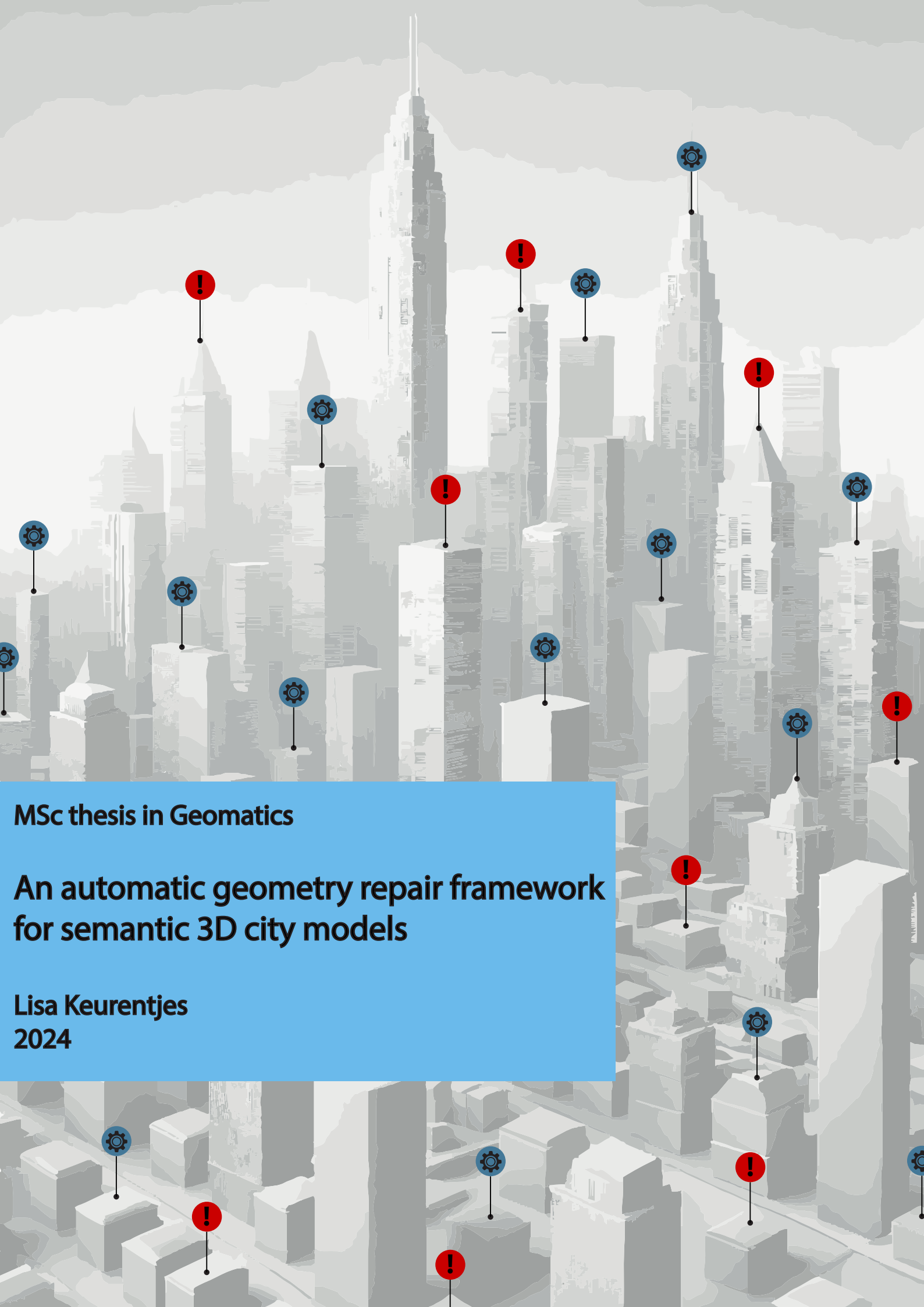


MSc thesis in Geomatics

An automatic geometry repair framework for semantic 3D city models

Lisa Keurentjes
2024

MSc thesis in Geomatics

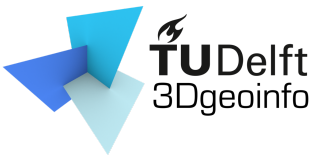
An automatic geometry repair framework for semantic 3D city models

Lisa Keurentjes

November 2024

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:



3D geoinformation group
Delft University of Technology

Supervisors:	dr. Hugo Ledoux
	Ivan Pađen
Co-reader:	dr. Martijn Meijers

Abstract

The growing complexity of urban environments has intensified the need for accurate 3D city models to support simulations and analyses in various fields such as urban planning, energy demand, and computational fluid dynamics (CFD). However, these models often contain geometric and topological errors, such as non-watertight solids, intersecting volumes, and missing surfaces, limiting their usability. This thesis presents **AUTOr3pair**, an automatic framework designed to repair semantic 3D city models and address these issues.

The process begins with validating 3D city models using val3dity, which ensures geometric validity based on ISO 19107 standards. Errors are addressed hierarchically, starting with ring-level fixes and progressing to solid interaction-level corrections, focusing on localized repairs to minimize alterations while preserving geometric and topological integrity. Existing repair methods are integrated alongside new algorithms designed to meet the specific needs of different use cases, including visualization, energy demand estimation, solar potential analysis, and CFD simulations. Written in C++ for optimal performance, the framework supports CityJSON and OBJ. The repairs prioritize maintaining semantic consistency and minimizing data loss, but textures are excluded. This approach results in a robust validation and repair pipeline that generates detailed error reports and post-processing outputs, significantly improving the overall quality of 3D city models.

Extensive tests on real-world datasets, including 3DBAG tiles from Leiden and a dataset from Brussels, demonstrated that AUTOr3pair successfully repaired most geometric errors, raising validity rates to nearly 100%. The framework achieved additional requirements, such as watertight geometry and proper surface orientation, for the specific use cases, CFD, energy demand, visualization, and solar power estimation. While some floating-point errors and geometric differences, due to global repairs, persist in complex cases, AUTOr3pair significantly reduces manual pre-processing and improves model suitability for various applications.

This thesis demonstrates that automatic geometry repair is feasible and essential for improving the quality and usability of 3D city models. It provides a foundation for further research and development, particularly in extending the framework to support more file types and refining its capabilities for additional use cases.

Acknowledgements

Within this section, I would like to express my gratitude to several individuals who played an important role during this thesis research. First and foremost, I would like to sincerely thank Dr. Hugo Ledoux for his guidance and ongoing support throughout the development of this Master's thesis. His expertise and constructive feedback played a crucial role in shaping this work, and his teaching has broadened my understanding of 3D geoinformation. Also, his understanding of my rowing career helped spread the workload over more years, resulting in great rowing results (5th place at the World University Games in Chengdu, China) and a well-written thesis.

I would also like to thank Ivan Paden for his technical support and advice, particularly in developing and refining the CFD use case, which has been invaluable. I am also grateful to Dr. Martijn Meijers, the co-reader of this thesis, for his constructive feedback and insightful comments, which helped enhance the quality and clarity of this work.

Lastly, I would like to acknowledge my family, especially my parents, for their unwavering encouragement, which has been a constant source of strength throughout my academic journey. A heartfelt thank you also goes to my boyfriend for his support and understanding during this time. Finally, to my close friends and rowing teammates at Proteus-Eretes, thank you for always being there for me and bringing balance and joy to my life during this intense study period.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

CGAL	Computational Geometry Algorithms Library
GIS	geographical information system
GISs	geographical information systems
CFD	Computational fluid dynamics
LOD	Level of detail
ISO	International Standardization Organization
LiDAR	Light Detection and Ranging
RANSAC	Random sample consensus
OGC	Open Geo-spatial Consortium
SDIs	spatial data infrastructures
XML	Extensible Markup Language
GML	Geography Markup Language
UML	Unified Modeling Language
ADE	Application Domain Extension
API	Application Programming Interface
tu3djson	TU Delft 3D JSON specification
CLI	command-line interface

1 Introduction

1.1 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, for example, wind field or flood simulations (Willenborg et al., 2016). These simulations and analysis have become essential for urban planning decision-making and analytics. Some example use cases can be found in Figure 1.1. 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 is a representation of 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, ranging from automatic construction by photogrammetry and laser scanning to manually processed 2D drawings. In Section 2.1, construction methods will be elaborated.

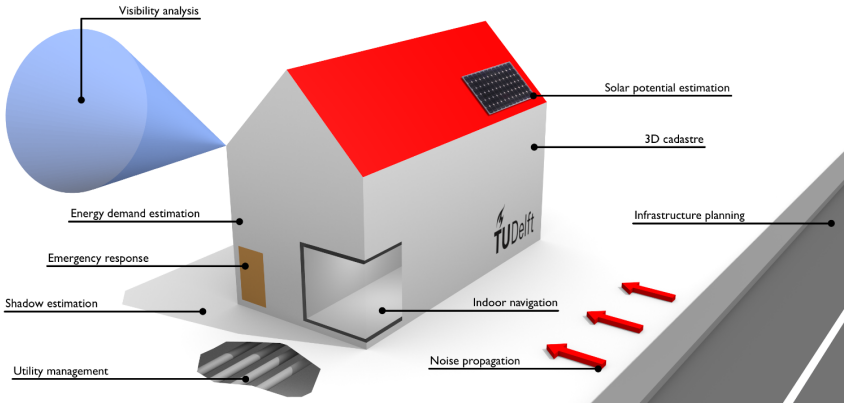


Figure 1.1.: Examples of use cases of 3D city models (taken from Biljecki et al. (2015))

1.2 Motivation and problem statement

3D city models come in many varieties, depending on the use cases. To standardize 3D city models into classes, the Open Geo-spatial Consortium (OGC) attempts to classify grades of 3D data with the Level of detail (LOD) categorization (Biljecki, 2017). The geometric detail and the semantic complexity increase with each level (Figure 1.2), which is further explained in Section 2.2.1. 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 they can be used. To validate the quality of the data and achieve interoperability, the International Standardization Organization (ISO) created the 19100 series (Wagner et al., 2013), which standardizes everything from services to the spatial schema of the data (Section 2.2.2 explains more on the standards for geometry).

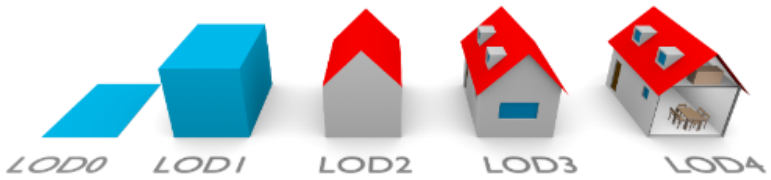
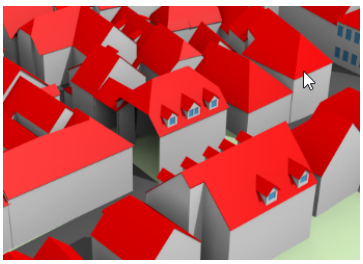
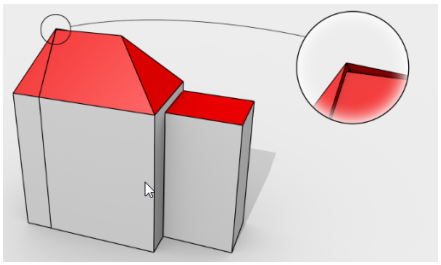


Figure 1.2.: The five Level of details of the OGC CityGML 2.0 (taken from Biljecki (2017))

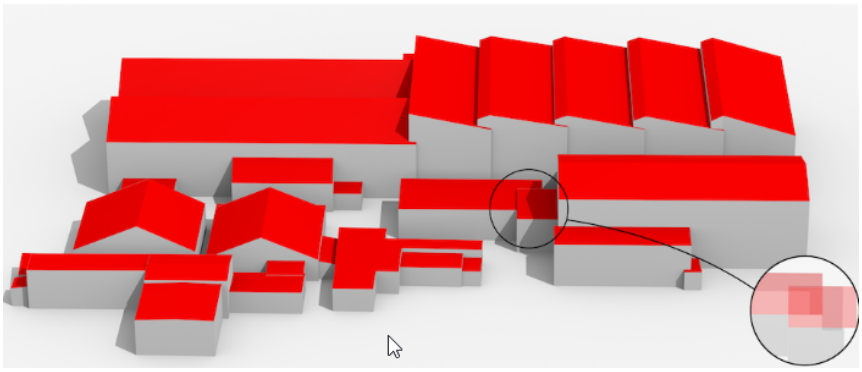
Depending on the purpose of the model, use cases can have additional requirements complementary to the ISO standards. For example, a model used for analytic purposes such as heating demand simulation and watertight geometry is mandatory (Coors et al., 2020). However, when the same model is used for visualization, this is not required. According to Biljecki et al. (2016a), many 3D city models are 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 1.3).



(a) a missing surface



(b) nonwatertight solid



(c) intersecting volumes

Figure 1.3.: Examples of 3d city model errors (taken from Biljecki et al. (2016a))

Errors hinder the further analysis or processing of these models, so pre-processing of the models is needed. Biljecki et al. (2016a) states that the only solution now is to spend a substantial amount of hours manually repairing the data. El Morr et al. (2022) even states that data scientist spend around 60% of their time on cleaning and organizing data while only spending around 7% on actually using the data (Figure 1.4). These percentages are probably higher for data where 3D geometry is present. As manual repair of 3D City models is very time-consuming and prone to errors, automatic repair methods are highly desirable. However, Semi-automatic repair tools, such as those available in MeshLab, offer a middle ground by assisting users with specific tasks like hole filling or orientation, which makes semi-automatic repair not very robust (Botsch, 2010). While these tools can help the repair process, they are often made for triangular meshes, which means your 3D geometry needs to be converted to triangular mesh and back, usually resulting in a loss of detail. Additionally, semi-automatic methods still require a significant level of manual intervention, leading to suboptimal results and potential new errors (Alam et al., 2014).

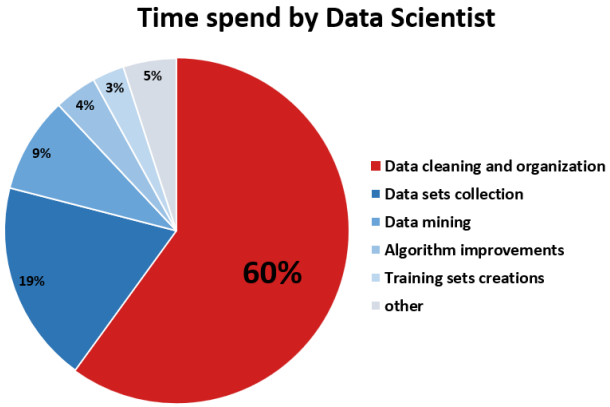


Figure 1.4.: Time spend by data scientist according to El Morr et al. (2022)

1.3 Research Objectives

A significant number of 3D city models suffer from geometric and topological errors. Semi-manual repair, the current solution, is time-consuming and error-prone, which motivates the need for automatic repair methods. There is a growing field of science that deals with automatic repair (Ledoux, 2018). However, existing approaches often focus on specific error types (for example, Hole filling or orientation) or brute-force global repair methods. Therefore, a new approach to systematically address and hierarchically navigate through all errors based on user needs is essential. Motivating the following main research objective:

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

This thesis aims to create an algorithmic repair framework for the automatic pre-processing of 3D city models tailored to user-defined use cases. This software serves as a proof of concept. It focuses on repairing validity for geometric errors according to the ISO 19107 standard

(discussed in Chapter 4) and meeting additional requirements outlined in Chapter 5. The algorithm integrates existing repair methods (Chapter 3) and introduces complementary newly developed repair techniques.

The research will investigate the following research questions:

(a) What is needed to achieve geometric validity?

(b) Is it possible to achieve geometric validity using automatic repair?

(c) How to achieve geometric validity using automatic repair?

(d) How to preserve semantics during automatic repair?

(e) How to achieve validity for different use cases?

  (I) How are repair methods to achieve geometric validity affected by different use cases?

  (II) What extra repairs are needed to achieve validity for different use cases?

(f) What degree of validity can be achieved? And to what extent does this improve current 3D city models?

1.4 The scope of this Thesis

This thesis serves as a proof of concept to demonstrate the feasibility and effectiveness of an automated approach to repairing semantic 3D city models. Substantiated by A software framework called AUTOr3pair (Figure 1.5 shows the software's logo). AUTOr3pair is open source and the code can be found on GitHub at this AUTOr3pair repository. The framework is written in C++ because performance is a critical factor, seeing that 3D city models often consist of vast amounts of data and repairs consist of numerous 3D calculations. C++ is also suitable for writing efficient processing pipelines, using static typing, inheritance and references for better memory management.

Figure 1.5.: Logo of the developed software framework, AUTOr3pair

The AUTOr3pair framework operates seamlessly for repairing CityJSON, providing universal repairs. Also, the basics of repairing OBJ are supported for better support for the use case Computational fluid dynamics (CFD). AUTOr3pair is written so it could easily be extended to repair more file types on which the repair process is done. For this thesis, the correctness of the data schema is assumed, so defensive programming is not needed. For this thesis, four use cases are chosen for which the repair process will be adjusted. These use cases are:

1. Computational fluid dynamics
2. Energy demand

3. Visualization
4. Solar power estimation

This thesis and the software framework AUTOr3pair use the following scope for the repair process:

- CityObjects are repaired when:
  - Type is `Building` or its sub-parts. *Users can repair other types, but correctness is not guaranteed*
  - Geometry type is one of the following geometric entities: MultiSurface, CompositeSurface, Solid, MultiSolid, and CompositeSolid
- Repairs focus on meeting the geometric standards from the ISO 19107 and 19125, keeping the chosen use cases in mind
- Additional geometric and semantic requirements for the chosen use cases will be repaired
- No external data is used for repairs. Only the geometry of the `CityObject` itself will be used.
- Repairs are done hierarchically and, therefore, as locally as possible; global repairs will only be used when there is no other option.
- In terms of data processing, semantic and material information of surfaces is retained, but textures are omitted from the repair process. The framework attempts to include this information when possible, leaving it out only when dealing with entirely new surfaces.
- Repairs will change and, therefore, delete as little as possible.
- Repairs can change the LOD of an object, but the `LOD` member of the geometry must change into the new LOD.

In addition to defining the scope of the repair process, the need for robustness in AUTOr3pair is crucial for ensuring its practical applicability in diverse scenarios. Given that 3D city models can vary widely in complexity and data integrity, the framework must be able to handle errors or inconsistencies without compromising the overall repair process. Robustness ensures the system can manage unexpected geometries or edge cases without reporting errors, maintaining reliability across different use cases and datasets. This robustness not only makes the framework more flexible but also enhances its ability to perform reliable and accurate repairs across various applications.

1.5 Thesis outline

To achieve the research objective(s), this thesis comprises eight chapters. Starting with the introduction (Chapter 1)), which establishes the context for the research. Chapter 2 delves into essential concepts related to the automatic repair of 3D city models. The chapter covers the background of 3D Citymodels, including level-of-detail, construction, and file types such as CityJSON and OBJ geometry format. Furthermore, it introduces the scientific research related to this graduation project, specifically validating primitives. Chapter 3 focuses on the current repair methods, which can be reused in AUTOr3pair.

The repair methodology is outlined in Chapter 4. It focuses on answering research questions (a), (b), and (c) by explaining how val3dity is used for validating the data and proposing approaches for repairing geometric errors. Alternative solutions and engineering decisions are also addressed for several methods. After that, Chapter 5, implementation of use cases, focuses on answering the research question: "How to achieve validity for different use cases?" ((e)) by answering its sub-questions ((II) and (I)) per use case.

In Chapter 6, the implementation of the findings of chapters 4 and 5 is described for the developed software framework AUTOr3pair. It details steps such as reading CityJSON, the repair loop post-processing, and generating repair reports. It also explains how preserving semantics (item (d)) is done. The research concludes with the experiment (Chapter 7), presenting findings from testing the software framework and results from the test data, answering the research question "What level of validity can be achieved?" (Question (f)). It also discusses some shortcomings of the software framework, such as repair decisions, user-friendliness, and loss of textures.

Afterward, a conclusion highlights the proof of concept, limitations, and contributions, followed by recommendations for future work (Chapter 8). These recommendations suggest avenues for expanding the tool's capabilities, exploring more file types and use cases, and conducting further research on texture handling in 3D GIS applications.

2 Background

This chapter provides an overview of the foundational concepts required for understanding the construction and characterization of 3D city models. Firstly, various methods for acquiring 3D city models are discussed. The chapter then explores the characterization of 3D city models in terms of their geometric, semantic, and appearance attributes. Lastly, validation methods for 3D city models are introduced.

2.1 Acquisition of 3D city models

Different methods can be used to construct a 3D city model. To decide the suitable method, it needs to be decided which real-world features need to be mapped, resulting in a LOD (explained in Section 2.2.1) (Biljecki et al., 2014). Figure 2.1 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 3.3, while if you for instance only need the roof for sun-analysis you would have enough when using a 3D city model with LOD 3.3).

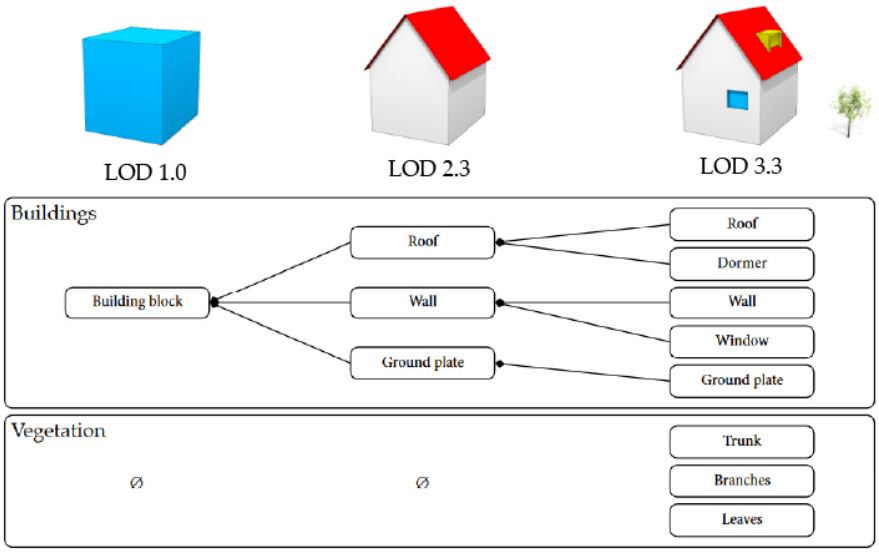


Figure 2.1.: The concept of the presence of city objects and their elements in three different LODs of a 3D city model (modified from Biljecki et al. (2014))

The most common methods for automatically constructing 3D city models are lasergrammetry and photogrammetry, or a combination (Singh et al., 2013). Figure 2.2 illustrates the critical

difference between photogrammetry and LiDAR, which are explained in the following sections.

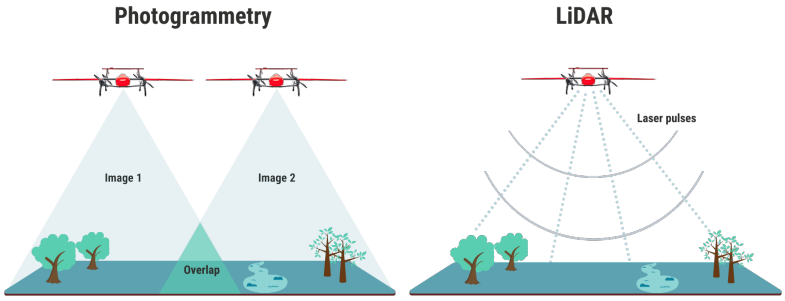


Figure 2.2.: Photogrammetry and Light Detection and Ranging (LiDAR) acquisition (taken from landsurveyors (2023))

2.1.1 LiDAR

Lasergrammetry, also known as LiDAR or Laser scanning, uses laser light to measure distances to objects. It measures the time it takes for laser pulses to bounce back from surfaces to create a detailed 3D point cloud, which is, according to Singh et al. (2013), mostly done with drones or airplanes (Figure 2.2). The biggest challenge for automatically reconstructing 3D city models from large-scale airborne LiDAR point clouds is that the vertical walls are typically missing (Huang et al., 2022). Huang et al. (2022) and Nys et al. (2020) both propose that urban buildings typically consist of planar roofs connected to the ground by vertical walls. This proposition can be used to create a pipeline to construct buildings. As shown in Figure 2.3, the first step from point cloud (a) to a 3D building (e) is plane extraction (b). This can be done with Random sample consensus (RANSAC) (Huang et al., 2022) or similar approaches (Nys et al., 2020). The generated surfaces need to be intersected by their topological relationships (c), which completes the roof of the building. With the roof known, a footprint can be generated, or existing footprint data can be used (d). The footprint and the roof are connected with the use of extrusion, which forms the walls of the building (Huang et al., 2022). 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).

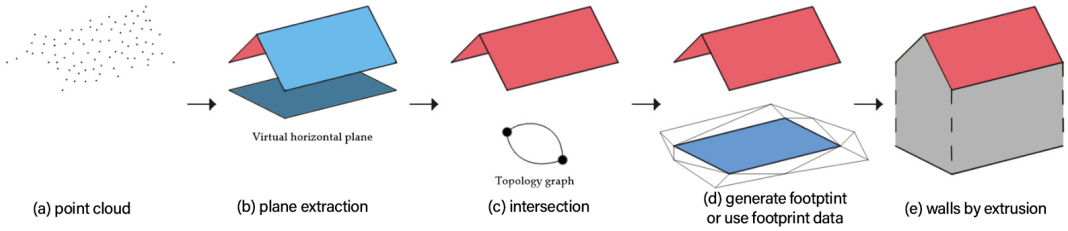


Figure 2.3.: Pipeline for modeling a building from a point cloud (modified from Nys et al. (2020))

2.1.2 Photogrammetry

Photogrammetry is a technique used to obtain precise measurements and 3D information about physical objects or environments through the analysis of photographs. It can be done with aerial images (Figure 2.2), satellite images, or with close-range imagery (Singh et al., 2013). Hajji and Jarar Oulidi (2021) state that photogrammetry mostly remains a complementary method to collect more detail, for example, to extract dormers and window locations (LOD 2.2 - LOD 3.3). According to Flamanc et al. (2003), the reason for not using photogrammetry to construct 3D city models automatically is the higher processing time, particularly for highly complex projects. However, ongoing advancements in software are continually improving processing efficiency. Figure 2.4 shows the pipe for getting a dense point cloud out with the help of photogrammetry. The first step is extracting key points (b) on overlapping images. The key points are matched (c) across multiple images, and with the camera parameters, the location of the camera (d) a sparse point cloud can be calculated (Reitinger et al., 2007). With the help of triangulation and dense matching a dense point cloud can be made from the images (e) and with surface reconstruction 3D models can be made (Javadnejad et al., 2021).

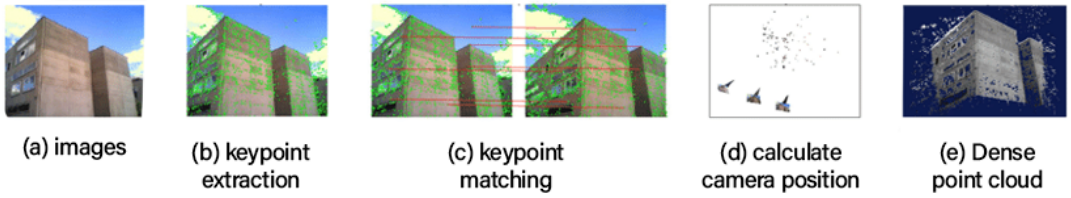


Figure 2.4.: Pipeline getting a dense point cloud with Photogrammetry (modified from Reitinger et al. (2007))

Flamanc et al. (2003) argues that lasergrammetry and photogrammetry can best be used in combination with 2D ground maps; seeing the knowledge of the ground allows a massive step in terms of quality and reliability. But to get the highest LOD 3D city models Fruh and Zakhor (2003) proposes using a combination of aerial photos and airborne laser scans, merging them with ground views. But as mentioned in Section 2.2.1, different use cases require different LODs, so combinations are not always needed.

2.2 Characterization of 3D City models

The characterization of 3D city models consists of four key dimensions: Level of detail, Geometry, Semantics, and Appearance. Metadata can also be used as an extra key to add additional data. Depending on the 3D city model, one or more of these characteristics are present. The subsections will explore their contributions and requirements for 3D city models.

2.2.1 Level of detail of 3D city models

3D city models are digital representations of urban environments that capture the spatial and geometric details of buildings, infrastructure, and terrain in three dimensions. 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 it conveys the complexity of the models and their degree of abstraction from the real-world (Biljecki et al., 2015). In Figure 1.2, 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, also known as the "TU Delft LODs", focused on the grade of the exterior geometry of buildings. This provides a stricter specification and allows less modeling freedom (Figure 2.5).

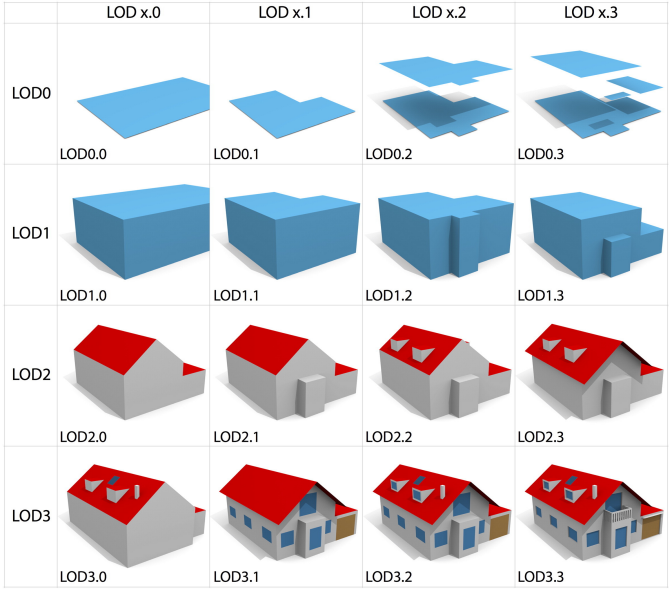


Figure 2.5.: The refined LODs (taken from Biljecki et al. (2016b))

Different use cases require different LODs. Biljecki (2017) argues that some applications (for example, visualization) can strongly benefit from higher LODs. Still, his experiments have indicated that the benefit provided by the added detail may be minuscule, which results in the cost of producing a finer LOD being much higher than the benefit it brings. Biljecki (2017) also indicates that some use cases require minimum LOD (for example, estimation of the solar potential of roofs is not advised on abstract LOD1 block models).Figure 2.6 illustrates the impact of using different LODs on the same dataset, changing the visual representation and, with that, altering the use of the data set.

Figure 2.6.: Composite rendering of a dataset in four LODs (1.2, 2.2, 2.3, and 3.3)(taken from Biljecki (2017))

2.2.2 Geometry - ISO 19107

In 2003, the OGC standardized the representation and the analysis of data in 3D space for geographical information systems (GISs) (Francois et al., 2010). It was documented in ISO 19107 and revised in 2019 (the International Organization for Standardization and Consortium, 2019). The ISO 19107 specifies conceptual standards for geometries as vectors and topology schemas. Figure 2.7 shows the 4 defined 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).

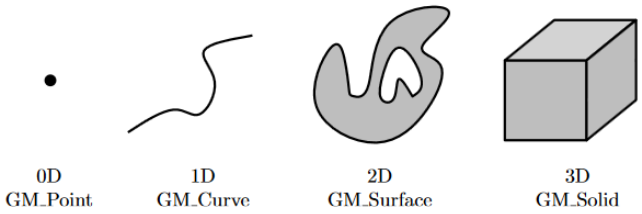


Figure 2.7.: The ISO 19107 primitives (taken from Ohori et al. (2022))

These four primitives can also be combined into sets of the same dimension. There are two types of sets a `GM_Aggregate` (called `multi*`) and a `GM_Complex` (called `composite*`). The difference between both sets is that a `GM_Complex` is a set of primitives that share a common coordinate system and parts of their geometries, while `GM_Aggregate` has no connection between them. This results in the ISO object hierarchy shown in Figure 2.8.

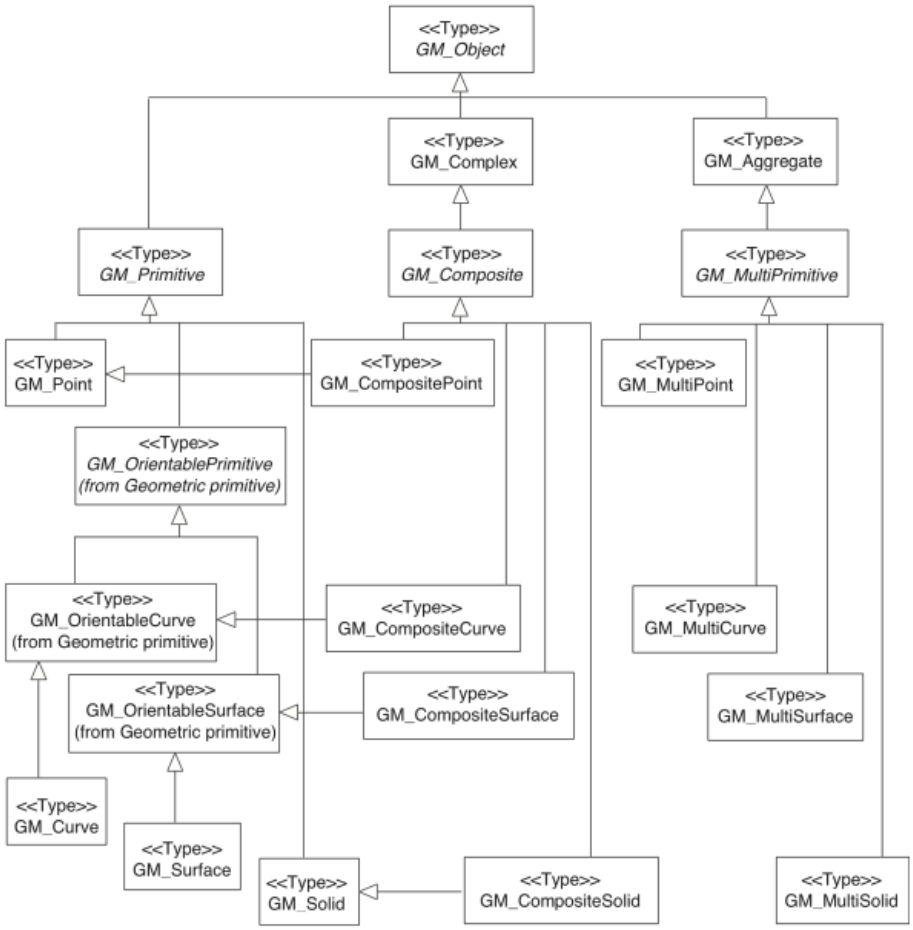


Figure 2.8.: Objects hierarchy (taken from Francois et al. (2010))

In 2004, the OGC enhanced the Standards with ISO 19125 (the International Organization for Standardization and Consortium, 2004), which standardizes additional requirements for simple Simple feature access. This standard uses the same primitives but adds Line and LinearRing as a subclass for curves and polygon as a subclass for the surfaces. Figure 2.9 visually represents the primitives and their sub-classes, including aggregates and complexes. Each primitive is a class with its requirements. Note that for a primitive to be valid, all its lower-dimensional primitives need to be valid. The requirements of the primitives, which are needed for the scope of this thesis, are:

- **Point** - A Point instance of the Geometry is a single location given by a direct position. Seeing it is a single location ISO 19107 states that:
  - The boundary is empty: $\forall P; P.\text{boundary.isEmpty} = \text{TRUE}$
  - The segment is zero: $\forall P; P.\text{segment} \rightarrow \text{length} = 0$

- **Surface** - A Surface is defined by one exterior boundary and 0 or more interior boundaries, which define holes in the Surface. The boundaries are made from Rings. When a surface is simple and planar, it is called a **Polygon**. For a surface to be a polygon, the ISO 19125 requires the following standards:
  - It is topologically closed
  - The boundary of a Polygon consists of a set of LinearRings
  - Boundaries do not cross: $\forall c_1, c_2 \in \text{Polygon.Boundary}(); c_1 \cap c_2 = \emptyset$
    But they may intersect at one point: $p, q \in \text{Point}; p, q \in c_1; p \cap q \in c_2$
  - No cut lines, spikes or punctures: $\forall P \in \text{Polygon}.P = \text{Closure}(\text{Interior}(\text{P}))$
  - The interior of every Polygon is a connected point set
  - The exterior of a polygon with holes is not connected
  - Exterior ring must have the opposite orientation of its interior ring(s) (counterclockwise versus clockwise)
- **MultiSurface** - A MultiSurface is a 2-dimensional collection of Surfaces, which has no extra requirements. It can be called simple if all surfaces are polygons.
- **CompositeSurface** - A CompositeSurface is a 2-dimensional collection of surfaces, Besides the individual requirements per surface. The ISO 19107 also requires that:
  - All surfaces are connected: $\forall s1, s2 \in CompositeSurface; s1 \cup s2 = \text{One surface}$
  - Interior should not overlap: $\forall s1, s2 \in CompositeSurface; s1 \cap s2 = \emptyset$

  To be called simple, each edge can have a maximum of two incident surfaces. A CompositeSurface can be called a shell when:
  - It is closed if it bounds a volume $\forall c; c.\text{Volume.isEmpty} = \text{FALSE}$
  - It consists of a set of Polygons
- **Solid** - A Solid is defined by one exterior boundary and 0 or more interior boundaries, which define cavities in the Solid. The boundaries are made from surfaces and can be called a shell. The Requirements are:
  - It is topologically closed and watertight
  - The boundary (shell) of a Solid consists of a set of Polygons
  - Boundaries do not cross: $\forall poly_1, poly_2 \in \text{Solid.Boundary}(); poly_1 \cap poly_2 = \emptyset$
    But they may intersect at one point: $p, q \in \text{Point}; p, q \in poly_1; p \cap q \in poly_2$
  - No cut lines, spikes or punctures: $\forall P \in \text{Solid}; P = \text{Closure}(\text{Interior}(\text{P}))$
  - The interior of every Solid is a connected point set
  - The exterior of a Solid with holes is not connected
  - Boundary (shell) must be constructed of polygons oriented that when viewed from the exterior, the points are ordered counterclockwise (the normal must point to the exterior).
- **MultiSolid** - A MultiSolid is a 3-dimensional collection of Solids, which has no extra requirements. It can be called simple if all solids are made with polygons.

- **CompositeSolid** - A CompositeSolid is a 3-dimensional collection of solids; besides the individual requirements per solid, the ISO 19107 also requires that:
  - All solids are connected and form one solid: $\forall S1, S2 \in CompositeSolids; S1 \cup S2 =$ One solid
  - Interior should not overlap: $\forall S1, S2 \in CompositeSolids; S1 \cap S2 = \emptyset$

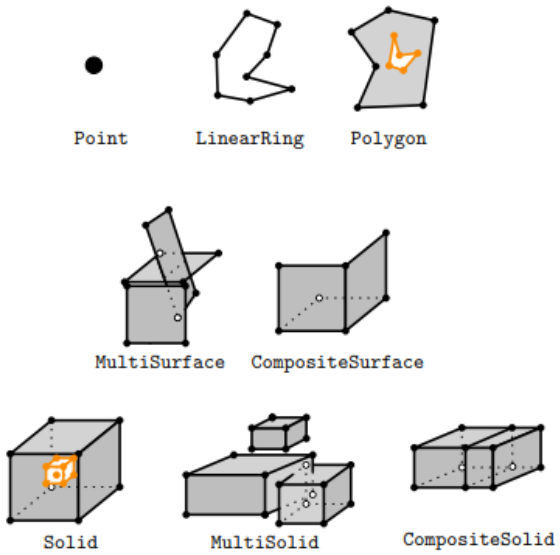


Figure 2.9.: Visual representation of `GM_Objects`(taken from Ledoux (2018)

While a geometry can be called valid if it satisfies all requirements, it is essential to note that achieving validity does not always occur. For example, Figure 2.10 shows twelve solids, eight of which are invalid. S1 does not satisfy the requirement to be watertight. S2 and S5 seem to be visually fixed, but only S2 is valid, seeing S5 uses an inner shell instead of one exterior shell (having a 'dent'). Also, S8 seems visually valid; it doesn't meet the requirement for the interior to be connected. Producing and using valid geometries is essential for many reasons. For example, valid geometries in 3D city models reflect the actual physical conditions when used in simulation use cases. Also, valid geometries in 3D city models play a role in the stability of simulations and contribute to the reproducibility of simulations. Furthermore, simple, valid geometries facilitate the generation of high-quality meshes, which are needed for several use cases of 3D city models. Lastly, valid geometries facilitate better data exchange between file types, tools and platforms.

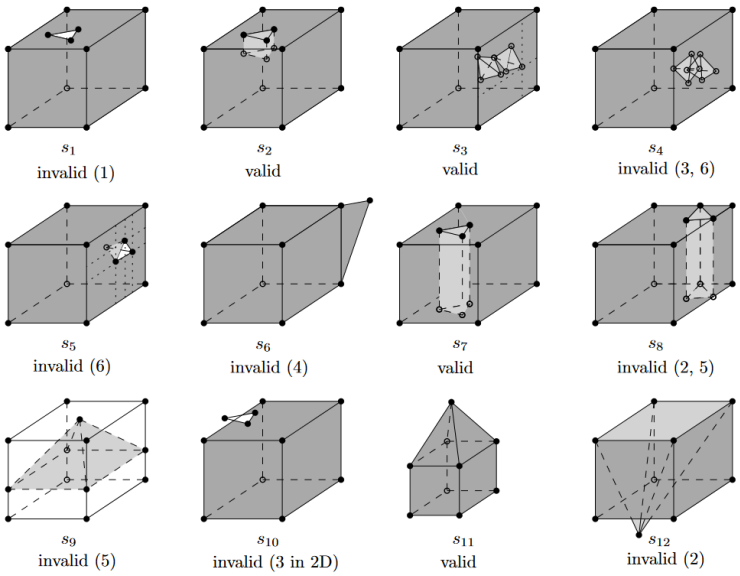


Figure 2.10.: 12 solids (taken from Ohori et al. (2022))

2.2.3 Semantics

According to Stadler and Kolbe (2007), an increasing number of applications of 3D city models rely on complex semantic information. Brodeur (2012) explains that semantics is the study of how signs (like words or symbols) relate to what they represent. In the context of 3D city models, semantics represent the meaning associated with different elements within the model. In a semantic 3D city model, objects (and their sub-parts) are labeled with their "Real world" meaning and attributes (Willenborg et al., 2018). Figure 2.11 shows how a 3D city model can be decomposed into semantics and geometry. Stadler and Kolbe (2007) argues that spatio-semantic coherence is highly beneficial for using 3D city models because of the possibility for data validation and integration.

- **Data validation** could be more accurate. As the model includes geometry and semantics, more detailed consistency rules, such as spatial and topological constraints between different features, can be specified to ensure accurate representation. For example, it can be better to ensure rooms are disjointed and fully contained within their surrounding building shell.
- **Data integration** could be more precise. Semantic information can be used for harmonization by proposing points for adjustment, such as aligning the terrain with a building's door and adjusting features like paths accordingly.

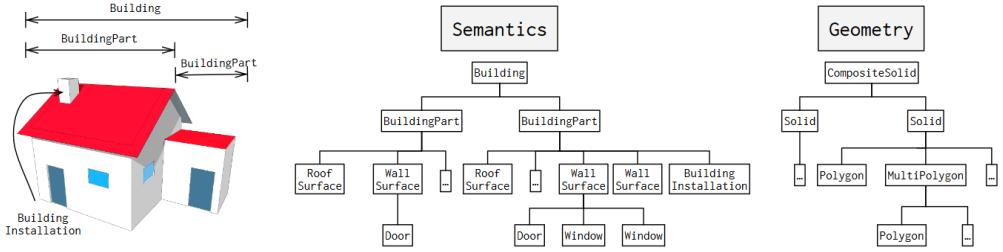


Figure 2.11.: Semantics versus geometry (taken from Ledoux et al. (2019))

To avoid every 3D city model having its classes to decompose a city (for example, "building" can be called "house" in a different model), semantic models prescribe standard classes (Ohori et al., 2022). For example, the OGC defines that CityGML can only have ten types of thematic classes (further explained in Section 2.3.1), of which "Building" is the only one in the scope of this thesis. The OGC also standardized semantic names of the surfaces of each class (Kolbe et al., 2021). Figure 2.12 show the possible semantics surfaces for objects with the thematic type building. They are broadly used in visualization and simulation tools because they always have the same name.

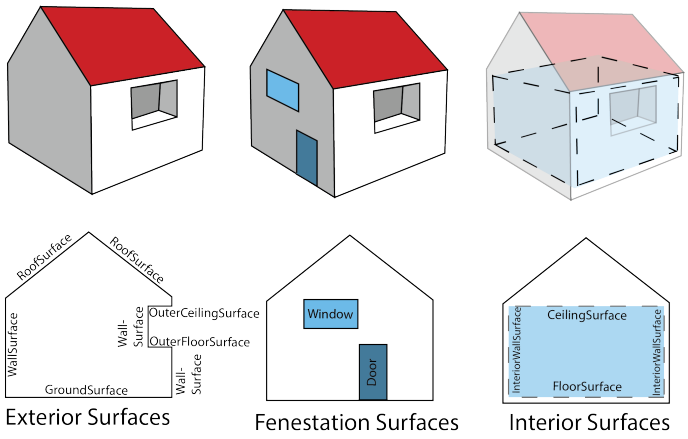
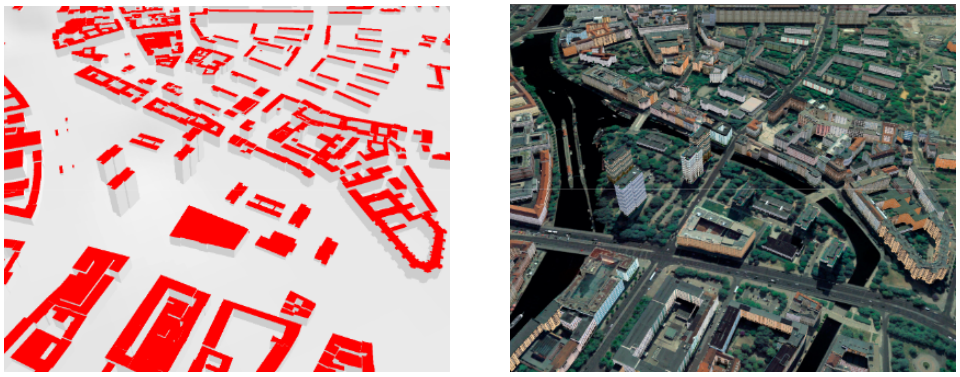


Figure 2.12.: The possible Semantics for Buildings CityGML

2.2.4 Appearance - Materials & textures

Besides semantics, which gives the meaning of objects, 3D geometries can be supplemented with textures and/or materials for a better impression of their appearance. Buchholz (2006) argues that visual appearance of surfaces enrich 3D city models. Figure 2.13 illustrates how specifying the visual appearance of surfaces such as facades, roofs, and terrain is crucial in enhancing visual realism, making it more engaging and immersive. Also, materials and textures help provide a sense of the geographic context of the city Buyukdemircioglu and Kocaman (2020). For example, regions can have distinct architectural styles, which can be visualized by textures and materials. Realistic representation is crucial for 3D city model use cases, such as urban planning, aiding positioning, and disaster management (Mao, 2011).



