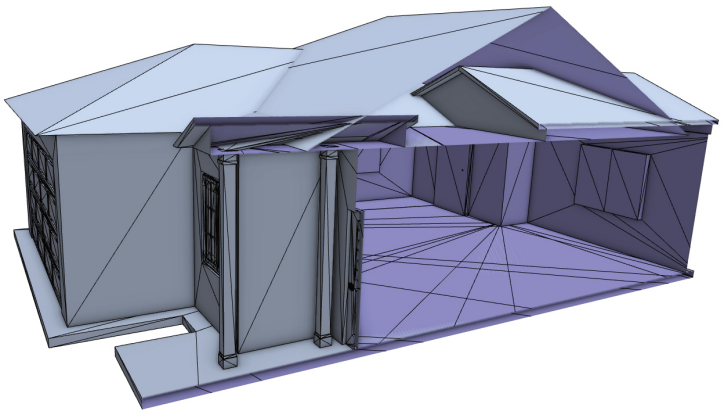
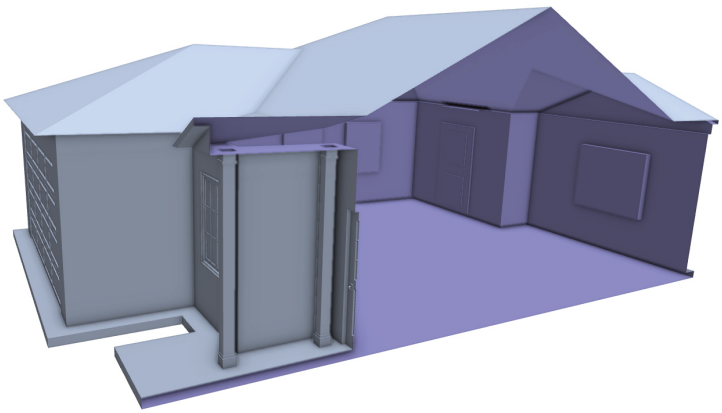
MSc thesis in Geomatics for the Built Environment

# Outer Surface Extraction for Complex 3D Building Models

Yifang Zhao

2020





TU Delft

**MSc thesis in Geomatics**

# Outer surface extraction for complex 3D building models

Yifang Zhao

June 2020

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
Department of Urbanism
Faculty of the Built Environment & Architecture
Delft University of Technology

| | |
|---|---|
| Supervisors: | Dr. Liangliang Nan |
| | Dr. H. Hugo Ledoux |
| Co-reader: | Dr. Ravi Peters |

# Abstract

In recent years, 3D building models have become increasingly widespread and are intensively exploited in fields of computer graphics and geometry processing. One of the common surface representations of 3D building models is polygon mesh, which is both compact and efficient in terms of exchange format and data processing respectively. Downstream applications of polygon meshes can be found in the fields of urban planning, digital mapping, and fluid simulation.

However, the aforementioned applications usually require the input to be watertight and manifold, which is not always fulfilled by existing building models. Moreover, the interior structures of a building model are also considered redundant in certain applications. From such practical demands comes our graduation project, i.e. trying to recover watertight and manifold outer surface from error-ridden 3D building models.

Existing methods with respect to outer surface extraction can be categorized into two types: surface-oriented methods and volumetric methods. The former focuses on one particular type of artifacts and operates directly on the surfaces of the defective model. Since surface-oriented methods mainly introduce local operations where needed, unnecessary changes of the original model can be avoided and the result is of high fidelity. Whilst, the latter generates an intermediate representation of the original model, based on which the outer surface is extracted. The volumetric methods are more heuristic for our project since they are designed especially for multiple artifacts and the results are gauranteed with some desired properties.

In this thesis, we propose a hybrid approach for the extraction of outer surface from error-ridden 3D building models, which aims at recovering a watertight and manifold outer shell of the original model. The advantage of our method is that it is non-parametric, fully automatic, and have no assumptions for the input. Moreover, the small features of original model are kept to the greatest extent after processing. Our method can be divided into four steps: `1)` pre-processing, `2)` constrained tetrahedralization, `3)` classification, and `4)` outer surface extraction. All six types of artifacts listed in this paper are gradually resolved during these steps, resulting in a watertight and manifold representation.

The results from our experiments turn out that our methodology can generate valid results in most cases, while preserving input faces and small features at the same time. Comparing with several state-of-the-art methods, our results still possess superior properties in terms of validity and integrity.

# Acknowledgements

This year is hard for us graduates who have to work from home due to the outbreak of covid-19. Thanks to the help from my supervisors and the company of my friends, otherwise I would not have finished this thesis.

I would like to express my gratitude to the first supervisor, Liangliang Nan. Thanks for sharing with me the theoretical knowledge that supports this thesis and the platform as well as data on which the whole project builds. His previous works in the relevant field are especially heuristic and helpful during the implementation of our methodology. Without his generous help, I would have to start from scratch for basic works. Also, I would like to thank my second supervisor, Hugo Ledoux, for providing comprehensive comments and constructive suggestions at several critical time points. I would also like to give thanks to the co-reader, Ravi Peters, who has made practical improvements on my draft thesis.

In addition, I would like to thank my friend, Jinglan Li, for her thoughtful kindness and encouragement during my studies. In the end, I would like to express my appreciation to my family. Their constant support is what keeps me going.

# Contents


**1 Introduction** **1**
- 1.1 Problem statement . . . 1
  - 1.1.1 Artifacts . . . 2
- 1.2 Research questions . . . 2
- 1.3 Thesis outline . . . 3

**2 Related work** **5**
- 2.1 Surface-oriented methods . . . 5
- 2.2 Volumetric methods . . . 8

**3 Methodology** **11**
- 3.1 Overview . . . 11
- 3.2 Pre-processing . . . 11
  - 3.2.1 Duplication Removal . . . 11
  - 3.2.2 Self-intersection Removal . . . 12
- 3.3 Constrained tetrahedralization . . . 12
  - 3.3.1 Behavior control . . . 14
  - 3.3.2 Properties . . . 14
- 3.4 Interior/exterior classification . . . 15
- 3.5 Outer surface extraction . . . 19
  - 3.5.1 Non-manifoldness removal . . . 19

**4 Implementation** **21**
- 4.1 Programming specifics . . . 21
  - 4.1.1 Exact arithmetic . . . 21
  - 4.1.2 2D CDT . . . 21
  - 4.1.3 AABB tree . . . 22
- 4.2 Practical optimizations . . . 22
  - 4.2.1 Grouping of tetrahedra . . . 22
  - 4.2.2 Reuse of 2D boundaries . . . 23

**5 Results & Evaluation** **25**
- 5.1 Results . . . 25
- 5.2 Evaluation . . . 25
  - 5.2.1 Optimization . . . 25
  - 5.2.2 Validity . . . 28
  - 5.2.3 Comparison . . . 30

**6 Conclusions** **33**
- 6.1 Research overview . . . 33
- 6.2 Discussion . . . 33
- 6.3 Future work . . . 34

# List of Figures


1.1 Artifact chart . . . 3

2.1 Proximity calculation . . . 6
2.2 Hole filling algorithm . . . 6
2.3 Converting a nonmanifold surface to a manifold surface . . . 7
2.4 2D illustration of the repair pipeline . . . 7
2.5 Scan-converting polygonal models with a variety of degeneracies . . . 9
2.6 Illustration of Huang et al.'s method . . . 9
2.7 Illustration of Murali's Method . . . 10
2.8 Solid angle of a triangle with respect to a point. . . . 10

3.1 Illustration of the pipeline in two dimensional (2D) . . . 11
3.2 Exemplary model . . . 12
3.3 Self-intersecting roof surfaces . . . 13
3.4 Remeshing method . . . 13
3.5 Remeshed roof surfaces . . . 13
3.6 Polyhedra which can not be tetrahedralized without Steiner points . . . 14
3.7 Tetrahedralized exemplary model (convex). . . . 15
3.8 Tetrahedra of irregular shapes . . . 15
3.9 Ray casting classification in 2D . . . 16
3.10 Atypical gap when triangulation fails and our improved method by computing the closest point. . . . 16
3.11 Result comparison between voting criteria of Tzounakos's and ours . . . 17
3.12 Result comparision between classification method of Tzounakos's and ours . . . 18
3.13 Tetrahedralized exemplary model after classification. . . . 18
3.14 Non-manifoldness in extracted outer surface . . . 19
3.15 Edge duplication introduces boundary edges . . . 20
3.16 Outer surface of the exemplary model . . . 20

4.1 Exterior region of the exemplary model . . . 23
4.2 Two kinds of representative points . . . 23
4.3 All constructed 2D boundaries for classification . . . 24

5.1 Stepwise results of our program . . . 26
5.1 Stepwise results of our program (cont.) . . . 27
5.2 Estimated execution time w.r.t. the number of input faces. . . . 29
5.3 Close-ups of mesh details . . . 31

6.1 An exemplary non-manifold edge for which "pinching" fails . . . 34

# List of Tables

5.1 Statistics of input errors . . . . . 27
5.2 Statistics concerning optimization operations . . . . . 28
5.3 Execution times . . . . . 29
5.4 Validity check . . . . . 30
5.5 Comparison with state-of-the-art methods . . . . . 31

# Acronyms

**3D** three dimensional . . . . . 1
**2D** two dimensional . . . . . xi
**DT** Delaunay triangulation . . . . . 12
**CFD** computational fluid dynamics . . . . . 1
**CityGML** City Geography Markup Language . . . . . 2
**LODs** Level of Details . . . . . 2
**CT** constrained tetrahedralization . . . . . 12
**CDT** constrained Delaunay triangulation . . . . . 21
**CGAL** Computational Geometry Algorithms Library . . . . . 21
**EPEC** exact_predicates_exact_constructions . . . . . 21
**AABB** axis-aligned bounding box . . . . . 22
**API** application program interface . . . . . 22

# 1 Introduction

In recent years, three dimensional (3D) building models have become increasingly widespread and are intensively exploited in fields of computer graphics and geometry processing [Botsch et al., 2007]. One of the common surface representations of 3D building models is polygon mesh, which is a piecewise planar surface composed of a collection of spatially connected polygons. Due to the compactness and the efficiency both for rendering and querying, polygon meshes are becoming a de-facto standard in many application contexts and dominating popular 3D model repositories (e.g. ModelNet and ShapeNet).

One application of polygon meshes is that of urban modeling elaborated in [Musialski et al., 2013], which mainly consists of the creation of 3D geometric models of urban areas and individual buildings. Many other downstream applications also benefit from the reconstructed 3D urban models, such as urban planning, digital mapping, and entertainment industry [Biljecki et al., 2016]. Therefore, a significant deal of effort has been made in this field due to its place in both academic and commercial environment. In this context, many algorithms that are automatic and generative have been developed, e.g. image-based modeling ([Werner and Zisserman, 2002], [Dick et al., 2004], [Xiao et al., 2009]), LiDAR-based modeling ([Nan et al., 2010], [Zhou and Neumann, 2008], [Vanegas et al., 2012]), and inverse procedural modeling ([Vanegas et al., 2010], [Ripperda and Brenner, 2009]), which makes urban modeling more efficient, reliable, and cost-effective. To some extent, polygon meshes and urban related applications both benefit from mutual promotion.

Another usage of polygon meshes is as the input geometries of computational fluid dynamics (CFD), e.g. simulations of pedestrain wind, pollution dispersion, and heat transfer in urban areas. In this context, polygon meshes are often restricted by some validity constraints. For instance, the input geometric models must be closed, i.e. the sum of area vectors of all faces equals to zero, and free of self-intersections, in order to define correct boundary conditions. Thus, usually the most crucial part of a CFD simulation cycle , i.e. from geometry preparation to result analysis, is not solving partial differential equations, but the capabilities of generating valid input models.

## 1.1 Problem statement

Many softwares or libraries provide state-of-the-art tools or techniques for constructing polygon meshes either manully or automatically. Unfortunately, most existing polygon meshes fail to meet the requirements of the aforementioned downstream applications which can only accept a clean surface model as input. By describing a model as *clean*, we mean that it is closed and manifold. A polygon mesh which is not clean (e.g. non-closed and non-manifold) often contains artifacts like duplications, holes, non-manifoldness, self-intersections, etc (see Section 1.1.1). Such defects prevent polygonal meshes from further processing and analysing, restricting them to visulization purpose only. Usually more efforts have to be taken before the actual work begins in order to fix these geometric or topological errors.

Therefore, there is a necessity to make problematic models clean, i.e. to remove the aforesaid artifacts from a geometric model to produce a suitable model that can be further processed by downstream applications. Such a process is called model repair. Most model repair algorithms can be classified as being either *surface oriented* or *volumetric* [Botsch et al., 2007]. The former performs modifications locally, hence most features are well preserved. The inconvenience is that such algorithms are not automatic and need further interactions from the user. The latter generates an intermediate volumetric model from the input, based on which the outer surface is then extracted. As a result of the conversion, small structures

or features might be changed or even lost. Both *surface oriented* and *volumetric* methods have their own strengths and weaknesses and they are generally complementary in terms of automaticity and accuracy. Some representations of them are elaborated in Chapter 2.

As far as the author knows, there is no method that can both automatically and accurately restore a clean outer surface from an error-ridden model. Thus, there is still space in the field of model repair to be further explored towards a "one-click" solution.

### 1.1.1 Artifacts

Artifacts in 3D models are quite common to observe and can be easily introduced by modelers unintentionally. Some of them are trivial to handle with, while some are not, especially when there is a combination of several kinds of them. Common types of artifacts are listed below which is not guaranteed to be complete, but covers most defects encountered in real life.

