A framework for automatic repair of 3D city models

P2 - The research proposal

Lisa Keurentjes
student 4557670

1st supervisor: Hugo Ledoux
2nd supervisor: Ivan Pađen

November 2, 2024

1 Introduction

The world grows more complex every day with continuously increasing social, ecological, economic, and infrastructural challenges. To tackle this complexity, we tend to make cities "smarter by using simulations from various disciplines, such as wind field or flood simulations Willenborg et al. (2016). These simulations and analysis have become essential tools for decision-making in urban planning and analytics; some use cases can be found in Figure 1. For these simulations, models of the built environment are needed. With advances in technologies to collect 3D elevation information, the way practitioners model our built environment is rapidly changing from a 2D to a 3D representation, resulting in an increasing number of municipalities building up 3D city models (Kolbe and Gröger (2003)). A 3D City model represents an urban environment with a three-dimensional geometry of common urban objects and structures, with buildings as the most prominent feature (Biljecki et al. (2015)). These models can be derived from various construction methods, from automatic construction by photogrammetry and laser scanning to manually processed 2D drawings. In section 2.1.2, construction methods will be elaborated.

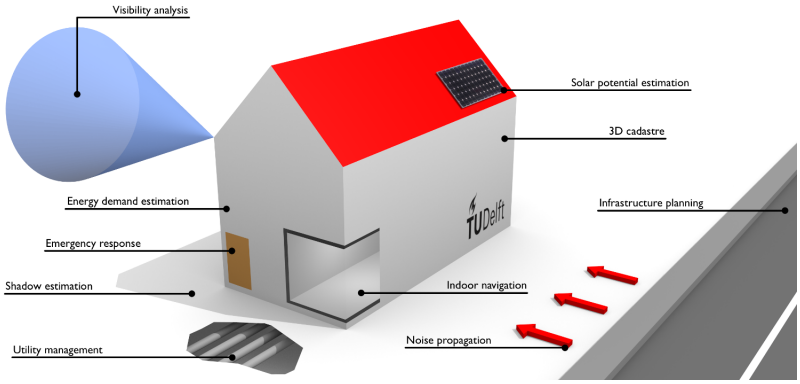


Figure 1: Some examples of use cases of 3d city models (Biljecki et al. (2015))

3D city models come in many varieties, depending on the use cases. To standardize 3D city models into classes, the Open Geospatial Consortium (OGC) attempts to classify grades of 3D data with the LOD categorization (Biljecki (2017)). The geometric detail and the semantic complexity increase with each level (Figure 2). When doing spatial analysis and/or simulation, two factors influence the accuracy of the results, namely the semantic and geometric level of detail (Biljecki (2017)) and the correctness of the data (Coors et al. (2020)). 3D city models, therefore, need to meet certain requirements before being used. To validate the quality of the data and achieve interoperability, the ISO 19100 series standards were created (Wagner et al. (2013)). However, depending on the purpose of the model, not all requirements are mandatory. For example, watertight geometry is not required if the model is used for visualization only. However, if the same model should be used for analytic purposes such as heating demand simulation, watertight geometry is mandatory (Coors et al. (2020)).

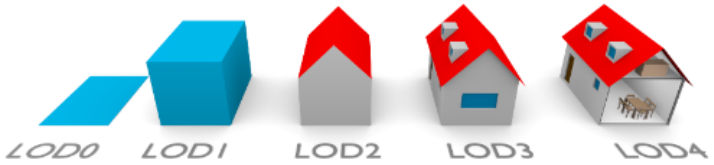
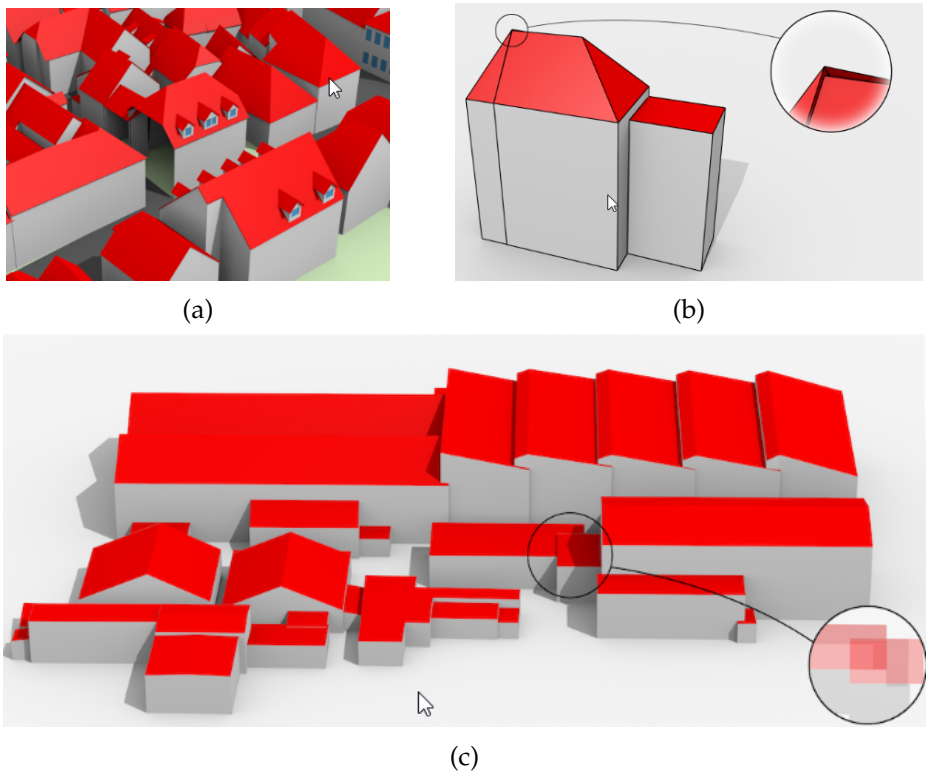


Figure 2: The five LODs of the OGC CityGML 2.0 (Biljecki (2017))

According to Biljecki et al. (2016a) a significant amount of 3D city models is not considered valid to the standards needed, they contain geometric as well as topological errors. Some examples of these errors are duplicate vertices, missing surfaces, nonwatertight solids, and intersecting volumes (Figure 3). Errors hinder the further analyzing or processing of these models, so pre-processing of the models needs to be done. Biljecki et al. (2016a) states that the only solution at this moment is to spend a substantial amount of hours manually repairing the data. Since manual repair of 3D City models is very time-consuming and prone to errors, automatic repair methods are highly desirable. There is a growing field of science that deals with automatic repair (Ledoux (2018)). Still, most automatic repair focuses on only one type of repair (for example Attene (2010) for polygons, Ledoux et al. (2014) for polygons and Mulder (2015) for solids) or leaves the model with other errors (Alam et al. (2013)).



(a) (b)

(c)

Figure 3: Some examples of 3d city model errors (Biljecki et al. (2016a)) (a) a missing surface, (b) non-watertight solid, (c) intersecting volumes

This thesis focuses on developing a framework for automatically repairing and reconstructing 3D city models. This framework aims to repair 3D city models to be valid according to the ISO standards and reconstruct towards additional requirements for various use cases. It will be supported by a software framework as proof of concept. Some example repairing methods are discussed in section 2.5 and will form the base for parts of some repairing methods in the software. To validate the 3D city models and see if the software works, val3dity (Ledoux (2018)) will be used. In section section 2.3, further details on the val3dity validator will be provided. The developed software aims to replace all the manual pre-processing steps before doing simulations or analysis with 3D city models.

This thesis proposal explains this research's context, motivation, and approach in further detail. section 2 provides an overview of the background and existing related work, containing background on 3D city models (section 2.1), their validity and as a response possible errors

(section 2.3), additional validity requirements for different use-cases (section 2.4) and some existing repair options (section 2.5). The research question and the proposed methodology for the proof of concept are discussed in section 3 and section 4. After that, tools and test data will be discussed (section 5). Lastly, an outline of the time-planning can be found (**??**).

2 Background and related work

2.1 3D Citymodels

2.1.1 Level-of-detail

The level of detail (LOD) is the most important specification of a 3D city model; seeing it indicates the model's grade and usability, seeing the concept conveys the complexity of the models and their degree of abstraction from the real world (Biljecki et al. (2015)). In Figure 2, the five base LODs are shown, but Biljecki et al. (2016b) argued that from a geometric point of view, the five LODs are insufficient and that their specification is ambiguous. Therefore, they proposed a refined set of 16 LODs focused on the grade of the exterior geometry of buildings, which provides a stricter specification and allows less modeling freedom (Figure 4).