(a) Without textures (b) With textures

Figure 2.13.: 3D city model of Berlin (taken from Buchholz (2006)

Although materials enhance user experience and support various applications, they come with challenges related to computational resources (Buchholz, 2006) and data availability (Buyukdemircioglu and Kocaman, 2020). Also, detailed textures might not provide significant advantages (Biljecki et al., 2016b) and may even introduce unnecessary complexity practical usability (Lei et al., 2023). Lastly, it is a challenge to keep visual similarity across multiple platforms and tools, seeing rendering phases and graphics resources could be different (Mao, 2011).

2.2.5 Metadata

According to Dietze et al. (2007), metadata can be essential for 3D city models, seeing it provides additional information about the spatial data. This allows users to better understand, interpret, and manage the 3D city model. However, Labetski et al. (2018) argues that Metadata is rarely used in practice, making integrating them in 3D spatial data infrastructures (SDIs) is difficult. The ISO has standardized a framework for metadata for geographical information system (GIS) in their ISO19115, with the objective to enable users to understand and correctly use the data (the International Organization for Standardization and Consortium (2014), the International Organization for Standardization and Consortium (2019), the International Organization for Standardization and Consortium (2023)). This standard includes multiple facets, including identification information, data quality details, spatial representation, and distribution specifics. But Labetski et al. (2018) notes that while these are important, further attributes are missing, the most prominent being the level of detail and semantic object classes.

2.3 Storage of 3D city models

Seeing 3D city models are constructed from various acquisition techniques as explained in Section 2.1, their structure, format, and characteristic (Section 2.2) will greatly vary (Ohori et al., 2022). To handle 3D city models, numerous data standards exist for storing and converting 3D city models across platforms. According to McHenry and Bajcsy (2008), in 2008, there were over 140 3D file formats available, which all have unique characteristics that make them suitable for specific applications. Formats not discussed in the following subsections are not part of the scope of this thesis. However, test data could come in different formats. This

data conversion tool can be used but could lead to information loss (McHenry and Bajcsy, 2008).

The subsections will explore semantic 3D city models using standards like CityGML and the efficient CityJSON (Ledoux et al., 2019). It also covers the technical side of geometric data storage, discussing the widely-used Wavefront OBJ format, which is also widely used for storing other 3D data (McHenry and Bajcsy, 2008). Lastly, it will explain the TU Delft 3D JSON specification (tu3djson) format, a simple 3D feature exchange format.

2.3.1 Semantic 3D city models - CityGML and CityJSON

According to Ohori et al. (2022), lack of support for semantics and attributes makes that most ways to store a 3D city model can not be used easily for simulations and/or analysis. Therefore, in August 2008, the OGC proposed CityGML as the international standard for representing and exchanging 3D city models (Gröger and Plümer, 2012).

CityGML defines 3D geometry and topology but also takes the objects' semantics, their thematic properties, taxonomies, aggregations, and interrelations are taken into account (Willenborg et al., 2016). For the geometry and topology, the standardized model Geography Markup Language (GML) is used (Gröger and Plümer, 2012). GML is an Extensible Markup Language (XML) encoding for the storage of geographic information modeled according to the conceptual modeling framework used in the ISO 19100 (Cox et al., 2004). CityGML, therefore, follows ISO19107 for geometry explained in Section 2.2.2, but it has two extra restrictions (Ledoux, 2018). These two restrictions are:

1. `GM_curves` can only be linear, which results in `lineStrings` and `LinearRings` (not in the scope of this thesis)
2. `GM_surfaces` can only be planar, which results in `Polygons`

CityGML is specified formally by Unified Modeling Language (UML) diagrams illustrating the structures and concepts for all semantic models (Gröger and Plümer, 2012). Figure 2.14 gives a module overview, the vertical boxes show the different thematic modules and the horizontal modules specify concepts that apply to all thematic modules.

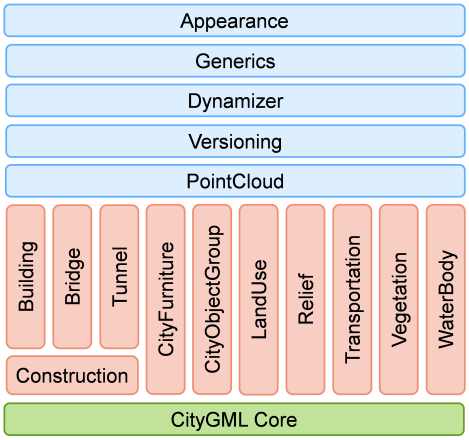


Figure 2.14.: The CityGML models (taken from Saeidian et al. (2023))

The building model is the core of CityGML, which allows the representation of thematic and spatial aspects of buildings, building parts, and accessories (Kolbe et al., 2005). Buildings are represented in LOD 0 to LOD 4 (Figure 1.2), where LOD 0 can either be represented by (horizontal) 2.5D polygons with roof level height or with footprint level height, while LOD4 represents a detailed building including, wall openings, and interior structures (Gröger and Plümer, 2012).

CityGML makes use of spatio-semantic coherence (Section 2.2.3) and the UML describes for which LOD and thematic (sub-)module, which semantics can be used. For example, the building model can only use the semantics of Figure 2.12. CityGML reuses known and used standards in other fields for the appearances(Section 2.2.4). Namely, X3D specifications for materials and COLLADA standard for Textures (Gröger and Plümer, 2012), and metadata (as explained in Section 2.2.5) can be saved. When users want to model additional objects or use extensions on the standards, an Application Domain Extension (ADE) can be used (Gröger and Plümer, 2012). ADE's define new feature types (with new attributes, geometries, and associations), which may be subtypes of existing types and uses inheritance to define application schemas (Gröger and Plümer, 2012).

CityJSON was represented in 2019 as a new JSON-based exchange format for the CityGML data model after Ledoux et al. (2019) argued that the XML-based exchange format of CityGML has several drawbacks. CityJSON implements most of the CityGML data model, and all of the CityGML modules have been mapped (Ledoux et al., 2019). CityJSON was designed from a programmer's perspective, which helps build software and Application Programming Interface (API)s. Also, CityJSON is around six times more compact with real-world dataset (Ledoux et al., 2019), which, according to van Liempt (2020), is a beneficial characteristic for web use, as it decreases the time of data transfer over a network.

The JSON encoding of cityJSON makes use of arrays (in square brackets `[ ]`) and dictionaries made by key-value pairs (in curly brackets {}). The specifications are explained and updated on their website: `https://www.cityjson.org/specs/` (Ledoux and Balazs, 2023). This Thesis focuses on version 2.0.0. The keys possible for CityJSON are shown in Figure B.1. The keys in bold are five mandatory keys, these are:

- `Type`, which is always `CityJSON`
- `version`, which is the version of cityJSON you are using
- `CityObjects` a dictionary of all the cityobjects.
- `vertices` an array of all the coordinates of the vertices.
- `Transform` values that need to be used to scale and transform the vertices

Other keys, such as metadata and appearance, store additional information. CityJSON offers the possibility to store metadata; six ISO19115 properties are supported (elaborated in Section 2.2.5), and with the Metadata Extension, more properties can be stored. Appearance is also used for CityGML X3D specifications and COLLADA. Geometry-templates can be used as a method to compress files, storing identical geometries by only defining them once. Like CityGML, CityJSON can be extended with ADEs, stored in extensions.

Figure B.2 shows the schema of a CityObject. A single `CityObject` stores in `Type` what kind of CityObject is dealt with and the possible types for CityObjects are shown in Figure B.3. In contrast to CityGML, City objects are "flattened out", meaning both 1-st and 2nd-level city objects are stored in the dictionary `CityObjects` (Ledoux et al., 2019). To still keep the hierarchy, the keys `parent` and `children` link objects to their sub (or parent) parts by their

"ID." `attributes` can be used to store extra properties about the object, however there are no standards or limitations. Lastly, `geographicalExtent` can store the bounding box of the City Object.

Unlike CityGML, the geometry doesn't store all the coordinates of its vertices but links the geometry to the `vertices` by using the boundaries to store in integer arrays. Each integer refers to an index in the `vertices` array. The `Geometry` defines the 3D geometric primitives, storing the 3D `boundaries` of the objects `type` and `Lod`. The eight geometry types are:

- `MultiPoint`
- `MultiLineString`
- `MultiSurface`
- `CompositeSurface`
- `Solid`
- `MultiSolid`
- `CompositeSolid`
- `GeometryInstance`.

This geometry is modeled following the ISO standards explained in Section 2.2.2. `MultiPoint`, `MultiLineString` and `GeometryInstance` are not in the scope of this thesis.

The boundaries of these geometry types are a hierarchy of arrays, depending on the type. Figure 2.15 shows that geometry arrays are built from the "lower" geometry type; for example, a composite solid consists of multiple solids, so a composite solid array is an array of solids. Note that complex (`Composite-`) and aggregates (`Multi-`) are on the same hierarchical level.

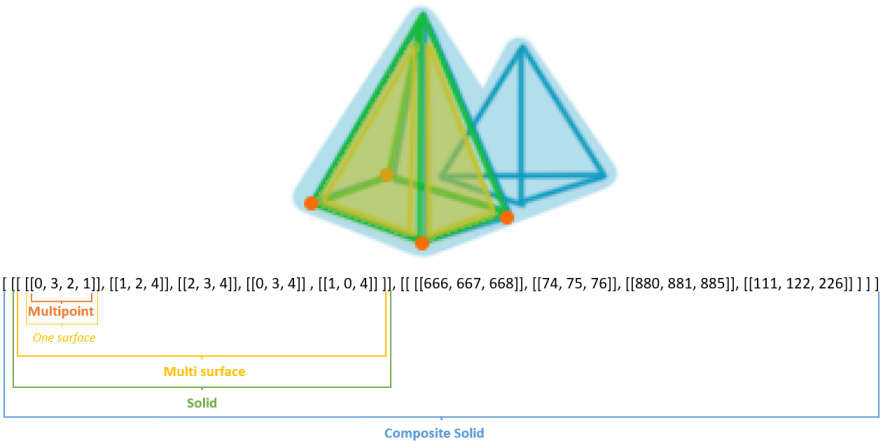


Figure 2.15.: Depths of the boundaries for different types

For the building LOD, CityJSON supports 16 refined "TU Delft LoDs" (Figure 2.5), in contrast to CityGML, which only used the five base LODs. Geometries may also store `semantics`, `materials` and `textures` as additional information representing semantics and appearance of

the CityObject. `semantics` has two values. `surfaces`, an array of array of semantic objects, and `values`, a hierarchy of arrays with integers, where integer refers to the index in the "surfaces". The `values` hierarchy is two less than the array `boundaries`, as a surface that has a depth of two can only have one semantic. `materials` is built slightly differently, seeing it is a collection of key-value pairs. The key is the theme of the material, referring to the themes in `appearance`. The value `values` is again a hierarchy of arrays with integers, which is two less than `boundaries`. `textures` is build the same a `materials`, but the depth of the `values` array is equal to `boundaries`, as the indices refer to the UV positions of the corresponding vertices. `textures` is not in the scope of this thesis. If a surface has no Semantics, Material, or textures, the integer (or indices for textures) is replaced by `null`.

2.3.2 Geometry file format - Wavefront OBJ format

Wavefront OBJ is an open data format for 3D graphics, developed in 1970 by Wavefront Technologies (Biljecki and Arroyo Ohori, 2015). When used for 3D city models, it does not contain any attributes or semantics; it only contains the 3D geometry of the building models. OBJ is a text-based format that stores each vertex's position and the faces that define each polygon as a list of vertices (bourke). It could also store the UV position of each texture coordinate vertex and vertex normals, but those are out of the scope of this thesis. Vertices are stored as `v x y z`, where x y z: are the x, y, and z coordinates for the vertex, which can be floats (Listing 2.1 shows four example vertices).

```
# Vertices
v 1.0 1.0 1.0
v 1.0 1.0 -1.0
v 1.0 -1.0 -1.0
v 1.0 -1.0 1.0
```

Listing 2.1: Example vertices .obj File

Faces are stored as `f v1 v2 v3 ....  vn`, where v are integers, where integer refers to the index in the vertices (starting index = 1). A face needs at least three integers stored in a counter-clockwise order to show the face normal (Listing 2.1 shows two examples of faces).

```
# face(s)
f 1 2 3 4
f 5 6 7 8
```

Listing 2.2: Example face .obj File

Objects and sub-objects (geometries) can also be named and grouped together. With the tag `o` you can give the object (all geometries in the file) a name, and with the tag `g` all the faces till the next `g` are grouped together (Listing 2.3). The vertices can be stored within the tags of a group or as a whole at the beginning of the file, seeing the indexes of the vertices do not change. Groups can also get materials that describe the visual aspects of the polygons; this is done by `usemtl [material name]`, where the material name matches a named material definition in an external `.mtl` file. Listing B.1 shows an example obj file with the object name "ThreeSolids" and three groups and their material, named "Tetrahedron," "Hexahedron," and "Octahedron."

```
o [object name/id]
  ...
  g [group name/id]
  ... #faces
```

Listing 2.3: object and groups .obj File

A drawback of OBJ geometries is that they do not allow the storage of all ISO 19107 primitives. Also, holes and inner shells can not be used. Lastly, attributes and semantics cannot be stored for different parts/elements.

2.3.3 Simple 3D feature exchange - tu3djson

tu3djson is developed by the TU Delft 3D geoinformation research group (geoinformation research group). It is built like geoJSON, an open standard format designed for representing simple geographical features, standardized in 2016 (Butler et al., 2016). But seeing that geoJSON is restricted to 2D primitives, tu3djson was developed to store 3D features in a simple format. Figure B.4 shows the schema for a tu3djson object, with in bold the mandatory keys, namely `type`, which is always tu3djson, and `feature` which contains 0 or more feature objects. The `features` consist of `type`, naming a type or id, `properties`, which can be anything, and `geometry`. A geometry object must have a `type`, `boundaries`, and `vertices`. The possible types for the geometry are:

- `MultiPoint`
- `MultiLineString`
- `MultiSurface`
- `CompositeSurface`
- `Solid`
- `MultiSolid`
- `CompositeSolid`

Based on the type, the `boundaries` are a hierarchy of arrays with integers, with the same depths as CityJSON (explained in Section 2.3.1). The integers refers to the index of the `vertices`.

2.4 Validation of geometries

Because the ISO standardizes requirements for primitives (outlined in Section 2.2.2), geometries in 3D city models should compile with the definitions for their concerned primitives. As a result of the ISO19107 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 2.16.

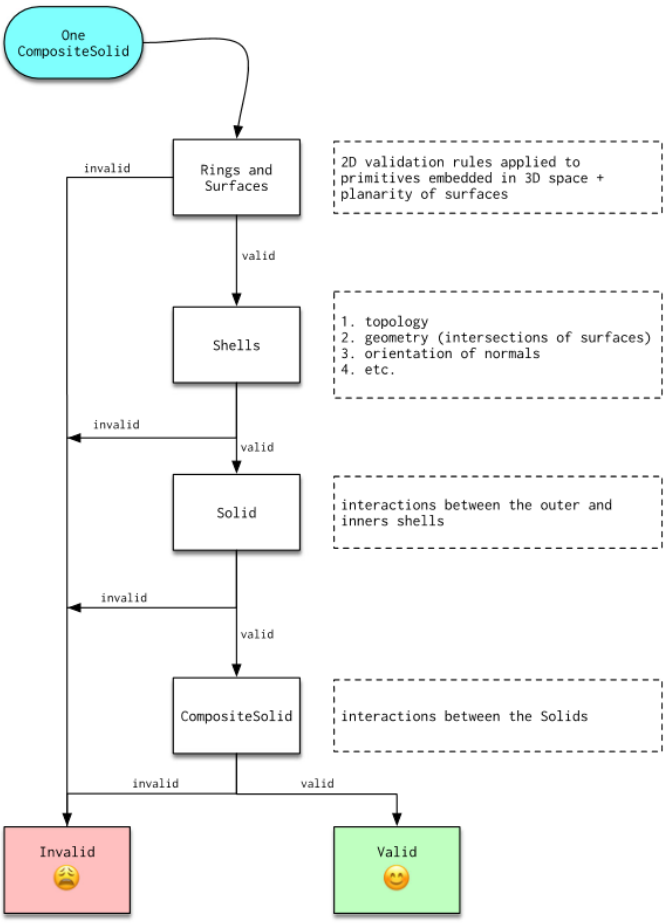


Figure 2.16.: Validation workflow (taken from Ledoux (2018))

To automatically handle the validation process, Ledoux (2018) implemented a validator called *val3dity*. At each level, the validator checks the requirements; if they are not met, it reports an error code. val3dity has a total of 36 error codes, which are shown in Figure 2.17. 27 of those are primitive-based and tested by unit tests. They are divided into five different dimensions, as follows:

- LinearRings (error `1xx`), being the lower-dimensional primitives, are validated and projected on a 2D plane.
- Polygons(error `2xx`) are validated on the interactions and orientations of its LinearRings, also projected on a 2D plane.
- Shells and CompositeSurfaces (error `3xx`) are validated in 3D space on topology, orientation, and interactions of its Polygons in 3D space.
- Solids (error `4xx`) are validated on the interactions and orientations of their shells.
- CompositeSolids(error `5xx`) are validated on the intersections and interaction of its solids.

The other 11 errors are based on CityGML and CityJSON (error 6xx), IndoorGML (error 7xx), and "other" errors (error 9xx), being input and unknown, which influence the validation process. Errors on IndoorGML and input are not part of the scope of this thesis.

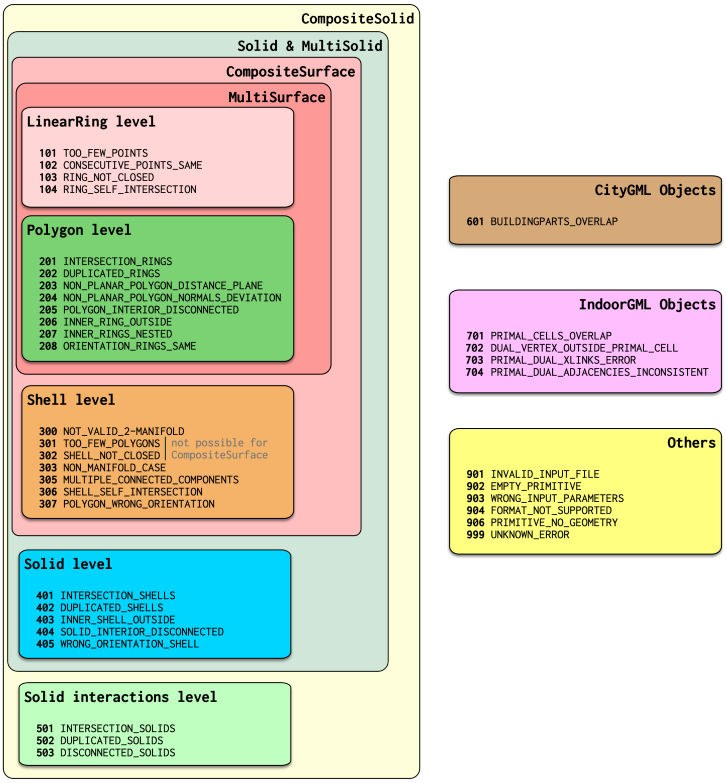


Figure 2.17.: val3dity error codes (updated version[1] from Ledoux (2018))

In 2016, Biljecki et al. (2016a) analyzed the most common geometric and semantic errors in CityGML datasets. They found that datasets without errors are rare and that if (nearly) valid, they are primarily LOD1 models. Studying 37 datasets, they observed that sometimes errors appear to be random (for example, missing wall as shown in Figure 1.3a) in a dataset, while others are encountered more systematically. According to Biljecki et al. (2016a), the most geometric common errors in datasets are (in no particular order):

- Geometries not properly snapped - Invalid solids (Error 3xx) often occur due to not snapped geometries, leading to intersecting surfaces and non-watertight solids. Mostly, these errors are tiny, so close inspection by zooming in is necessary to detect them. Figure 1.3b shows an example of this.
- Non-planarity - Non-planar polygons are the most common error in all the datasets tested. Most planarity issues arise from deviations of just a few centimeters. Although this might not pose problems for many applications, it is invalid according to the standards.

[1] updated version from https://val3dity.readthedocs.io/en/latest/errors/

- Polygon orientation - Polygons having a wrong orientation prevent polygons from forming watertight solids and are less useful for spatial analysis. Figure 2.18 shows an example dataset with wrong orientations.

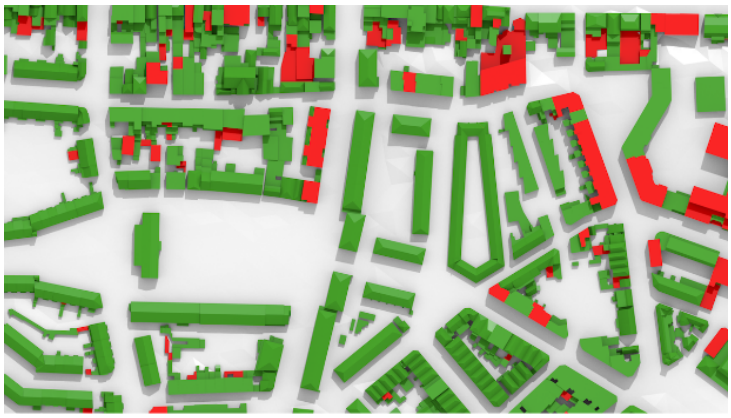


Figure 2.18.: Orientation of roof surfaces, where green is correct, red is wrong (taken from Biljecki et al. (2016a))

Biljecki et al. (2016a) also noted that intersecting solids (for example Figure 1.3c) could be a common error. Upon manual inspection, they found that building parts overlap, resulting in incorrectly calculated building volumes. But *val3dity* wasn't able to cover CompositeSolids (error `5xx`) or overlapping building parts (error `601`) at that time.

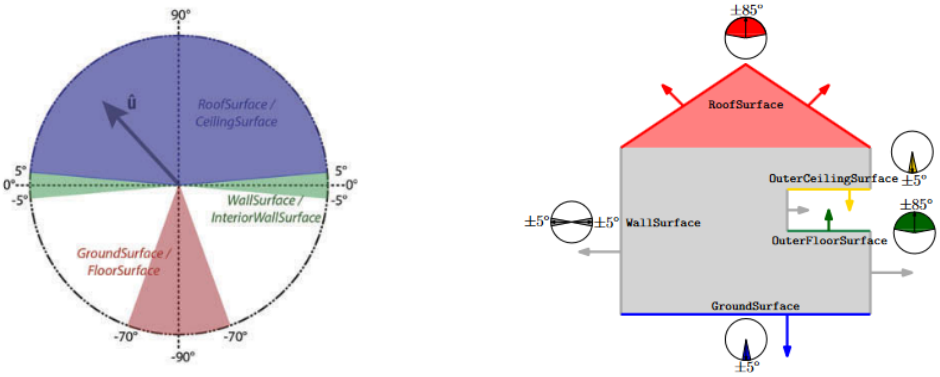
2.5 Validation of Semantics

According to Wagner et al. (2016), validation of semantics can be described in three different ways, namely:

1. Validation on the thematic model. For example, ensure that elements like `BuildingPart` are consistently child elements of a `Building`. This validation is not in the scope of this thesis; however, when deleting a parent or a child feature because it cannot be repaired, the thematic classes should be changed (described in Section 6.2.3).
2. Validation of semantics in relation to the geometry. For example, is a `GroundSurface` label surface really a surface that represents the ground plate of a construction? This form of validation will be discussed in this section.
3. Validation of mandatory attributes for (thematic) models. For example, is in the metadata of `ConstructionEvent`, the `dateOfEvent` really a date? This validation is not in the scope of this thesis.

Wagner et al. (2016) validates semantic surfaces for `Building` and its subparts based on the normal vector of each face. They propose that `RoofSurface` and `OuterFloorSurface` have a positive z-direction, `GroundSurface` and `OuterCeilingSurface` have a negative z-direction and `WallSurface` have a zero z-coordinate. Boeters et al. (2015) adds an angle range to the requirements for the normal, resulting in Figure 2.19a. Note that also classification validation for interior surfaces is added to this model. Biljecki et al. (2016a) uses the same angle

ranges as Boeters et al. (2015), but adds the `OuterCeilingSurface` and `OuterFloorSurface` for validation, resulting in Figure 2.19b. Doors, windows, and closure surfaces are not considered in the three models.



(a) Validation model of Boeters et al. (2015)

(b) Validation model of Biljecki et al. (2016a)

Figure 2.19.: Semantic surfaces of buildings based on the normal of the surface

Biljecki et al. (2016a) argues that there are limitations to the validation based on the normal approach. For example, `RoofSurface` and `OuterFloorSurface` are validated against the same range of angles, which could result in a false positive when validating. Also the validation based on normals is not conclusive for all architecture, for example Figure 2.20 shows how `WallSurface`'s (grey) are now labeled `RoofSurface` (red) to be considered valid based on the normals.

Figure 2.20.: Example of an uncertain situation for semantic validation (taken from Biljecki et al. (2016a))

3 Existing repair methods

This chapter discusses the Odd-Even Paradigm vs. SetDiff Paradigm, followed by existing repairs; only existing repair methods are discussed. Resulting in focusing on scientific papers for which an implementation is available (and thus ignoring purely theoretical solutions). The existing method sections are divided into geometric repair. Secondly, simplify meshes, which helps (repaired) geometries reduce complexity for improved performance and can help with additional requirements for, for example, CFD. Lastly, adding and/or repairing semantics is needed for additional requirements such as estimating solar power.

3.1 Odd-Even Paradigm vs. SetDiff Paradigm

For validating and repairing geometries, it is vital to determine what the interior and exterior regions are. The odd-even paradigm (Foley, 1996) is a straightforward method for determining whether a point is inside a complex geometry with multiple boundaries, such as polygons with holes or shells. This method works by drawing a (imaginary) line from the point in question to infinity in any direction. The paradigm establishes the point's location by counting how often this line intersects with the geometry's boundaries (Figure 3.1). If the count of intersections is odd, the point is considered inside the geometry; if the count is even, the point is outside.

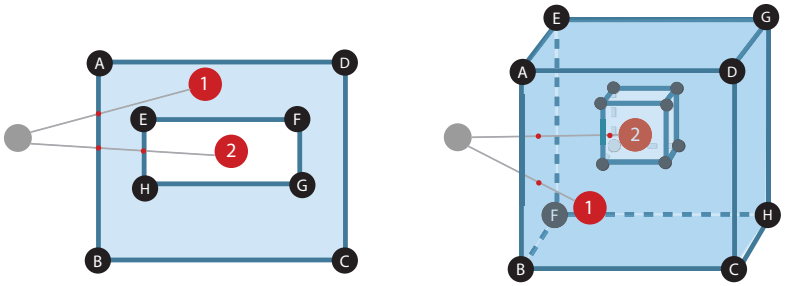


Figure 3.1.: Odd-Even paradigm, where 1 is in both situations inside (odd number of intersections) and 2 is outside (even number of intersections)

Ledoux et al. (2014) proposed the set difference (setDiff) paradigm. This method explicitly subtracts the interior spaces from the exterior boundary to accurately define the shape. Starting with the area or volume enclosed by the exterior ring or shell, the setdiff paradigm subtracts the areas or volumes enclosed by each interior ring or shell using the set difference operation. For polygons, this can be mathematically expressed as: $P = r \setminus (r_1 \cup r_2 \cup \ldots \cup r_n)$ where r represents the exterior boundary, and $r_1, r_2, \ldots, r_n$ represent the interior boundaries. For 3D shells, this operation defines the solid's volume by subtracting the union of all interior shells from the exterior shell.

The odd-even and setdiff paradigms are helpful in different situations. The odd-even paradigm is simple and fast, making it great for validating if a point is inside or outside a shape.

However, for example, when shells are not closed, it can yield incorrect results. The setDiff paradigm, on the other hand, is more robust in yielding results but has difficulty finding errors. For example, an inner shell outside is no problem, seeing it is always exterior space. Therefore, the best paradigm to use differs for each repair.

3.2 Geometric repair

According to Zhao et al. (2013), geometric repair methods can be divided into Local and Global Approaches. Botsch (2010) uses roughly the same division, but local approaches are called surface-oriented, and global approaches are called volumetric. Attene et al. (2013) analyzed that local repair methods use local modification, which does not modify the whole mesh. This repairs the defects, but not creating new defects can not be guaranteed. On the other hand, global repair methods use completely re-meshing the object, which results in guaranteed results but loses details of the object. In the following subsections, both approaches are explained with existing repair methods.

3.2.1 Local repair methods

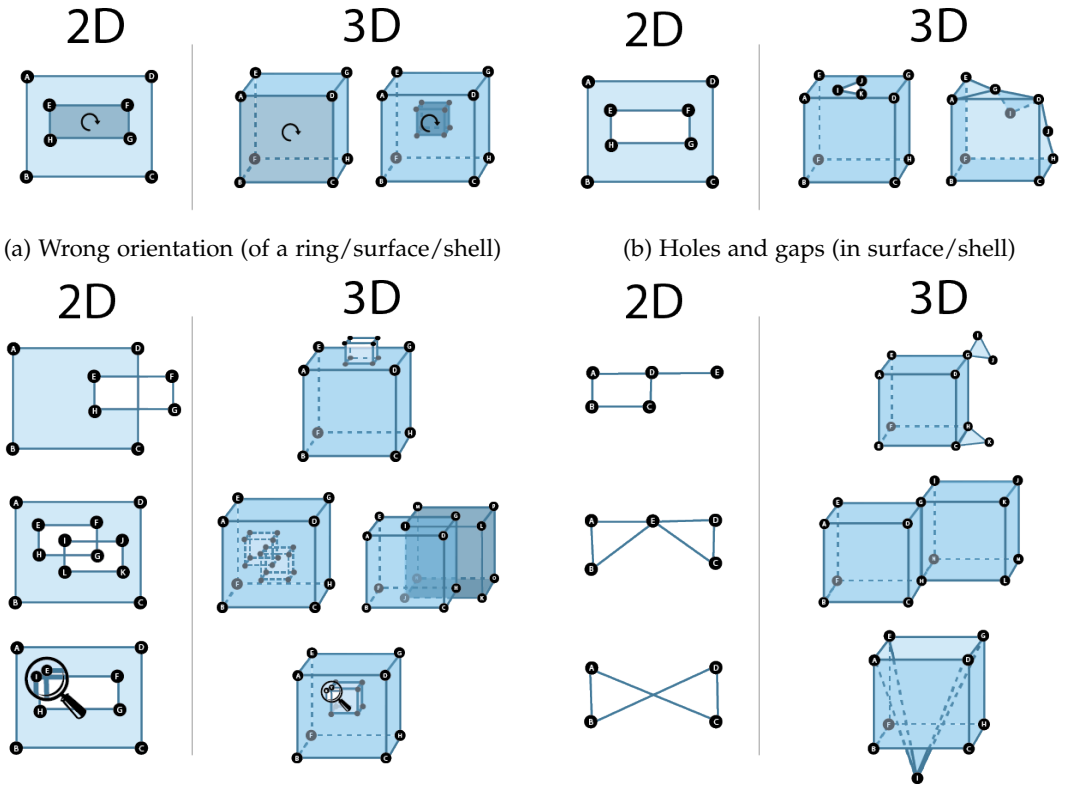
Local repairs try to identify and resolve artifacts explicitly. They change the model only minimally and can preserve the original structure (Botsch, 2010). However, Zhao et al. (2013) argues that this approach can only solve one or a few types of errors. Therefore, these repair methods are rarely fully automatic, as the model cannot have other errors. This makes local repairs not very robust (Botsch, 2010). Figure 3.2 shows 4 of the most common errors repairable by local repair methods, each of them will be covered in the next subsections. Note that these local errors can happen in 2D and 3D.

Orientation

Orientation errors arise in 2D when the interior ring(s) do not have an opposite orientation from their exterior ring (clockwise versus counterclockwise). In 3D orientation, errors arise when the normal polygons from the exterior shell do not point outwards (counterclockwise orientation when viewed from the outside) or inwards for interior shells. One or more polygons need to be flipped to repair the wrong orientation. Wagner et al. (2015) proposes to find the polygons that need to be flipped by calculating the normal and intersecting it with all the polygons. Intersection can be calculated with the Moller Trombore Algorithm (Möller and Trumbore, 1997). When the number of intersection points is odd the orientation is valid, otherwise it needs to be flipped. A prerequisite for this method is that the shell is closed.

Another method, proposed by Takayama et al. (2014), is a sampling of visibility through ray shooting. This method samples random points on polygons, which are used as the origin for the rays. For each point, two rays are shot, one in a random direction and one in its inverse direction, ensuring the same number of rays are fired from both sides of the polygon. The polygons normal needs to be flipped towards the side where there are the least rays that intersect with other polygons. This method doesn't have the prerequisite to have a closed shell and is therefore more robust, however more calculations need to be done.

Loriot et al. (2023) also implemented Orientation Functions for Computational Geometry Algorithms Library (CGAL), which can compute or change the orientation of faces and surfaces. This method translates the shell to a Polygon mesh. When the connectivity (and or orientation) is unknown, the set of faces is called a polygon Soup. The function `CGAL::Polygon_mesh_processing::orient_polygon_soup()` ensures that the polygons are consistently oriented; this also works for non-closed shells. The



(a) Wrong orientation (of a ring/surface/shell)

(b) Holes and gaps (in surface/shell)

(c) Overlap and duplicates (of rings/shells)

(d) Ring Self-intersections/ Shell non-manifolds)

Figure 3.2.: Examples of local repairable errors

outputted Surface mesh can then be outward- or inward-oriented with the function `CGAL::Polygon_mesh_processing::orient()`, but a prerequisite for this method is that the shell is closed.

Hole and Gap filling

Holes and gaps are not necessarily errors in 2D, depending on the data structure; a polygon may have holes, but in 3D, holes and gaps can result in a shell not being closed. Solids need to consist of watertight shells; those so-called openings need to be closed.

All the existing repair methods for closing a shell can close openings of triangular meshes. Some of the existing shell closing algorithms are:

- Tekumalla and Cohen (2004) proposed filling openings of triangular meshes using a Moving Least Squares projection. The Moving Least Squares projection involves a two-step algorithm, iteratively making the opening convex and adding vertices until the opening is closed. This method smoothly fills holes and can be applied to both planar and non-planar openings.
- Jun (2005) introduced a method capable of filling complex openings. Their algorithm projects the opening onto a plane and divides it into several sub-openings. These sub-

openings are further projected and divided until the holes become simple enough to be triangulated.

- Zhao et al. (2007) proposed closing openings in triangle meshes using the Advancing Front Mesh Technique. This technique iterates by advancing over the edges of the opening and adding new triangles to bridge the gap until it is closed. Afterward, the algorithm smooths the filled triangles by refinement based on the Poisson equation. This method is effective for non-planar complex openings as well.
- Botsch (2010) suggests employing Liepa's hole-filling algorithm (Liepa, 2003), which triangulates the opening to mimic the triangles of the boundary area. The patched opening is then refined by adjusting the positions of the new vertices. Like the previous methods, this approach is suitable for non-planar complex openings.
- Rashidan et al. (2022) proposed another local repair method called "triangular mesh repair," focusing on filling holes specifically in 3D city models. This method utilizes triangulation of polygons but can only close planar and almost planar openings.
- Loriot et al. (2023) implemented opening filling functions for CGAL. This method involves translating the shell to a polygon mesh, which must be triangulated using `CGAL::Polygon_mesh_processing::triangulate_faces()`. Openings can then be filled using either `CGAL::Polygon_mesh_processing::triangulate_hole()` or `CGAL::Polygon_mesh_processing::triangulate_and_refine_hole()`.

A drawback of all these methods is that they only work for triangular meshes, which result in translating the geometry in a triangular mesh and afterwards detriangulating it to the original data structure, which is prone to errors.

Overlap

Overlap errors can arise in 2D as well as in 3D by intersecting rings (2D), shells(3D), or even solids (3D). The most common approach to change or create new geometries from existing (overlapping) geometries is Boolean point set operations. Figure 3.3 shows the four primary set operations, union, intersection, difference, and symmetric difference (Ohori et al., 2022). Boolean operations offer a systematic method to resolve overlaps. Applying operations such as union allows the overlapping regions to be merged into a single geometry. Intersection can isolate the common volume shared by overlapping shapes, while difference can subtract one geometry from another to eliminate the intersected portion. The symmetric difference, on the other hand, can be used to retain only the non-overlapping parts of the intersecting geometries.

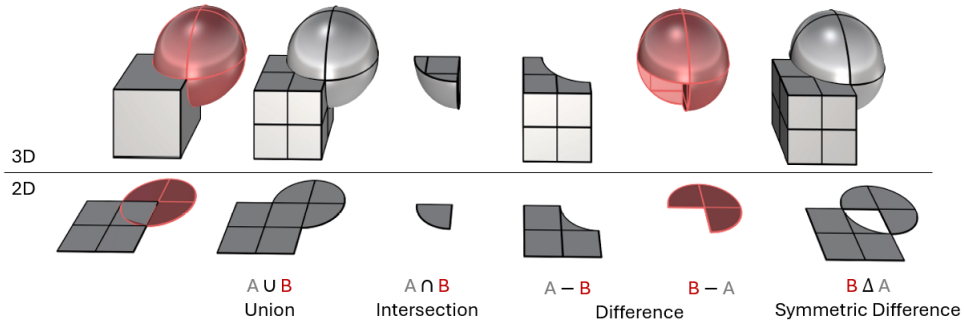


Figure 3.3.: Boolean operations in 2D and 3D

Although boolean operations are versatile, Barki et al. (2015) argues that besides the utility and importance, boolean computations are complex due to topological changes and geometric degeneracy. To address this challenge, the Nef polygons and polyhedra theory was developed (Bieri, 1995). This data model represents the geometry based on local pyramids, which can be made by intersecting the neighborhood of a vertex with a sphere (or circle in 2D) (Granados et al., 2003). Figure 3.4 shows the local pyramids of two Nef-Polygons. On Nef-geometries, a boolean operation can be computed in three steps:

1. Subdivision, which overlays the geometries and creates an overall structure with new vertices, edges, and faces at the intersections.
2. Selection, which checks which vertices, edges, and faces should be part of the output
3. Simplification, which removes all the other vertices, edges, and faces.

Seel (2023) and Hachenberger and Kettner (2023) implemented these operations in CGAL for 2D and 3D geometries, although it is not a prerequisite, the algorithm is less robust for nonsimple geometries. CGAL also has a boolean operation for meshes implemented (Loriot et al., 2023), but it has perquisites that the mesh needs to be triangular and closed. Barki et al. (2015) therefore proposed a more robust boolean operations on 3D meshes, which work on non-manifold and not-closed meshes. However, the code he implemented is not open source.

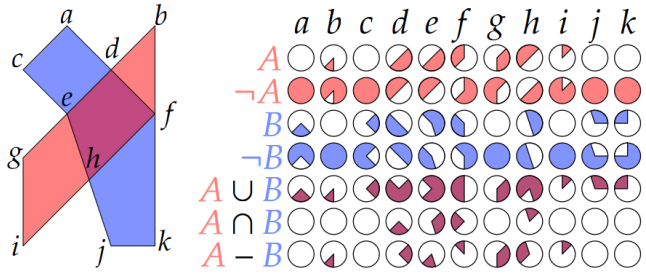


Figure 3.4.: Nef polygons boolean operations and with their local pyramids (taken from Ohori et al. (2022))

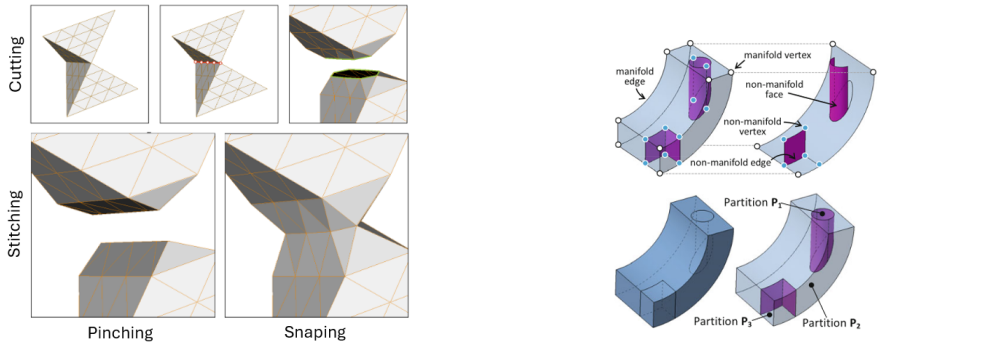
Self intersection (2D) and Non-manifolds (3D)

Non-manifold errors always arise from a form of self-intersecting. Figure 3.2d shows the three most common cases, being:

1. Pieces collapsing in a lower dimension (line for 2D or dangling face for 3D),
2. Self-intersection on an edge or point creating two or more objects in the same dimension,
3. Self-intersection elsewhere creating two or more objects in the same dimension.

For 2D non manifolds Subramaniam (2003) proposed a method for partitioning self intersecting polygons into simple polygons by using the non-zero winding number rule. Alternatively, Ledoux et al. (2014) proposed a method based on triangulation that uses two repair options, namely an extension of the odd-even algorithm and a point set difference rule. This method, named *prepair*, focuses on automatically repairing single GIS polygons. Ohori et al. (2012) proposed an extension, named *pprepair*, which can repair a set of polygons.

For 3D non-manifold Gueziec et al. (2001) proposes a method called cutting and stitching for triangulated meshes. Firstly, the intersecting edges and points are marked, then disconnected, by cutting faces. The following step stitches edges by pinching the geometry into two parts or snapping the two parts back together (Figure 3.5a). Alternatively, Mikchevitch and Pernot (2013) proposed a method to split non-manifold geometries into partitions (Figure 3.5b); however, this method only works when all the nonmanifold parts are shell objects.



(a) Method of Gueziec et al. (2001)

(b) Method of Mikchevitch and Pernot (2013)

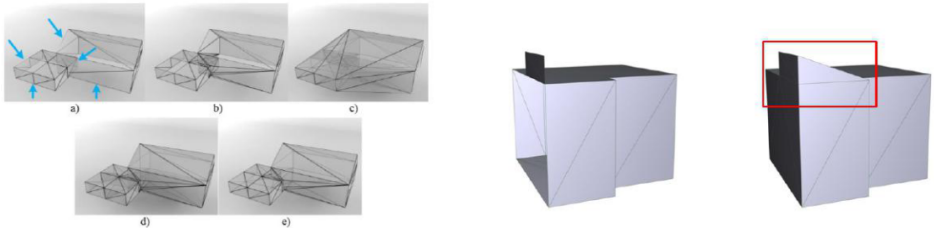
Figure 3.5.: Repairing 3D Non-Manifolds

3.2.2 Global repair methods

Global repairs use intermediate volumetric representation from which an output model can be extracted (Botsch, 2010). Instead of identifying errors and resolving those, a new representation is made based on the (interior) space of the original geometry. Botsch (2010) argues that global repair methods are typically fully automatic and often implemented robustly. However, the downside is that it resamples the geometry, often resulting in a loss of detail. It also changes the original data structure, so after the repair, you possibly need to convert back to the original data structure, which can cause new errors. Converting back to the original data structure also often result in loss os semantics, materials and textures. The two most common global repair methods, Shrink wrapping and Voxelization, are covered in the following subsections.

Shrink wrapping

The concept of mesh wrapping has been around for some time. In 1998 Bernardini and Bajaj proposed a method for wrapping manifolds using Alpha-shapes and in 1999 Kobbelt et al. proposed a shrink wrap approach for re-meshing. However, these methods were not proposed for flawed models with holes or intersections. Therefore, Lee et al. (2010) developed the Cartesian shrink-wrapping technique. The method uses a Cartesian grid, which is adaptively refined until the target resolution forms a watertight shell. Although this method is automatic Lee et al. (2010) argued that complex geometries remains challenging and often requires significant user intervention for the correct output. The algorithm also needed improvements for memory efficiency. Zhao et al. (2013) proposed a new shrink-wrapping method based on graph theory, where all the building vertices are graph vertices. The graph applies tetrahedralization on all faces and its convex hull (Figure 3.6a). These method effectively repairs gaps, holes and self-intersections, however it cannot repair overshoots (Figure 3.6b) and is very sensitive to floating point errors.



(a) (b)

Figure 3.6.: Shrink wrap (a) method (b) example of a defect (taken from Zhao et al. (2013))

Alpha wrapping

Alpha wrapping is a derivative of the "alpha shapes" concept (Edelsbrunner and Mücke, 1994). Alliez et al. (2023b) implemented this function for CGAL, which could be used on point clouds or triangulated meshes. The function `CGAL::alpha_wrap_3()` uses shrink wrapping based on two parameters. Alpha, the main parameter, determines which features will appear in the output by controlling the size of the empty spaces. The second parameter, offset, controls the tightness of the result. Figure 3.7 shows how both values affect the output mesh. Finding the correct parameters to solve defects and preserve detail still often requires significant manual user intervention for the proper output.

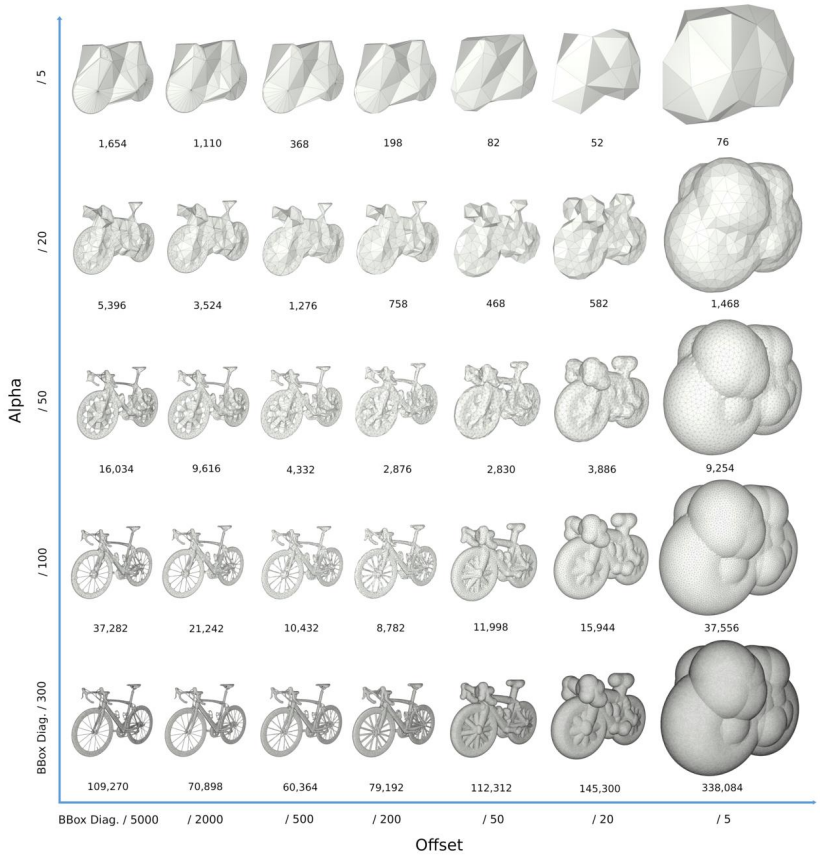


Figure 3.7.: Different alpha and offset values (taken from Alliez et al. (2023b))

Voxelization of Polygon Mesh

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 3.8 in this repair method, input is converted into a binary 3D grid. This method is very robust but has two significant drawbacks: potential shift of the geometry and potential loss of attributes. Also the slow processing of voxelization is not ideal, 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.