- **Duplication.** Two or more geometries are geometrically or topologically identical. Duplicated vertices and edges that are created to avoid non-manifoldness are not included.
- **Non-manifoldness.** Both vertices and edges can be non-manifold. According to Zilske et al., a vertex is called non-manifold if the star[1] of it is not homeomorphic to a disk. An edge is non-manifold if it is connected by more than two faces. Such vertices and edges are often desribed as singular and complex respectively (see Figure 1.1a and Figure 1.1b).
- **Self-intersection.** Two or more faces intersect with each other (see Figure 1.1c).
- **Hole & gap.** Both of them are caused due to missing faces and, in most cases, can be identified by finding a closed circle of boundary edges (see Figure 1.1d).
- **Misorientation.** The neighboring faces have the opposite orientations. To some extent, misorientation can also be regarded as non-manifoldness when it is represented by halfedge data structures (see Figure 1.1e).
- **Interior geometry.** Geometries that are inside the model domain (see Figure 1.1f). Although most existing softwares accept models with interior geometries, there are lots of applications where they are considered as a redundancy, e.g. building model with Level of Details (LODs) in City Geography Markup Language (CityGML) ([Biljecki et al., 2016]). Thus, they are also treated as a defect.

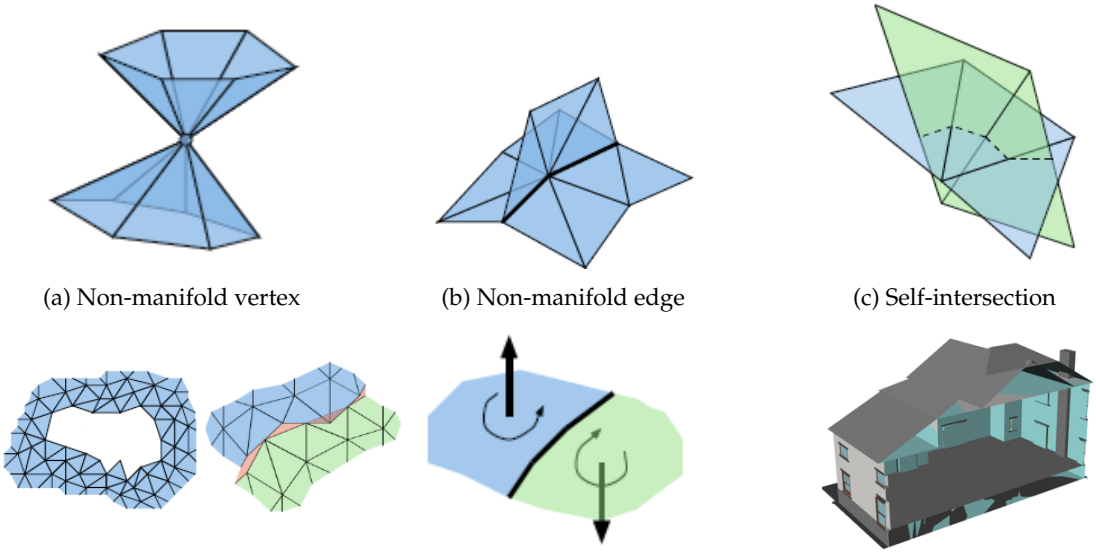
## 1.2 Research questions

Despite the widespread use of polygon meshes, there is still bottlenecks preventing flawed 3D models from being successfully utilized by downstream applications. Hence, the goal of this project is to come up with a method that can automatically and robustly repair a flawed 3D model so that it is suitble for further processing and analysis. As is stated in Section 1.1.1, besides traditional artifacts like duplications and self-intersections, interior geometries are also considered as a defect, hence need to be removed. Based on the statements above, the research question of this project can be defined as follows:

***How can we accurately extract a watertight and manifold outer surface from an error-ridden 3D building model?***

Due to the very nature of this project, i.e. aiming at resolving six types of artifacts mentioned in Section 1.1.1, traditional surface-oriented method is not preferable for its single functionality. Thus, our project will be carried out using a volumetric method, which guagrantees watertightness of the output.

[1] The set of triangles inident to a vertex.



(a) Non-manifold vertex (b) Non-manifold edge (c) Self-intersection

(d) Hole & gap (e) Misorientation (f) Interior geometry

Figure 1.1: (a)-(e): Artifact chart. Adapted from [Botsch et al., 2007]. (f): Building model from [Fan and Wonka, 2016]. Rendered in Mapple.

As is mentioned in Section 4.1, for volumetric methods, there are three common steps: `a)` space partition (i.e. convert the original model into an intermediate volumetric representation), `b)` classification (i.e. classify each volume as either being inside or outside of the input model), and `c)` outer surface extraction (i.e. output faces that lie between interior and exterior volumes). In this context, two sub-questions are inevitable:

- How can the model domain be spatially partitioned?
- How to robustly determine a volume (or a point) as either being inside or outside of the model?

Herein, *model* refers to polygon mesh, i.e. the face of it can be a triangle, a quadrilateral, etc, as long as it is a polygon. Meshes containing curved surfaces are, instead, not within the scope of this project. Also, only the geometrical and topological information is used in our methodology. Others, such as colors, textures, and semantics, are also excluded from disussion.

## 1.3 Thesis outline

The following content of this thesis are composed of five chapters. Chapter 2 provide an overview of representative methods that have done by predecessors with respect to outer surface extraction. All the methods mentioned are divided into two categories: `a)` surface-oriented methods and `b)` volumetric methods.

In Chapter 3, the methodology supporting this project is detailed stepwise. An exemplary model is used during the whole process to intuitively present the intermediate result of each step. Chapter 4 explains how the methodology is implemented specifically in terms of, e.g. underlying data structures and optimization operations.

In Chapter 5, several extracted outer surfaces are presented with optimization effect analysed. Then, a comparison between our results and that produced by state-of-art open softwares are provided, stating

the advantages of our methodogy.

The whole thesis comes to an end in Chapter 6, where the research question is answered, the pros and cons of our methodology, are addressed and the possible solutions for further improvement are discussed.

# 2 Related work

In this chapter, various methods with respect to outer surface extraction are presented and discussed. These methods can be generally categorized as being either surface-oriented or volumetric. The former focuses on one particular type of artifacts and operates directly on the surfaces of the defective model. Since surface-oriented methods mainly introduce local operations where needed, unnecessary changes of the original model can be avoided and the result is of high fidelity. Whilst, the latter generates an intermediate representation of the original model, based on which the outer surface is extracted. The volumetric methods are more heuristic for our project since they are designed especially for multiple artifacts and the results are gauranteed with some desired properties (e.g. watertightness, orientation consistency, and absence of interior structures).

## 2.1 Surface-oriented methods

Surface oriented methods operate directly on the input model by finding and repairing the artifacts locally. For example, holes can be found by identifying the edges of which the number of adjacent polygons is one and can be filled by a fair triangulation that minimizes some weight function. Since such algorithms are performed only on the defective area, structures in the normal area are not affected and the perturbation to the input model is minimized. On the downside, surface oriented methods are usually unifunctional, i.e. tailored for one single type of artifacts, such as normal orientation, hole filling, gap closing, and duplicate removal. Thus one has to apply multiple operations when a combination of the artifacts is present. Moreover, these methods usually assume that the input model satisfies certain quality requirements which is not the case for most existing models and the output model cannot be guaranteed to be closed and manifold since errors other than the target one might exist after processing. Hence additional manual post-processing might be needed in order to get a closed manifold. Below we introduce several typical surface-oriented methods.

`Consistent normal orientation` is essential to some volumetric algorithms (e.g. generalized winding number) and also improves visual effect when textures are attached to the model. Borodin et al. [2004] propose a combinatorial method based on both proximity and visibility to consistently orient all normals of any mesh so that the front-faces of polygons are seen from the outside. It first divides the input model into several patches within which the polygons are oriented consistently. Then proximities between neighboring patches are computed which indicates the possibilities that two patches are adjacent (see Figure 2.1), and visibilities for both sides of a patch showing how much it can be seen from the corresponding half-space. The theoretical foundation is that, for a consistent surface mesh, these two values should be maximized comparing to that of which patches are flipped. Thus, the devided patches are continuously merged and flipped until both proximities and visibilities get their maximums. The result is desirable in almost all practical cases and passable only when there are lots of coplanar polygons.

`Hole filling` is a fundamental routine in the field of model repair and also a building block of many other repair algorithms. Liepa [2003] describes a method to fill holes with patches that are minimally distinguishable from the neighboring meshes, i.e. the vertex density and the geometric structure are similar to those of the surrounding area. It mainly consists of steps including hole identification, hole triangulation, mesh refinement, and fairing (see Figure 2.2). The point density and smoothness of the patching mesh are adjusted in accordance with the surrounding mesh within the last two steps. This method works reliably for most holes in practice. However, the requirement for the input model is

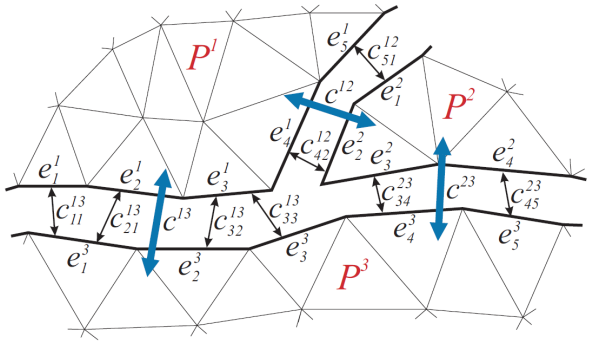



Figure 2.1: Proximity calculation. Retrieved from [Borodin et al., 2004].

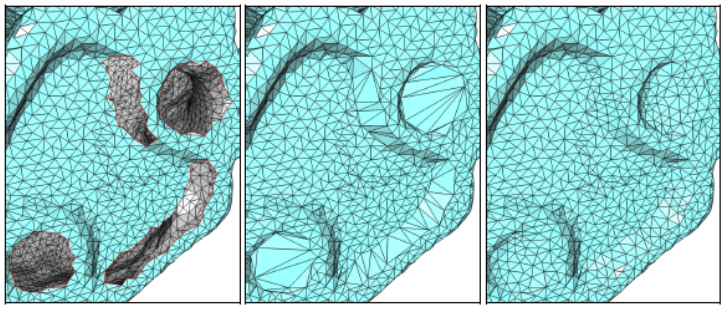


Figure 2.2: Hole filling algorithm. From left to right: holes in the Stanford bunny, triangulated, meshed and faired. Adapted from [Liepa, 2003].

highly demanding that it should be consistently oriented, connected (i.e. no separate patches), and manifold, which restricts its application to a large extend.

`Manifoldness` is a prerequisite for many downstream applications e.g. 3D printing. Common issues with respect to manifoldness are complex edges (also called singular edges) and singular vertices. Complex edges are edges with more than two adjacent faces and singular vertices refer to those connected by edges forming two or more polygonal fans [1]. Guéziec et al. [2001] propose a method based on cutting and stitching to remove complex edges and singular vertices from non-manifold sets of polygons. Cutting simply means duplicating singular vertices for its incident connected components (a connected component of a singular vertex is a sets of connected incident faces which shares no complex edges within them), and stitching means identifying and merging pairs of geometrically coincident boundary edges (including "pinching" and "snapping", see Figure 2.3). The algorithm only modifies topological structures of the input model but ignores its physical coordinates, i.e. no floating point operations. This method can handle any unintended topological singularities automatically but does not address other geometric errors like self-intersecting faces.

`Hybrid methods` also exist that try to resolve a range of problems in one framework. For example, in [Chu et al., 2019], input meshes are processed for finding elementary manifold patches which are then connected into large coherent manifold surfaces using global optimization (see Figure 2.4). The algorithm can handle duplications, mis-orientations and self-intersections but gaps, since it is designed mainly for open meshes and is only guaranteed for manifoldness and consistent orientation.

[1] A polygonal fan is a bunch of connected polygons that share one same vertex and form a 2-manifold.

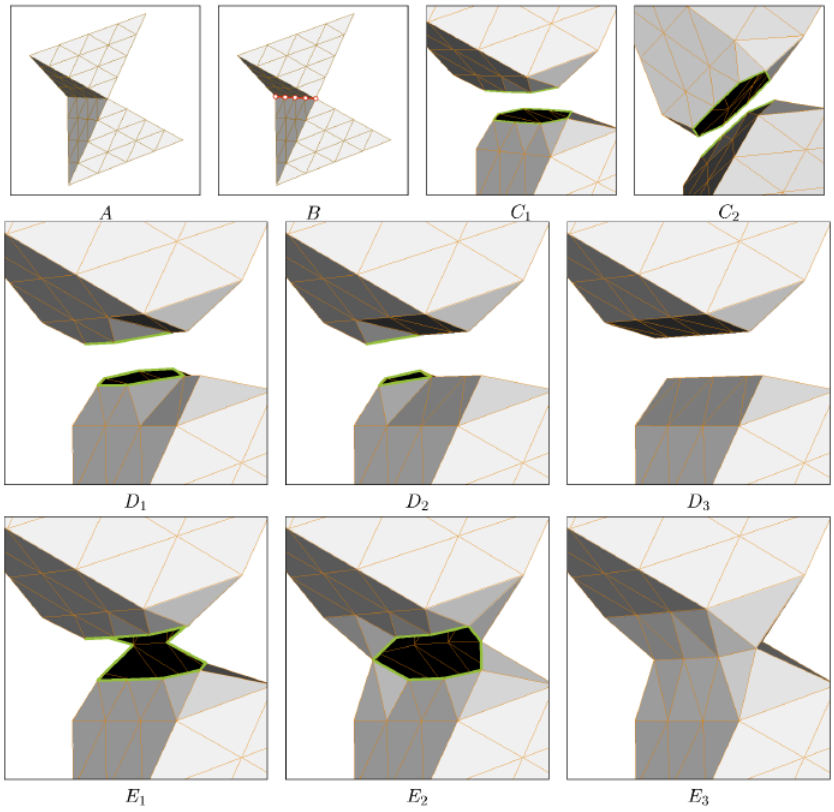



Figure 2.3: Converting a nonmanifold surface to a manifold surface. A, B, C: Cutting through singular edges: For illustrative purposes, topologically disconnected vertices are shown physically apart. We implement two stitching strategies: "pinching" edges along the same boundary (D) or "snapping" together edges belonging to different boundaries (E). Retrieved from [Guéziec et al., 2001].

