of the input surfaces (LoD0.3 or 0.4 abstractions) will never intersect each other. The only interaction these shapes can have is surfaces touching. To merge these extrusion only the vertical faces that were created by the extrusion are required to be removed or trimmed, see Figure 16. The vertical faces are all the faces that have a normal with a Z-component of 0.

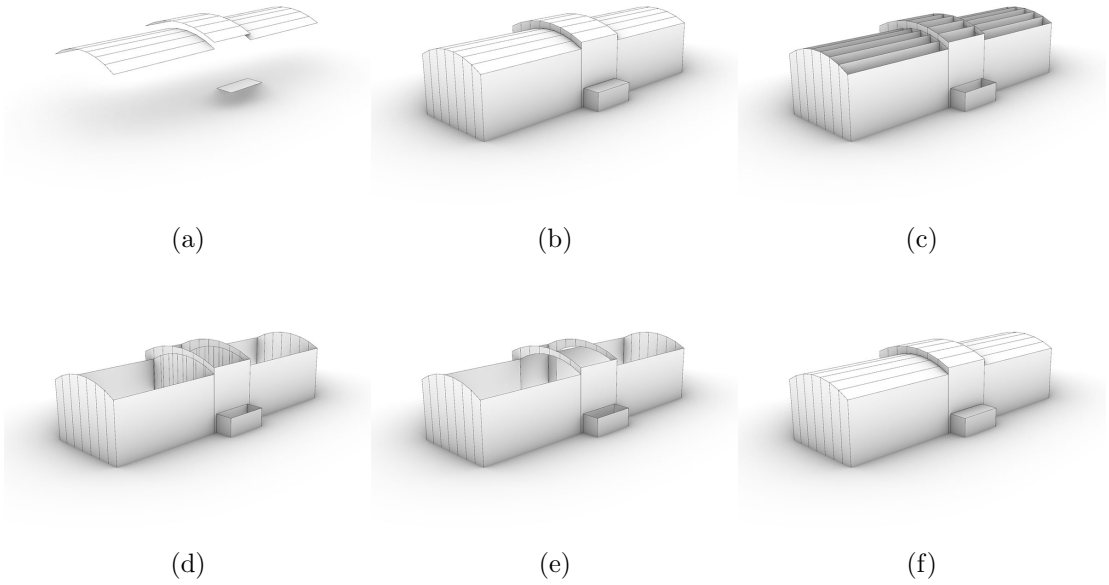
The first sub-step of the trimming process is the removal of duplicate equal vertical faces. If two, or more, roof surfaces touch, extruding them would result in duplicate surfaces along the edges where the roof surfaces are touching, see Figures 16a, 16b and 16c. Eliminating these faces effectively joins the extrusions from these neighbouring roof surfaces into a single shape without the need of any complex processes.

The second sub-step is the collection and splitting of the remaining vertical faces. Vertical faces that rest against each other can be used to split the faces into smaller subsurfaces that are either completely inside or outside of the building. The sub-faces that are inside of the building can be filtered out by again eliminating the duplicate surfaces, as described in the first sub-step.

For the creation of the bottom surfaces no complex processes are required. Because the input faces do not overlap in a XY-plane, the "ground" surfaces created by their extrusion also do not overlap. Therefore, the bottom surfaces that rest against each other with parallel edges can be merged into a single shape.

In theory, the trimmed vertical faces make a closed watertight shape together with the roof and bottom surfaces. Depending on the input shapes, the resulting shape will be an LoD1.3 (if the input were the LoD0.3 roof surfaces) or Lod2.2 (if the input were the LoD0.4 roof surfaces) abstraction.

A similar process for surface semantics that is applied for the LoD1.0 and LoD1.2 shape is applied for LoD1.3 and 2.2. The type of the surfaces that have a normal with a Z-component of close to 0 are set to *WallSurface*. The type of the surfaces that are at the



(a) (b) (c)

(d) (e) (f)

Figure 16: The steps for the LoD2.2 creation. a) The starting surfaces are LoD0.4. b) The starting surfaces are extruded downwards to the lower Z bounds. c) The vertical faces are isolated. d) Duplicate surfaces are removed from the collection. e) The left over surfaces are split by the surfaces that they touch and again the duplicate surfaces are removed from the collection. f) The resulting collection is joined by the earlier isolated non-vertical faces to create a airtight solid.

footprint height are set to *GroundSurface*. The type of the remaining surfaces are set to *RoofSurface*.

The described methods create LoD1.2, 1.3, and 2.2 abstractions that are based on the roof structure of the model. However, the LoD framework of Biljecki et al. [2016] does not clearly state if the LoD1 and LoD2 groups are based on the roof structure or the footprint. A simple refinement to this approach allows for a footprint based creation.

For LoD1.2 footprint based exterior abstraction the footprint can be extruded upwards to the building's max height. This does however required the footprint to be generated, which is a more complex process that relies more on the IFC model's geometry, see Section 3.5.

For LoD1.3 and 2.2 footprint based exterior abstraction the roofing surfaces that overhang over the footprint can be trimmed down before being used for extrusion, see Figure 17. The rest of the processes for creating both LoD1.3 and 2.2 remain exactly the same. However, like for the LoD1.2 case the trimming of the roof surfaces for LoD1.3 and 2.2 requires the footprint to be generated.

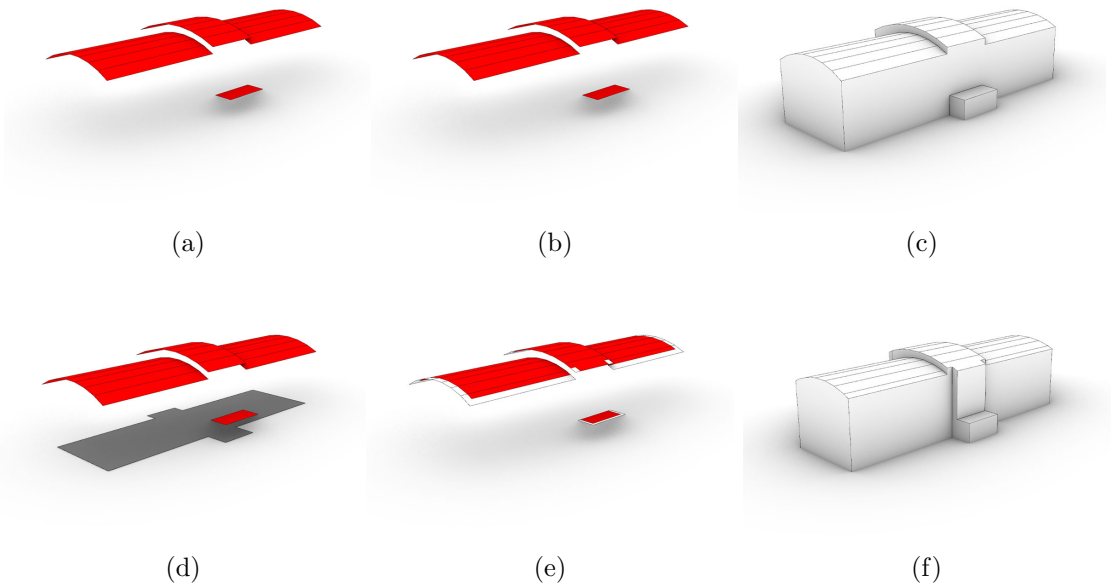
When the footprint based extraction is used, the LoD0.2, 0.3, and 0.4 roof surfaces can differ from the LoD1.2, 1.3, and 2.2 roof surfaces respectively. The LoD0.2, 0.3 and 0.4 roof surfaces remain untrimmed, so when overhang is present over the footprint, this will be reflected by the roof surfaces. The footprint based LoD1.2, 1.3, and 2.2 roof surfaces can possibly cover a smaller area in comparison.

3.5 High geometric dependent abstraction

The high geometric dependent abstraction processing extracts elements that are based on the model's outer shell (LoD0.2 and 0.3 storey representation and the LoD0.2 and 0.3 footprint). Unlike the very high geometric dependent abstraction processes (see Section 3.6), these processes only sample the outer shell at certain intervals: at the storey heights respectively and at the building's footprint heights.

LoD0.2 and 0.3 storey abstraction rules are not well described in existing documentation. So, we tried to apply the rules defined for the exterior to the interior storeys. For LoD0.2 a storey is represented as a horizontal surface (group) representing all the objects in a building at a certain storey elevation. This surface (group) will span the entire building at a single height. This loosely resembles the single top surface of the exterior LoD0.2 roof surface logic. For LoD0.3 a storey is represented as a horizontal surface (group) representing all the objects at a certain storey elevation that are related to that storey. This surface (group) can span only a part of the building at a single height. Depending on how the IFC model has been constructed this resembles the multiple horizontal flat plane LoD0.3 logic.

Due to the close relation of the LoD0.2 and 0.3 storey representation, their basic output is often very similar if not identical. For the development of an IFC to GIS method it



(a) (b) (c)

(d) (e) (f)

Figure 17: The difference between roof structure based LoD2.2 creation (top row: sub figure a), b), and c) and footprint based LoD2.2 creation (bottom row: sub figure d), e), and f). The first column shows the input geometry required in a) & d). It can be seen that for footprint based extraction both the roof structure (in red) and the footprint surface (in gray) is required. The second column shows the utilized faces for downward extrusion highlighted in red in d) & e). It can be seen that some surfaces are not used for footprint based extraction (in white) these are parts of the roof surfaces that extend over the footprint. The final column shows the resulting shapes.

was considered interesting to use two different, although related, approaches to construct them: LoD0.2 follows a simple approach which creates a simple storey representation. This is a robust process and works on most IFC models. LoD0.3 follows a more complex approach which requires a well constructed input IFC model but will also result in a more accurate storey representation with expanded surface information.

Similarly, the LoD0.2 and 0.3 footprint abstraction rules are also not well described in documentation. Due to time constraints it was chosen to create the same shape for both. A horizontal surface (group) representing all the objects in a building at a certain user submitted footprint elevation.

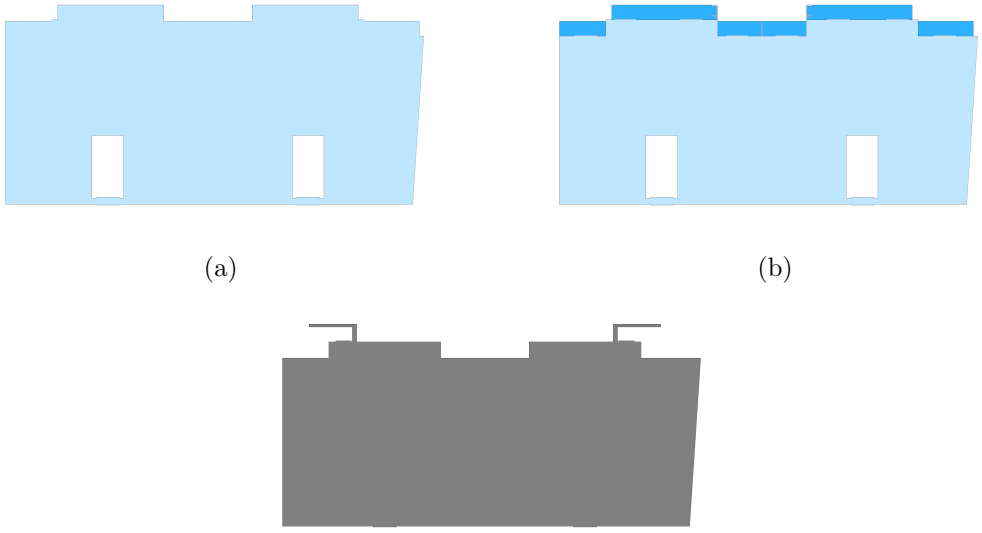
The footprint, LoD0.2 storey and LoD0.3 storey abstractions, although different in complexity, are closely related, see Figure 18. To keep the description of the three processes limited, only for the LoD0.2 simple approach the complete process will be described. The other sections covering the LoD0.3 complex approach and LoD0.2/LoD0.3 footprint abstraction will only describe how they differ from the simple approach. For the footprint of LoD0.2 and 0.3 exactly the same geometry and approach is used, so both the LoD0.2 and 0.3 footprint is described as one.

3.5.1 Simple approach for storey abstraction

This abstraction method leads to an LoD0.2 storey representation by taking a horizontal section through all the input model's space dividing objects at each of the storeys' height. These heights are found by taking the elevation attribute of the *IfcBuildingStorey* objects. If modelled properly this height should represent the elevation of the top of the construction slab of a storey. If multiple storeys are present with the same or similar elevation values they are grouped together and handled as if they are a single storey object. This can occur when a model is constructed from multiple IFC files.

Each unique space dividing object in an IFC file is presumed to be a volumetric air-tight shape. Splitting these shapes horizontally will thus create a closed planar horizontal wire. These wires can be converted into a planar horizontal surface. The resulting surfaces are collected for further processing.

The shapes created during this selection process do not completely represent the storey geometry. Only the shapes that are intersected by the elevation plane have their planar section representation created. Depending on how the model is constructed, objects such as the floors and balconies often may have horizontal faces that are flush or close to flush with the storey's elevation. These flush faces do thus not intersect with the elevation plane, see Figure 20. To prevent them from being ignored, the surfaces that are close to horizontal and are partially within a ± 0.15m buffer of the storey height are projected to the storey height and selected as well.



(a) (b)

(c)

Figure 18: The differences between an LoD0.2 storey (a), LoD0.3 storey (b) and a footprint (c). The LoD0.2 and 0.3 abstraction represents the same first storey of the model displayed in Figure 19. The footprint abstraction is located 1 m lower. a) The LoD0.2 storey is a simple surface with openings representing the staircase shafts. b) The LoD0.3 storey expands on this by making a distinction between the balcony surfaces as exterior (in dark blue) and the rest of the interior. c) The footprint is a simpler surface than LoD0.2 by eliminating the openings representing the staircase shafts.

Figure 19: 3D view of the model that is used for the example LoD0.2 and 0.3 storey extraction and the footprint extraction in Figure 18.

The resulting collection of surfaces is cleaned. This thinning done in two steps. Firstly, duplicate surfaces are removed. These duplicates can occur when a floor structure is constructed from different geometric *IfcObjects* to represent the different structure layers that fall within the ±0.15m flush face buffer. Secondly, surfaces which are completely encapsulated by a single other surface are removed. This will remove surfaces representing the walls that are completely resting on top of the floors.

The thinned out surfaces are merged into a single, or group of surfaces, representing the 2D section of the storey at a certain storey elevation. The geometry can be stored in a *BuildingStorey* CityJSON city object representing the group of storeys at that elevation. The used surface type for all the surfaces is *FloorSurface*. The attributes of the *IfcBuildingStorey* object(s) at that elevation can be copied to the city object. Each of these CityJSON objects has a child relationship to the main building's CityJSON object to keep the clear structure in the reconstructed model.