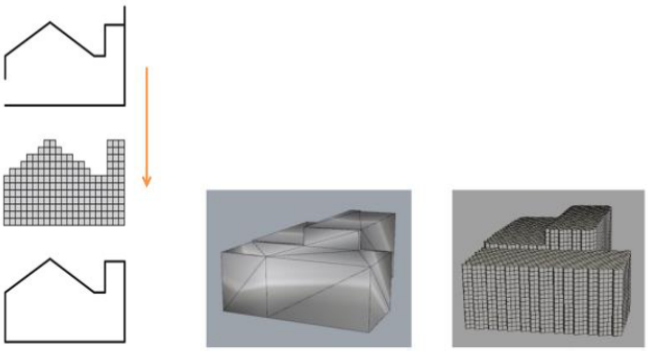


Figure 3.8.: Voxelization (taken from Mulder (2015))

3.3 Simplifying meshes

A complicated geometry can degrade the mesh quality, which leads to errors (taken from Park et al. (2020)). Therefore, simplification helps in the repair process. It is an additional requirement for some use cases. Park et al. (2020) proposed a simplification methodology, which can be seen in Figure 3.9. The prerequisite for this method is that a building consists of one simple solid. Therefore, the preprocessing consists of repairing the geometry. Also, if the geometry consists of multiple primitives, it is joined (boolean union)into one. The next step is to classify the faces by marking the not-insignificant faces. The marked faces planes bound the base of the simplified model, which is generated from the volume.

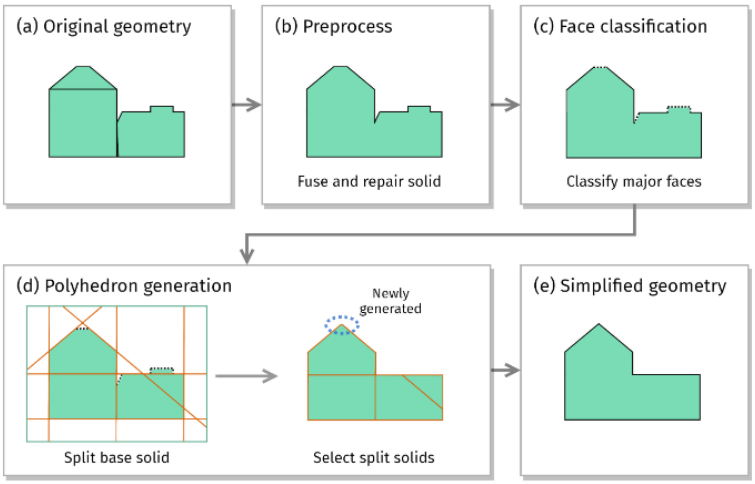


Figure 3.9.: Simplification methodology of Park et al. (2020)

3.4 Adding and/or repairing semantics

As discussed in Section 2.5, semantic surfaces can be validated against face normals of the geometry. Boeters et al. (2015) proposed an method to automatically enhancing CityGML LOD2 with surface semantics. This method is for `Building` and its sub-classes and has as a prerequisite that the geometry is valid. It assumes that the geometry is bounded by surfaces and based on the normals, it classifies the surfaces accordingly (Figure 2.19a). Although this is not a repair method, the methodology can be used to add semantics to surfaces that are now labeled `None` and change the nonvalid surfaces to the correct type. Diakité et al. (2014) also proposed a method for automatic semantic labeling of buildings based on heuristic rules. Although most heuristic rules are based on normal orientation checking, some are based on the relationships between the components. These relationships also make it possible to classify doors and windows. There is no prerequisite for the geometry to be valid, which makes the program more robust. However, a big drawback is that the standardized names (Figure 3.10) aren't used. Therefore, the algorithm is not usable for CityGML and CityJSON, making it less useful for this thesis.

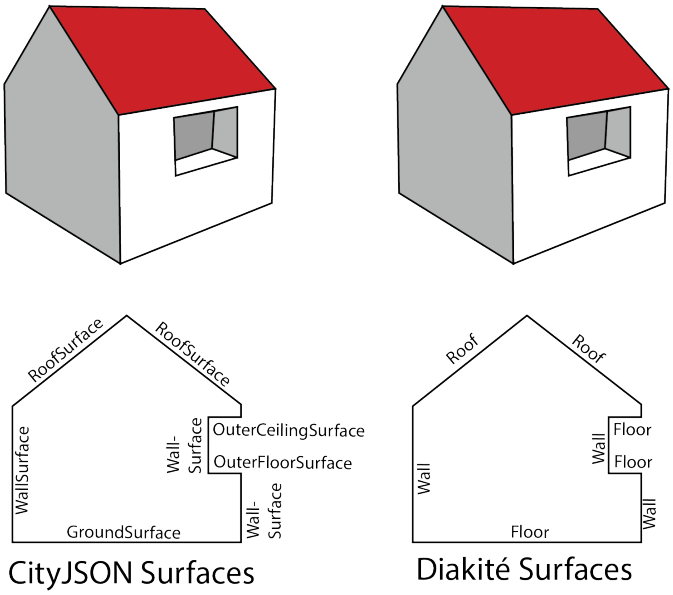


Figure 3.10.: semantics CityJSON vs semantics Diakité

4 Methodology for automatic repair of semantics 3D city models

The workflow of the proposed repair methodology is provided in Figure 4.1. The automatic repair process is done in the repair loop. This chapter will focus on the methodology of geometric repairs by explaining how val3dity can be used for the repair process (Section 4.1). Followed by repair approaches for all val3dity errors divided per primitive level (Section 4.2, Section 4.3, Section 4.4, Section 4.5, Section 4.6 and Section 4.7). Preserving semantics and materials will also be discussed for each repair approach. The methodology for use case implementation and user requirements and its repairs will be discussed in Chapter 5.

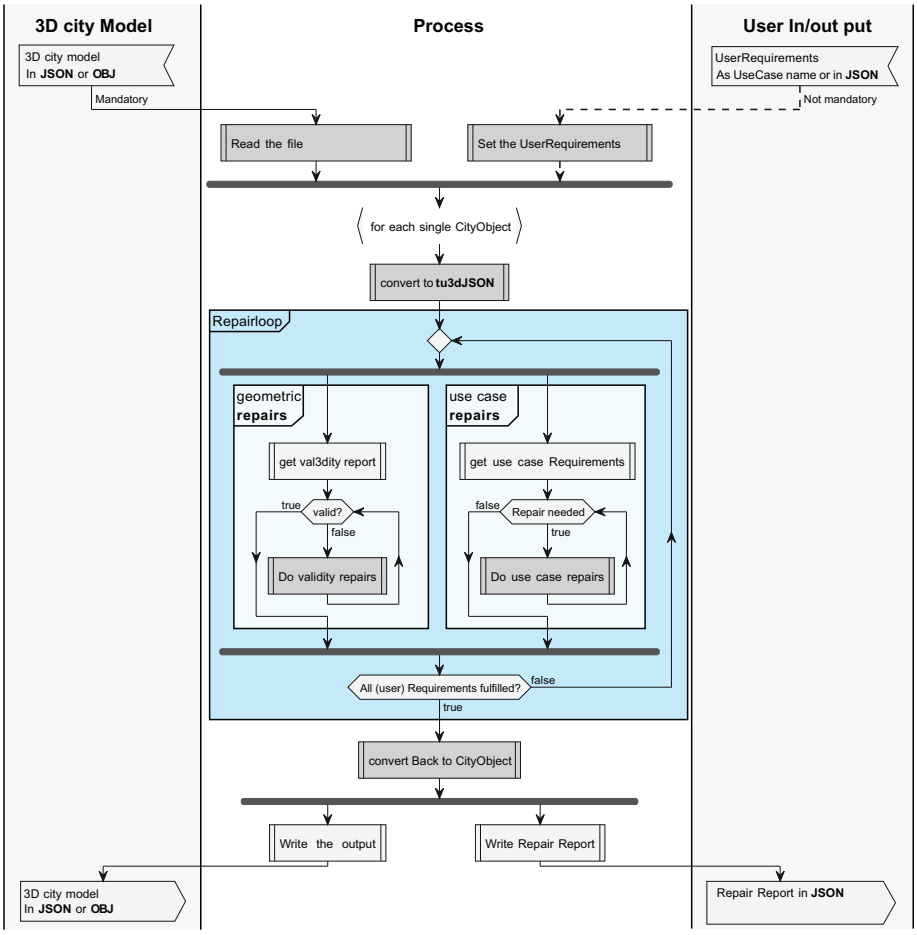


Figure 4.1.: Flowchart of the methodology

4.1 Validation by val3dity

As explained in Section 2.4, val3dity could be used for validating geometries and finding errors. val3dity is a command-line interface (CLI), but also has a web application. To use it in other programs, val3dity can also be compiled as a library; when included, the API can validate files and return a report with errors. Figure 4.2 shows the schema of a val3dity report, listing errors at 3 levels:

1. errors with the input files (errors 9xx), which are not in the scope of this thesis.
2. errors with the features, e.g., Buildings in CityJSON (errors 6xx and 7xx), in light red
3. errors with the geometries (errors 1xx – 5xx), in red

When a primitive is validated, the report starts the repair process. By using the blue key (Figure 4.2), the error and the location (`id`) of the error can be found. The report, therefore, can be used to trigger doing the corresponding repair on the invalid location (which is explained in the following subsections). Seeing that val3dity works hierarchically, it can be assumed that its lower-level primitives are valid. Also, errors on the same level are made hierarchically (e.g., 206 before 208), so those requirements can also assumed to be valid. These assumptions help with the perquisites of local repair processes.

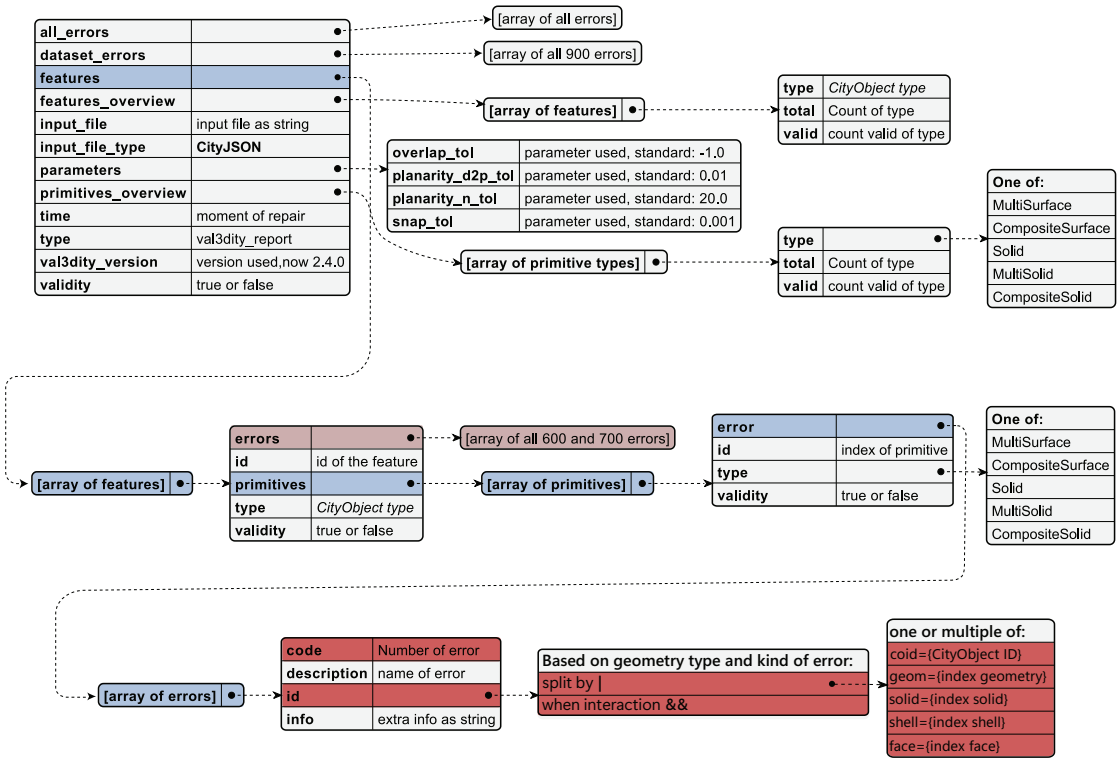


Figure 4.2.: Val3dity report structure (used data in red, keys to get there in blue)

While both ISO 19017 and CityJSON mention that each Surface must be planar (i.e., all its points, used for both the exterior and interior rings, must lie on a plane), the concept of

tolerance is not mentioned, which Biljecki et al. (2016a) believes is a shortcoming. *Val3dity* uses the concept of tolerance to ignore small errors (Ledoux, 2018). Tolerance is used for three situations, namely:

- Planarity of polygons, which is checked in two ways:
  - Are all vertices on the same plane? This is calculated by fitting a plane by the least-square adjustment. For all vertices, the distance from the plane is calculated. If the distance is less than the `planarity_d2p_tol` (whose default is 1 mm), the polygon will be seen as planar.
  - Are all possible triangulation polygons in the same plane? This is calculated by comparing the normals of all the triangles after triangulation. If the deviation of the normals is less than `planarity_n_tol` (which default is 20 degrees), the polygon will be seen as planar.
- Snapping close vertices, which is done by calculating the distance of two vertices. If they are less than `snap_tol` (which default is 1 mm) apart, the two vertices will be snapped together.
- distance between primitives, which can be used in two situations:
  - Erosion, which can be used when two should not be overlapping. When the overlap is less than `overlap_tol` (which default is 0 mm), the two primitives do not overlap (Figure 4.3).
  - Dilatation, which can be used when two are disjoint. When the overlap is less than `overlap_tol` (which default is 0 mm), the two primitives are not disjoint(Figure 4.3).

The values of these tolerances can inputted in val3dity by the user. Therefore, for the repair methodology, the user should also be able to set these tolerances.

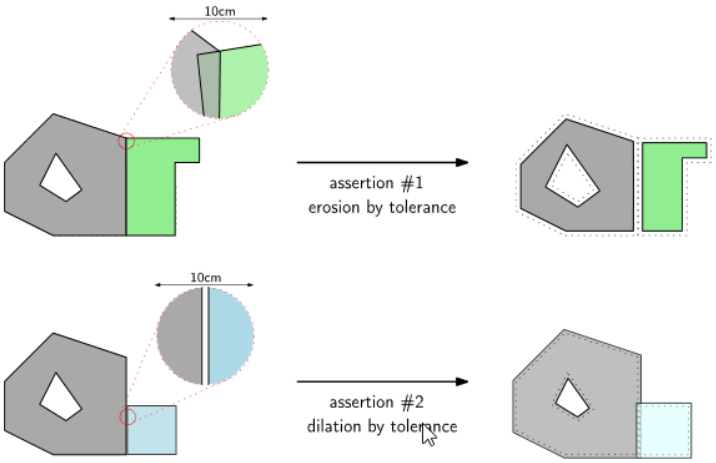


Figure 4.3.: Example of how the tolerance is applied when validating a Composite-Solid containing 2 Solids(taken from Ledoux (2018))

4.2 Ring level repair approaches

As explained in Section 2.2.2, rings are simple and closed curves and have a few standards to be considered valid. The following subsections delve into non-compliant standards (based on the val3dity error codes Figure 2.17) and their corresponding repair approaches.

101 - Too few points - A (Linear)ring is a closed LineString, a composite of Lines. If there is only 1 line (a curve with 2 points), there is no chance of repairing it, and therefore, only deletion is an option (Figure 4.4). If the ring is an inner ring, it will be deleted, but if it is the outer boundary of a polygon, all the inner rings will also be deleted. The semantics and materials will also be deleted. Suppose the polygon was essential for the shell. In that case, it will later in the process generate a too few polygons (repair 301) or a shell not closed (repair 302) error, for which repairs are explained in Section 4.4 (repairs shown in Figure 4.16 and Figure 4.17).

Figure 4.4.: Approach to repair: Too few points

102 - Consecutive points are the same - A ring is simple if it does not pass through the same point twice. Two consecutive points can be the same on two occasions: (1) they are the same point, (2) they are the same point because they are within the snap-tolerance (`snap_tol` explained in Section 4.1). To repair this error, only the first, in sequential reading order, of the two vertices is kept (Figure 4.5). Note that only keeping one of the points could result in having too few points afterward (repair 101). The semantics and materials of the geometry will not be changed.

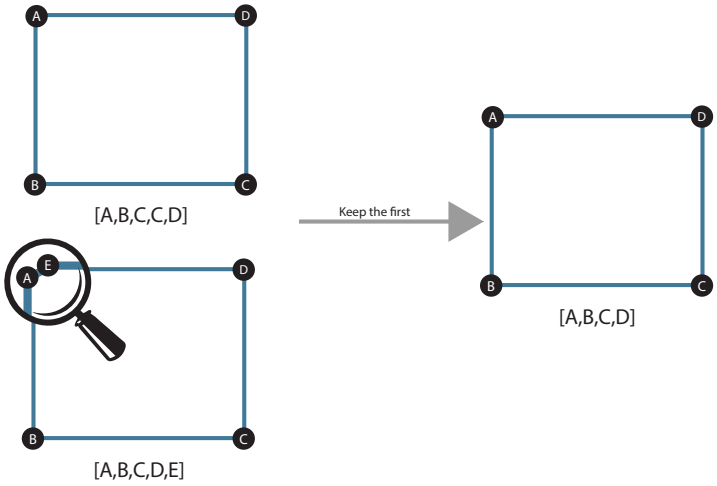


Figure 4.5.: Approach to repair: Consecutive points same

103 - Ring not closed - For a ring to be closed, the start point must be the same as its endpoint. In CityJSON and OBJ, this error cannot happen; seeing rings are not defined by ending it with the start point. Therefore, this error and its repair are out of the scope of this thesis. If the repair process is extended to formats such as GML rings and to JSON-FG, this repair could be done by adding the first point to the end of the ring. The semantics and materials of the geometry will not be changed.

104 - Ring self-intersection - A ring is simple if it does not pass though the same point twice, therefor a ring should not self intersect, intersection can happen at an explicit point or with edges that cross each-other. To decide how to repair the first check is if the points are not co-linear, resulting in the ring collapsing into a line. If that is true (example 1 Figure 4.6), the only repair option is deletion. Similarly to repair 101, if the ring is an inner ring, it will be deleted, but if it is the outer boundary of a polygon, all the inner rings will also be deleted. If the ring is not (completely) collapsed into a line, the convex hull of the points is used as a repair (example 2,3,4 Figure 4.6). The convex hull, which is the smallest convex polygon possible to describe a point set, is the most robust point-to-polygon algorithm available (Commandeur, 2012). The semantics and materials of the geometry will not be changed.

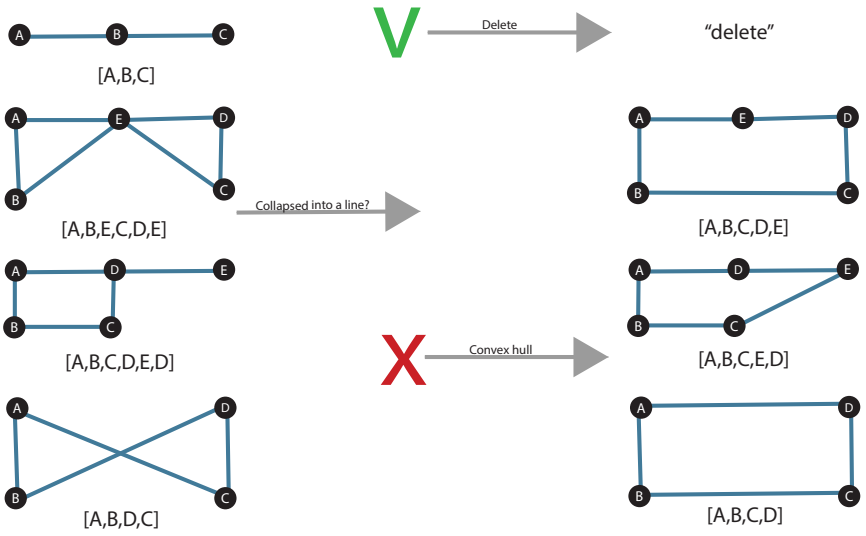


Figure 4.6.: Approach to repair: ring self-intersection

Another less robust option for repairing self-intersection could have been the calculation of a concave hull (Da, 2023). Using a Concave hull instead of a convex hull could capture more intricate details of a ring. For example, it could help get rid of dangling pieces (for example, case 3 Figure 4.6)) by seeing those vertices as outliers. However, the drawback of using a concave hull is that it is parameter-dependent; the $alpha$-value needs tuning for every ring of every dataset to find the correct outlier detection and not splitting the ring into multiple rings. Having to set the correct α-value makes the algorithm less robust comparing to a convex hull.

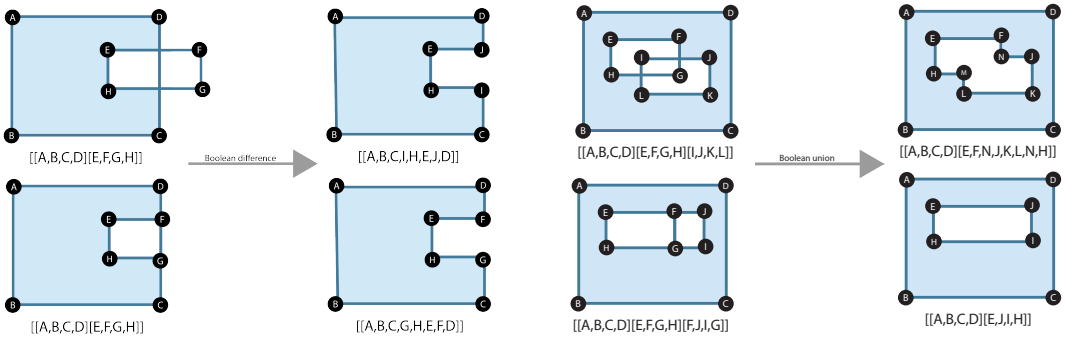
Ledoux et al. (2014) also proposed a repair solution using a constrained triangulation, which is also robust and already implemented in the automatic repair tool *prepair*. However, the

drawback of this algorithm is that disconnected interiors (which happen with cases 2 and 4 of Figure 4.6) are handled by splitting the ring into multiple rings. For local-level repairs, this thesis opted for the most straightforward method to ensure that rings remain intact, leading to the choice of convex hull over constrained triangulation.

4.3 Polygon level repair approaches

As explained in Section 2.2.2, polygons are simple and planar surfaces and have a few standards to be considered valid. The following subsections delve into non-compliant standards (based on the val3dity error codes Figure 2.17) and their corresponding repair approaches.

201 - Intersection rings - In a polygon, boundaries cannot cross, and the exterior of a polygon with holes is not connected. When rings in a polygon intersect, neither standard is complied with. This can happen in two situations: when an inner ring intersects with the exterior ring (Figure 4.7a) or when two inner rings intersect (Figure 4.7b). This intersection could (partly) have the same edge or overlap with the edge of another ring. Boolean operations are used to repair this. When an inner ring intersects with the exterior ring (Figure 4.7a), the inner rings needs to be subtracted from the exterior, therefore the difference is calculated. When two inner rings intersect, they need to become 1, and therefore, the union is calculated (Figure 4.7b). The semantics and materials of the geometry will not be changed.



(a) Intersection exterior and inner ring

(b) Intersection two inner rings

Figure 4.7.: Approach to repair: Intersecting Rings

202 - Duplicate rings - In a polygon, boundaries cannot cross, so boundaries can also not be duplicates. This can happen when the same ring is used twice or when points are within the snap tolerance (`snap_tol` explained in Section 4.1). Preferably as the repair, only the first ring should be kept (Figure 4.8), but at this moment, val3dity doesn't make a difference between 201 and 202 anymore, so the boolean repairs as described in Figure 4.7 is used. For duplicate inner rings, this doesn't change the outcome, but when an inner ring is a duplicate of the exterior, the whole outer ring will be deleted, which deletes the polygon as a whole. When a polygon is deleted, its semantics and materials are also deleted; otherwise, the semantics and materials of the geometry will not be changed.

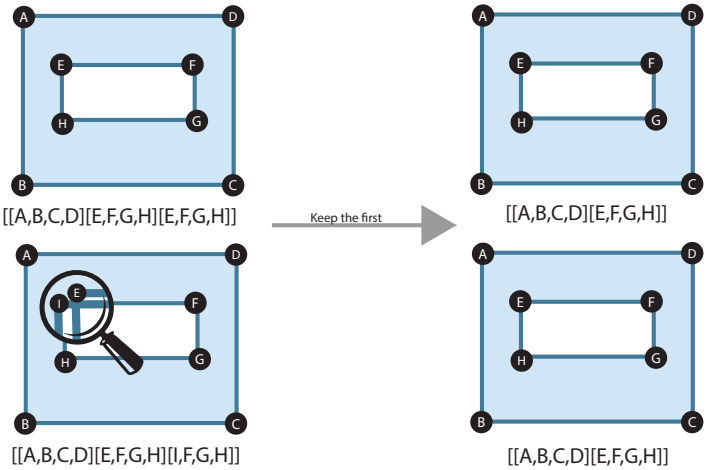


Figure 4.8.: Approach to repair: Duplicated rings

203 - Non-planar polygon distance plane - Polygons need to be planar, so Val3dity validates this by checking if all points are less than the planarity tolerance (`planarity_d2p_tol` explained in Section 4.1). When points are not within the tolerance, the first check is if points are within the Max projection tolerance given by the user (Section 6.3). Suppose one of the points is above the projection tolerance; then The polygon will be triangulated (Figure 4.9). Otherwise, the outlier points are projected onto a new plane (Figure 4.9) by these four steps:

1. Calculate the best-fitting plane for all the points in the polygon
2. Find the outliers that are outside of the planarity tolerance
3. Calculate a new plane without the outliers
4. Project the outliers on the new plane

When the points are projected, the semantics and materials of the geometry will not be changed. However, when a polygon is triangulated, all the faces will inherit the semantics and materials of the original polygon.

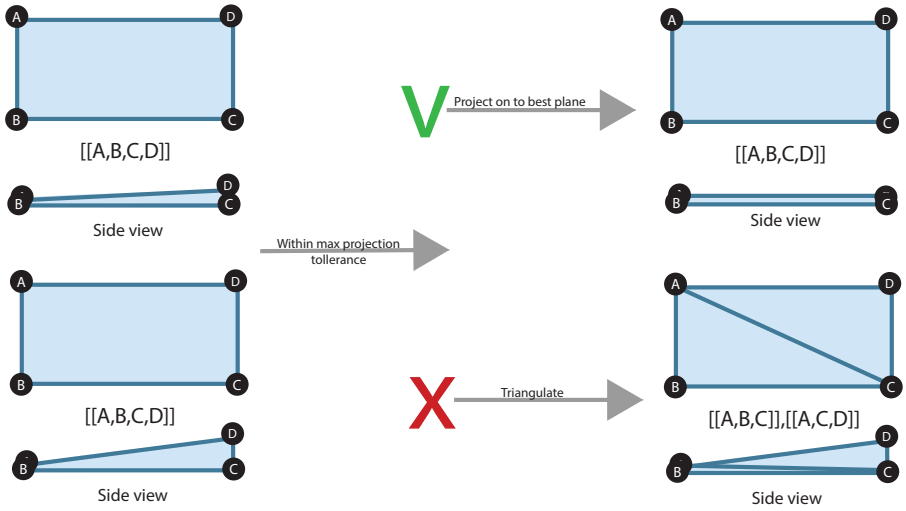


Figure 4.9.: Approach to repair: Non-planar polygon distance plane

204 - Non-planer polygon normal Deviation - Polygons need to be planar, next to the planarity tolerance Val3dity also validates that all possible planes, formed by (non co-linear) 3 points in a polygon, have normals that do not deviate more than the tolerance for planarity based on normals deviation (`planarity_n_tol` explained in Section 4.1). This validation is added when all points are within the planarity tolerance, but there is a ”fold” in the polygon.

1. Triangulate the polygon
2. Calculate all the normals and compare those
3. Find the triangles with the deviation and mark their points as fold points
4. Calculate a new plane without the fold points
5. Project the fold points on the new plane

The semantics and materials of the geometry will not be changed.

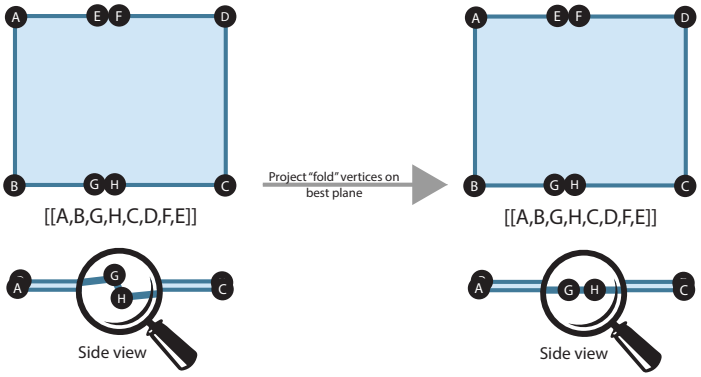


Figure 4.10.: Approach to repair: Non-planar polygon normal deviation

205 - Polygon interior disconnected - The interior of every Polygon needs to be a connected point set; therefore, if one or more holes disconnect the interior, the polygon needs to be split into two (or more) polygons (Figure 4.11). This can be done by subtracting the holes from the interior and then looping over all the created (sub-)polygons that are formed. All the parts will get the semantics and materials of the original polygon.

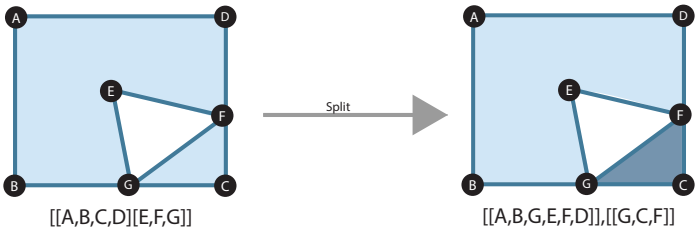


Figure 4.11.: Approach to repair: Polygon interior disconnected

206 - Inner ring outside - The exterior of a polygon with holes cannot be connected. Therefore, inner rings cannot be outside of the polygon. When an inner ring is outside, the first check is if the user wants to keep all introduced geometries (Section 6.3). If not, the inner ring, which is outside, is deleted, and the semantics and materials of the geometry will not be changed. Otherwise, the inner ring orientation is reversed and added as a separate polygon (Figure 4.12). The added polygon will receive the same semantics and materials as the original polygon.

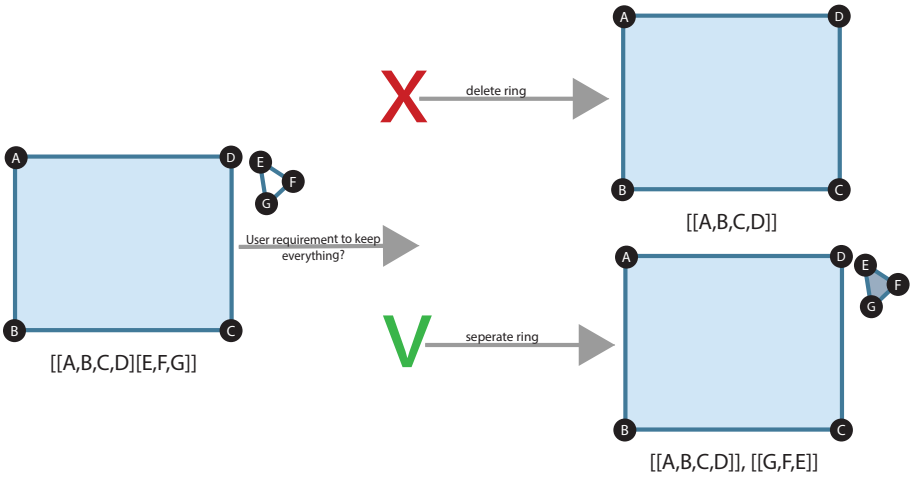


Figure 4.12.: Approach to repair: Inner ring outside

207 - Inner rings nested - The exterior of a polygon with holes cannot be connected. Therefore, inner rings cannot be nested. When an inner ring is nested, first, it is checked if the user wants to keep all introduced geometries (Section 6.3). If not the nested inner ring is deleted, , also the semantics and materials of the geometry will not be changed. Otherwise, the nested inner ring

orientation is reversed and added as a separate polygon (Figure 4.13) with the same semantics and materials as the original polygon.

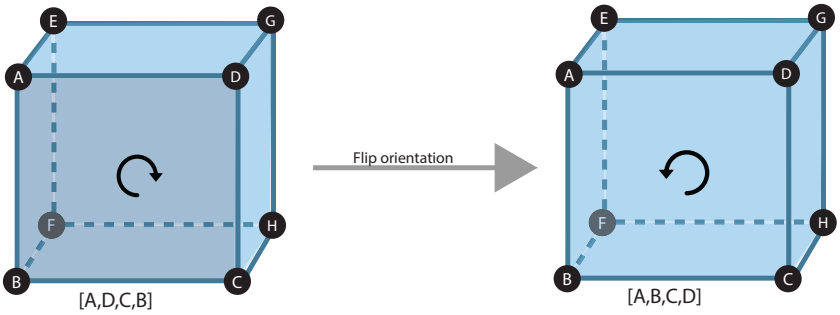


Figure 4.13.: Approach to repair: Inner ring nested

208 - Orientation rings same - The exterior ring must have the opposite orientation of its interior ring(s). Exterior rings must be oriented counterclockwise and the interior(s) clockwise. If rings have the same orientation, the orientation of the exterior ring is calculated, and all inner rings are oriented in the opposite orientation by reversing them if they are the same (Figure 4.14). If the exterior was the ring that was in the "wrong" direction, the shell would end up not being manifold (repair 303) or having a polygon with a wrong orientation (repair 307) and would be repaired accordingly. The semantics and materials of the geometry will not be changed.

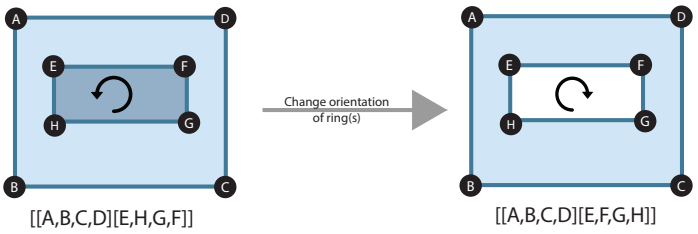


Figure 4.14.: Approach to repair: Orientation rings same

4.4 Shell level repair approaches

As explained in Section 2.2.2, Shells are defined by a closed set of polygons and have a few standards to be considered valid. Some of these standards are also needed for CompositeSurfaces, although they are not shells. The following subsections delve into non-compliant standards (based on the val3dity error codes Figure 2.17) and their corresponding repair approaches.

300 - Not valid 2-manifold - This error happens when the exact error is unknown. Since we do not know the problem, local repair methods are impossible. Therefore, a global method should be used. Alpha wrap (as explained in Section 3.2.2) is the chosen repair approach (Figure 4.16), seeing its robustness and easy-to-use implementation in CGAL (Alliez et al., 2023b). Seeing

alpha wrap outputs a triangulation, detriangulation can be done by re-meshing planar patches. Preserving and materials can be done by checking overlap with original polygons. If a new polygon is the same or entirely inside an original polygon, the semantics and materials of the original polygon will be assigned. If multiple original polygons are (partly) inside a new polygon, the largest overlap will be used for semantics and materials. If a new polygon overlaps with no original polygons, its value will be `NONE` unless polygons surround it in the same plane with all the same semantics and materials. The NONE values could later be repaired with the methodology explained in Section 3.4.

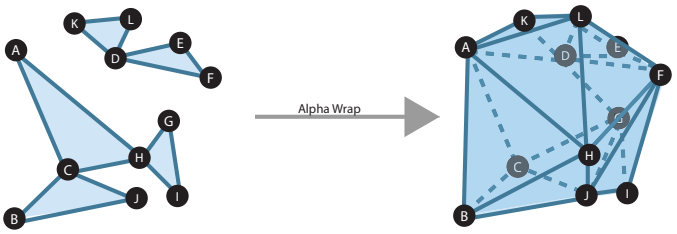


Figure 4.15.: Approach to repair: Not valid 2-manifold

An alternative methodology could be to fill the polygons with point samples and use point cloud reconstruction methods (Section 2.1.1) to reconstruct and repair geometries. Methods such as Polygonal Surface Reconstruction or Advancing Front Surface Reconstruction could be useful, however alpha wrapping is preferred for its simplicity, speed, and robustness.

301 - Too few polygons - A shell needs to be closed. Therefore, a shell needs at least four polygons. However, the only exception is a triangular pyramid with one missing triangle Alam et al. (2014). If there is only one polygon or if there are 2 or 3, but they are on the same plane, the shell can never be closed (first example of Figure 4.16). If the shell is an inner shell, it will be deleted; otherwise, if the user requires all geometries to be watertight (Section 6.3), the shell will be deleted, and with it, the whole solid. If the shell is the outer boundary, it will be made into a MultiSurface, and all inner shells of the rest of the solid will be flipped and become part of the multisurface. The semantics and materials will be the same as the original.

To keep the repair as local as possible, secondly, hole filling (as explained with 302) is tried (second example of Figure 4.16). The semantics and materials stay the same, and the new polygons will get the value `NONE`. If the hole filling doesn't work, alpha wrap, as explained with 300, is used (third example of Figure 4.16), and also, the preservation of semantics and materials of 300 is used.

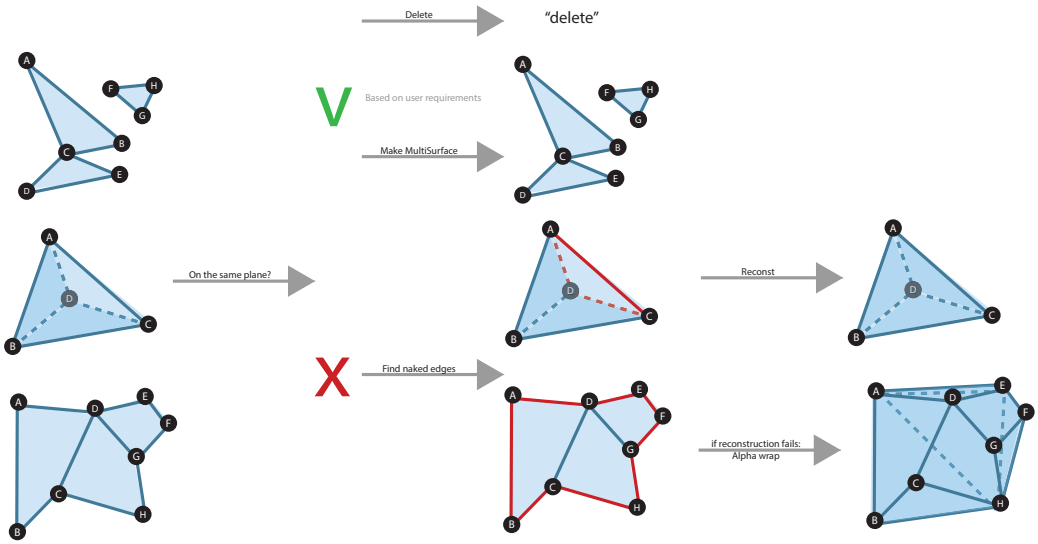


Figure 4.16.: Approach to repair: Too few polygons

302 - Shell not closed - A shell needs to be closed; therefore, holes and gaps need to be filled. To do this. First, naked edges need to be found, and from there, reconstruction of holes and gaps can be done (first example of Figure 4.17). The semantics and materials of the original polygons will stay the same. If a new polygon overlaps with no original polygons, its value will be `NONE` unless it is surrounded by polygons in the same plane with all the same semantics and materials. When reconstruction fails, alpha wrapping, as explained at 300, can be used (second example of Figure 4.17), and also, the preservation of semantics and materials of 300 is used.

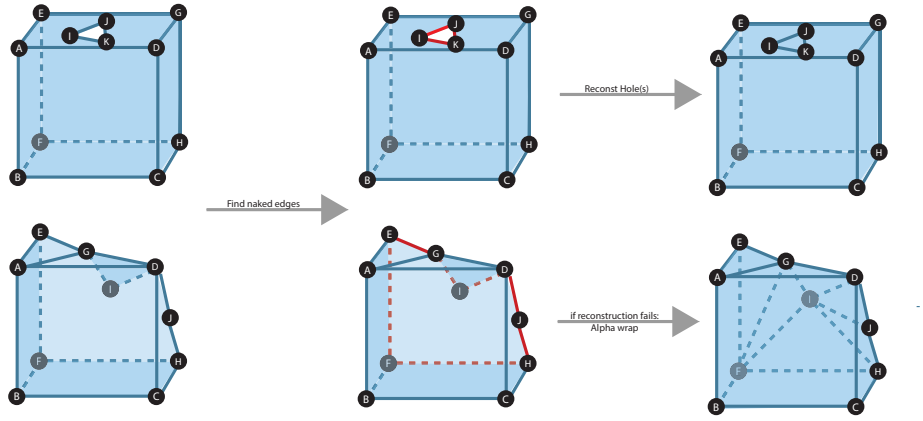


Figure 4.17.: Approach to repair: Shell not closed

303 - Non-manifold case - A shell needs to be closed, and a composite surface needs to consist of all connected surfaces. Therefore, there cannot be more than 2 polygons incident to an edge, and the vertex cannot have "umbrellas". Sometimes, 303 is also returned when it is actually 307. The approach for this repair consists of the following steps:

- Find the edge(s) or vertexes which are overused
  - If none are found, it is a 307 error, and a polygon needs to be flipped
- separates the connected components by splitting on the edge or vertex
- If the new connected component(s) doesn't have a volume (first example Figure 4.18)
  - For outer shells and CompositeSurfaces, if the user wants to keep all introduced geometries (Section 6.3), the new components are added as a separate MultiSurface. Otherwise, they are deleted.
  - For inner shells, the new components are deleted
- if the new connected components have a volume: (second example Figure 4.18)
  - For outer shells and CompositeSurfaces, if the user wants to keep all introduced geometries (Section 6.3) as one component, the new component is added as part of a CompositeSolid. Otherwise, the new component will be added as a separate solid
  - For inner shells, the shell will be split into 2 (or more) inner shells.

The polygons will all keep their original semantics and materials even if flipped.

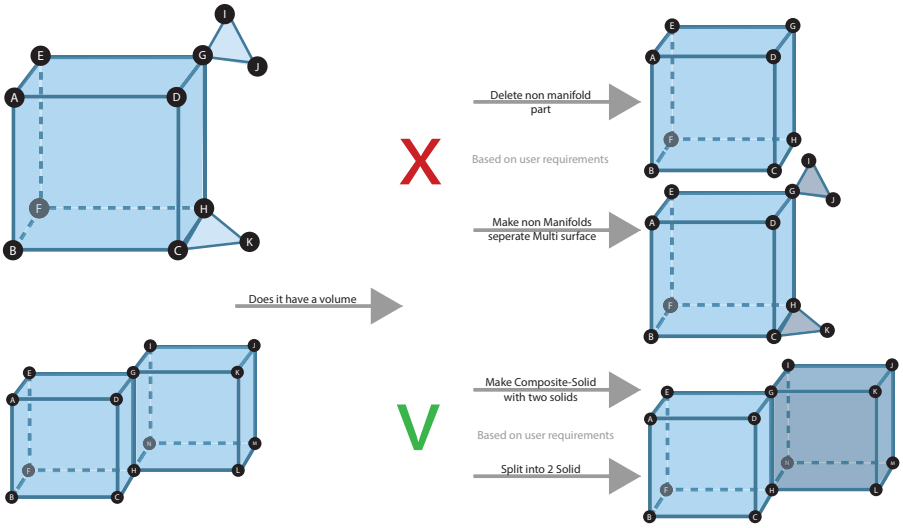


Figure 4.18.: Approach to repair: Non manifold case

305 - Multiple connected components - A shell needs to be closed, and a composite surface needs to consist of all connected surfaces. Therefore, there cannot be disconnected parts. The approach for this repair consists of the following steps:

- separate the connected components

- If the new connected component(s) doesn't have a volume (first example Figure 4.19)
  - For outer shells and CompositeSurfaces, if the user wants to keep all introduced geometries (Section 6.3), the new components are added as a separate MultiSurface. Otherwise, they are deleted.
  - For inner shells, the new components are deleted
- if the new connected components have a volume: (second example Figure 4.19)
  - For outer shells and CompositeSurfaces, if the user wants all geometries to be watertight (Section 6.3), the new component is added as part of a MultiSolid. Otherwise, the new component will be added as a separate solid
  - For inner shells, the shell will be split into 2 (or more) inner shells.

The polygons will all keep their original semantics and materials even if flipped.

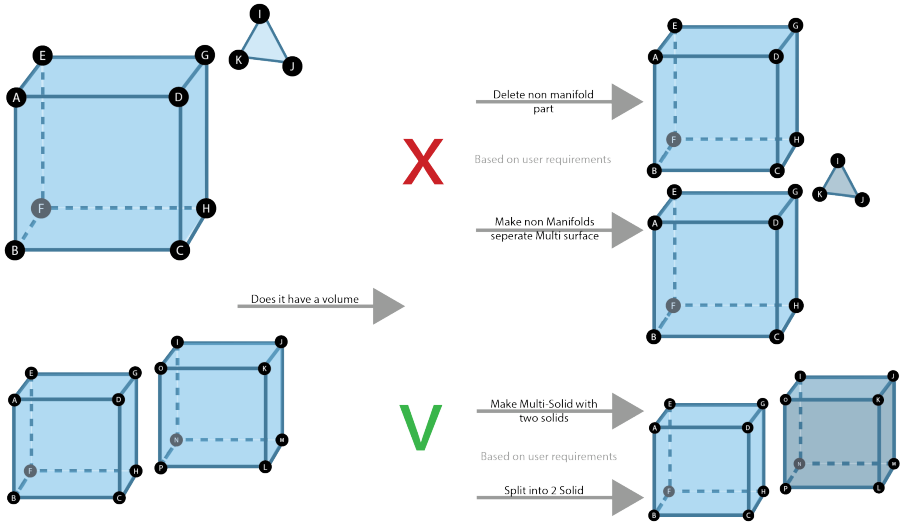


Figure 4.19.: Approach to repair: Multiple connected components

306 - Shell self-intersection - for Shells and CompositeSurfaces interior should not overlap. The local repair approach would be deleting the intersecting faces and patching the made holes using the methodology of 302. The semantics and materials for new polygons are the same as original polygons if a new polygon is the same or entirely inside an original polygon. When the local repair approach fails, alpha wrapping, as explained at 300, can be used (the second example of Figure 4.20), and the preservation of semantics and materials of 300 is used.