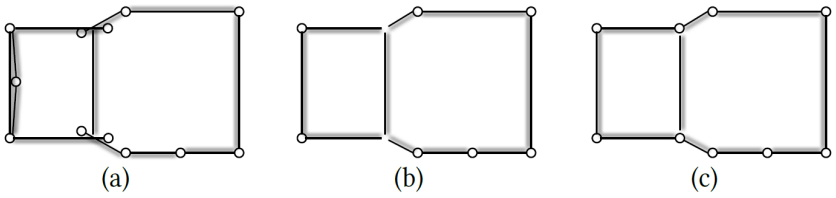



Figure 2.4: 2D illustration of the repair pipeline. (a) depicts the input mesh whose faces are shown as line segments, with shadow showing backside of a face. The input mesh has flipped faces, redundant faces and self intersections. (b) shows the cleaned and reoriented face patches after the visual processing step. (c) shows how the patches are further connected into large and coherent pieces by the manifold mesh reconstruction step. Retrieved from [Chu et al., 2019].

## 2.2 Volumetric methods

Volumetric methods extract the outer surface by applying classification on an intermediate volumetric representation generated from the input model. The representation are cells partitioned by various approaches such as regular grid, BSP-tree and tetrahedralization. Each subdivided cell can be classified as being inside or outside of the input model. Then all the polygons incident to the interior and exterior cells intuitively constitute the final outer surface. Volumetric methods are usually fully automatic and the extracted outer surfaces are gauranteed to be watertight due to the very nature of the space partitioning. However, they may introduce loss of small features when the original surfaces are not respected during space partitioning. Moreover, the extracted outer surface might still be non-manifold in extreme cases where edges and vertices are shared by multiple interior cells. Below we introduce several volumetric methods according to the way they partition the 3D space.

`Regular grid` is first used by Nooruddin and Turk [2003] to repair models with holes, double walls, and intersecting parts. They first convert the input model into voxels, and then classify each voxel as being inside or outside by intersecting a ray with the model. They have proposed two different interpretations for classification with respect to the intersecting points: 1) Parity count. Shooting a ray from a voxel, if the number of intersecting points is odd, this voxel is classified as being interior, otherwise not. 2) Ray stabbing. Shooting a ray from the outside, if a voxel lies between the first and the last intersecting points, it is considered inside, otherwise not. Figure 2.5 roughly illustrates the principle in 2D. Finally, a Marching Cubes algorithm is used to extract the isosurfaces between the interior and the exterior. This method guarantees a manifold output but the classification is rather heuristic and often not reliable, especially in cases where a combination of artifacts has to be dealt with. Recently, this space partition approach is adopted again in [Huang et al., 2020], but they use a different method for classification. They mark voxels as either being occupied (i.e. intersecting with input faces) or exterior (see Figure 2.6) and ignore voxels that are completely inside. The outer surface is represented by faces between occupied voxels and empty voxels.

`Adaptive grid` is used by Bischoff et al. [2005] to repair arbitary triangle soups. The method only requires two parameters as input, an error tolerance $\epsilon$ and a maximum diameter $\rho$ up to which the holes are filled. A difference of adaptive grid from regular grid is that it saves space overhead in homogeneous area and allows relatively more detailed features to be preserved. However, the resolution of the voxelization is still limited.

`BSP tree` is a way for space partitioning exploited by Murali and Funkhouser [1997] to build volumetric representations from triangle soups. The leaf nodes of the BSP tree are closed convex spatial regions and other nodes represent all input polygons functioning as splitting planes. Afterwards each region is classified as either being solid or empty by computing a solidity coefficient ranging from -1 to 1 (negative for empty whilst positive for solid). The solidity coefficient of a region is associated with the connectivities to its neighboring regions. If two adjacent regions share no polygon in between, their solidity coefficients should be consistent, i.e. both positive or both negative. Then the surface can be extracted as the boundary polygons between solid and empty regions. An intuitive pipeline in 2D is given in Figure 2.7. One of the drawbacks of this method is that it can only handle models representing one solid object, not objects with self-intersections. Moreover, the computation of the BSP is non-robust and inefficient.

`Tetrahedralization` is nowadays a main trend to tessellate the interior of a given model. Many softwares have been developed such as CGAL, TetGen, Quartet, and Fade3D. Recently, Hu et al. [2018] propose a tetrahedral meshing technique that automatically converts a triangle soup into an approximately constrained tetrahedralization without tuning parameters or manual intervention. Each tetrahedra is then classified as being inside or outside of the model by employing the generalized winding number algorithm (see Figure 2.8). Finally, the outer surface is extracted as the boundary representation of the interior tetrahedra. This method, however, heavily relies on consistent normal orientation of the input

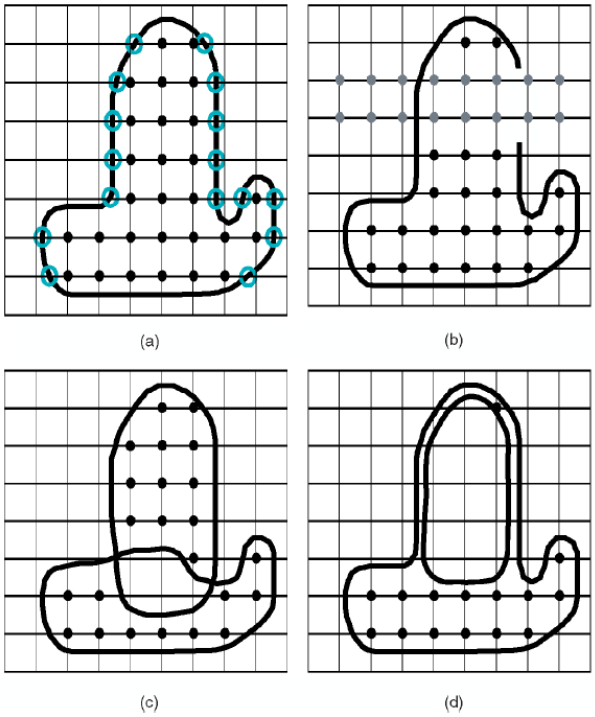



Figure 2.5: Scan-converting polygonal models with a variety of degeneracies. (a) Scan-converting a closed model using the parity count method. The black dots represent voxels that are inside the model. the blue circles show where the scanlines intersect the model. (b) Scan-converting a model with ahole. The gray dots represent voxels for which we do not know whether they are inside or outside the model. Scan converting from multiple directions solves this problem. (c) Scan-converting a model with intersecting parts using parity count. This is another instance where reay stabbing yields better results. (d) Scan-converting a double-walled model using parity count. This shows why some models require the ray stabbing method. Retrieved from [Nooruddin and Turk, 2003].

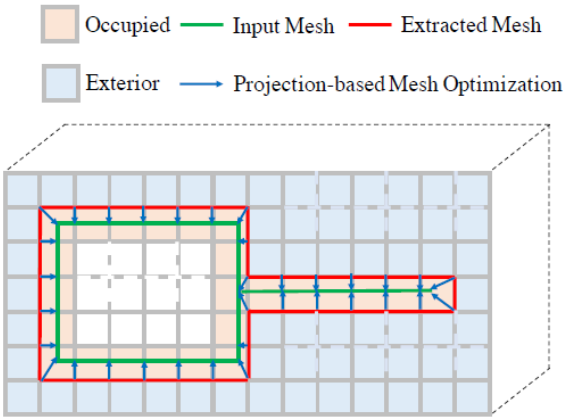



Figure 2.6: Illustration of Huang et al.'s method. Retrieved from [Huang et al., 2020].

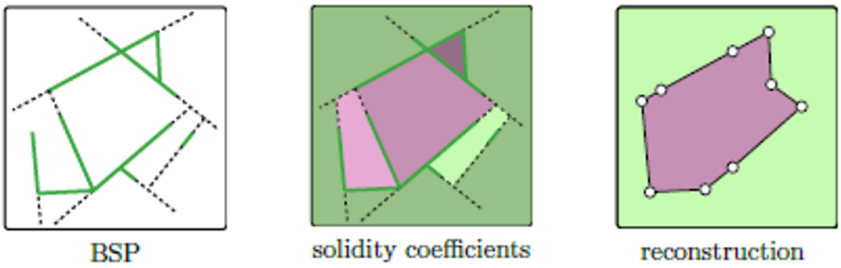



Figure 2.7: Illustration of Murali's Method. Adapted from [Botsch et al., 2007].

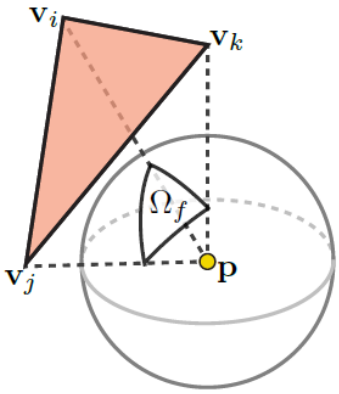



Figure 2.8: Solid angle of a triangle with respect to a point. Generalized winding number is computed as dividing solid angle by $4\pi$. Retrieved from [Jacobson et al., 2013].

model due to the usage of generalized winding number. Tetrahedra near the wrongly oriented surfaces are likely to be wrongly classified, leading to an incorrect meshing result. In addition, the vertices of sharp features could be displaced after processing.

Besides, Ju [2004] proposes an method to repair arbitrary triangle soups by using an octree grid as the underlying volumetric structure. The algorithm is memory-less and can be used to process models of huge size. But it fails to handle sharp features, i.e.if the voxel overlaps with the input geometry, the corresponding mesh may be eliminated or broken into pieces. Shen et al. [2005] proposes a volumetric repair method by generalizing the moving least-squares approach. This method outputs watertight models and gaps can also be bridged in an intuitive way automatically. Unfortunately, it cannot handle models containing interior geometries.

# 3 Methodology

## 3.1 Overview

In this chapter, our method will be illustrated stepwise with an error-ridden 3D building model (see Figure 3.2) which contains duplications, self-intersections, mis-orientations, holes, non-manifold vertices and edges, and interior structures. At first, the model is pre-processed in order to remove duplications and self-intersections. Then, the pre-processed model is tetrahedralized with its original faces preserved and each tetrahedron is classified as either being inside or outside of the pre-processed model. After that, a closed shell of the original model is extracted by trivially selecting all triangles incident to the inside and the outside tetrahedra. In the end, a post-processing step is performed in order to resolve non-manifoldness of the closed shell. A 2D pipeline is given for better understanding (see Figure 3.1).

## 3.2 Pre-processing

In the first step of our methodology, two sub-steps are performed in sequence: `a)` remove duplications and `b)` resolving self-intersections. Substep `a)` is performed again after substep `b)` since there might be new duplications introduced after self-intersections are resolved. These operations are necessary in order to fulfill the requirements of the used mesh data structure and the following steps in Section 3.3.

### 3.2.1 Duplication Removal

Duplicated faces are redundancies which often lead to incorrect result in geometric algorithms like ray-casting classification and generalized wind number ([Jacobson et al., 2013]). It is worth noting that a face can be duplicate both geometrically and topologically. Thus, when detecting deplicated faces, both vertex labels and positions should be checked. After all duplications are detected, we can randomly keep one of them and discard the others.

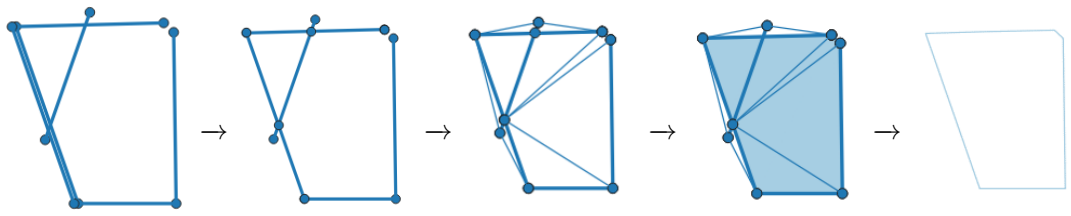



Figure 3.1: Illustration of the pipeline in 2D. The input model containing various defects (a). Pre-processed model with duplications and self-intersections removed (b). Constrained tetrahedralization on pre-processed model (c). Each tetrahedron is classified as either being interior or exterior (d). Extracting triangles between interior and exterior tetrahedra as outer surface (e).

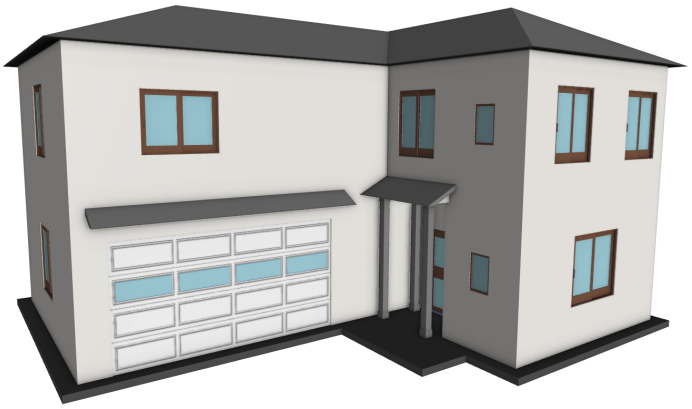


Figure 3.2: Exemplary model

### 3.2.2 Self-intersection Removal

Self-intersecting faces are easier to observe when the target model is rendered with wireframes, e.g. ridgelines penetrates roof surfaces in Figure 3.3. To resolve such errors, we need to remesh the original face. For a specific face, we first find all its intersecting faces, then construct all the intersecting geometries, and in the end, remesh (e.g. using Delaunay triangulation (DT)) the original face with the constructed geometries (Figure 3.4). After remeshing, the original model are replaced by a new one with, as expected, more faces and vertices since addtional intersecting geometries are constructed and added to the model (see Figure 3.5).