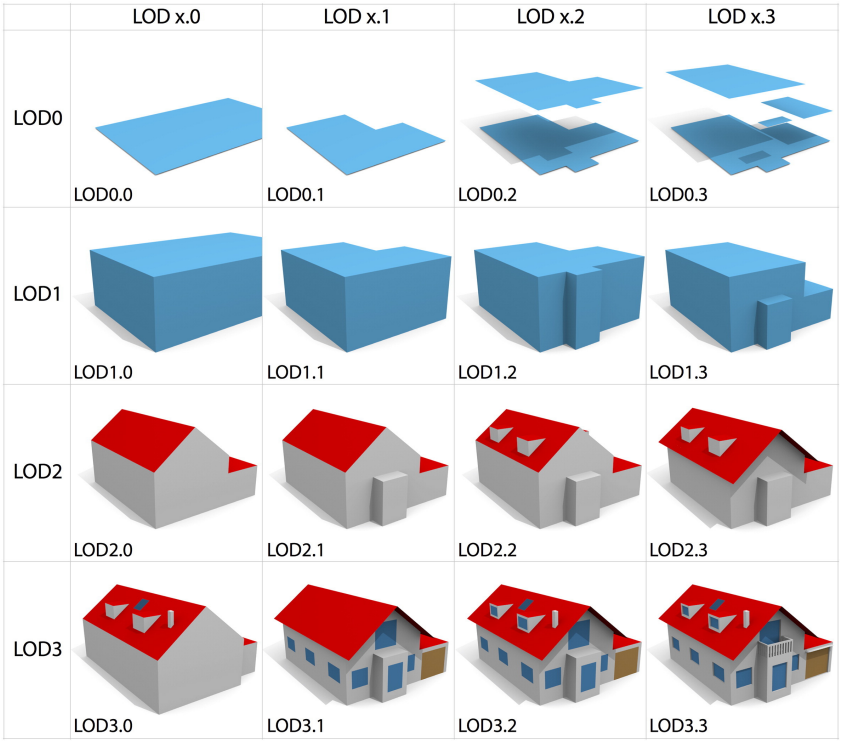


Figure 4: the refined LOD's (Biljecki et al. (2016b))

2.1.2 Construction

Different methods can be used to construct a 3d city model. To decide the right method, it needs to be decided which real-world features need to be mapped, resulting in a LOD (Biljecki et al. (2014)). Figure 5 shows an example of how different LODs have different lists of real-world features geometrically mapped (for example, if you need windows for your simulation, you need to acquire a 3d city model of LOD-C, while if you for instance only need the roof for sun-analysis you would have enough when using a 3d city model with LOD-B). The

most common methods to construct a 3d city model are GPS, lasergrammetry (using drones or airborne lidar), photogrammetry (for example, using Close range, Aerial, and Satellite images), and combinations of these three (Singh et al. (2013)). Hajji and Jarar Oulidi (2021) state that lasergrammetry is well suited to the development of building scale and large urban scale (LOD 1.0 - LOD 2.1) and that Photogrametry mostly remains a complementary method to collect more detail, for example, dormers and window locations (LOD 2.2 - LOD 3.3).

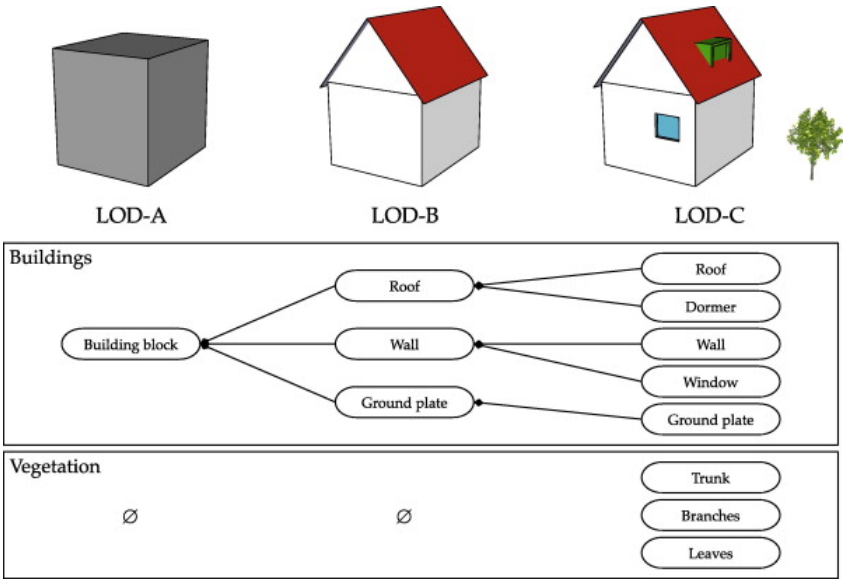


Figure 5: A simplified explanation of the concept of the presence of city objects and their elements in three different LODs of a 3D city model (Biljecki et al. (2014). Note that LOD-A is nowadays named LOD 1.0, LOD-B is named LOD2.3, and LOD-C is named LOD3.3.

2.1.3 CityJSON

Although many ways exist to store a 3d city model, for example, obj, ply, and gIFT, most can not be used easily for simulations and/or analysis. This is because they mostly focus on geometries but lack support for semantics and attributes (Ohori et al. (2022)). In 2008, the Open Geospatial Consortium (OGC) standardized CityGML as a representation and the exchange of 3D city models, resulting in being used on a worldwide scale (Gröger and Plümer (2012)). CityGML is a format based on the Extensible Markup Language (XML) and the ISO 19100 standards family and also takes the objects' semantics, their thematic properties, taxonomies, aggregations, and interrelations are taken into account (Willenborg et al. (2016)). Ledoux et al. (2019) argued that the XML-based exchange format of CityGML has several drawbacks and therefore represented CityJSON, a new JSON-based exchange format for the CityGML. CityJSON aims to be compact while not losing any information and to be friendly for web and mobile development.

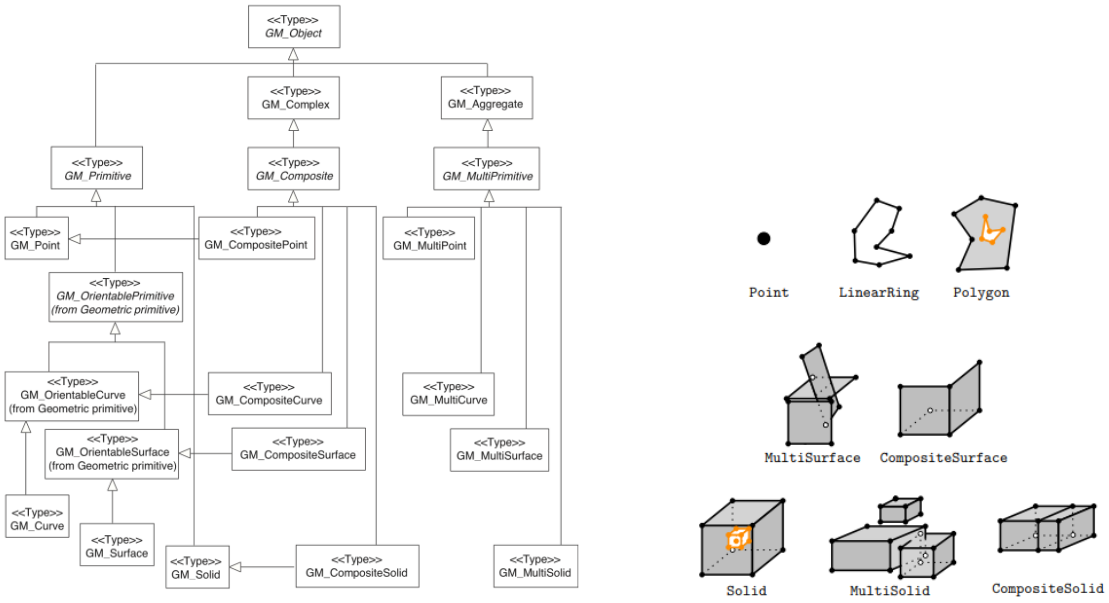
2.2 Geometry

To represent geometries, CityJSON uses the ISO 19107 geometric primitives (Ledoux (2018)). The ISO 19107 standard defines 4 primary primitives (`GM_primitives`), namely `GM_point` (0D), `GM_curve` (1D), `GM_surface` (2D) and `GM_solid` (3D), where the n-dimension is build from (n-1)-dimension primitives (FRANCOIS et al. (2010)). These four primitives can be combined into "groups" of the same dimension and saved as a composite (called `composite*`) or an

aggregate primitive (called `multi*`), which results in the ISO object hierarchy shown in Figure 6a. CityGML, and therefore CityJSON, have two restrictions, seeing it uses a subset of the ISO19107 (Ledoux (2018)). These two restrictions are:

1. `GM_curves` can only be linear, which results in `lineStrings` and `LinearRings`
2. `GM_surfaces` can only be planar, which results in `Polygons`

These restrictions result in Figure 6b being the geometries a CityJSON (and with that a 3d city model) consists of (different from the figure points are type `multipoint` and linear rings and or line strings are type `multLineString`, but can consist only one point or line).