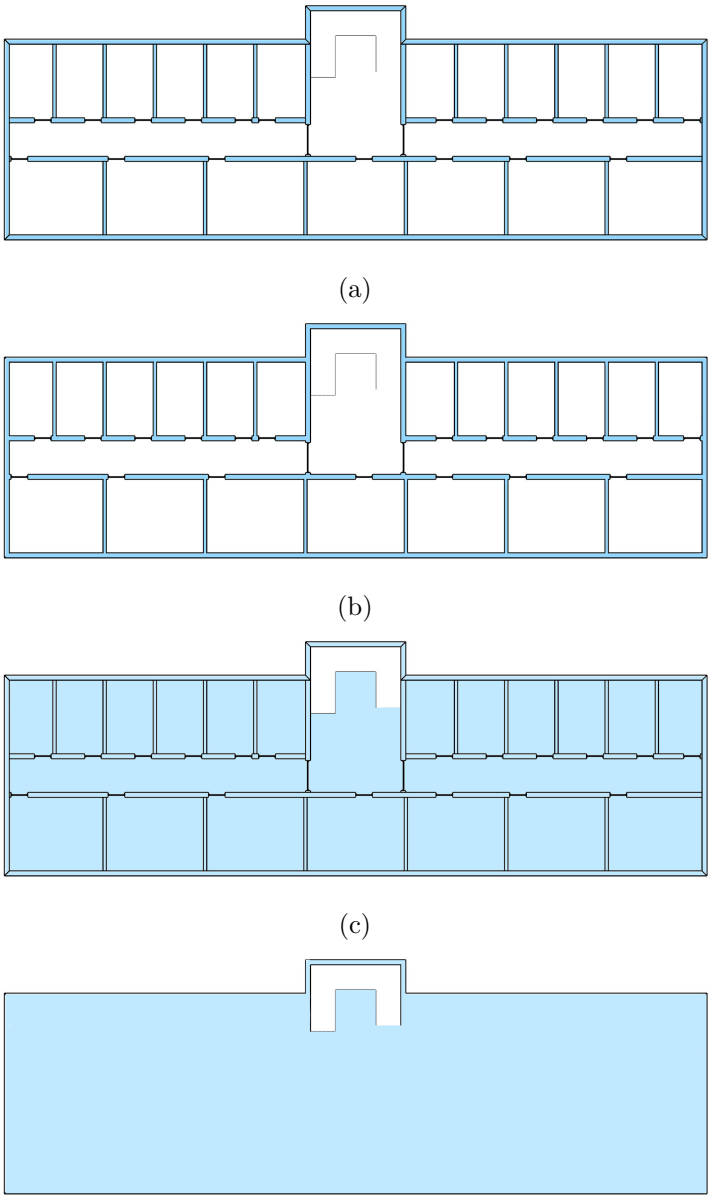


Figure 20: Difference between using intersecting geometry only and augmenting it with planer surfaces that fall within a buffer. a) Represents the intersection with geometry only. b) The undesired resulting storey geometry. c) Represents the intersection augmented with planar surfaces. d) The desired resulting storey geometry

3.5.2 Complex approach for storey abstraction

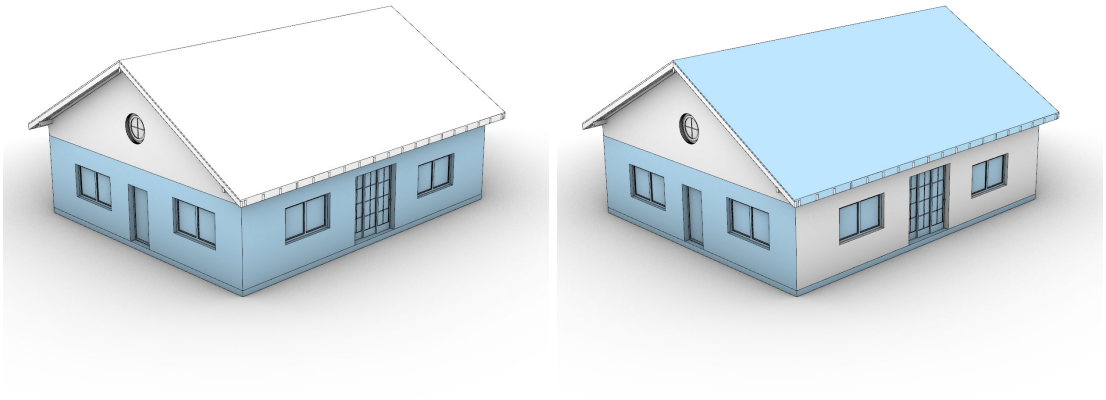
The storeys of the LoD0.3 abstraction are created in a similar manner as LoD0.2 but with two major differences. Firstly, the horizontal section at the storey elevation does not split all the objects at that elevation. It only splits the objects that are related to that *IfcBuildingStorey* object or group of *IfcBuildingStorey* objects at the storey elevation. This allows the accurate inclusion of half or split storey elevations across a building model. However, this requires the IFC model to have the correct objects bound to the storey surfaces. Secondly, during the process the surfaces are divided based on if they are interior or exterior. This makes it possible to isolate balconies and parts of the (flat) roof structure and to not include these outdoor parts in downstream analysis, if, for example, only interior is required.

To fetch the objects that are related to the storey that is being abstracted its related *IfcRelContainedInSpatialStructure* object has to be acquired, see Figure 21. If the model is properly constructed, this object represents a relationship between a relating structure/object and a set of related elements. In this case the relating structure is the *IfcBuildingStorey* and the related elements are the *IfcProducts* located at this storey. As with the LoD0.2 storey abstraction process, *IfcBuildingStorey* objects that have a similar or close to similar storey elevation are grouped. In the LoD0.3 case this means that the *IfcRelContainedInSpatialStructure* of all the relating grouped *IfcBuildingStorey* objects are used to create a single output abstracted storey shape (i.e. a surface or a multisurface).

To separate the interior surfaces from exterior surfaces, a different merging process is executed after the buffered horizontal intersection has been made. Because the final abstracted surfaces are not all simply merged into a single section, like in the LoD0.2 case, the thinning process has to be limited. While for the LoD0.2 abstraction the duplicates and completely encapsulated surfaces were removed, for LoD0.3 only the duplicates are removed. This is followed by a splitting process in which every surface is split by the other surfaces that overlap with it. After this, the duplicate split surfaces are filtered out. This results in a mosaic like representation of the storey section consisting of unique non-overlapping surfaces.

To test which surface is exterior and which one is interior, the voxels are used. For every surface in the mosaic a group it is tested if the voxels above it are representing the exterior. If any of the voxels is representing the exterior, the surface is assumed to be exterior. The interior and exterior surfaces are separately grouped. The surfaces of each of these groups are merged and stored in the GIS file.

Both the interior and exterior surfaces can be stored in the same *BuildingStorey* CityJSON city object representing the group of storeys at that elevation. The used surface type is *FloorSurface* for the interior surfaces and *OuterFloorSurface* for the exterior surfaces. The attributes of the *IfcBuildingStorey* object(s) at that elevation can be simply



(a) (b)

Type	Name
Project	Projekt-FZK-Haus
Site	Gelaende
Site	
Building	FZK-Haus
Building Storey	Erdgeschoss
Building Storey	Dachgeschoss

(c)

Figure 21: The relationship between a storey object (c) and the objects situated on that storey, highlighted in blue (a) is facilitated by *IfcRelContainedInSpatialStructure* objects. This relationship is often created by the modeller and is not always correct (b).

copied to the city object. Each of these CityJSON objects has a child relationship to the main building's CityJSON object to keep the clear structure in the abstracted model.

3.5.3 Footprint abstraction

The footprint abstraction method (LoD0.2 and LoD0.3) follows a very similar approach as the LoD0.2 storey abstraction. Instead of the storey's elevation, the footprint elevation is taken as the height of the horizontal section. The final LoD0.2 surface is further refined by selectively removing inner loops. Certain voids, such as elevator or staircase shafts, are not of importance for the (outdoor) footprint geometry. Including them will bloat the file sizes, increase downstream complexity and possibly confuse later users. Not all

inner loops should be removed though, e.g. inner loop can also represent a courtyard which is an element that should be reflected in the footprint.

The voids are filtered with the help of the voxel grid. For every inner ring's void area it is tested if the voxels above it are representing the interior. If any of the voxels is representing the interior, the ring is ignored.

The surfaces representing the footprint are stored in the same city object in the CityJSON file as the roof abstractions. The footprint shape is considered to be exterior. The type of the surface is set as *GroundSurface*.

Unlike the LoD0.2 and 0.3 storey abstractions the LoD0.2 and 0.3 footprint abstractions are exactly the same geometry. In the current version of CityJSON it is not possible to store the same geometry with multiple LoDs. So an identical copy of the geometry has to be stored if both LoD0.2 and 0.3 output is required.

3.6 Very high geometric dependent abstraction

The very high geometric dependent abstraction processing extracts LoD abstractions that are based on the complete model's outer shell also adding windows and doors (LoD3.2 exterior). It is the highest depended abstraction method and requires well made models to function properly.

The creation of the LoD3.2 abstraction can be split into three main processes: surface filtering, surface refinement and the fetching process of the attribute data. A visual example of the complete process can be seen in Figure 22.

3.6.1 Surface filtering

The aim of surface filtering is to isolate the surfaces that play a role in the construction of the exterior shell of the building. The filtering is done with the help of a voxel assisted ray casting process. Ray casting is a resource intensive process, because of that a filtering process based on the voxels is done in advance.

During the voxelisation process, described in Section 3.2, the voxel objects store data related to their intersections. Important for this process is that the voxels store if they are intersecting, with what they are intersecting and if they are part of the interior or exterior voids. With this information a coarse filtering of the objects can be made. The intersecting voxels that have one, or more, neighbouring voxels that are non-intersecting and external are assumed to represent the exterior shell of the building. The *IfcObjects* with which these filtered voxels intersect are assumed to have at least a single surface that is part of the exterior shell, see Figure 24. The unique intersecting objects of this group of voxels are collected and used in the ray casting process.

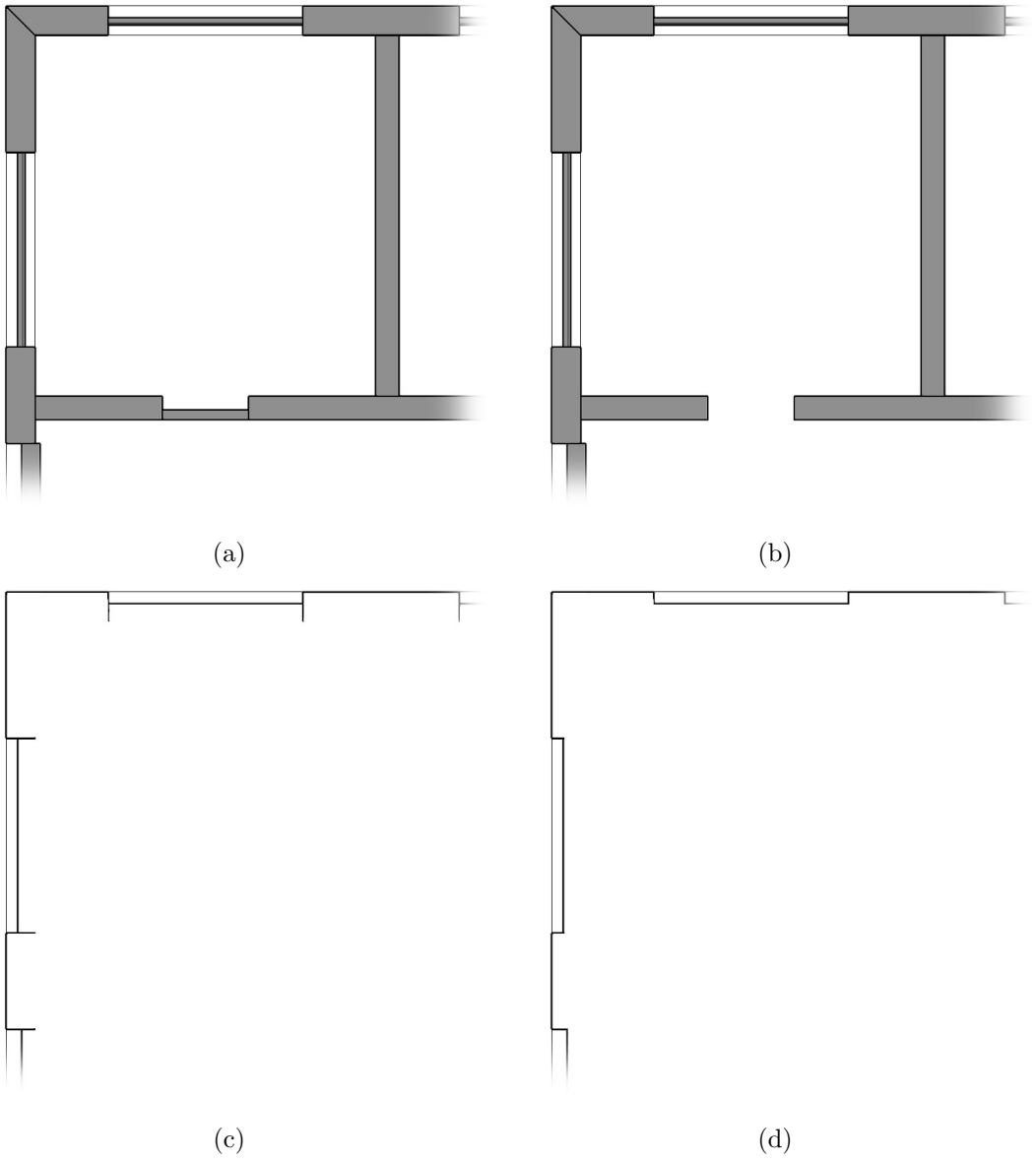


Figure 22: The steps of LoD3.2 abstraction displayed in a part of the floor plan shown in Figure 23. a) Displays the starting situation after the *IfcClass* filtering and complex shape simplification. b) The objects that are close to the exterior of the building model are isolated by using the intersecting voxels that neighbour an exterior void voxel. c) The completely inner surfaces are filtered out by voxel assisted ray casting. d) The surfaces are split and again filtered out by voxel assisted ray casting

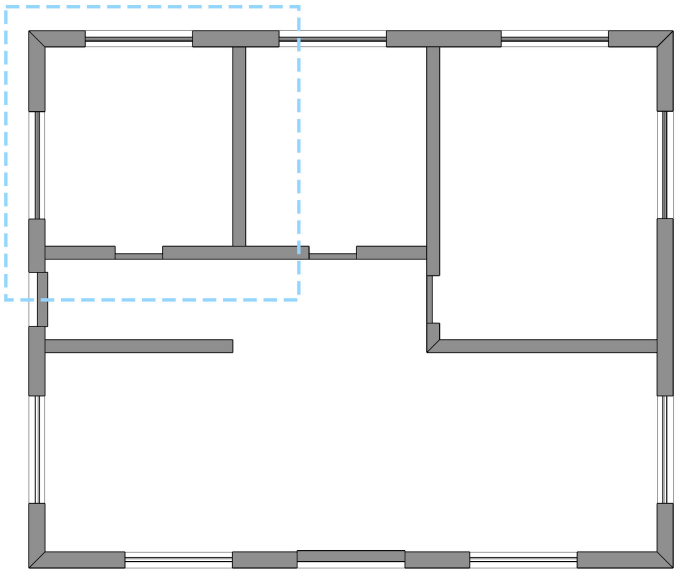


Figure 23: Complete floor plan of the FZK_Haus model with the area that is used in Figures 22, 24, and 25

The surfaces of the collected objects are then filtered with the help of the voxel assisted ray casting process, see Figure 25. This is done by populating each surface with a point grid. This point grid is constructed in the same manner as described in Section 3.4.1. From these points rays are cast to the centres of the surrounding non-intersecting exterior voxels. For every point on the surface the exterior voxels that fall within $1.5 \times voxelSize$ are selected as ray target points. Every ray is tested for an intersection with any of the surrounding meshed surfaces (except the surface from which it originates). If the ray intersects with anything, it is considered a hidden ray. If the surface has at least one ray that is not hidden, it is considered of importance and collected for further processing.

3.6.2 Surface refinement

The surfaces that were collected are at least partially exterior. However, these surfaces can still have parts that are interior, see Figure 22c. These interior parts need to be eliminated to be able to create a closed outer shell. During this surface refinement step, the surfaces are split and filtered.

Splitting complex surfaces can be a slow and unstable process. The process can be simplified by polyhedralisation of the surfaces prior to processing. As mentioned in Section 3.4.1 IFC supports both explicit and implicit geometry while CityJSON only support

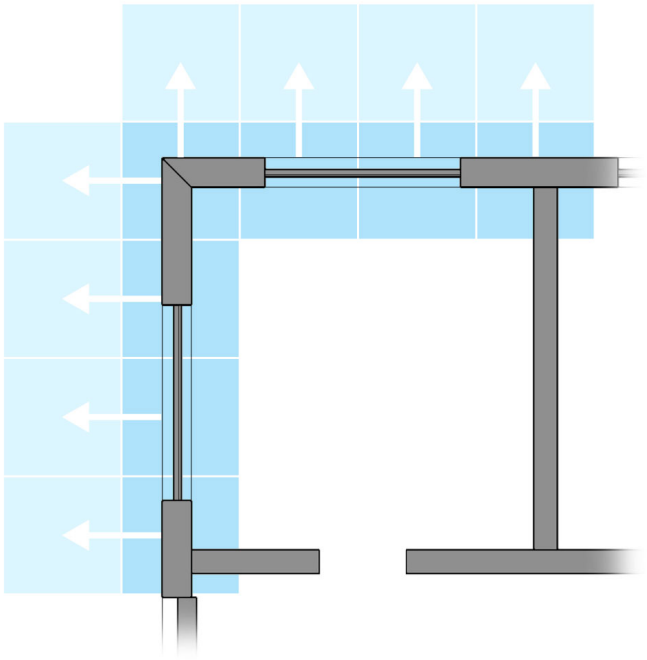


Figure 24: Plan view that shows how an example of the coarse filtering of objects. The intersecting voxels (dark blue), that neighbour voxels that are non-intersecting and external (light blue), intersect with object that have at least a single surface that is part of the exterior shell. Some of the interior walls are included in the selection because they partially intersect with a voxel that is queried.

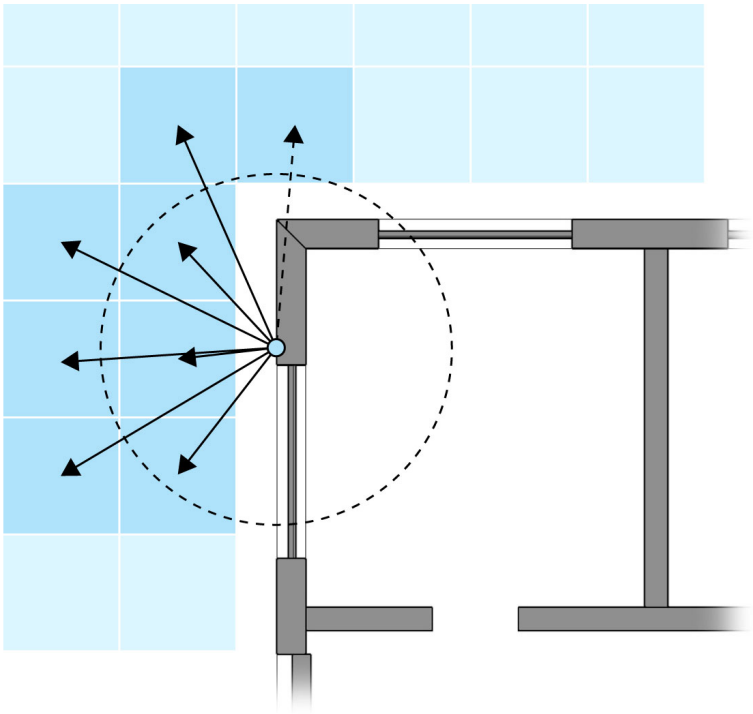


Figure 25: 2D plan view that shows how rays are cast to detect exterior faces. The blue point is the point from which the rays (black arrows) are cast to the centres of the surrounding voxels (dark blue) that fall within the $1.5 \times voxelSize$ range (the dotted ring). Although displayed in 2D in this image this process is executed in 3D and not 2D.

explicit geometry. So, the complex implicit surfaces have to be simplified regardless and doing this early in the process increases robustness.