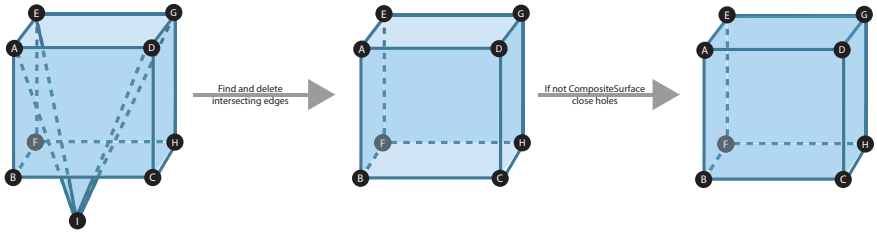


Figure 4.20.: Approach to repair: Shell self-intersection

307 - Polygon wrong orientation - A closed shell needs to be constructed in such a way that all polygons are counterclockwise for the outside, and CompositeSurfaces need to form a connected surface and, therefore, need the same orientation as its neighbors. To repair the wrong-oriented polygon needs to be flipped (Figure 4.21). The polygons will all keep their original semantics and materials even if they are flipped.

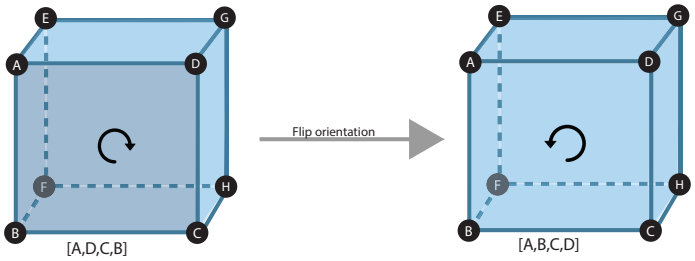
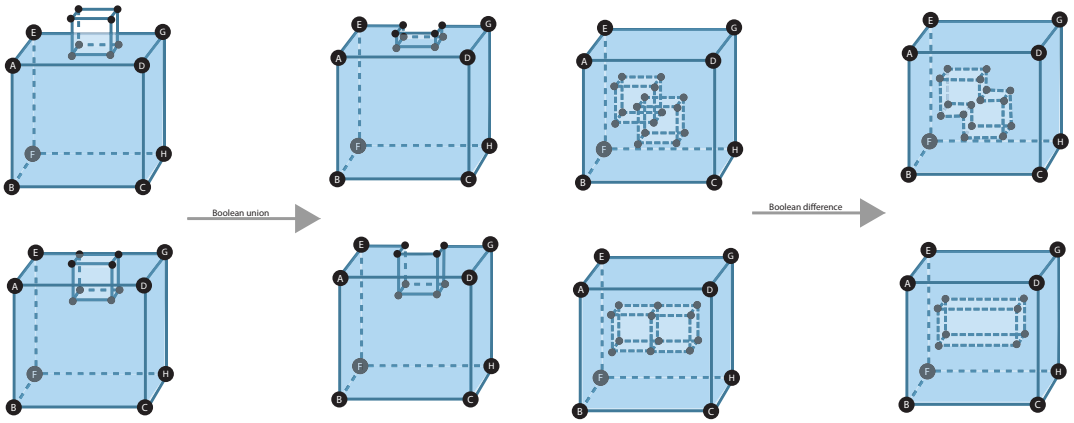


Figure 4.21.: Approach to repair: Polygon wrong orientation

4.5 Solid level repair approaches

As explained in Section 2.2.2, Solids are defined by 0 or more shells and have a few standards to be considered valid. The following subsections delve into non-compliant standards (based on the val3dity error codes Figure 2.17) and their corresponding repair approaches.

401 - intersection shells - In a solid, boundaries cannot cross, and the exterior of a solid with holes is not connected. When shells intersect in a solid, neither standard is complied with. This can happen in two situations, with inner shells intersecting with the exterior ring (Figure 4.22a) or with two inner shells intersecting (Figure 4.22b). This intersection could have (partly) the same edge or overlap over the boundary of another shell. Boolean operations are used to repair this. When an inner shell intersects with the exterior shell (Figure 4.22a), the inner shell needs to be subtracted from the exterior. Therefore, the difference is calculated. When two inner shells intersect, they need to become 1, and therefore, the union is calculated (Figure 4.22b). The semantics and materials are kept for all the polygons. When a polygon results from two merged polygons, it will get the semantics and materials of the largest, and when they are the same size, it will get the semantics and materials of the first.



(a) Intersection exterior and inner shell (b) Intersection two inner shells

Figure 4.22.: Approach to repair: Intersecting Shells

402 - Duplicate Shells - In a solid, boundaries cannot cross, so boundaries can also not be duplicates. This can happen when the same shell is used twice or when points are within the snap tolerance (`snap_tol` explained in Section 4.1). As a repair, only the first of the duplicates are kept (Figure 4.23). For duplicate inner shells, this doesn't change the geometry's volume or visual appearance, so the semantics and materials will stay the same. However, when an inner shell is a duplicate of the exterior, the inner shell will be deleted, changing the volume and the appearance. Also, the semantics and materials of this shell have been deleted.

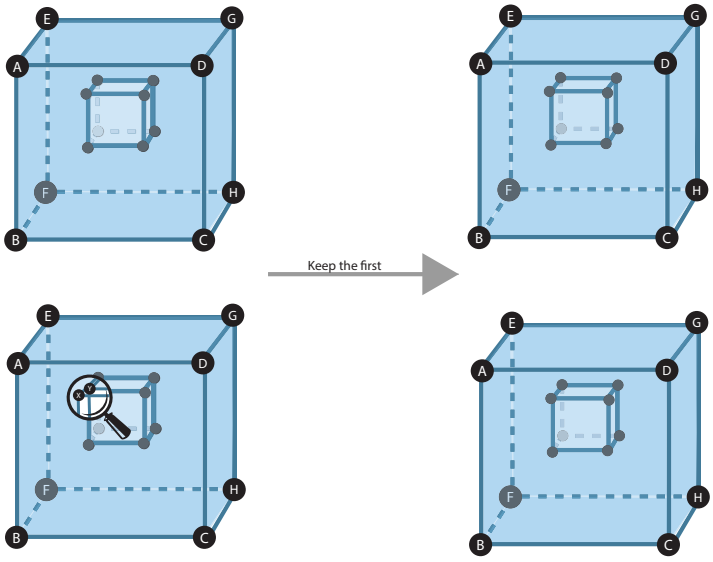


Figure 4.23.: Approach to repair: Duplicated Shells

403 - Inner shell Outside - The exterior of a solid with holes cannot be connected. Therefore, inner shells cannot be outside of the polygon. When an inner shell is outside, first check if the user wants to keep all introduced geometries (Section 6.3). If not, the inner shell, which is outside, is deleted together with its semantics and materials. Otherwise, the inner ring orientation is reversed and made into a solid (Figure 4.24). The semantics and materials will be kept even though the polygons are flipped. If the user requires all geometries to be watertight (Section 6.3), the new solid is added as a separate solid. Otherwise, the solid is added as part of a multi-solid, changing the type when the original type was solid or composite-solid.

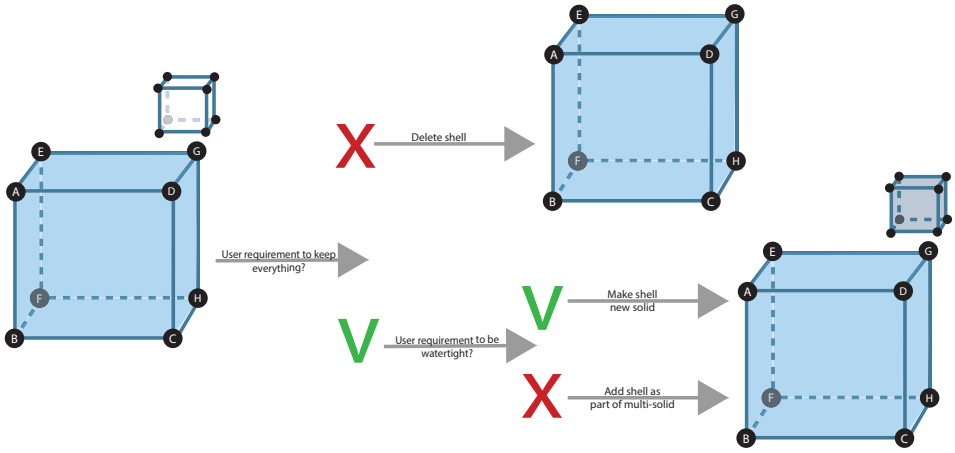


Figure 4.24.: Approach to repair: Inner Shell outside

404 - Solid interior disconnected - The interior of a solid needs to be a connected point set; therefore, if one or more holes disconnect the interior of the solid, it needs to be split into two (or more) solids (Figure 4.25). This can be done by subtracting the holes from the interior and looping over all the created (sub-)solids that are formed. The polygons of the new solids will keep their semantics and materials even if they are flipped. If the user requires all geometries to be watertight (Section 6.3), the new solid is added as a separate solid. Otherwise, the solid is added as part of a multi-solid, changing the type when the original type was solid or composite-solid.

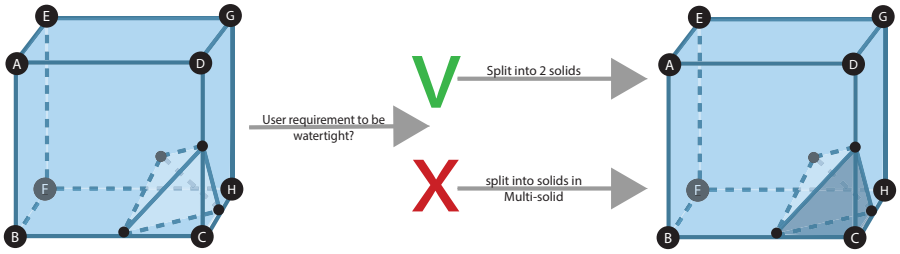


Figure 4.25.: Approach to repair: Solid interior disconnected

405 - Wrong orientation Shell - Boundary (shell) should be constructed of polygons oriented that when viewed from the exterior, the points are ordered counterclockwise (the normal must point to the exterior). To repair this, every shell needs to be checked for orientation, and when wrong, the orientation needs to be flipped by reversing all the polygons of the shell (Figure 4.26). The semantics and the materials are not changed when polygons are flipped.

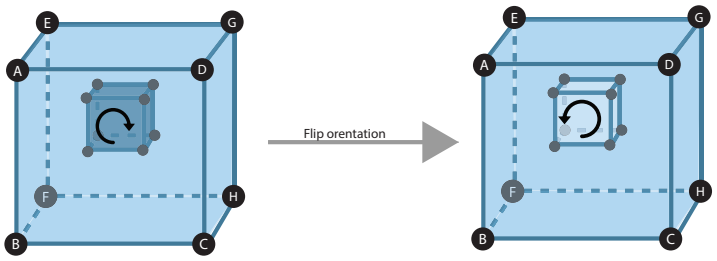


Figure 4.26.: Approach to repair: Wrong orientation shell

4.6 Solid interaction level repair approaches

As explained in Section 2.2.2, Composite solids are defined by 2 or more solids and need to be connected but not overlapping to be considered valid. The following subsections delve into non-compliant standards (based on the val3dity error codes Figure 2.17) and their corresponding repair approaches.

501 - Intersection Solids - Interior of a Composite-Solid may not overlap. If two more solids overlap, the overlap ratio is calculated by the overlap/first solid. This ratio is compared with the merge tolerance (Section 6.3). If the ratio is above the tolerance, a Boolean union makes the two solids into one (second and third example of Figure 4.27). The semantics and materials are kept; however, when a polygon is a result of two merged polygons, it will get the semantics and materials of the largest, and when they are the same size, it will get the semantics and materials of the first. Otherwise, the first solid stays the same, but the second solid is changed into the boolean difference (first example of Figure 4.27), where all polygons keep their original semantics and materials.

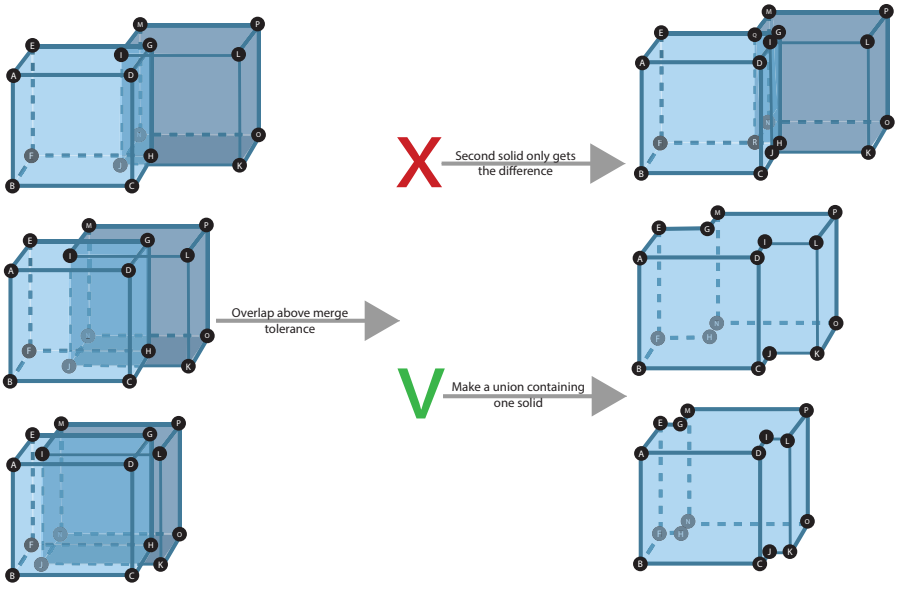


Figure 4.27.: Approach to repair: Intersection solids

502 - Duplicate Solids - The Interior of a CompositeSolid may not overlap, so solids cannot be duplicated. This can happen when the same solid is used twice or when points are within the snap tolerance (`snap_tol` explained in Section 4.1). As repair, only the first of the duplicates is kept (Figure 4.28), and the semantics and materials are kept the same for the remaining polygons.

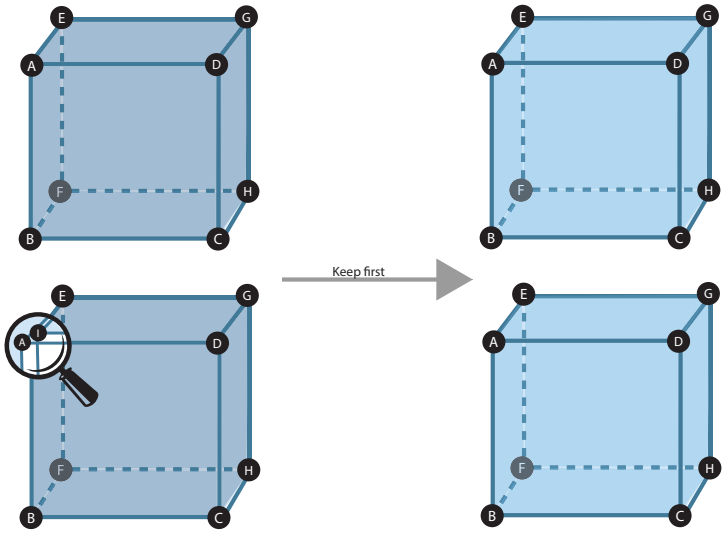


Figure 4.28.: Approach to repair: Duplicated solids

503 - Disconnected Solids - Interior of a Composite-Solid should be connected. If the user requires all geometries to be watertight (Section 6.3), the disconnected solid is added as a separate solid. Otherwise, the geometry type is changed to Multi-solid. The semantics and materials of the new geometries will be the same as the original.

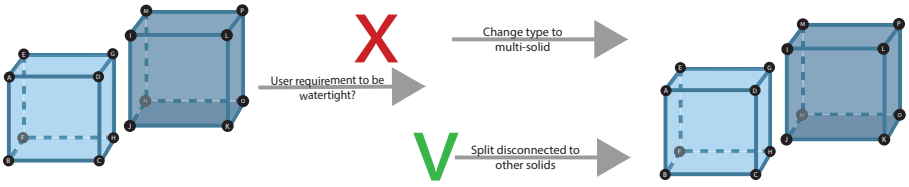


Figure 4.29.: Approach to repair: Disconnected solids

4.7 BuildingPart level repair approaches

The OGC states that `BuildingParts` are a physical or functional subdivision of a `Building` (Kolbe et al., 2021). This suggests that `BuildingParts` should not overlap, which can be validated by val3dity with the help of the overlap tolerance. As the OGC standards are not clear about `BuildingParts` being allowed to be disjoint, this is not validated nor repaired.

601 - BuildingParts overlap - is conceptually the same error as 501, which repairs the overlapping Interior of a CompositeSolid. Therefore, the same approach is used, except that if one or more `BuildingParts` are of geometry type Composite-Solid, the ratio of the overlap is calculated by the volume of overlap divided by the volume of the Composite-Solid instead of one of its solids. This ratio is also compared with the merge tolerance (Section 6.3). If the ratio is above the tolerance, the two `BuildingParts` are made into one by a Boolean union (Likewise to the second and third example of Figure 4.27). Otherwise, the first `BuildingPart` stays the same, but the second `BuildingPart` is changed into the boolean difference (Likewise to the first example of Figure 4.27). The semantics and materials will also be preserved in the same way as the repair approach for 501.

4.8 Global approach

If the proposed approaches per error will not solve the defects for unknown reasons, a global approach is needed. Users can stop the endless repair process of such defects with two parameters: `maxRepairDepth` and `TotalRepairDepth` (as explained in Section 6.2.2). When the repair tries to equal the depth of the parameters, the global repair will start, consisting of four stages (Figure 4.30); after each stage is checked to see if the geometry is valid. If it is still not valid, the next stage will be tried. While each stage is more robust, the geometric difference increases. The four stages are: Alpha wrap on polygons, Alpha wrap on vertices, Convex hull of geometry, and (oriented) bounding box.

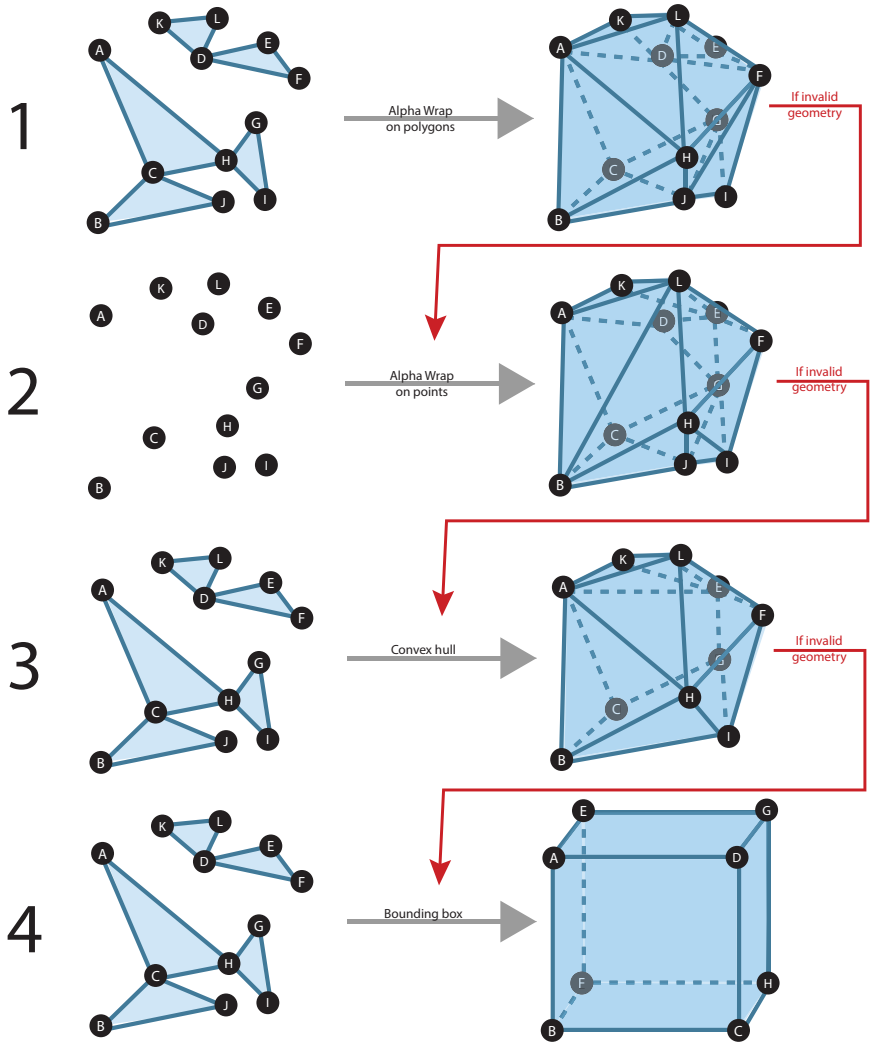


Figure 4.30.: Approach for global repair

5 Repairing 3D city models for specific applications

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 all applications. For example, Paden (2021) states that Computational Fluid Dynamics (CFD) has 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. This chapter consists of a broad overview of research for the 29 use cases described by Biljecki et al. (2015) (Section 5.1), followed by four sections focused on the specific use cases chosen for this thesis (Section 5.2, Section 5.3, Section 5.4 and Section 5.5).

5.1 Additional validity requirements for different use cases

The 29 distinct use cases of 3D city models described by Biljecki et al. (2015) have different additional requirements, summarized in Table A.1. Some additional requirements can be achieved by using the 3D model itself. Other needs to be done by collecting new and/or extra data. Since this thesis focuses on automatic repair, the ones not repairable are outside the scope and can be seen in *italic*. Two types of additional requirements remain, namely additional geometric requirements (in black) and additional semantics requirements (in blue).

Additional requirements could also help in deciding how to repair invalid geometries. For example, when a shell is not closed (error `302` and Figure 1.3a), changing a `Solid` into a `MultiSurface` would make it valid for the ISO requirements. However, when you want to use a 3D city model for Energy demand estimation, CFD, Volumetric density studies, or the 3D cadastre watertight `Solids` are needed (Table A.1). This results in, for example, capping the shell being a more logical repair solution for this use case than changing the geometry type.

5.2 Use case: Computational fluid dynamics (CFD)

Table 5.1 shows some of the validity issues and the importance of avoiding them in CFD (Paden, 2021). Although they are all ISO19107 requirements, not all are needed for every geometry type. Seeing non-watertight and intersections are only requirements for Solids and CompositeSolids. These are the only possible geometry types for CFD. Therefore, Buildings of type MultiSurfaces need to be converted into solids, and Buildings of type MultiSolid need to become CompositeSolid or separate nonintersecting Solids to check and repair these requirements.

Issue	Severity
Non-manifold	Severe
Non-watertight	Severe
Intersections	Severe
Duplicated outer surfaces	Severe
Wrong orientation	Moderate
Duplicated vertices	Moderate
Duplicated/missing inner surfaces	None

Table 5.1.: Validity issues and problems it creates in CFD, according to Paden (2021)

Additionally, Paden (2021) requires that buildings can not have small features, small edges, and also small gaps between buildings. The simplification method from Park et al. (2020) can be used for the small features and edges, which is explained in Section 3.3. Another method for simplification could be edge collapsing, which is also implemented in CGAL (Cacciola et al., 2023). The simplification algorithm uses a cost-driven half-edge-collapse operation, which involves removing an edge, a vertex, and two adjacent edges while optionally repositioning the remaining vertex. Cacciola et al. (2023) method employs two methods for the edge collapsing. The Lindstrom-Turk method performs "memoryless simplification" by selecting edges with the lowest collapse cost and computing replacement vertex positions based on shape, volume, and boundary constraints without comparing each step to the original mesh (Lindstrom and Turk, 2000). In contrast, the Garland-Heckbert method uses quadric matrices to encode distances to original mesh faces, selecting edges to collapse based on minimizing a quadratic error function, with enhancements for maintaining sharp borders and handling probabilistic geometry to tolerate noise (Heckbert and Garland, 1999). Figure 5.1 shows Sappho's Head model (leftmost, 34882 vertices) with different simplified outputs (1745 vertices) or the four Garland-Heckbert variations: plane (0.217912), probabilistic plane (0.256801), triangle (0.268872), and probabilistic triangle (0.490846).

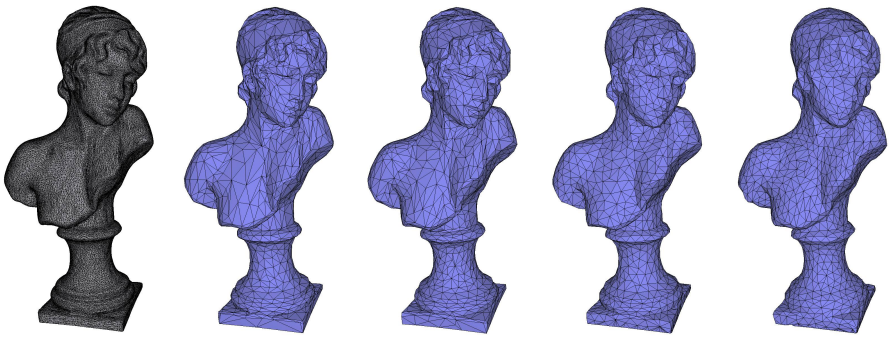
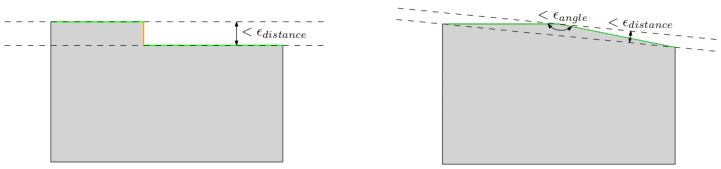


Figure 5.1.: Simplification option of "Sapho's head model" by the Garland-Heckbert method (taken from Cacciola et al. (2023))

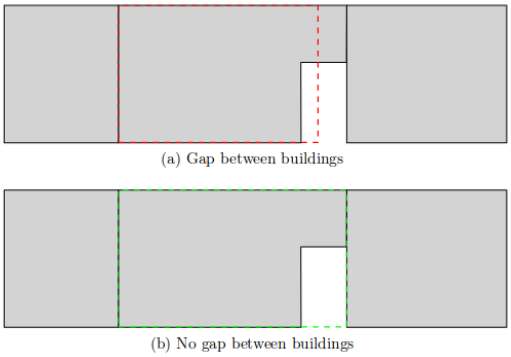
For the small gaps between buildings, Paden (2021) proposes generalizing building footprints. Commandeur (2012) created a half-space method where weighted lines (of the boundaries) decide the best fit. When parts of the boundary are parallel but within the threshold of the distance (Figure 5.2a) or when not parallel in the threshold of the angle (Figure 5.2b),

a weighted new generalized boundary will be formed. Instead of taking the two parts within the threshold as weight (Figure 5.3 a), you could also take surrounding footprints as weight for generalization (Figure 5.3 b).



(a) (b)

Figure 5.2.: Footprint generalisation thresholds (taken from Commandeur (2012))



(a) Gap between buildings

(b) No gap between buildings

Figure 5.3.: Footprint generalisation based on surroundings (taken from Commandeur (2012))

Lastly, Paden (2021) proposes that CFD input can not have small, sharp features and sliver triangles. Alliez et al. (2023c) proposes that mesh optimization could remove these with The ODT-smoother method. This method uses a perturber and exuder to target the worst mesh elements. The perturber adjusts vertex positions to eliminate slivers while the exuder re-weights mesh vertices to remove any remaining slivers.

5.3 Use case: Energy demand

According to Coors et al. (2020), the geometry of buildings influences energy demand. Assuming Equation 5.1, for example, the building's volume is used to estimate the internal area, which affects internal gains and the specific heat demand. Therefore, Sindram et al. (2016) proposes that building volume is essential and buildings should be watertight. In the ISO19107, being watertight is a requirement for Solids and CompositeSolids. Therefore, Buildings of type MultiSurfaces need to be converted into solids, and Buildings of type MultiSolid need to become CompositeSolid or separate nonintersecting Solids to check and repair these requirements.

$$\text{Heating} = \text{ventilation losses} + \text{conduction losses} - \text{solar gain} - \text{internal gains} \tag{5.1}$$

Equation 5.1 also prescribes that the polygon must be outwards oriented to calculate solar gain and the total area (elaborated more in Section 5.5). Lastly, the exterior surfaces are used to estimate ventilation and conduction losses. According to Willenborg et al. (2018), subdividing the exterior faces into semantic parts (roof surface, wall surface, and ground surface) could lead to better results, seeing component-related calculation could be used. Although Nouvel et al. (2017) argues that it doesn't make a significant change, the small impact allows a more realistic simulation. To validate and repair semantics values, the normal-based method explained in Section 3.4 could be used. Willenborg et al. (2018) also argues that knowledge of shared walls could influence the calculation, but seeing this information is not semantics in CityJSON. This is out of the scope of this thesis.

Lastly, seeing that Energy demand is calculated per building, it is essential that building(part)s do not intersect, seeing the volume is used twice. Figure 5.4 shows how repaired buildings using boolean operations on the geometry change the outcome of the energy demand. For buildingparts, merging could be a solution, but between different buildings, taking the difference is preferable, seeing the calculation is often address-based.

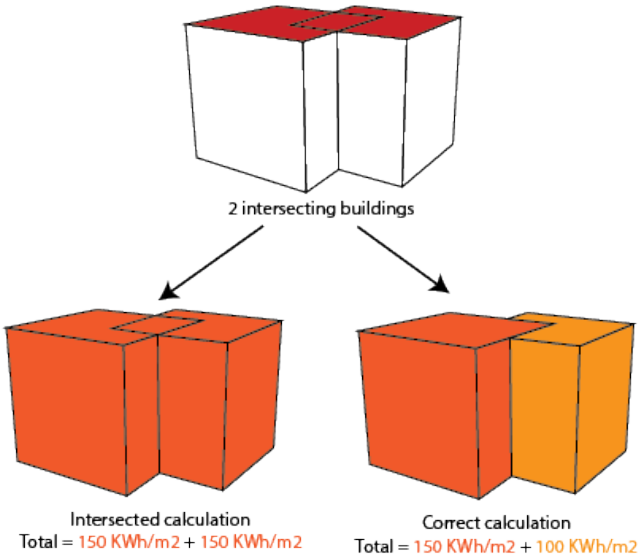


Figure 5.4.: Intersecting buildings and their Energy Demand estimation

5.4 Use case: Visualization

According to Ledoux (2017), duplicated surfaces and wrong orientation of surfaces have the most significant consequences for visualization purposes. Figure 5.5 shows how overlapping surfaces of different materials cannot be rendered. Therefore, it is important that the geometry of buildings are not only validated but also that different buildings are checked for intersections, and with boolean operations, overlapping surfaces will be prevented.

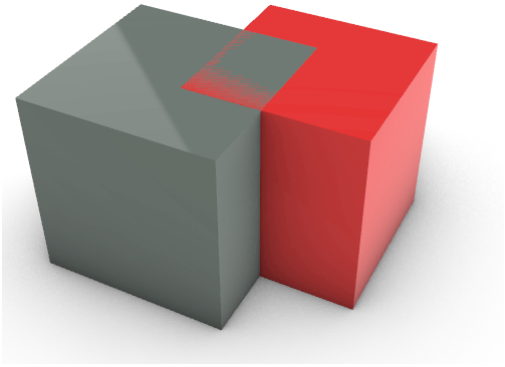


Figure 5.5.: Overlapping surfaces rendered

The wrong orientation of surfaces is a problem for visualization when culling is used for materials or textures. Culling prevents rendering surfaces facing away from the viewer, which optimizes the process (Zhang and Hoff, 1997). It uses the principle that objects are closed and only need to render the outside. When surfaces have the wrong orientation, the wrong side of the object is rendered, resulting in missing faces (Figure 5.6). When visualization programs use so-called double-sided shaders, this problem disappears. To repair this requirement, buildings of type MultiSurface must also be checked on orientation. Although making a MultiSurface into a solid of higher dimension is the only way to guarantee that the whole shell is oriented correctly, it also requires the shell to be closed. Taking Figure 5.6 as a reference, only some faces are oriented wrong; therefore, converting to a CompositeSurface can already solve this error, with the risk that the whole shell will be oriented incorrectly.

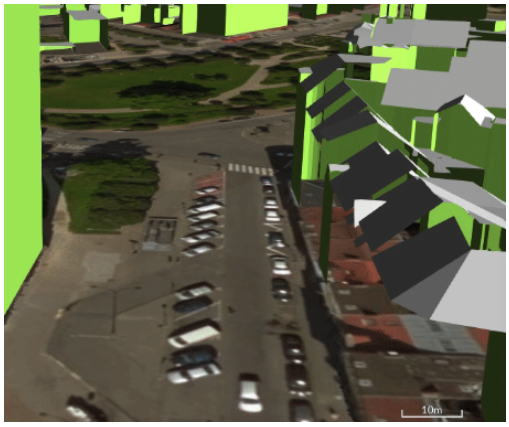


Figure 5.6.: Wrong oriented faces not rendered (taken from Ledoux (2017))

Lastly, Biljecki (2017) argues that a higher LOD enhances the visualization quality because it strongly benefits from more detail. Repairing for Visualisation asks for repairs as close to the original as possible; therefore, local repairs are favorable over global repairs.

5.5 Use case: Estimation of solar irradiation

Estimating solar irradiation aims to quantify the solar energy on surfaces, mainly for the potential for solar panels and heat calculations (Section 5.3). In the case of solar panel potential, seeing most solar panels are installed on roofs, the RoofSurface(s) extraction is favorable (Biljecki et al., 2016a). Therefore, the normal-based method explained in Section 3.4 could be used to validate and repair semantic roof face values. Ledoux (2017) also emphasizes the importance of orientation of the roofs with Figure 5.7. To repair this requirement, buildings of type MultiSurface must also be checked on orientation, which likewise to visualisation is done by converting in to CompositeSurface, with the risk that the whole shell will be oriented incorrectly.

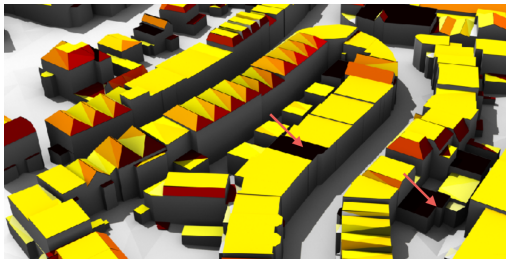


Figure 5.7.: Solar potential example within black roofs with no value (taken from Ledoux (2017))

For heat calculations, Biljecki et al. (2015) argues that the field extends towards vertical facades. Still, most calculations need the surfaces' materials and/or insulation, which are out of scope for this thesis. As explained in Section 5.3, shared wall knowledge could influence building heat calculation. However, when doing solar irradiation estimation on a 3D city model, this is irrelevant, seeing walls can be occluded by the other building when they share a wall. However, when buildings are apart, the occlusion of the shadow is significantly impacted by the LOD (Figure 5.8). Biljecki (2017) argues that although having the same area, their symmetric difference is considerable. Therefore, it could cast shadows on the wrong faces when detail is lost. Repairing for solar irradiation requires repairs as close to the original as possible; therefore, local repairs are favorable over global repairs.

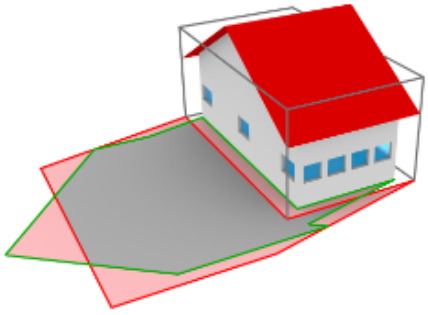


Figure 5.8.: Shadow cast by a detailed building vs a LOD1 block (taken from Biljecki (2017))

6 Implementation of AUTOr3pair

In this chapter, the implementation details of the methodology of Chapter 4 are described. Furthermore, the extra repairs and parameter tuning for Chapter 5 are discussed. The implemented code can be found on the AUTOr3pair repository

6.1 How to use AUTOr3pair

AUTOr3pair can be executed via the command line interface or within an Integrated Development Environment (IDE). Before running AUTOr3pair, you need to build the program using CMake. AUTOr3pair depends on CGAL, val3dity, and Nlohmann-json:

- CGAL needs to be installed. AUTOr3pair works with 5.5 (the version where Alpha wrap packages are introduced) and higher, and testing is done with 6.0. Using 6.0 is recommended, as it gives fewer segmentation errors when using Nef polyhedrons and surface meshes.
- Val3dity can be downloaded from their GitHub. AUTOr3pair depends on the "new" report structure, which has been implemented since 2.3.1, but the experiments (Chapter 7) have been done with version 2.5.1. Val3dity is dependent on CGAL, and also on Eigen (version used 3.4.0-4) and GEOS (version used 3.11.2)
- Nlohmann-json is included in the third-party directory and is on version 3.11.2.

Once AUTOr3pair is built, you can run it from the command line to repair 3D city models, with:

```
[location/]AUTOr3pair [3D city model to repair] [Use Case (file)] [LOD to repair]
```

Listing 6.1: Format for running

`[3D city model to repair]` is the path to the 3D city model you want to repair. AUTOr3pair supports CityJSON and OBJ file formats. Listing 6.2 shows how both file types can be run. How the input is handled is explained in Section 6.2.1.

```
[location/]AUTOr3pair 3DCityModel_file_to_repair.json
[location/]AUTOr3pair 3DCityModel_file_to_repair.obj
```

Listing 6.2: Run input files

`[optional: Use Case (file)]`: helps with setting parameters. predefined use case can be used, or users can define their composition of parameters by inputting a JSON. As shown in Listing 6.3 the use cases discussed in Chapter 5 can be called with a keyword (e.g., `CFD` for CFD, `ENERGYDEMAND` for Energy Demand, `VISUALISATION` for Visualization, `SOLARPOWER` for Solar Power estimation). All the Standards are explained in Section 6.3. The default repair settings are used if no use case (file) is provided.

```
[location/]AUTOr3pair [3D city model to repair] CFD
[location/]AUTOr3pair [3D city model to repair] ENERGYDEMAND
[location/]AUTOr3pair [3D city model to repair] VISUALIZATION
[location/]AUTOr3pair [3D city model to repair] SOLARPOWER
[location/]AUTOr3pair [3D city model to repair] User_preferences.json
```

Listing 6.3: Run Use case different use case parameters

`[optional: LOD]`: Can be used to specify a LOD to limit the repair process that LOD. To target a specific LOD, `-LOD[number]` needs to be added after the 3D city model, and in the case of the chosen use case behind the use case (Listing 6.4 shows an example where only geometries with LOD 2.1 will be repaired). Likewise to CJIO users should specify the complete LOD, so for example, 2 will not repair 2.0 or 2.1, the user should specify those.

```
[location/]AUTOr3pair [3D city model to repair] -LOD2.1
[location/]AUTOr3pair [3D city model to repair] [Use Case (file)] -LOD2.1
```

Listing 6.4: Run LOD as parameters

6.2 Program specifics

To test the proposed methodology of Chapter 4, AUTOr3pair was implemented in C++. C++ was chosen for its high performance and efficient memory management, which is needed for large 3D city models. Also, C++ object-oriented programming makes dealing with different kinds of geometries easier, making it easier to recreate, manipulate, and manage the objects. Lastly, the choice of C++ was influenced by the necessary software libraries for this project, which are the following:

- **val3dity** : can be used to validate 3D primitives against the ISO19107 Standards (as explained in Section 4.1). It can be used in C++ as a library to guide what to repair.
- **Nlohmann JSON** is also an open-source library that provides a simple interface for parsing, manipulating, and writing JSON. It is used for reading CityJSON and manipulating tu3djson during the repair process.
- **CGAL**: fully known as the Computational Geometry Algorithms Library, it is an open-source library that offers a wide range of robust geometric algorithms. AUTOr3pair uses many packages for reading, writing, and repairing objects. In the next sections, you can find which packages are used for what. The list below details a summarization of the packages used and their purposes:
  - **Geometric Objects**: Geometries are translated into Lists of points, Polyhedron, Nef_polyhedron and surface meshes. Depending on the libraries used, the exact kernel (`Exact_predicates_exact_constructions_kernel.h`)
    or the inexact kernel(`Exact_predicates_inexact_constructions_kernel.h`)
    is used to define precision.
  - **Manipulating geometries**: Depending on the repair libraries are used to give info about or manipulate the geometries, these are:
    - **Polygon Mesh Processing**: This set of libraries provides various tools for manipulating polygon meshes, which are collections of polygons that define the surface of 3D models.

        - **Surface mesh simplification**: is used for reducing the complexity of meshes while maintaining their overall shape.
        - **Boost and graph utilities** are used for converting Nef polyhedra to polygon meshes and performing Euler operations on polygonal surfaces.
        - **Delaunay triangulation** is used for triangulating non-planar faces.
        - **Alpha wrap** for generating alpha shapes in 3D.
        - **Kd tree** utilities and Orthogonal neighbor search for efficient searching and neighbor when using the Vertices list.
        - **Convex hull**: is used for computing convex hulls.
        - **Linear least squares fitting**: is used for fitting the best plane for nonplanar face computations.
    - **OBJ File I/O**: is used for reading from and writing to OBJ files, enabling the import and export of 3D models in the OBJ format.

6.2.1 Input

AUTOr3pair can repair 3D city models stored as CityJSON or OBJ. As explained in Section 2.3.1, CityJSON can store semantic 3D city models as JSON, which can be parsed with the help of Nlohmann JSON. It consists of at least five properties. However, only three are needed for the repair process: transform, CityObjects, and vertices. The other properties will be stored and copied to the output data, seeing they won't be modified. The key Transform will only be used to find the decompress factor for the tolerance (discussed more in Section 4.1), but does not change and, therefore, will also be 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; for the repair, new points may need to be added, or vertices may be deleted in the post-processing step. The CityObjects are converted into tu3djson to be repaired individually. As explained in Section 2.3.3 the properties differ slightly from those used in CityJSON. Figure 6.1 explains how a CityJSON CityObject is converted in a tu3djson feature. The GeometryObject gives the base for the feature. Boundary and type of the geometry are directly copied into Geometry alongside the (transformed) vertices. All the semantic properties of the GeometryObject are copied to the properties so they can be changed during the repair process. After the repair process, an attribute AUTOr3paired to the attributes and the Geographical extent (and possibly LOD) can be changed in the post-processing (Section 6.2.3).

As stated in Section 1.4, this thesis focuses on repairing `Building` and its sub-parts. During the repair process, each City Object is checked to see if the type should be repaired. Users can extend the types begin repaired with the parameter `ExtendScope` by adding those types to the list. However, the correctness of these repairs is not guaranteed. Users can also opt for repairing only one LOD during the repair process. This can be done by stating the LOD as one of the inputs. This is so that datasets, such as the 3DBAG (Section 7.3.1), can repaired with other repair parameters per LOD.

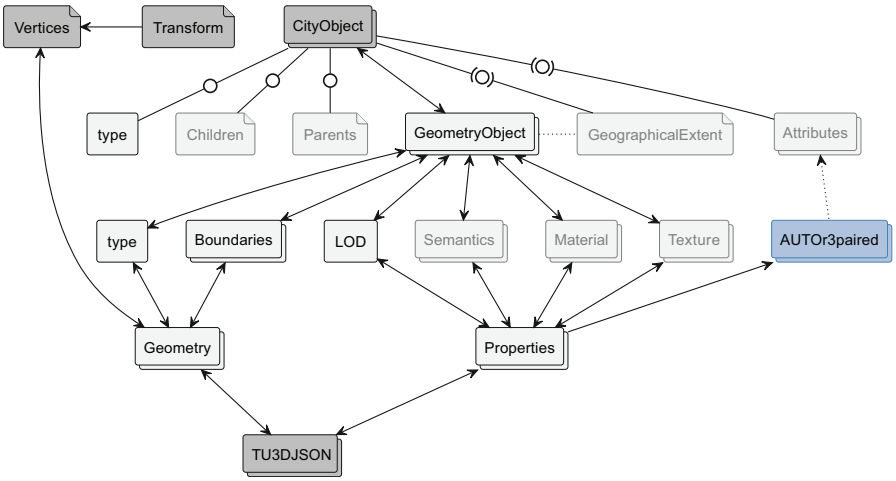


Figure 6.1.: Flowchart of translating CityObject to TU3dJSON feature

For the use case CFD also, 3D city models storage as OBJ can be repaired. As explained in Section 2.3.2 does not contain semantics or metadata, but it could have materials or groups, however for simplification they are now discarded. To parse the OBJ CGAL's I/O stream package is used (Fabri et al., 2023), which outputs the vertices and polygons in arrays. With CGAL's Polygon Mesh Processing package (Loriot et al., 2023), the faces are grouped into connected components. CGAL also takes care of conversion between OBJ's 1-based boundaries, which are converted into 0-based boundaries. Figure 6.2 illustrates how the vertices and faces are translated through a CityObject "like" JSON array into tu3djson features on which the repair process is done. The type of CityObject is always building, so it is always in the scope of the repair process. The user standard *OBJ_geomtype* tells what kind of geometry primitive the faces form. The default value is Solid, but users could change it to other primitives (Section 6.3). The value is set according to the type of GeometryObject, and the boundary depth is set accordingly. Based on the *OBJ_geomtype* also, parameters `Watertight` and `orientation` are changed accordingly to the type, so no lower order geometry primitives will be outputted. For example, when the type is Solid, repairs can not be done by adding Mult-/CompositeSurfaces.

CGAL's Polygon Mesh Processing package (Loriot et al., 2023) automatically deletes faces with too few points (val3dity error 101) or faces with duplicate points (with the same index, same point but different index is okay) behind each-other (val3dity error 102). Therefore, both errors are repaired before finding the connected components by deleting the faces (with error 101) or deleting the duplicate point (error 102).

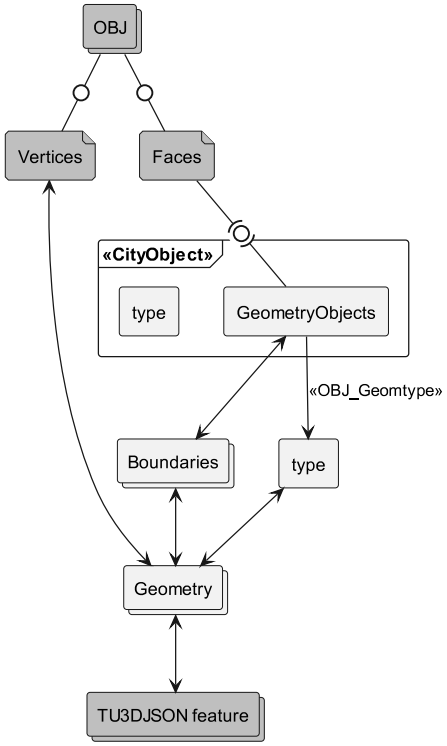


Figure 6.2.: Flowchart of translating OBJ to TU3dJSON features

6.2.2 Repair framework

As shown in Figure 2.16, the repair method consists of two parts, which will be discussed in the following subsections: the geometric repairs (as explained in Chapter 4) and the extra use case repairs (as described in Chapter 5). An object is only considered fully repaired if all (user) requirements are fulfilled. However, two parameters can stop the repair process when stuck in an infinite loop (for example Figure 6.3) or if the repair method cannot repair the geometry. The default value of `maxRepairDepth` is 50 and is used for how many times repairs on the same level can be done (e.g.Ring level or Polygon level). On the otherhand `TotalRepairDepth` has a default value of 500 and is the maximum repairs on one object. The `maxRepairDepth` is used when repairs (for unknown reason) do not work on an object, while `TotalRepairDepth` can be used when repairs form a loop, for example Figure 6.3 describes how deleting duplicates points (when being in the `snap_tol`) in a triangle leads to an infinite loop. The chosen implementations of the repairs (as explained in the following subsections) prevent this kind of loop (for my tested data (Section 7.1)). However, to make the code more robust for unknown new loops, these parameters are helpful (Algorithm 1 (Appendix C) shows how both are used in the Geometric Repair).

