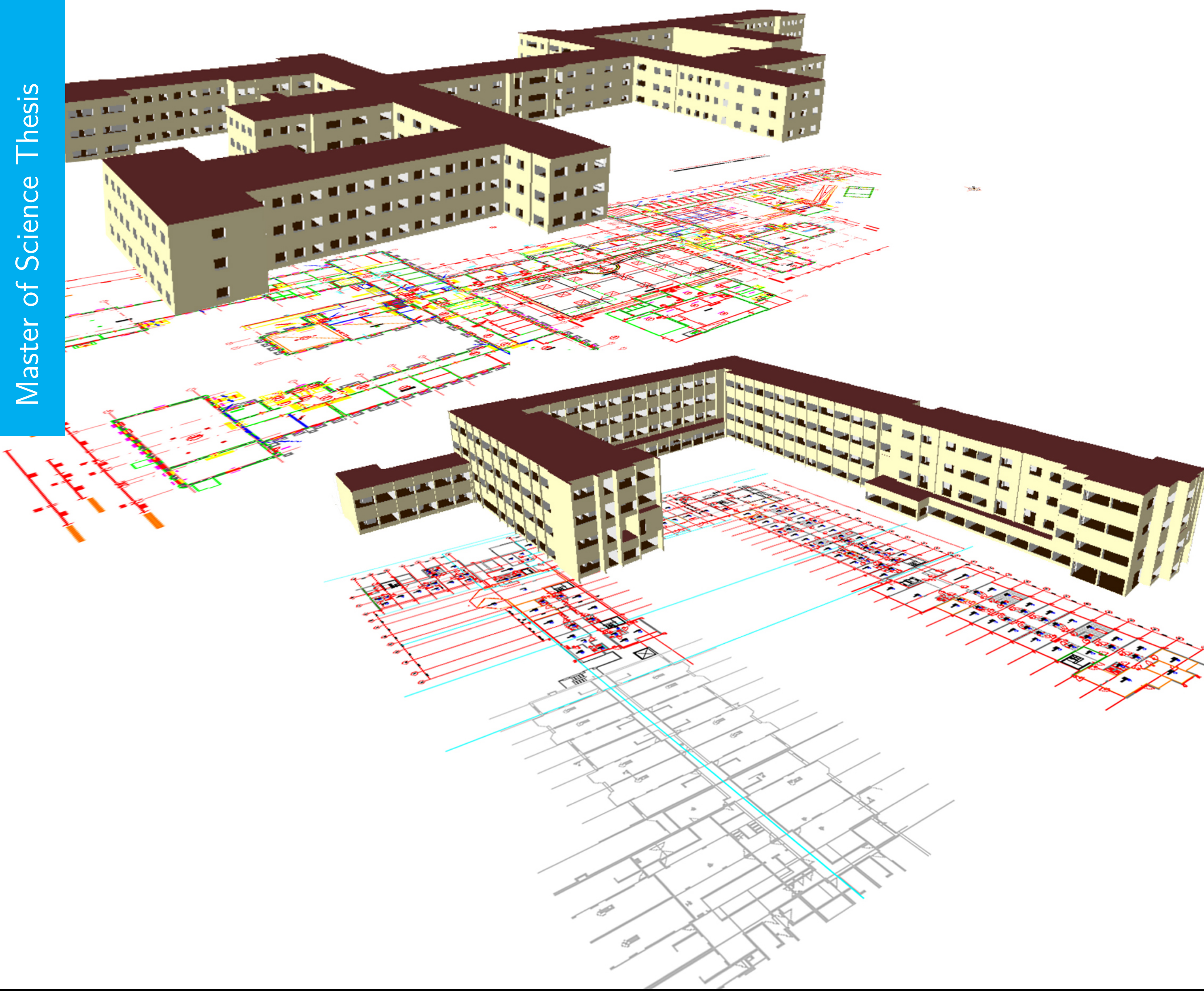
Integration of 2D architectural floor plans into Indoor OpenStreetMap for reconstructing 3D building models

Haoxiang Wu

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Integration of 2D architectural floor plans into Indoor OpenStreetMap for reconstructing 3D building models

Master of Science Thesis

For the degree of Master of Science in Geomatics at Delft University of Technology

Haoxiang Wu

November 2, 2015

Faculty of Faculty of Architecture and the Built Environment (BK) · Delft University of Technology

The work in this thesis was supported by company MoreForYou. Their cooperation is hereby gratefully acknowledged.

Abstract

All existing data sources used to reconstruct 3D building models have certain restrictions. An eye-catching alternative is IndoorOSM, one of the most popular examples of the newly evolved crowdsourced geodata. The potential power of this rich and simple-formatted data source has been proven by many researches. However, a fatal flaw of IndoorOSM is also pointed out, which is its accuracy. Another promising data source that has been looked into is 2D architectural floor plans. They are also commonly available and full of detailed indoor information. Due to the inconsistencies and ambiguities existing among real-life floor plans, previous researches all are established on different user cases. Although sharing a common pipeline, they differ from each other in every step, from the data structure, processing procedure to the 3D reconstruction method. The combination of these two data sources can be beneficial, because architectural floor plans can offer IndoorOSM better accuracy and extensive indoor information while IndoorOSM can provide a unified data structure and 3D reconstruction workflow for information extracted from floor plans.

Based on a throughout review of the characters of real-life floor plans, a set of rules are proposed to redraw architectural floor plans from real life. These rules mainly focus on reorganizing information contained in floor plans, taking advantages of the layering and blocking functions supported by CAD application. The original geometry and graphical representation in the raw floor plans is reserved as much as possible. Redrawing is only required when unstandardized representation is encountered. Then, an automatic process is accordingly presented to extract desired information from the redrawn floor plans into an IndoorOSM database. Finally, highly detailed CityGML LoD4 models with interior structures can be generated using a method proposed by Dr. Marcus Goetz. The pipeline is tested with several floor plans from real life for 3D reconstruction. User feedback validated the feasibility and efficiency of the redrawing rules.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to thank my supervisors Ass.Prof.Dr.Sisi Zlatanova and Ph.D. Liu Liu for their guidance and support throughout this whole project. And I also want to thank Drs. C.W. Quak for his valuable opinions on the writing of this thesis and any other researchers who have given me suggestions. In addition, I owe my special thanks to Bart Kroesbergen and his team from company MoreForYou for their support of the data and guidance. Finally, I would like to thank all my classmates from Geomatics who have accompanied me through these two years of studying abroad.