The approximated surfaces are then split by the surrounding surfaces that it touches. This splitting process creates a pool of surfaces that are either completely exterior or interior. To detect if the surface is visible another voxel assisted ray cast process is used. This is similar as described in Section 3.6.1 and shown in Figure 25. However, for this process no point grid to cast from is required. Instead, a single point on the surface should suffice. From this point rays are cast to all the exterior voxels that fall within $1.5 \times voxelSize$ distance from the origin point. If a single ray is not hidden, the surface is considered to be completely visible.

The surfaces that are considered to be completely visible can be merged based on their surface type. Due to the complexity of the types three distinctions are made: doors, windows and others. Surfaces that touch, have the same normal, and are part of the same type group can be merged into a single surface.

The merged surfaces utilizing the other similar types will have to get a type for output

in the GIS file. This is easy when the merged surfaces are all of the same *IfcClass*. However, this is not always the case. To determine the surface type, the areas of all the input faces for a single merge process are computed. The surface type that is represented by the largest sum of areas is the type that is being used for the resulting surface.

The actual determination of the suitable GIS surface types was challenging. The documentation of CityGML 3.0 [Gröger et al., 2012] is open for interpretation. This mostly comes to light in the LoD3.2 abstraction due to the complex nature of the output shapes. There are many cases that are ambiguously documented in the CityGML documentation. An example of this is the roof structure. According to the CityGML3.0 documentation a *RoofSurface* is ".. a surface that delimits major roof parts of a construction". However, When a roof has overhang, there is an underside of the roof that is exterior. Is this also a surface "that delimits major roof parts of a construction" or is it an *OuterCeilingSurface* which is described as "... a surface that belongs to the outer building shell with the orientation pointing downwards". Both surface types seem to be suitable in this case.

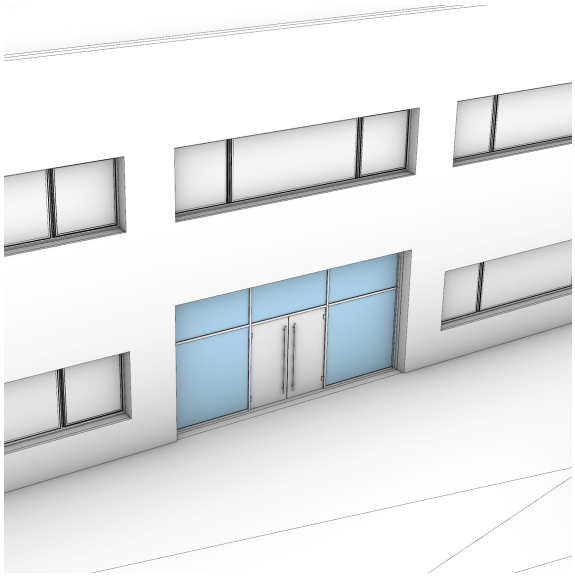
Because this research is focused on the abstraction of the geometry, it was chosen to keep the semantic surface type logic simple and lightweight. Therefore, there has been no extensive research on finding a solution to this issue. However, it is important to note that this issue exists and should be addressed in the future.

3.6.3 Acquiring attribute data

To be able to merge the surfaces based on their type, but also to correctly store it as a CityJSON surface, the attribute data is required. In most cases the *IfcObject* class is utilized, e.g. *IfcWall* objects are walls and *IfcRoof* objects are roofs, see Table 2 for a full overview of the BIM to GIS types. However, in certain cases using the *IfcObject* class is not enough. If a surface originates from an *IfcWindow* or *IfcDoor* object it can easily be typed as either a window or door respectively. However, windows and doors can also be part of *IfcCurtainWall* objects. *IfcCurtainWall* objects are usually constructed out of *IfcPlate* and *IfcMember* objects, see Figure 26. If a door is present in a curtain wall, it is modelled as an *IfcDoor* object. If a window is present, it is modelled as an *IfcPlate* object. Consequently, a method had to be developed to detect if an *IfcPlate* object is a window.

The most reliable way of detecting windows that are nested in *IfcCurtainWall* objects is by the material rendering properties which are stored in the *IfcSurfaceStyleRendering* class. This rendering class has a transparency attribute that ranges from 0 to 1. If the transparency is larger than 0.25 the surface is considered to be a window surface.

Relying on the *IfcMaterial* object's name attribute is an alternative option. This data is faster and easier to access. Often if the material represents glass the word "glass" or "glazed" is part of the material's name. However, this is not always the case. There can be many reasons for this. The major one being the language that is used in the



(a)

Type	Name
Plate	Systemelement:Verglasung:2543360
Material layer	Glas - klar
Plate type	Systemelement:Verglasung
Material layer	Glas - klar

(b)

Type	Name
Plate	System Panel:GLASS_10mm ZW:1086062
Material layer	Glass_ALUPROF_Clear
Plate type	System Panel:GLASS_10mm ZW
Material layer	Glass_ALUPROF_Clear

(c)

Figure 26: a) The window/door structure of the main entrance of the digital HUB model (its windows highlighted in blue) is modelled as an *IfcCurtainWall* where the windows are *IfcPlate* class objects. b) The model is modelled by a German party, so relying on the material name which is expected to be English (c) would not function.

IfcClass	CityJSON type
IfcDoor	All surfaces: *Door*
IfcWindow	All surfaces: *Window*
IfcPlate with glass like material	All surfaces: *Window*
IfcSlab	If below ground level: Horizontal: *FloorSurface* Other faces: *WallSurface* If Above ground level: Horizontal: *RoofSurface* Other faces: *WallSurface*
Other classes	*WallSurface*

Table 2: The translation logic when going from *IfcClass* to CityJSON surface type.

model, see Figure 26. If a language other than English is used the words for "glass" and "glazed" would have to be translated.

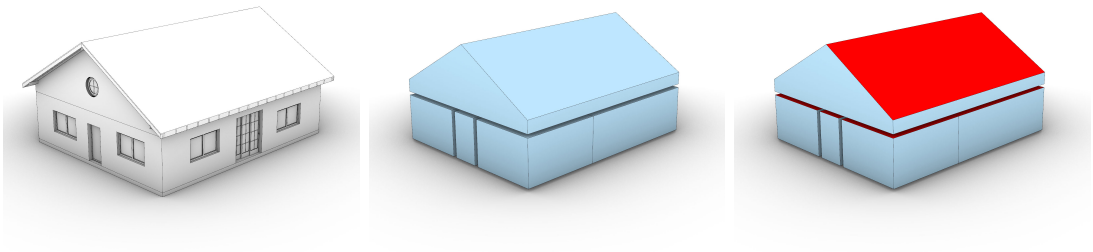
The way of detecting the windows passes through all the three options. First the class type is used. If the type is *IfcCurtainWall* or *IfcPlate*, the *IfcMaterial* class's name attribute is searched for. If this name contains the words "glass" or "glazed" the surface is typed as a window. If this is not the case the final option is followed and the *IfcSurfaceStyleRendering* class properties are used.

3.7 Interior geometric dependent abstraction

The interior extraction methods are developed for LoD0.2, 1.2, 2.2 and 3.2 interior room abstractions. Each room in the IFC model will be abstracted into a unique shape. The LoD0.2 abstraction of a room is limited to a projected ceiling outline while the LoD1.2, 2.2 and 3.2 room abstractions are fully volumetric abstractions.

The abstracted shapes representing the interior rooms are based on the *IfcSpace* objects. These *IfcSpace* objects are modelled similarly in the IFC model as their abstracted GIS representations will be, see Figure 27. They are both modelled as a shell representing the transition between tangible geometry and interior void. This makes it a fairly easy process to transition from IFC to CityJSON. However, relying on the *IfcSpace* data does bring some challenges. If there is no *IfcSpace* data present or if the geometry representation is incorrect the abstracted shape will also be missing or incorrect.

The room objects are stored in the GIS file as a child object relating to the story object at which they are located. This storey is found by using the relationship between the *IfcSpace* and *IfcStorey* objects. This means that if this relationship is incorrect, the resulting abstracted room object will be the child of the incorrect storey object. This



(a) (b) (c)

Figure 27: Comparison between the IFC interior space objects (b) and the CityJSON interior space objects (c) of the FZK_haus model (a). Aside from the surface types the geometric shape of the IFC spaces and CityJSON spaces are identical.

will however have no effect on the physical placement of the geometry, just its semantic relationship.

To create the LoD0.2, 1.2 and 2.2 interior abstractions the ceiling structure of the *IfcSpace* has to be isolated, see Figure 28. This process is similar to the roof structure detection described in Section 3.4.1 but slimmed down. We assume all *IfcSpace* objects are simple volumetric solids that only span a single storey. In practice this means that it is assumed all surfaces representing the ceiling structure are not overlapping with each other in the XY-plane. This assumption allows the detection process to be made more robust and computationally effective. However complex rooms/ceiling structures that do not adhere to the assumption could be detected incorrectly due to a simplified implementation.

The simplified process is executed per room by selecting all the surfaces representing an *IfcRoom* object and eliminating all the surfaces that have a normal with a Z-component of close to 0. Now the top surfaces can be detected with the help of ray casting. Because it is assumed no ceiling surfaces overlap, this can be done with a single ray cast per surface. For each ray it is evaluated if it intersects with any surface other than the surface from which it is cast. If the ray does not intersect with anything, the surface is considered part of the ceiling structure.

For LoD0.2, the ceiling surfaces are projected flat, merged and translated to the Z height of the lowest point of the original *IfcSpace* object. Similarly to the LoD0.2 roof outline of the exterior shell, this surface get the type of the top surface. In this case that is the *+ProjectedCeilingOutline* type. This is because the surface is based on the ceiling structure of the room and not based on the floor structure. In most cases these would be very similar, however in certain cases this could deviate.

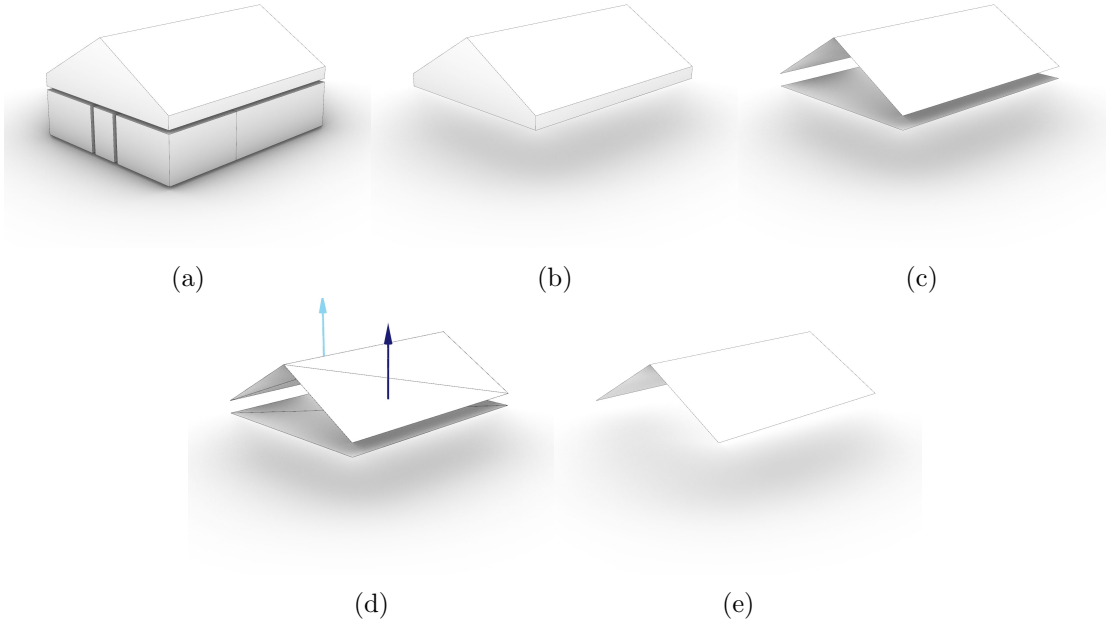


Figure 28: The steps that are taken for the ceiling detection. a) The starting shapes are the *IfcSpace* objects b) For a more clear visual one room is isolated. c) The vertical faces are discarded. d) A ray is cast from the surface the test for intersection. e) The surfaces from which a non-intersecting ray is cast are collected, these are the surfaces representing the ceiling structure.

To create the LoD1.2 abstraction the flattened merged surface that has been created for LoD0.2 is extruded upwards to the top height of the *IfcSpace* object's geometry. The type of the surfaces that have a normal with a Z-component of close to 0 are set to *InteriorWallSurface*. Of the two surfaces that are left, the type of the top surface is set to *CeilingSurface* and the type of the bottom one to *FloorSurface*.

LoD2.2 uses the un-flattened ceiling surfaces. These are extruded downwards to the lowest height of the original *IfcSpace* object. If the room has multiple surfaces representing the ceiling, e.g. an attic room underneath a gable roof, the extruded surfaces have to be merged together. Because we assume that no surfaces of the ceiling structure overlap, we can merge the surfaces using the same method as described in Section 3.4.3. The type of resulting shape's surfaces that have a normal with a Z-component of close to 0 are set to *InteriorWallSurface*. The type of the lowest of the remaining untyped surfaces is set to *FloorSurface*. The type of the rest of the surfaces is set to *CeilingSurface*.

The LoD3.2 abstraction shapes for the interior are not abstractions. They are almost 1:1 conversions of the *IfcSpace* geometry. As mentioned before, *IfcSpace* objects and CityJSON space objects are very similar, see figure 27. If the *IfcSpace* objects are modelled properly this will result in format compliant LoD3.2 interior abstractions. The

only exception is that IFC supports implicit geometry while CityJSON only supports explicit geometry. If implicit geometry is used to represent the surfaces modelling the room, the surfaces are approximated by polyhedral surface approximation. This creates a detailed approximation of the specific surface consisting of all flat surfaces.

Detecting surfaces representing walls is more of a challenge. Similarly as with the LoD1.2 and 2.2 abstractions, surfaces that have a normal with a Z-component of close to 0 are walls. However, angled walls can occur in LoD3.2. Consequently, there can be a case where a wall has a normal with a Z-component that is not close to 0. To be able to correctly detect those surfaces as walls an extra rule is added. If a surface normal falls within a 20 degree angle of a horizontal plane the surface is considered a wall.

Similarly, floor surfaces create a challenge to be detected as well. Both for the LoD1.2 and 2.2 case, there is only a single floor surface. The LoD3.2 shape can have multiple surfaces representing the floor. To detect this a ray cast is done from each surface upwards along the Z-axis. If it intersects with any other surface, except itself, it is considered a floor surface. All the remaining surfaces are marked as ceiling surface.

4 Implementation details

4.1 BIM2GEO application

The in Section 3 described methods have been implemented in a software application. This allows us to test the viability of these methods in an automated manner. The application that has been developed for this purpose is the open source IfcEnvelopeExtractor. The IfcEnvelopeExtractor is a cross platform (Windows & Linux) console application that takes a IFC model as input and automatically converts it to a CityGML/CityJSON file with minimal user assistance. The application is lightweight and can be utilized on local systems and servers. A pre-compiled windows or Ubuntu Linux versions can be acquired from the application's GitHub page. The utilized version for this research is V0.2.6.

The application is C++ based and utilizes IfcOpenShell 0.7 and OCCT 7.5.3 libraries for accessing the BIM data and processing the geometry. In conjunction to the IfcEnvelopeExtractor, the C++ library CJT was developed. CJT is an open source library that enables editing of CityGML/CityJSON files and the conversion of OCCT geometry to CityGML/CityJSON. This library is an integral part of the IfcEnvelopeExtractor, but can also be used independently in other C++ applications.

4.1.1 Supported I/O file formats

The IfcEnvelopeExtractor application accepts BIM models in the open Industry Foundation Classes (IFC) format. However, only a subset of IFC versions is currently supported. The list of supported IFC versions can be seen in Table 3. The developed application also adheres to the georeferencing standards for IFC4. For IFC2x3, the property sets (Psets) approach developed for IfcGref [Delft University of Technology, nd] are utilized. For IFC4x3 the same approach of IFC4 is applied.