(a) (b)

Figure 6: the geo-primitives of ISO 19107 (a) objects hierarchy (FRANCOIS et al. (2010)) (b) visual representation (Ledoux (2018))

2.3 validation of geo-primitives

Seeing that CityJSON uses ISO 19107 primitives, the primitives should compile with the definitions for this ISO standard.As a result of ISO 19107 stating that for a 3D primitive to be valid, all its lower-dimensionality primitives should also be valid, Ledoux (2013) argues that validation of a solid needs to be performed hierarchically, starting from the lowest dimensionality primitives. Ledoux (2018) extended the hierarchical validation with `multisolids` and `compositesolids` can also be validated, resulting in Figure 7.

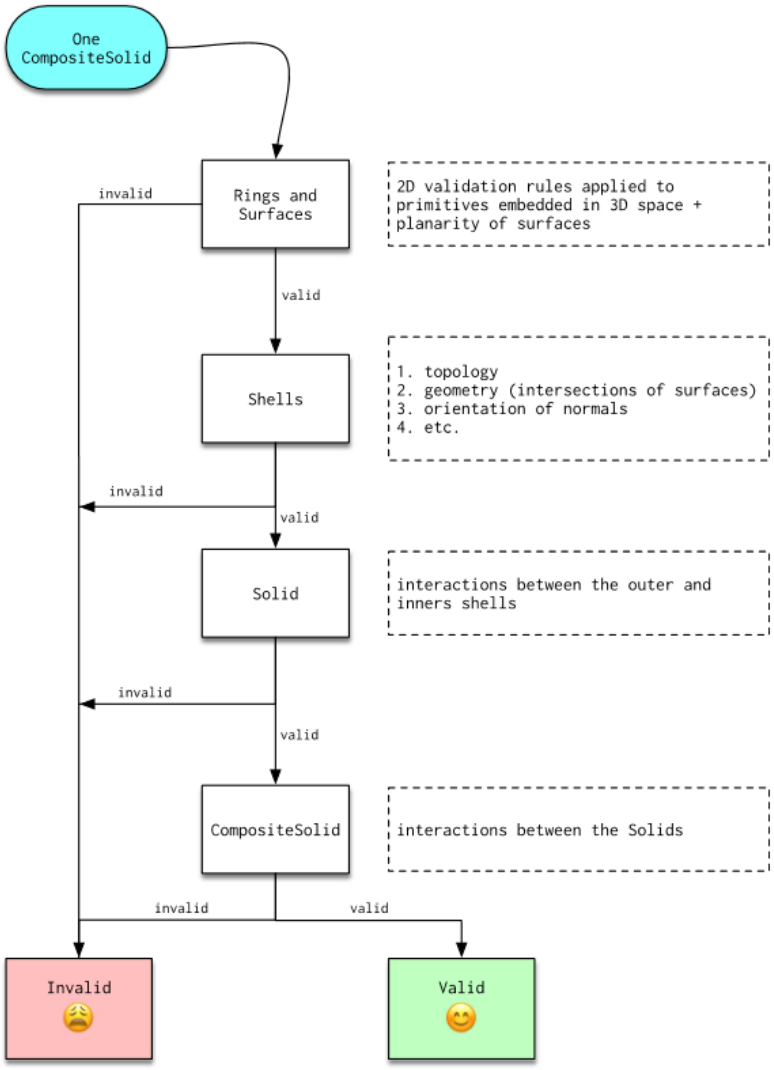
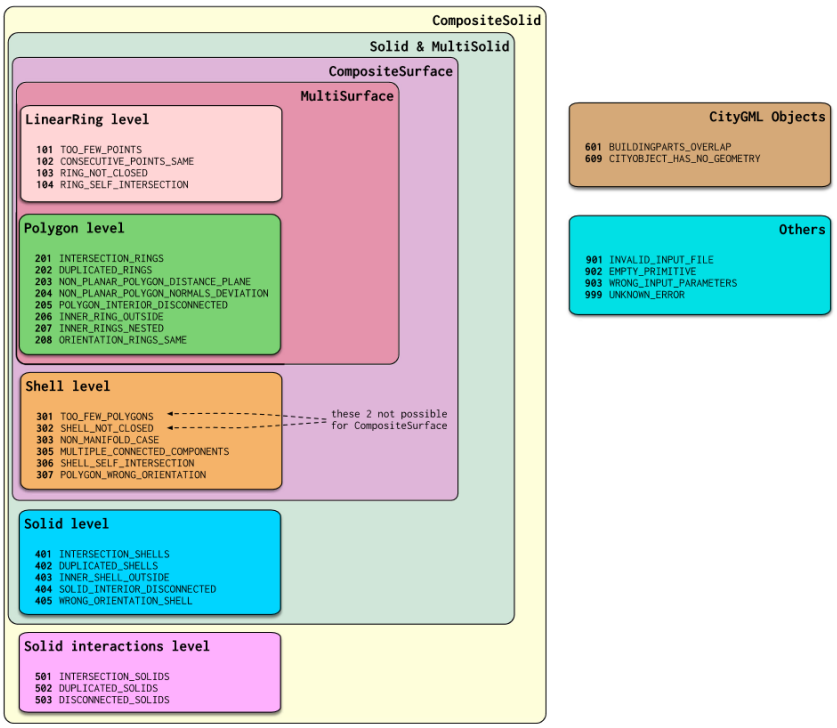


Figure 7: validation workflow (Ledoux (2018))

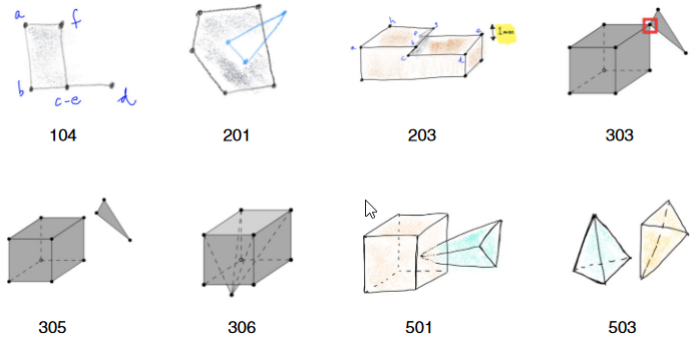
To automatically handle the validation process, Ledoux (2018) implemented a validator, called *val3dity* based on this methodology. At each level, the validator checks the requirements; if those are not met, it reports an error and an error code. Val3dity has 32 error codes (Figure 8a); 26 are primitive-based and divided over the different dimensions, and the other six are based on CityGML, input, and unknown errors. The primitive errors are based on unit tests developed for validation requirements (some examples are shown in Figure 8b). The error list is, therefore, essentially the validation requirements list.

2.4 Additional validity requirements for different use cases

Although primitive validity is a step in the right direction, it does not necessarily mean that a 3d city model is ready to be used. For example, Paden (2021) states that Computational Fluid Dynamics (CFD) have some additional requirements not covered by ISO19107. Additional requirements for the application of 3d city models are, according to Coors et al. (2020), use-case-dependent. This results in not all requirements being mandatory for all use cases and that requirements for a certain use case can contradict the requirements of other use cases. Table 1 summarizes the additional requirements for the 29 distinct use cases of 3d city mod-



(a)



(b)

Figure 8: Val3dity errors (a) listed by error code (b) example unit tests (Ledoux (2018))

els described by Biljecki et al. (2015). Some extra requirements can be achieved by using the 3D model, but other needs to be met by collecting new and/or additional data. Since this thesis focuses on automatic repair, the unrepairable ones are outside the scope and can be seen in *italic*. The additional requirements can be divided in two types of repair, namely reconstructing the geometry and adding semantics to surfaces. The use cases for which geometry reconstruction is the additional requirement are Energy demand estimation (Coors et al. (2020), Geo-visualisation (and visualization enhancement) (Coors and Zipf), and Computational fluid dynamics (Paden (2021)). The use cases for which adding semantics is the additional requirement are: Estimation of the solar irradiation (Biljecki et al. (2015)), Estimation of the propagation of noise in an urban environment (Biljecki et al. (2015), Emergency response and Routing Jebur (2022). Note that if `door` surface does not exist, those semantics cannot be added, resulting in needing to collect new and/or extra data, which is outside the scope of the thesis.