Delft, University of Technology
November 2, 2015

Haoxiang Wu

Chapter 1

Introduction

1-1 Context

Nowadays, people spend more time indoor than outdoors, as a result of which the demand of indoor information increases more than ever [1, 2]. Tasks like indoor navigation and emergency management all require such information [3]. Several global companies, such as Bing [4], Google [5] and Navteq [6], have already made their efforts trying to seize this ever-growing market by providing indoor-related services or applications. However, for now most of them are limited to 2D indoor maps. On the other hand, 3D building models have been proved to be an efficient tool to present indoor environment. It provides an immersive visualization allowing people to virtually wander inside the building to have a more intuitive perspective [7, 8].

Normally, 3D building models can be obtained through Building Information Modeling (BIM), photogrammetry and LIDAR (Light Detection and Ranging). For designing purpose from an architect's view, architects use various architectural softwares, e.g. AutoCAD, Sketchup, Rhino, to create 3D building models by hand. Although very time-consuming and larbor-intensive, models created in this way are very exhaustive, showing the buildings from all aspects: internal, external, from beneath and from above [9]. Both aerial photogrammetry and LIDAR solved the problem of massive data collection [10]. Traditionally, 3D urban model can be obtained through semi-automatic interaction between proficient operators and aerial imagery on a photogrammetry station [11]. Recently, large-scale production has been achieved due to the emergence of more automated computerized tools [12]. Fig. 1-1 shows some building models that are reconstructed from aerial imagery. Methods using point cloud collected by LIDAR can be generally categorized into model-driven methods and data-driven methods. To put it simple, model-driven methods try to fit parameterized building models to the point cloud, while data-driven methods try to find parametric planes for building roofs in the point cloud, possibly grouped with buildings' orthogonal projection information (e.g. ground plans and cadastral maps), to reconstruct the building [13, 14, 15]. Fig. 1-2 illustrates the general steps in data-driven methods to extract a building's roof planes from point cloud. For photogrammetry and point cloud, in addition to possible human intervention

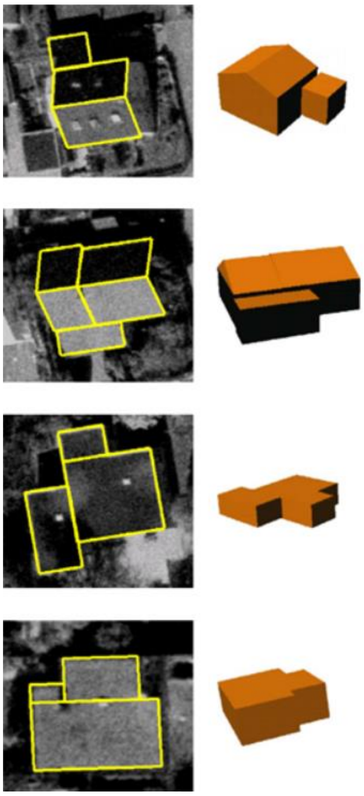
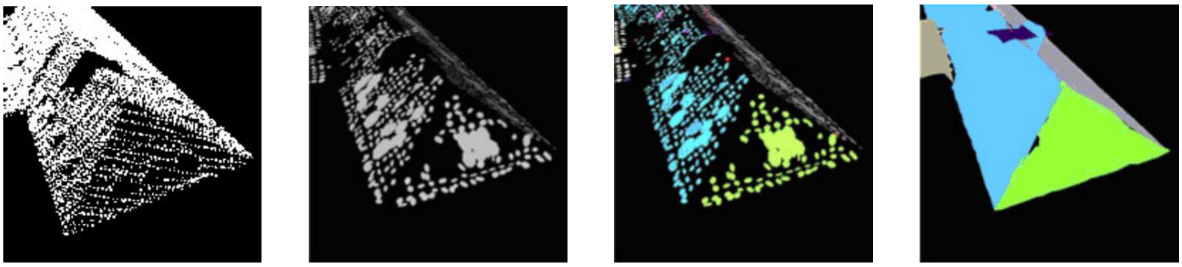


Figure 1-1: Reconstructed building models from aerial imagery [16]

and restrictive data resources that require professional instruments, a critical disadvantage is that only external facades of buildings can be rebuilt without any information about the interior environment. Thus, it is very necessary to find another data source that is commonly available and easily retrievable that can be used for automatic 3D reconstruction for buildings' indoor space.

2D architectural floor plans, as a standard way to express design by architects and are widely used in the field of architecture, are an extensively available data source with a wealth of information that is suitable for 3D reconstruction. Space subdivision is represented by simple planar geometries (usually lines and polygons) with notations of their measurements. Texts and symbols are used to present semantic information, such as name and usage of a subspace or texture of walls [18]. Furthermore, connectivity network of subspaces can be obtained by searching for connectors that are usually represented by specific symbols, such as doors, windows, staircases, elevators, escalators and slopes [19]. There are various formats of 2D architectural floor plans. Hand drawings, digital scanned copies and vector formats like CAD, all are very widespread. Another advantage of 2D architectural plan is its considerable accuracy since architectural floor plans are strictly drawn based on buildingsąŕ real dimensions under certain scale [20]. Therefore, 2D architectural floor plan is a very promising data source for automatic reconstruction of 3D building models with indoor information.