Version	Name
2.3.0.1	IFC2x3 TC1
4.0.2.1	IFC4 Add2 TC1
4.3.2.0	IFC4x3 ADD2

Table 3: Supported IFC versions

IfcEnvelopeExtractor outputs CityGML models in the CityJSON V2.0 encoding. The CityJSON file is a single output file that contains all the abstracted shapes. To expand the ease of use, the application can also export copies of the abstracted shapes in Wavefront .obj and .STEP file formats. These files do not support LoD in a similar way as CityJSON does. To accommodate for this, the .obj and step file output are not a single file but a collection of files each representing a single (or part) of an LoD abstraction.

4.1.2 Supported LoD abstractions

The IfcEnvelopeExtractor can generate and output a large selection of LoD abstractions. These are the abstractions that are described in this paper, but the software can also generate a large group of experimental abstraction and non-standardized abstractions shapes that are not described in this paper. These range from simple shapes, like a combination of footprint restricted LoD2.2 with unrestricted roof structure, to extremely complex shapes, like a 1:1 conversion. In total the tool can output 17 different LoD: 8 standard and 9 non-standard. LoD0.4 is the only non-standard LoD covered in this paper.

4.1.3 Application Configuration

The application is controlled via configJSON files that define the user settings. To set up the application, the user is only required to set a minimum of 3 different settings. The input path(s), the output path and the desired LoD. If no other settings are configured in the file, they are either automatically chosen by the application or a default value is used. This allows the user to only set 3 settings or expand to the 30+ settings that are available to fine-tune the application for their unique need.

Setting up such a configJSON can be a daunting task for a normal BIM and/or GIS user that has no programming background. To facilitate a non-programmer user a python based GUI was also developed. The GUI functions as a front that exposes a selection of the available settings for easy configuration, see figure 30. After the offered settings are configured, the GUI creates a configJSON, feeds it to the correct executable, and deletes the configJSON. The GUI is available from the GitHub page as both a Python file and a pre-built Windows executable.

The GUI can also be used as an initial step for the development of custom configJSON files. Instead of feeding the file directly to the executable it can also only generate and store a configJSON file. This configJSON file can then be edited with a text editor. This can be of use if more complex settings are desired than the sub-set exposed by the GUI. This can only be done via the configJSON file. The more complex settings that are only accessible from the config file are settings such as the voxel intersection logic, custom model rotations and max thread count.

The configJSON can be a complex file. To make it easier for the user, but also the parser, most of the settings are clustered in groups. Each group covers their own topic of settings: the desired file paths, the desired LoD output, voxel related settings, IFC related settings, CityJSON related settings, additional outputs format and the tolerances. An example of a full configJSON can be seen in figure 29.

```
{
    "Filepaths": {
        "Input" :
        [
            "path to IFC file",
            "Potential path to other IFC file"
        ],
        "Output" : "path to export (City)JSON file",
        "Report" : "path to export report JSON file"
    },
    "LoD output": [ 0.0, 0.2, 0.3, 1.0, 1.2, 1.3, 2.2, 3.2, 5.0 ],
    "Voxel":{
        "Size": 1,
        "Store values" : 0,
        "Logic" : 3
    },
    "IFC": {
        "Rotation angle" : 90,
        "Default div": true,
        "Ignore proxy": true,
        "Div objects" : [],
        "Ignore voids" : 0,
        "Simplify geometry" : true,
        "Ignore simplification" : [],
        "Correct placement" : true
    },
    "JSON" : {
        "Footprint elevation": 1,
        "Footprint based" : 0,
        "Horizontal section offset": 0,
        "Generate footprint": 1,
        "Generate roof outline": 1,
        "Generate interior": 0,
        "Generate exterior": 1,
        "Generate site": 0,
        "Georeference" : 1,
        "Merge semantic objects": 1,
    },
    "Output format" : {
        "STEP file" : 1,
        "OBJ file" : 1
    },
    "Tolerances" : {
      "Spatial tolerance" : 1e-6,
      "Angular tolerance" : 1e-4,
      "Area tolerance" : 1e-4
    },
    "Generate report": 1,
    "Threads": 12
}
```


Figure 29: Example configJSON with all the available settings

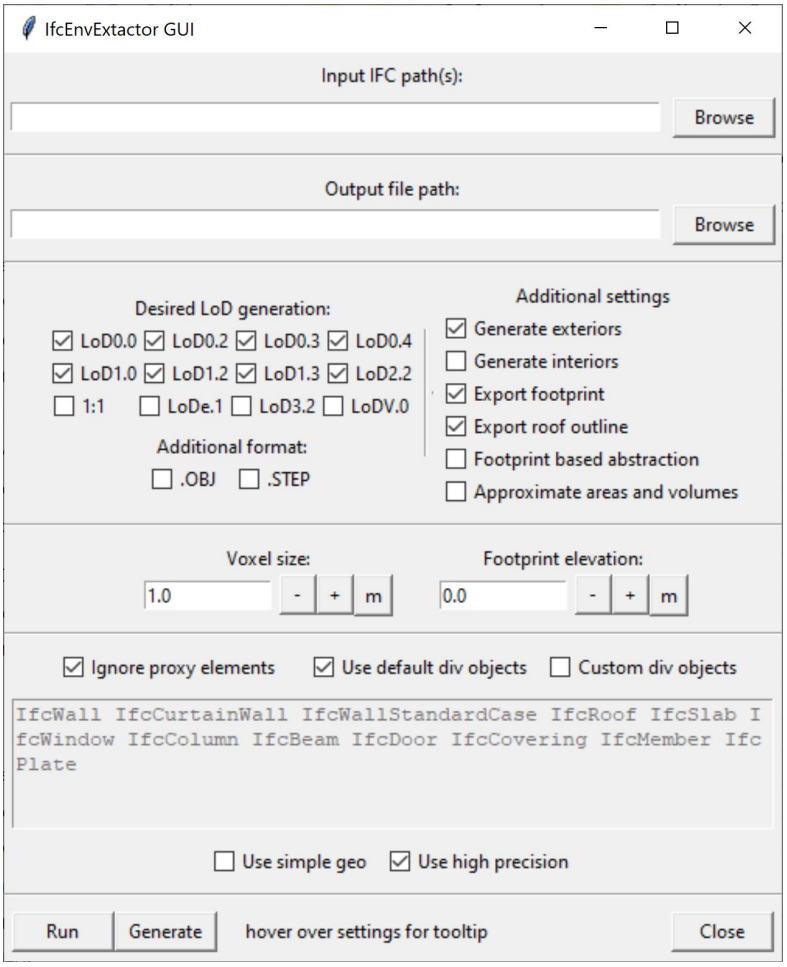


Figure 30: GUI of the IfcEnvelopeExtractor v0.2.6.

File name	Source	IFC version	Object count	Storeys	X, Y, Z size (m,m,m)
AC20-FZK-Haus	IAI/KIT	IFC4	102	2	15.0, 13.0, 6.5
AC20-Institute-Var-2	IAI/KIT	IFC4	896	5	64.4, 42.2, 15.4
FM_ARC_DigitalHub	RWTH Aachen	IFC4	775	3	64.3, 35.6, 12.1
PRAHA_GO_V5	CHEK	IFC4	3588	7	91.4, 61.2, 26.1
Demo_Lisbon_2025	CHEK	IFC4	591	7	33.3, 11.4, 19.4
Demo_Ascoli Piceno_v2	CHEK	IFC4	3040	8	61.6, 57.7, 22.1

Table 4: Summary of the models used to test performance.

4.2 Used models

The models used to test the performance of the developed methods were picked to represent a wide range of different types of buildings. Some of these are use cases in the CHEK project (see introduction). The models range from very simple small single housing models to moderately complex multiple storey office buildings. Table 4 summarises the main characteristics of the models. The models are visualised in Figure 32.

Some of the models required some editing prior to processing. This is because the models did not adhere to the input requirements of both the methods and the IfcEnvelopveExtractor. As was mentioned in Section 3.1.2 objects are selected based on their type. This works well as long as the IFC model is constructed using the proper types. However, the site of the model is occasionally modelled utilizing incorrect types. For example, the FM_ARC_DigitalHub model includes a road that is modelled as an *IfcSlab* object, see Figure 31. If this kind of improper site modelling occurred, the object's type was changed or the object was removed from the model.

4.3 Used settings

The used settings to generate the abstractions has set as uniformly as possible. For the majority of the results the ConfigJSON was created using the GUI. So no deep settings were changed. Most of the settings were identical between the models:

- Active Desired LoD generation: LoD0.0, 0.2, 0.3, 0.4, 1.0, 1.2, 1.3, 2.2 and 3.2
- Active Additional settings: Generate Exterior, Generate interior, Export footprint, Export Roof outline
- Active Other settings: Ignore proxy elements, Use default div objects, Use high precision
- Voxel size = 0.3 metre
- Footprint elevation = 0 metre

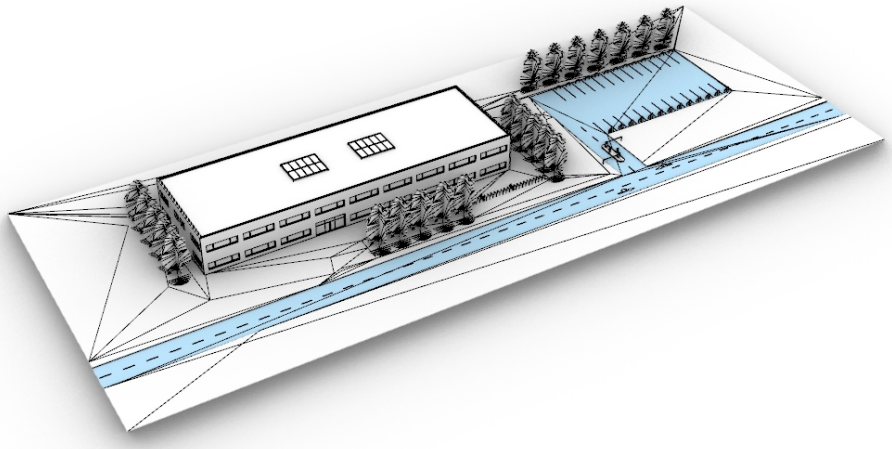
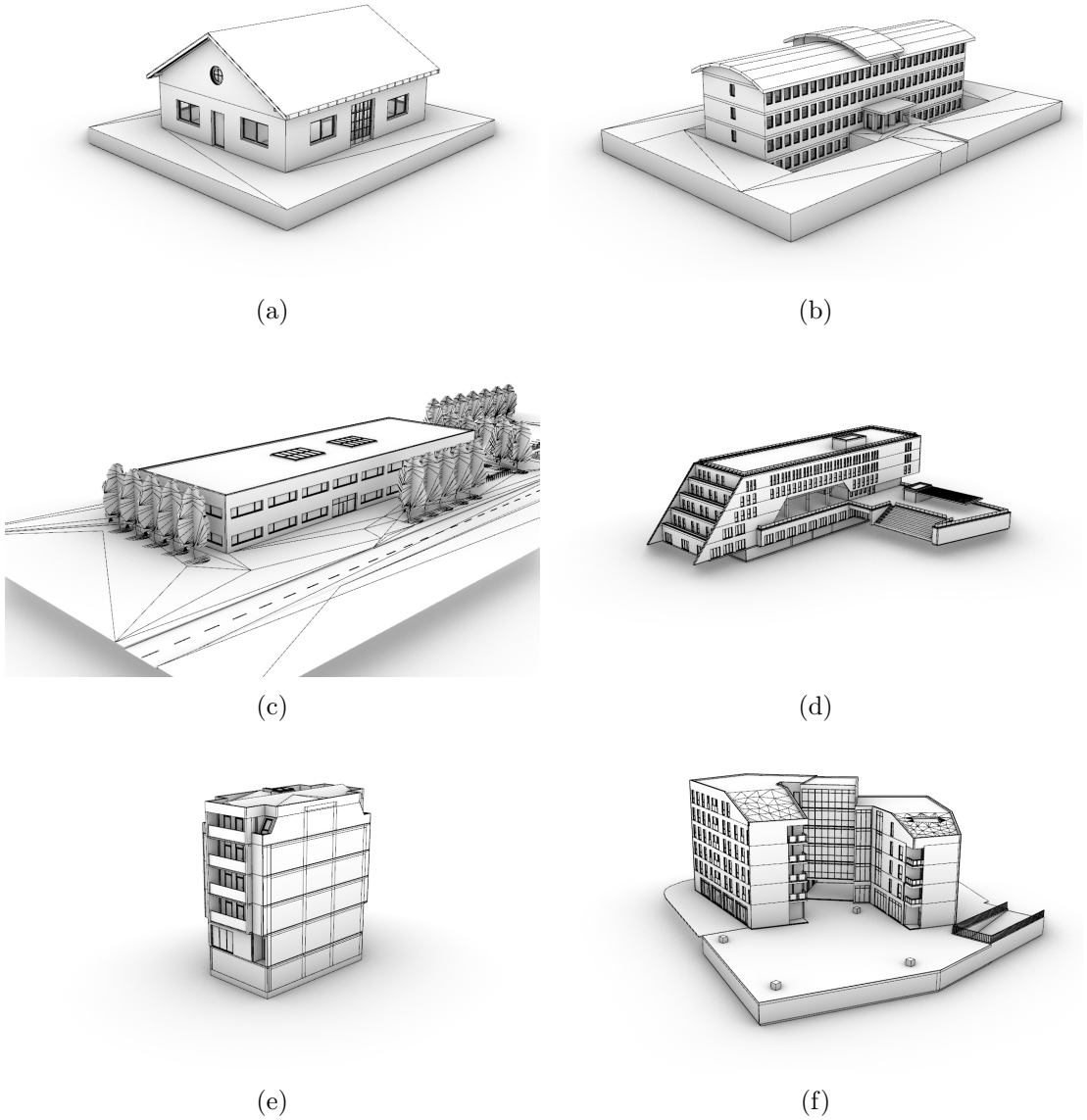


Figure 31: The FM_ARC_DigitalHub model. highlighted in blue is a road that is not part of the building but is incorrectly modelled as an *IfcSlab* object that is part of the *IfcBuilding* object. This object was manually removed from the model prior to processing. If this object was included it would incorrect influence the outcome.

Although uniformity in the settings was attempted when running the tool for all test models, there are some deviations in the settings that were deliberately chosen. These deviations can be split into two groups, model specific deviations and evaluation specific deviations. Model specific deviations cover the changes that were made in the input settings to accommodate the proper processing of the input model. Evaluation specific deviations cover the choices that were made in the input settings to accommodate the different evaluations (see Section 5). The model specific deviations are covered here. The evaluation specific settings are covered in Section 5.

4.3.1 Model specific settings

For the PRAHA_GO_V5 model the proxy elements are not ignored and the footprint elevation is set to 250.5 metre. This was done because important elements of the building were modelled as *IfcBuildingElementProxy* that otherwise would be ignored, such as the external staircase/grandstand. For the Demo_Lisbon_2025 model, the voxel size has been set to 0.15 metre. This was done because there are some narrow parts in the model that otherwise would not be correctly registered.



(a) (b) (c) (d) (e) (f)

Figure 32: The different models that are used to evaluate the performance of the developed methods. a) AC20-FZK-Haus model, b) AC20-Institute-Var-2, c) FM_ARC_DigitalHub, d) Demo_Lisbon_2025, e) Demo_Lisbon_2025, f) Demo_Ascoli Piceno_v2. The figures representing these models are not to the same scale.

5 Results, Evaluation & Discussion

This chapter covers three different types of evaluations: the geometric accuracy, the semantic accuracy and the model complexity. The three evaluations are described in Sections 5.1 to 5.3. Each section starts with a sub-section that covers how the evaluation is set up and what the results are, followed by one or more sub-sections discussing the results of the evaluations.

5.1 Geometric accuracy

5.1.1 Evaluation set up

Setting up tests to quantify the quality and accuracy of the generated geometry is challenging. The LoD abstractions do not have a proper ground truth. GIS models that are used in practice are not candidates to fulfil this role. For example an LoD1.3 abstraction that comes from a GIS database will presumably be less accurate than the LoD1.3 abstraction that is based on a BIM source. The GIS sourced model will most likely be based on in field measurements, containing occlusion and noise, issues that the developed methods do not have to deal with. Additionally, many of the LoD abstractions that the method can generate are not available from GIS databases.

We can, however, visually compare the input IFC model to the abstracted shapes. In this way, it can be evaluated if the shapes of the abstractions are as expected and if they show any elements that the input models do not show. Additionally, we can augment the visual inspection with a test to see if the volumetric abstraction shapes are closed volumes.

Instead of assessing the output in binary form as either 1 (correct/as expected) or 0 (incorrect/with issues), we give three scores. A score of 1 signifies an expected output based on the input model. Additionally, volumetric models should also be closed volumes. A score of 0.5 signifies a model with minor issues. The model visually resembles the expected output based on the input model, but there are some unexpected issues. For example, in a volumetric model, there could be some minor self intersections that prevent it from being closed. These models are expected to still be useable to some extent in GIS analyses. A score of O signifies a model with issues that would prevent it from being usable in any GIS analyses. This ranges from missing faces to output shapes that do not resemble the input IFC model.

The settings that were described in Section 4.3 were used to generate the output for this evaluation.