Use-case	Additional requirement(s)	Source
Estimation of the solar irradiation	building needs to have an identifiable surface(s) with type `RoofSurface`	Biljecki et al. (2015)
Energy demand estimation	buildings need to be consist of 1 `solid` (or `composite solid`)	Coors et al. (2020)
Aiding positioning	*none found*	
Determination of the floor space	*Not needed, but help-full: attribute with number of floors*	Biljecki et al. (2015)
Classifying building types	*Attribute with classification*	
Geo-visualisation and visualisation enhancement	no overlapping `polygons`	Coors et al. (2020)
Visibility analysis	*none found*	
Estimation of shadows cast by urban features	*real world position and true north needs to be correct*	Doellner et al. (2005)
Estimation of the propagation of noise in an urban environment	Not needed, but helpful: semantics for `surface(s)` of the building *and attributes naming the material of a surface*	Biljecki et al. (2015)
3D cadastre	*attributes about the physical counterparts of the legal objects*	Biljecki et al. (2015)
Visualisation for navigation	*attributes naming the material of surfaces and/or photocopies on surfaces*	Coors and Zipf
Urban planning	*An High LOD*	Köninger and Bartel (1998)
Visualisation for communication of urban information to citizenry	*An High LOD and attributes naming the material of a surface*	Biljecki et al. (2015)
Understanding SAR images	*none found*	
Facility management	*LOD4: Interior walls and or rooms*	Bleifuss et al.
Automatic scaffold assembly	*attributes containing legal and other thematic data*	Köninger and Bartel (1998)
Emergency response	*Attributes Information about building entry points* and/or identifiable surface(s) with type `door`	Biljecki et al. (2015)
Lighting simulations	*none found*	
Radio-wave propagation	*attributes naming the material of surfaces*	Kolbe and Donaubauer (2021)
Computational fluid dynamics	No small features, small edges, and small gaps between buildings	Paden (2021)
Estimating the population in an area	*attribute with number of residents*	Biljecki et al. (2015)
Routing	building needs to have identifiable surface(s) with type `door`	Jebur (2022)
Forecasting seismic damage	*geometry attribute containing geographic position, and relationship to their immediate neighborhood*	Redweik et al. (2017)
Flooding	*Cityobjects of type* `land-use`	Jebur (2022)
Change detection	*None found*	
Volumetric density studies	*correct height of buildings, so preferably at least LOD2*	Biljecki et al. (2015)
Forest management	*none found*	
Archaeology	*none found*	

Table 1: Additional requirements for different use-cases

2.5 Current repair methods

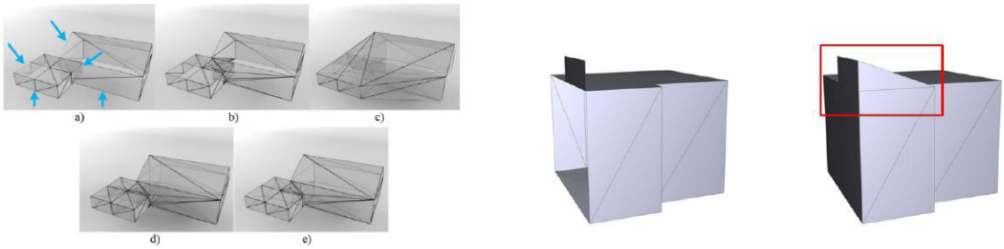
In this section, I only discuss the existing repair methods and the scientific papers for which an implementation is available (and thus ignore purely theoretical solutions). The existing repair methods for 3D city models, in publication order, are:

- Polygon mesh repair
- Local repair of CityGML Alam et al. (2013)
- Shrink wrapping method Zhao et al. (2013)
- Repairing GIS Polygons with triangulation Ledoux et al. (2014)
- Voxelization Mulder (2015) and octree extension Sindram et al. (2016)
- Triangular mesh approach Rashidan et al. (2022)

Polygon mesh repair can be used for general Computer-aided Design (CAD) model repair. Attene et al. (2013) analyzed numerous proposed Polygon mesh repair methods in terms of capabilities, properties, and guarantees, resulting in a distinction between local and global repair methods is defined. Local repair methods use local modification, which don't modifying the whole mesh. This repairs the defects, but not creating new defects can't be guaranteed. On the other hand, global repair methods make use of completely re-meshing the object, which results in guaranteed results, but loses details of the object.

According to Mulder (2015), 3D building models have different characteristics than most CAD models, resulting in needing different methods. Alam et al. (2013) and Zhao et al. (2013) described the first repair methods for repairing 3D city models. Alam et al. (2013) repair method is focused on *Local repair of CityGML*. This method iterates through all the geometries and repairs found defects, which is efficient and fast. However, the method can not solve all defects.

Zhao et al. (2013) made a global repair method called *Shrink Wrapping*. This method is based on graph theory, where all the building vertices are graph vertices. The graph applies tetrahedralization on all faces and its convex hull (Figure 9a). These method effectively repairs gaps, holes and self-intersections, however it cannot repair overshoots (Figure 9b) and is very sensitive to floating point errors.



(a) (b)

Figure 9: Shrink wrap (a) method (b) example of defect (Zhao et al. (2013))

Ledoux et al. (2014) proposed a method to automatically *repair single Gis polygons*, named *prepair*. This method is based on triangulation and uses two repair options, namely an extension of the odd-even algorithm and a point set difference rule (setdiff). Seeing *prepair* can only repair a single polygon, Ohori et al. (2012) proposed *pprepair*, which can repair a set of polygons.

In 2015, another global repair method was proposed by Mulder. This method is called *Voxelization* and is a voxel-based repair method. As shown in Figure 10 in this repair method, input is converted into a binary 3S grid. This method is very robust but has two significant drawbacks: the potential shift of the geometry and the possible loss of attributes. Also the slow processing of voxelization is a drawback, to optimize this process Sindram et al. (2016) presented an extension of this method introducing the use of an *Octree*. This approach significantly reduces computation time while preserving the same robustness as the original algorithm.

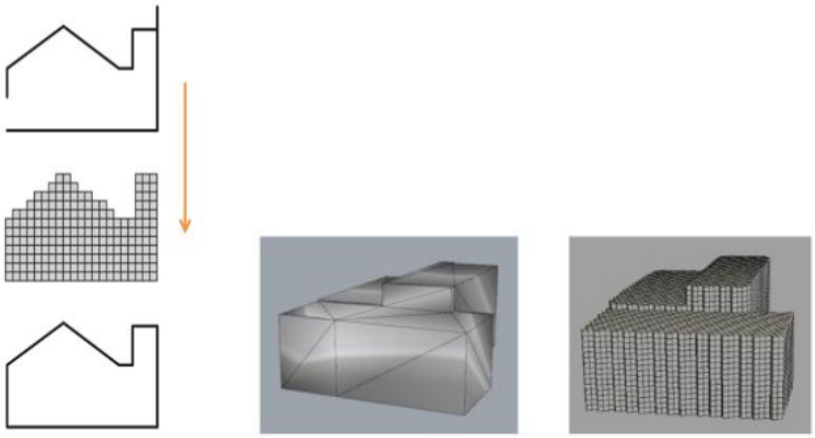


Figure 10: Voxelization (Mulder (2015))

Lastly, Rashidan et al. (2022) proposed another local repair method. *triangular mesh repair* focuses on filling holes in 3D city models and is based on the triangulation of polygons. This method results in watertight solids but only works if holes are relatively planar.

2.6 Conclusions from literature review

The main conclusions of the literature review are:

1. 3D city models come in many varieties based on their use-case and construction method. The level of detail (LOD) is the most important specification.
2. A way to store a 3D city model is by using CityJSON, a JSON replacement of CityGML. CityJSON is based on the ISO19107 geo-primitives and, therefore, follows its standards.
3. Currently, no algorithm automatically does the pre-processing of making a 3D city model validate, leaving users with many manual repairs before a 3D city model is useful.
4. If you want to validate geo-primitives on the ISO 19107 standards, it needs to be done hierarchically, starting from the lowest dimension. This results in the repair process also needing to be hierarchical.
5. Use-cases sometimes have additional requirements for a 3D city model to be valid. This can be distinguished into two requirement types: geometry reconstructing and adding semantics to surfaces.
6. Two kinds of repair algorithms exist: local and global. Local algorithms are less robust but more efficient and better for detail.

3 Research questions

This Thesis has the following main research objective:

Develop a software framework for the automatic repair and reconstruction of 3D city models to accomplish validity and facilitate different use cases.

This research aims to develop an algorithm that automatically pre-processes 3D city models for user-defined use cases. This algorithm will focus on repairing the validity errors, based on the ISO 19107 standard discussed in section 2.3, and full-filling extra requirements for different use cases as described in section 2.4. The algorithm will combine existing repairing methods (section 2.5 and complementary newly developed repair methods.

3.1 Sub-questions

To achieve the main research objective, it will be supported by the following sub-questions:

(a) How to achieve ISO19107 validity with the use of automatic repair?

(b) How to facilitate additional requirements in automatic repair based on geometry reconstruction?

(c) How to facilitate additional requirements in automatic repair based on adding semantics?

(d) What level of validity can be achieved? To what extent does this improve current 3D city models?