(a) (b) (c) (d)

Figure 1-2: Approximate roof modeling. (a) Segmented LIDAR points for a roof surface. (b) Local plane patches fit to the LIDAR points. (c) Planar patches grouped together based on similarity of normals. (d) Approximate boundaries of planar roofs. [17]

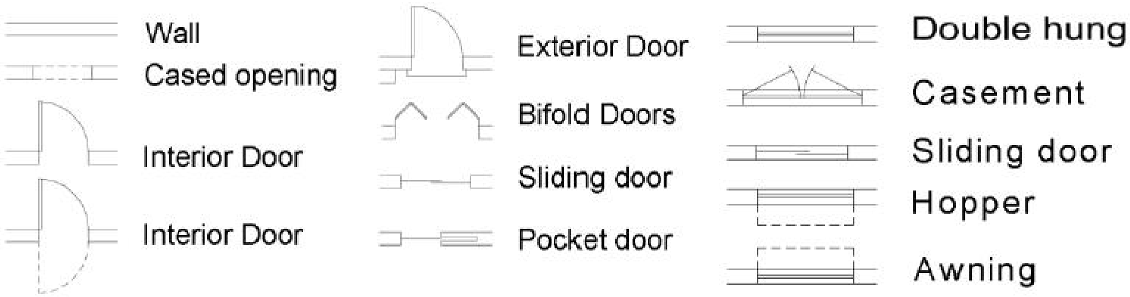


Figure 1-3: Reconstructed building models from aerial imagery [24]

1-2 Challenges in using floor plans for 3D reconstruction

Researchers have been trying to automate the process of reconstructing 3D building models from 2D floor plans. However, there are some challenges in doing so caused by the characters of real-life floor plans. The first one is the format of the input floor plans. Basically, there are two categories of floor plans in real life: paper-based and CAD-based [20]. In earlier times when CAD (Computer-aided Drafting or Computer-aided Design) is not yet popularized, floor plans were drawn manually by architects on paper. Some of these hand drawings were scanned and saved in digital format for archiving. Even nowadays, paper floor plans still dominate the architectural workflow [20]. Nevertheless, the use of CAD has tremendously increased the productivity of the designers and improved the quality of design [21]. There are various CAD software, open-source or proprietary, modelling for both 2D and 3D or solely 2D, being widely used in the field of architecture, MEP (Mechanical, Electrical and Plumbing) or structural engineering [22]. Depending on the software by which these computer-drawn floor plans are created they might be saved in different file formats [23]. Due to this reason, existing systems all are developed based on certain input format and can only achieve the expected results with their designated format. For systems that use paper-based floor plans as input, they have to take some extra steps by adopting certain image processing techniques to vectorize the floor plans before they can further deal with those extracted primitives like CAD-based systems do [20].

The second reason that impedes the realization of fully automation is the varying representa-

Figure 1-4: Examples of material symbols in floor plan [24]

Figure 1-5: Polymorphous representation of walls, windows and doors [20]

tion of the drawings, especially for those graphical symbols. These graphical symbols include symbols for architectural components (e.g. windows and doors), construction material (e.g. concrete and wood) and cross-references [24]. Fig. 1-3 shows some typical symbols of walls, doors and windows. Fig. 1-4 shows some examples of material symbols commonly used in floor plans. Although some common rules and standards of architectural floor plans have been developed for designers as explicit guidelines [25, 26, 27], it does not completely eliminate the ambiguities within real-life floor plans. It is because between these drawing standards and symbol libraries, differences exist. For a single semantic symbol, it can be represented in several different ways according to different standards [28]. Designers can choose freely among them based on their purpose for the drawing. Besides, since none of these standards are mandatory, in reality tiny differences might still exist in symbols that are claimed to be conformable with a particular standard, as designers might alter and adapt the representation of the symbols to some degrees to their own drafting habit and artistic incline [24]. Furthermore, the characters of a same symbol can change emphasizing on different aspects at different design stages with different level of details [29]. Fig. 1-5 illustrates this situation with four different possible representations of a same case, where there is a wall with one window and one door on it. Last but not least, these standards and guidelines are developed for architectural use. They might not be suitable for 3D reconstruction.

Unfortunately, existing researches, besides being restricted to the input format, focus on applying different techniques of image processing, symbol recognition and 3D reconstruction based on certain assumed representation of the floor plans. Their performance depends on how closed the floor plans are to the desired case. Very few analysis of real-life floor plans have been carried out to give a guideline for a standardized representation of the input floor plans to facilitate 3D reconstruction.

1-3 Objectives and research question

Based on the context described above, the goal of this thesis is to propose a (semi-)automatic process to extract information from 2D architectural floor plans in the form of IndoorOSM for 3D reconstruction. The main research question to be addressed is:

How to integrate 2D architectural floor plans as input data into the 3D reconstruction pipeline of IndoorOSM?

This research question can further be divided into three underlying questions:

1. What information about indoor environment is contained in real-life floor plans and among them which can be exported into IndoorOSM for 3D reconstruction?
2. In what way can the information to be used be extracted from the floor plans?
3. How should the extracted information be reorganized in the form of IndoorOSM?

1-4 Research scope

There are several things will not be considered in this thesis:

1. Input floor plans. The input floor plans in this thesis are 2D architectural floor plans. Any other floor plans designated for other application purpose will not be considered.
2. Building structure. Since a floor plan is an aerial plan view that is horizontally cut approximately 4 feet above the floor, it cannot fully present a buildingąŕs indoor spatial structure. In this thesis, only normal-structured buildings will be considered, which means there is no room on each floor that crosses several floors in the building. The height of every room cannot exceed the height of the floor it belongs to.
3. 3D reconstruction. This thesis will focus on analysis, processing and information extraction of 2D floor plans for 3D reconstruction. The 3D reconstruction will be carried out by algorithm developed by other researchers.
4. Geo-referencing. Usually architectural floor plans only contain measurements of a building without any information about its geographical location. Thus, models created in this thesis will not be geo-referenced or be geo-referenced manually.