## 3.3 Constrained tetrahedralization

After the flawed model is preprocessed, two types of artifacts listed in Section 1.1.1 (i.e. duplication and self-intersection) have been resolved. The next step is to deal with the other three types of artifacts (i.e. hole & gap, misorientation, and interior geometry) simultaneously using a volumetric method which has already been explained in Section 1.2 and Section 2.2.

As an answer to the first research sub-question raised in Section 1.2, the mesh domain represented by the input model is spatially partitioned into a set of tetrahedra by applying constrained tetrahedralization (CT) on it. The techniques in 2D (i.e. constrained triangulation) has been well developed. However, this problem is far from solved in 3D as there exist polyhedra (see Figure 3.6) which can only be tetrahedralized with Steiner points and, moreover, the optimal number of Steiner points is still not figured out [Si, 2015].

Despite the issues above, there are several well developed softwares available which can preform CT on a polygon mesh, e.g. TetGen. TetGen is an open software with its source code freely available here. In this project, we are using TetGen for CT construction. For technical details of CT in TetGen, we refer to [Si, 2015].

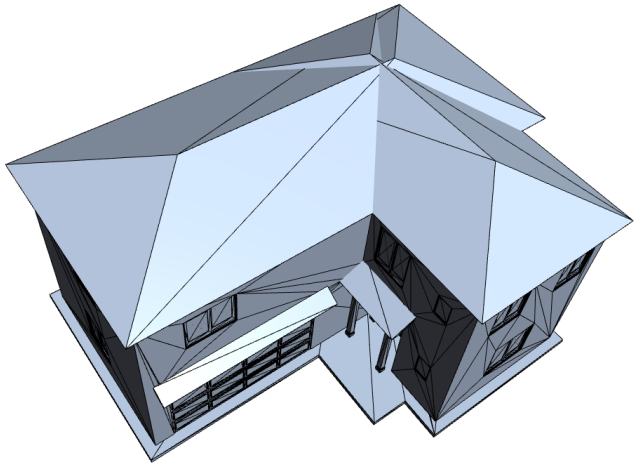
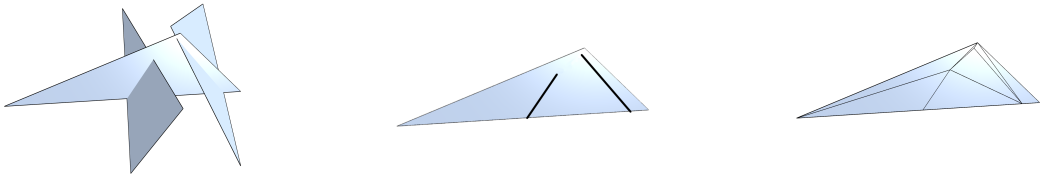


Figure 3.3: Self-intersecting roof surfaces



(a) A simplified example (b) Constructed intersections (c) Remeshed problematic face

Figure 3.4: Remeshing method. (a) A triangle intersected by two other triangles (a subset extracted from Figure 3.3). (b) Two intersecting line segments (bold) are constructed. (c) DT is preformed on vertices of both original triangle and constructed geometries.

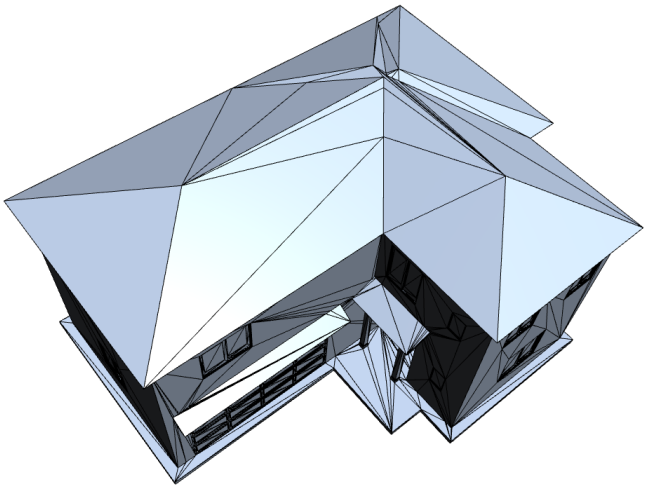


Figure 3.5: Remeshed roof surfaces

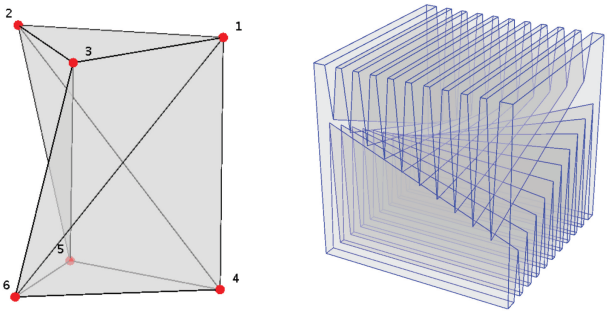



Figure 3.6: Polyhedra which can not be tetrahedralized without Steiner points. Adapted from [Si, 2015].

### 3.3.1 Behavior control

Before using TetGen, there are a few prerequisites for the input polygon mesh: `1)` no duplications and `2)` no self-intersections, which have been dealt with during the previous step. TetGen accepts a parameter string consisting of dozens of switches to control the behavior of tetrahedralization. In this project, we are using the following switches for reasons described behind:

- `-Y` Preserves the input surface mesh (does not modify it). Although the element quality of the result mesh may not be optimal, it doesn't pose a problem to later steps.
- `-M` No merge of coplanar facets or very close vertices. In this way, small features of the input model can be preserved to the utmost extent.
- `-c` Retains the convex hull of the PLC. TetGen has the build-in functionality to filter out the exterior tetrahedra based on the spatial connectivity, i.e. a tetrahedron is judged as being exterior if it is connected to the outer space. Thus, cavities of a flawed model will be throwed away. We keep all the terahedra and re-classify them based on our improved classification method.
- `-p/1e-7` Tetrahedralizes a piecewise linear complex (PLC) and set the minimum dihedral angle for detection of self-intersecting faces to be 1e-7.
- `-T1e-18` Sets a tolerance small enough to ignore warnings when two very close points are found.
- `-n` Retains tetrahedra neighbors for further use in Section 3.5.

### 3.3.2 Properties

The aforementioned parameters for behavior control are selected according to our demand with respect to the tetrahedralization. In this project, the input model with deficiencies are expected to be transformed into an intermediate tetrahedralization which preserves all the features in the original model. However, the quality of each tetrahedron is not important and has no effect on the shape of the outer surface. Nonetheless, the number of tetrahedra is desired to be minimized for the sake of the efficiency of our program.

The tetrahedralized exemplary model is shown in Figure 3.7. Here we list two main properties of it:

- `Constrained` Input constraints including facets and edges are strictly preserved without subdivision. Steiner points may be added, but only exist in the interior of the input model (those present on the input boundary are either removed or shifted into the interior). In this way, the extracted outer surface resembles the original model to an upmost extend. As a compromise, the quality of generated tetrahedra are not gauranteed. A large number of tetrahedra of irregular shapes (see Figure 3.8) might exist. Thus, the constructed tetrahedralization might not be Delaunay. However, this is acceptable since our methodology does not rely on the actual shapes of the tetrahedra.

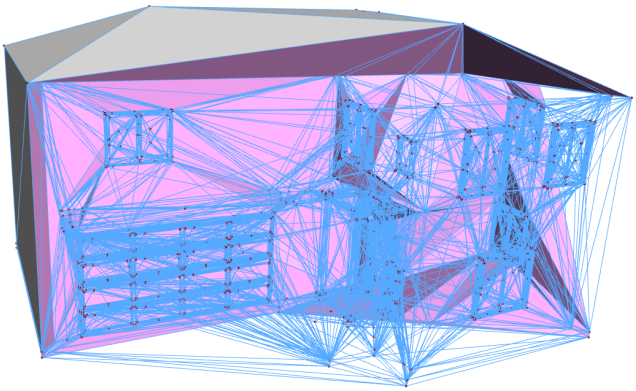


Figure 3.7: Tetrahedralized exemplary model (convex).

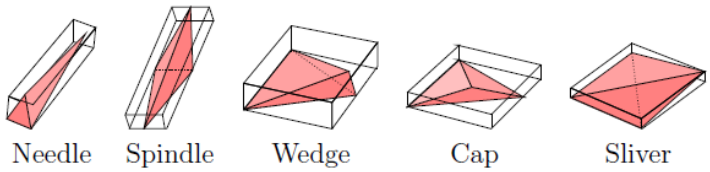



Figure 3.8: Tetrahedra of irregular shapes. Adapted from [Si, 2013].

- `Convex` This tetrahedral mesh is convex, i.e. the concave regions are also filled with tetrahedra which are not supposed to be part of the model domain. For a closed input model, such tetrahedra are exactly excluded by the original surfaces, hence can be identified by infection from the outside (i.e. growing until reaching model surfaces). These tetrahedra can be directly removed afterwards. But for models with holes, tetrahedra in cavities will be infected as well, which might greatly undermine the integrity of the original model. Thus, we decide to keep all tetrahedra from the original tetrahedral mesh, leaving the concave tetrahedra to the next classification step.

## 3.4 Interior/exterior classification

After CT, the 3D domain represented by the exemplary model has been partitioned into a set of tetrahedra. The next step is to determine, for each of them, whether it lies inside of the model domain or outside.

There exist several methods for the interior-exterior classification of a tetrahedron (or a point) with respect to a polygon mesh in 3D space as being mentioned in Section 2.2. However, they all have certain limitations, e.g. generalized winding number proposed by Jacobson et al. requires consistent orientation of the input model, visibility-based method proposed by Nooruddin and Turk can not handle models with complex interior structures, and the build-in method used in TetGen can only function well if there is no holes on the model surface. We hereby propose an improved classification method based on [Tzounakos, 2019], which is independent of any artifacts listed in Section 1.1.1.

The main idea of Tzounakos's method is that of ray casting classification in 2D. For a given 3D point,

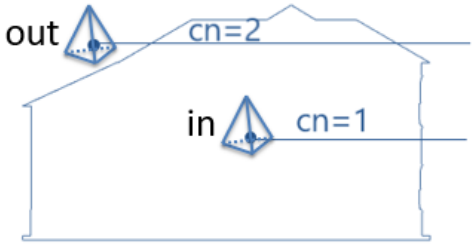
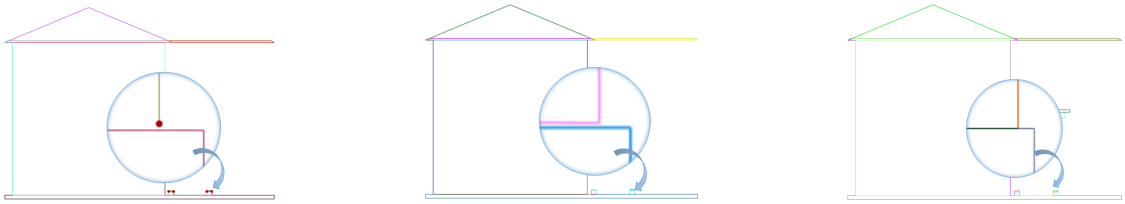



Figure 3.9: Ray casting classification in 2D (cn: crossing number)



(a) Atypical gap (b) Gap being wrongly closed (c) Our optimized result

Figure 3.10: Atypical gap when triangulation fails and our improved method by computing the closest point.

first a cross section passing through it is created which, at the same time, intersects the model and results in dozens of intersecting line segments. From these line segments, a closed polygon is extracted representing the boundary of them. Based on the extracted polygon, the given point can be classified by casting a ray from it and counting the crossing number between the ray and the polygon (see Figure 3.9).

However, the method presented in [Tzounakos, 2019] contains various issues in terms of robustness and efficiency. Therefore, we have made some improvements as follows:

**Closing gaps** Once the cross section is made and the intersecting line segments are extracted, it is possible that there exist some open vertices, i.e. vertices connected by only one line segment. This is very likely due to holes or gaps existing in the original model. In order to provide a valid input for ray casting classification, those open vertices have to be dealt with. This is the so-called *closing gaps* precedure in the original paper.

Tzounakos performs triangulation over all open vertices within a cross section and keeps the shortest incident edge for each open vertex. This method is designed for typical holes which is assumed to be relatively small so that the open vertices usually appear as pairs. However, cases exist that the hole is represented by a large dangling face as shown in Figure 3.10a. This is possible when a modeler fails to stitch the two faces. In this case, the open vertex is isolated where the aforementioned assumption fails. Moreover, if such open vertices appear multiple times, the original triangulation method may introduce incorrect line segments which will destroy the expected result thoroughly (see Figure 3.10b).

To better handle cases where isolated open vertices are encountered, a closest point (lying on line segments not containing the open vertex) to the open vertex is computed and connected to it. In this way, the isolated open vertex is connected to the closest point which Our optimaized result is shown in Figure 3.10c.

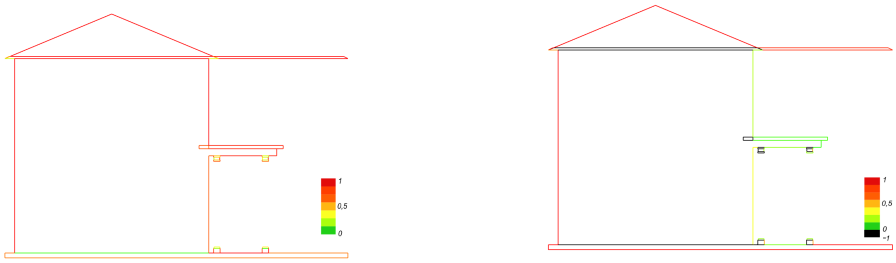


Figure 3.11: Result comparison between voting criteria of Tzounakos's (left) and ours (right).