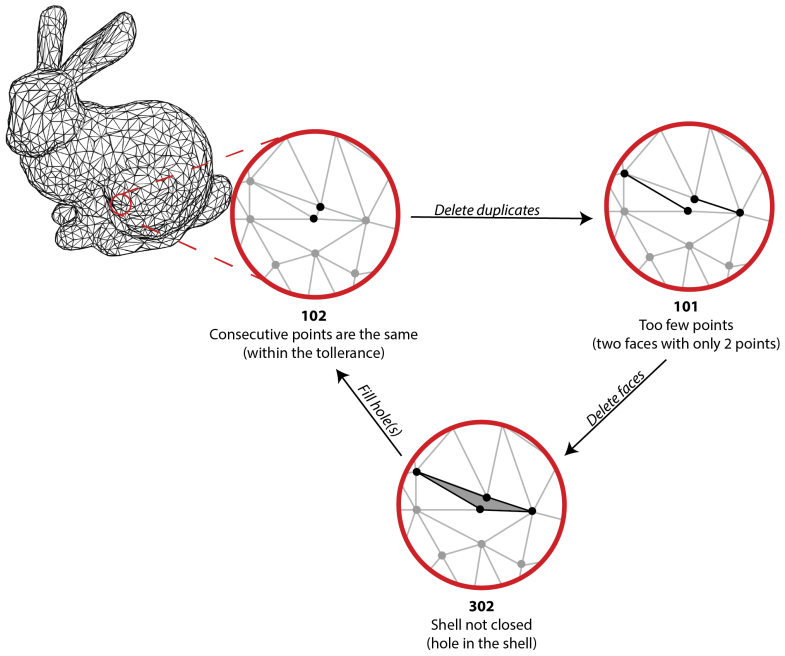


Figure 6.3.: Loop of errors while dealing with a sliver

The `ShowProgress` parameter can be used to allow users to follow the process in real-time. This enables users to see which items have been validated and which repairs have been completed, making it easier to monitor the progress of AUTOr3pair directly in the terminal. Meanwhile, the `Debugging` parameter provides users with more detailed insights when needed. It generates extensive output, including the vertices list, which is particularly useful for debugging or for users who wish to examine specific steps of the program in greater detail.

Geometric repairs

As explained in Section 4.1, val3dity could be used to validate geometries and find errors; with the known errors, the specific repairs could be called. To implement this a Geometry Class is used, which can save the tu3djson, validate it with val3dity and send the part that needs repair into the correct repair function. Seeing different kinds of geometry primitives have different boundary depths (Figure 2.15), the geometry class is inherited by the specific type (shown in Figure C.1 (Appendix C)). The pipeline for the repair process is presented in Algorithm 1 (Appendix C). It uses the val3dity report summarized as a map (Algorithm 2 (Appendix C)) to order all the repairs per category. For validation, the four standards (i.e.Tollerances) are used for the validation (Algorithm 2 of Algorithm 2) and can be changed according to user preference (Section 6.3). To give the user options on what to do only some repairs, two parameters (`solveAll` and `ErrorsToRepair`) are added (Line 10 of Algorithm 2). The default value of `solveAll` is True, but when `ErrorsToRepair` gets input, the value is changed to False. If a user wants to repair only specific errors, he can add the error code to the `ErrorsToRepair` array (the default value is an empty array). This repair will not be added to the ordered map if an error is not in the array (and `solveAll` is False).

The repairs on geometry are done in hierarchical order. It is possible to do multiple repairs in the same category at the same time (e.g.repair multiple ring errors). However, the geometry is validated again after all the repairs in the same category instead of doing the higher-order repairs. Seeing changes could influence other repairs. Some repairs can split the geometry into multiple parts (Line 33 of Algorithm 1) or change the type (Line 36 of Algorithm 1). If this happens, the geometry repair process is stopped (also when not valid yet) and reinitialized as multiple or other type Geometry repairs.

Implementation of ring level

To speed up the repair process for use cases where detail is not a main requirement, the parameter `SkipLowRepairs` can be used. The default value is False, but when set to True, it deletes the whole polygon instead of doing the local repairs on the rings. This is only done for Solid and higher order primitives; seeing these deleted faces will be re-added in the shell repairs. When a polygon is deleted, the semantic and material values are deleted from their arrays. Still, the values and original polygons are kept for re-assigning semantics and material values in later repairs. The following subsections explain the implementation of the methodologies proposed in Section 4.2.

101 - Too few points - val3dity gives in the `id` on which face this Ring error appears. To delete the incorrect rings, all rings must be checked. If the outer boundary had less than 3 points, the whole polygon is deleted; otherwise, only the hole is deleted. When a polygon is deleted, the semantic and material values are deleted from their arrays.

102 - Consecutive points are the same - val3dity gives in the `id` on which face this Ring error appears. To find the consecutive points, all rings are checked. The index is the same when two consecutive points are the same cause. The first is kept. However, when two consecutive points are the same cause their distance is in the `snap_tol`, the second point is snapped to the location of the first point. This is done by changing the coordinate of the second point in the vertices list and deleting the index (So other polygons using this vertex are also changed. Although both follow the same approach of keeping the first vertex, the different approach follows from Figure 6.3.

104 - Ring self-intersection - val3dity gives in the `id` on which face this Ring error appears. To test which ring(s) self-intersect, each ring is projected to a 2DCGAL polygon (Giezeman and Wesselink, 2023), projecting onto the best plane calculated by linear least squares fitting of CGAL (Alliez et al., 2023d); this package can check if a polygon is simple. If the ring is not simple, CGALs convex hull algorithm is used (Hert and Schirra, 2023a). When the ring is collapsed into a line, the ring is deleted; when the ring is deleted is the outer boundary, the whole polygon is deleted; otherwise, only the hole is deleted. When a polygon is deleted, the semantic and material values are deleted from their arrays.

Implementation of polygon level

Likewise, to ring level, the parameter `SkipLowRepairs` can speed up the process by deleting the polygon instead of doing the local repair per polygon. When a polygon is deleted, the semantic and material values are deleted from their arrays. Still, the values and original polygons are kept for re-assigning semantics and material values in later repairs. The following subsections explain the implementation of the methodologies proposed in Section 4.3. Although polygons are 2d objects, all repairs are done in 3d space to avoid floating point changes in the vertices. This is more robust for the output, but it risks that, for example, boolean operations will not work, seeing rings can be on slightly different planes.

201 - Intersecting rings - val3dity gives in the `id` on which face this polygon error appears. To find the intersecting rings, all the rings are translated to (separate) CGAL Nef polyhedra (Hachenberger and Kettner, 2023). Then, using the SetDiff paradigm (Section 3.1), the polygon is made by subtracting the inner rings from the exterior ring:

$$Polygon = Ring_0 \setminus (Ring_1 \cup Ring_2 \cup \ldots \cup Ring_n)$$

This changes the outer boundary when it intersects with a hole and unions overlapping holes. To ensure the repaired polygon is normal in the same direction as the original, the angle between the normals is calculated. When the normals are opposed, the face is flipped by reversing all rings.

202 - Duplicate rings - val3dity recognizes this error as 201, so implementation of 201 is used.

203 - Non planar polygon distance - val3dity gives in the `id` on which face this polygon error appears and in the `info` the distance to the fitted plane. If the distance exceeds twice the `planarity_d2p_tol`, the face is triangulated; otherwise, outlier points are projected onto the plane of the correct points. Triangulation (Loriot et al., 2023) is done by using a CGAL surface mesh and when a projection is needed, a best-fitted plane is calculated, and outlier points are adjusted accordingly.

204 - Non planar polygon deviation - val3dity gives in the `id` on which face this polygon error appears. The face is made into CGAL Delaunay triangulation (Jamin et al., 2023) to find the point which makes a fold in the polygon. For each pair of triangles, the angle between the normals is calculated. The plane's vertices are counted when the angle exceeds the `planarity_n_tol`. The three vertices with the lowest count form the new plane, and all other points are projected onto the new plane by changing the vertex coordinate.

205 - Interior Disconnected - val3dity provides the `id` of the face with this error. To find the disconnected part(s), all rings of the surface are translated to (separate) CGAL Nef polyhedra (Hachenberger and Kettner, 2023). Then, using the SetDiff paradigm (Section 3.1), the polygons are made by subtracting the inner rings from the exterior ring:

$$Polygon_1, Polygon_2 \ldots Polygon_n = Ring_0 \setminus (Ring_1 \cup Ring_2 \cup \ldots \cup Ring_n)$$

Seeing one (or more) inner ring split the exterior, more than one polygon can be retrieved. Calculating the angles ensures correct orientation, likewise to 201. All the polygons get the value of the original polygon for semantics and material.

206 - Inner Ring Outside - val3dity gives the `id` of the face with this error. The outer boundary is converted into a 2d CGAL polygon (Giezeman and Wesselink, 2023), projecting onto the best plane calculated by linear least squares fitting of CGAL (Alliez et al., 2023d). Using the even-odd paradigm (Section 3.1) for each inner ring is checked if the first point is within the exterior ring. When this is not true, parameter `KeepEverything` (default value False) decides what happens. When False, the inner ring is deleted. When True, the ring's orientation is reversed and added as a new polygon.

207 - Inner Ring Nested - val3dity provides the `id` of the face with this error. All inner rings are converted into a 2d CGAL polygon (Giezeman and Wesselink (2023)), projecting onto the best plane calculated by linear least squares fitting of CGAL (Alliez et al., 2023d). Using the even-odd paradigm (Section 3.1) for each inner ring is checked if the first point is within another inner

ring. When this is not true, parameter `KeepEverything` (default value False) decides what happens. When False, the inner ring is deleted. When True, the ring's orientation is reversed and added as a new polygon.

208 - Orientation Rings Same - val3dity gives the `id` of the face with this error. The normal of all rings is calculated, and each inner ring normal is used to calculate the angle with the outer ring normal. If they are oriented the same the ring gets flipped by reversing the ring.

Implementation of shell level

The following subsections explain the implementation of the methodologies proposed in Section 4.4.

300 - Non valid 2-manifold - val3dity provides the `id` of the shell with this error. All polygons of the shell are converted into a polygon soup, which is converted into as CGAL surface mesh (Botsch et al., 2023), with the help of CGALs Polygon mesh Processing (Loriot et al., 2023). On this surface, Alpha wrap from CGAL is used (Alliez et al., 2023a), wrapping it with alpha 1.3 and offset 0.3. The selection of the Alpha and offset values was made following the recommendations provided by my supervisor. The repaired mesh is then converted back into a shell format by extracting the faces and corresponding vertices. The semantics and materials are assigned as explained in Section 6.2.2.

301 - Too Few Polygons - val3dity provides the `id` of the shell with this error. First, it is checked if all vertices of the shell are on the same plane. If this is the case, the shell is based on the user requirement `watertight`, and the shell is deleted (if true) or changed into a CompositeSurface (if false). If the shell is not on the same plane, the method from 302 is used. Likewise to 302, when the holes/gaps cannot be patched, alpha wrap from CGAL is used (Alliez et al., 2023a) as explained in 300, Non-valid 2-manifold. The semantics and materials are reassigned as explained in Section 6.2.2, and the filled holes will get the Null value.

302 - Shell Not Closed - val3dity provides the `id` of the shell with this error. The shell is converted to a CGAL triangulated surface mesh (Botsch et al., 2023). then, it iterates over all the half edges to find if it is a boundary edge. If a boundary edge is found, the hole (of which the boundary is) is triangulated with the help of acCGALs Polygon mesh Processing (Loriot et al., 2023). Afterward, the repaired mesh is converted back into a shell format by extracting the faces and corresponding vertices. When no boundary edge and no hole is found, alpha wrap from CGAL is used (Alliez et al., 2023a) as explained in 300, Non-valid 2-manifold. The semantics and materials are reassigned as explained in Section 6.2.2, and the filled holes will get the Null value.

This implementation has some preconditions that cannot always be met; most importantly, the border edges of the hole must not intersect, and shells should be manifold, but seeing those (303 and 306) are checked later than this error/repair, this sometimes results in needing a global repair (further discussed in Section 7.4.4). Also, holes smaller than 10E-3 cannot be repaired with this CGAL function. However, the error will keep showing up, which also results in the need for a global repair.

303 - Non-manifold Shell - val3dity gives the `id` of the shell and the index of the polygon with this error. First, it is checked if there is a face "double-sided" (two faces that are the same but normal in different directions); if that is true, that face is left out. If that is not the case, all the half edges are mapped to determine what kind of non-manifold problem there is. If none of the half edges is used more than once, the problem is a non-manifold vertex. The repair is done by region-growing shells over edges. Which "cuts" shells on their non-manifold vertices. If there are overused half edges, the same region growing of shells is done, but shells cannot

grow over the overused edges. When all shells are formed, it is tested if, by flipping, shells can be combined. This implementation cannot solve unrepairable nonmanifolds as discussed in Section 7.4.2, which results in those cases needing global repairs.

305 - Disconnected Components - val3dity provides the `id` of the shell with this error, which is then converted into a CGAL triangulated surface mesh (Botsch et al., 2023). Using CGAL's connected components algorithm (Loriot et al., 2023), distinct shell regions are identified, and each face is assigned to a connected component. Components are sorted by size, with the largest prioritized. The function then checks if each component encloses a valid volume using CGAL's volume calculation (Loriot et al., 2023). Components with a volume, or if `KeepEverything` is true, are retained. Depending on whether the shell is inner or outer and the `watertight` parameter, extra components are added as separate geometries or inner shells.

306 - Shell self intersection - val3dity provides the `id` of the shell with this error, , which is then converted into a CGAL triangulated surface mesh (Botsch et al., 2023). CGAL's self-intersection algorithm (Loriot et al., 2023) detects which faces self-intersect and those faces are removed. The holes created by the removed faces are filled using the repair method of 302, Shell Not Closed.

307 - Polygon Wrong Orientation - val3dity gives the `id` of the shell and the index of the polygon with this error. The wrongly oriented polygon is flipped by reversing the rings. For semantics and material, the value of the polygon will be reused after it has been flipped.

Implementation of solid level

The following subsections explain the implementation of the methodologies proposed in Section 4.5.

401 - Intersection Shells - val3dity provides the `id` of the solid with this error and the `info` provides which shells intersect. All shells in the solid are converted to CGAL Nef polyhedra (Hachenberger and Kettner (2023)). Then, using the SetDiff paradigm (Section 3.1), the Solid is made by subtracting the inner shells from the exterior shell:

$$Solid = Shell_0 \setminus (Shell_1 \cup Shell_2 \cup \ldots \cup Shell_n)$$

This changes the outer shell when it intersects with an inner shell and unions overlapping inner shells. The original semantics and materials of the polygons will be reassigned to the new (partly subtracted) polygons.

402 - Duplicated Shells - val3dity recognizes this error as 201, so implementation of 201 is used.

403 - Inner Shell Outside - val3dity provides the `id` of the solid with this error and the `info` provides which shell is outside. `KeepEverything` (default value False) decides what happens with the incorrect shell. When False, the inner shell is deleted. When True, the shell's orientation is reversed, and `Watertight` (default value False) decides how it is added. When the True the inner shell is added as a new solid, if false the geometry type is changed to MultiSolid (or stays MultiSolid) and the shell is added as a solid of the MultiSolid. For semantics and material, the value of the shell when it was an inner shell is reused for the new solid.

404 - Solid Interior Disconnected - val3dity provides the `id` of the solid with this error. To find the disconnected part(s), all shells are translated to (separate) CGAL Nef polyhedra

(Hachenberger and Kettner (2023)). Then, using the SetDiff paradigm (Section 3.1), the Solid is made by subtracting the inner shells from the exterior shell:

$$Solid_1, Solid_2 \dots Solid_n = Shell_0 \setminus (Shell_1 \cup Shell_2 \cup \dots \cup Shell_n)$$

The inner shells are subtracted from the outer boundary. The volumes formed are retrieved and `Watertight` (default value False) decides how it is added. When True the parts are split into multiple solids, otherwise the parts become solids in a MultiSolid (and likewise to 403 the geometry type is changed to/stays MultiSolid). The original semantics and materials of the polygons will be reassigned to the polygons of the solids as explained in Section 6.2.2.

405 - Wrong Orientation Shell - val3dity provides the `id` of the solid with this error and the `info` provides which shell is wrongly oriented. All the polygons of the wrongly oriented shell are flipped by reversing the rings. For semantics and material, the value of the polygons will be reused after they are flipped.

Implementation of solid Interaction level

The following subsections explain the implementation of the methodologies proposed in Section 4.6.

501 - Intersection Solids - val3dity provides the `id` of the geometry with this error and the `info` provides which solids intersect. The two solids are converted to CGAL Nef polyhedra (Hachenberger and Kettner (2023)). If an intersection has a volume above `MergeTol` (default value 0.1 (10% of the first encountered volume)), the solids are merged:

$$Solid_0 = Solid_0 \cup Solid_1$$

Otherwise, the intersecting volume is subtracted from the second solid:

$$Solid_0 = Solid_0 \;\; \& \;\; Solid_1 = Solid_1 \setminus Solid_0$$

The original semantics and materials of the polygons will be reassigned as explained in Section 6.2.2.

502 - Duplicated Solids - val3dity provides the `id` of the geometry with this error and the `info` provides which solids are duplicates. The first solid is kept, and the second is deleted from the geometry.

503 - Disconnected Solids - val3dity provides the `id` of the geometry with this error. All solids are converted to CGAL Nef polyhedra (Hachenberger and Kettner (2023)). A connectivity graph represents the connections between solids based on their intersections. The graph is then used to find connected components using a breadth-first search. Each connected component is grouped together into a new geometry, being a MultiSolid when consisting of multiple solids or otherwise a Solid. The original semantics and materials of the polygons stay the same.

Implementation of BuildingPart level

The following subsections explain the implementation of the methodologies proposed in Section 4.7.

601 - Intersecting building parts - val3dity provides the `id` of the id and geometry which intersect. The two geometries are converted to CGAL Nef polyhedra (Hachenberger and Kettner (2023)). If an intersection has a volume above `MergeTol` (default value 0.1 (10% of the first encountered volume)), the geometries are merged:

$$Geometry_0 = Geometry_0 \cup Geometry_1$$

Otherwise, the intersecting volume is subtracted from the second solid:

$$Geometry_0 = Geometry_0 \;\; \& \;\; Geometry_1 = Geometry_1 \setminus Geometry_0$$

The original semantics and materials of the polygons will be reassigned to the new (partly subtracted) polygons as explained in Section 6.2.2.

Implementation when other repairs fail

When the repair depth is reached (`maxRepairDepth` and `TotalRepairDepth`), local repairs do not have the desired effect. Therefore, a global approach is used, consisting of 4 steps, becoming more radical with each step. The steps are done per shell to keep as much detail as possible. After each step, the validity is tested; when invalid, the next step is done. Likewise to the local repairs having codes for their specific error, the global repairs did get the codes 1000 t/m 1003, to be easily findable in the repair report (Section 6.2.4)

1. **1000** - CGAL's Alpha wrap on the polygon soup based on the original boundary (Alliez et al., 2023a). Similarly, to 300, the alpha and offset values used are 1.3 and 0.3.
2. **1001** - CGAL's Alpha wrap on the original vertices (Alliez et al., 2023a). Similarly, to 300, the alpha and offset values used are 1.3 and 0.3.
3. **1002** - CGAL's Convex Hull Algorithm (Hert and Schirra, 2023b) is used to compute the smallest convex polyhedron that encloses the set of points from the geometry.
4. **1003** - Making a bounding box around the shell always ensures geometric validity. When a shell is one plane instead of a bounding box, a bounding plane is returned. CGAL's optimal bounding box (Katrioplas and Rouxel-Labbé, 2023) is used.

Preserving semantics and materials

To preserve semantics and material, their link to faces needs to be maintained. Therefore, an unordered map containing all the surfaces is used to manage the semantics and materials of each surface in the 3D city model. At the start of the repair per geometry, information about semantics and materials is retrieved per surface and stored in the map (lower part of Figure 6.4). After making changes or repairs to the geometry, modified faces need to be re-associated with semantics and material information. The four methods to do this are as follows:

- When a face is split into multiple parts, all parts get the original semantics and materials.
- When a face is flipped, it will retrieve the same semantics as the original.

- When a face as overlap with one original face it will get the same semantics and materials as the overlapping face.
- When a face is merged from multiple faces or overlaps with multiple, the biggest overlapping surface semantics and materials will be used. When multiple have precisely the same size, the semantics and materials of the first found surface will be used (seeing the faces are checked in adding order).

When no value can be assigned using these methods, the value `null` is used for the semantics and materials. When the tu3djson is converted back to the CityObject per surface, the semantics and materials are reassigned by writing their value to their corresponding JSON array.

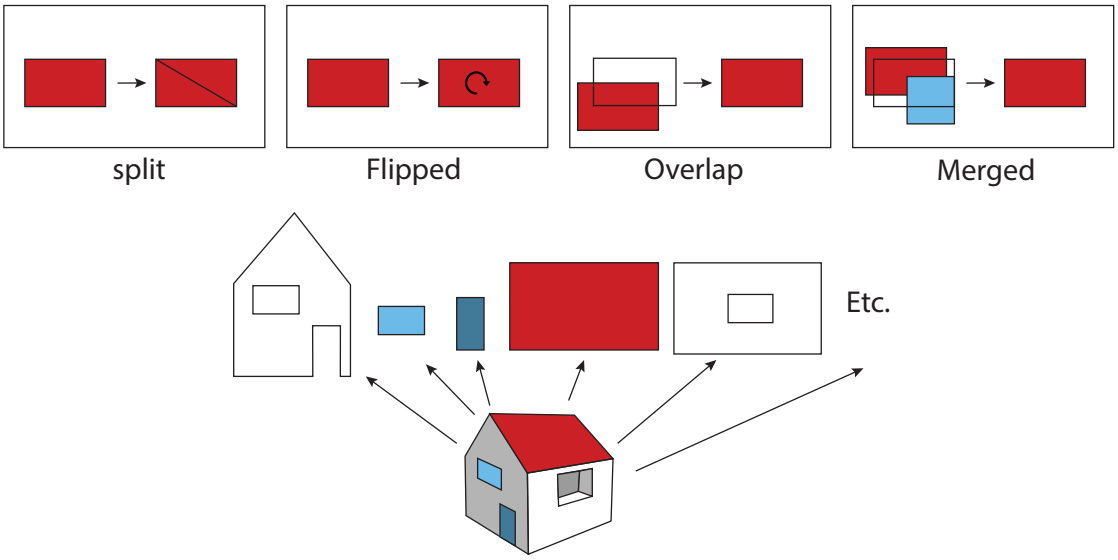


Figure 6.4.: Sementics and materials preservation

Extra use cases repairs

These repairs are not the result of a found error after validation but enhancements to make 3d city models more usable for use cases as explained in Chapter 5.

Validate and repair semantics

The semantics repair function ensures that each face of the geometry is correctly classified as either a ground, roof, or wall surface based on its orientation. The normal vector of each face is calculated and compared to the Z-axis to determine the angle between them. If the angle is close to 0°, the face is classified as a roof; if it's close to 180°, the face is classified as ground; otherwise, it is considered a wall. The angles used are based on Section 2.5.

The semantics repairs start by checking if the geometry already contains semantic information. If semantics already exist, it tries to use existing surface type indexes by looking through the list. If a matching surface is found (i.e., the "type" field matches the surface type, and no other fields are present), the index of that surface is used. A new surface with the specified surface type is created and added to the list if no matching surface is found. If semantics do not exist, a new surface-type list is made.

For (re-)assigning values, two parameter options exist: `SemanticsAdd` and `SemanticsValidation` (Figure 6.5). When `SemanticsAdd` is used, only faces without a value are assigned new semantics. When `SemanticsValidation` is used, all the faces are reassigned. Based on the parameter, the function loops through the boundaries, and the normal is computed for each face that needs repair. The corresponding semantic value (ground, roof, or wall) is assigned based on its orientation. If the geometry does not have predefined semantics, a new value assigns the correct semantic value to each face by evaluating its orientation for each boundary in the geometry.

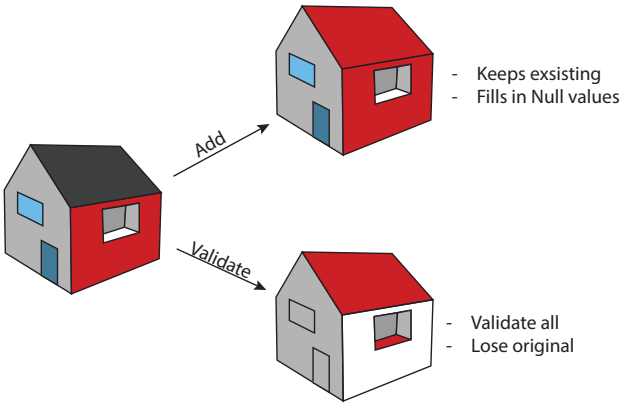


Figure 6.5.: Difference between semantic parameters

Watertight and Orientation Repairs
The watertight repair process ensures that all geometric primitives are closed and consist of one volume. Transforming non-solid geometries into solids when necessary. Geometries classified as ”MultiSurface” or ”CompositeSurface” have their type changed to ”Solid,” and their boundaries are wrapped inside a new boundary array to ensure the CityJSON schema. Similarly, if the geometry type is ”MultiSolid,” it is converted into ”CompositeSolid” to repair a single cohesive volume. These changes ensure the geometry adheres to the watertight standard, repairing defects such as holes and disconnected parts. The orientation repair process focuses on converting any ”MultiSurface” types into ”CompositeSurface ” so that those geometries are also checked for shell errors.

Triangulation, Mesh Simplification, and Mesh smoothing
The repairs are done per shell. Each shell is converted into a CGAL surface mesh (Botsch et al., 2023), which is triangulated with the help of acCGALs Polygon mesh Processing (Loriot et al., 2023).Border vertices are constrained to maintain the original shape for non-closed geometries when a mesh is not closed. after that Mesh simplification is executed in two steps:

1. Edge colapse, using CGAL's triangulated Surface Mesh Simplification (Cacciola et al., 2023). The process targets collapsing edges, which are longer than 0.5 * the body diagonal of the oriented bounding box (Katrioplas and Rouxel-Labbé, 2023). Edges that are needed to preserve the structural integrity and intended geometry of the model are marked as constrained edges if they are edges between two faces with normals with angles of 90 degrees or less. Retaining these edges ensures that essential contours and design features remain intact during simplification.

2. Mesh smoothing is accomplished through CGAL's Polygon mesh processing smoothing function (Loriot et al., 2023), which moves vertices towards a weighted barycenter of their neighbors based on the mean curvature flow. The process uses shape smoothing with five iterations to refine the mesh while preserving the boundaries.

To lose slivers where possible, CGALs Polygon mesh Processing (Loriot et al., 2023) is used to remesh planar patches. Which detects and simplifies planar regions of a mesh by merging adjacent faces into fewer or larger triangular polygons.

6.2.3 Post-processing of the 3D City-model

Post-processing a 3D City model after repairs is essential to ensure the data is clean and optimized for future use. This process involves two key steps: cleaning the file to remove inconsistencies introduced during repairs and enhancing the data to improve its accuracy and utility.

Cleaning the data

Cleaning CityJSON data is essential to ensure its validity according to the schema, especially after repair processes that may introduce inconsistencies. This involves removing unnecessary elements, correcting structural issues, and ensuring all objects and relationships correctly align with the schema requirements.

Unused vertices - Due to the repair and reconstruction process, there could be duplicate 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, renumber the ones that need to be changed, and deletes all the orphan vertices, after which all the high vertices must be enumerated again in the boundaries.

Changing the scale - Repairs could add points to the vertices list during the repair process. Seeing one vertex must be an array with 3 integers, all the vertices, even if they could be doubles, are translated to integers. Only rounding the points to integers results in (floating) point errors (mostly found in 203 and 204). Therefore, the scale is redefined by checking how much bigger the scale must be (sizing up by 10) so floating point differences, resulting from rounding to integers, are less than the `snap_tollerance`. The maximum number of coordinates is 0.1 * `Maxint`, which is 2147483647 assuming 32-bit system, or 0.1 *`Minint` (for negative coordinates), which is -2147483648 assuming 32-bit system. The implementation works with int64, which can be much higher, to find out how much the scale needs to change, but when the number becomes above/below the 32-bit max, the points are rescaled till they are below 0.1* `Maxint`. So, the 3D city models stay useable for all use cases, which may use 32-bit systems, with the risk of keeping floating point errors.

Empty Boundaries - When, due to repairs, geometries are deleted, they end up with an empty boundary. A geometry object must have one member with the name `boundaries`, with a hierarchy of arrays so that an empty boundary would invalidate the CityJSON schema. Therefore, the whole geometry object, which is not mandatory, is deleted.

Changing Multi-/CompositeSolids to Solid - When a MultiSolid or a CompositeSolid ends up with only one solid. The type is changed to Solid, and the boundaries of the geometry (and also semantics and material) are changed accordingly. This makes the file better readable because the Solid geometries match their type.

Enhancing the data

Enhancing CityJSON data involves refining and optimizing it after the repair process to improve its usability and ensure accuracy. This includes modifying geometric structures, updating metadata, and improving the classification of city objects.

Detriangulation - For some of the repairs, geometries are converted to triangulated meshes, which are repaired. However, to keep the data as close to the original, detriangulation is used. Seeing no algorithm

Detriangulation is done in three steps (visualized in Figure 6.6):

1. Find coplanar patches: This is done by calculating and normalizing the normal for all triangles. The triangles with the same normal are on the same plane and grouped.
2. Find connected components in coplanar patches: This is done by building a graph based on triangles sharing edges.
3. Merging triangles into a face involves the identification and processing of the border half-edges. These are half-edges in the mesh that do not have an opposite (paired) half-edge. The procedure can be outlined as follows:
   a) The first step is to detect all half-edges that do not have a corresponding opposite half-edge. These edges form the boundary of the surface or hole within the triangle mesh.
   b) Starting from any border half-edge, the algorithm proceeds to *walk* over the connected half-edges. The next half-edge in the sequence is found by following the boundary. This process continues until the walk completes a closed loop, forming a ring. Then, it starts from an unused other border edge if existing.
   c) To find the order between the outer boundary and interior holes, the rings are sorted based on the area they enclose. The ring with the largest enclosed area is considered the outer boundary, while the smaller rings represent holes.

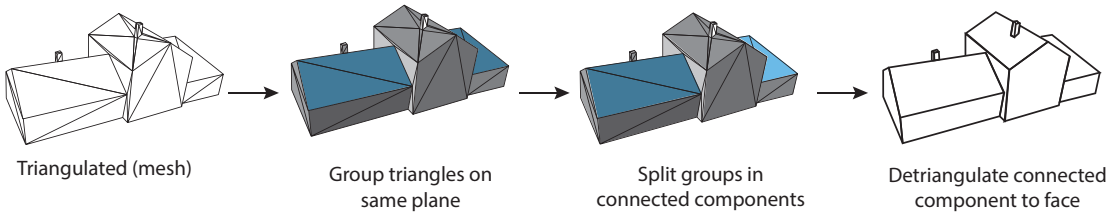


Figure 6.6.: Detriangulation approach

Changing geographical extent - When a CityObject has the `geographicalExtent` member, it will be updated after repair. This is done by collecting all the object's vertices and recalculating the boundingBox. Also, `geographicalExtent` in the `metadata` is updated if it exists. These values are in the real-world coordinate system of the dataset; the resulting bounding box is scaled and translated by the values in `transform` before it is added.

LOD - For 3D city models, the classification of objects by LOD enhances the efficiency and precision of data-driven applications (Dukai, 2018). Therefore, (re-)classifying the LOD after a repair would enhance the data. However, the model Dukai (2018) is not yet transferable to the

real data set. Therefore, now only the LOD of geometry is changed when the last resort repair, making a bounding box, is done. This method changes the LOD to 1.0.

6.2.4 Output

The output consists of the repaired 3D city model in its original format (CityJSON or OBJ) and a detailed repair report. The report provides an overview of the repairs made and metadata on the process, while the repaired model preserves its original structure.

Repaired 3D city model

The repaired 3D city models will be written back to their original file format (CityJSON or OBJ). When no repairs are done (the file is the same as the original), no output 3D city model will be written.

For CityJSON, the output is made by copying the original file so that untouched properties (such as type, version, and transform) will be the same as the original. Copying the whole CityJSON ensures that properties not obligatory (such as metadata and appearance) are in the repaired 3D city model. The new array will replace the original vertices array, and the new CityObjects array will be filled by converting the TU3dJSONs back to CityObjects (Figure 6.1). An extra key, AUTOr3pair, is added (Figure 6.7), which shows when the model was repaired, how many of each feature were repaired, and some information about the program and where the Repair report (Section 6.2.4) is stored. When the parameter `addrepair` is used, there will also be a key AUTOrepair added to every CityObject, showing which repair(s) are done on it.

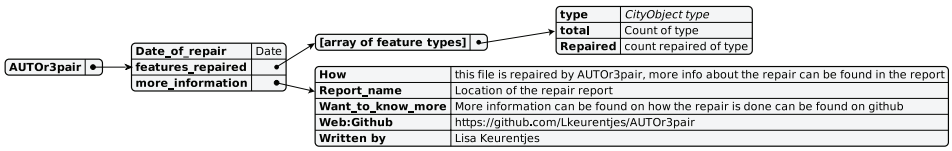


Figure 6.7.: AUTOr3pair key for repaired 3D city models

To write output, OBJ CGAL's I/O stream package is used (Fabri et al. (2023)), which writes the vertices and polygons from a range. The vertices will come from the repaired vertices array. The polygons come from the CityObject "like" Figure 6.2. CGAL also manages conversion between 0-based boundaries to OBJ's 1-based boundaries. Objects are not grouped in the OBJ output at this moment.

Repair Report

In addition to the repaired 3D city model also a repair report is written. The repair report is meant to give the user clarification of how the 3D city model is repaired. The repair report is similar to val3dity in JSON format and is structured roughly the same. Figure 6.8 visualizes the used schema. The report consists of 14 properties, whose names are supposed to be self-explanatory. The key `Parameters` shows which parameters are used for the repair process (as explained in Section 6.3.1 so the user can see which influenced the repair process. `Data_errors` is used when a file is unreadable or if val3dity outputs a data error. `features_overview` and `primives_overview` shows the number of repaired objects per type. `input_file` and `output_file` and their types show which file is repaired and where the repaired 3D city model is saved. To also give the user some metadata on the process the

key `time` shows the moment the file is repaired and the key `version` shows which version of AUTOr3pair is used. The key `type` is always AUTOr3pairReport.

`repaired` is a boolean to show if there is something in the 3D city model repaired. `Features` gives an array of all features and subkey `repaired` shows if this feature is repaired and `features_repaired` lists all the ids of repaired features. To find out more about the repairs done, the user can see per `Primitive` which repairs are done in which order by checking `repairs`. `repairs` show per round what kind of repair(s) is done and per repair, the id/name, description, and change in boundary by showing the `boundary_before` and `boundary_after`. To make the boundary change also clear for OBJ conversion between 0-based boundaries to OBJ's 1-based boundaries is done before the report is written.

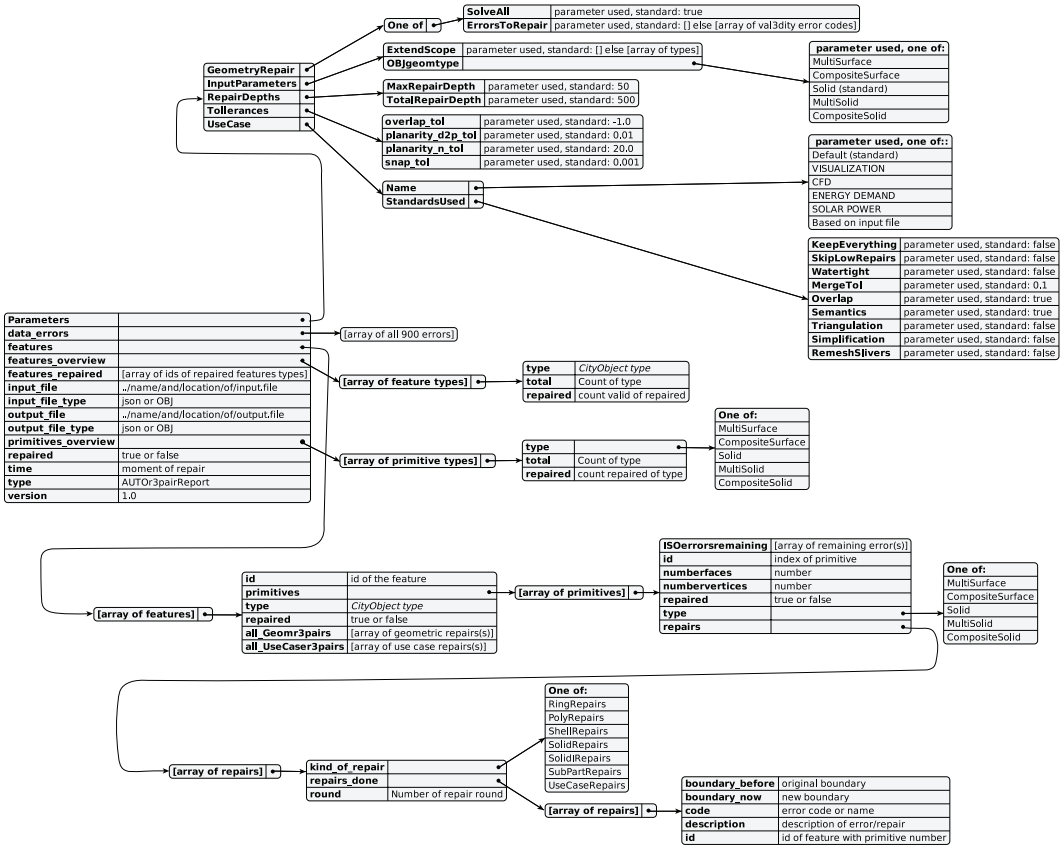


Figure 6.8.: Repair report

6.3 Parameters

Parameters can influence the repair process. The parameters can be changed by users by inputting a use case (explained in Section 6.3.1) or by inputting an input `JSON`. All the parameters and their standard values are discussed below, and in Section 6.2, which processes the parameter influences are mentioned. The parameters can be divided into six categories: input parameters, output parameters, Repair depths, geometry repairs, val3dity tolerances,

and use case parameters. The first five can only be changed by using an input `JSON`, while the use case parameters are grouped by their use case.

Input parameters influence how the 3D city models are handled during the repair process. The parameters are:

- `OBJ_geomtype`: `"Solid"`, which decides what kind of geometric primitive the geometries in an OBJ file are. Alternately, users can change the value to other geometric primitives (MultiSurface, Compositesurfce, MultiSolid, CompositeSolid)
- `ExtendScope`: `[]` *(empty list)*, can add other types of CityObjects to repair. For this thesis, only building and its subparts are in the scope, but users can add other types by listing them in this parameter.

Output parameters influence what the terminal shows during the repair process, and for cityJSON, if extra attributes are added to explain the repair process. The parameters are:

- `ShowProgress`: `TRUE`, When the boolean is set to true, the terminal shows which items it has validated and which repairs are done so that the user can follow the progress of AUTOr3pair.
- `Debugging`: `FALSE`, When the boolean is set to true, the terminal shows more extensive output, such as the vertices list. This can be used when debugging the program or when a user wants to look further into specific steps.
- `AddAttribute`: `FALSE`, When the boolean is true, CityJSON CityObjects get an extra attribute with what is repaired. This makes the file more significant (primarily for large 3D city models). This is only done when the user changes the parameter. The standard attribute to the file says that it is repaired and when it will always be added.

Repair Depth(s) stop the repair process when their threshold is reached. The parameters are:

- `maxRepairDepth`: 50, is the maximum times the same category repair can be done on a CityObject, before doing the global approach, alpha wrap. Users can change the value according to their time limitations.
- `TotalRepairDepth`: 500, is the maximum times the same CityObject is repaired before doing the global approach alpha wrap. Users can set the value lower if they want the repair process to be faster and higher if they want to give the repair process more tries. However, manual testing did not yield better or different results with a higher value, seeing when 500 is reached it was mostly because of repair bugs which could not be solved by trying it more often.

Geometry repair Standards can help users exclude certain errors from the repair process. The parameters are:

- `solve_all`: `TRUE`, When the boolean is set to true, all errors found by val3dity will be repaired.
- `errors_to_solve`: `[]` *(empty list)*, When users want to exclude errors, they can add them to the list, and repairs for those errors will not be done. Due to how val3dity is implemented, higher order errors will also not be repaired, seeing those are not checked. When users add errors to this parameter, `solve_all` will be changed to False automatically.

val3dity Tolerances are the tollerances of val3dity as explained in Section 4.1. The parameters used for AUTOr3pair are:

- `overlap_tol`: 0.001, which is the tolerance for testing the overlap between primitives in CompositeSolids and BuildingParts. The maximum allowed distance for overlaps. It helps to validate the topological relationship between Solids forming a CompositeSolid. In contrast to val3dity, the overlap tolerance should always be the same or higher than the snap tolerance. Otherwise, it results in the repair loop from Figure 6.9. When a user chooses a lower value, an error is outputted, and the value is changed to the same as the snap tolerance.
- `planarity_d2p_tol`: 0.01, which is the tolerance for planarity based on a distance to a plane. The distance between every point forming a surface and a plane must be less than the threshold.
- `planarity_n_tol`: 20, which is the planarity tolerance based on normal deviation. It helps to detect small folds in a surface. the threshold refers to the normal of each triangle after the surface has been triangulated
- `snap_tol`: 0.001, which is the tolerance for how close vertices can be together before they are snapped into the same point.

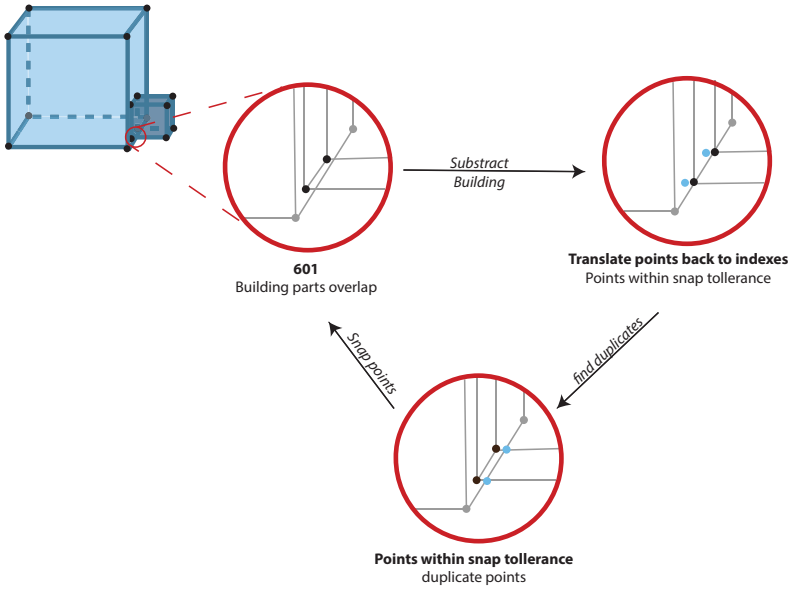


Figure 6.9.: Repair loop when overlap tolerance is smaller than snap tollerance

6.3.1 Use case parameters

As discussed in Chapter 5, some use cases have additional requirements for a 3D city model to be valid. Therefore, a user can also add additional requirements to the repair process. Users can give one of the four use cases. Then, a standard set of the parameters is used, shown in

Table 6.1. How they influence the repair process is per use case discussed below. User can also make their own standard set using an input `JSON` to change the combination of parameters to their use case.

	Default AUTOr3pair	**CFD**	**Energy Demand**	**Visualization**	**Solar Power Estimation**
`KeepEverything`	FALSE	FALSE	FALSE	TRUE	TRUE
`SkipLowRepairs`	FALSE	TRUE	TRUE	FALSE	FALSE
`Watertight`	FALSE	TRUE	TRUE	FALSE	FALSE
`Orientation`	FALSE	FALSE	FALSE	TRUE	TRUE
`MergeTol`	↔ 0.1	↔ 0.25	↔ 0.75	↔ 0.1	↔ 0.5
`Overlap`	TRUE	FALSE	FALSE	FALSE	FALSE
`SemanticsAdd`	TRUE	FALSE	TRUE	FALSE	TRUE
`SemanticsValidate`	TRUE	FALSE	FALSE	FALSE	TRUE
`Triangulate`	FALSE	TRUE	FALSE	FALSE	FALSE
`Simplification`	FALSE	TRUE	FALSE	FALSE	FALSE
`RemeshSlivers`	FALSE	TRUE	FALSE	FALSE	FALSE

Table 6.1.: Use cases standards with default values

Use case : CFD requires that geometries are watertight and non-intersecting. Therefore, `Watertight` is set to true, which converts all geometries to Solids or CompositeSolids, and `Overlap` is set to False, which next to the overlap within a CityObject also validates and repairs overlap between different CityObjects. Seeing small details is not very important `SkipLowRepairs` is True, which deletes faces with Ring or Polygon errors and therefore starts repairing on shell level. For the simulations, it does not matter if objects are merged or split; the `MergeTol` is set to 25%, so geometries will be merged if their overlapping volume is 25% of the first encountered. Additionally, CFD needs triangulated objects (`Triangulates`, with no small features, sharp edges, and sliver triangles. `Simplification` and `RemeshSlivers` are therefore set to True.

Use case: Energy demand requires buildings to be watertight for accurate volume calculations, `Watertight` is set to true, converting all geometries to Solids or CompositeSolids. Objects also must be non-intersecting; therefore, `overlap` is False. Additionally, external surfaces should be subdivided into semantic parts (roof, wall, and ground surfaces) for more precise energy demand estimation, so `SemanticsAdd` is set to true to add (missing) semantics. However, existing semantics are not validated and repaired(`SemanticsValidate`); seeing windows and doors can affect the energy calculation. Lastly, seeing energy demand is often done per building `MergeTol` is adjusted to 75%, which makes the chance of buildings merging small, and seeing small details is not very important `SkipLowRepairs` is True, which deletes faces with Ring or Polygon errors and therefore starts repairing on shell level.

Use case: Visualization requires that building geometries have correctly oriented surfaces to avoid rendering errors, so `Orientation` is set to true, converting MultiSurfaces to CompositeSurfaces also to have orientation check on the shell. To prevent rendering issues caused by overlapping surfaces, `Overlap` is set to False, ensuring that intersections between buildings are validated and repaired. However, `MergeTol` is adjusted to 10%, seeing merging buildings don't change details. Lastly, local repairs are preferred to maintain a higher level of detail, so

`KeepEverything` is set to True to keep all (dangling) parts in geometries, and `SkipLowRepairs` is set to False.