1-5 Contributions

Generally, in this thesis, a complete pipeline for processing 2D architectural floor plans is proposed with the purpose of integrating 2D architectural floor plans into the 3D reconstruction workflow of IndoorOSM proposed by Dr. Marcus Goetz. By using architectural floor plans as input data source, the data source for 3D reconstruction that is currently very restrictive can be extended. Additionally, the relatively high accuracy of architectural floor plans in can help solve the biggest problem of crowdsourced data like IndoorOSM that the generated 3D models are often distorted since the accuracy of the data source is not guaranteed. In turns, IndoorOSM provides architectural floor plans with a unified data structure and 3D reconstruction workflow that the extracted information can be put into.

To be more specifically, previous researches on processing floor plans for 3D reconstruction, although share a common pipeline, all make their own assumptions or requirement on the input floor plans based on the data format or the methods they are going to use. In this thesis, based on a throughout review of the characters of real-life floor plans, a set of rules are proposed to redraw input floor plans to make them unified in terms of format, graphical representation and information segmentation, and compatible with the later 2D processing steps. These rules mainly focus on reorganizing information contained in floor plans, taking advantages of the layering and blocking functions supported by CAD application. The original geometry and graphical representation in the raw floor plans is reserved as much as possible. Redrawing is only required when unstandardized representation is encountered. Besides, auxiliary information and some other architectural information that are not used in the reconstruction process can also be saved in other separated layers, so that the redrawn floor plans are not proprietary and can still be used for other application purposes, such as instruction for construction.

Then, an automatic process is accordingly presented to extract desired information from the redrawn floor plans into an IndoorOSM database. This process includes drafting error fixing, wall reconstruction, opening and contour reconstruction. In drafting error fixing step, in addition to the problem of disjoint vertices that has been looked at by previous researches, some redundancies that might be contained in the floor plans are newly considered in this thesis. In wall reconstruction step, instead of recognizing parallel line pairs as walls, this thesis tries to group line segments in the wall layer into closed polygons. Although the topological information of walls is ignored, the problem of possible incorrect detections is avoided. In contour reconstruction step, more possible layout between openings and its adjacent walls that have not been considered in previous researches is looked into in this thsis.

1-6 Chapter overview

The rest of this thesis will be organized as follows:

Chapter 2 first gives a thorough analysis of real-life floor plans. All the representation possibilities of the information to be used in the floor plans will be summarized. Based on that, a preliminary conclusion that floor plans from real life must be redrawn according to certain rules to facilitate an automatic 3D reconstruction will be drawn. Next, an overview of some previous researches in this area is presented, which covers both the methods to process 2D architectural floor plans and their corresponding reconstruction algorithms. Decisions on what information should be used, in what way the information should be prepared for 3D reconstruction and the reconstruction method to be used are also made in this chapter based on the literature study.

Chapter 3 first illustrates how the input floor plans should be redrawn in detail, which includes the content to be kept, the specific representation of the symbols, the layering and the format. After that, each step of processing the redrawn floor plans is explained. After the information is extracted from the redrawn floor plans, it is exported to a database for 3D reconstruction.

Chapter 4 presents the implementation of the proposed algorithms and the testing results from several cases of real-life floor plans.

Chapter 5 concludes this thesis research.

Chapter 2

Background

This chapter introduces the background of the thesis. The characters of 2D architectural floor plans are presented first. Then a literature study of previous researches on extracting and reorganizing of 2D information in the floor plans is carried out to find out which method can contribute to this thesis and what needs to be improved. Last, some methods of reconstructing 3D models from 2D information are introduced and the method developed by Dr. Marcus Goetz is chose for this thesis.

2-1 Overview of real-life floor plans

In last chapter, it is mentioned that there are there are some challenges caused by the characters of real-life floor plans that impede the realization of fully automatic 3D reconstruction from 2D floor plans. They are basically the diversified formats of the floor plans and the varying representation of symbols in the floor plans. In this section, the representation problem will be explained in details. First, some commonly seen contents in architectural floor plans will be reviewed. Among them, walls, doors and windows are determined to be the most basic elements in the 3D reconstruction of building models with indoor information. Then, the representation of these three elements will be further introduced respectively.

2-1-1 Content

Fig. 2-1 shows example of an architectural floor plans commonly seen in real life. The contents in this floor plan can be generally divided into two categories: auxiliary and architectural. The auxiliary ones are center lines, dimension lines, numbers and texts. Center lines go through the center of objects (usually the main walls and openings), indicating their location and orientation. It usually connects with a circle, within which there is letter and/or number for its identification. Dimension lines are used to show the measurement of an object. It can be used to indicate length, width, diameter, etc. Fig. 2-2 is the zoom-in view of the area bounded by the dark blue box in Fig. 2-1. In Fig. 2-2, there are two center lines going