File name	LoD0.0	LoD0.2	LoD0.3	LoD0.4	LoD1.0	LoD1.2	LoD1.3	LoD2.2	LoD3.2
AC20-FZK-Haus	1	1	1	1	1	1	1	1	0.5
AC20-Institute-Var-2	1	1	1	1	1	1	1	1	1
FM_ARC_DigitalHub	1	1	1	1	1	1	1	1	0.5
PRAHA_GO_V5	1	1	1	1(0)*	1	1	1	1(0.5)*	0
Demo_Lisbon_2025	1	1	1	1	1	1	0.5	0.5	0.5
Demo_Ascoli Piceno_v2	1	1	1	1	1	1	0.5	0.5	0

Table 5: The success rate of the exterior shell extraction. *Output differs per execution between the first value and the value between brackets.

File name	LoD1.2	LoD1.3	LoD2.2
AC20-FZK-Haus	1	1	1
AC20-Institute-Var-2	1	1	1
FM_ARC_DigitalHub	1	1	1
PRAHA_GO_V5	1	1	0.5
Demo_Lisbon_2025	1	1	0.5
Demo_Ascoli Piceno_v2	1	0.5	0.5

Table 6: The success rate of the footprint restricted exterior shell extraction.

5.1.2 Results

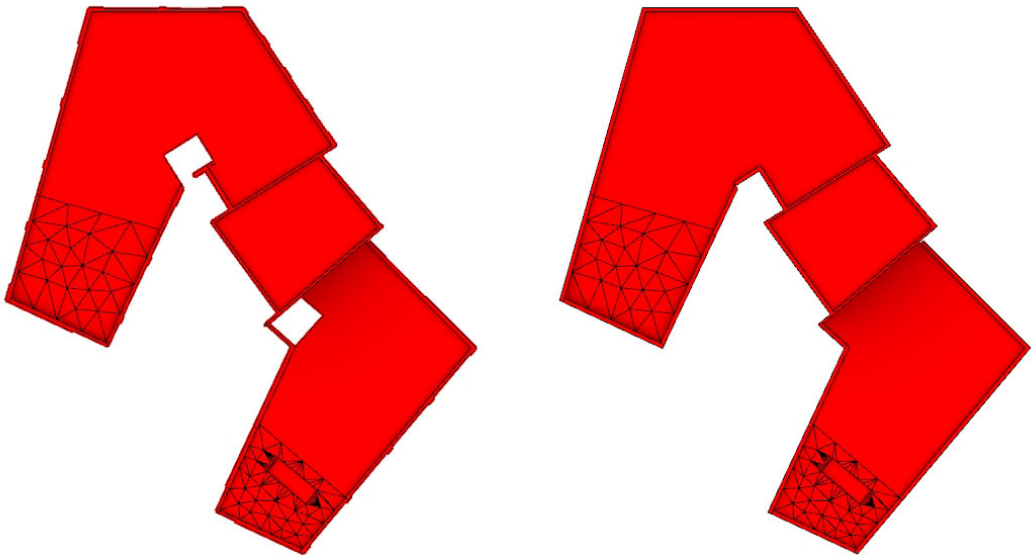
The results of this inspection can be seen in Tables 5 and 6. The exterior abstraction scores in Table 5 show that the non-volumetric abstractions perform very well. The methods applied seem robust and accurate. However, the scores for the volumetric models are not all good. The scores of LoD1.0 and LoD1.2 are very high, but LoD1.3 and more complex abstractions show lower scores. The scores for the footprint restricted volumetric abstraction, see Table 6, show similar results. From these scores, it can be concluded that the more complex the abstraction shape is, the lower the scores are. The interior abstractions all performed as expected (1), regardless if the output is 2D, 2.5D or 3D.

5.1.3 Discussion of the results

Propagation of IFC model related issues The results summarized in Tables 5 and 6 do not always cover the whole story. Even when the abstraction scores 1, the

abstracted model could still have issues that make it unusable. This is because these scores are computed based on the output created from the input model, which might be faulty. If the IFC model contains unexpected issues or uncertainties, these can lead to undesirable output, even if the methods implemented in the IfcEnvelopeExtractor tool work as they should. The Demo_Ascoli Piceno_v2 model shows two examples of these input related issues (Figure 33).

Firstly, the tool only utilises the selected space dividing objects as a split between the interior and exterior. The Demo_Ascoli Piceno_v2 model has staircase shafts that can be accessed via an underground parking lot, which is connected to the exterior of the model. The entrance to the staircases and the parking lot are openly connected to each other. Consequently, there is no clear division between which space is part of the interior and which space is part of the exterior. The staircase shafts and all directly connected spaces are considered exterior by both the tool and the methods. In theory, they are, because there is no barrier between the interior and exterior. But most likely, this is not expected by the user. This behaviour becomes apparent in different LoDs: The LoD3.2 exterior, the LoD0.3 storeys, the LoD0.2 and 0.3 footprints and the footprint restricted LoD1.2, 1.3, and 2.2.



(a) (b)

Figure 33: a) Top view of the footprint restricted LoD2.2 abstraction of Demo_Ascoli Piceno_v2. b) Top view of the LoD2.2 abstraction of Demo_Ascoli Piceno_v2. The footprint restricted abstraction has two cut outs in the model caused by the staircase shafts in the footprint that are considered exterior. Although this is correct behaviour a possibly more desirable result would be if these staircase shafts were interior and filtered out of the footprint.

A solution for this would be a direct fix in the IFC file. However, if this direct connection between the shafts and the exterior is correct and required for the design, an alternative solution should be offered. A potential improvement could be to offer an option that could rely on other IFC classes to identify the split between interior and exterior.

The second issue that the Demo_Ascoli Piceno_v2 model exposes can be clearly noted in the LoD0.3 storey abstraction, but is also present in the LoD3.2 exterior abstraction. The IFC model has gaps ($\geq$ 1cm) in its facade due to modelling errors. This causes issues in the high and very high geometric dependent abstractions. The LoD0.3 storeys abstractions show therefore some areas that are incorrectly typed as *OuterFloorSurface* while they are actually interior, see Figure 34. Modelling errors that result in gaps of this size can be considered as an user error and should be fixed in the input IFC file.

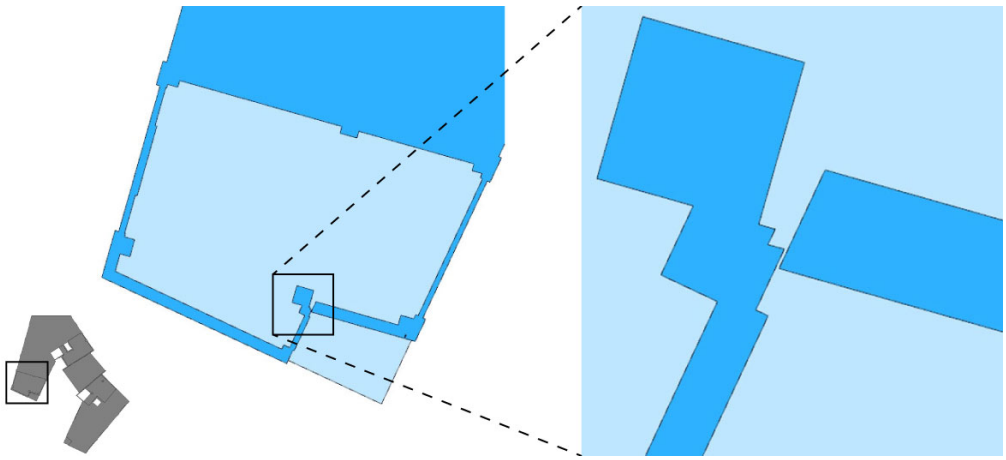


Figure 34: The LoD0.3 interior ground floor abstraction of the Demo_Ascoli Piceno_v2 model shows an incorrectly detected interior surface. This is caused by a gap of around 1 cm between one of the columns and doors.

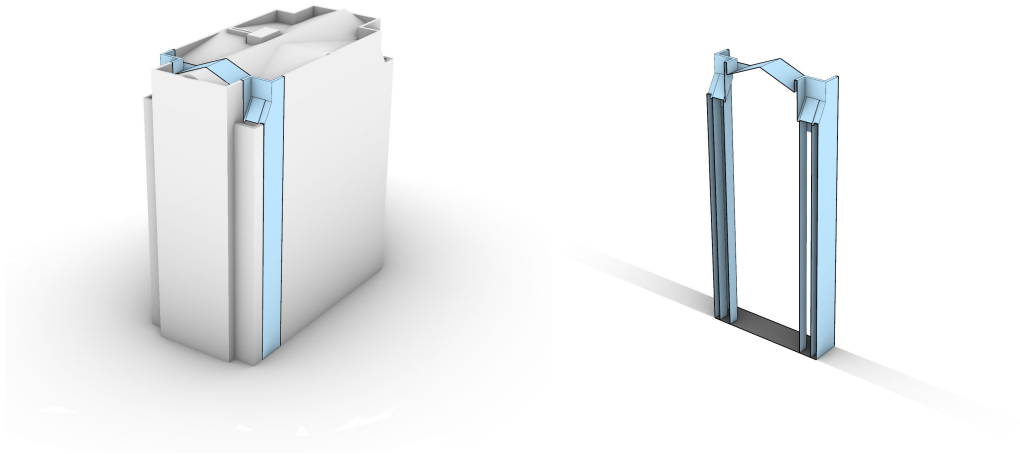
Tolerances/precision The volumetric abstractions show two different issues that can occur which are closely related. The first issue is incorrect vertical face elimination in LoD1.3 and 2.2. The second one is incorrect trimming of surfaces in LoD3.2. Both are related to tolerances in the input IFC models in relation to the Boolean operations that are used.

The tool utilizes a precise tolerance of 1e-6m (0.001mm). IFC allows the use of such a precise value, however, the models are made by humans. So even if the way the data is stored in the model can be very precise, it might be modelled with larger errors. At a tolerance/precision of 1e-6m, small gaps are not easily noticed by modellers, nor are they important for most BIM-related cases. However, the IfcEnvelopeExtractor does behave differently when these gaps are present in the model.

Both the Demo_Lisbon_2025 model and Demo_Ascoli Piceno_v2 model have such small

gaps in the roof structure. During the creation of LoD1.3 and 2.2, the roof representing shapes are extruded downwards, see Section 3.4.3. Afterwards, the vertical faces are filtered and trimmed to create a solid shape. Due to the small gaps in the surfaces, which are used to create the extrusions, the vertical faces do not always rest against each other. These small gaps prevent the vertical faces to be correctly filtered and trimmed. The very small tolerance and the gaps do not interact well with the Boolean and similarity processes. So, remnant faces are left in the model, preventing a proper closed solid to be formed, see Figure 35.

As can be seen in Table 5, the LoD0.3 and 0.4 of both the Demo_Lisbon_2025 model and Demo_Ascoli Piceno_v2 model have a perfect score even though these small gaps are present. This is because the precision of the output GIS files is 1e-3m (1mm). The gaps causing these issues are often smaller than the GIS precision. So, due to their extremely small size, they are not conveyed in the GIS output of these LoDs.



(a) (b)

Figure 35: b) A section slice of the Demo_Lisbon_2025 LoD2.2 abstraction model (a) that clearly shows one of the remnant surfaces that has been incorrectly exported. This should be a hollow ring, but a group of surfaces originating from the diagonal roofing structure are preventing it from being truly hollow.

Incorrect trimming caused by too precise tolerances can be noted in almost all LoD3.2 abstraction shapes. Even for the simplest building that has been evaluated, the AC20-FZK-Haus model, some surfaces are not correctly trimmed down. This usually results in a situation where the abstracted GIS model is watertight, but not a solid. It is prevented from being a solid by a set of untrimmed surfaces poking into the interior of the shape, see Figure 36. In all evaluated LoD3.2 models that did not score a perfect 1, this incorrect trimming occurs at certain places. The areas around windows and doors

seem particularly prone to this issue. The simplification of complex objects helps with reducing the occurrences of this issue. However, curtain-walls are not simplified by this simplification process and we often see issues here as well (Figure 37).

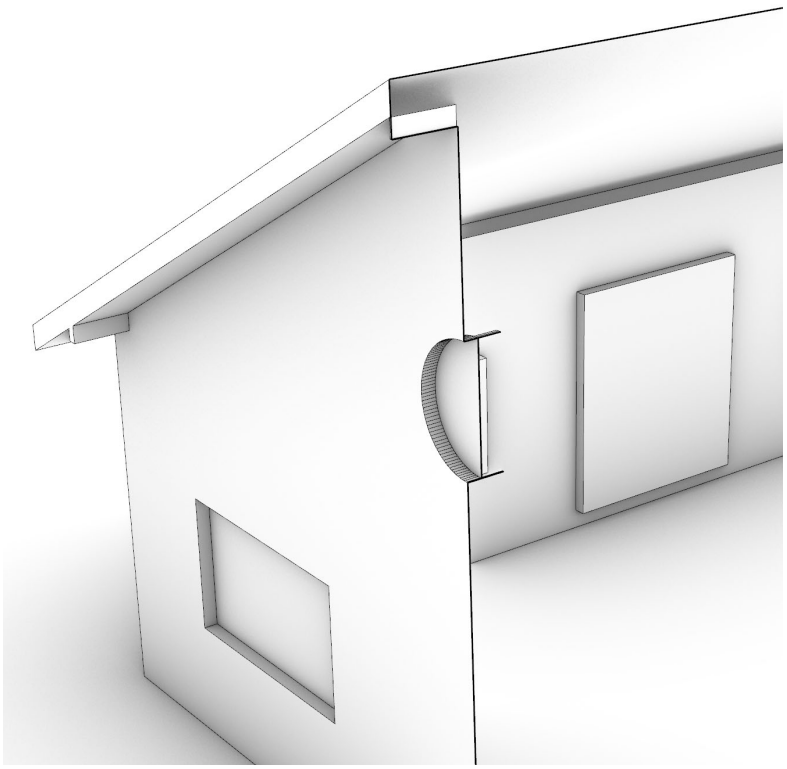


Figure 36: A section of the LoD3.2 abstraction of the AC20-FZK-Haus model that shows some of the incorrectly trimmed surfaces at the round window. These surfaces should be trimmed and the interior part should have been discarded.

The PRAHA_GO_V5 model and Demo_Ascoli Piceno_v2 model show very badly performing LoD3.2 abstractions. This is because these abstractions not only have the incorrectly trimmed surfaces, but also have some surfaces that are completely missing. Even though the missing surfaces are a different issue, the main cause is the same: incorrect surface trimming. The surfaces are missing due to the second ray casting process, which is covered in Section 3.6.2. The second ray casting process that occurs after the trimming is fairly lightweight. It can be so lightweight because it was assumed that when a surface was split, the resulting sub-surfaces will be either completely interior or exterior. This assumption is true if the surface is correctly split. However, if it is not, it can still be partially interior and partially exterior. In that case, the location of the point from which the rays are cast will determine if it is considered interior or exterior, even though it is actually both. Regardless of the outcome, the LoD3.2 abstraction will have issues: either the surface is missing, or the surface partially sticks into the solid.

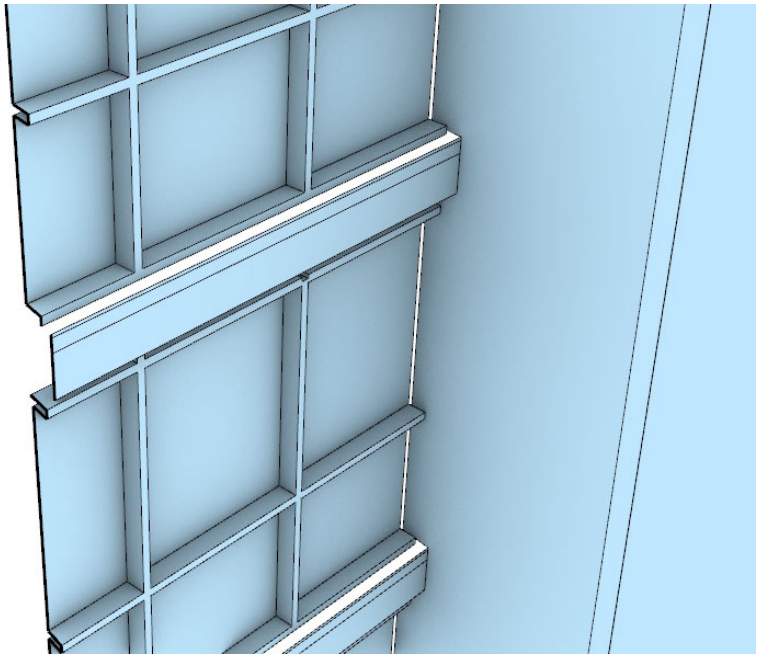
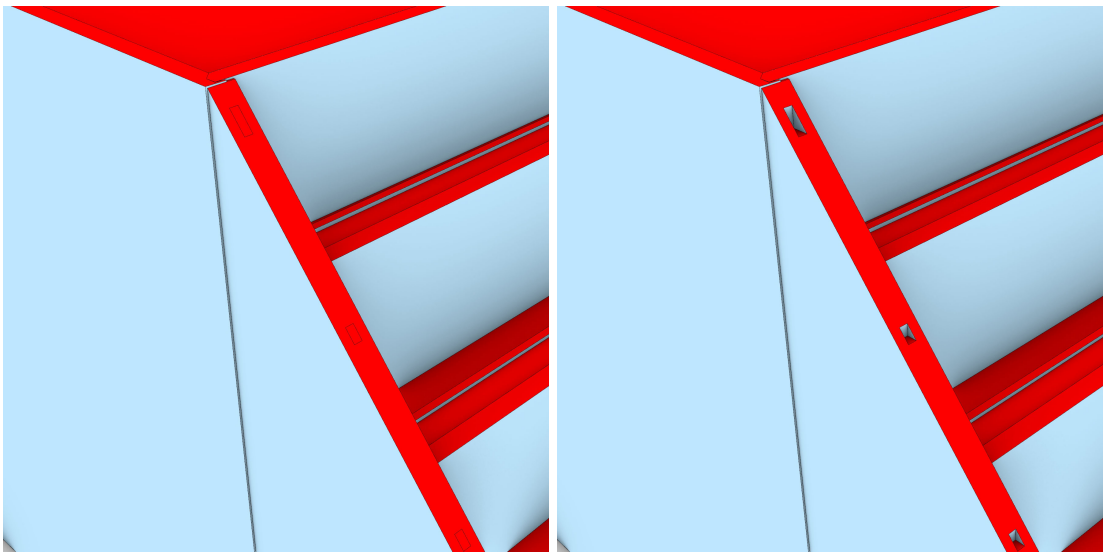


Figure 37: Isolated section of an LoD3.2 external abstraction showing a curtain wall. The detailed geometry can cause issues in the implementation which results in surfaces being incorrectly trimmed or missing.