Use case: Estimation of solar power requires accurately oriented roof surfaces to estimate solar irradiation correctly, so `Orientation` and `SemanticsAdd` and `SemanticsValidate` are set to true to validate and repair roof face orientation and semantic values. Detailed local repairs are preferred over global adjustments to ensure shadow accuracy, so `KeepEverything` is set to true, and `SkipLowRepairs` is set to False. Additionally, to avoid intersecting roofs, `overlap` is False. Lastly, seeing solar power potential is sometimes done per building `MergeTol` is adjusted to 50%.

7 Experiments

This chapter has four parts. The first three discuss testing the program and assessing the repaired output 3D (city) models. The last section discusses the strategies used and the usability of AUTOr3pair.

7.1 Unit tests

To validate that AUTOr3pair is working correctly, a Python "pytest" framework[1] is implemented. The tests run AUTOr3pair using the `subprocess` library and feed it with various files and parameters. The framework tests, among others:

1. All geometry repairs per error code. For test data, the val3dity test data is converted into CityJSON (where possible as multiple geometry primitive types) and OBJ and named after the error. Figure 7.1 gives eight examples of these "simple" unit tests. Each repair is validated on:
    - Is the error present when validating the 3D city model before repairing? Which is done with the help of val3dity.
    - Does the AUTOr3pair run correctly while repairing the 3D city model?
    - Is the error not present when validating the 3D city model after the repair process? Which is done with the help of val3dity
    - Does the repair report yield the error *and when possible is the new boundary correctly changed*.
    - *when applicable, do different parameters yield (correct) different results?*
    - *when applicable, are the materials and semantics changed accordingly?*
    - Is the CityJSON schema of the output 3D city model file valid? Which is done with the help of CJIO
2. Valid and invalid parameters, to check their use and influence on the repair process
3. All additional use case repairs per parameter, for example, converting MultiSurfaces to watertight Solids and repairing all None value semantics.
4. Various invalid file formats and empty files to check if correct run errors are displayed
5. Various valid files to check no repairs will be done when not needed.

[1] Krekel et al. (2004)

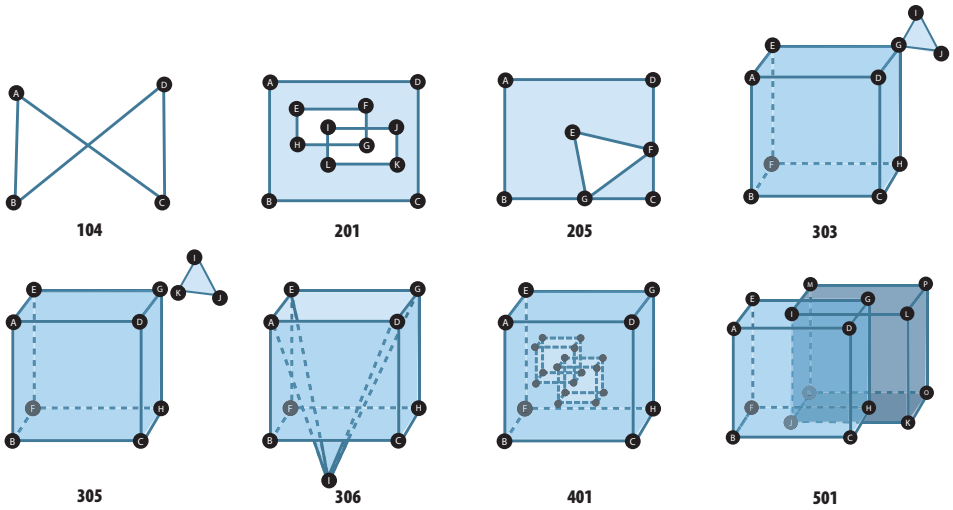


Figure 7.1.: Eight example unit tests

All unit tests can run automatically to verify that the compilation went smoothly and that there are no bugs. Writing and saving all output files require a lot of space; therefore, the output is deleted after the tests. When users want to manually check the test output data, the marker `"keep"` can be used, which skips the deleting step. The total number of tests done is 747, of which 618 test repairs in different configurations (Figure 7.2).

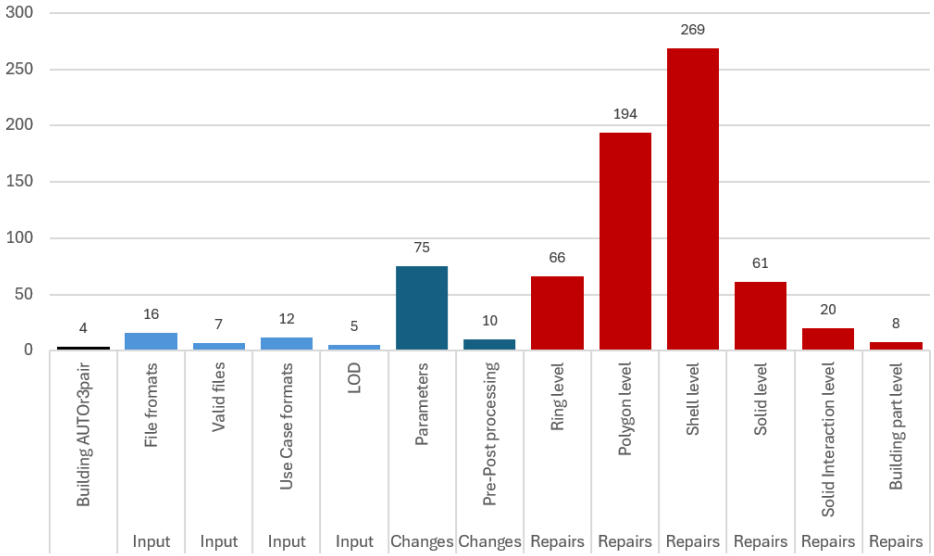
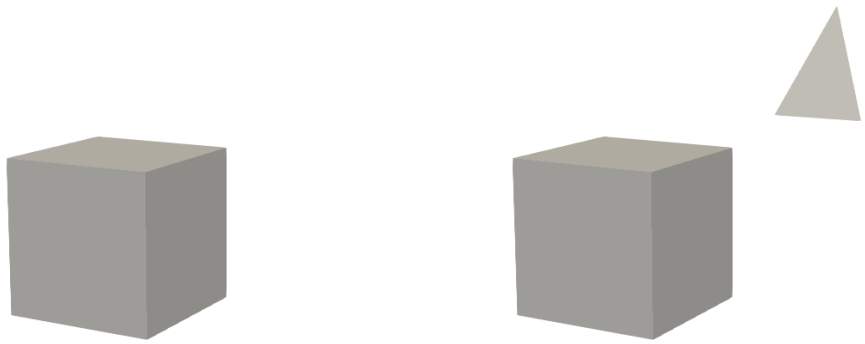


Figure 7.2.: Number of tests per category

7.2 Effect of (use cases) parameters

Various parameters are critical in geometry processing and repair. This section shows the effect of parameters on simple repairs. Section 7.3 will also show some of the impact of parameters on a larger scale.

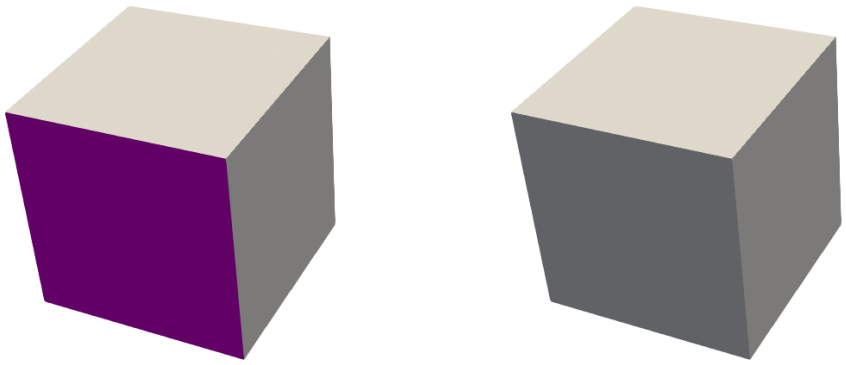
KeepEverything: The `KeepEverything` parameter ensures that all elements of geometry are retained during the repair process and prioritizes preserving geometric details. In Figure 7.3, an inner shell that was outside is repaired. Without `KeepEverything`, the inner shell is deleted, but with `KeepEverytthing`, the shell is flipped and added as a different geometry.



(a) Without KeepEverything (b) With KeepEverything

Figure 7.3.: Parameter KeepEverything

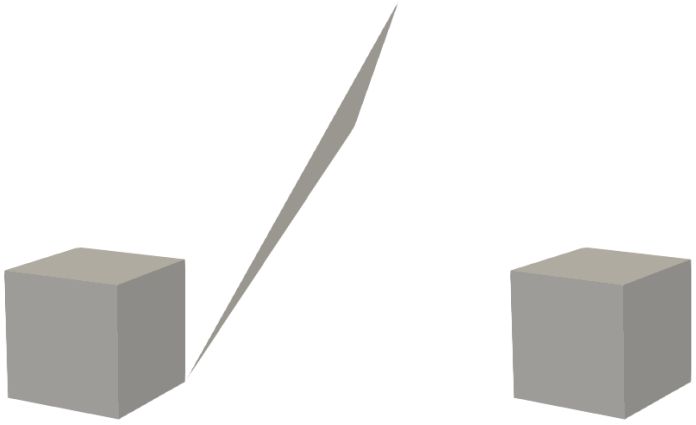
Watertight and Orientation: The `Watertight` parameter ensures that all geometries are converted into watertight solids during the repair process, which is crucial for accurate volume calculations. The `Orientation` parameter ensures that all geometries are converted into CompositeSurfaces during the repair process, adding orientation, connection, and self-intersection checks. Figure 7.4 shows the difference between repairing a MultiSurface cube with a wrongly oriented surface (in purple) as a MultiSurface or as a CompositeSurface (parameter `Orientation`) or Solid (parameter `Watertight`)



(a) Repaired as MultiSurface (b) Repaired as CompositeSurface or Solid

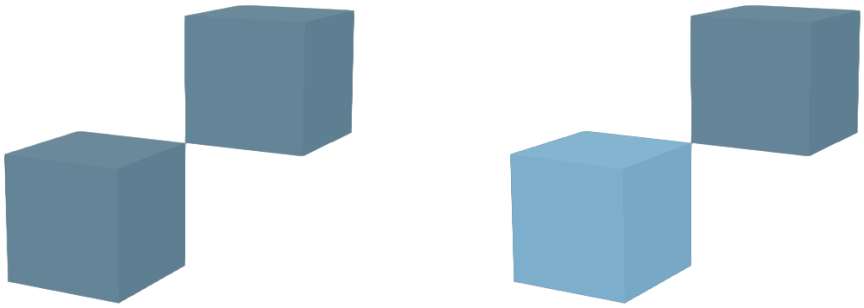
Figure 7.4.: Parameter Orientation and Watertight (in purple backside of surface)

Parameters `Watertight` and `Orrientation` also influence repairs when the geometry is split into multiple parts. Figure 7.5 shows a non-manifold shell consisting of a cube and a separate surface. When repairing this, the shell is split into two parts. When `Watertight` is used, the surface will not be added as a different geometry, seeing it could never form a watertight solid. Figure 7.6 also shows a non-manifold shell, but seeing they have a volume, they are both kept. When `watertight` is used, the geometry is split into two solids instead of combining them in a not watertight MultiSolid.



(a) Without watertight (b) With watertight

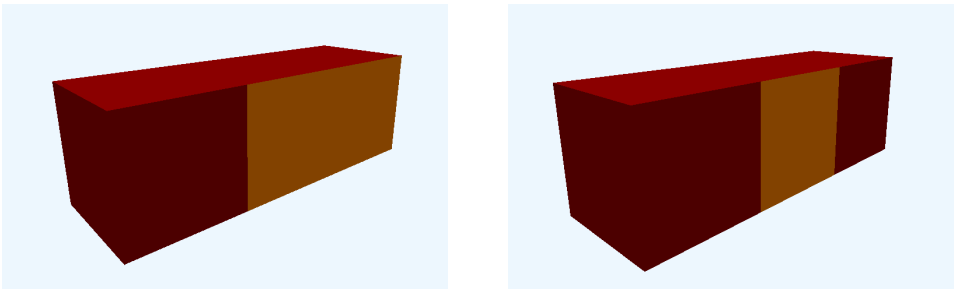
Figure 7.5.: Parameter Watertight when splitting a surface in geometry



(a) Without watertight, one MultiSolid (b) With watertight, two Solids

Figure 7.6.: Parameter Watertight when splitting two solids in a geomerty

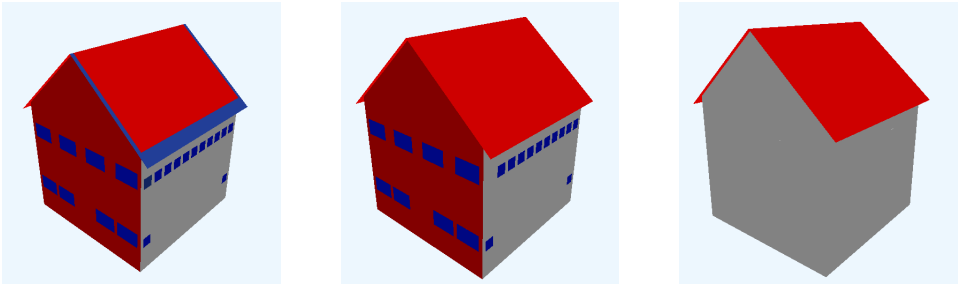
MergeTol: The `MergeTol` parameter controls the tolerance for merging overlapping geometries. It determines how much two geometries can overlap before merging into a single entity. In Figure 7.7, the repair of three cubes, of which two overlap, are shown with a high and low `MergeTol`. With a low `MergeTol`, overlapping geometries are combined into a larger geometry. Conversely, with a high `MergeTol`, geometries are kept separate, and the second one is changed by taking the difference.



(a) With MergeTol of 0.1 (b) With MergeTol of 0.9

Figure 7.7.: Parameter Mergetol

Semantics: The semantics parameters `SemanticsAdd` and `SemanticsValidate` better the semantics of objects. As shown in Figure 7.8 parameter `SemanticsAdd` only add semantic values to Null values, which can cause wrong semantics (for example, the roof value on the wall) but preserves values not in the validation (such as windows). The parameter `SemanticsValidate` changes values of faces that already have values, repairing incorrectly labeled faces but also labeling faces such as windows wrong.



(a) Original semantics (b) Parameter semanticsAdd (c) Parameter semanticsValidate

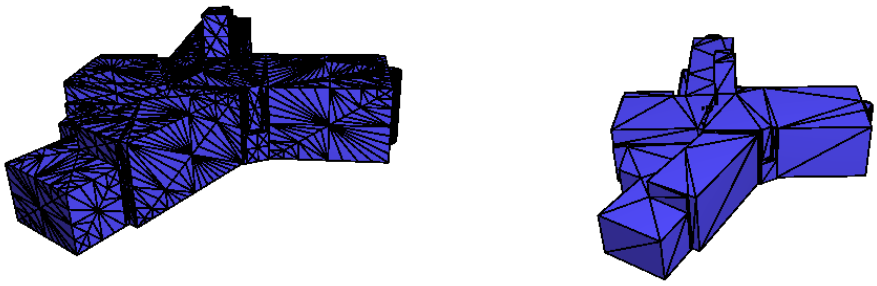
Figure 7.8.: Parameters semantics

Triangulate: The `Triangulate` parameter ensures that all surfaces are converted into triangles during the repair process. In Figure 7.9a, a building consisting of multiple merged parts is shown. When parameter `Triangulate` is used, the output geometry will consist of triangulated faces (without holes); City objects that are not repaired are also triangulated (when in scope) if this parameter is used.

(a) Without triangulation (b) With triangulation

Figure 7.9.: Parameter triangulation

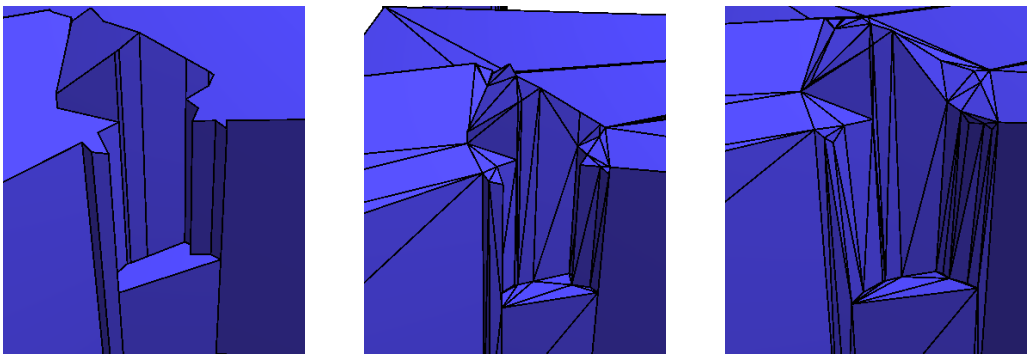
RemeshSlivers: The `RemeshSlivers` parameter focuses on eliminating small triangles in the geometry. In Figure 7.10, an object containing sliver triangles is shown before and after re-meshing. Without `RemeshSlivers`, these narrow triangles remain in the model, potentially affecting the quality of the analysis. When `RemeshSlivers` is enabled, the geometrical "resolution" is lowered by minimalizing the number of triangles on the same plane.



(a) Without Remesh (b) With Remesh

Figure 7.10.: Parameter RemeshSlivers

Simplification: The `Simplification` parameter reduces the complexity of geometries by removing unnecessary details, such as small features and small edges. The example in Figure 7.11 shows a building with small column-like features attached to the facade. Without `Simplification`, the model retains its intricate features. With `Simplification` enabled, the geometry is simplified by collapsing small edges and smoothing the model, resulting in a lower-resolution model that maintains the overall shape and structure.



(a) Without simplification (b) Triangulated (c) With simplification

Figure 7.11.: Parameter Simplification

Show progress: Although printing to the terminal slows down the execution, it could be nice to see some progress in what is happening. When the `Showprogress` parameter is used, Listing 7.1 shows an output example. Tabs space the Output to differentiate the main task from the sub-tasks. The two biggest tasks are adding the vertices, which can be very time-consuming with big data sets due to the search for duplicate vertices. And the Repair loop. For adding the vertices, the % of how it is is shown in steps of 25%. In the repair loop, the repairs for each building are shown per geometry. For geometric repairs, the location given by val3dity is shown. For use case repairs, it shows which use case is being done.

```
Start parsing the json file
Finished parsing the json file
Set the vertices and check for duplicates
    At 25% of adding the vertices
    At 50% of adding the vertices
    At 75% of adding the vertices
Start the repair loop
Start the repair of: object
    Pre processing: changing duplicate vertex indexes
    CityObject with id object.0 geometry has the following errors: [203]
        Repaired Geometery object-0 with error 203 on "id=1|geom=0|shell=0|face=6"
        Repaired Geometery object-0 with error 203 on "id=1|geom=0|shell=0|face=7"
        Repaired Geometery object-0 with Use Case repair: Validate semantics
    CityObject with id object.1 geometry has the following errors: []
        Repaired Geometery object-1 with Use Case repair: Validate semantics
Repair loop is done
Post processing: Parent child relations
Post processing: Orphan vertices
Post processing: Change Multi-/Composite-Solid to one solid
Post processing: Change Metadata
Write output file(s)
```

Listing 7.1: Example output

7.2.1 disadvantage of many parameters

Although using many parameters enhances the repair process's flexibility, it can also result in contradictions or recurring repair loops. An example that can result in contradictions is `KeepEverything` and `Watertight` when splitting a non-manifold shell with a dangling surface. `KeepEverything` says that everything needs to be preserved; however, `Watertight` argues that everything needs to be watertight. So when repairing the surface, it is kept but added as Solid. Then, the shell does not have enough surfaces, so the shell is deleted.

Another example of 2 conflicting parameters is `SkipLowRepairs` and `snap_tol`. When `SkipLowRepairs` is used, polygon repairs are done by deleting the face. But repair 302, shell not closed, could bring back the error. This case happened a few times in the test data of Section 7.3 when polygons report 104. Instead of adding the vertex to the new location, the face is readded. When the face is the same as the original due to `snap_tol`, the face will detect 104 again, which restarts the repair, creating a loop.

7.3 Repairing well-known 3D city models

In this section, open-data 3D city models are repaired and evaluated. For repairs, two factors count: (1) the validity percentage and (2) the geometric difference from the original. The validity percentage is evaluated with val3dity (Section 4.1). The file will be validated before and after the repair process, and the valid percentage of buildings will be compared. the parameters used for val3dity are the standard parameters except for `overlap_tol`, which will be the same as the `snap_tol` as explained in Section 6.3. Also, for some cases, the remaining errors will be evaluated, and hypotheses will be made on why these errors aren't repaired/ are made.

The Hausdorf distance is used to evaluate the geometric differences. It reflects the most significant distance one would travel to cover every point in both sets, making it useful in shape comparison and computer vision tasks (Tang et al.). It is symmetric and non-negative, providing a robust way to quantify spatial differences between two sets. Each geometry in a

CityObject is compared with the geometry (or geometries if it is split). To give some context, the distance is compared with the largest pairwise distance between vertices for the same geometry. The Hausdorf distance calculation script can also be found in the GitHub repository under tools.

7.3.1 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.

Although the 3DBAG is comprehensive in its inclusion of detailed geographic information, it contains numerous (minor) geometric errors that can affect its precision and usability. These errors stem from inconsistencies in data processing and limitations in capturing complex urban environments. As test data for this thesis, the group of BAG tiles surrounding the old center of Leiden was chosen (Figure 7.12). The set consists of 20 small tiles (file size between 1 and 7 MB) and one bigger tile (file size 17.7MB).

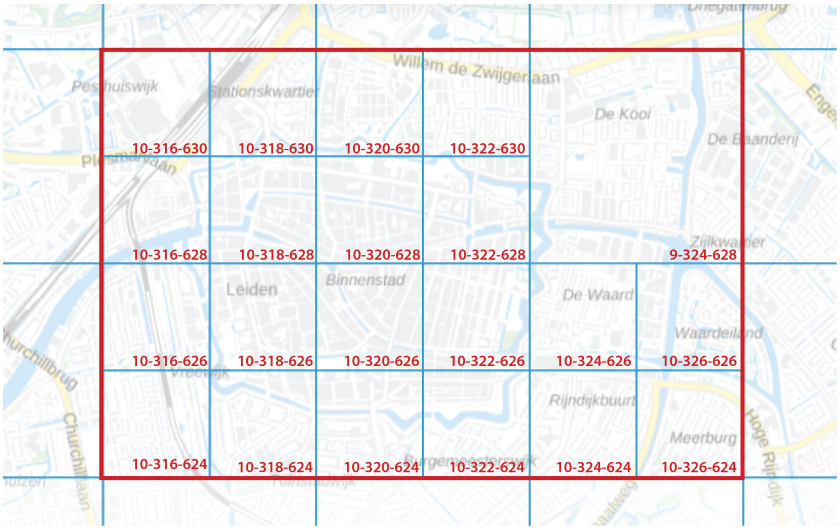


Figure 7.12.: 3DBAG tiles from Leiden and their names

Evaluating the validity of the tiles, the percentage of valid buildings is between 85% and 99% (Figure 7.13). The geometries consist of MultiSurfaces as footprints and solids in 3 different LODS. The existing errors are on the Ring, Polygon, and Shell levels. The Ring errors consist of duplicate vertices and ring self-intersections. The polygon errors found are mostly planarity issues and one intersecting ring. The shell errors consist primarily of closed shells and non-manifold/orientation problems. For complicated buildings, there are also some shell self-intersections.

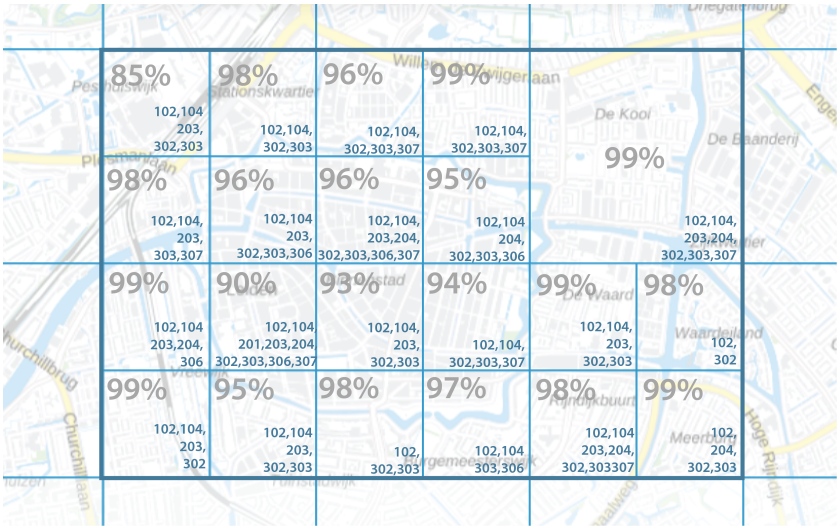


Figure 7.13.: 3DBAG tiles from Leiden and their validity (in grey the percentage of valid buildings and in blue the existing errors)

The 3DBAG tiles are repaired with the default parameters, except for changing `"SemanticsValidate"` to false. This way, the file is repaired as efficiently as possible, and the existing semantics are kept, but surfaces ending up with `Null` get a semantic assigned. After repair, the validity percentages are all mostly 100% (Figure 7.14). Tile 10-316-630 has the lowest percentage, but that is due to 3 buildingparts intersecting (only 1 error) and having only 64 buildings. The remaining errors seem to come from rounding the vertices list of CityJSON to integers cause the errors are only found after the new json is made. Although the scale is changed to avoid this problem (Section 6.2.3), there will always remain small floating point errors.

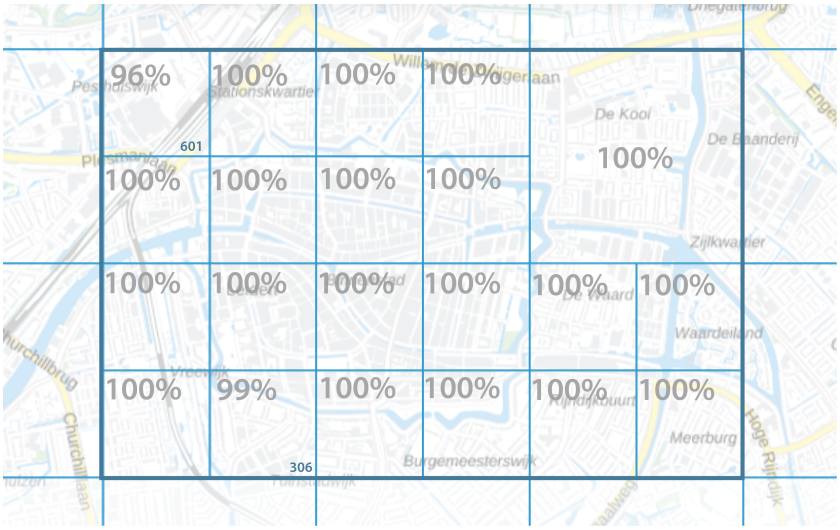


Figure 7.14.: 3DBAG tiles from Leiden and their validity after repair (in grey the percentage of valid buildings and in blue the existing errors)

When evaluating the geometric difference(Table 7.1), most differences are lower than 1% of the distance of the CityObject. However, there are some considerable differences, although often not visible when comparing (Figure 7.15). The explanation for these differences comes from repairing error buildingparts overlap (601) and deleting parts such as shell not closed (302), multiple connected components (305), and self-intersections (306). When repair 601 merges two objects or takes part of the object by taking the difference, the object and its size change drastically. When part of an object is deleted because it is not manifold (302) or connected (305) or when faces are deleted seeing they intersect (306), this results in original points being "far" away from the end geometry. When the maximum difference is 0.001, the only geometric difference is the snap tolerance, which snaps vertices together. The bounding boxes are, at the moment, not taken into account when getting the Hausdorf per LOD; this is because using the bounding box changes the LOD, which makes comparing to the original difficult, seeing the original LOD is not saved. However, a Bounding box is only used one time in the whole dataset. This bounding box in 10-324-624 results from one almost planar shell part, resulting in non-manifold and self-intersecting in Alpha wraps and convex hulls.

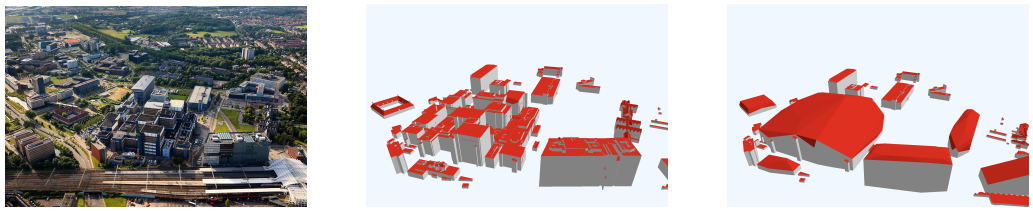
Table 7.1.: Geometric difference after repair of the 3DBAG tiles

Tile Name	**Hausdorff Distance LOD 0**	**Hausdorff Distance LOD 1.2**	**Hausdorff Distance LOD 1.3**	**Hausdorff Distance LOD 2.2**	**Global repair used**
9-324-628	0.001 (0.0%)	0.48 (0.6%)	34 (22%)	37 (25%)	12 (0.01%)
10-316-624	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	9.8 (20%)	3 (0.01%)
10-316-626	0.001 (0.0%)	0.001 (0.0%)	23 (22%)	24 (24%)	2 (0.01%)
10-316-628	0.001 (0.0%)	0.001 (0.0%)	40 (28%)	39 (27%)	5 (0.1%)
10-316-630	0.0 (0.0%)	38 (32%)	38 (32%)	70 (24%)	14 (5%)
10-318-624	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	18 (24%)	5 (0.01%)
10-318-626	0.001 (0.0%)	0.001 (0.0%)	26 (28%)	27 (29%)	22 (0.5%)
10-318-628	0.001 (0.0%)	22 (29%)	26 (33%)	24 (30%)	23 (0.5%)
10-318-630	0.001 (0.0%)	0.001 (0.0%)	60 (32%)	40 (22%)	8 (0.3%)
10-320-624	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	16 (28%)	3 (0.01%)
10-320-626	0.001 (0.0%)	0.001 (0.0%)	5.5 (6%)	27 (33%)	14 (0.3%)
10-320-628	0.001 (0.0%)	0.001 (0.0%)	23 (26%)	27 (28%)	27 (0.5%)
10-320-630	0.001 (0.0%)	0.001 (0.0%)	12 (21%)	25 (23%)	13 (0.3%)
10-322-624	0.001 (0.0%)	0.001 (0.0%)	4.5 (23%)	4.7(29%)	10 (0.2%)
10-322-626	0.001 (0.0%)	0.001 (0.0%)	0.5 (2%)	30 (24%)	11 (0.3%)
10-322-628	0.001 (0.0%)	0.001 (0.0%)	28 (34%)	28 (34%)	15 (0.4%)
10-322-630	0.001 (0.0%)	0.001 (0.0%)	6.5 (35%)	6.5 (35%)	6 (0.01%)
10-324-624	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	10 (40%)	6 (0.1%)
10-324-626	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	6.8 (7%)	2 (0.01%)
10-326-624	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	8.9 (28%)	2 (0.01%)
10-326-626	0.001 (0.0%)	0.001 (0.0%)	0.001 (0.0%)	29 (30%)	2 (0.01%)

(a) Before (b) After

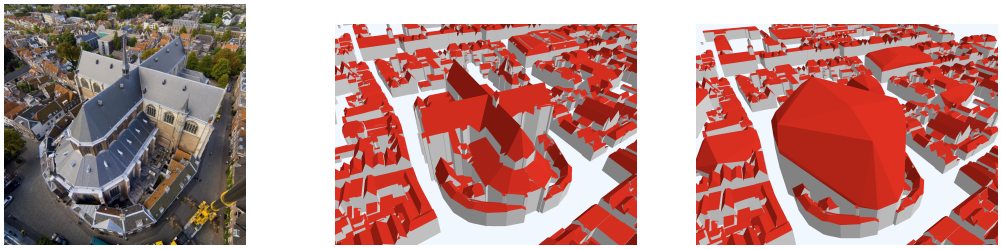
Figure 7.15.: 3D bag tiles before and after repair to visually see geometric difference

To Evaluate the geometric difference not based on points, the last column of Table 7.1 shows how often global repairs are used (which uses the points but usually changes the volume). Global repairs are rarely required in lower LOD's, and almost always come from acLOD 2.2. Considering the percentage of the primitives on which global repairs are performed, it seems negligible. Nonetheless, most global repairs are done on big buildings and well-known monuments. Two examples of this are the LUMC (Figure 7.16) and the Pieterskerk (Figure 7.17), which also in real life consist of lots of buildingparts "glued" together. In higher LOD's (above 2), these "glued" parts result in shell errors. The shell errors, which cannot be solved, consist of the shell not closed (302), Non-Manifold (303), and (self-intersections) 306. Not being able to close the shell is a result of the hierarchical order of shell repairs (further explained in Section 7.4.4) and being unable to solve tiny openings (around 10E-3 as explained in Section 6.2.2). Unable to solve Non-manifolds comes partly due to the "unrepairable non-manifolds" (as described in Section 7.4.2) and partly due to the hard-coded decision to keep the first part when there are no watertight parts (further discussed in Section 7.4.3). Unrepairable self-intersection results from the snap tolerance, which, likewise to Figure 6.9 sometimes, snap points back to the original location instead of the newly found non-intersecting location.



(a) Aerial photo (b) Before (c) After

Figure 7.16.: LUMC in Leiden



(a) Aerial photo (b) Before (c) After

Figure 7.17.: Pieterskerk in Leiden

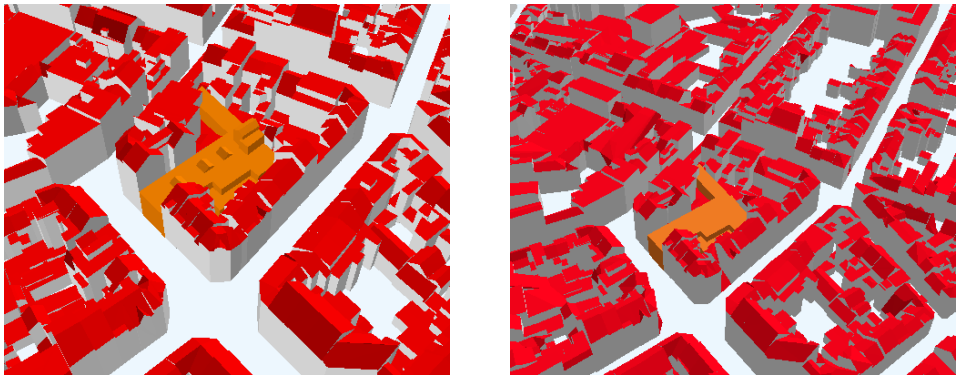
7.3.2 Brussel

Brussel also provides an open dataset of 3D buildings, similar to the 3DBAG. The 3D city model used to be available in CityGML format. However, it is not available for downloading anymore, so one of the old tiles is used. For this experiment, a found tile is converted to CityJSOn using CJIO. In contrast, the 3DBAG buildings are not Solid but MultiSurface. In addition to the many ring and polygon errors found, an additional complication with this dataset is the orientation of the surface normals (Figure 7.15a). MultiSurfaces have no orientation rules, so these problems will not be changed when repaired. To also repair these, the dataset should be repaired with `Orientation` or `Watertight`. To compare the different effects, both standards are tested.

At the start, only the ring and polygon errors already make the validity of the buildings in the dataset only 62%. In Table 7.2, you can see that the validity improves in all cases. In contrast to the 3DBAG, none of the datasets reach 100% validity, but the remaining errors are all caused by floating point errors due to conversion to int. The high Hausdorf is caused by the decision to keep the first found object in a Non-Manifold (303) and multiple connected components (305). For example, in Figure 7.18, a building is seen with small building parts on the roof. During the repair process, these parts are lost.

Table 7.2.: Result of Brussel tile after different repairs

Tile Name	**Validity after repair**	**Hausdorff Distance**	**Global repair used**	**Remaining Errors**
Default repair	99%	10 (10%)	23 (1%)	104, 203, 204
Orientation	99%	102 (80%)	237 (10%)	203
Watertight	99%	99 (78%)	83 (3%)	203, 302, 601



(a) Before (b) After

Figure 7.18.: Building in Brussel which loses details

The use of more global repair than the 3DBAG has two reasons. The first is that the data quality is lower at the start, which results in more repairs; these repairs add more points, which results in more floating point errors. Also, there are many duplicate vertexes (102) errors, which can't always all be solved before the `maxRepairDepth` is reached. The second reason is the conversion, which adds shell repairs to the repair process. Figure 7.19 shows one geometry converted to a solid as one outer shell. Seeing the building consisting of multiple buildings, there are almost always errors and a need to split geometries.

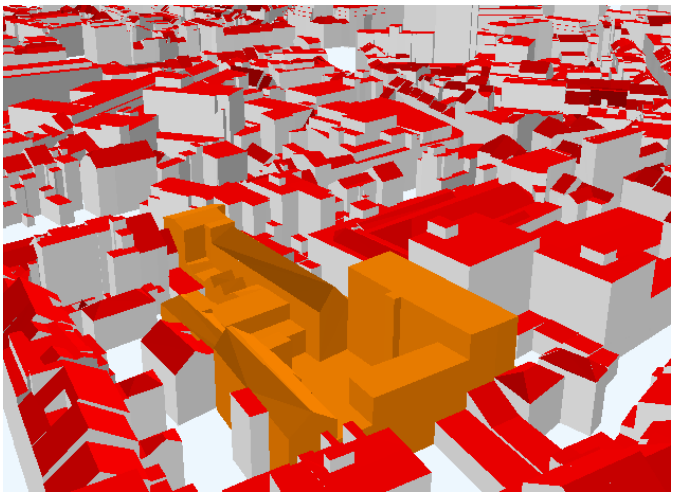
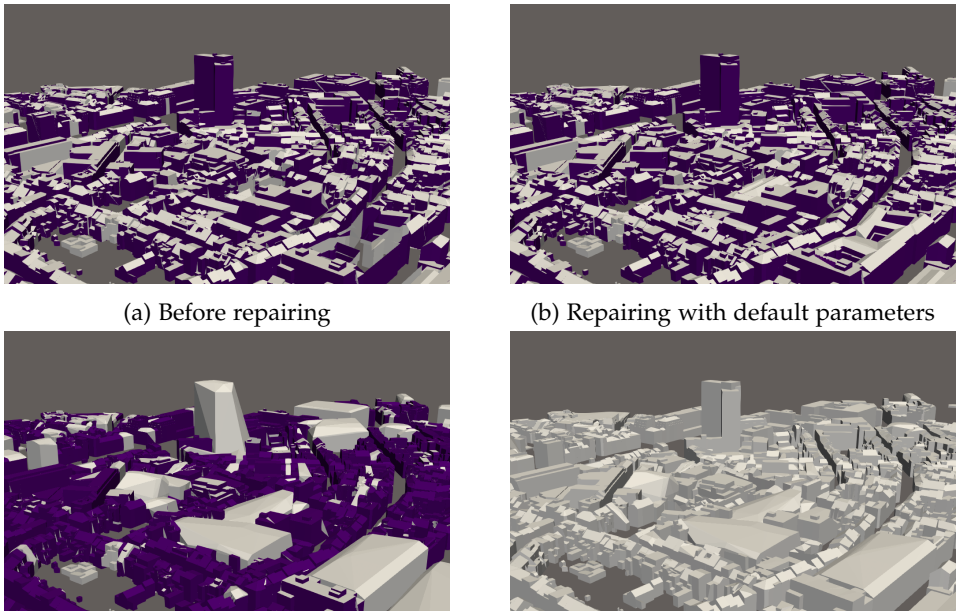


Figure 7.19.: Example of one building in Brussel data set (in orange one building geometry)

While visually comparing the output (Figure 7.20), the effect of the parameters can be seen. None of the wrongly oriented faces are flipped using the default repair parameters. However, most faces flip while using one of the parameters. As discussed in Section 5.4, the risk of the parameter orientation is that it uses the first face in the array to decide how the rest of the faces need to be oriented. Due to the high number of wrong orientations, the whole building is often inside out (Figure 7.20c). In that regard, The watertight parameter is more robust, as shown in Figure 7.20d.



(a) Before repairing

(b) Repairing with default parameters

(c) Repairing with Orientation

(d) Repairing with Watertight

Figure 7.20.: Repairing Brussel dataset with in purple the backside of surfaces

7.3.3 Data-sets CityJSON website

Lastly, some experiments were done to compare the effect of different use cases on a dataset. The example datasets from The CityJSON website are used. These datasets have a variety of geometry types and LODs. Table 7.3 shows each dataset its validity before the repair process. Its validity after the repair process and the Hausdorf distance (and the % of the geometry distance) are shown.

Table 7.3.: Repairing the example CityJSON datasets from their website

Dataset	Semantics	Geometric Validity Buildings Before	Repair Use Case	Geometric Validity Buildings After	Hausdorff Distance
3DBAG	True	98%	Default	100%	103 (34%)
			CFD	94%[2]	259 (85%)
			Energy demand	99%	259 (85%)
			Visualisation	100%	259 (85%)
			Solar power estimation	100%	270 (90%)

Continued on the next page

[2]Validated with overlap tolerance None due to segmentation error in val3dity

Table 7.3 – continued from previous page

Dataset	Semantics	Geometric Validity Buildings Before	Repair Use Case	Geometric Validity Buildings After	Hausdorff Distance
Den Haag	True	62%	Default	93%	0.1 (1%)
			CFD	59%[3]	15 (30%)
			Energy demand	99%	15 (30%)
			Visualisation	93%	0.1 (1%)
			Solar power estimation	93%	0.1 (1%)
Ingolstadt	True	70%	Default	99%	19 (30%)
			CFD	Segmentation[4]	Error[4]
			Energy demand	Segmentation[4]	Error[4]
			Visualisation	Segmentation[4]	Error[4]
			Solar power estimation	Segmentation[4]	Error[4]
Montréal	True	86%	Default	100%	0.36 (0.2%)
			CFD	98%	52 (33%)
			Energy demand	99%	167 (56%)
			Visualisation	100%	71 (90%)
			Solar power estimation	99%	71 (90%)
Railway	False	99%	Default	100%	0.03 (3%)
			CFD	50%	0.69 (72%)
			Energy demand	91%	0.69 (72%)
			Visualisation	100%	0.69 (72%)
			Solar power estimation	93%	0.69 (72%)
Rotterdam	True	76%	Default	100%	1.4 (2%)
			CFD	99%	59 (55%)
			Energy demand	99%	60 (55%)
			Visualisation	100%	60 (55%)
			Solar power estimation	99%	60 (55%)
Vienna	True	49%	Default	59%[5]	15 (36%)
			CFD	1%[6]	15 (36%)
			Energy demand	1%[7]	15 (36%)
			Visualisation	1%[8]	15 (36%)
			Solar power estimation	52%[9]	15 (36%)

[3] Validated with overlap tolerance None due to CGAL exception in val3dity, resulting in many 601 errors within tollerance

[4] Segmentation error in making of CGAL surface meshes

[5] validated with overlap tolerance None due to CGAL exception in val3dity, resulting in >120 601 errors within tolerance, 99% of the primitives are valid

[6] Validated with overlap tolerance None due to CGAL exception in val3dity, resulting in >200 601 errors within tolerance, 98% of the primitives are valid

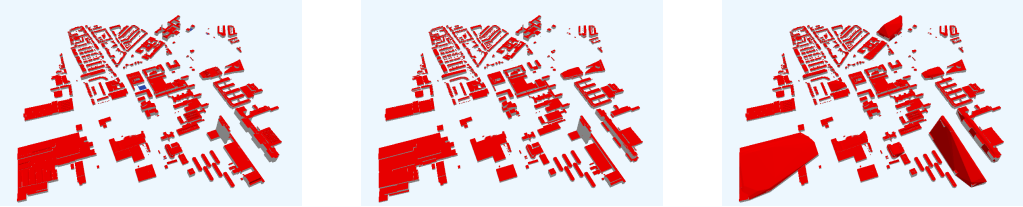
[7] Validated with overlap tolerance None due to CGAL exception in val3dity, resulting in >300 601 errors within tolerance, 99% of the primitives are valid

[8] Validated with overlap tolerance None due to CGAL exception in val3dity, resulting in >300 601 errors within tolerance 99% of the primitives are valid

[9] Validated with overlap tolerance None due to CGAL exception in val3dity, resulting in >120 601 errors within tolerance, 99% of the primitives are valid

The data shows that different repair use cases impact the geometric validity of buildings and the Hausdorff distance to varying extents. However, the general trend is that the geometric validity improves after repairs across all datasets and use cases. There is a noticeable correlation between CFD results and the lowest geometric val3dity. This is mainly because of the simplification and smoothing resulting in snap tolerance differences in the outputted model, which make a lot of shell intersections (306). On the other hand, the default parameters and visualization tend to yield the best results in validity, with the default yielding slightly better results regarding geometric differences. Likewise to the 3DBAG, global repairs are rarely required in lower LOD's and become more needed in higher detailed buildings.

3DBAG shows fairly good results concerning the geometric val3dity, but produces relatively high hausdorf distances. These results mostly stem from the decision to keep the first found object in a non-manifold (303) and multiple connected components (305). Figure 7.21 shows that three of the biggest buildings needed global repair, resulting in the lower LOD 1.3 giving a better result of the reality. These global repairs are needed in all the use cases.



(a) Before (b) After LOD1.3 (c) After LOD2.2

Figure 7.21.: 3DBAG results from use case EnergyDemand

Den Haag starts with one of the lowest geometric validity percentages (62%) and sees significant improvements post-repair. The Hausdorff distances remain relatively low (e.g., 0.1 in the default and visualization cases), which suggests repairs can be done very locally. This can also be seen in Figure 7.22, which shows no large wraps or convex hulls for global repairs.

Figure 7.22.: Den Haag Visualization without global repairs

Ingolstadt - The dataset is among the most detailed tested. However, the whole dataset is in MultiSurface. Therefore, errors such as missing window frames (Figure 7.23) will not be repaired unless watertightness is needed. Because the shell is not closed (302), it is accompanied by detriangulation; windows in the same plane as the wall disappear. In the tested version of AUTOr3pair, all the use cases except the default result in segmentation errors, which is likely a result of CGAL having problems with the complexity of the buildings. In the default repair, it is noticeable that with the detailed semantic information, it is better not to use validation of the semantics, seeing it downscales the semantic detail (Figure 7.24).