3.2 Scope

This thesis will focus on making a framework for the repair of 3D city models in CityJSON format. Where the files themselves are not broken, so defensive programming is not needed. This framework will focus on the following:

- Repairing cityobjects of type `Building` and type `BuildingPart` and focuses on repairing the individually.
- Repairing so that the ISO19107 standards are met
- Repairing the following additional requirements:
  - geometry reconstruction - making `buildings` one watertight `solid` (or one watertight `composite solid`), which is necessary, among other things, for energy demand reconstruction.
  - geometry reconstruction - No small features, small edges, and small gaps between buildings, which is necessary, among other things, for computational fluid dynamics.
  - Adding semantics - Define the type of added surfaces during ISO and other additional requirement repairs, which is necessary, among other things, for the estimation of solar power and noise propagation.

When having enough time or after the thesis, this framework can be extended with, for example, the extra requirements for Geo-visualisation and Emergency response routing. The repairs them-self will have the following scope:

- The repairs will not use auxiliary data; only the geometry of the `city-Object` itself will be used.
- The repairs will be as local as possible, and global repairs will only be used when no other option exists.
- The repairs will focus on preserving the semantics of the original object.
- The repairs will be deleted as little as possible.
- The repairs can change the LOD of an object, but the `LOD` member of the geometry must change into the new LOD.

4 Methodology

The workflow of the proposed methodology is provided in Figure 11. The method consists of 4 steps: Read the CityJSON, the repair loop, post-processing the data, and write the new CityJSON. When finished, the original file can be compared with the output file, and progress can be evaluated. These steps will be written into a C++ software framework; these steps will be discussed in the following subsections.

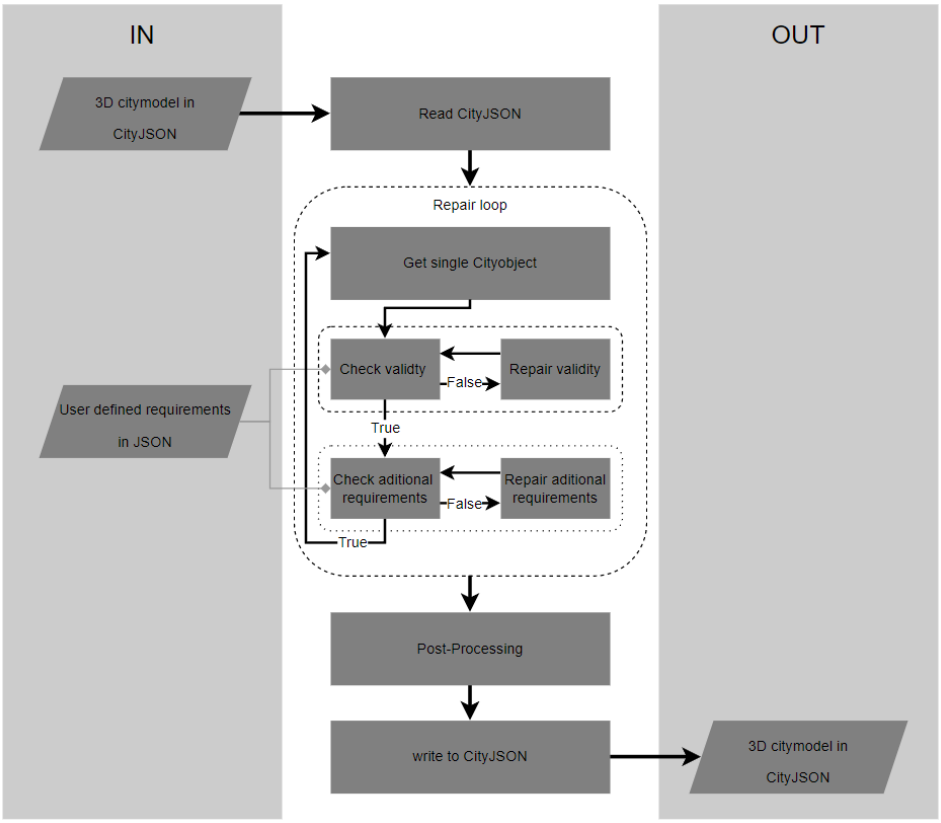


Figure 11: Flowchart of the methodology

4.1 Read the CityJSON

The first stage of this method consists of reading the CityJSON. As discussed in section 2.1.3, a CityJSON consists of at least five keys. The three needed for the repair process are: `"transform"`,

"CityObjects" and "vertices". The other keys will be stored and copied to the output data, seeing they won't change. The "Transform" will only be used to find the decompress factor for the tolerance (discussed more in section 4.2.2), but also won't be changed and copied to the output data. The "vertices" will be placed in a vector instead of the array they are in because a vector size can change dynamically. A changeable size is needed, seeing for the repair new points may need to be added or in the post-processing step vertices may be deleted. The "CityObjects" will be sent directly into the repair loop.

4.2 Repair loop

As shown in Figure 7, the repair loop consists of three sub-steps, namely getting a single CityObject, doing automatic repair based on ISO19107 validity, and doing automatic reconstruction based on use-case(s). For the repair and reconstruction extra input is needed, a user can add this input in a JSON file, otherwise default standards are used.

4.2.1 Getting a Singles objects

Seeing "CityObjects" is a dictionary; a single object can be captured by iterating over all objects. Primarily, a single cityObject consists of multiple parts, for example, Geometry and Attributes. To avoid losing any parts, the cityObject is written into a TU3DJSON. A TU3DJSON is a simple format to store/exchange 3D features Ledoux (2021). The object will be saved in the "features" with as "vertices" the whole vertices vector. The advantage of using TU3DJSON over just using the geometry is that a TU3DJSON can be read by val3dity (moreover in section 4.2.2) and it can be easily translated back into a cityObject.

4.2.2 automatic repair

Each Cityobjects validity will be evaluated based on the ISO 19107 standards (described in section 2.2). This will be done with a validator called *val3dity* implemented by Ledoux (2018). *Val3dity* can read TU3DJSON as input, which allows checking on CityObject at the time. As described in section 2.3 *val3dity* works hierarchical, resulting in that solving an error doesn't mean that there are the cityobject is valid. Therefore, after each repair, val3dity will review the cityObject again until there are no more errors. There is no plan for what to do when an error cannot be solved, which results in an object never being valid. A possible solution would be a maximum number of loops before going to the next object.

Val3dity uses the concept of tolerance to ignore small errors (Ledoux (2018)). Tolerance is used for four situations, namely:

- Planarity of polygons: Are all vertices on the same plane (default value in *val3dity* is distance-to-plane: 1cm) and/or are two polygons in the same plane (default value in *val3dity* is normals orientation: 20 degrees)
- Snapping between vertices: When are vertices "the same" and snapped together (default value in *val3dity* is (distance-to-plane): 1mm)
- Erosion by overlap of solids or BuildingParts: When two should not be overlapping, see Figure 12 (default value in *val3dity* is unused/0cm)
- Dilatation by overlap of solids or BuildingParts: When two are disjoint but should be one solid, see Figure 12 (default value in *val3dity* is unused/0cm)

In *Val3dity* tolerance values can be decided by user defined input or the default values can be used. For the automatic repair the same concept and default values will be used, resulting in a user deciding which (small) errors can be ignored. When making all the repair function maybe other choices or other tolerance decisions will occur, these will also get default values, but can be changed by user input.

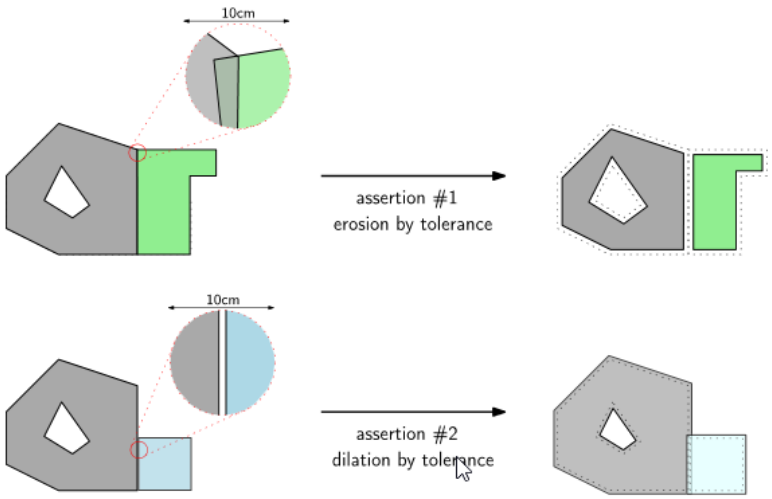


Figure 12: Example of how the tolerance is applied when validating a CompositeSolid containing 2 Solids(Ledoux (2018))

The repair methods will use inheritance because one dimension can have multiple errors (Figure 8a). According to Bieman and Zhao (1995), it is a perfect way to organize abstraction and a superb reuse tool, as it uses a base class with multiple sub-classes. The sub-classes can use the base class, for example, to read geometry and then have their own repair function. This results in, for example, ring level one base class, named ring level, and four subclasses for the four errors and their possible reparation.