**Twin ray voting** Once the gaps are closed, we can extract some connected components (i.e. polylines) from line segments. A subset of these components will be then selected to form the 2D boundary. The criteria for selection are boundary similarity (a voted value ranging between 0 and 1) and manifoldness in 2D. In the original methodology, a component is more likely to be a boundary if the intersection numbers of twin rays modulo 2 are not equal, otherwise not. This method is not robust even with a considerable number of twin rays generated (in original paper, about 100 twin rays are used for voting). However, for our project, this is way more time-consuming since there often exist tens of thousands tetrahedra for a single model and we can not afford to do so many times of classification. Thus, we have adapted the decision for calculating boundary similarity as follows:

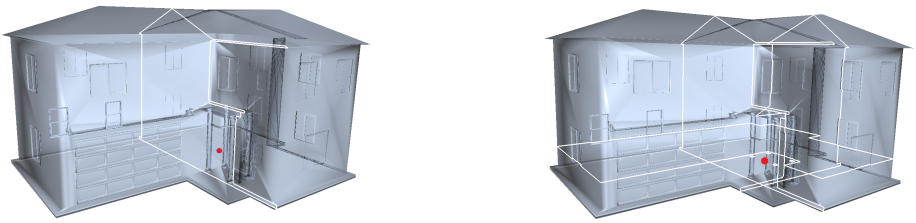
*A component is surely a boundary when there is no intersections for one of the twin rays. A component is probably* ***not*** *a boundary when twin ray intersection numbers modulo 2 are equal.*

This improved decision is tested robust with only four twin rays and hence is approximately 20 times faster than the original method. A comparison between the original and our adapted criteria is shown in Figure 3.11.

**Interior closed components removal** After twin ray voting, several closed components (i.e. polylines with its two vertices being identical) are created, some of which might be inside of another. Those interior closed components need to be removed. In the original method, these components are detected by applying a threshold on their boundary similarities, i.e. a component is decided interior if the boundary similarity is smaller than 0.5. This criterion is non-robust threoretically. We have adapted it by geometrically checking if a component lies inside of another.

**Ray casting classification** After the 2D boundary is extracted, ray casting method is used for the final classification. However, this classification method fails for a practical case as shown in Figure 3.12. In this case, the target point is located between two pillars which is classified as being inside of the 2D boundary if only one cross section is used. This is obviously non-robust since such cases are quite common. Thus, we have improved this method by employing 2D boundaries in three dimensions. The target point can only be classified interior if it is so for all three planes (i.e. x constant, y constant, and z constant).

The intermediate result after optimized classification is shown in Figure 3.13. At this stage, all interior tetrahedra have been identified.



(a) One dimension (b) Three dimension

Figure 3.12: Result comparision between classification method of Tzounakos's (left) and ours (right). The red point stands for the target. The white polylines are extracted 2D boundaries.

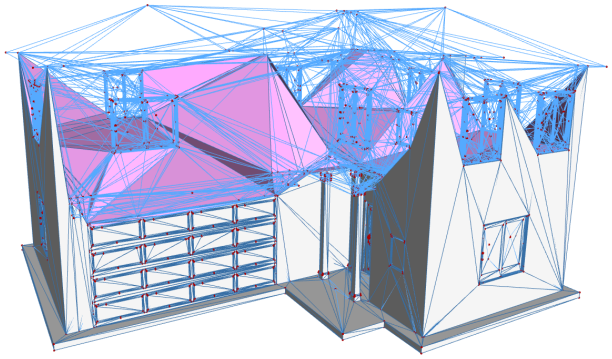
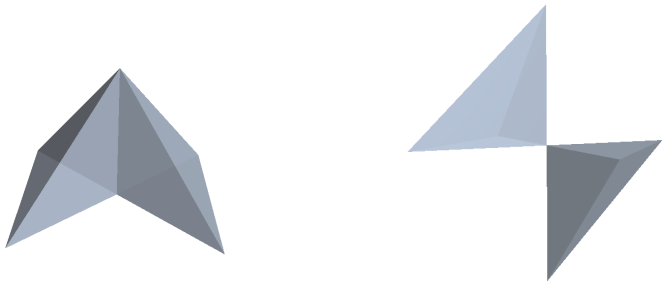


Figure 3.13: Tetrahedralized exemplary model after classification.



(a) Non-manifold edge (b) Non-manifold vertex

Figure 3.14: Non-manifoldness in extracted outer surface. (a) Two (or more) interior tetrahedra are incident to an edge but share no faces in between them. (b) Two (or more) nonadjacent tetrahedra are incident to a vertex.

## 3.5 Outer surface extraction

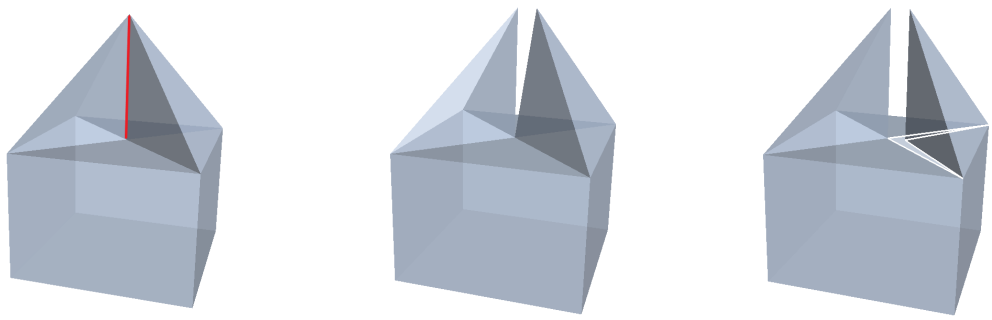
The intermediate result presented in Figure 3.13 are a collection of all interior tetrahedra from which the outer surface of our exemplary model can be trivially extracted. The idea is to select triangles that are incident to tetrahedra labelled as interior and exterior. Artifacts of misorientations, holes (gaps), and interior geometries are all handily resolved during this stage. Since every tetrahedron is naturally watertight, the outer surface of this intermediate tetrahedralization is so as well. However, non-manifoldness issues might still exist in extreme cases which requires the extracted triangle mesh to be post-processed.

### 3.5.1 Non-manifoldness removal

Although the tetrahedron is manifold itself, a combination of them is often not the case. Examples are shown in Figure 3.14. Such cases prevent the extracted triangle mesh from being a manifold and must be solved before outputing the final outer surface. Our strategies are, `a)` for non-manifold edge, we duplicate the edge such that each pair of faces incident to it shares one common edge, `b)` for non-manifold vertex, we duplicate the vertex such that every star of it shares one common vertex.

In some cases, boundary edges are introduced after edge duplication, meaning the extracted triangle mesh no longer watertight. Such non-manifold edges are only allowed for one vertex to be splited, and the incident faces over the other vertex automatically form a closed triangle fan (see Figure 3.15). In this project, all non-manifold edges are duplicated for both vertices. Hence boundary edges could be created inevitably and it is necessary to perform another operation to stitch such boundary edges.

Stitching is performed after duplication, and is the last step of our methodology. To stitch boundary edges, we first find all pairs of boundary edges that are geometrically identical and pointing to the opposite directions. Then, for a pair of boundary edges, their vertices are checked to see if they can be both merged or not. If the two pairs of endpoints can be both merged without introducing non-manifoldness, we create a new edge, adjust topological structures over it, and remove the other two boundary edges. The exemplary result is shown in Figure 3.16.



(a) Non-manifold edge (b) Duplicating one vertex (c) Duplicating two vertices

Figure 3.15: Edge duplication introduces boundary edges. Duplicated vertices are shifted for illustrative purposes. (a) Non-manifold edge are rendered red. (b) Duplicating the upper vertex results in a manifold. (c) Duplicating two vertices introduces boundary edges (rendered white).

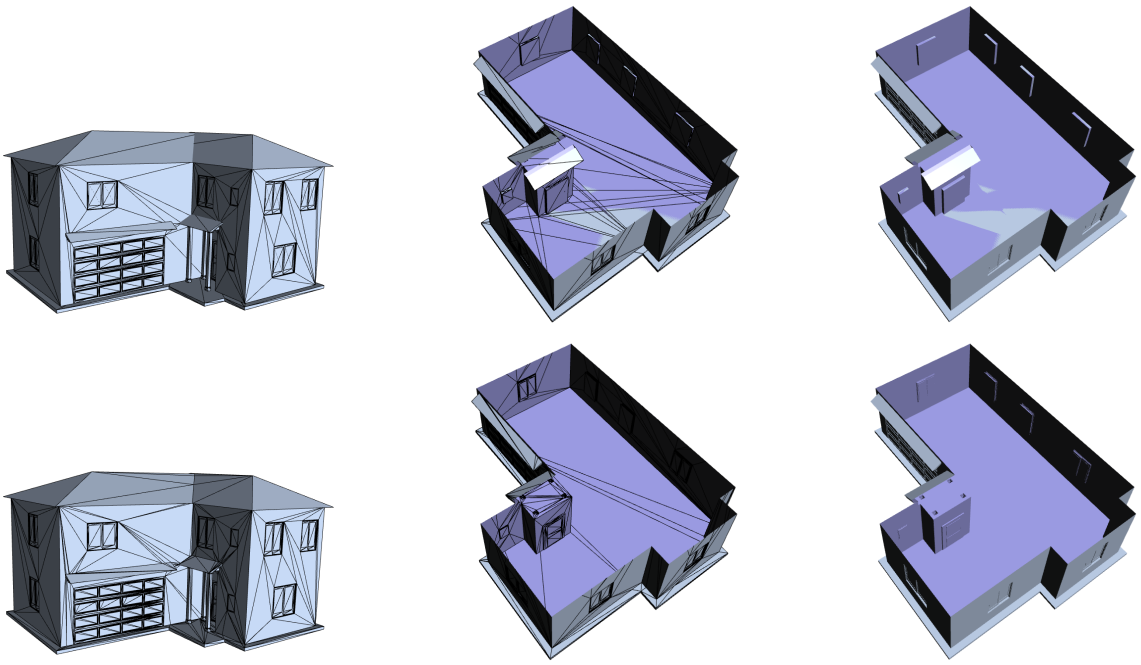


Figure 3.16: Outer surface of the exemplary model. Top: original model. Bottom: Outer surface. Left: Wireframe model. Middle: Clipped wireframe model. Right: Clipped model.

# 4 Implementation

This chapter describes the implementation details of our methodology including programming specifics and dramatic optimizations for efficient performance. The method proposed in [Tzounakos, 2019] is fully re-implemented with improvements in terms of efficiency and robustness. Optimizations including grouping and reuse are introduced which plays a critical role in reducing the execution time of our program.

## 4.1 Programming specifics

From a programming perspective, the source code of this project is written in C++ and mainly contributed by three third parties: `1)` Computational Geometry Algorithms Library (CGAL), `2)` Easy3D, and `3)` TetGen. exact_predicates_exact_constructions (EPEC) kernel and 2D constrained Delaunay triangulation (CDT) from CGAL are used for exact arithmetic and extraction of 2D boundary respectively. Easy3D provides ready-made C++ code for solving non-manifoldness, duplications, and self-intersections. TetGen helps construct CT from pre-processed model.

### 4.1.1 Exact arithmetic

The source code for pre-processing operations is migrated from Easy3D [Nan, 2018] with mesh data structure being replaced by CGAL Surface Mesh, in order to use EPEC kernel. The reason behind this is a fundamental problem with respect to geometric algorithms. Traditionally, exact algorithm is implemented by inexact floating-point arithmetic, which occasionally leads to incorrect results for correct input data due to rounding errors. However, the basic data structure used to represent polygon mesh in Easy3D is implemented using floating point numbers which can only guarantee limited precision. Such an inexact data type often leads remeshing operation (see Section 3.2.2) to fail. Remeshed triangles (see Figure 3.4c) might be judged intersecting again afterwards, since the positions of the exact constructed vertices will shift slightly when represented by inexact data type. A practical example is that, in Maple[1], self-intersections could still be detected for an already remeshed model. To avoid rounding errors, in this project, exact arithmetic is used during mesh loading, pre-processing, and classification (not CT since TetGen uses `double` as the underlying data type. Yet it is still a direction for further improvement towards a fully exact program). The only drawback of using exact arithmetic is that computing exactly is more time-consuming than that using floating-point numbers.

### 4.1.2 2D CDT

As is mentioned in Section 3.4, classification consists of two steps: `a)` 2D boundary extraction and `b)` ray casting classification. The underlying concept of it builds on the method proposed in [Tzounakos, 2019].

During 2D boundary extraction, 2D CDT from CGAL is used which can automatically handle intersecting constraints and duplicate geometries if `Exact_intersections_tag` is chosen. With the class `Triangulation_vertex_base_with_info_2`, we can also add custom information to vertices, e.g. the constrained degree (number of incident constraints), which is used to identify open vertices. 2D boundary

[1] A software developed using Easy3D.

extraction requires to find all connected components. A connected component is simply a polyline. The connected components are traced among triangulation in the following way:

- If the constrained degree of a vertex equals 2, we trace two sub-components in directions indicated by two incident constraints and connect them as one.
- Otherwise, we trace components in directions indicated by all (1, 3 or more) incident constraints.

In addition, the intersections between constraints are constructed using exact arothmetic as is explained in Section 4.1.1 which guarantees the extracted 2D boundary is exactly conforming to the input model.