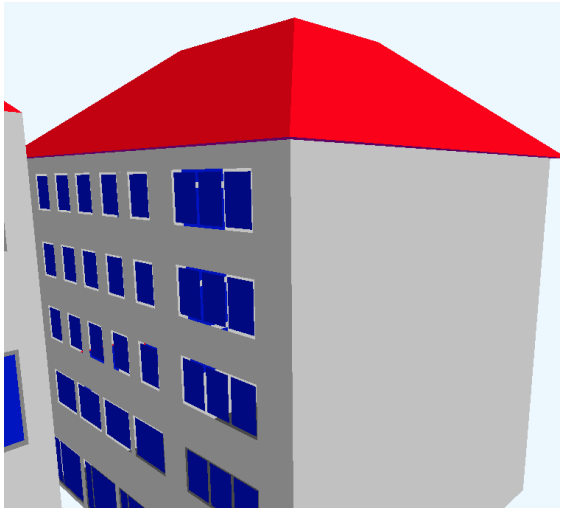
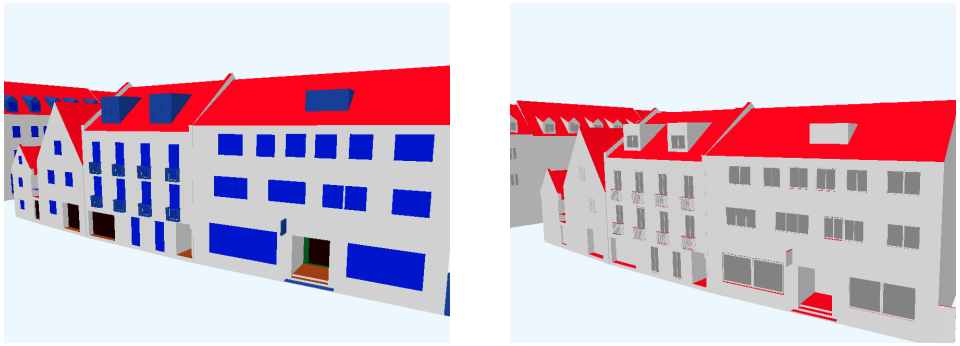


Figure 7.23.: Ingolstadt missing window frames



(a) Semantics before repair

(b) Semantics after validation

Figure 7.24.: Semantic validation on Ingolstadt

Montreal achieves high results in geometric val3dity by achieving (almost) 100% in all the use cases, with only a few global repairs (Figure 7.25). However, it does show a high geometric difference; in Visualization and Solar power estimation, these are the result of a deleted part, but in Energy damnd, that building is wrapped, but it seems to take a wrong point resulting in the trunk going to a point outside, the building (Figure 7.26a). Also, Montreal is the first data set showing an error regarding preserving the semantics; the building shown in Figure 7.26b shows that it has a wall surface on the place where the roof should be.

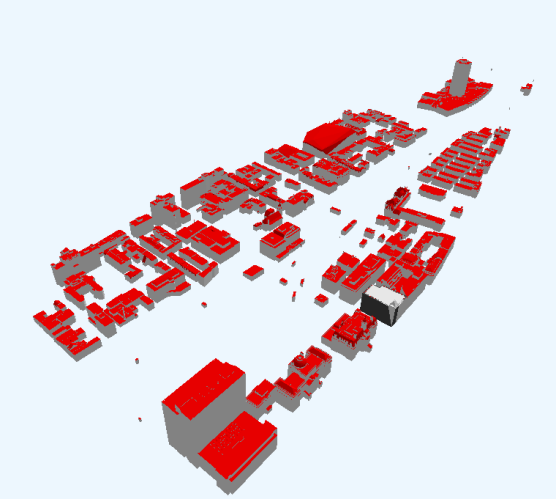
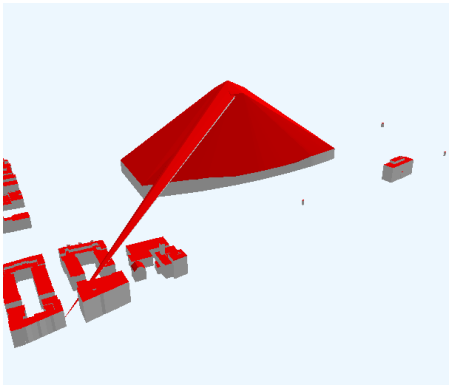
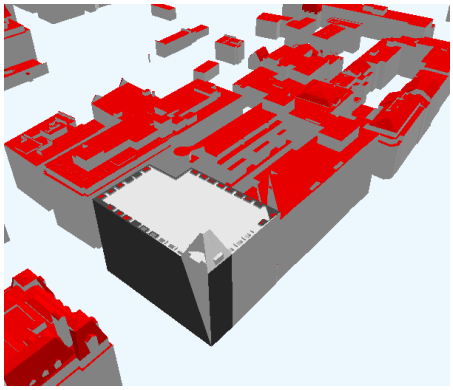


Figure 7.25.: Result from repairing Montreal



(a) Point problem in Energy Demand



(b) Semantics problem in Solar power

Figure 7.26.: Errors after repairing Montreal

Railway starts with a near-perfect geometric validity (99%). However, use cases (like CFD) reduce its validity significantly (to 50%). Also, while there are only 50 buildings (parts) in the file to repair, visually questionable repairs are done. For example, Figure 7.27 shows how a building ends up with a missing wall, and Figure 7.28 shows how missing walls are solved with a graceless wrap, losing all detail in the building.

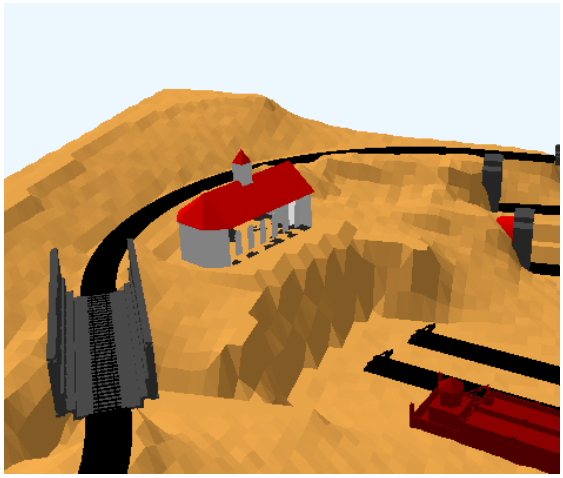
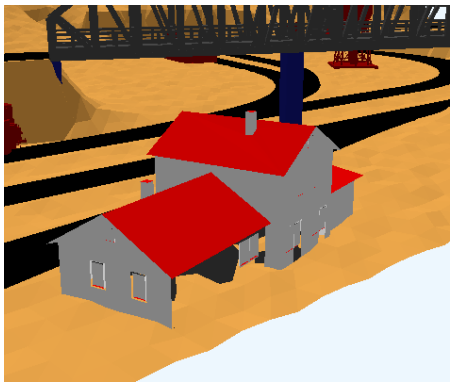
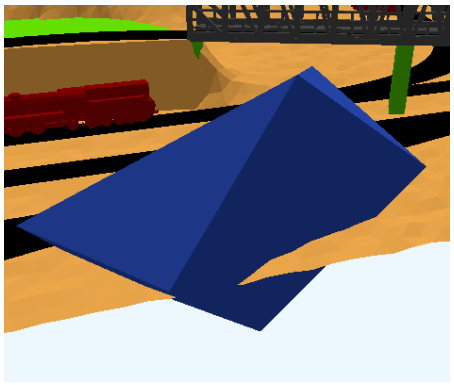


Figure 7.27.: Building missing a wall



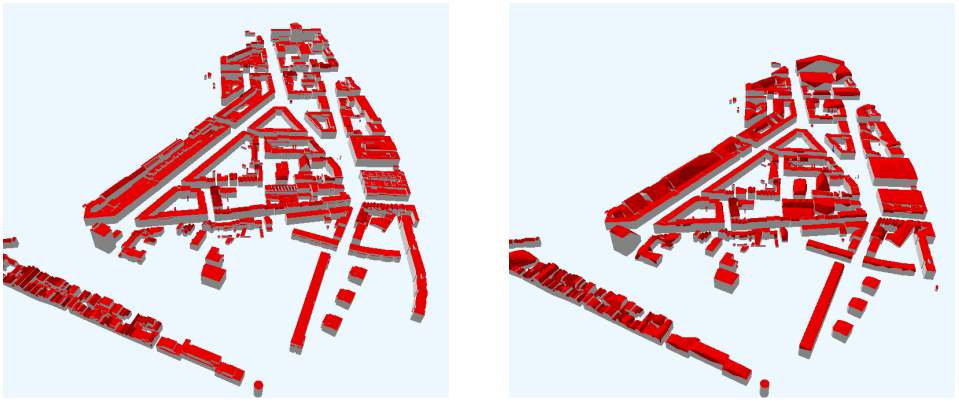
(a) House without wall in Visualization



(b) Wrap in Energy demand

Figure 7.28.: Doing unrealistic wrap on building in Railway

Rotterdam produces improved validity to (almost) 100% on all the use cases. However, in contrast to the other datasets, many global repairs are used. As shown in Figure 7.29, these global repairs are less noticeable in comparison to, for example, Figure 7.21. The many global results from Non-manifold vertices in Composite solids, which are unrepairable due to being one object (further discussed in Section 7.4.2.



(a) Before (b) After

Figure 7.29.: Many compact global repairs in Rotterdam

Vienna shows very low validity after repairing, primarily resulting from features with 601 errors ($>$ 120 in Default and visualization, $>$ 300 in Solar Power and Energy demand). Due to a bug in val3dity, they are checked with an overlap tolerance of None, while they are repaired with an overlap tolerance of 0.001; during the repair process, val3dity suggests that they are repaired. Vienna does, however, vastly improve the validity of the primitives, which is around 95% at the start and almost 100% at the end. Figure 7.30 shows the result after energy demand (which is only 1% valid), which shows no visible signs of being completely invalid.

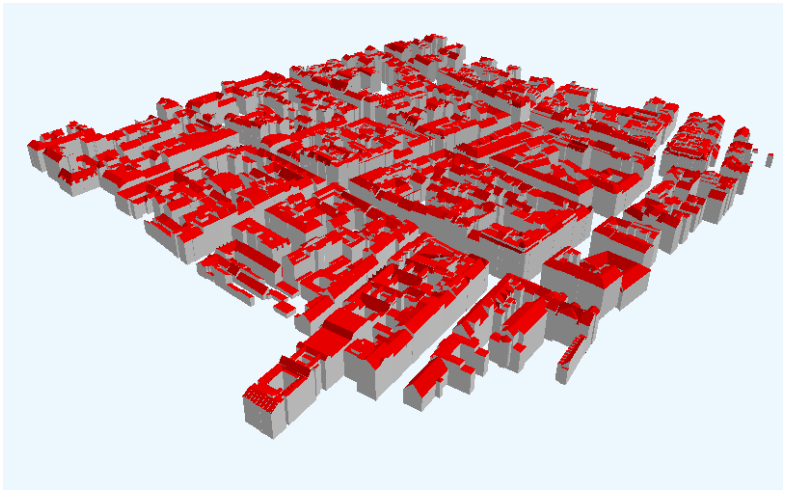


Figure 7.30.: Vienna after Energy Demand repair

7.4 Discussion

While AUTOr3pair offers robust solutions for common errors in 3D city models, several aspects of its design and functionality may lead to unintended results or reduce usability. The following sections will discuss some flaws of AUTOr3pair.

7.4.1 Specific repair situations

AUTOr3pair uses strategies to address common errors in 3D city models. These strategies work for common situations; however, they sometimes result in (maybe) unwanted results in certain error situations. For example:

- **104 - Using outliers for the repair:** When the convex hull of a ring is made, all vertices are considered, which could result in using outliers as a base. Using a Concave hull instead of a convex hull could capture more intricate details of a ring. For example, it could help get rid of dangling pieces; however, setting the `alpha`-value needs tuning for every ring of every dataset, therefore requiring manual help. Additionally, using *Prepair* (Ledoux et al., 2014) could provide different results by offering more automated solutions for repair, but it requires the use of WKT, introducing an additional geometry conversion step.
- **201 - Deleting the Outer Boundary:** When an inner ring is the same (or almost the same by snapping), The used approach results in deleting the outer boundary of a polygon and, therefore, the whole polygon. Losing an entire polygon could be unwanted, especially for MultiSurfaces and CompositeSurfaces, which do not have closing shells as a repair, which can bring back polygons.
- **205 - Keeping small polygons:** When splitting a polygon, seeing its interior is disconnected, all parts are kept. However, this could result in small (sliver) polygons. Using a threshold could be a solution; however, setting an appropriate threshold is possibly different per model, as it directly impacts the accuracy of the repairs and would, therefore, need manual help.
- **401 - Deleting the Outside Shell:** When an internal shell is the same (or almost the same due to snapping), the used repair strategy deletes the outside shell to preserve the model's validity. However, this can result in the loss of essential geometries.
- **404 - Keeping small shells:** When splitting a shell, seeing its interior is disconnected, all parts are kept. However, this could result in shells with a shallow volume. Using a threshold could be a solution; however, setting an appropriate threshold is possibly different per model, as it directly impacts the accuracy of the repairs and would, therefore, need manual help.
- **503 - Snapping Close Volumes:** The used repair strategy splits the volumes into separate parts. However, snapping together volumes could be a more logical solution when two volumes are close but not perfectly aligned. However, the snapping distance needs to be carefully managed to avoid unintended distortions; the snapping approach is not implemented.
- **503 - Not filling inner holes when merging solids:** The used repair implementation for intersecting solids doesn't fill inner holes when merging. This should happen when following the SetDiff paradigm; however, to preserve geometric detail, this is not done.

7.4.2 Unrepairable non-manifolds

Manifolds similar to the topology shown in Figure 7.31 cannot be repaired. The methodology uses the splitting of objects on the non-manifold edges or vertices. When the non-manifold edge/vertex is part of the same "volume", it cannot be split. This results in needing global repairs. Alphawrap can also not repair this case, seeing it has the same non-manifold edge/vertex. A solution for this would be to use a bigger offset. However, this changes the geometry in places where it was correct. Therefore, a convex hull is mainly used, which loses the inner "hole" of the torus shape.

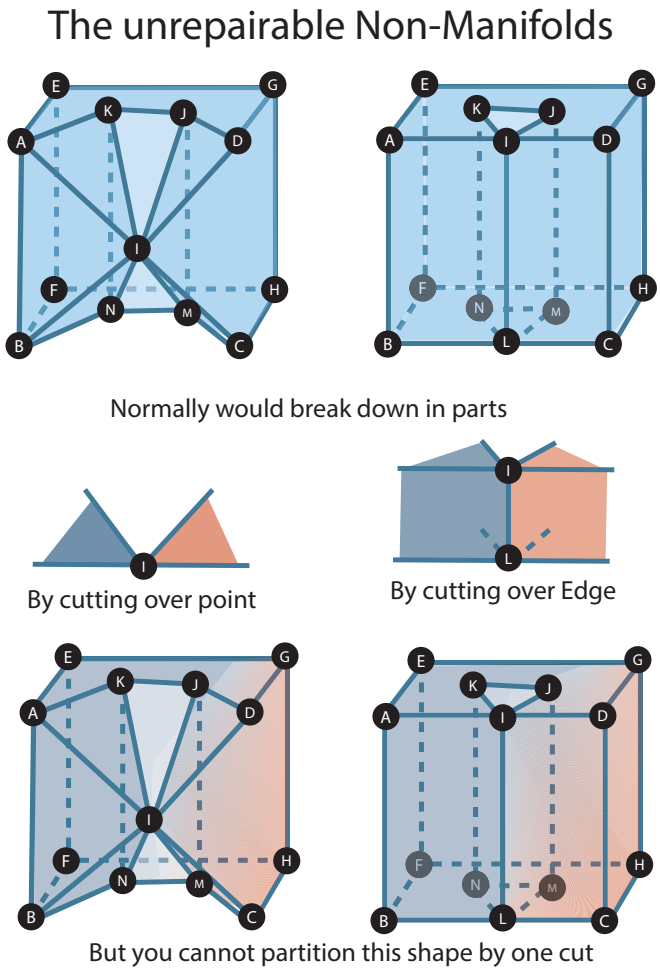


Figure 7.31.: Unrepairable non-manidfolds

Solutions to repair this more locally would be:

- Making an extra (plane) cut somewhere in the geometry, which results in 2 parts, but the algorithm for where the second cut needs to come should be well thought out always to yield the same results.
- splitting the nonmanifold edge/vertex into two parts and moving those away from each other. This will solve the Non-Manifold problem, but where the edges/vertices need to

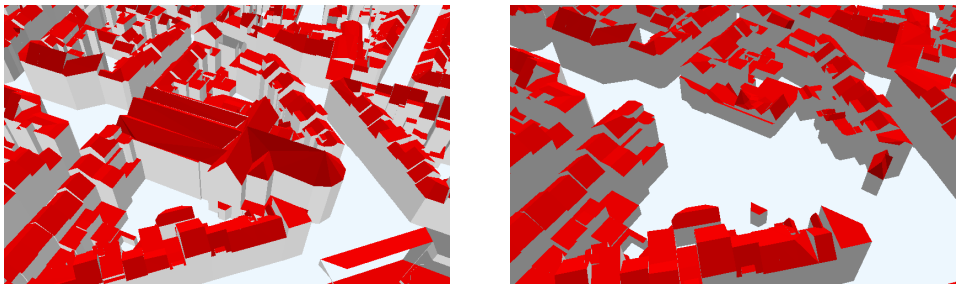
move should be a well-thought-out algorithm. Moving could result in different topologies. For example when moving the non-manifold edge from Figure 7.31, it could move more to the inside, so it will become a hole, which doesn't touch the outer boundary, or it could move the edge slightly apart to open the hole.

- The most robust solution would be to divide the volume in tetrahedrons, likewise to using triangulation on surfaces. However, this will split the building into many parts and introduce many new faces, which may not be optimal for all use cases.
- Lastly, the cutting and stitching method from (Gueziec et al., 2001) (discussed in Section 3.2.1 could be used as a repair approach for this kind of manifolds.

7.4.3 "Hardcoded" repair decisions

The AUTOr3pair framework incorporates several "hardcoded" repair decisions to streamline the process. For instance, when encountering duplicate geometries or overlapping features, the framework is programmed always to retain the first instance and discard subsequent ones. While this approach simplifies decision-making during repairs, it can also lead to unintended consequences, such as losing potentially critical geometric details or user-preferred configurations. Also, using the same parameters as thresholds and predefined snapping distances may not benefit all geometries. Though effective for specific buildings, others would not have the best repair. These choices limit the repair process's flexibility and may require user intervention to ensure the desired outcomes.

The hard-coded decision that is most striking for the geometric difference is to keep only the first part when none of the parts have a volume and directly delete parts without a volume. This is done in Non-manifold (303) and multiple connected components (305). For big buildings, which consist of numerous "glued" parts, these often result in parts of the building being deleted. For example, Figure 7.32 shows a church whose most prominent part is deleted during the repair process, resulting in only a small side part remaining.



(a) Before (b) After

Figure 7.32.: Deleting a big not watertight part

7.4.4 Dependence on val3idty report schema

The AUTOr3pair framework relies heavily on val3dity and its report structure to validate the file and identify errors and their locations (Section 4.1). When the structure of the val3dity report changes by, for example, changing the schema or when the ID of the error changes, AUTOr3pair will not get the information it needs for the repair process. Therefore, changing the

val3dity version requires corresponding adjustments in AUTOr3pair to maintain its functionality. This results in strict version dependencies, which make the program less user-friendly, and when val3dity upgrades, AUTOr3pair will also need to upgrade.

Also, output choices and/or bugs of val3dity affect the repair process of AUTOr3pair. For example, when a 302 (shell not closed) error is found, it can, depending on the number of "holes" report the same error more than once (Figure 7.33). This first resulted in repairing the same shell twice and adding it twice to the geometry before finding out this bug.

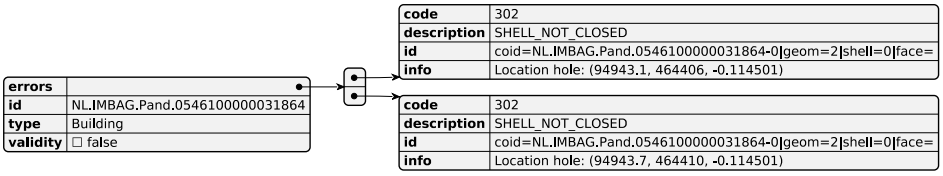


Figure 7.33.: val3dity reporting 302 error two times

Also, the hierarchical order of val3dity influences the repair process as mentioned in Section 6.2.2 shell not closed (302) can not be repaired when there are shell self-intersecting problems (306). However, when holes are filled, AUTOr3pair doesn't know yet if the shell has self-intersections, resulting in a repair loop that ends in a global repair.

7.4.5 Usability of repair report

While the repair report provides valuable insights into the repair process, it has several shortcomings. Firstly, when vertices are altered or removed due to being orphaned, the report cannot reconstruct the boundaries using the output point set. One potential solution is to output the boundaries as the actual vertices instead of their indexes, significantly increasing the file size. Alternatively, the entire list of vertices could be added to the report before cleaning the orphaned vertices, but this approach still fails to indicate when and which points are projected. Secondly, for 3D city models requiring extensive repairs, the report file can grow up to four times the original model's size, making it difficult to handle in terms of memory. Lastly, when a geometry is split into multiple geometries or change types, the user must find which of the primitive(s) is/are the next to follow for the repair trail.

7.4.6 User friendliness

The current AUTOr3pair framework, while powerful, is not particularly user-friendly. Its command-line interface and reliance on technical knowledge make it challenging for non-developer users to navigate and utilize effectively. Building and running the C++ is no easy task, primarily due to installing prerequisites from CGAL libraries. Using a containerization tool would have helped streamline this process and simplified deployment. The lack of a graphical user interface (GUI) limits the ability to visualize the repair process, hindering understanding and slowing the workflow. This results in the program being less accessible to a broader audience.

7.4.7 Losing original groups OBJ

When using AUTOr3pair to process OBJ files, the original grouping information within the OBJ format is lost during the repair process. This can be problematic for users who rely on these groups and their names to manage complex 3D city models. The repair process

will disrupt the structure and hierarchy initially defined, resulting in loss of information, particularly in applications where the grouping of objects is essential for tasks like rendering, simulation, or further editing.

7.4.8 Testing preserving of semantics

Currently, no automated method exists for testing the preservation of semantics during model repairs, making manual verification necessary. This manual process involves carefully inspecting repaired models to ensure that semantic data, such as object classifications and attributes, remain accurately aligned with their corresponding geometries. However, manual testing is time-intensive and prone to oversight, especially in complex models.

7.4.9 Need for generalization for CFD

As explained in Section 5.2, applying simplification through the generalization of buildings and their footprints would be highly effective for addressing small gaps between buildings. Currently, the simplification parameter focuses solely on individual objects without considering their impact on the surrounding environment. To implement this, a function is required that identifies and simplifies geometries based on predefined distance and angle thresholds between surfaces. This function would merge or remove minor surfaces that meet these criteria while maintaining key structural elements.

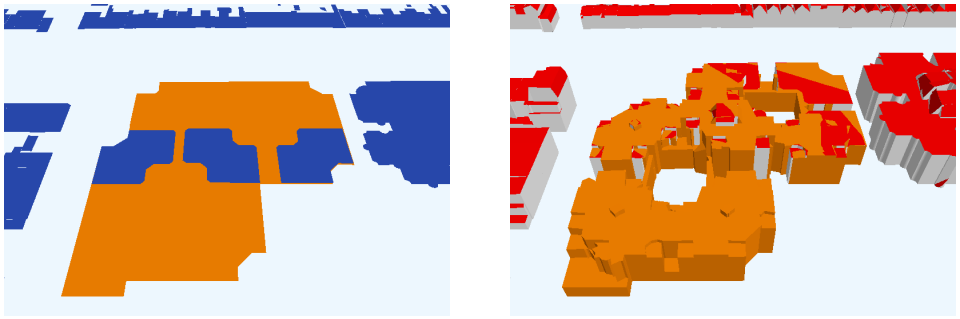
7.4.10 Repair intersection between geometries

The overlap parameter is not functional at this moment. It should control the detection and repair of overlapping surfaces between different geometries, even when they are of varying geometry types. The original approach was to use CGALs AABB tree computations (Alliez et al., 2023e) for finding intersections. The AABB tree allows for logarithmic time complexity for many geometric queries by narrowing down potential intersecting areas through bounding boxes. Combining triangles from different surface meshes into a single collection, the AABB tree treats them as one geometric space, which helps with finding intersections faster.

Finding intersections was no problem; however, when intersections were found, the repair process raised some questions that need further research. Geometrically, intersections between "watertight" parts can be done using the same approach as building parts intersect (601), and intersections between two surfaces can follow the approach of the intersection of rings (201) when they are on the same plane. When a surface intersects with another surface on a different plane, the solution becomes less straightforward, as you would need to cut parts of the surfaces, raising the question of whether to keep all segments (allowing them to be split) or retain only one part—and if so, which part to keep. And the same kind of problem would happen when a surface intersects with a "watertight" object.

Also, what to do with the semantics and attributes of the object is a point of discussion. For example, building NL.IMBAG.Pand.0546100000041634 (Figure 7.34) in the 3DBAG intersects with five other buildings (In Figure 7.34a, only three can be seen, the other two are stacked below the big footprint). When only looking at the footprints, it would seem logical to merge them into their own, but this loses all the attributes from the three other buildings connected to the footprints. So, a more rational approach would seem to be to carve out the footprints from the big footprint. But that approach doesn't work when two footprints slightly overlap; how can it be decided which will get the overlapping part? When looking in 3D, this decision-making only becomes messier. How can you determine what part belongs to which building, or should it just merge everything into one object? In both situations, which semantics and materials are preserved and which are discarded? Also, the question merge approach has

multiple options: should everything be merged into one volume, or should the building become separate building parts glued together by their parent-child relationships? Also, again, the question can be raised: does this approach work the same for two buildings that slightly overlap?



(a) Footprint (b) LOD 2.2

Figure 7.34.: Overlap in 3DBAG

8 Conclusions

The main objective of this thesis was to *Develop a framework for the automatic repair and reconstruction of 3D city models to facilitate different use cases and implement a prototype.* The made framework AUTOr3pair serves as the proof of concept for this prototype. As shown with the experiments (Chapter 7), AUTOr3pair repairs geometric errors to be valid according to the ISO 19107 standards. The framework also integrates additional requirements for use cases, as outlined in Chapter 5. and is implemented as explained in Chapter 6. In this chapter, the thesis research questions are reviewed to assess the research behind the implemented prototype, AUTOr3pair. Based on this review, the contribution to the current state of the art and the limitations of the used approach(es) are presented. Lastly, based on the limitations, future work is recommended.

8.1 Research overview

To review the result of the research objective set in Section 1.3, each research question will be answered with short answers based on the evidence presented in the previous chapters. Furthermore, the answers will highlight the research contributions to developing the implemented prototype.

What is needed to achieve geometric validity? - Achieving geometric validity requires adhering to the ISO 19107 Standards. In the ISO standards, each geometric primitive has its requirements, and for a primitive to be valid, all its lower-dimensional primitives must be valid. Therefore, validity also needs to be achieved in a hierarchical order.

How to achieve geometric validity using automatic repair? - Achieving geometric validity using automatic repair can involve local and global repair strategies. Local repairs are used for specific repairs and, therefore, are less robust but preserve the model better. On the other hand, global repairs work for all kinds of repairs, but they significantly alter the original geometry. With the help of val3dity, validity errors can be identified, and the correct local repair can be used. Identifying and repairing errors hierarchically ensures the least geometric difference between the outputted model and the original. Global repairs can form the safety net to achieve geometric validity when local repairs are insufficient.

Is it possible to achieve geometric validity using automatic repair? - Yes, the experiments demonstrate that 100% validity is achievable but that global repairs are needed for some geometries. Based on the original data quality and LOD, how often global repair is required differs. At this moment, not all datasets can achieve 100% validity, partly due to floating point errors and partly due to data quality.

How to preserve semantics during automatic repair? - Semantics and materials can be preserved when they keep their link to the face (polygon). This is done by mapping the information to the face instead of using the location of the face in the boundary. Using the SetDiff paradigm, newly formed faces can be relinked to their original semantics if they are completely or partly inside the original face.

How to achieve validity for different use cases? - Achieving validity for different use cases involves tailoring the automatic repair process to meet specific application requirements. The requirements for CFD, energy demand, visualization, and solar power estimation have been explored, and with the help of parameters, the repair process is adjusted. Parameters can affect pre and post-processing around repairs and help make repair decisions, resulting in the usability of the 3D city model for a range of applications.

What degree of validity can be achieved? And to what extent does this improve current 3D city models? - The degree of validity achieved through automatic repair is highly effective. AUTOr3pair demonstrates an ability to repair almost all the errors that val3dity identifies. The improvement to current 3D city models is substantial, as the repaired models are made more reliable and usable for various applications, which minimizes manual pre-processing. Additionally, maintaining semantic consistency throughout the repair process ensures that the models are geometrically valid and retain their functional utility across different use cases. Overall, the framework significantly enhances the quality, reliability, and application range of 3D city models.

In summary, this thesis successfully addresses the main research questions and demonstrates the feasibility and effectiveness of an automatic repair framework for 3D city models by providing the prototype AUTOr3pair. The findings emphasize the potential of automated processes to enhance the quality and usability of 3D city models, paving the way for more efficient workflows in various geospatial applications by taking over the data pre-processing step.

8.2 Contributions

Throughout this thesis, the methodology formulates repair concepts. While most of these concepts are not new, the combined approach offers valuable contributions to the current state of the art. The most significant contributions include:

- **Repairframework focussed on local repairs** - Compared to other automatic local repair frameworks, AUTOr3pair is more robust due to its approach of error identification and matching the best local repair. The use of val3dity to identify errors hierarchically ensures that all errors can be resolved locally, minimizing alterations to the model. This allows for less geometric difference and better preservation of the original structure. The global repair safety net ensures that the 3D city model will still be outputted with geometric validity when local repairs are insufficient. This dual-step strategy, by prioritizing local repairs but providing global repairs as a fallback option, enhances the reliability and flexibility of the repair process. This improves the ability to maintain geometric accuracy during the repair process.
- **Preserving of semantics** - Preserving semantics is a key advancement in AUTOr3pair. Traditional methods solely focus on repairing geometric defects. AUTOr3pair ensures that semantic information (and materials) remain linked to their corresponding faces throughout the repair process. The framework ensures that repairs, whether involving face splitting, merging, or flipping, do not disrupt the semantic integrity of the model, allowing the model to remain functional for use cases that rely on semantics data. This dual focus on geometry and semantics marks an important evolution in the field.
- **Userinput on repair process** - Allowing user input to influence the repair process in AUTOr3pair adds significant flexibility and enhances its contribution to the state of the art. By providing adjustable parameters—such as allowed geometry types and val3dity

tolerances—users can fine-tune the process to meet the specific needs of their use case. These parameters enable users to control repair decisions, thus optimizing the repair for use cases of choice. Additionally, AUTOr3pair includes premade sets of parameters for CFD, energy demand, visualization, and solar power estimation based on research, allowing users to apply optimal settings for these specific applications. This customization ensures the repaired models are better suited to the specific use case requirements, making AUTOr3pair more versatile and applicable.

- **Detriangulation of (repaired) geometries** - Detriangulation of repaired geometries significantly advances the state of the art by addressing the limitations of working on triangulated meshes. Most existing repair methods focus on triangular meshes, which require converting the model into a triangular format and then back into the original geometry after the repair, a process prone to errors. By implementing detriangulation before the output, AUTOr3pair bypasses this error-prone conversion step and preserves the model's original structure more effectively. This approach ensures that the repaired geometry retains its geometric integrity without needing further modifications, such as semantic or material adjustments, that might otherwise occur during re-meshing. As a result, detriangulation in AUTOr3pair reduces data loss and enhances model precision, providing a reliable repair process across various use cases.

8.3 Limitations

Apart from the contributions to the current state of the art made by the methodology en de prototype, some limitations have also been identified. These limitations, which impact both the robustness and usability of the approach, are addressed in this section.

8.3.1 Global repairs are needed

Although the automatic approach focuses on local repairs, global repairs are still needed to reach 100% geometric validity. These global repairs led to significant issues in buildings' geometric integrity and visual quality, as shown in Figure 7.16 and Figure 7.17. A few reasons why global repairs are needed are needed are:

Combination of self-intersection and nonclosed shell - The repair approach chosen for nonclosed shells (302) uses a CGAL method, which doesn't work when there are, or if it creates self-intersections. Figure 8.1 shows an example of a geometry that would need a global repair.

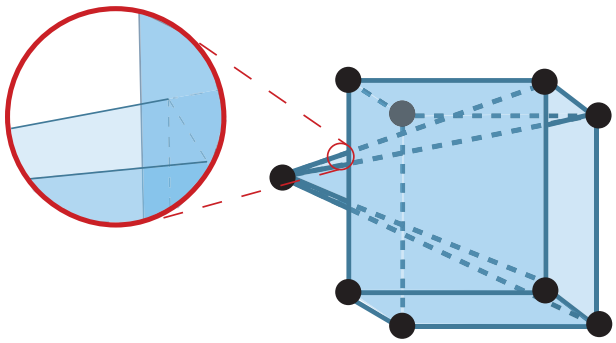


Figure 8.1.: Global repair needed when nonclosed shell self intersects

Holes that are too small - The repair approach chosen for nonclosed shells (302) uses a CGAL method, which cannot fill tiny holes. However, different methods have the same problem. When, for example, geometries have a topology similar to Figure 8.2. Filling the hole results in (almost) colinear triangles, which give segmentation errors in CGAL's surface meshes and CGAL's Nef polyhedron, making AUTOr3pair unable to finish the repair process.

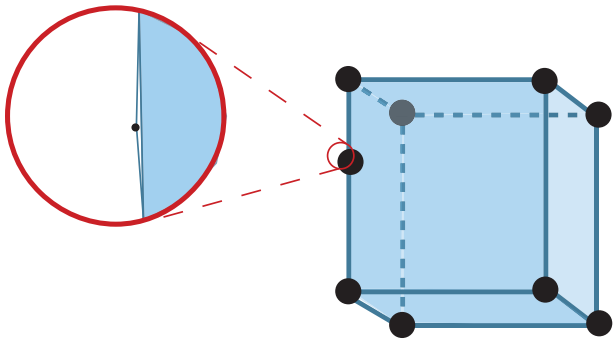


Figure 8.2.: Global repair needed when nonclosed shell opening is very small

Unrepairable non-manifolds - As discussed in Section 7.4, non-manifolds with topology consisting of one volume (Figure 8.3) cannot be repaired with the chosen repair approach. Therefore, these non-manifolds need Global repairs.

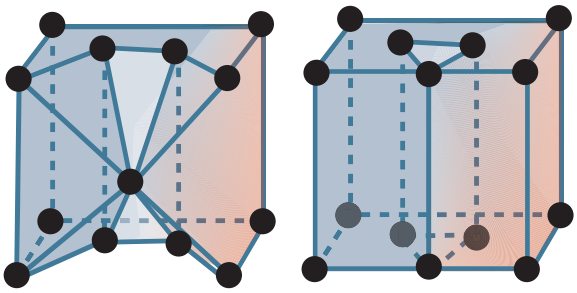


Figure 8.3.: Global repair needed when unrepairable non-manifold

Snaptolerance - Although during the repair process, there are measures taken. For example, by having the same overlap tolerance (Section 6.3) and projecting vertices when nonplanar polygons are found not to create Figure 6.3. It could still be argued that repairing with a snap tolerance is never a good idea. In the experiments, geometries sometimes still ended up with, for example, ring or shell self-intersection (104 and 306) due to snapping to old points (Figure 8.4).

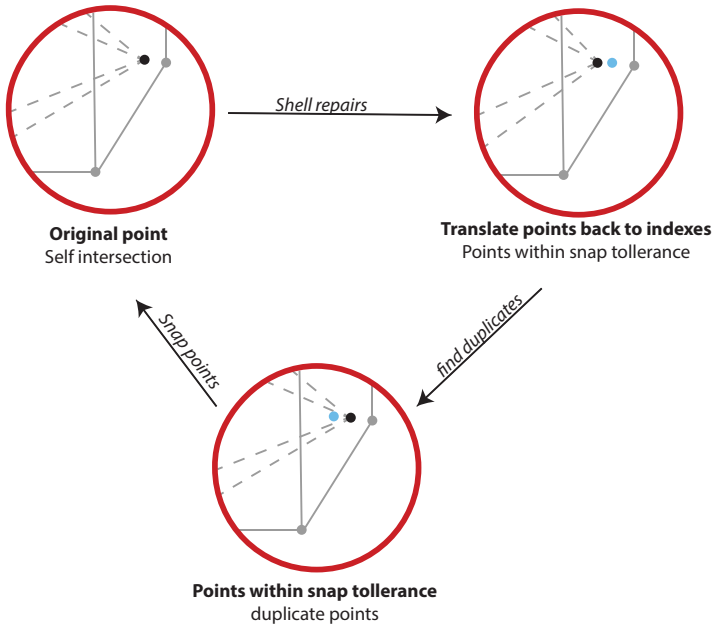


Figure 8.4.: Global repair needed when unrepairable non-manifold

Skiplowrepairs - Although this parameter makes the repair process for most geometries much faster, it results in repair loops for some geometries (Figure 8.5). This happens when a face with an error is readded in shell not closed (302) repair. Re-adding a wrong face can occur due to snap tolerance, when points are being snapped wrongfully to create, for example, 104, 203, or 204 errors.

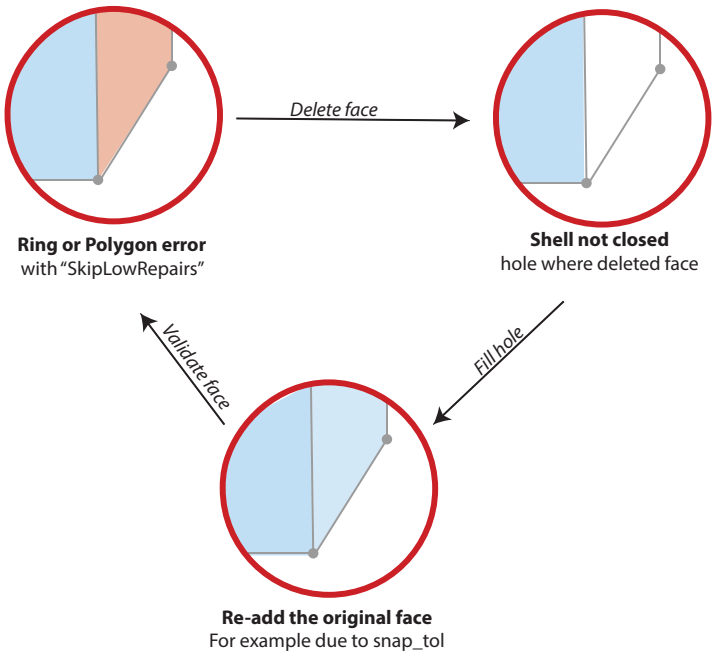


Figure 8.5.: Global repair needed when unrepairable non-manifold

8.3.2 What to keep

One notable limitation in the repair framework is the hard-coded approach for handling non-manifold (303) and multiple connected components (305) errors. When objects lack volume, AUTOr3pair automatically retains only the first part and deletes subsequent ones. While simplifying the repair process, this approach can result in unintended deletions. As discussed in Section 7.4.3, this mostly happens in large composed buildings that consist of numerous interconnected parts. For example, Figure 7.32 shows how such a building loses a substantial part of its structure. Although smaller, similar challenges happen also in smaller formats. Figure 8.6 is an outputted building of the de Ingolstadt dataset, where part of the outer boundaries were deleted. Although the building ends were valid, the repair approach compromised the building's geometric accuracy.

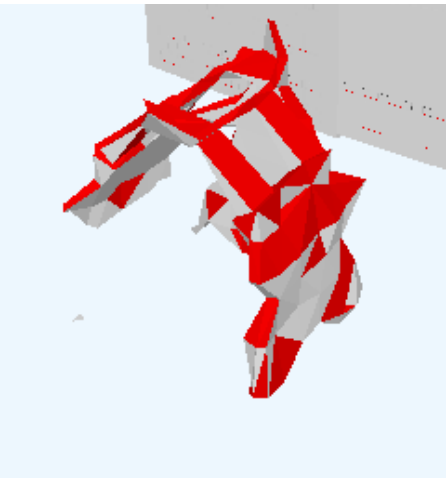


Figure 8.6.: Valid building, which lost all its geometric intergrity

8.3.3 OBJ non-repairs

OBJ validation and repair are limited when fixing non-manifold geometries. For instance, splitting a non-manifold object composed of two cubes (Figure 8.7) into separate objects may resolve the initial problem, but upon revalidation, the same non-manifold issues reappear due to OBJ's file structure, which does not oblige faces to be grouped in geometries. The decision of AUTOr3pair to make the groups with finding connected components limits the repair process of non-manifold geometries, seeing each time the outputted file is repaired, it will output the same results.

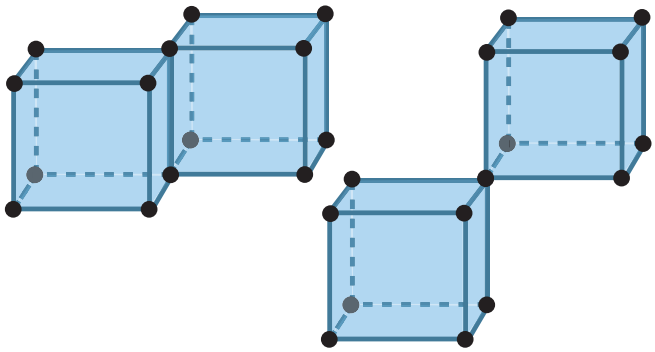


Figure 8.7.: Non-manifold examples which don't repair

8.3.4 Decisions per object

While tuning parameters can help refine the repair process for specific use cases, optimizing parameters on a geometry or group basis would further enhance the output quality. Such an

approach allows tailored repairs that consider each object's characteristics, improving the usability of the outcomes for use cases. Applying individualized parameters per object increases manual workload, which demands categorization or active user input for parameter assignment. While this approach requires more effort, the potential benefits would be substantial.

8.3.5 Floating point errors

Floating-point errors are always a risk, mainly when working with automatic processes. In the experiments, most output errors resulted from floating-point inaccuracies. Although the approach of changing the scale when translating the vertices to integers (as discussed in Section 6.2.3) tackles a lot of floating point errors, some inaccuracies still seem to be inevitable, especially in highly detailed models.

8.3.6 File sizes

Repairing large datasets presents significant challenges due to memory and CPU constraints, making the process very time-consuming. For efficient processing, the maximum suggested file size for quick repairs is 10 MB, with tests indicating manageable repair times of up to 25 MB. Beyond this threshold, using all the vertices demands substantial memory, and repair operations become markedly slower. This comes mostly due to the time constraint of the kd-tree for the vertices, which is, on average, O(n log n), which, in the worst case, leads to O(n) if the tree becomes unbalanced. Users aiming to repair larger datasets could better split their files using tools like cjio than attempting to process the entire dataset simultaneously, as this approach maintains efficiency and reduces resources.

8.3.7 Texture deletion

In this framework, textures are removed from repaired objects, as texture handling was beyond the scope of this thesis. This limitation affects the visual quality of 3D city models, particularly for applications that rely on accurate texture mapping. Addressing this would require methods to track UV positions and adjust textures in alignment with geometric repairs, an area for potential improvement in future work.

8.4 Recommendations for future work

Several research and development opportunities are recommended to develop further and enhance the AUTOr3pair framework. These recommendations focus on expanding the tool's capabilities, increasing its versatility, and improving its overall user experience. Critical areas for future work include supporting additional file formats, integrating val3dity more seamlessly, extending semantic validation, preserving textures during repairs, and developing a comprehensive 3D GIS application.

8.4.1 More input and output file types

Supporting additional file formats of 3D city models would increase the usability for preprocessing for more target platforms. Notably, since AUTOr3pair already uses the tu3djson format internally, the support of more data file formats would be researching and developing converters for features/geometries to tu3djson. The internal use of tu3djson format was done to minimize the complexity of integrating new formats and ensure the repair process will stay the same on different file types.

8.4.2 Additional repair for more use-cases

To further enhance the automatic repair framework, AUTOr3pair, future work should focus on incorporating additional use cases. From the 29 use cases described in Section 5.1, only four are implemented. Expanding the range of supported use cases would make the framework more usable for a wider audience. New use cases could come from tuning the existing parameters and researching and developing additional repair methods tailored to the unique requirements of these new applications. Lastly, integrating feedback from new use cases could refine existing methods and inspire new innovative repair techniques.

8.4.3 Intergrating val3dity and AUTOr3pair into one tool

As explained in the discussion, AUTOr3pair is heavily dependent on val3dity's report structure (Section 7.4.4). Therefore, integrating val3dity and AUTOr3pair into one combined tool would tackle the schema dependence, ensuring seamless interaction between validation and repair processes. Also, combining these tools would optimize memory usage and computational resources, as both would utilize the same internal framework. This would streamline the workflow and improve the repair framework efficiency by reducing memory needs for internal data structures and exchange between those.

8.4.4 Automatic validation and repair for more semantic values

As explained in Section 3.4, existing semantic validation and repair methods can not validate all existing semantic values. AUTOr3pair only validates and repairs three types of surfaces,`Roofsurface`, `wallSurface`, and `FloorSurface` (Section 6.2.2). More research should be done to enhance validation and repair for semantic values further. The possibility of validating semantic values of doors and windows could help automatic repair for use cases such as Routing and Navigation.

8.4.5 Validation for preserving of semantics

As discussed in Section 7.4.8, the absence of an automated method to validate semantic preservation in repairs remains a critical gap. Future research should prioritize developing a validation tool that automatically checks if semantic data—such as classifications and attribute relationships—are consistently maintained and accurately mapped to repaired geometries. This tool could streamline the process by detecting and flagging misalignments, missing semantics, or reassignments introduced during repair. Such a validation approach would standardize semantic integrity, enhancing the reliability of repairs and ensuring models meet the high semantic standards required for advanced applications like simulations and detailed urban analysis.

8.4.6 Intergrating automatic validation and repair for LODs

As explained in Section 6.2.3, integration of the Level of Detail (LoD) classification should be considered for further work. Seeing classification helps with categorizing data; it makes it easier to analyze and interpret 3D city models when the LOD is correct. The repairs which change the data should, therefore, be re-classified. Some use cases would even require validation and, if needed, re-classify all the existing geometries.

8.4.7 Research on keeping and extending textures

For this thesis, the textures of 3D city models were out of the scope. Therefore, the textures of repaired objects are deleted. Developing methods to integrate texture preservation within the existing repair algorithms will ensure that textures remain consistent and accurately aligned

with repaired geometries. Future work would need to focus on the UV positions of the corresponding vertices and how they change depending on the repair process.

8.4.8 3D GIS application for preparing 3D City data

Currently, no single 3D GIS application can handle all aspects of 3D city model management. Programs such as ArcGIS and QGIS are growing in their 3D data options but are still limited. An ideal application would allow users to prepare data visually by collecting it through imports, applications, or web services and then exporting it in multiple file formats. This would make the repair process much more user-friendly.

Extra functions like splitting or combining datasets would make the pre-processing process visually accessible. It could also provide robust repair capabilities—whether repairing entire datasets at once or singling out specific objects for user-guided decisions. Developing such a comprehensive tool would significantly enhance the efficiency and effectiveness of managing and repairing 3D city models.