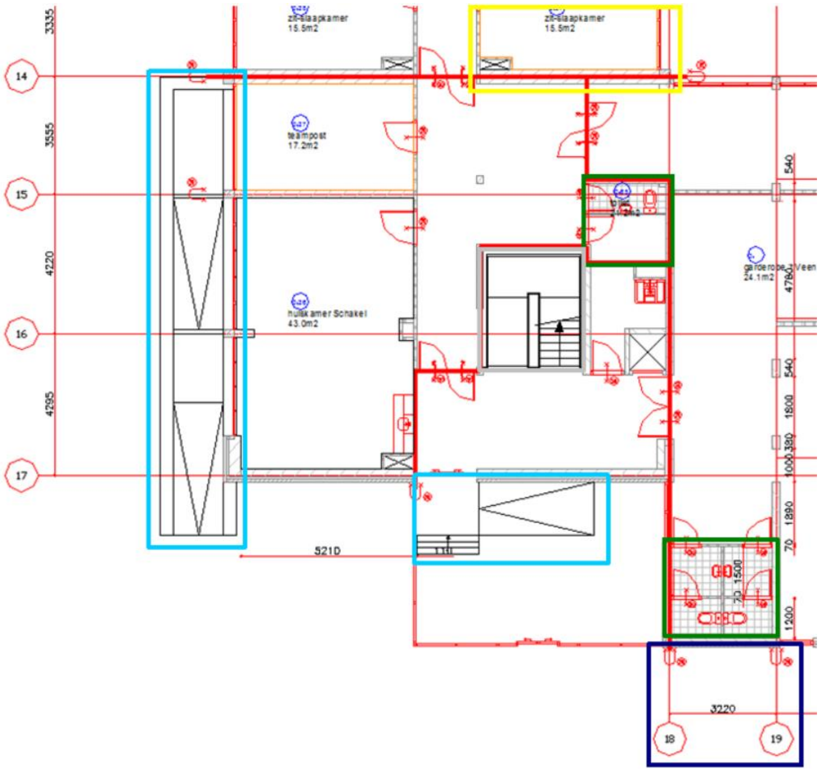


Figure 2-1: Example of an architectural floor plans

through two walls, between which there is a dimension line perpendicularly intersecting with them showing the distance between these two walls.

Besides being used to identify the center lines and to indicate measurements, texts and numbers are often used for additional description of objects. For example, in Fig. 2-3, which is the zoom-in view of the area bounded by the yellow box in Fig. 2-1, the room number, room type and the area of room are presented. Moreover, some other auxiliary elements, such as section lines, cross-reference symbols, opening number symbols, north arrows and legends, are also very commonly seen in architectural floor plans.

The architectural contents that can be recognized in Fig. 2-1 are walls and columns, windows and doors, material hatch patterns, stairs and elevators, furniture and facilities, and objects outdoors. Among them, walls and columns are the main structural entities in a building, which are also the most important elements in the 3D reconstruction constituting the skeleton of the building structure. Doors and windows are also indispensable elements in 3D reconstruction since they closure the contours of rooms formed by its surrounding walls. Besides, openings can be used to obtain the connectivity between rooms in indoor environment in further phases once a 3D model has been rebuilt. Those reasons mentioned above make walls, columns, doors and windows the most basic elements used by existing methods to reconstruct 3D building models. Thus, their characteristics will be analyzed with more detail in the following sections.

Moreover, material or texture of some objects is often required to be denoted in architectural floor plans. There are three types of material symbols in CAD software regularly used for this purpose. They are hatch pattern, solid fill and gradient fill. In Fig. 2-1, different hatch

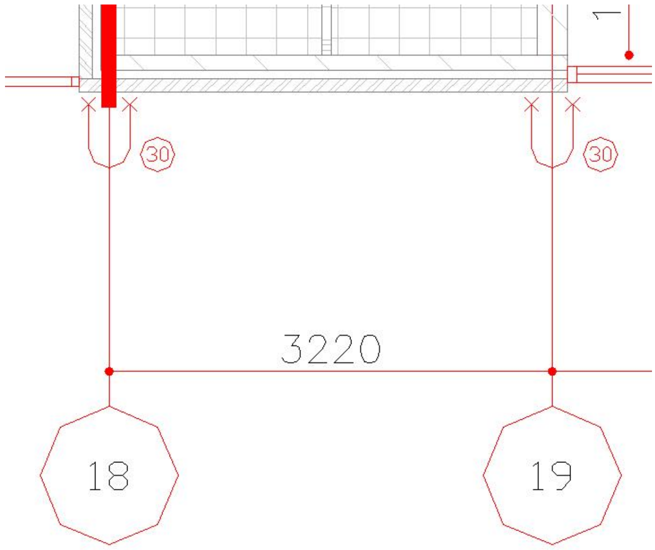


Figure 2-2: Center lines and dimension lines

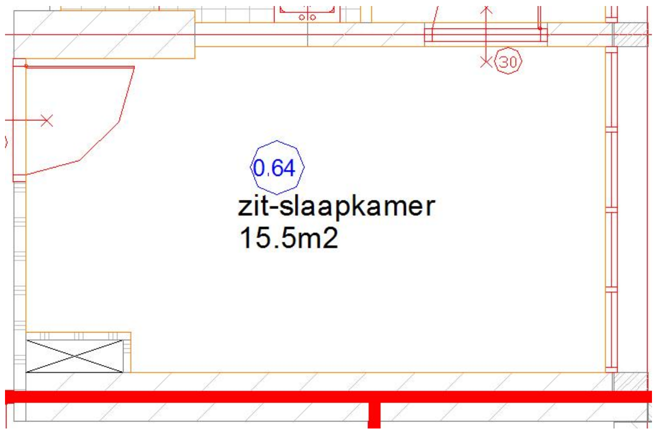


Figure 2-3: Descriptive information of a living room

patterns are used for walls as well as the tiled floor in bathrooms.

Stairs and elevators are another important architectural element in architectural floor plans. In Fig. 2-1, the stairs are bounded by the grey box. Fig. 2-4 shows some other examples of stairs symbols, from which it can be seen that although the representation of stairs in floor plans varies, they all share some common parts, e.g. the steps, the direction of flight indicating by an arrow, and the rails. From a topological view, stairs and elevators play a role of connector between different floors. In both 2D and 3D indoor navigation application, vertical connectors like stairs and elevators will first be searched when a cross-floor route is going to be determined. However, in this thesis, stairs and elevators will not be used in the reconstruction of the 3D building model since navigation is not the priority in this thesis. After a 3D model is rebuilt, future work can be invested into the reconstruction of stairs to obtain the topological information to be used for further applications, such as navigation.