Propositions on how to repair all possible val3dity dimension errors are described in the following list:

- **Ring level**
  - **101 - Too few points** - Delete the ring, no chance of repairing, if it was important polygon probably ends up showing 301 or 302 as next error (too few/shell not closed), there try capping to get the ring (or later polygon) back.
  - **102 - Consecutive points are the same** - Delete the second point which is the same. User can decide a tolerance for when points needs to be snapped together cause they are also the "same" point.
  - **103 - Ring not Closed** - *Cannot happen with CityJSON*; otherwise, repair it by putting the first point also last.
  - **104 - Ring self-intersection** - This error can happen in three cases (1) Point is used twice, (2) two points have the same coordinate, (3) edges cross each other (see Figure 13) The two main solutions, splitting surfaces into multiple surfaces and or use concave or convex hull. To decide which to choose, try both and see what the next errors are. Ledoux et al. (2014) automatic repair tool *prepair* can help with this repair.
- **Polygon level**

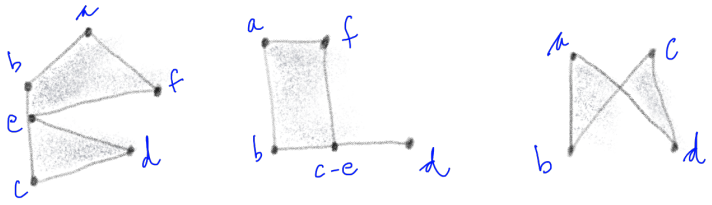


Figure 13: Ring self-intersection errors (validity docs)

- **201 - Intersection rings** - This can happen in two situations: (1) the inner ring intersects with the exterior ring, and (2) two inner rings intersect. When case one happens, make a cut out by calculating the ring intersection points and creating a contour. When case 2 happens combine the two inner rings into one by again constructing a contour.
- **202 - Duplicate rings** - Deleting the Duplicate ring. Users can decide a tolerance for when points need to be snapped together, which can result in a duplicate or not.
- **203 - Nonplanar polygon distance plane** - There needs to decide when to split the plane into multiple planes and when to use the least square to find out the best base plane. A user-set tolerance can decide this. When changing the base plane, the "wrong" points can be projected back on the plane by normal-based projection.
- **204 - Nonplaner polygon normal Deviation** - The same method as for error 203 can be used.
- **205 - Polygon interior disconnected** - Split the polygon into two (or more) polygons so they become separate polygons. User can decide a tolerance, which can results in interior being disconnected. *prepair* of Ledoux et al. (2014) can help with this repair
- **206 - Inner ring outside** - Deleting the inner ring, since keeping would introduce new geometries. *prepair* of Ledoux et al. (2014) can help with this repair
- **207 - Inner rings nested** - Deleting the nested ring,, since keeping would introduce new geometries. *prepair* of Ledoux et al. (2014) can help with this repair
- **208 - Orientation rings same** - Changing the orientation so the exterior is counterclockwise and the interior(s) clockwise. To find out the outside ring ray tracing in combination with the Möller–Trumbore algorithm could be used (Möller and Trumbore (1997)).

- **Shell level**
  - **300 - Not valid 2-manifold** - This error happens when the exact error is unknown. Seeing we don't know the problem, local repair methods are not possible, therefore a global method should be used, for example
  - **301 - Too few polygons** - If a solid has less than four polygons, then it is not possible to repair; the only exception is a triangular pyramid with one missing triangle Alam et al. (2013). We want to delete as little as possible, so this solid could be replaced with a load one bounding box, bounding the existing surfaces. A user-defined tolerance for a minimal volume of this bounding box will help decide if the shell isn't too incomplete to repair.
  - **302 - Shell not closed** - The proposed algorithm of Rashidan et al. (2022) (section 2.5) could be used for filling gaps or capping solids. When the gaps are not (almost) planar, local repair methods are insufficient, and a global method should be used.

  - **303 - Non manifold case** - First check orientation as described by error 307, if error remains find the the incident polygons and split them into a new geometry. Based on user-defined tolerance, the new geometries can be kept or deleted.
  - **305 - Multiple connected components** - Find the disconnected polygon and split it into a new geometry. Based on user-defined tolerance, the new geometries can be kept or deleted.
  - **306 - Shell self-intersection** - Using a global repair method for this error is most robust
  - **307 - Polygon wrong orientation** - To find out the outside use, ray tracing in combination with Möller–Trumbore algorithm could be used (Möller and Trumbore (1997)). Then, check if the outside is counterclockwise.

- **Solid level**
  - **401 - intersection shells** - This can happen in two situations: (1) the inner shell intersects with the exterior shell, (2) two inner shells intersect. When case one happens, make a cut out by calculating the intersection points and creating a contour. When case 2 happens combine the two inner shells into one by again constructing a contour.
  - **402 - Duplicate Shells** - Deleting the Duplicate shell. Users can decide a tolerance for when points need to be snapped together, which can result in a duplicate or not.
  - **403 - Inner shell Outside** - Deleting when the inner shell, since keeping would introduce new geometries
  - **404 - Solid interior disconnected** - Split the solid into two (or more) solids so they become separate solids. User can decide a tolerance, which can results in interior being disconnected.
  - **405 - Wrong orientation Shell** - To find out the outside use, ray tracing in combination with Möller–Trumbore algorithm could be used (Möller and Trumbore (1997)). Then, check if the outside is counterclockwise.

- **Solid interaction level**
  - **501 - Intersection Solids** - Find the intersection plane and use it to cut the solid(s), then trim the smallest parts.
  - **502 - Duplicate Solids** - Deleting the Duplicate solid. Users can decide a tolerance for when points need to be snapped together, which can result in a duplicate or not.
  - **503 - Disconnected Solids** - Two options based on user-defined tolerance; if the solids are close (user-defined tolerance), use generalization to combine them into 1 when far to change the type to `multiSolid`.

When local repair methods fail and cannot repair the object, 3D alpha wrapping will be used as a last resort(Alliez et al. (2022)). This *CGAL* component generates a valid triangulated surface mesh that strictly contains the input (watertight, intersection-free, and 2-manifold). The algorithm used is based on *shrink-wrapping* and lets users define two parameters, which have an impact on the level of detail of the output mesh.

4.2.3 reconstruction for additional requirements

As discussed in section 2.4, some use-cases have additional requirements for a 3D city model to be used. Therefore, a user can also add additional requirements to the repair process. The

default value for this will be doing none of the repairs, and based on the user input, reconstruction based on the use-case chosen by the user can be done. The rebuilding will work the same as the repair process: checking if the requirement is met, not repairing, and then checking again. Propositions on reconstructing the use cases discussed in the scope of this thesis (section 3.2) can be found below.

For computational fluid dynamics *geometry reconstruction* is needed, *buildings can not have small features, small edges, and also small gaps between buildings*. Paden (2021) describes that the first two can be simplified and the last with building footprint generalization.For simplification Park et al. (2020) developed a 5 step plan (Figure 14). After repairing, faces are divided into major and minor based on area, distance, and angle to adjacent faces and a user-defined threshold. After that, the minor faces can be generalized with the help of automatic mesh generation (Paden (2021)). For building footprint generalization, the half-space method can be used (Commandeur (2012)). The half-space method uses weighted lines (boundaries) to decide the best fit. In combination with user-defined thresholds for angles and differences, gaps between buildings can be discharged.

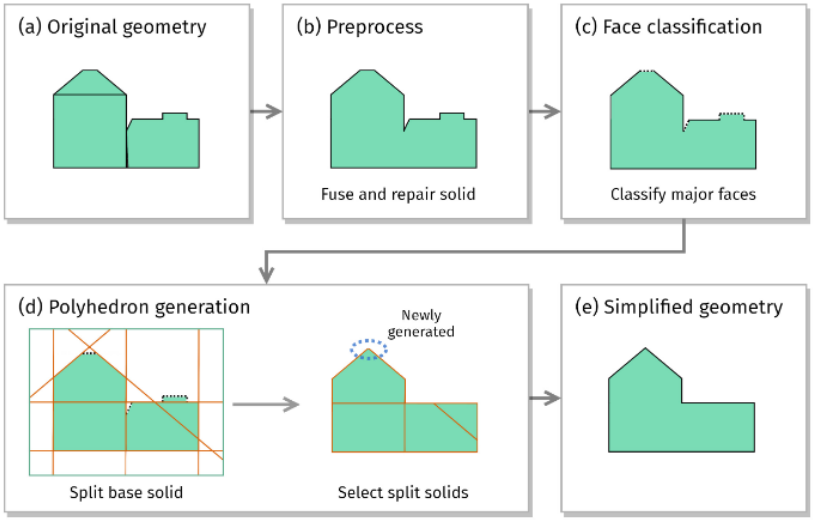


Figure 14: 5 step simplification plan(Park et al. (2020))

For energy demand reconstruction *geometry reconstruction* for non watertight buildings are needed, seeing *buildings need to be one* `solid` *(or (*`CompositeSolid`*)*. After automatic repair (section 4.2.2), existing and composite solids should be watertight. But `buildings` could also exist of `MultiSolids` or `CompositeSurfaces` or `MultiSurfaces`. `MultiSolids` could become single solids by splitting nonadjacent solids and making them into new geometries. `CompositeSurfaces` and `Multisurfaces` are not checked if they are watertight, so a possible method to check this is translating those into solids and then doing the *val3dity* solid checks again.