A Use case Requirements

Table A.1.: Additional requirements for different use-cases

Use-case	Additional requirement(s)	Source
Estimation of the solar irradiation	Building needs to have an identifiable surface(s) with type `RoofSurface`; Surfaces need to be oriented outwards (also when being a `MultiSurface`); *Preferably architecturally detailed models;*	Biljecki et al. (2015) and Biljecki et al. (2016a) and Biljecki (2017) and Ledoux (2017)
Energy demand estimation	Buildings need to be consist of 1 (watertight) `solid` (or `composite solid`); Buildings and/or buildingparts may not intersect; Preferably building needs to have identifiable semantic surface(s);	Coors et al. (2020), Willenborg et al. (2018) and Sindram et al. (2016)
Aiding positioning	*Detailed (≥ LOD3) preferably textured facades for comparisons with images;*	Coors et al. (2000) and Mao (2011)
Determination of the floor space	*Attribute with number of Storeys;*	Biljecki et al. (2015)
Classifying building types	*Attribute with classification;*	
Geo-visualisation and visualisation enhancement	No overlapping `polygons`; correct orrientation of`polygons`; *Preferably architecturally detailed models (≥LOD2.3);*	Ledoux (2017) and Biljecki (2017)
Visibility analysis	Building needs to have an identifiable surface(s) with type `Windows`; *Architecturally detailed models (≥LOD3) prefereably containing interior (LOD4); Attribute with number of Storeys;*	Biljecki et al. (2015) and Biljecki (2017)
Estimation of shadows cast by urban features	*Real world position and true north needs to be correct; Materials for light reflection; Preferably architecturally detailed models (≥LOD2.3);*	Doellner et al. (2005) and Biljecki (2017)

Continued on the next page

Table A.1 – continued from previous page

Use-case	Additional requirement(s)	Source
Estimation of the propagation of noise in an urban environment	Preferably building needs to have identifiable semantic surface(s); *Materials for sound absorption; Preferably architecturally detailed models (≥LOD2.3);*	Biljecki et al. (2015) and Kurakula and Kuffer (2008)
3D cadastre	Buildings need to be consist of 1 (watertight) `solid` (or `composite solid`); Buildings and/or buildingparts may not intersect; *Attributes about the physical counterparts of the legal objects;*	Stoter and van Oosterom (2005) and Biljecki et al. (2015)
Visualisation for navigation	Building needs to have an identifiable surface(s) with type `Windows` and/or `Door`; *Textures for building facades; Materials for building facades; Preferably architecturally detailed models (≥LOD2.3);*	Coors and Zipf
Urban planning	Preferably building needs to have identifiable semantic surface(s); *Materials and textures for visualisation; Preferably architecturally detailed models (≥LOD2.3);*	Köninger and Bartel (1998) and Mao (2011)
Visualisation for communication of urban information to citizenry	Building needs to have an identifiable surface(s) with type `Windows`; *Architecturally detailed models (≥LOD3) prefereably containing interior (LOD4); Attribute with number of Storeys;*	Biljecki et al. (2015) and Biljecki (2017)
Understanding Synthetic Aperture Radar (SAR) images	*none found*	
Facility management	*Architecturally detailed models containing interior (LOD4);*	Bleifuss et al. (2009)
Automatic scaffold assembly	*Preferably architecturally detailed models (≥LOD2.3); Materials for absorption;*	Corre and Lostanlen (2009)
Emergency response	Building needs to have an identifiable surface(s) with type `Door`; *Attributes Information about building entry points Preferably architecturally detailed models containing interior (LOD4);*	Biljecki et al. (2015)
Lighting simulations	*none found*	
Radio-wave propagation	*Preferably architecturally detailed models (≥LOD2.3); Materials and textures for absorption;*	Kolbe and Donaubauer (2021)

Continued on the next page

Table A.1 – continued from previous page

Use-case	Additional requirement(s)	Source
Computational fluid dynamics	No small features, small edges, and small gaps between buildings; Buildings need to be consist of 1 (watertight) `solid` (or `composite solid`) which don't intersect; Polygon triangles in triangulated objects cannot be slivers;	Paden (2021)
Estimating the population in an area	*Attribute with number of residents; Attribute with number of Storeys for per m2 calculations;*	Biljecki et al. (2015) and Alahmadi et al. (2016)
Routing	Building needs to have an identifiable surface(s) with type `Door`; *Attributes Information about building entry points Preferably architecturally detailed models containing interior (LOD4);*	Jebur (2022)
Forecasting seismic damage	Building needs to have an identifiable surface(s) with type `RoofSurface` (and preferably the type of roof);*Metadata containing geographic position, and relationship to their immediate neighborhood; Attribute with number of Storeys; Materials for vulnerability of surface;*	Redweik et al. (2017) and Mao (2011)
Flooding	*City objects of type `land-use` as Digital terrain model; Materials and textures for insights into surfaces;*	Jebur (2022) and Mao (2011)
Change detection	*Preferably Architecturally detailed models (≥LOD2) for most detailed change*	Biljecki et al. (2015)
Volumetric density studies	Buildings need to be consist of 1 (watertight) `solid` (or `composite solid`); Buildings and/or building parts may not intersect; *Preferably Architecturally detailed models (≥LOD2) for roofs*	Biljecki et al. (2015)
Forest management	*City objects of type `Vegetation`*	Biljecki et al. (2015)
Archaeology	*none found*	

B Schema‘s of file types used

B.1 CityJSON

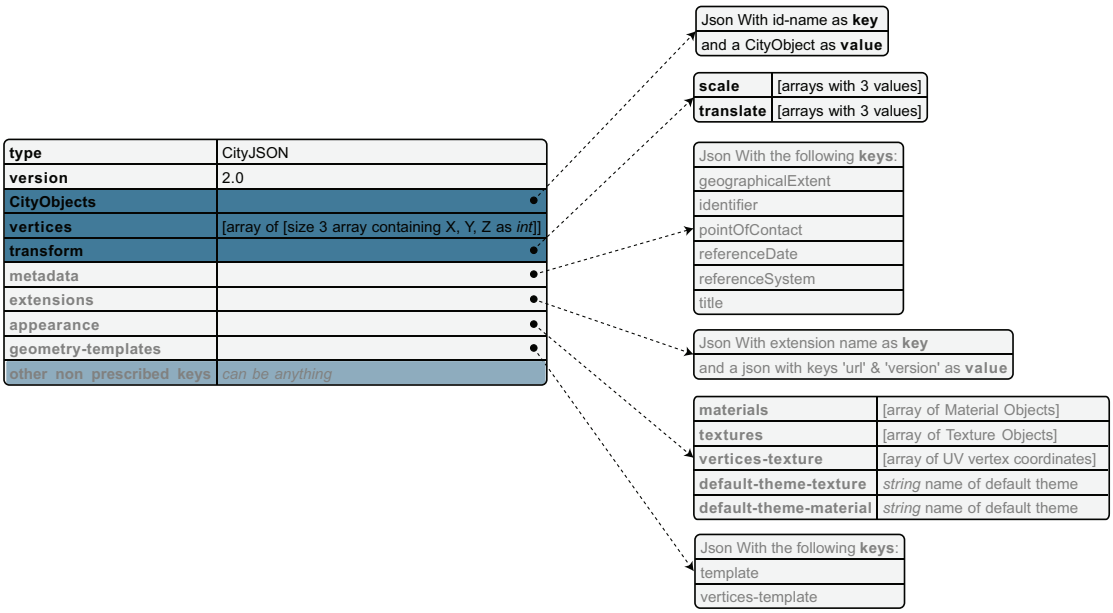


Figure B.1.: CityJSON keys

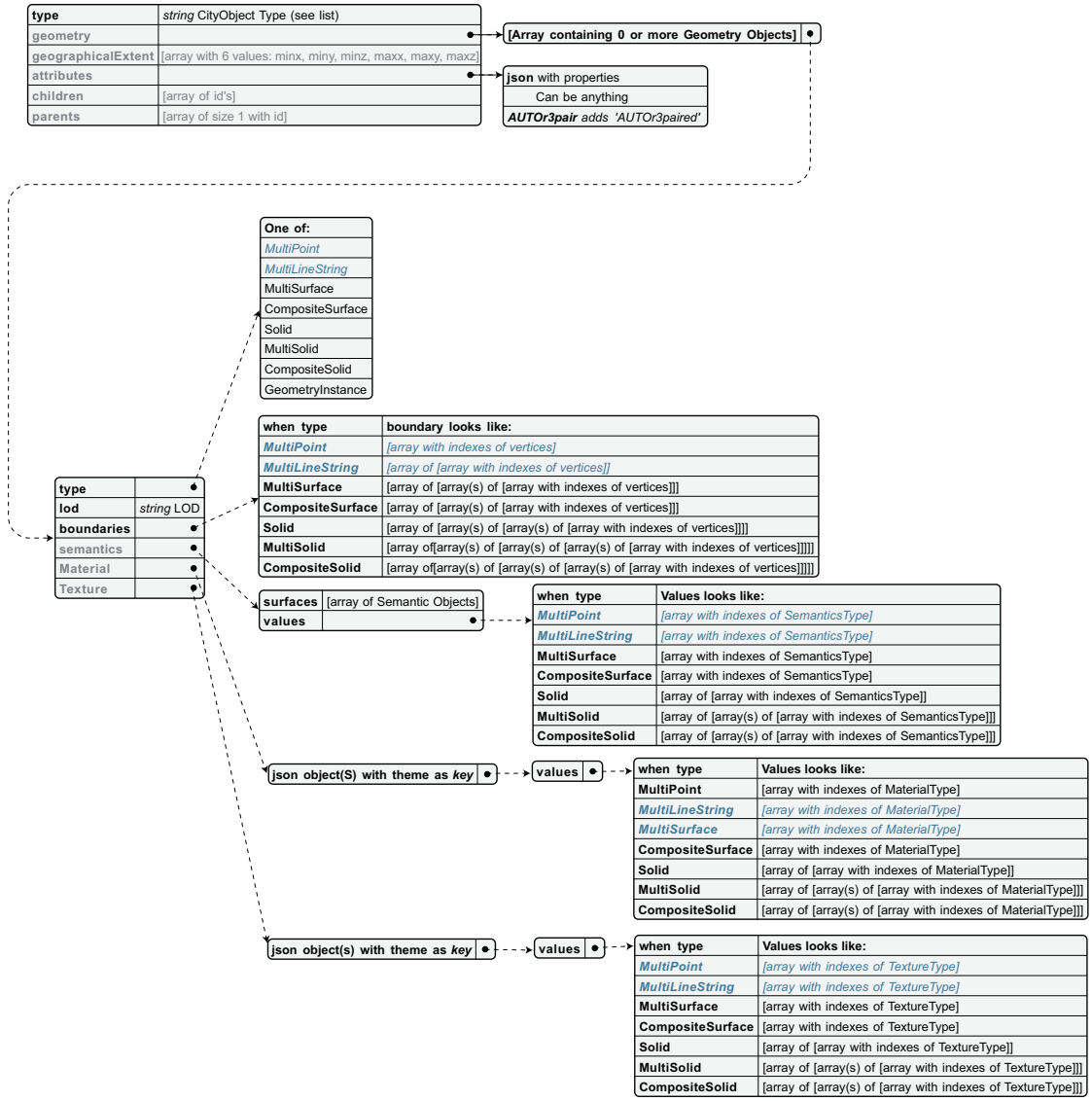


Figure B.2.: CityObject in a CityJSON

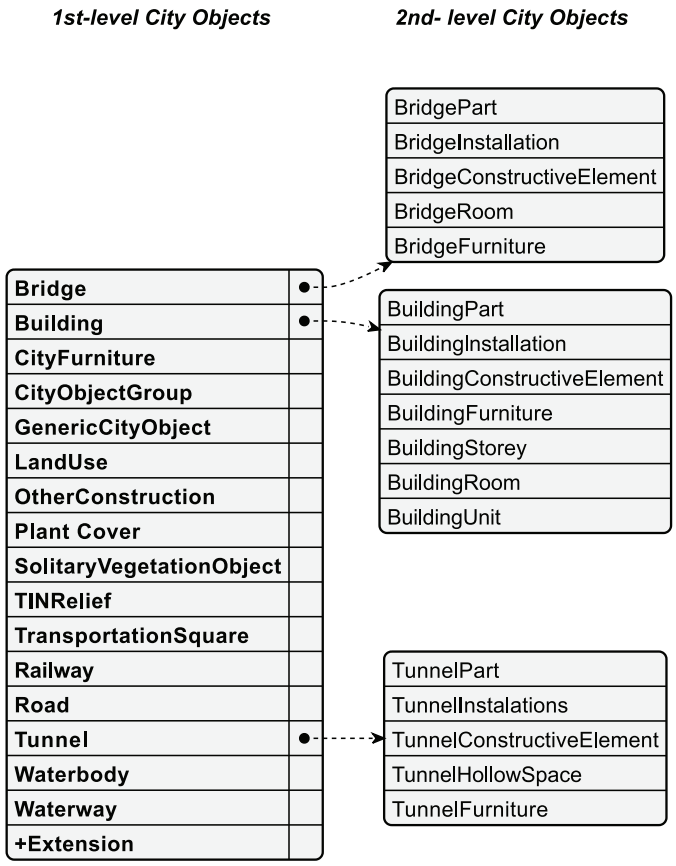


Figure B.3.: CityObject types

B.2 Wavefront OBJ

```
mtllib solids.mtl
o ThreeSolids

g Tetrahedron
usemtl Red
v 0.5773502691896258 0.5773502691896258 0.5773502691896258
v -0.5773502691896258 -0.5773502691896258 0.5773502691896258
v -0.5773502691896258 0.5773502691896258 -0.5773502691896258
v 0.5773502691896258 -0.5773502691896258 -0.5773502691896258
f 1 2 4
f 1 3 2
f 1 4 3
f 2 3 4

g Hexahedron
usemtl Green
v 0.5773502691896258 3.5773502691896257 0.5773502691896258
v 0.5773502691896258 3.5773502691896257 -0.5773502691896258
v 0.5773502691896258 2.4226497308103743 0.5773502691896258
v 0.5773502691896258 2.4226497308103743 -0.5773502691896258
v -0.5773502691896258 3.5773502691896257 0.5773502691896258
v -0.5773502691896258 3.5773502691896257 -0.5773502691896258
v -0.5773502691896258 2.4226497308103743 0.5773502691896258
v -0.5773502691896258 2.4226497308103743 -0.5773502691896258
f 5 6 10 9
f 5 7 8 6
f 5 9 11 7
f 6 8 12 10
f 7 11 12 8
f 9 10 12 11

g Octahedron
usemtl Blue
v 4.0 0.0 0.0
v 2.0 0.0 0.0
v 3.0 1.0 0.0
v 3.0 -1.0 0.0
v 3.0 0.0 1.0
v 3.0 0.0 -1.0
f 13 15 17
f 13 16 18
f 13 17 16
f 13 18 15
f 14 15 18
f 14 16 17
f 14 17 15
f 14 18 16
```

Listing B.1: Example .obj File with 3 groups

B.3 TU3djson

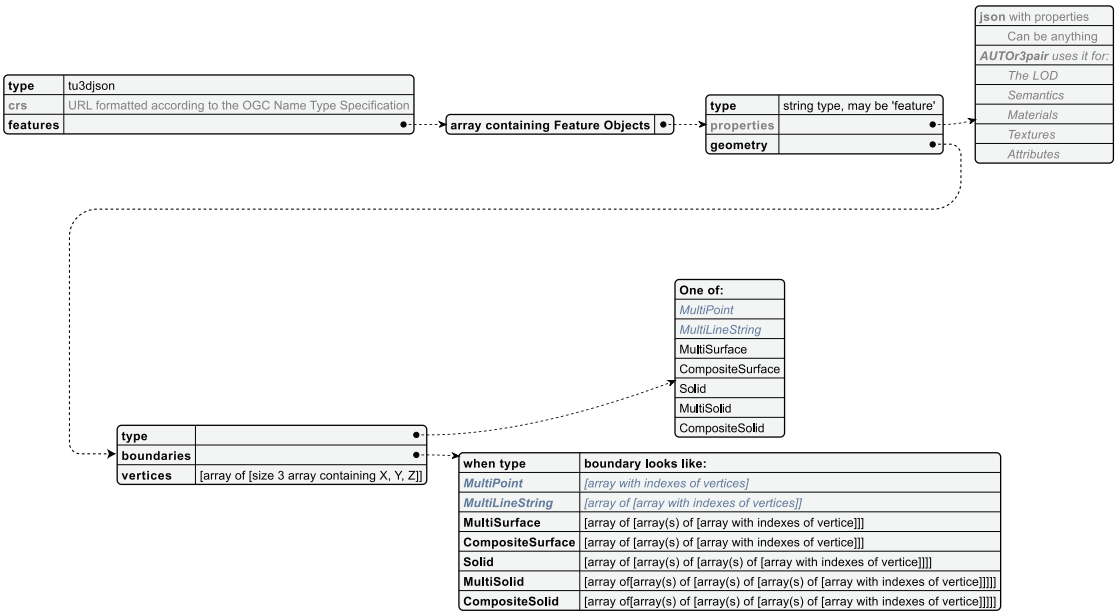


Figure B.4.: TU3DJSON Object

C Algorithm implementation

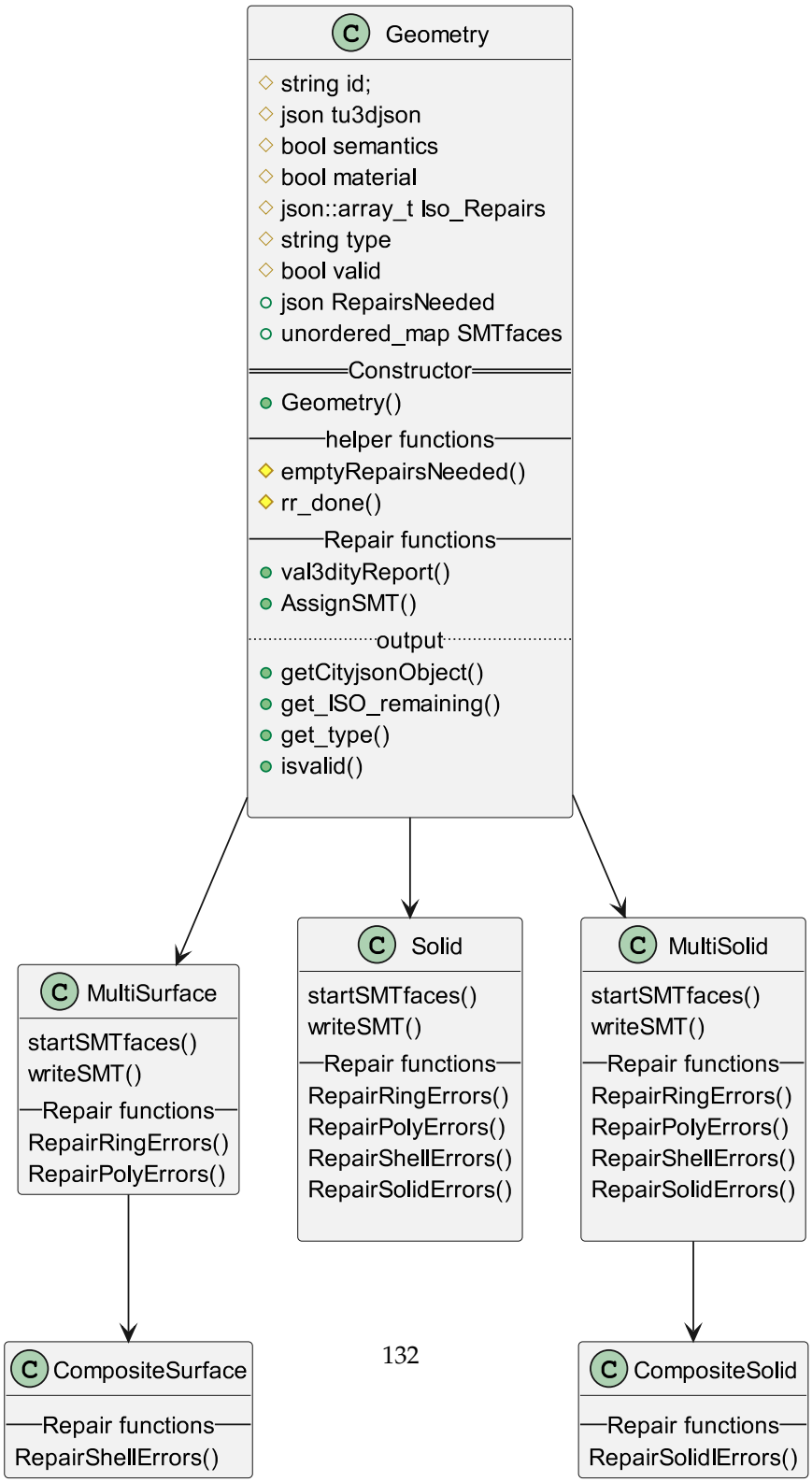


Figure C.1.: inheritance of the geometry class

Algorithm 1 Geometry repair

Input:

- Geometry
- Standards (for val3dityReport and *TotalRepairDepth*, *MaxRepairDepth*)

Output: Repaired geometry/geometries

```
1:  procedure GeometryRepair
2:      RepairsNeeded, Valid ← val3dityReport(Geometry, Standards)
3:      RepairDepth ← 0
4:      ErrorTypes ← {RingErrors, PolyErrors, ShellErrors, SolidErrors, SolidIErrors}
5:      LastErrorCatogorie ← RingErrors
6:      for i ← 1 toTotalRepairDepth do
7:          if RepairDepth < MaxRepairDepth or Valid  then
8:              break
9:          end if
10:         for ErrorType in ErrorTypes do
11:             if RepairsNeeded[ErrorType] is not empty then
12:                 if ErrorType is LastErrorCatogorie then
13:                     RepairDepth + 1
14:                 else
15:                     LastErrorCatogorie ← ErrorType
16:                     RepairDepth ← 0
17:                 end if
18:                 if ErrorType is RingErrors then
19:                     Repair the ring errors
20:                 else if ErrorType is PolyErrors then
21:                     Repair the polygon errors
22:                 else if ErrorType is ShellErrors then
23:                     Repair the shell errors
24:                 else if ErrorType is SolidErrors then
25:                     Repair the solid errors
26:                 else if ErrorType is SolidIErrors then
27:                     Repair the Solid interaction errors
28:                 end if
29:                 break
30:             end if
31:         end for
32:         if Geometry is split into multiple geometries then
33:             return Repaired geometries //GeometryRepair will be reinitialized
34:         end if
35:         if Geometry changed Primitive types then
36:             return Repaired geometry //GeometryRepair will be reinitialized
37:         end if
38:         RepairsNeeded, Valid ← val3dityReport(Geometry, Standards)
39:     end for
40:     return Repaired geometry
41: end procedure
```

Algorithm 2 val3dityReport

Input:

- Geometry
- (standards for val3dity (*snap_tol*, *planarity_d2p_tol*, *planarity_n_tol*, *overlap_tol*)
- Standard with what to solve (*solveAll*, *ErrorsToRepair*)

Output:

- a map with errors sorted Repairsneeded
- Boolean Valid

```
1:  procedure VAL3DITYREPORT
2:      report ← val3dity::validate(tu3djson, snap_tol, planarity_d2p_tol, planarity_n_tol, overlap_tol)
3:      Valid ← false
4:      if report["validity"] = false then
5:          for feature in report["features"] do
6:              for primitive in feature["primitives"] do
7:                  for error in primitive["errors"] do
8:                      errorCode ← error["code"]
9:                      errorInfo ← error["id"]
10:                     if solveAll or ErrorsToRepair contains errorCode then
11:                         if errorCode in RingErrors then
12:                             RepairsNeeded["RingErrors"].push_back({errorCode, errorInfo})
13:                         else if errorCode in PolyErrors then
14:                             RepairsNeeded["PolyErrors"].push_back({errorCode, errorInfo})
15:                         else if errorCode in ShellErrors then
16:                             RepairsNeeded["ShellErrors"].push_back({errorCode, errorInfo})
17:                         else if errorCode in SolidErrors then
18:                             RepairsNeeded["SolidErrors"].push_back({errorCode, errorInfo})
19:                         else if errorCode in SolidIErrors then
20:                             RepairsNeeded["SolidIErrors"].push_back({errorCode, errorInfo})
21:                         end if
22:                     end if
23:                 end for
24:             end for
25:         end for
26:         if RepairsNeeded is empty then
27:             Valid ← true
28:         end if
29:     else
30:         Valid ← true
31:     end if
32: end procedure
```

D Reproducibility and self-assessment

The methodology of this thesis required focusing on software development, resulting in an open-source C++ framework. A significant amount of time was spent figuring out existing repair approaches and C++ libraries like CGAL. To adhere to the FAIR principles (Wilkinson et al., 2016), the data used for the experiments were publicly accessible, and 3D city models, such as those from the CityJSON website and the 3dBAG have formats that were well-documented and licensed (Reproducibilty grade 3/3). Seeing that AUTOr3pair is meant as a "preprocessing tool" for users, data preprocessing was not done, and therefore, no reproducibility is needed. The data used for the pytest is val3dity test data (.poly format) converted to CityJSON and OBJ; all these files and the Python conversion code are findable on Git Hub.

The methodology applied in this thesis was comprehensive and built on existing approaches in the field of 3D city model repair. Val3dity, an open-source geometric validation tool, is used, and validation is built into the framework for semantic and additional use-case requirements. Newly introduced repair approaches were implemented in the framework or done with the help of CGAL, which is also open-source. This thesis's combination of methodology documentation and an open-source framework with open-source libraries makes the methodology easily reproducible (Reproducibilty grade 3/3).

Based on C++ (and Python for the pytest), the computational environment utilized gives the best performance for large data sets. However, while the environment is fully functional, building and running the C++ is no easy task, primarily due to installing prerequisites from CGAL libraries. Therefore, the Computational Environment's reproducibility could benefit from a containerization tool, which streamlines the installation of dependencies, ensures consistent runtime configurations, and simplifies deployment across various platforms (Reproducibilty grade 2/3).

Regarding results, the AUTOr3pair framework addressed the issues of geometric validity for semantic 3D city models with promising results for multiple use cases. However, some limitations, such as the loss of textures and certain "hardcoded" repair decisions, can yield slightly different results per 3D city model. Although there were good results, these minor issues reduced the overall reproducibility and robustness of the framework (Reproducibilty grade 2/3).

This thesis, along with the development of the AUTOr3pair framework, has laid a foundation for reproducibility by using publicly accessible test data, open-source tools, and detailed methodology documentation. The methodology closely follows the principles and techniques of the Master of Geomatics. Throughout this thesis and the implementation of the AUTOr3pair framework, many challenges were encountered, enabling me to gain new knowledge in various topics. Automatic repair from AUTOr3pair extends beyond geomatics-specific use cases by serving related disciplines, such as architecture, the game industry, and other geometry-focused data sciences.

Bibliography

M. Alahmadi, P. M. Atkinson, and D. Martin. A Comparison of Small-Area Population Estimation Techniques Using Built-Area and Height Data, Riyadh, Saudi Arabia. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(5):1959–1969, May 2016. ISSN 2151-1535. doi: 10.1109/JSTARS.2014.2374175. URL https://ieeexplore.ieee.org/abstract/document/6987250. Conference Name: IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.

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, 5.6 edition, 2023a. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgAlphaWrap3.

P. Alliez, C. Jamin, L. Rineau, S. Tayeb, J. Tournois, and M. Yvinec. 3D Mesh Generation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023b. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgMesh3.

P. Alliez, C. Jamin, L. Rineau, S. Tayeb, J. Tournois, and M. Yvinec. 3D Simplicial Mesh Data Structure. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023c. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgSMDS3.

P. Alliez, S. Pion, and A. Gupta. Principal Component Analysis. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023d. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgPrincipalComponentAnalysisD.

P. Alliez, S. Tayeb, and C. Wormser. 3D Fast Intersection and Distance Computation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023e. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgAABBTree.

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.

H. Barki, G. Guennebaud, and S. Foufou. Exact, robust, and efficient regularized Booleans on general 3D meshes. *Computers & Mathematics with Applications*, 70(6):1235–1254, Sept. 2015. ISSN 0898-1221. doi: 10.1016/j.camwa.2015.06.016. URL https://www.sciencedirect.com/science/article/pii/S0898122115003028.

F. Bernardini and C. Bajaj. Sampling and Reconstructing Manifolds Using Alpha-Shapes. *Proc. 9th Canad. Conf. Comput. Geom.*, July 1998.

H. Bieri. Nef Polyhedra: A Brief Introduction. In H. Hagen, G. Farin, and H. Noltemeier, editors, *Geometric Modelling*, Computing Supplement, pages 43–60, Vienna, 1995. Springer. ISBN 978-3-7091-7584-2. doi: 10.1007/978-3-7091-7584-2_3.

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 and K. Arroyo Ohori. Automatic Semantic-preserving Conversion Between OBJ and CityGML. *Eurographics Workshop on Urban Data Modelling and Visualisation*, page 6 pages, 2015. ISSN 2307-8251. doi: 10.2312/UDMV.20151345. URL https://diglib.eg.org/handle/10.2312/udmv20151345. Artwork Size: 6 pages ISBN: 9783905674804 Publisher: The Eurographics Association.

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 01989715. doi: 10.1016/j.compenvurbsys.2016.04.005. URL https://linkinghub.elsevier.com/retrieve/pii/S0198971516300436.

R. Bleifuss, A. Donaubauer, J. Liebscher, and M. Seitle. Entwicklung einer CityGML-Erweiterung für das Facility Management am Beispiel Landeshauptstadt München. In *Angewandte Geoinformatik 2009: Beiträge zum 21. AGIT-Symposium*. Wichmann, 2009.

R. Boeters, K. Arroyo Ohori, F. Biljecki, and S. Zlatanova. Automatically enhancing CityGML LOD2 models with a corresponding indoor geometry. *International Journal of Geographical Information Science*, 29(12):2248–2268, Dec. 2015. ISSN 1365-8816. doi: 10.1080/13658816.2015.1072201. URL https://doi.org/10.1080/13658816.2015.1072201. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/13658816.2015.1072201.

M. Botsch. *Polygon mesh processing*. A K Peters, Natick, Mass, 2010. ISBN 978-1-56881-426-1. OCLC: ocn423214772.

M. Botsch, D. Sieger, P. Moeller, and A. Fabri. Surface Mesh. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgSurfaceMesh.

p. bourke. Object Files (.obj). URL https://paulbourke.net/dataformats/obj/.

J. Brodeur. Geosemantic Interoperability and the Geospatial Semantic Web. In W. Kresse and D. M. Danko, editors, *Springer Handbook of Geographic Information*, Springer Handbooks, pages 291–310. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-540-72680-7. doi: 10.1007/978-3-540-72680-7_15. URL https://doi.org/10.1007/978-3-540-72680-7_15.

H. Buchholz. *Real-time visualization of 3D city models*. PhD thesis, Universität Potsdam, 2006. URL https://publishup.uni-potsdam.de/frontdoor/index/index/docId/1253.

H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and S. Hagen. The GeoJSON Format. Request for Comments RFC 7946, Internet Engineering Task Force, Aug. 2016. URL https://datatracker.ietf.org/doc/rfc7946. Num Pages: 28.

M. Buyukdemircioglu and S. Kocaman. Reconstruction and Efficient Visualization of Heterogeneous 3D City Models. *Remote Sensing*, 12(13):2128, Jan. 2020. ISSN 2072-4292. doi: 10.3390/rs12132128. URL https://www.mdpi.com/2072-4292/12/13/2128. Number: 13 Publisher: Multidisciplinary Digital Publishing Institute.

F. Cacciola, M. Rouxel-Labbé, B. Şenbaşlar, and J. Komaromy. Triangulated Surface Mesh Simplification. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgSurfaceMeshSimplification.

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, T. Huch, and U. Kretschmer. Matching buildings: pose estimation in an urban environment. In *Proceedings IEEE and ACM International Symposium on Augmented Reality (ISAR 2000)*, pages 89–92, Oct. 2000. doi: 10.1109/ISAR.2000.880928. URL https://ieeexplore.ieee.org/abstract/document/880928?casa_token=grlkzDkEARsAAAAA:Nex1iFTgotVRc2KB5uW_lMgpL3iGixN5_qrGisPBZImNEs4ac4DzS_3nTR_D8SomWjIANnBUTA.

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.

Y. Corre and Y. Lostanlen. Three-Dimensional Urban EM Wave Propagation Model for Radio Network Planning and Optimization Over Large Areas. *IEEE Transactions on Vehicular Technology*, 58(7):3112–3123, Sept. 2009. ISSN 1939-9359. doi: 10.1109/TVT.2009.2016973. URL https://ieeexplore.ieee.org/document/4798203. Conference Name: IEEE Transactions on Vehicular Technology.

S. Cox, P. Daisey, R. Lake, C. Portele, and A. Whiteside. *Geography Markup Language (GML) Encoding Specification v3.1.1*. Feb. 2004. doi: 10.13140/2.1.2846.2401.

T. K. F. Da. 2D Alpha Shapes. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgAlphaShapes2.

A. Diakité, G. Damiand, and G. Gesquière. Automatic Semantic Labelling of 3D Buildings Based on Geometric and Topological Information. *9th 3DGeoInfo Conference 2014 - Proceedings*, Nov. 2014.

L. Dietze, U. Nonn, and A. Zipf. Metadata for 3D City Models. 2007.

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.

B. Dukai. *Exploring the automatic Level of Detail inference for the validation of buildings in 3D city models | TU Delft Repository*. PhD thesis, TU delft, Delft, 2018. URL https://repository.tudelft.nl/record/uuid:5e9ed2f0-ec9e-4d9e-9a6b-57488ddd0222.

H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1): 43–72, Jan. 1994. ISSN 0730-0301. doi: 10.1145/174462.156635. URL https://dl.acm.org/doi/10.1145/174462.156635.

C. El Morr, M. Jammal, H. Ali-Hassan, and W. El-Hallak. Data Preprocessing. In C. El Morr, M. Jammal, H. Ali-Hassan, and W. EI-Hallak, editors, *Machine Learning for Practical Decision Making: A Multidisciplinary Perspective with Applications from Healthcare, Engineering and Business Analytics*, International Series in Operations Research & Management Science, pages 117–163. Springer International Publishing, Cham, 2022. ISBN 978-3-031-16990-8. doi: 10.1007/978-3-031-16990-8_4. URL https://doi.org/10.1007/978-3-031-16990-8_4.

A. Fabri, G.-J. Giezeman, and L. Kettner. I/O Streams. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgStreamSupport.

D. Flamanc, G. Maillet, and H. Jibrini. 3D City Models: An Operational Approach Using Aerial Images and Cadastral Maps. 2003. URL https://www.semanticscholar.org/paper/3D-CITY-MODELS%3A-AN-OPERATIONAL-APPROACH-USING-AND-Flamanc-Maillet/29ec85ee8cb7c24f259fc06c139786a617a12978.

J. D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1996. ISBN 978-0-201-84840-3.

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

C. Fruh and A. Zakhor. Constructing 3D city models by merging aerial and ground views. *IEEE Computer Graphics and Applications*, 23(6):52–61, Nov. 2003. ISSN 1558-1756. doi: 10.1109/MCG.2003.1242382. URL https://ieeexplore.ieee.org/abstract/document/1242382. Conference Name: IEEE Computer Graphics and Applications.

T. D. D. geoinformation research group. Tudelft3d/tu3djson: description of the tu3djson standard (TU Delft 3D JSON). URL https://github.com/tudelft3d/tu3djson.

G.-J. Giezeman and W. Wesselink. 2D Polygons. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgPolygon2.

M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, M. Seel, G. Battista, and U. Zwick. Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, and Implementation. Sept. 2003. doi: 10.1007/978-3-540-39658-1_59.

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.

A. Gueziec, G. Taubin, F. Lazarus, and B. Hom. Cutting and stitching: Converting sets of polygons to manifold surfaces. *Visualization and Computer Graphics, IEEE Transactions on*, 7: 136–151, May 2001. doi: 10.1109/2945.928166.

P. Hachenberger and L. Kettner. 3D Boolean Operations on Nef Polyhedra. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgNef3.

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.

P. S. Heckbert and M. Garland. Optimal triangulation and quadric-based surface simplification. *Computational Geometry*, 14(1):49–65, Nov. 1999. ISSN 0925-7721. doi: 10.1016/S0925-7721(99)00030-9. URL https://www.sciencedirect.com/science/article/pii/S0925772199000309.

S. Hert and S. Schirra. 2D Convex Hulls and Extreme Points. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023a. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgConvexHull2.

S. Hert and S. Schirra. 3D Convex Hulls. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023b. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgConvexHull3.

J. Huang, J. Stoter, R. Peters, and L. Nan. City3D: Large-Scale Building Reconstruction from Airborne LiDAR Point Clouds. *Remote Sensing*, 14(9):2254, Jan. 2022. ISSN 2072-4292. doi: 10.3390/rs14092254. URL https://www.mdpi.com/2072-4292/14/9/2254. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

C. Jamin, S. Pion, and M. Teillaud. 3D Triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgTriangulation3.

F. Javadnejad, R. Slocum, D. Gillins, M. Olsen, and C. Parrish. Dense Point Cloud Quality Factor as Proxy for Accuracy Assessment of Image-Based 3D Reconstruction. *Journal of Surveying Engineering*, 147:04020021–1, Feb. 2021. doi: 10.1061/(ASCE)SU.1943-5428.0000333.

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.

Y. Jun. A piecewise hole filling algorithm in reverse engineering. *Computer-Aided Design*, 37 (2):263–270, Feb. 2005. ISSN 0010-4485. doi: 10.1016/j.cad.2004.06.012. URL https://www.sciencedirect.com/science/article/pii/S0010448504001320.

K. Katrioplas and M. Rouxel-Labbé. Optimal Bounding Box. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgOptimalBoundingBox.

L. P. Kobbelt, J. Vorsatz, and U. a. Labsik. A Shrink Wrapping Approach to Remeshing Polygonal Surfaces. *Computer Graphics Forum*, 18(3):119–130, 1999. ISSN 1467-8659. doi: 10.1111/1467-8659.00333. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00333. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.00333.

T. Kolbe, G. Gröger, and L. Plümer. CityGML - Interoperable access to 3D city models. *Geo-information for Disaster Management*, Jan. 2005. ISSN 978-3-540-24988-7. doi: 10.1007/3-540-27468-5_63.

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.

T. H. Kolbe, T. Kutzner, C. S. Smyth, C. Nagel, C. Roensdorf, and C. Heazel. OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard, Sept. 2021. URL http://www.opengis.net/doc/IS/CityGML-1/3.0.

H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin. pytest x.y, 2004. URL https://github.com/pytest-dev/pytest. original-date: 2015-06-15T20:28:27Z.

V. K. Kurakula and M. Kuffer. 3D Noise Modeling for Urban Environmental Planning and Management. 2008. Meeting Name: REAL CORP.

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.

A. Labetski, K. Kumar, H. Ledoux, and J. Stoter. A metadata ADE for CityGML. *Open Geospatial Data, Software and Standards*, 3(1):16, Nov. 2018. ISSN 2363-7501. doi: 10.1186/s40965-018-0057-4. URL https://doi.org/10.1186/s40965-018-0057-4.

R. landsurveyors. Choosing the Right Technology: Lidar or Photogrammetry for Accurate Drone Surveying, Apr. 2023. URL https://www.rvslandsurveyors.com/lidar-or-photogrammetry.

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. How useful are current 3D city models?, Mar. 2017. URL https://speakerdeck.com/hugoledoux/how-useful-are-current-3d-city-models.

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 and D. Balazs. CityJSON Specifications 2.0.0, Sept. 2023. URL https://www.cityjson.org/specs/2.0.0/.

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.

Y. K. Lee, C. K. Lim, H. Ghazialam, H. Vardhan, and E. Eklund. Surface mesh generation for dirty geometries by the Cartesian shrink-wrapping technique. *Engineering with Computers*, 26(4):377–390, Aug. 2010. ISSN 1435-5663. doi: 10.1007/s00366-009-0171-0. URL https://doi.org/10.1007/s00366-009-0171-0.

B. Lei, R. Stouffs, and F. Biljecki. Assessing and benchmarking 3D city models. *International Journal of Geographical Information Science*, 37(4):788–809, Apr. 2023. ISSN 1365-8816. doi: 10.1080/13658816.2022.2140808. URL https://doi.org/10.1080/13658816.2022.2140808. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/13658816.2022.2140808.

P. Liepa. Filling holes in meshes. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03, pages 200–205, Goslar, DEU, June 2003. Eurographics Association. ISBN 978-1-58113-687-6.

P. Lindstrom and G. Turk. Image-driven simplification. *ACM Trans. Graph.*, 19(3):204–241, July 2000. ISSN 0730-0301. doi: 10.1145/353981.353995. URL https://dl.acm.org/doi/10.1145/353981.353995.

S. Loriot, M. Rouxel-Labbé, J. Tournois, and I. O. Yaz. Polygon Mesh Processing. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgPolygonMeshProcessing.

B. Mao. Visualisation and Generalisation of 3D City Models. 2011. URL https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-48174. Publisher: KTH Royal Institute of Technology.

K. McHenry and P. Bajcsy. An Overview of 3D Data Content, File Formats and Viewers. *Technical Report*, 2008.

A. Mikchevitch and J.-P. Pernot. Methodology for automatic recovering of 3D partitions from unstitched faces of non-manifold CAD models. *Engineering with Computers*, 31:73–84, Sept. 2013. doi: 10.1007/s00366-013-0325-y.

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.

R. Nouvel, M. Zirak, V. Coors, and U. Eicker. The influence of data quality on urban heating demand modeling using 3D city models. *Computers, Environment and Urban Systems*, 64: 68–80, July 2017. ISSN 0198-9715. doi: 10.1016/j.compenvurbsys.2016.12.005. URL https://www.sciencedirect.com/science/article/pii/S0198971516304306.

G.-A. Nys, F. Poux, and R. Billen. CityJSON Building Generation from Airborne LiDAR 3D Point Clouds. *ISPRS International Journal of Geo-Information*, 9(9):521, Sept. 2020. ISSN 2220-9964. doi: 10.3390/ijgi9090521. URL https://www.mdpi.com/2220-9964/9/9/521. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

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 in 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.

B. Reitinger, C. Zach, and D. Schmalstieg. Augmented Reality Scouting for Interactive 3D Reconstruction. pages 219–222, Jan. 2007. doi: 10.1109/VR.2007.352485.

B. Saeidian, A. Rajabifard, B. Atazadeh, and M. Kalantari. A semantic 3D city model for underground land administration: Development and implementation of an ADE for CityGML 3.0. *Tunnelling and Underground Space Technology*, 140:105267, Oct. 2023. ISSN 0886-7798. doi: 10.1016/j.tust.2023.105267. URL https://www.sciencedirect.com/science/article/pii/S0886779823002870.

M. Seel. 2D Boolean Operations on Nef Polygons. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. URL https://doc.cgal.org/5.6/Manual/packages.html#PkgNef2.

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.

A. Stadler and T. H. Kolbe. Spatio-semantic coherence in the integration of 3D city models. 2007. URL https://mediatum.ub.tum.de/1145757.

J. E. Stoter and P. J. M. van Oosterom. Technological aspects of a full 3D cadastral registration. *International Journal of Geographical Information Science*, 19(6): 669–696, July 2005. ISSN 1365-8816. doi: 10.1080/13658810500106042. URL https://doi.org/10.1080/13658810500106042. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/13658810500106042.

L. Subramaniam. *PARTITION OF A NON-SIMPLE POLYGON INTO SIMPLE POLYGONS*. PhD thesis, 2003.

K. Takayama, A. Jacobsen, L. Kavan, and O. Sorkine-Hornung. A Simple Method for Correcting Facet Orientations in Polygon Meshes Based on Ray Casting. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):53–63, Dec. 2014. ISSN 2331-7418. URL https://users.cs.utah.edu/~ladislav/takayama14simple/takayama14simple.html.

M. Tang, M. Lee, and Y. J. Kim. Interactive Hausdorff Distance Computation for General Polygonal Models.

L. Tekumalla and E. Cohen. A Hole-Filling Algorithm for Triangular Meshes. Jan. 2004.

the International Organization for Standardization and O. G. Consortium. ISO 19125-1:2004(en), Geographic information — Simple feature access — Part 1: Common architecture, 2004. URL https://www.iso.org/obp/ui/en/#iso:std:iso:19125:-1:ed-1:v2:en.

the International Organization for Standardization and O. G. Consortium. ISO 19115-1:2014, 2014. URL https://www.iso.org/standard/53798.html.

the International Organization for Standardization and O. G. Consortium. ISO 19107:2019(en), Geographic information — Spatial schema, 2019. URL https://www.iso.org/obp/ui/en/#iso:std:iso:19107:ed-2:v1:en.

the International Organization for Standardization and O. G. Consortium. ISO 19115-3:2023, 2023. URL https://www.iso.org/standard/80874.html.

J. N. H. van Liempt. CityJSON: does (file) size matter? 2020. URL https://repository.tudelft.nl/islandora/object/uuid%3A4aad07f4-8f64-46b1-aad3-3d4abe36c5bf.

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.

D. Wagner, N. Alam, M. Wewetzer, M. Pries, and V. Coors. Methods for Geometric Data Validation of 3d City Models. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL15:729–735, Dec. 2015. ISSN 2194-9034 The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. doi: 10.5194/isprsarchives-XL-1-W5-729-2015. URL https://ui.adsabs.harvard.edu/abs/2015ISPArXL15..729W. ADS Bibcode: 2015ISPArXL15..729W.

D. Wagner, H. Ledoux, C. Roensdorf, S. Thum, D. Hintz, F. Biljecki, J. Stoter, E. Casper, B. Joachim, V. Coors, and L. Walstijn. *OGC CityGML Quality Interoperability Experiment*. Aug. 2016.

M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. G. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. C. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, Mar. 2016. ISSN 2052-4463. doi: 10.1038/sdata.2016.18. URL https://www.nature.com/articles/sdata201618. Publisher: Nature Publishing Group.

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

B. Willenborg, M. Sindram, and T. H. Kolbe. Applications of 3D City Models for a Better Understanding of the Built Environment. In M. Behnisch and G. Meinel, editors, *Trends in Spatial Analysis and Modelling: Decision-Support and Planning Strategies*, Geotechnologies and the Environment, pages 167–191. Springer International Publishing, Cham, 2018. ISBN 978-3-319-52522-8. doi: 10.1007/978-3-319-52522-8_9. URL https://doi.org/10.1007/978-3-319-52522-8_9.

H. Zhang and K. E. Hoff. Fast backface culling using normal masks. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, I3D '97, pages 103–ff., New York, NY, USA, Apr. 1997. Association for Computing Machinery. ISBN 978-0-89791-884-8. doi: 10.1145/253284.253314. URL https://doi.org/10.1145/253284.253314.

W. Zhao, S. Gao, and H. Lin. A Robust Hole-Filling Algorithm for Triangular Mesh. volume 23, pages 22–22, Nov. 2007. ISBN 978-1-4244-1579-3. doi: 10.1109/CADCG.2007.4407836.

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.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.