Depending on the level of detail, there might be also furniture and facilities for emergency

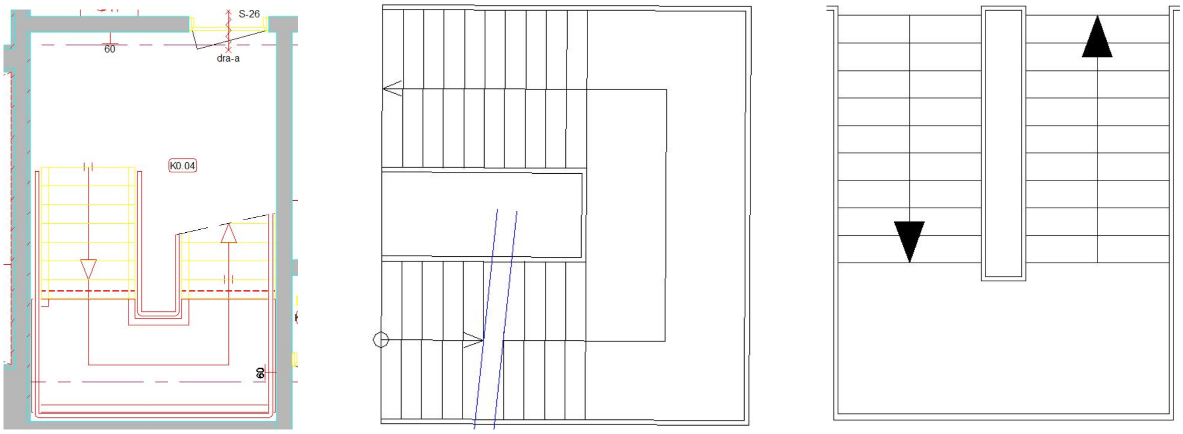


Figure 2-4: Symbols of stairs

control in a floor plan. Fig. 2-5 is the zoom-in view of one of the areas in Fig. 2-1 that are bounded by the green boxes, which is actually a bathroom. Furniture such as toilet, shower, wash-basin, ceramic tiles can be identified from it. Fig. 2-6 shows some other examples. These contents will not be used in this thesis as well, but they can be used to determine a function of a space or for emergency control applications in future study.

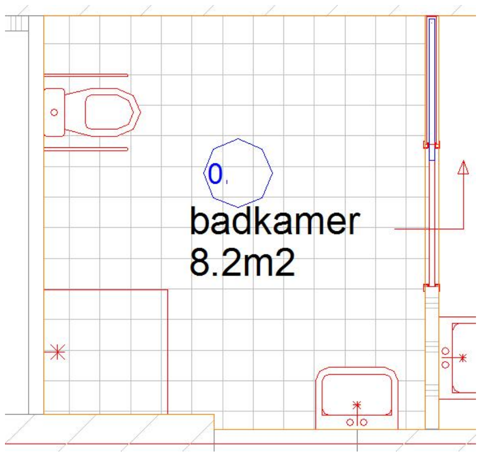
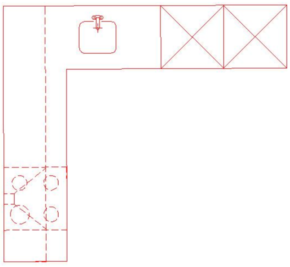
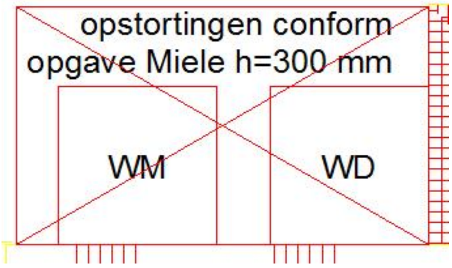


Figure 2-5: Toilet, shower, wash-basin, ceramic tiles

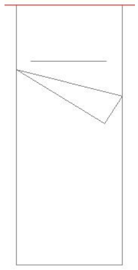
In addition to those auxiliary lines and symbols and interior objects introduced above, there are normally some other objects outside the building in a floor plan. They include balconies, canopies, railings, air-conditioning brackets, outdoor stairs and ramps and so on. The two sky blue boxes in Fig. 2-1 indicate stairs and ramps outside the building. Since this thesis focus on the reconstruction of the indoor environment, these outdoor objects will not be considered as well. Table 1 summarizes the contents that commonly show up in architectural floor plans.



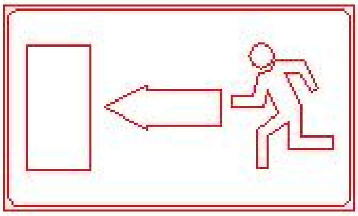
(a) Kitchen range



(b) Washing machine and washing dryer



(c) Bed



(d) Fire escape

(e) Fire hose

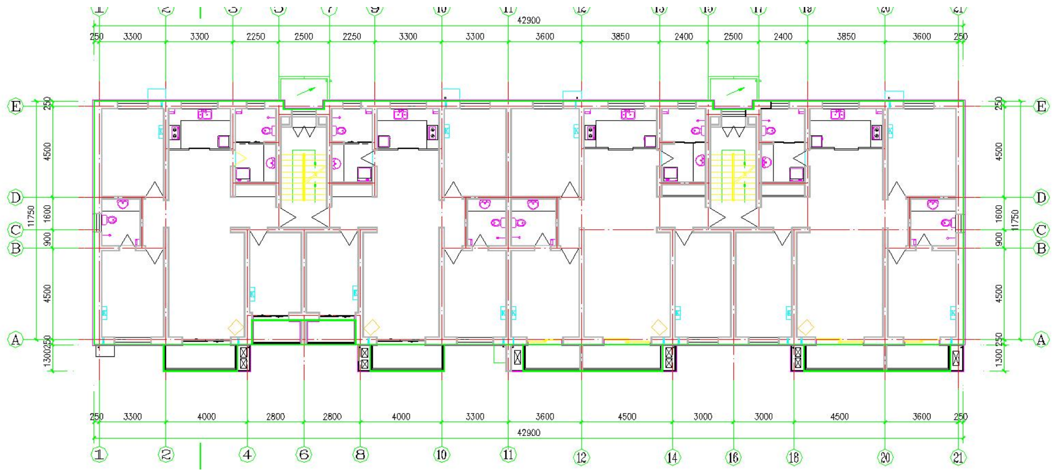
Figure 2-6: Symbols of furniture and emergency control facilities