### 4.1.3 AABB tree

During our implementation, axis-aligned bounding box (AABB) tree is used multiple times for 1) computing intersection of a plane (i.e. cross section) against triangles, 2) querying the number of intersections of a ray against sets of segments (twin ray voting), 3) computing the closest point of an open vertex (closing gaps), and 4) calculating the shift distance of a vertex in the outer surface.

For closest point computation, as the AABB tree itself contains the target open vertex, the result will be the vertex itself. CGAL doesn't provide ready-made application program interface (API) which can ignore certain primitives while querying. Therefore, we have written a new member function of AABB tree which allows to pass a skip functor as parameter to ignore the segment to which the open vertex belongs when the tree is being built. In this way, the closest point computation can provide correct result.

## 4.2 Practical optimizations

During Section 3.4, classification has to be performed three times (for three dimensions) for every tetrahedron. Since there might exist a large number of tetrahedra, the running time of classification procedure is not guaranteed and might be considerably long. Therefore, we have proposed two optimization methods in order to reduce the number of needed classifications and avoid unnecessary constructions of 2D boundaries.

### 4.2.1 Grouping of tetrahedra

As a matter of fact, the 3D model naturally divides the 3D space into several regions (polyhedra) based on its faces. A region, in our program, is then subdivided into a set of tetrahedra which share no faces or subfaces of the original model in between. Therefore, two adjacent tetrahedra are doubtless to be in the same region if their shared face is not a face or subface of the original model. Such tetrahedra can be briefly described as being spatially connected. Based on this decision, all the tetrahedra can be roughly grouped into a set of regions. Among these grouped regions exists a particular one, in which all tetrahedra are connected to the exterior space of the original model (see Figure 4.1).

Regions other than that are protected by the original model, i.e. isolated from the exterior space. Those regions are guaranteed to be inside of the model domain, hence there is no need to perform classifications on them. As for the exterior region shown in Figure 4.1, there might exist some tetrahedra which, although spatially connected to the outer space, are part of the model domain in effect. In other words, these tetrahedra are exactly the holey regions of the original model. To recover tetrahedra in holey regions, all tetrahedra in the exterior region have to be classified. Nevertheless, this is already a small portion of the original number.

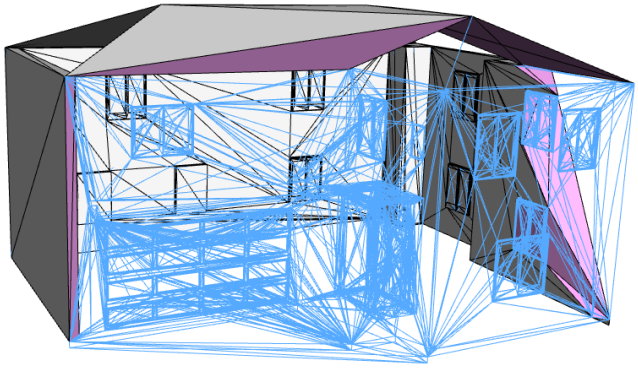
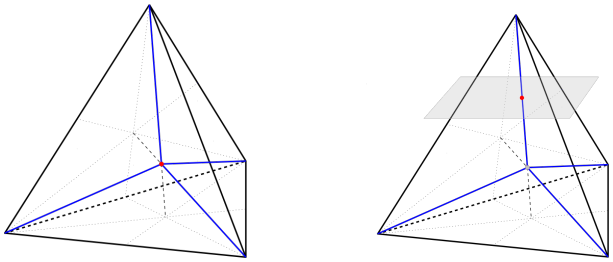


Figure 4.1: Exterior region of the exemplary model



(a) (b)

Figure 4.2: Two kinds of representative points. (a) If no suitable 2D boundaries for reuse, we construct a new crossing plane passing through the centroid. (b) If there is a suitable 2D boundary, we compute a new representative point for classification.

### 4.2.2 Reuse of 2D boundaries

The most time-consuming part of a classification cycle is the construction of the 2D boundary. Thus, reusing previously constructed 2D boundaries also contributes to the efficiency of our program. The idea is that, for a tetrahedra to be classified, if there exists an already constructed 2D boundary that is applicable, then reuse it rather than constructing a new one. A 2D boundary is considered applicable for a specific tetrahedron if they intersect (other than touch) each other. In this way, lots of time will be saved with respect to 2D boundary constructions.

However, this brings another issue. Previously, a tetrahedron is represented by its centroid during classification. Since the centroid lies on the cross section, it is coplanar with the constructed 2D boundary. Now, a previously reconstructed 2D boundary is selected for reuse, which is unlikely coplanar with the centroid of the current tetrahedron. Hence the centroid of the current tetrahedron can not represent its position when being classified by the reused 2D boundary. Another representative point should be constructed and this point must be inside the tetrahedron as well as coplanar with the reused 2D boundary. In our implementation, this point is computed by intersecting the supporting plane of the 2D boundary with one of the four medians[2] of the target tetrahedron (see Figure 4.2). In extreme cases that the supporting plane is coplanar with one of the medians, the centroid is applicable for classification, i.e. no need to compute another representative point.

In this way, quite a number of 2D boundaries avoid being constructed. An example is shown in Figure 4.3

[2] A line segment joining a vertex with the centroid of the opposite face.

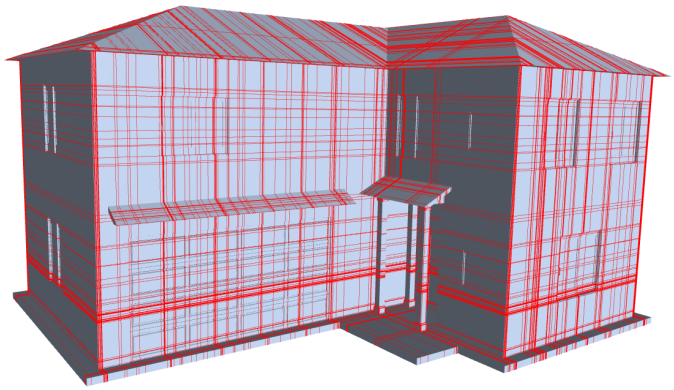


Figure 4.3: All constructed 2D boundaries for classification

where only 281 2D boundaries are constructed while, formerly, this number is expacted to be 29604.

# 5 Results & Evaluation

In this chapter, a few building models with a variety of artifacts mentioned in Section 1.1.1 are processed by our program and the results are presented. From these extracted outer surfaces, statistics with respect to the number of simplices[1], the optimized number of classifications, and execution time are presented and discussed.

## 5.1 Results

As is shown in Chapter 3, our program is designed in a structural way which allows the intermediate result at each stage to be retained and the corresponding statistics to be outputed. As is stated in Section 1.2, the goal of this project is to extract a closed and manifold outer surface out of the error-ridden 3D building model. All types of artifacts should be removed as well as the interior geometries. The result should exactly represent the input mesh domain. However, the extracted outer surface may not be homeomorphic to a sphere of which the genus is 0 since the input model could be concave and have complex structures. Moreover, no assumption of single input model is made. As a result of that, it is possible that the result consists of more than one object if the original model contains multiple disjoint components, e.g. windows. Thus, this method is potentially able to handle inputs containing multiple solids.

So far, the input data for our program are all building models from [Fan and Wonka, 2016]. These models are triangle meshes constructed from photographs and each of them contains a certain combination of the aforementioned artifacts (see Table 5.1). Degeneracy is not considered a type of artifacts, but poses problems for tetrahedralization. Hence we detect and remove degenerated faces during pre-processing. The absence of such faces have no effect on the integrity of the input model. The number of duplications refers to duplicated faces that are detected and removed during pre-processing, excluding the faces that are kept as the one for each set of duplications. Statistics of mis-orientations and interior geometries are not collected yet but do exist in all tested models. These models are processed by our program in a fully automatic way. In Figure 5.1, the visualized results after every main step in our methodology are presented and they are arranged in the same order as that of Table 5.1.

## 5.2 Evaluation

In this section, the underlying statistics of the results shown in Figure 5.1 will be presented and discussed in terms of optimization and validity. After that, we compare our method with three state-of-the-art methods

### 5.2.1 Optimization

As is stated in Section 4.2, two optimization operations have been designed in order to quicken the program so that the execution time is acceptable. For a pratical model, the whole processing cycle could

[1] points, edges, and triangles

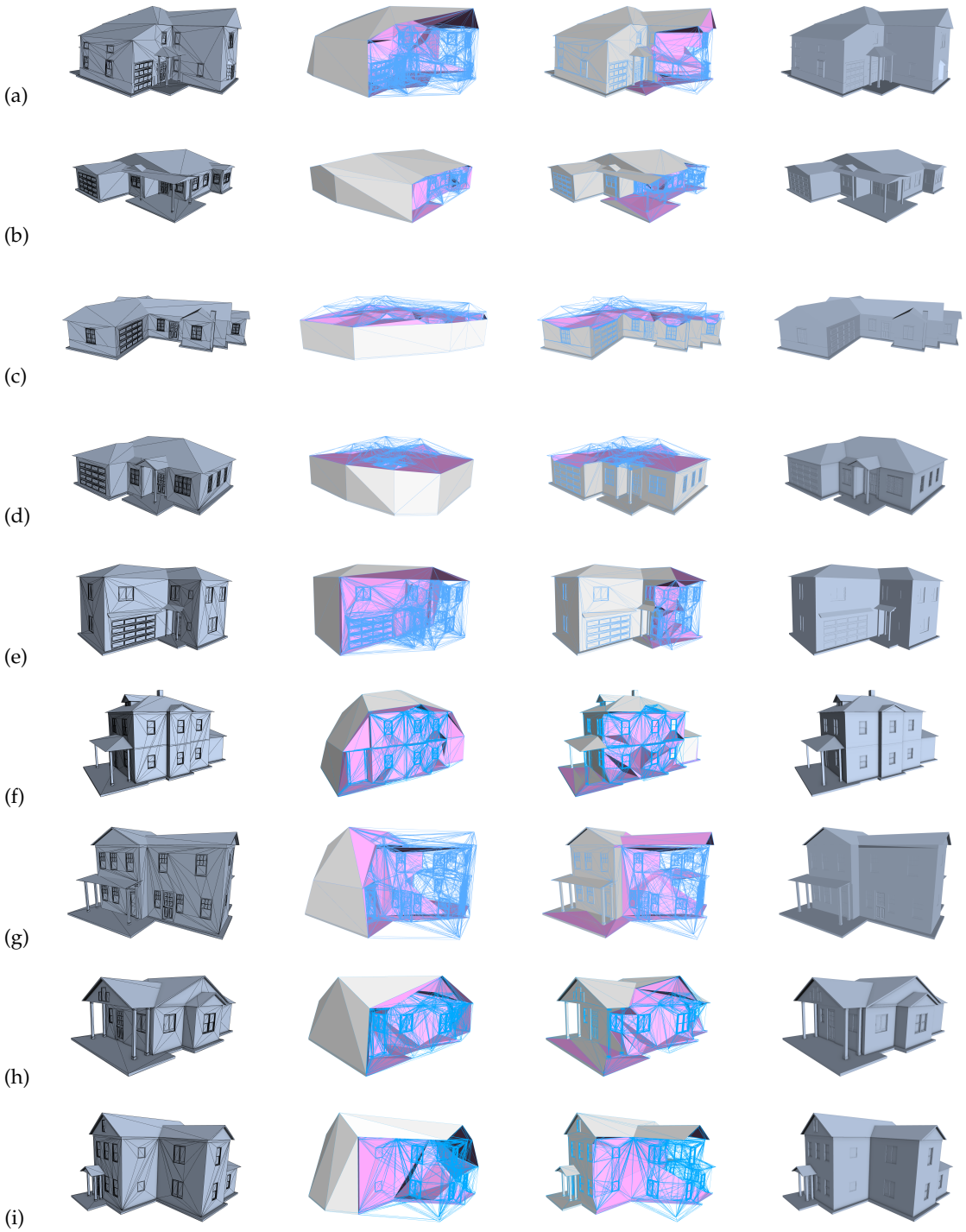



Figure 5.1: Stepwise results of our program. From left to right: input model, original CT, classified CT, outer surface. (a) ∼ (i): task_1_x, task_2_x, task_4_x, task-1-x, task-2-x, task-7-x, task-8-x, task-11-x, task-12-x).