The issues that are encountered in LoD1.3, 2.2 and 3.2 could possibly all be avoided by processing the IFC objects with a less precise tolerance. A tolerance of 1e-4m (0.1mm) is still precise, but could possibly avoid all, or a set of the small gaps that influence the tool. However, it should further be investigated if this could cause other issues. It is important to note that the issues covered here are implementation issues. The method could however improve if a change in tolerances was incorporated/covered.

Unpredictable behaviour The PRAHA_GO_V5 model also shows another issue that is not related to tolerances: unpredictable outcomes. The LoD2.2 output can be either a closed shape or not. This can differ per individual execution of the tool while using exactly the same settings. We assume this is caused by an improper implementation of a surface simplification process during the LoD0.4 creation. The surfaces of the LoD0.4 roof structure seem to always be properly detected. Before further processing, the LoD0.4 roof structure goes through a simplification step. This eliminates redundant vertices and edges. However, occasionally some small inner rings are incorrectly removed from the roof surfaces, see Figure 38. If so, the LoD0.4 roof surfaces can now overlap each other, leading to erroneous surfaces. The resulting LoD2.2 abstraction based on these LoD0.4 roof surfaces is not a closed solid. The method and implementation do not anticipate that this can occur.



(a) (b)

Figure 38: The LoD2.2 abstraction of the PRAHA_GO_V5 model shows unpredictable behaviour. a) The process can output an incorrect result when small inner rings are ignored. b) The process can also output a correct result when the inner rings are kept. This is an issue that can occur in the LoD0.4 abstraction and is passed on to LoD2.2

5.2 Semantic accuracy

5.2.1 Evaluation set up

There is no proper ground truth to quantify the quality of the semantic accuracy. This issue was already addressed in Section 5.1. The issues here are very similar. The only difference is that instead of inaccurate or missing geometry, the GIS sources containing the ground truth can also have incorrect or missing surface types.

It is, however, possible to visually evaluate the surface types of each LoD abstraction and to check if their type adheres to the description in the CityGML3.0 encoding's standard data dictionary [Kolbe et al., 2021].

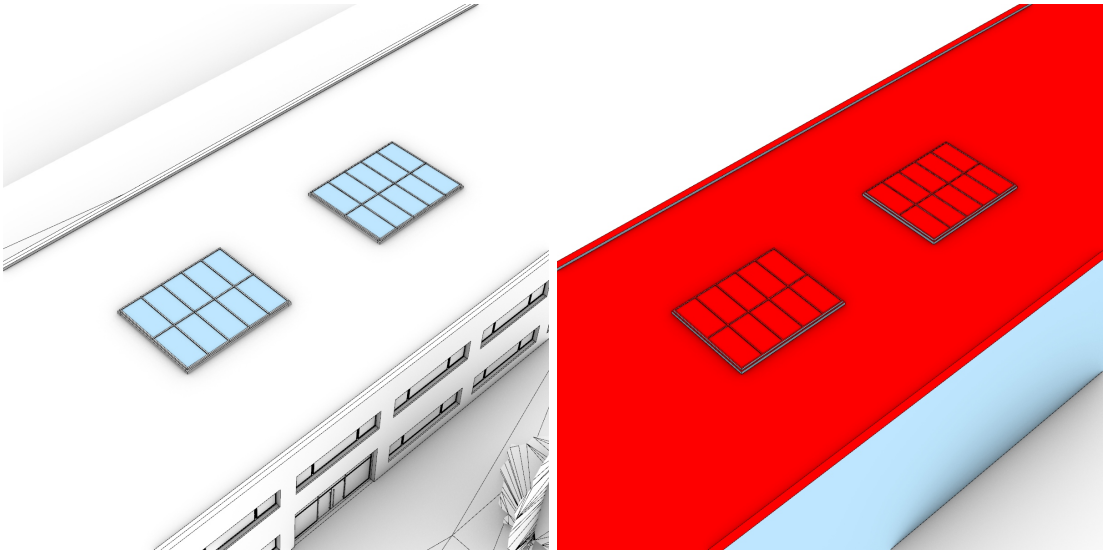
5.2.2 Results

The CityGML 3.0 encoding standard [Kolbe et al., 2021] can be vague in certain situations, as was noted in Section 3.6.2. This vagueness mostly creates issues in complex shapes, such as LoD3.2. The evaluation of the LoD3.2 exterior shapes therefore requires interpretation, which makes this evaluation subjective. Consequently, it was decided to omit the LoD3.2 exterior from the results. This issue was not encountered during the evaluation of the LoD3.2 interior shapes. These shapes are a lot simpler than the exterior shapes. Due to this, the LoD3.2 interior evaluation has been included.

The tests showed that the surface types of all abstractions are as expected when compared to the CityGML 3.0 feature type documentation. The success rate of both the interior and exterior abstractions show that, even though the way that the surface types are determined is very simple, they are also accurate. Even on models that were not true solids, such as the Demo_Lisbon_2025 model, the surface types could still be accurately determined. Note that the surfaces that were incorrectly modelled in the input IFC file were not included here, since these surfaces are not supposed to be there.

5.2.3 Discussion of the results

Missing roof windows Even though the surface types were technically correct in the FM_ARC_DigitalHub and Demo_Lisbon_2025 abstracted models, there is something to be noted. Both these models have windows that are part of the roofing structure. The FM_ARC_DigitalHub model has two large planar skylights and the Demo_Lisbon_2025 model has smaller angled windows in the roofing structure. Due to the simplicity of the rules that are used to determine the surface types, these were not classified as windows in the abstracted shapes, see Figure 39. LoD1.3 officially requires no surface types and LoD2.2 does not require the surface types that signify if a surface is a window or a door. However, the value of the models could be enlarged if window surfaces that are part of the roof structure can be identified as such.



(a) (b)

Figure 39: a) The FM_ARC_DigitalHub model has windows that are part of the roofing structure, highlighted in blue. b) These windows are not conveyed to the LoD2.2 abstraction. Instead the surfaces are typed as *RoofSurface*. This behaviour adheres to the description of LoD2.2.

5.3 Model complexity

5.3.1 Evaluation set up

To test the effect of the abstraction methods on the file size complexity, a set of evaluations is made.

The first evaluation compares the size of the different output files with the original file size. Table 7 compares the size of an 1:1 converted .obj file, the combined CityJSON abstraction containing nine abstracted representations of the model (LoD0.0, 0.2, 0.3, 0.4, 1.0, 1.2, 1.3, 2.2, and 3.2 interior and exterior) and the LoD3.2 CityJSON abstraction to the original file. The 1:1 .obj file was added to this evaluation to include the file size of the input model with all explicit geometry. Similar to .obj, CityJSON only allows explicit geometry while IFC also allows implicit geometry. This .obj model/file is also used for the other evaluations in this section. The actual file sizes which are used for this comparison can be found in the Appendix in Table 11.

The second evaluation covers the geometric complexity of the different LoD abstractions. To quantify the complexity, the number of triangles that result from a triangulation/meshing of each of the models is counted. Table 8 compares the number of triangles of the exterior abstractions with the 1:1 OBJ conversion of the IFC model. To compute the number of triangles of the CityJSON abstractions, the shapes were triangulated

File name	OBJ file size (%)	CityJSON combined file size* (%)	CityJSON LoD3.2 file size (%)
AC20-FZK-Haus	14.43	0.43	0.39
AC20-Institute-Var-2	39.63	8.71	8.06
FM_ARC_DigitalHub	303.70	1.35	0.78
PRAHA_GO_V5	3 751.33	7.77	7.11
Demo_Lisbon_2025	548.52	1.26	0.65
Demo_Ascoli Piceno_v2	800.52	4.82	2.19
Average	909.69	4.05	3.20

Table 7: Comparison of the file size of the 1:1 OBJ conversion, the combined output GIS file and the output GIS file containing LoD3.2 only to the input IFC file. All the files contain both interior and exterior abstractions. *The combined output file contains LoD0.0, 0.2, 0.3, 0.4, 1.0, 1.2, 1.3, 2.2, and 3.2.

using Rhino3D. The actual triangle counts, which are used for this comparison can be found in the Appendix in Table 13.

The interior abstractions were not included in this evaluation because not all LoD abstractions that are generated have interiors. Additionally, not all LoD have the same interior elements modelled. For example, the LoD0.2 interior abstraction has floor and rooms, while the LoD1.2 interior abstraction only has rooms included. Including the interior would thus influence the triangle count without directly reflecting the model's geometric complexity.

The third evaluation covers the effects of the complex object simplification process. This process was covered in Section 3.1.3. Table 9 compares three different approaches to simplifying *IfcWindow* and *IfcDoor* objects to the original triangular polygon count. To quantify the complexity, we again count the numbers of triangles that result from a triangulation/meshing of each of the models. To compute the number of triangles of the CityJSON abstractions the shapes were triangulated using Rhino3D. The actual triangle counts which are used for this comparison can be found in the Appendix in Table 12.

This third evaluation is expanded by an evaluation of the presence of windows in the LoD3.2 abstraction. Section 3.1.3 describes how the selective application of voids can negatively impact the accuracy of window and door data. To quantify this, the number of windows and doors are counted both in the input IFC model and each LoD3.2 abstraction, see Table 10. This was counted manually for the IFC file since automated approaches implemented in BIM Authoring software proved to be unreliable. The windows in the IFC file that were taken into account are both *IfcWindow* objects and *IfcPanel* objects that are see-through.

5.3.2 Discussion of the results

File size The file size evaluation shows a clear size reduction when going from BIM (IFC) to Geo (CityJSON), see Table 7. Even the combination file, which includes nine different abstracted representations of the model, has a significant smaller file size than the input model. The average file size of the output is 4% of the original IFC input model. The LoD3.2 files are even smaller. These single file models, although still keeping a large amount of details of the original files, have an average file size that is 3% of the original IFC input model.

This file size reduction is reached by a combination of factors. The most interesting one for this research is the reduction due to shape abstraction. Simpler geometry requires less storage space. Due to the magnitude of reduction it can be concluded that the simpler abstracted shapes have a notable effect. But also other factors play a role in a significant file reduction: eliminated semantic data, differences in file encoding, and the way geometry is stored.

During the abstraction processes, semantic data is reduced because not all attributes are copied to the GIS file. For the exterior only the attributes of the *IfcBuilding*, *IfCDoor*, and *IfcWindow* objects are copied to the abstracted GIS model. If LoD3.2 is not opted for, only the attributes of the *IfcBuilding* object are copied. For the interior, the attributes of the *IfcRoom* and *IfcBuildingStorey* objects are copied. However, all the other attributes are ignored. Depending on the model this can be a very large data volume. Depending on how this is stored in the input IFC model, it can have a notable effect on the file size of the output CityJSON file.

The abstracted data is stored in a GIS file format, in this case CityJSON. CityJSON is a fairly lightweight way to store data in general. IFC in contrast is a rather heavy file format. The difference in the way the same data is stored in each file type can already influence the resulting file size. In some cases 1:1 converting IFC data to CityJSON will already reduce the file size.

IFC allows for the storage of implicit geometry, while CityJSON does not. Therefore, all the implicit geometry that is used in the output of the IfcEnvelopeExtractor has to be converted and stored as explicit geometry. Depending on how the input IFC model is constructed, this can have a bloating effect on the CityJSON files.

Polygon count Table 8 shows as expected a trend of increased polygon count for less abstracted shapes. This trend is valid for most models. However, there are some deviations. The PRAHA_GO_V5 and Demo_Ascoli Piceno_v2 models show a LoD0.2 abstraction that has a higher or similar triangle count compared to the LoD1.2 abstraction. This is caused by the LoD0.2 footprints of these models, see Figure 40. The LoD0.2 footprint in both cases is more complex than the roof outline. As was described in Section 4.3, the evaluated abstraction models are based on the roof outline and not the footprint. So the footprint's complex mesh does not have an effect on the LoD1.2

File name	Triangle count per-1000 (‰) LoD0.2	LoD0.3	LoD0.4	LoD1.2	LoD1.3	LoD2.2	LoD3.2
AC20-FZK-Haus	0.19	0.19	0.27	0.56	0.56	0.74	65.53
AC20-Institute-Var-2	0.47	0.42	1.27	1.04	1.23	3.49	85.35
FM_ARC_DigitalHub	0.09	1.25	1.25	1.04	2.39	2.39	9.45
PRAHA_GO_V5	0.04	0.22	0.43	0.04	0.43	0.68	3.27*
Demo_Lisbon_2025	0.17	0.72	1.01	0.22	1.02	3.23	6.54
Demo_Ascoli Piceno_v2	0.26	0.40	1.53	0.15	0.54	6.77	37.82*
Average	0.20	0.53	0.96	0.34	1.03	2.71	34.66

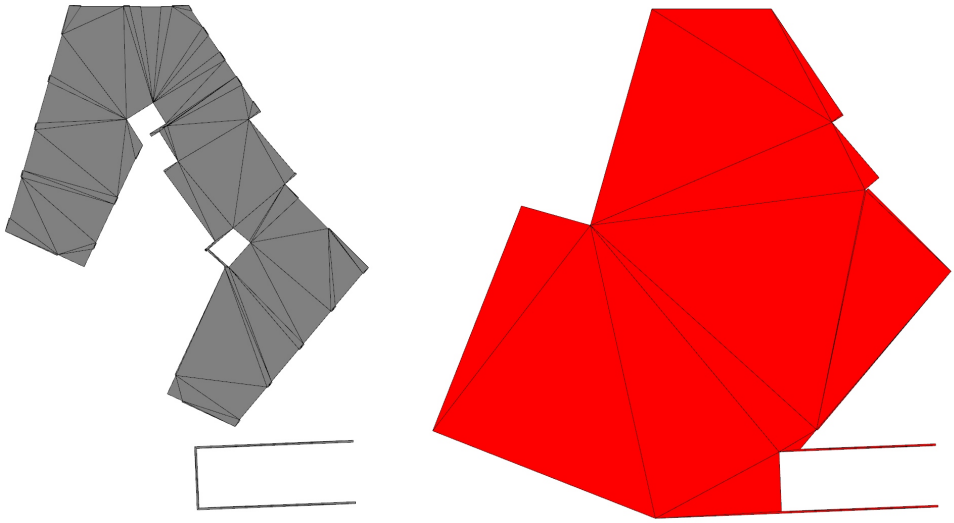
Table 8: Comparison of the number of triangles compared to the 1:1 OBJ representation of the IFC file. This is just comparing the exterior geometry. LoD0.0 and 1.0 has been excluded because they will always have a triangle count of 2 and 12 respectively. Note that this table is in per-thousand/per-mille and not percentages. *Abstractions were considered faulty (< 0.5), see Section 5.1.

mesh, while it has for LoD0.2. The other input models do not have a footprint that is significantly more complex than the roof outline, so this behaviour is not seen in those cases.

5.3.3 Complex object simplification

When looking at the results of the complex object simplification, an expected reduction of triangles can be observed when going from full complex object use to bounding box abstraction (Normal), see Table 12. In some of the models, another reduction of triangles can be observed when going from bounding box abstraction (Normal) to bounding box abstraction augmented with selective void applying (Simple). The AC20-FZK-Haus, AC20-Institute-Var-2 and FM_ARC_DigitalHub models show large complexity reduction, the Demo_Ascoli Piceno_v2 model shows some reduction, The PRAHA_GO_V5 and Demo_Lisbon_2025 models show no reduction. Looking to the window count in Table 6, this inconsistent behaviour is corroborated. The models that show a large triangle reduction when selectively applying voids also show a large reduction in the LoD3.2 window count.