Table 2-1: Contents of architectural floor plans

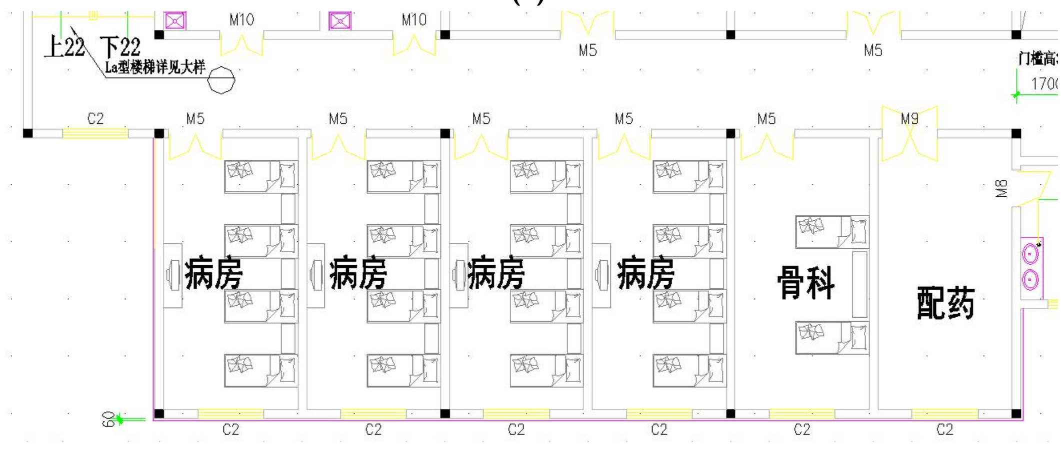
<table>
<tr><th>Category</th><th colspan="2">Elements and objects</th><th>To be used in 3D extrusion</th></tr>
<tr><td rowspan="4">Auxiliary</td><td colspan="2">Center lines</td><td>No</td></tr>
<tr><td colspan="2">Dimension lines</td><td>No</td></tr>
<tr><td colspan="2">Texts and numbers</td><td>No</td></tr>
<tr><td colspan="2">Other symbols</td><td>No</td></tr>
<tr><td rowspan="6">Architectural</td><td colspan="2">Wall and columns</td><td>Yes</td></tr>
<tr><td colspan="2">Doors and windows</td><td>Yes</td></tr>
<tr><td colspan="2">Material symbols</td><td>No</td></tr>
<tr><td colspan="2">Stairs and elevators</td><td>No</td></tr>
<tr><td>Furniture and facilities</td><td>Toilet, shower, wash-basin, ceramic tiles, kitchen range, washing machine, washing dryer, bed, wardrobe, closet, desk, carpet, fire escape, fire hose, etc.</td><td>No</td></tr>
<tr><td>Outdoor objects</td><td>Balcony, canopy, railings, air-conditioning brackets, outdoor stairs and ramps, etc.</td><td>No</td></tr>
</table>

2-1-2 Walls

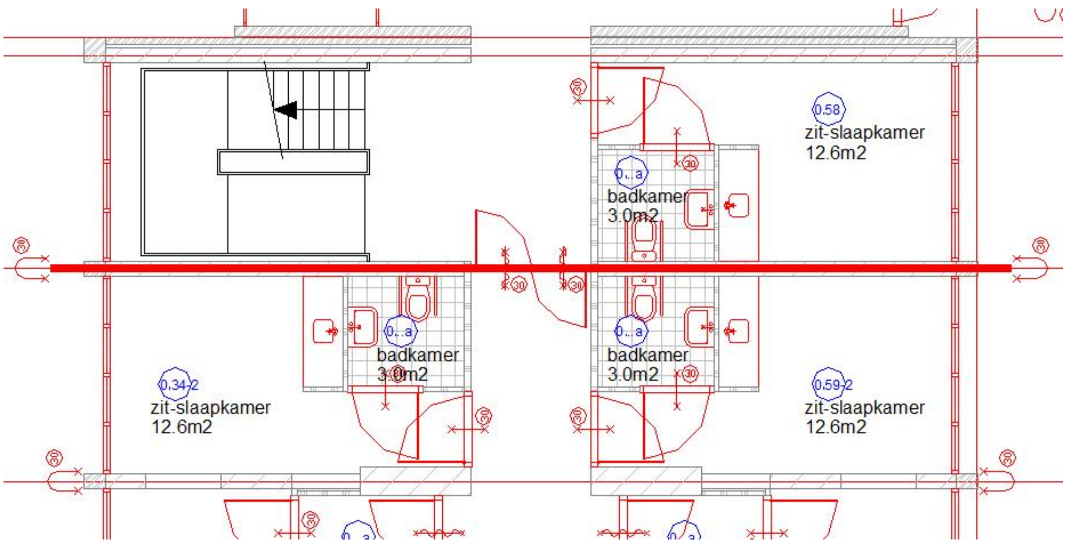
There are several ways of drawing walls and columns. Fig. 2-7 shows three typical ways that are found from the study of a set of architectural floor plans. In Fig. 2-7a, each wall with columns amid it is represented by a single polygon; in Fig. 2-7b, walls and columns are drawn separately; in Fig. 2-7c, even a wall is also separated into several parts based on the material of part of the wall. Besides the material of the wall, in some floor plans exterior walls and interior walls are also drawn separately. To make it clearer, Fig. 2-8 provides a demonstration of a simple room drawn in four of these different ways. In Fig. 2-8a all connected walls and columns are drawn by a single polygon, just like Fig. 2-7a; Fig. 2-8b and Fig. 2-8c shows two different ways of separating walls and columns, corresponding to Fig. 2-7b; Fig. 2-8d corresponds to Fig. 2-7c, in which exterior walls and interior walls are also drawn separately.