For estimation of solar power and estimation of propagation of noise *adding semantics's* is needed, seeing the *surfaces need to be divided into types*, namely `RoofSurface`, `WallSurface` and `GroundSurface`. When, during a repair, a surface is added, it has no semantics. The method described by Alam et al. (2014) is used to define the surface types. The technique identifies the `GroundSurface` as the surface with the smallest Z-Coordinates and the least deviation concerning the direction of the normal vector. After that, the `WallSurfaces` are identified by sharing

a standard edge with the `GroundSurface` and having a normal vector close to (-)90 degrees. To decide what is close enough, the user can define a tolerance, or a default tolerance can be used. Lastly, the `RoofSurfaces` are the surfaces remaining after identifying the other two.

4.3 post-processing

It could happen that, due to the repair and reconstruction process, there will be duplicate vertices or orphan vertices. To clean-up the vertices list before writing it in the new CityJSON a C++ replica of the python method used in CJIO will be used. This method removes duplicate vertices, checks all objects, renumbers the ones that need to be changed, and deletes all the orphan vertices, after which all the high vertices need to be enumerated again in the boundaries.

It could also happen that a CityObject is deleted for some reason, which can result in missing parent-child relationships. Also, these one-sided relationships need to be cleaned up. This will be done by a lookup method, and when none is found, the relationship will be deleted.

4.4 Evaluate the Output

To evaluate the output and measure the result, there will be two-factor counting, namely (1) the repair percentage and (2) the geometric difference from the original. *The repair percentage* will calculate how many of the 3D city models were considered valid before and how many of the 3d city models were considered valid after the repair and reconstruction. This results in a percentage of improvement. To make repairs that are not too radical *the geometric difference* between both models is also calculated. This will be done by comparing the before and after 3D city models and measuring the similarities. The more similar the better, this is to prevent drastic repairs, such as deleting. Both factors will make a good balance to evaluate the improvement.

Some errors may not be solvable, so it would be nice to notify the user why no repair is possible. A way to do this would be to output a repair report next to the new 3D city model. A user could find in this report which CityObjects are repaired and/or reconstructed, which are still invalid, and, if known, why they could not be repaired.

5 Tools and datasets used

5.1 Test data

The proof of concept will be made to repair CityJSON. The following datasets will be used as test data for the repair software:

1. **simple geometries** - The CityJSON website list some simple geometries. They are firstly used to check if the software works for singular objects. They will be manually be made to contain errors and see if the software can repair those.

2. **Cities on dataset page** - The CityJSON website also list some cities and their validity. Those will be used as the first cities to be repaired.

3. **3DBAG** - The 3D BAG is an up-to-date data set containing 3D building models of the Netherlands. The 3D BAG is open data and can be downloaded from the 3D BAG Download site. It contains 3D models at multiple levels of detail, which are generated by combining two open data sets: the building data from the BAG and the height data from the AHN.

4. **Cities/regions around the world with open datasets** - The 3d geoinformation group made a list of all the Cities/regions around the world with open datasets. The list can be found on their website. These will be used when extra datasets are needed to test for automatic repair.

5.2 Tools

The software framework will be used as a proof of concept and will be written in **C++**. The two reasons to use C++ are that C++ provides performance and memory efficiency, which results in the program being faster than, for example, python when using large data sets, and that C++ allows easy use of inheritance, which can be used in the repair classes. To make the C++ software framework, The following open-source tools will be used:

- **Nlhoman** will be used to read and write JSON files. Seeing both the input files, namely the 3d city model (in CityJSON) and the user-defined requirements, and the output, the repaired 3d city model, will be in JSON.
- **Tu3djson** will bu used to store single 3D features in the repair process. Seeing the repair loop will focus on repairing one object at a time.
- **Val3dity** (with GDAL and CGAL) will be used to validate 3D primitives according to the international standard ISO19107. Seeing it shows what needs to be repaired. Val3dity accepts Cityjson as well as tu3djson input.
- **Repair helper tools** will be used to repair 3D features with existing repair methods or help repairs. Rewriting the existing code for the repairs is a waste of time. The helper tool will consist of for example, prepair and a triangulation tool
- **Cjio** will be used for post-processing of the repaired file. Seeing it can easily remove duplicate vertices and orphan vertices. Although the code of cjio is in Python, it will form the base for a C++ version.

For visualization I will use **up3date** in **Blender**. Up3date allows importing, editing, and exporting new instances of CityJSON-encoded 3D city models in Blender. All buildings' levels of detail (LoD), attributes, and semantic surfaces are stored and accessed via Blender's graphical interface. For fast visualization, I will also use **Ninja**, the official CityJSON web viewer.

Seeing not all open data cities (**??**) are in CityJSON **CityGML-tools**, which converts CityGML to CityJSON, or **Autoconverter**, which convert most of the other 3d city file types to CityJSON, could be used to create more test data cities.

References

N. Alam, D. Wagner, M. Wewetzer, J. V. Falkenhausen, V. Coors, and M. Pries. Towards automatic validation and healing of cityGML models for geometric and semantic consistency. 2013. doi: 10.1007/978-3-319-00515-7_5.

N. Alam, D. Wagner, M. Wewetzer, J. von Falkenhausen, V. Coors, and M. Pries. Towards Automatic Validation and Healing of CityGML Models for Geometric and Semantic Consistency. In U. Isikdag, editor, *Innovations in 3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, pages 77–91. Springer International Publishing, Cham, 2014. ISBN 978-3-319-00515-7. doi: 10.1007/978-3-319-00515-7_5. URL https://doi.org/10.1007/978-3-319-00515-7_5.

P. Alliez, D. Cohen-Steiner, M. Hemmer, C. Portaneri, and M. Rouxel-Labbé. 3D Alpha Wrapping. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.5 edition, 2022. URL https://doc.cgal.org/5.5/Manual/packages.html#PkgAlphaWrap3.

M. Attene. A lightweight approach to repairing digitized polygon meshes. *The Visual Computer*, 26(11):1393–1406, Nov. 2010. ISSN 1432-2315. doi: 10.1007/s00371-010-0416-3. URL https://doi.org/10.1007/s00371-010-0416-3.

M. Attene, M. Campen, and L. Kobbelt. Polygon mesh repairing: An application perspective. *ACM Computing Surveys*, 45(2):15:1–15:33, 2013. ISSN 0360-0300. doi: 10.1145/2431211.2431214. URL https://doi.org/10.1145/2431211.2431214.

J. M. Bieman and J. X. Zhao. Reuse through inheritance: a quantitative study of C++ software. *ACM SIGSOFT Software Engineering Notes*, 20(SI):47–52, Aug. 1995. ISSN 0163-5948. doi: 10.1145/223427.211794. URL https://doi.org/10.1145/223427.211794.

F. Biljecki. Level of detail in 3D city models. 2017. doi: 10.4233/uuid:f12931b7-5113-47ef-bfd4-688aae3be248. URL https://repository.tudelft.nl/islandora/object/uuid%3A6fe1dea8-53b3-4734-9e0c-ff01ed393d79.

F. Biljecki, H. Ledoux, J. Stoter, and J. Zhao. Formalisation of the level of detail in 3D city modelling. *Computers, Environment and Urban Systems*, 48:1–15, Nov. 2014. ISSN 0198-9715. doi: 10.1016/j.compenvurbsys.2014.05.004. URL https://www.sciencedirect.com/science/article/pii/S0198971514000519.

F. Biljecki, J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin. Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, 4(4):2842–2889, Dec. 2015. ISSN 2220-9964. doi: 10.3390/ijgi4042842. URL https://www.mdpi.com/2220-9964/4/4/2842. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

F. Biljecki, H. Ledoux, X. Du, J. Stoter, K. H. Soon, and V. H. S. Khoo. The most common geometric and semantic errors in cityGML. In *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume IV-2-W1, pages 13–22. Copernicus GmbH, Oct. 2016a. doi: 10.5194/isprs-annals-IV-2-W1-13-2016. URL https://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/IV-2-W1/13/2016/. ISSN: 2194-9042.