The majority of this unexpected behaviour can be traced back to the implementation. IfcOpenShell is used to access the geometry of an IFC model. This C++ library is supposed to be able to access geometry without voids applied. However, its behaviour showed to be inconsistent. For the AC20-FZK-Haus, AC20-Institute-Var-2 and FM_ARC_DigitalHub models this worked flawlessly. However, for the the PRAHA_GO_V5 and Demo_Lisbon_2025 models all voids were always applied and for



(a) (b)

Figure 40: The APC model has a footprint (a) which is a lot more complex than the roof outline (b). A wireframe representation highlights this well. The difference in area of the two is caused by the underground parking lot which is not reflected in the footprint but is detected as roofing structure that should be included.

File name	Full LoD3.2 (%)	Normal LoD3.2 (%)	Simple LoD3.2 (%)
AC20-FZK-Haus	17.45*	6.55	0.61
AC20-Institute-Var-2	24.42	8.53	0.76
FM_ARC_DigitalHub	-*	0.95	0.30
PRAHA_GO_V5	4.54*	0.33*	0.33*
Demo_Lisbon_2025	23.45*	0.68	0.68
Demo_Ascoli Piceno_v2	77.92*	3.78*	3.67*

Table 9: Comparison of the effect of the complex shape simplification functions on the final triangular polygon count. The full LoD3.2 has no complex shape simplification applied, the normal LoD3.2 has only bounding box simplification applied, the simple LoD3.2 has bounding box and selective void simplification applied. This is just comparing the geometry. LoD3.2 is the outer shell only. *Abstractions that are considered faulty (< 0.5) if tested for geometric accuracy

the Demo_Ascoli Piceno_v2 voids seem to be almost randomly applied. All the voids were modelled as expected via the *IfcOpeningElement* and *IfcFeatureElementSubtraction* objects and all the models were of the same IFC version. It is not clear what causes this issue. It could be that the models contain unexpected combinations of objects that can cause IfcOpenShell to behave incorrectly.

The increasing complexity when not utilizing any complex object simplification (Full) has negative effects aside from the increase of complexity. When the complex objects are used as is it reduces the processing speed noticeably. Sometimes the slow processing prevents the model from being processed at all. This is the case for the FM_ARC_DigitalHub model. But the issues were not limited to processing speed. The LoD3.2 related issues noted in Section 5.1 were more prevalent when complex shapes were used as is. More incorrectly trimmed and missing surfaces were present in such a way that all the LoD3.2 would have scored a 0 if they were used for the geometric accuracy tests. The usability of these models is therefore considerably lower than the models that utilized the abstracted complex objects.

File name	IFC	Normal LoD3.2	Simple LoD3.2
AC20-FZK-Haus	11	11	0
AC20-Institute-Var-2	206	206	0
FM_ARC_DigitalHub	72	72	20
PRAHA_GO_V5	372 (362)	362	362
Demo_Lisbon_2025	19 (11)	11	11
Demo_Ascoli Piceno_v2	509	475	424

Table 10: Comparison of the window count in the input models and the LoD3.2 abstractions. The normal LoD3.2 has only bounding box simplification applied, the simple LoD3.2 has bounding box and selective void simplification applied.

6 Conclusions

This report describes our research to convert highly detailed BIM models to Geo models at different LoDs. A method has been developed and implemented that takes an IFC model as input and converts it to 9 different LoDs depending on the user's needs, including volume-based and surface-based LoDs of both interior and exterior. Five abstraction-specific processing methods have been developed, based on how much of the geometry of the input IFC model the specific step relies on. The abstraction steps range from low geometric dependent abstraction to very high geometric dependent abstraction and interior geometric dependent abstraction. The output LoDs are obtained by a mix of the five methods, which have been implemented into a prototype application, tested and evaluated on several BIM models.

Our study shows that the developed and implemented BIM-to-Geo conversion method achieves robust and accurate results, particularly with non-volumetric and interior abstraction methods, which consistently scored highly across various evaluations. However, the accuracy of volumetric and complex shape abstractions diminishes as the complexity of the output geometric representations increases, largely due to modelling errors in input IFC models, surface trimming inaccuracies, and tolerance-related issues.

The challenges faced in this research include the propagation of issues present in the IFC model, such as ambiguous interior-exterior delineations and modelling gaps, which can have a negative impact on abstraction outcomes. Additionally, the precision tolerances in input models often lead to small surface gaps that hinder proper Boolean operations, causing incomplete or inaccurate geometric representations, especially at higher levels of detail.

Furthermore, the results reveal that abstraction complexity impacts file size reduction and processing efficiency significantly, with simpler models achieving smaller sizes of output models and faster processing. Overly complex models without appropriate simplification may also suffer from reduced geometric accuracy.

To improve BIM-to-Geo conversions, future work should consider adaptive tolerance settings, enhanced IFC model pre-processing, and alternative methods for defining interior-exterior boundaries, especially in designs with interconnected or open spaces. Overall, while the current methodology provides a solid foundation for BIM-to-GIS integration, addressing the identified limitations will further enhance its reliability and applicability in real-world scenarios.

7 Appendix

7.1 LoD framework summary

The rules of the LoD abstractions, that were used to develop the abstraction methods, are described in this appendix. These rules define each output LoD. Although the rules are mostly based on the LoD framework of Biljecki et al. [2016], there are some differences. Some of these differences are related to the fuzziness in the existing LoD definitions that had to be given a certain interpretation for a proper implementation. Additionally, the framework of Biljecki et al. [2016] was developed when field measurements combined with 2D building polygons were the main source for 3D city models, but our research derives 3D city models from BIM models. Existing LoD definitions do not always match the BIM-derived models (e.g. more geometrical details on facades), and therefore we had to make specific choices to make them suitable. Also, due to time constraints and/or the complexity of the IFC data, certain choices were made that deviate from the framework as well. Finally, a major difference is the exclusion of semantic surface types in our implementation of the framework. The rules that we used are geometric-based only.

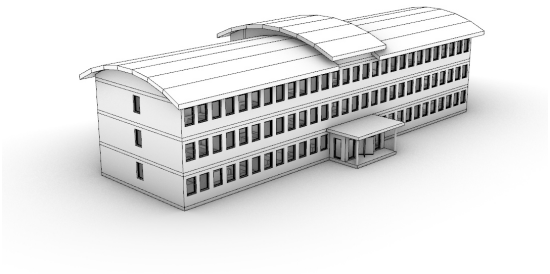


Figure 41: Input IFC model used for the following abstraction examples.

LoD0.0

2D bounding box representation of the input model, see Figure 42.

The representation consists of:

- Roof surface
  - Low geometric dependent abstraction (0).
  - Exterior representation.

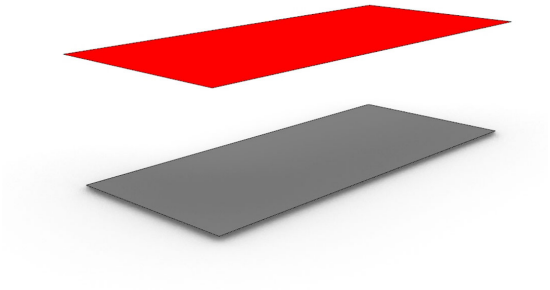


Figure 42: LoD0.0 abstraction created by the described methods. The input IFC model can be seen in Figure 41.

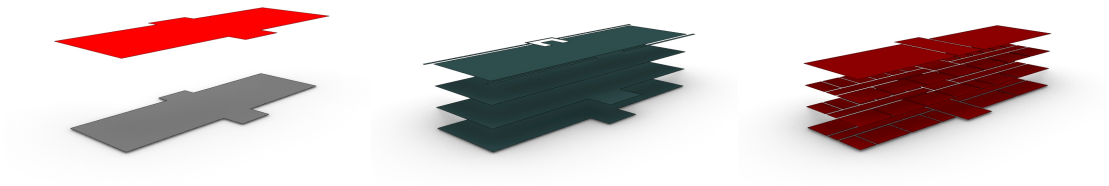
  - *RoofSurface* type or *+ProjectedRoofOutline* when no footprint extraction is toggled.
  - $n = 1$.
  - Created by selecting the top surface of the oriented smallest bounding box around the input model.

- Ground surface
  - Low geometric dependent abstraction (0).
  - Exterior representation.
  - *GroundSurface* type.
  - $n = 1$.
  - Created by selecting the top surface of the bounding box around the objects of the input IFC model located in a ± 0.15 metre buffer around the footprint elevation. The resulting surface is placed at the footprint height.

LoD0.2

Simple 2D surface representation of the input model, see Figure 43.

Every surface group is represented by simple flat surface with a normal of $(0, 0, x)$. For example, the surfaces representing the roof are all located at the same Z height and represented by a single polygon if they touch or intersect. Overhang is allowed between surfaces. So, surfaces representing the roof are allowed to overhang over the



(a) (b) (c)

Figure 43: LoD0.2 abstraction created by the described methods. a) Shows the footprint and roof structure. b) Shows the storeys. c) Shows the rooms. The input IFC model can be seen in Figure 41.

surfaces representing the footprint. Surfaces representing different storeys are allowed to overhang each other as well. The representation consists of:

- Roof surface(s)
  - Mid geometric dependent abstraction (1).
  - Exterior representation.
  - *RoofSurface* type or *+ProjectedRoofOutline* when the footprint extraction is not toggled.
  - $n \geq 1$.
  - Created by isolating the top surfaces of the roof structure and projecting these surfaces flat. These surfaces are merged to simplify the shapes. The simplified surfaces are placed at footprint height if no footprint extraction is toggled. The simplified surfaces are placed at the building's max Z if footprint extraction is toggled.
- Ground/footprint surface(s)
  - High geometric dependent abstraction (2).
  - Exterior representation.
  - *GroundSurface* type.
  - $n \geq 1$.
  - Created by taking a section through the entire IFC model at the footprint height. Close to flush horizontal surfaces that fall within a ± 0.15m buffer are added to the selection as well. These flat surfaces are merged to create the the footprint geometry. The rings that represent inner shafts and openings

are eliminated to create simple surface. Identical geometry and approach as LoD0.3 and 0.4.

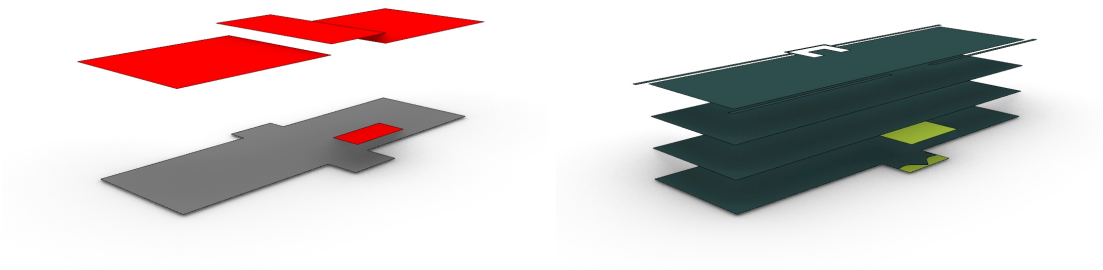
- Storey surface(s)
  - High geometric dependent abstraction (2).
  - Interior representation
  - *FloorSurface* type.
  - If IFC file includes *IfcBuildingStorey* objects $n \geq 1$, else $n = 0$.
  - Created by taking a section through the entire IFC model at the storey's height. Close to flush horizontal surfaces that fall within a ± 0.15m buffer are added to the selection as well. These flat surfaces are merged to create the storey geometry.
- Room (ceiling) surface(s)
  - Interior geometric dependent abstraction (4).
  - Interior representation
  - *+ProjectedCeilingOutline* type.
  - If IFC file includes *IfcSpace* objects $n \geq 1$, else $n = 0$.
  - Created by isolating the top surfaces of each of the *IfcSpace* objects and projecting these surfaces flat. These surfaces are merged to simplify the shapes. The simplified shape is placed at the lowest height of the *IfcSpace* object's original geometry.

LoD0.3

2D surface representation of the input model with multiple heights, see Figure 44.

Every surface that touches is grouped and flattened to have a face normal of $(0, 0, x)$. Overhang is allowed between surfaces from different sources. For example, surfaces representing the roof are not allowed to overhang over each other, but are allowed to overhang over the surfaces representing the footprint. Surfaces representing different storeys are allowed to overhang each other as well. The representation consists of:

- Roof surface(s)
  - Mid geometric dependent abstraction (1).
  - Exterior representation
  - *RoofSurface* type.
  - $n \geq 1$.



(a) (b)

Figure 44: LoD0.3 abstraction created by the described methods. a) Shows the footprint and roof structure. b) Shows the storeys. The input IFC model can be seen in Figure 41.

  - Created by isolating the top surfaces of the roof structure. The neighbouring surfaces are grouped together. These groups are flattened and placed at the group's max Z height. Overhang over a lower flattened roof surface is eliminated by trimming the lower surface.

- Ground/footprint surface(s)
  - High geometric dependent abstraction (2).
  - Exterior representation
  - *GroundSurface* type.
  - $n \geq 1$.
  - Created by taking a section through the entire IFC model at the footprint height. Close to flush horizontal surfaces that fall within a ± 0.15m buffer are added to the selection as well. These flat surfaces are merged to create the the footprint geometry. The rings that represent inner shafts and openings are eliminated to create simple surface. Identical geometry and approach as LoD0.2 and 0.4.
- Storey surface(s)
  - High geometric dependent abstraction (2).
  - Interior representation
  - *FloorSurface* & *OuterFloorSurface* type.
  - If IFC file includes *IfcBuildingStorey* objects $n \geq 1$, else $n = 0$.

- Created by taking a section through the objects that are related to the storey at the storey's height. Close to flush horizontal surfaces that fall within a ± 0.15m buffer are added to the selection as well. The surfaces are split and it is tested which are part of the interior and exterior. The surfaces of each group are merged to create the storey geometry.

LoD0.4 (LoD added to existing LoD framework)

2.5D surface representation of the roofstructure of the input model, see Figure 45.

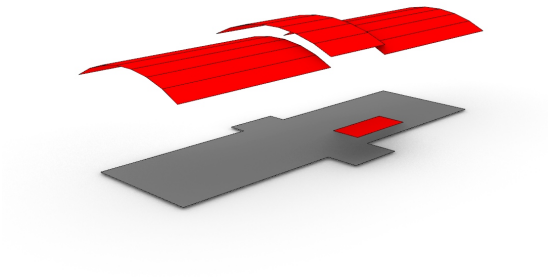


Figure 45: LoD0.4 abstraction created by the described methods. The input IFC model can be seen in Figure 41.

Overhang is allowed between surfaces from different sources. For example, surfaces representing the roof are not allowed to overhang over each other, but are allowed to overhang over the surfaces representing the footprint.

The representation consists of:

- Roof surface(s)
  - Mid geometric dependent abstraction (1).
  - Exterior representation
  - *RoofSurface* type.
  - $n \geq 1$.
  - Created by isolating the top surfaces of the roof structure. Overhang over a roof surface is eliminated by trimming the lower surface.
- Ground/footprint surface(s)
  - High geometric dependent abstraction (2).

- Exterior representation
- *GroundSurface* type.
- $n \geq 1$.
- Created by taking a section through the entire IFC model at the footprint height. Close to flush horizontal surfaces that fall within a ± 0.15m buffer are added to the selection as well. These flat surfaces are merged to create the the footprint geometry. The rings that represent inner shafts and openings are eliminated to create simple surface. Identical geometry and approach as LoD0.2 and 0.3.

LoD1.0

3D box representation of the input model, see Figure 46.

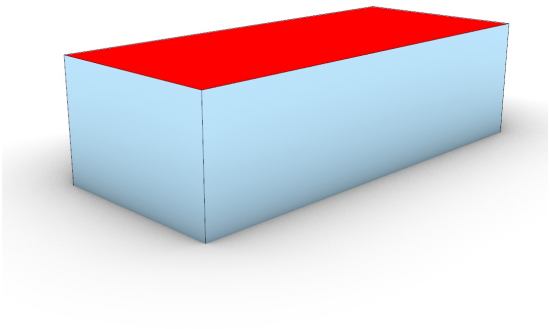


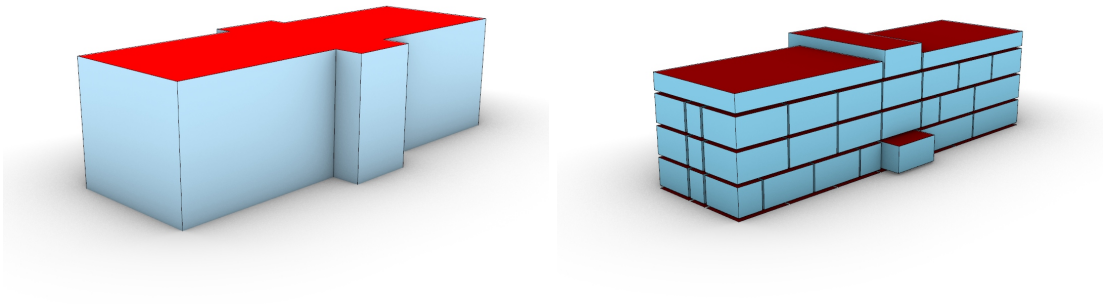
Figure 46: LoD1.0 abstraction created by the described methods. The input IFC model can be seen in Figure 41.