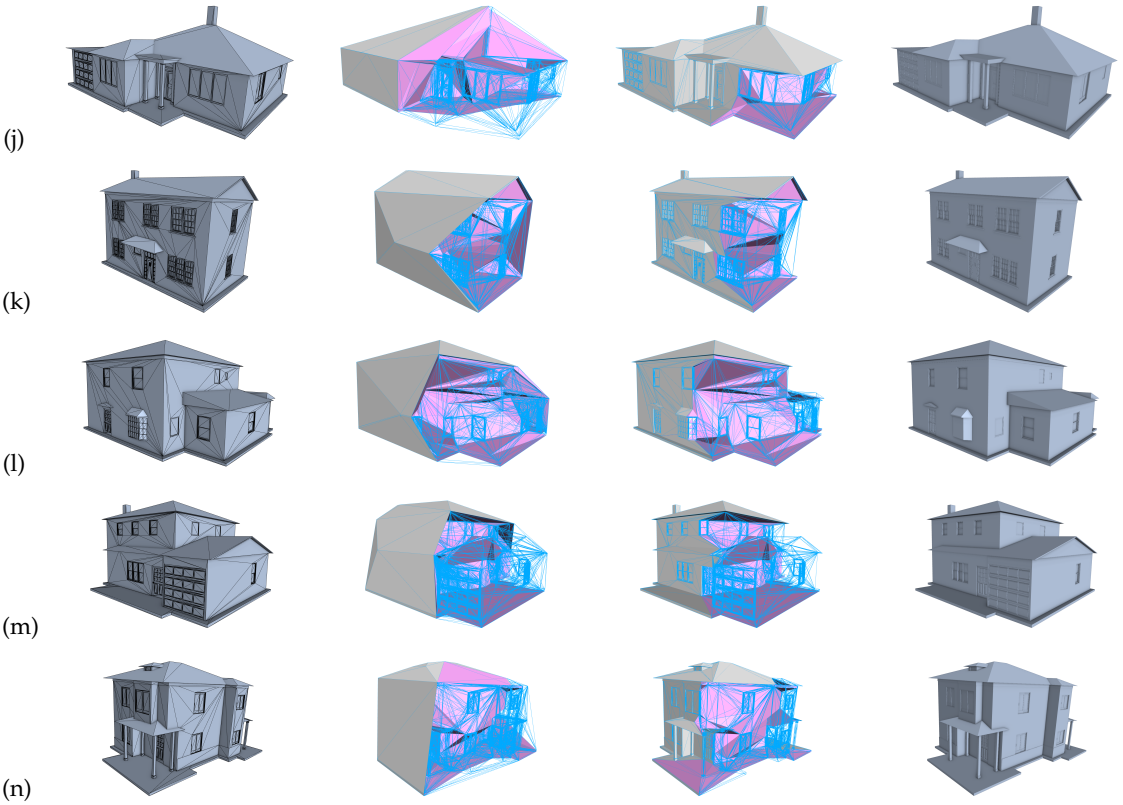



Figure 5.1: Stepwise results of our program (cont.). From left to right: input model, original CT, classified CT, outer surface. (j) ∼ (n): task-16-x, task-17-x2, task-19-x, task-20-x, task-21-x).

| | f | v | degeneracy | duplication | self-int | NM e | NM v | BD e |
|---|---|---|---|---|---|---|---|---|
| **task_1_x** | 11262 | 7970 | 607 | 908 | 807 | 1405 | 2 | 845 |
| **task_2_x** | 13386 | 8698 | 770 | 1710 | 638 | 2071 | 1 | 779 |
| **task_4_x** | 28759 | 28002 | 355 | 12990 | 6086 | 1148 | 141 | 20100 |
| **task-1-x** | 16724 | 27188 | 426 | 90 | 1745 | 342 | 64 | 9665 |
| **task-2-x** | 9048 | 7520 | 283 | 672 | 90 | 514 | 4 | 409 |
| **task-7-x** | 7120 | 6300 | 209 | 52 | 421 | 241 | 34 | 394 |
| **task-8-x** | 15759 | 10394 | 867 | 2034 | 259 | 2462 | 1 | 918 |
| **task-11-x** | 5751 | 4856 | 218 | 236 | 170 | 343 | 13 | 382 |
| **task-12-x** | 13058 | 15198 | 228 | 98 | 2170 | 394 | 44 | 3862 |
| **task-16-x** | 6640 | 4882 | 273 | 194 | 37 | 495 | 0 | 264 |
| **task-17-x2** | 12730 | 14918 | 647 | 36 | 2194 | 172 | 4 | 3291 |
| **task-19-x** | 6435 | 5950 | 316 | 184 | 137 | 272 | 25 | 793 |
| **task-20-x** | 6721 | 5834 | 256 | 96 | 114 | 227 | 31 | 425 |
| **task-21-x** | 9267 | 7307 | 351 | 714 | 251 | 665 | 1 | 508 |

Table 5.1: Statistics of input errors. f: number of triangles. v: number of vertices. degeneracy: number of denegerate triangles. duplication: number of duplicated triangles. self-int: number of self-intersecting triangles. NM e: number of non-manifold edge. NM v: number of non-manifold vertices. BD e: number of boundary edges. Misorientation and holes also exist in every model, but are not counted yet.

| | CT | grouping | reuse | LoO (‰) |
|---|---|---|---|---|
| **task_1_x** | 43530 | 12564 | 275 | 2.1 |
| **task_2_x** | 39983 | 11089 | 273 | 2.3 |
| **task_4_x** | 110550 | 61035 | 886 | 2.7 |
| **task-1-x** | 104543 | 56800 | 794 | 2.4 |
| **task-2-x** | 38459 | 9834 | 274 | 2.4 |
| **task-7-x** | 42774 | 12257 | 381 | 3.0 |
| **task-8-x** | 41911 | 11543 | 324 | 2.6 |
| **task-11-x** | 27293 | 7689 | 279 | 3.4 |
| **task-12-x** | 67098 | 22334 | 590 | 2.9 |
| **task-16-x** | 24241 | 9027 | 286 | 3.9 |
| **task-17-x2** | 50180 | 16863 | 345 | 2.3 |
| **task-19-x** | 33472 | 7989 | 314 | 3.1 |
| **task-20-x** | 35282 | 9494 | 320 | 3.0 |
| **task-21-x** | 39958 | 10221 | 292 | 2.4 |

Table 5.2: Statistics concerning optimization operations. LoO: level of optimization.

take hours without applying these optimizations.

`Grouping` The first operation is to group all the tetrahedra into a set of regions, each of which is spatially connected in the interior. By grouping, most regions can be determined as being inside of the model immediately as they are protected by the model surface (i.e. not connected to the outer space). Thus, the number of needed classifications is greatly reduced since we do not have to classify tetrahedra in these regions. Among these regions, one particular region has been identified which connects to the outer space of the model. This region consists of tetrahedra forming `1)` concave places and `2)` cavities (see Figure 4.1), which are supposed to be exterior and interior respectively. All tetrahedra in this region have to be classified. The latter, i.e. tetrahedra in cavities, are exactly what should be collected.

`Reuse` The other operation is to reuse pre-constrcuted 2D boundaries for classification. It does not reduce the number of needed classifications but makes classifications more efficient. This operation greatly contributes to the performance of our program since 2D boundary construction is the most time-consuming part during classification.

The statistics with respect to optimizations are shown in Table 5.2. First the number of original tetrahedra that need to be classified is given. Then the reduced number after CT and `grouping` is presented, followed by the actual number of construcued 2D boundaries. At last, paramters indicating the level of optimization (LoO) are calculated as dividing the number of constructed 2D boundaries by the number of needed 2D boundaries. The number of needed 2D boundaries are 3 times of the number of tetrahedra in CT, since every classification is performed by constructing 2D boundaries in three planes. The corresponding execution times for each step are collected in Table 5.3.

From the statistics in Table 5.2, it can be observed that the two optimization operations has dramatically improved the efficiency of our program. The actual number of constructed 2D boundaries has reduced approximately 400 times. Even though, the execution time for classification is still relatively long, taking up most (about 97%) of the total execution time.

### 5.2.2 Validity

Although efficiency is important in effect, the most significant properties of the results are the geometric and topologic validities, i.e. whether they are watertight and manifold.

| | pre-process | CT | classification | OS extraction | in total |
|---|---|---|---|---|---|
| **task_1_x** | 23.8 | 1.7 | 938.3 | 0.2 | 964.2 |
| **task_2_x** | 24.0 | 1.8 | 957.0 | 0.2 | 984.1 |
| **task_4_x** | 33.4 | 3.8 | 5820.4 | 0.2 | 5859.5 |
| **task-1-x** | 32.6 | 3.7 | 4338.7 | 0.2 | 4376.0 |
| **task-2-x** | 21.9 | 1.5 | 598.7 | 0.2 | 622.3 |
| **task-7-x** | 25.0 | 2.2 | 1838.0 | 0.2 | 1865.5 |
| **task-8-x** | 28.3 | 1.7 | 1219.8 | 0.2 | 1250.2 |
| **task-11-x** | 18.7 | 1.0 | 751.0 | 0.1 | 770.9 |
| **task-12-x** | 37.3 | 2.9 | 2759.1 | 0.4 | 2799.8 |
| **task-16-x** | 12.4 | 0.6 | 595.2 | 0.1 | 608.5 |
| **task-17-x2** | 30.7 | 2.3 | 1521.2 | 0.3 | 1554.8 |
| **task-19-x** | 19.1 | 1.2 | 967.0 | 0.2 | 987.6 |
| **task-20-x** | 19.5 | 1.4 | 1115.0 | 0.2 | 1136.2 |
| **task-21-x** | 23.0 | 1.1 | 914.8 | 0.2 | 939.2 |

Table 5.3: Execution times (s)

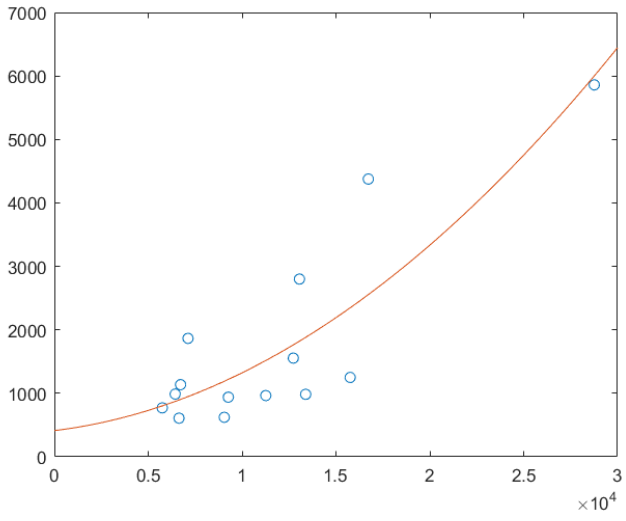



Figure 5.2: Estimated execution time w.r.t. the number of input faces.

| | f | v | degeneracy | duplication | self-int | NM e | NM v | BD e |
|---|---|---|---|---|---|---|---|---|
| **task_1_x_os** | 8166 | 4069 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task_2_x_os** | 7260 | 3630 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task_4_x_os** | 28998 | 14147 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-1-x_os** | 27276 | 13336 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-2-x_os** | 6018 | 2963 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-7-x_os** | 8904 | 4348 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-8-x_os** | 7358 | 3665 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-11-x_os** | 4970 | 2437 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-12-x_os** | 14790 | 7279 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-16-x_os** | 4642 | 2323 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-17-x2_os** | 11110 | 5550 | 0 | 0 | 0 | 0 | 0 | 18 |
| **task-19-x_os** | 6178 | 3025 | 0 | 0 | 0 | 0 | 0 | 0 |
| **task-20-x_os** | 7358 | 3591 | 0 | 0 | 0 | 0 | 0 | 8 |
| **task-21-x_os** | 5990 | 2961 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 5.4: Validity check

In computer graphics, watertight meshes usually refer to that forming closed objects, i.e. no holes exist on the surface, and manifoldness means that everywhere on the surface mesh is locally homeomorphic to a disk. These formal definitions which are quite intuitive, are rather problematic when being implemented in pratice. From a programming perspective, another set of definitions is easier to operate. That is, a polygon mesh is watertight if there is no boundary edges presented and manifold if all edges and vertices are manifold as well. Such definitions, though looks abstract, are indeed handy for validating meshes programmatically.

There are ready-to-use tools in existing softwares (e.g. Mapple and MeshLab), which support the detection of boundary edges and non-manifoldness. Thus, the corresponding validity check is not implemented within our program. We have checked for validity all our results in MeshLab and the corresponding result is presented in Table 5.4.

From Table 5.4, it can be seen that there are still boundary edges present in `task-17-x2_os` and `task-20-x_os`, which is due to the "pinching" strategy we use for stitching boundary edges see Figure 6.1. Nevertheless, our method removes nearly all the artifacts present in the testing models, and generates watertight and manifold outer surfaces for 12 models out of 14. `Degeneracy`, `duplication`, and `self-intersection` are detected using exact arithmetic (see Section 4.1.1) during the execution of our program in order to produce reliable statistics. When being exported as files, such artifacts might appear again due to round-off errors.

### 5.2.3 Comparison

We have compared our approach with three recently proposed methods, in particular, TetWild [Hu et al., 2018], MeshViewer [Chu et al., 2019], and ManifoldPlus [Huang et al., 2020]. Although these methods might have certain requirements for the input model and target at different properties for the output, we try to compare with them based on the goal of our project (see Table 5.5). The first objective is the validity of the output, for which the numbers of non-manifold edges, non-manifold vertices, and boundary edges are collected. `Degeneracy`, `duplication`, and `self-intersection` are not addressed since the detection of such artifacts are easily affected by the round-off errors which are easily introduced during importing and exporting operations. Besides non-manifoldness and boundary edge, we also collect statistics with respect to execution time and deformation. Execution time reflects the practical value of an algorithm. Deformation of a processed model is calculated as the average shift

| | NM e | NM v | BD e | time | deformation |
|---|---|---|---|---|---|
| **original** | 767.9 | 26.1 | 3045.4 | - | - |
| **TetWild** | 726.5 | 252.4 | 0 | 1315.9 | 2.3e-5 |
| **MeshViewer** | 112.8 | 0.6 | 281.5 | 625.4 | 8.2e-3 |
| **ManifolfPlus** | 0 | 0 | 0 | 31.8 | 4.1e-6 |
| **ours** | 0 | 0 | 1.9 | 1765.6 | 1.6e-7 |

Table 5.5: Comparison with state-of-the-art methods. All values are averages of the 14 building models.