F. Biljecki, H. Ledoux, and J. Stoter. An improved LOD specification for 3D building models. *Computers, Environment and Urban Systems*, 59:25–37, Sept. 2016b. ISSN 0198-9715. doi: 10.1016/j.compenvurbsys.2016.04.005. URL https://www.sciencedirect.com/science/article/pii/S0198971516300436.

R. Bleifuss, A. Donaubauer, J. Liebscher, and M. Seitle. Entwicklung einer CityGML-Erweiterung für das Facility Management am Beispiel Landeshauptstadt München. page 10.

T. J. F. Commandeur. Footprint decomposition combined with point cloud segmentation for producing valid 3D models. 2012. URL https://repository.tudelft.nl/islandora/object/uuid%3Ac0c665f7-0254-42c6-895b-cb59acc079f2.

V. Coors and E. Zipf. Mona 3d– Mobile Navigation Using 3d City Models.

V. Coors, M. Betz, and E. Duminil. A Concept of Quality Management of 3D City Models Supporting Application-Specific Requirements. *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88(1):3–14, Feb. 2020. ISSN 2512-2819. doi: 10.1007/s41064-020-00094-0. URL https://doi.org/10.1007/s41064-020-00094-0.

J. Doellner, H. Buchholz, M. Nienhaus, and F. Kirsch. Illustrative visualization of 3D city models. In *Visualization and Data Analysis 2005*, volume 5669, pages 42–51. SPIE, Mar. 2005. doi: 10.1117/12.587118. URL https://www.spiedigitallibrary.org/conference-proceedings-of-spie/5669/0000/Illustrative-visualization-of-3D-city-models/10.1117/12.587118.full.

A. FRANCOIS, R. Raffin, and M. Daniel. Geometric Data Structures and Analysis in GIS: ISO 19107 Case study. Nov. 2010.

G. Gröger and L. Plümer. CityGML – Interoperable semantic 3D city models. *ISPRS Journal of Photogrammetry and Remote Sensing*, 71:12–33, July 2012. ISSN 0924-2716. doi: 10.1016/j.isprsjprs.2012.04.004. URL https://www.sciencedirect.com/science/article/pii/S0924271612000779.

R. Hajji and H. Jarar Oulidi. Development of the BIM Model. In *Building Information Modeling for a Smart and Sustainable Urban Space*, pages 41–62. John Wiley & Sons, Ltd, 2021. ISBN 978-1-119-88547-4. doi: 10.1002/9781119885474.ch3. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119885474.ch3. Section: 3 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119885474.ch3.

A. K. Jebur. Application of 3D City Model and Method of Create of 3D Model- A Review Paper. *Saudi Journal of Civil Engineering*, 6(4):95–107, Apr. 2022. ISSN 25232657, 25232231. doi: 10.36348/sjce.2022.v06i04.005. URL https://saudijournals.com/media/articles/SJCE_64_95-107.pdf.

T. H. Kolbe and A. Donaubauer. Semantic 3D City Modeling and BIM. In W. Shi, M. F. Goodchild, M. Batty, M.-P. Kwan, and A. Zhang, editors, *Urban Informatics*, The Urban Book Series, pages 609–636. Springer, Singapore, 2021. ISBN 9789811589836. doi: 10.1007/978-981-15-8983-6_34. URL https://doi.org/10.1007/978-981-15-8983-6_34.

T. H. Kolbe and G. Gröger. Towards unified 3D city models. 2003. URL https://mediatum.ub.tum.de/1145769.

A. Köninger and S. Bartel. 3d-Gis for Urban Purposes. *GeoInformatica*, 2(1):79–103, Mar. 1998. ISSN 1573-7624. doi: 10.1023/A:1009797106866. URL https://doi.org/10.1023/A:1009797106866.

H. Ledoux. On the Validation of Solids Represented with the International Standards for Geographic Information. *Computer-Aided Civil and Infrastructure Engineering*, 28(9):693–706, 2013. ISSN 1467-8667. doi: 10.1111/mice.12043. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/mice.12043. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/mice.12043.

H. Ledoux. val3dity: validation of 3D GIS primitives according to the international standards. *Open Geospatial Data, Software and Standards*, 3(1):1, Feb. 2018. ISSN 2363-7501. doi: 10.1186/s40965-018-0043-x. URL https://doi.org/10.1186/s40965-018-0043-x.

H. Ledoux. TU3DJSON, Sept. 2021. URL https://github.com/tudelft3d/tu3djson. original-date: 2021-08-10T06:47:50Z.

H. Ledoux, K. Ohori, and M. Meijers. A triangulation-based approach to automatically repair GIS polygons. *Computers & Geosciences*, 66, May 2014. doi: 10.1016/j.cageo.2014.01.009.

H. Ledoux, K. Arroyo Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(1):4, June 2019. ISSN 2363-7501. doi: 10.1186/s40965-019-0064-0. URL https://doi.org/10.1186/s40965-019-0064-0.

D. T. Mulder. Automatic repair of geometrically invalid 3D City Building models using a voxel-based repair method. 2015. URL https://repository.tudelft.nl/islandora/object/uuid%3A8ef4459d-b940-4007-bc3c-d87349015129.

T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, Jan. 1997. ISSN 1086-7651. doi: 10.1080/10867651.1997.10487468. URL https://doi.org/10.1080/10867651.1997.10487468. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10867651.1997.10487468.

K. Ohori, H. Ledoux, and R. peters. *3D moddeling of the build enviroment*. Feb. 2022. URL https://github.com/tudelft3d/3dbook/releases.

K. A. Ohori, H. Ledoux, and M. Meijers. Validation and Automatic Repair of Planar Partitions Using a Constrained Triangulation. *Photogrammetrie - Fernerkundung - Geoinformation*, pages 613–630, Oct. 2012. ISSN ,. doi: 10.1127/1432-8364/2012/0143. URL https://www.schweizerbart.de/papers/pfg/detail/2012/78561/Validation_and_Automatic_Repair_of_Planar_Partitio?af=crossref. Publisher: Schweizerbart'sche Verlagsbuchhandlung.

I. Paden. Automatic reconstruction of 3D city models tailored to urban flow simulations. page 59, June 2021.

G. Park, C. Kim, M. Lee, and C. Choi. Building Geometry Simplification for Improving Mesh Quality of Numerical Analysis Model. *Applied Sciences*, 10(16):5425, Jan. 2020. ISSN 2076-3417. doi: 10.3390/app10165425. URL https://www.mdpi.com/2076-3417/10/16/5425. Number: 16 Publisher: Multidisciplinary Digital Publishing Institute.

H. Rashidan, A. Rahman, I. Musliman, and G. Buyuksalih. TRIANGULAR MESH APPROACH FOR AUTOMATIC REPAIR OF MISSING SURFACES OF LOD2 BUILDING MODELS. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVI-4/W3-2021:281–286, Jan. 2022. doi: 10.5194/isprs-archives-XLVI-4-W3-2021-281-2022.

P. Redweik, P. Teves-Costa, I. Vilas-Boas, and T. Santos. 3D City Models as a Visual Support Tool for the Analysis of Buildings Seismic Vulnerability: The Case of Lisbon. *International Journal of Disaster Risk Science*, 8(3):308–325, Sept. 2017. ISSN 2192-6395. doi: 10.1007/s13753-017-0141-x. URL https://doi.org/10.1007/s13753-017-0141-x.

M. Sindram, T. Machl, H. Steuer, M. Pültz, and T. Kolbe. VOLUMINATOR 2.0 – SPEEDING UP THE APPROXIMATION OF THE VOLUME OF DEFECTIVE 3D BUILDING MODELS. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, III-2:29–36, June 2016. doi: 10.5194/isprs-annals-III-2-29-2016.

S. P. Singh, K. Jain, and V. R. Mandla. Virtual 3d City Modeling: Techniques and Applications. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL2:73–91, Aug. 2013. ISSN 2194-9034 The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. doi: 10.5194/isprsarchives-XL-2-W2-73-2013. URL https://ui.adsabs.harvard.edu/abs/2013ISPAr.XL2b..73S. ADS Bibcode: 2013ISPAr.XL2b..73S.

D. Wagner, N. Alam, and V. Coors. Geometric validation of 3D city models based on standarized quality criteria. In *Urban and Regional Data Management, UDMS Annual 2013 - Proceedings of the Urban Data Management Society Symposium 2013*, pages 197–210. May 2013. ISBN 978-1-138-00063-6. doi: 10.1201/b14914-24. Journal Abbreviation: Urban and Regional Data Management, UDMS Annual 2013 - Proceedings of the Urban Data Management Society Symposium 2013.

B. Willenborg, M. Sindram, and T. Kolbe. Semantic 3D City Models Serving as Information Hub for 3D Field Based Simulations. June 2016.

Z. Zhao, H. Ledoux, and J. Stoter. Automatic repair of CityGML LOD2 buildings using shrink-wrapping. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-2/W1:309–317, Sept. 2013. doi: 10.5194/isprsannals-II-2-W1-309-2013.