The representation consists of:

- Outer shell
  - Low geometric dependent abstraction (0).
  - Exterior representation.
  - *RoofSurface*, *GroundSurface*, or *WallSurface* type.
  - $n = 1$.
  - Created by computing the oriented smallest bounding box around the input model.

LoD1.2

2.5D solid representation of the input model with uniform flat top surfaces, see Figure 47.



(a) (b)

Figure 47: LoD1.2 abstraction created by the described methods. a) Shows the outer shell. b) Shows the rooms. The input IFC model can be seen in Figure 41.

Downward extrusion of the LoD0.2 top surfaces (*RoofSurface* or *CeilingSurface*) down to the footprint Z height of the model that is being processed. Or, upwards extrusion of the LoD0.2 base surfaces (only functional *GroundSurface*) for up to the top Z height of the model that is being processed. The representation consists out:

- Outer shell(s)
  - Mid geometric dependent abstraction (1).
  - Exterior representation.
  - *RoofSurface*, *GroundSurface*, or *WallSurface* type.
  - $n \geq 1$.
  - Created by extruding the LoD0.2 external roof surface(s) down to the footprint height.
- Inner shell(s)
  - Interior geometric dependent abstraction (2).
  - Interior representation.
  - *CeilingSurface*, *FloorSurface*, or *InteriorWallSurface* type.
  - If IFC file includes *IfcSpace* objects $n \geq 1$, else $n = 0$.

- Created by extruding the LoD0.2 internal ceiling surface(s) down to the room's floor height.

LoD1.3

2.5D solid representation of the input model with flat top surfaces, see Figure 48.

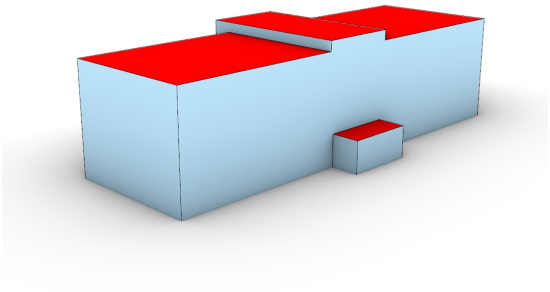
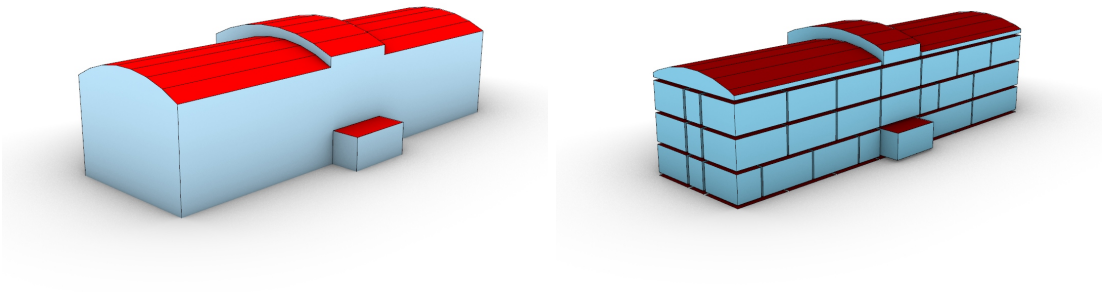


Figure 48: LoD1.3 abstraction created by the described methods. a) Shows the outer shell. b) Shows the rooms. The input IFC model can be seen in Figure 41.

Downward extrusion of the 0.3 top surfaces (*RoofSurface*) down to the footprint Z height of the model that is being processed. The top surfaces can first be trimmed down to eliminate the XY overhang over the footprint before extruding downwards. The representation consists out:

- Outer shell(s)
  - Mid geometric dependent abstraction (1).
  - Exterior representation.
  - *RoofSurface*, *GroundSurface*, or *WallSurface* type.
  - $n \geq 1$.
  - Created by extruding the LoD0.3 external roof surface(s) down to the footprint height. The resulting solids are merged.



(a) (b)

Figure 49: LoD2.2 abstraction created by the described methods. a) Shows the outer shell. b) Shows the rooms. The input IFC model can be seen in Figure 41.

LoD2.2

2.5D solid representation of the input model, see Figure 49.

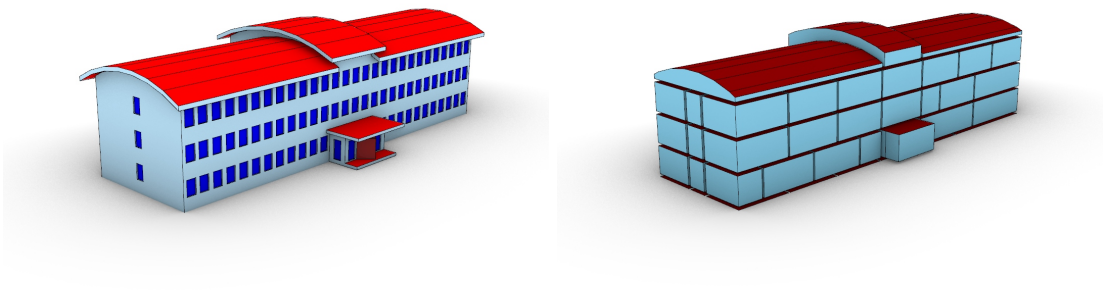
Downward extrusion of the 0.4 top surfaces (*RoofSurface*) for the exterior or the *IfcSpace* object's top surfaces down to the footprint Z height of the object that is being processed. The top surfaces can first be trimmed down to eliminate the XY overhang over the footprint before extruding downwards. The representation consists out:

- Outer shell(s)
  - Mid geometric dependent abstraction (1).
  - Exterior representation.
  - *RoofSurface*, *GroundSurface*, or *WallSurface* type.
  - $n \geq 1$.
  - Created by extruding the LoD0.4 external roof surface(s) down to the footprint height. The resulting solids are merged.
- Inner shell(s)
  - Interior geometric dependent abstraction.
  - Interior representation.
  - *CeilingSurface*, *FloorSurface*, or *InteriorWallSurface* type.
  - If IFC file includes *IfcSpace* objects $n \geq 1$, else $n = 0$.

– Created by extruding the top surfaces(s) of each *IfcSpace* object down to the room's floor height. The resulting solids per room are merged.

LoD3.2

3D full detail solid representation of the input model, see Figure 50.



(a) (b)

Figure 50: LoD3.2 abstraction created by the described methods. a) Shows the outer shell. b) Shows the rooms. The input IFC model can be seen in Figure 41.

The representation consists out:

- Outer shell(s)
  - Very high geometric dependent abstraction.
  - Exterior representation
  - *RoofSurface*, *GroundSurface*, *WallSurface*, *Window*, or *Door* type.
  - $n \geq 1$.
  - Created by isolating the (partially) outer surfaces of the input model with the help of ray casting. These surfaces are then trimmed to eliminate the parts that are not exposed to the exterior. The surfaces are typed and merged.
- Inner shell(s)
  - Interior geometric dependent abstraction.
  - Interior representation.
  - *CeilingSurface*, *FloorSurface*, or *InteriorWallSurface* type.
  - If IFC file includes *IfcSpace* objects $n \geq 1$, else $n = 0$.

– Created by 1:1 converting the geometry if the *IfcSpace* objects.

7.2 Raw result data

File name	IFC size (kB)	OBJ size (kB)	CityJSON size* (KB)	CityJSON LoD3.2 size (KB)
AC20-FZK-Haus	25 111	3 624	107	98
AC20-Institute-Var-2	10 678	4 232	931	861
FM_ARC_DigitalHub	13 746	41 747	185	107
PRAHA_GO_V5	28 786	1 079 857	2 237	2 047
Demo_Lisbon_2025	14 912	81 795	188	97
Demo_Ascoli Piceno_v2	20 975	168 957	1 012	459

Table 11: Comparison of the file size of the input file, the combined output file and the output file containing LoD3.2 only. All the files contain both interior and exterior abstractions. *This CityJSON output file contains LoD0.0, 0.2, 0.3, 0.4, 1.0, 1.2, 1.3, 2.2, and 3.2.

File name	IFC/OBJ	Full LoD3.2	Normal LoD3.2	Simple LoD3.2
AC20-FZK-Haus	21 606	3 771*	1 416	132
AC20-Institute-Var-2	42 416*	10 359	3 620	322
FM_ARC_DigitalHub	172 532*	-	1 631	520
PRAHA_GO_V5	3 502 204	155 915*	11 452	11 452
Demo_Lisbon_2025	317 134	74 374*	2 160	2 160
Demo_Ascoli Piceno_v2	617 644	481 242*	23 360*	22 682*

Table 12: Comparison of the effect of the complex shape simplification functions on the final triangular polygon count. The full LoD3.2 has no complex shape simplification applied, the normal LoD3.2 has only bounding box simplification applied, the simple LoD3.2 has bounding box and selective void simplification applied. This is just comparing the geometry. LoD3.2 is the outer shell only.

File name	IFC/OBJ	LoD0.2	LoD0.3	LoD0.4	LoD1.2	LoD1.3	LoD2.2	LoD3.2
AC20-FZK-Haus	21 606	4	4	6	12	12	16	1 416
AC20-Institute-Var-2	42 416	20	18	54	44	52	148	3 620
FM_ARC_DigitalHub	172 532	16	216	216	12	412	412	1 631
PRAHA_GO_V5	3 502 204	150	770	1 520	130	1 534	2 394	11 452
Demo_Lisbon_2025	317 134	54	228	321	70	322	1 023	2 074
Demo_Ascoli Piceno_v2	617 644	161	248	945	94	334	3 563	23 360

Table 13: Comparison of the triangular polygon count of the input and output exterior abstractions. This is just comparing the geometry. LoD0.0 and 1.0 has been excluded because they will always have a triangular polygon count of 2 and 12 respectively.

References

Bello Pérez, F. O. (2015). Generación de CityGML con base en IFC. Master's thesis, Universidad Distrital Francisco José de Caldas.

Biljecki, F., Ledoux, H., and Stoter, J. (2016). An improved LOD specification for 3D building models. *Computers, environment and urban systems*, 59:25–37.

Biljecki, F., Lim, J., Crawford, J., Moraru, D., Tauscher, H., Konde, A., Adouane, K., Lawrence, S., Janssen, P., and Stouffs, R. (2021). Extending CityGML for IFC-sourced 3D city models. *Automation in Construction*, 121.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., and Çöltekin, A. (2015). Applications of 3D city models: State of the art review. *ISPRS International Journal of Geo-information*, 4(4):2842–2889.

de Laat, R. and van Berlo, L. (2011). Integration of BIM and GIS: The development of the CityGML GeoBIM extension. In Kolbe, T. H., König, G., and Nagel, C., editors, *Advances in 3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography. Springer-Verlag Berlin Heidelberg.

Delft University of Technology (n.d.). Ifcgref github. `https://github.com/tudelft3d/ifcgref/`. Accessed: 2025-09-04.

Deng, Y., Cheng, J. C., and Anumba, C. (2016). Mapping between BIM and 3D GIS in different levels of detail using schema mediation and instance comparison. *Automation in Construction*, 67:1–21.

Donkers, S., Ledoux, H., Zhao, J., and Stoter, J. (2016). Automatic conversion of IFC datasets to geometrically and semantically correct CityGML LOD3 buildings. *Transactions in GIS*, 20(4):547–569.

El-Mekawy, M., Östman, A., and Hijazi, I. (2012). An evaluation of IFC-CityGML unidirectional conversion. *International Journal of Advanced Computer Science and Applications*, 3(5):159–171.

Fan, H., Meng, L., and Jahnke, M. (2009). Generalization of 3D buildings modelled by CityGML. In Sester, M., Bernard, L., and Paelke, V., editors, *Advances in GIScience*, pages 387–405. Springer Berlin Heidelberg.

Gröger, G., Kolbe, T. H., Czerwinski, A., and Nagel, C. (2008). OpenGIS City Geography Markup Language (CityGML) encoding standard 1.0.0. OpenGIS encoding standard, Open Geospatial Consortium.

Gröger, G., Kolbe, T. H., Nagel, C., and Häfele, K.-H. (2012). OGC City Geography Markup Language (CityGML) encoding standard 2.0.0. OpenGIS encoding standard, Open Geospatial Consortium.

Isailović, D., Nadaždi, A., Parezanović, A., Petojević, Z., and Višnjevac, N. (2025). 3D city model conversion to BIM for automatic material quantities assessment. In *2025 European Conference on Computing in Construction*.

Isikdag, U. and Zlatanova, S. (2009). Towards defining a framework for automatic generation of buildings in CityGML using Building Information Models. In Lee, J. and Zlatanova, S., editors, *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, chapter 6. Springer Berlin Heidelberg.

Ji, Y., Wang, Y., Wei, Y., Wang, J., and Yan, W. (2024). An ontology-based rule mapping approach for integrating IFC and CityGML. *Transactions in GIS*, 28(3).

Jusuf, S. K., Mousseau, B., Godfroid, G., and Hui, V. S. J. (2017). Integrated modeling of citygml and ifc for city/neighborhood integrated modeling of CityGML and IFC for city/neighborhood development for urban microclimates analysis. *Energy Procedia*, 122.

Kang, T. W. and Hong, C. H. (2018). IFC-CityGML LOD mapping automation using multiprocessing-based screen-buffer scanning including mapping rule. *KSCE Journal of Civil Engineering*, 22(2):373–383.

Kolbe, T. H., Kutzner, T., Smyth, C. S., Nagel, C., and Heazel, C. R. C. (2021). OGC City Geography Markup Language (CityGML) part 1: Conceptual model standard. OGC standard, Open Geospatial Consortium.

Lam, P.-D., Gu, B.-H., Lam, H.-K., Ok, S.-Y., and Lee, S.-H. (2024). Digital twin smart city: Integrating ifc and citygml with semantic graph for advanced 3d city model visualization. *Sensors*, 24(12).

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Labetski, A., and Vitalis, S. (2019). CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(4).

Liang, Y. and Tan, Y. (2024). A systematic literature review of IFC-to-CityGML conversion. In Li, D., Zou, P. X. W., Yuan, J., Wang, Q., and Peng, Y., editors, *Proceedings of the 28th International Symposium on Advancement of Construction Management and Real Estate*, Lecture Notes in Operations Research. Springer Singapore.

Ma, Z. and Ren, Y. (2017). Integrated application of BIM and GIS: An overview. *Procedia Engineering*, 196:1072–1079.

Noardo, F., Arroyo Ohori, K., Krijnen, T., and Stoter, J. (2021). An inspection of IFC models from practice. *Applied Sciences*, 11(5):2232.

Noardo, F., Ellul, C., Harrie, L., Devys, E., Arroyo Ohori, K., Olsson, P., and Stoter, J. (2019). EuroSDR GeoBIM project: A study in Europe on how to use the potentials of BIM and Geo data in practice. In Stouffs, R., Biljecki, F., Soon, K. H., and Khoo, V., editors, *14th 3D GeoInfo Conference*, volume XLII-4/W15 of *International Archives of*

the Photogrammetry, Remote Sensing and Spatial Information Sciences, pages 53–60. ISPRS, Singapore.

Noardo, F., Harrie, L., Arroyo Ohori, K., Biljecki, F., Ellul, C., Krijnen, T., Eriksson, H., Guler, D., Hintz, D., Jadidi, M. A., Pla, M., Sanchez, S., Soini, V.-P., Stouffs, R., Tekavec, J., and Stoter, J. (2020). Tools for BIM-GIS integration (IFC georeferencing and conversions): Results from the GeoBIM Benchmark 2019. *ISPRS International Journal of Geo-Information*, 9(9).

Salheb, N., Arroyo Ohori, K., and Stoter, J. (2020). Automatic conversion of CityGML to IFC. In Wong, K., Ellul, C., Morley, J., Home, R., and Kalantari, M., editors, *ISPRS TC IV 3rd BIM/GIS Integration Workshop and 15th 3D GeoInfo Conference 2020*, volume XLIV–4/W1–2020 of *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 127–134. ISPRS, London, United Kingdom.

Stouffs, R., Tauscher, H., and Biljecki, F. (2018). Achieving complete and near-lossless conversion from IFC to CityGML. *International Journal of Geo-information*, 7(355).

Tan, Y., Liang, Y., and Zhu, J. (2023). CityGML in the integration of BIM and the GIS: Challenges and opportunities. *Buildings*, 13(1758).

van der Vaart, J. (2022). Automatic building feature detection and reconstruction in IFC models. Master's thesis, TU Delft Architecture and the Built Environment Delft, The Netherlands.

van der Vaart, J., Stoter, J., Agugiaro, G., Arroyo Ohori, K., Hakim, A., and El Yamani, S. (2024). Enriching lower LoD 3D city models with semantic data computed by the voxelisation of BIM sources. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 10:297–308.

Wu, I. and Hsieh, S. (2007). Transformation from IFC data model to GML data model: Methodology and tool development. *Journal of the Chinese Institute of Engineers*, 30(6):1085–1090.

Zhou, X., Zhao, J., Wang, J., Su, D., Zhang, H., Guo, M., Guo, M., and Li, Z. (2019). OutDet: an algorithm for extracting the outer surfaces of building information models for integration with geographic information systems. *International Journal of Geographical Information Science*, 33(7).