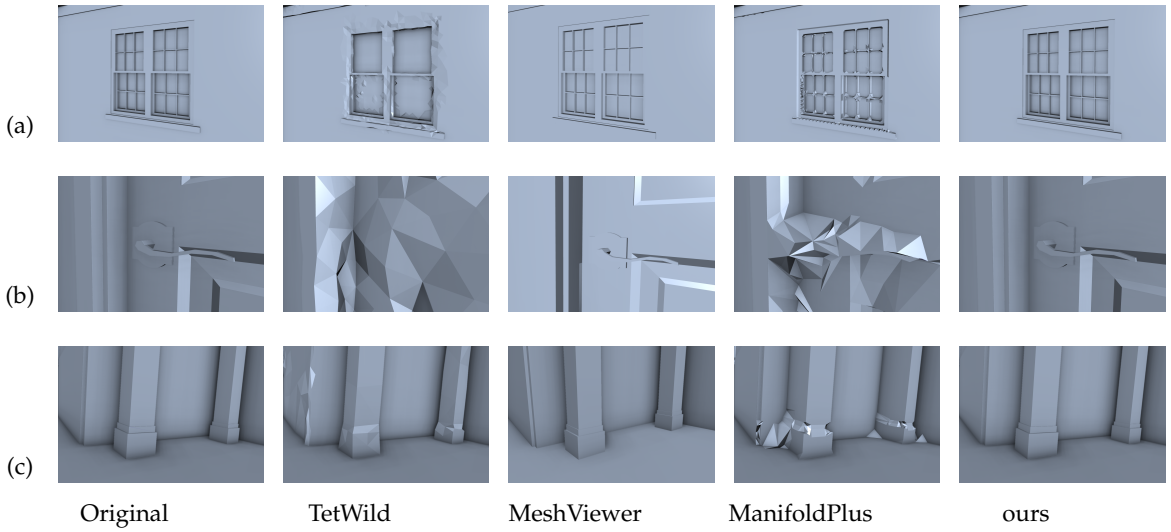



Figure 5.3: Close-ups of mesh details. (a) Window. (b) Handle. (c) Pillars.

distance of its vertices comparing to the original model, and indicates to what extent the integrity of original model is preserved. These two statistics are usually inversely proportional to each other, i.e. an algorithm usually spends more time in order to achieve less deformation. However, we know that the average shift distance can not completely describe the deformation of a processed model, e.g. a vertex could be displaced onto another face, but the calculated shift distance is zero. Therefore, we compare our result with others by close-ups at details (see Figure 5.3).

From Table 5.5 and Figure 5.3, we can see that TetWild and MeshViewer are far from success in terms of the validity of their results. Our method produces acceptable results with only a few boundary edges present in two models. The reason of such errors is that we only implement "pinching" operation to resolve non-manifold edge, while cases exist where "pinching" introduces boundary edges that can not be further stitched and "snapping" is needed (see Figure 6.1). ManifoldPlus guarantees valid results for all testing models. With respect to the efficiency, our method is similar to TetWild, i.e. relatively time-consuming, whilst ManifoldPlus is the fastest among the four methods. As for the integrity, TetWild and ManifoldPlus can not preserve small features very well. The results from MeshViewer looks plausible, but the deformation is the worst in effect. By contrast, our method preserves the details of the original model to the greatest extent.

# 6 Conclusions

In the end, the research question of our thesis is reviewed and the pros and cons of our methodology is discussed along with its potential applications.

## 6.1 Research overview

At this stage, our proposed methodology has been applied on real models in practice and results are retained with the corresponding statistics analysed. It is time to review the research question we have defined for our graduation project:

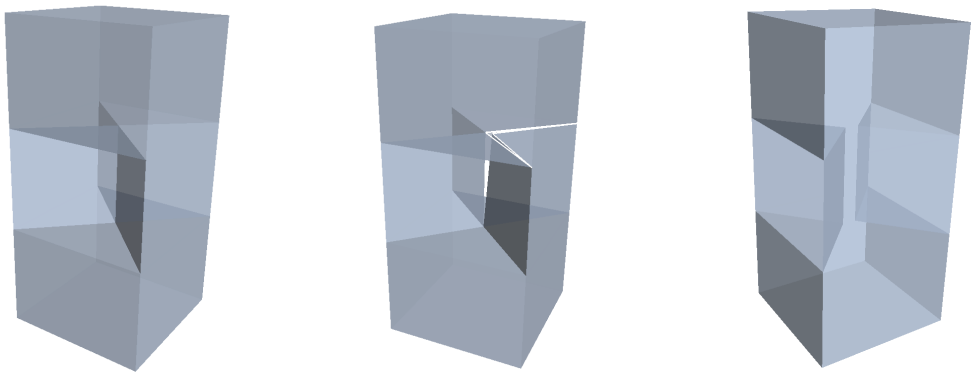
***How can we accurately extract a watertight and manifold outer surface from an error-ridden 3D building model?***

Well, before giving more detailed answer to this research question, we firstly confirm the feasibility of it. That is, a watertight and manifold outer surface **can** be extracted from an error-ridden 3D model. The particular method used in our thesis is hybrid, i.e. involving both surface-oriented and volumetric operations. The main steps of it can be roughly described as follows (where sub-questions are answered in the meantime):

- `Pre-processing` In this step, two types of errors are tackled, i.e. duplication and self-intersection. The purpose of this step is to meet the requirements of the next step, i.e. CT construction.
- `Constrained tetrahedralization` In this step, a CT is constructed out of the pre-processed model from the previous step. As a result, the model domain as well as the concave areas are divided into a set of tetrahedra. Faces of the original model are preserved and embedded in between them.
- `Classification` The purpose of this step is to find all tetrahedra that lie inside of the model domain. The classification method is built on [Tzounakos, 2019] and the performance has been greatly improved by re-implementation and optimization. The general idea of it is to extract a sliced closed boundary and classify by performing ray-casting method based on this closed polygon. More details can be found in the original paper.
- `Outer surface extraction` The last step is relatively simple since the boundary faces are implicitly identified once all tetrahedra are classified. The outer surface is composed of triangles incident to interior and exterior tetrahedra which naturally forms a (or multiple) watertight object(s). At this stage, the extracted outer surface might still be non-manifold. Thus, a post-processing step including duplicating and stitching is performed in order to get a watertight manifold.

## 6.2 Discussion

With respect to model repair, traditional methods usually assume the input model to meet certain requirements. For instance, method proposed by [Hu et al., 2018] heavily depends on the input orientation. Faces with opposite orientation will lead to incorrect result for generalized winding number classification. Another case is that of TetGen, which also has a classification functionality in order to remove exterior tetrahedra when generating CT. However, TetGen is not able to handle holes on the surface. If a hole is presented in the input model, the neighboring faces will not be preserved. Compared



(a) Non-manifold edge (b) Pinching (c) Snapping

Figure 6.1: An exemplary non-manifold edge for which "pinching" introduces boundary edges that can not be further stitched. Boundary edges are rendered white.

with other volumetric methods, our method possesses the following superior properties:

- `Non-parametric` No input parameter is needed.
- `Automatic` No user interactions during the processing and no manual edit afterwards.
- `No assumptions` Different from other methods as being mentioned in Section 2.2, our method poses no requirements for the input model.
- `Accurate` Small features are preserved without any simplifications or modifications.

Apart from the advantages of our methodology, there are certain limitations being noticed in practice.

- `Time-consuming` As is presented in Figure 5.2, for a single model with nearly 10,000 faces, it takes more than 10 minutes to get the outer surface. The execution time becomes even longer (exponentially) when the number of input faces increases.
- `Not exact` Exact arithmetic is always used except the construction of CT due to the fixed underlying data structure of TetGen. Therefore, exact numbers are truncated when being imported into TetGen. As a result of it, there is a chance that TetGen rejects the pre-processed model as invalid.
- `Non-robust resolution on non-manifoldness` As is shown in Figure 2.3, there are two strategies for stitching: "pinching" and "snapping". The former tends to stitch boundary edges that belong to the same solid, so that the model is split into two parts along this edge. Whilst, the latter stitches boundary edges of separate solids and makes the interior topologically connected. In our methodology, the way we resolve non-manifold edges resembles that of "pinching". However, cases exist where "pinching" fails and "snapping" is needed (see Figure 6.1).
- `Bumpy` Since we are using tetrahedra to fill the holes presented in the input model, the quality of the fixed region might not be smooth enough comparing to its neighbors. Using polyhedra to partition the model domain, e.g. hexahedra, may produce better result.

## 6.3 Future work

Based on the aforementioned limitations, we hereby come up with some future works aiming at improving the performance and robustness of our methodology.

- **Complete exact arithmetic** Exact arithmetic robustly supports geometric predicates and constructions, in particular, detecting and resolving self-intersections. EPEC kernel from CGAL is used for this purpose. However, due to the limitations of the external library we use, this kernel is not used all through the program. Exact coordinates of vertices are translated to floating point numbers when constructing CT, hence might be displaced afterwards. Such displacements have great influence on geometric calculations and might lead geometric predicates to their opposite. Therefore, it is desired to use exact numbers throughout the program when implementing our method.
- **Non-manifoldness removal** As is presented in Table 5.4 and Figure 6.1, there are still cases where our method fails to produce a watertight output. Problem exists in the last step when we duplicate non-manifold edges and try to stitch them using a "pinching" strategy. In future, such non-manifold edges must be identified and classified beforehand in order to apply suitable stitching approaches.
- **Multi-threading** As is shown in Table 5.3, classification takes up more than 90 percent of the total execution time. Since classification is performed each time for a single tetrahedron and all classifications are mutually independent, it is desired to perform classification in a multithreaded way in order to improve the efficiency. However, the EPEC kernel we use is currently not multi-threading safe. How to implement our methodology by multi-threaded programming is a direction of practical values.

# Bibliography


Biljecki, F., Ledoux, H., and Stoter, J. (2016). An improved lod specification for 3d building models. *Computers, Environment and Urban Systems*, 59:25–37.

Bischoff, S., Pavic, D., and Kobbelt, L. (2005). Automatic restoration of polygon models. *ACM Transactions on Graphics (TOG)*, 24(4):1332–1352.

Borodin, P., Zachmann, G., and Klein, R. (2004). Consistent normal orientation for polygonal meshes. In *Proceedings Computer Graphics International, 2004.*, pages 18–25. IEEE.

Botsch, M., Pauly, M., Kobbelt, L., Alliez, P., Lévy, B., Bischoff, S., and Röossl, C. (2007). Geometric modeling based on polygonal meshes.

Chu, L., Pan, H., Liu, Y., and Wang, W. (2019). Repairing man-made meshes via visual driven global optimization with minimum intrusion. *ACM Transactions on Graphics (TOG)*, 38(6):158.

Dick, A. R., Torr, P. H., and Cipolla, R. (2004). Modelling and interpretation of architecture from several images. *International Journal of Computer Vision*, 60(2):111–134.

Fan, L. and Wonka, P. (2016). A probabilistic model for exteriors of residential buildings. *ACM Transactions on Graphics (TOG)*, 35(5):155.

Guéziec, A., Taubin, G., Lazarus, F., and Hom, B. (2001). Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):136–151.

Hu, Y., Zhou, Q., Gao, X., Jacobson, A., Zorin, D., and Panozzo, D. (2018). Tetrahedral meshing in the wild. *ACM Trans. Graph.*, 37(4):60–1.

Huang, J., Zhou, Y., and Guibas, L. (2020). Manifoldplus: A robust and scalable watertight manifold surface generation method for triangle soups. *arXiv preprint arXiv:2005.11621*.

Jacobson, A., Kavan, L., and Sorkine-Hornung, O. (2013). Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (TOG)*, 32(4):33.

Ju, T. (2004). Robust repair of polygonal models. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 888–895. ACM.

Liepa, P. (2003). Filling holes in meshes.

Murali, T. and Funkhouser, T. A. (1997). Consistent solid and boundary representations from arbitrary polygonal data. *SI3D*, 97:155–162.

Musialski, P., Wonka, P., Aliaga, D. G., Wimmer, M., Van Gool, L., and Purgathofer, W. (2013). A survey of urban reconstruction. In *Computer graphics forum*, volume 32, pages 146–177. Wiley Online Library.

Nan, L. (2018). Easy3d: a lightweight, easy-to-use, and efficient c++ library for processing and rendering 3d data. `https://github.com/LiangliangNan/Easy3D`.

Nan, L., Sharf, A., Zhang, H., Cohen-Or, D., and Chen, B. (2010). Smartboxes for interactive urban reconstruction. In *ACM SIGGRAPH 2010 papers*, pages 1–10.

Nooruddin, F. S. and Turk, G. (2000). Interior/exterior classification of polygonal models. In *Proceedings Visualization 2000. VIS 2000 (Cat. No. 00CH37145)*, pages 415–422. IEEE.

Nooruddin, F. S. and Turk, G. (2003). Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205.

Ripperda, N. and Brenner, C. (2009). Application of a formal grammar to facade reconstruction in semiautomatic and automatic environments. In *Proc. of the 12th AGILE Conference on GIScience*, pages 1–12.

Shen, C., O'Brien, J. F., and Shewchuk, J. R. (2005). Interpolating and approximating implicit surfaces from polygon soup. In *ACM Siggraph 2005 Courses*, page 204. ACM.

Si, H. (2013). A quality tetrahedral mesh generator and 3d delaunay triangulator version 1.5. *User's Manual*.

Si, H. (2015). Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):1–36.

Tzounakos, N. (2019). Robust interior-exterior classification for 3d models. Master's thesis.

Vanegas, C. A., Aliaga, D. G., and Beneš, B. (2010). Building reconstruction using manhattan-world grammars. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 358–365. IEEE.

Vanegas, C. A., Aliaga, D. G., and Benes, B. (2012). Automatic extraction of manhattan-world building masses from 3d laser range scans. *IEEE transactions on visualization and computer graphics*, 18(10):1627–1637.

Werner, T. and Zisserman, A. (2002). New techniques for automated architectural reconstruction from photographs. In *European conference on computer vision*, pages 541–555. Springer.

Xiao, J., Fang, T., Zhao, P., Lhuillier, M., and Quan, L. (2009). Image-based street-side city modeling. In *ACM SIGGRAPH Asia 2009 papers*, pages 1–12.

Zhou, Q.-Y. and Neumann, U. (2008). Fast and extensible building modeling from airborne lidar data. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–8.

Zilske, M., Lamecker, H., and Zachow, S. (2007). Adaptive remeshing of non-manifold surfaces.

## Colophon

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