(a)



(b)



(c)

Figure 2-7: Examples of different ways of drawing walls

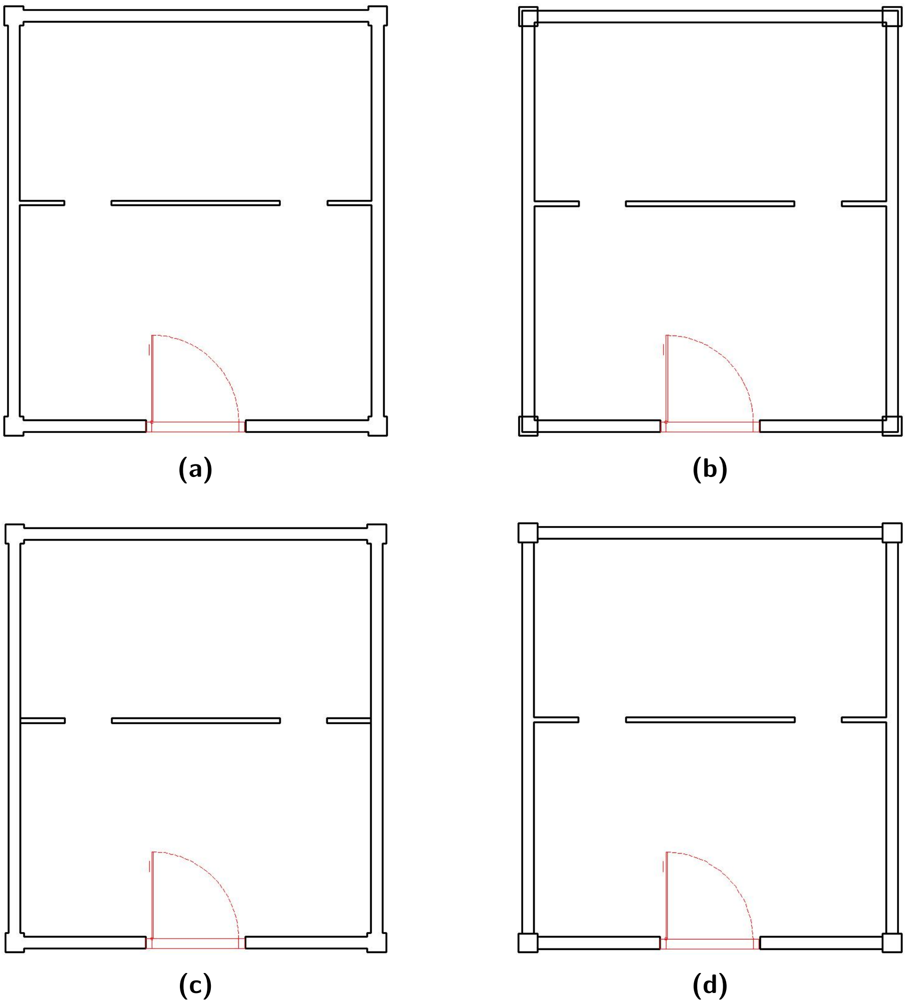


Figure 2-8: Demonstration of different ways of drawing walls

Additionally, in some of the floor plans that have been studied in this thesis, some decorative details and embossed bricks have been discovered on outer or inner side of the walls. Fig. 2-9 gives an example. Fig. 2-9b and Fig. 2-9c are respectively the blue area and green area in Fig. 2-9a.

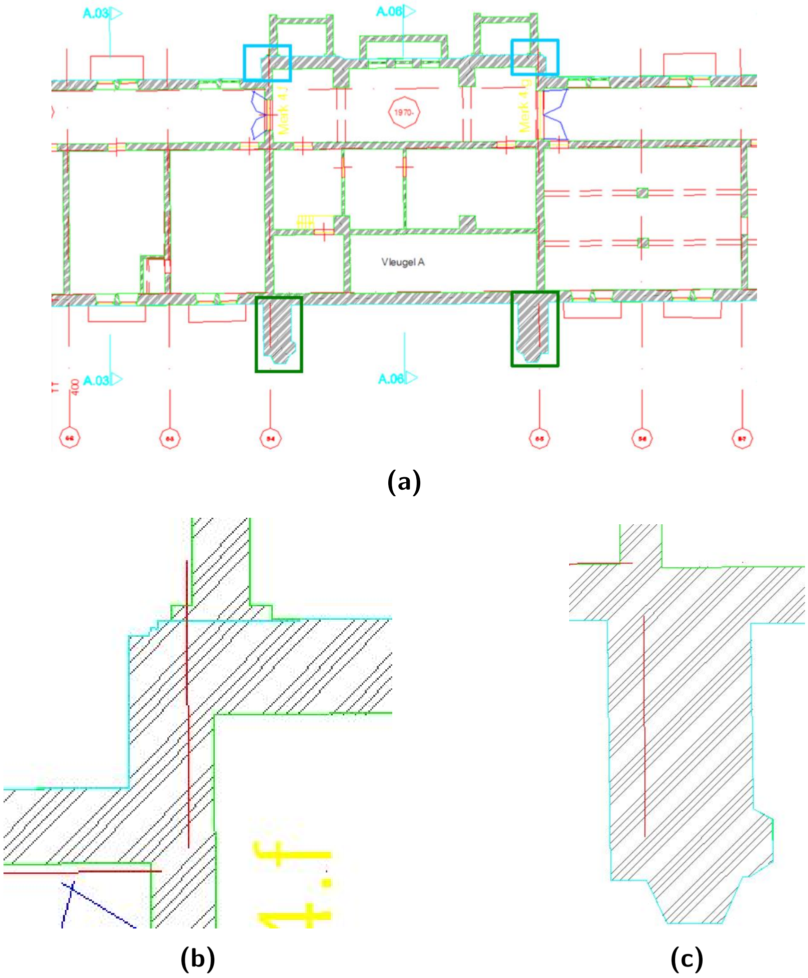


Figure 2-9: Decorative details on walls

Last, it is very common that in some buildings there are some hollow vertical shafts for air canal, pipelines and electric wires (Fig. 2-10).

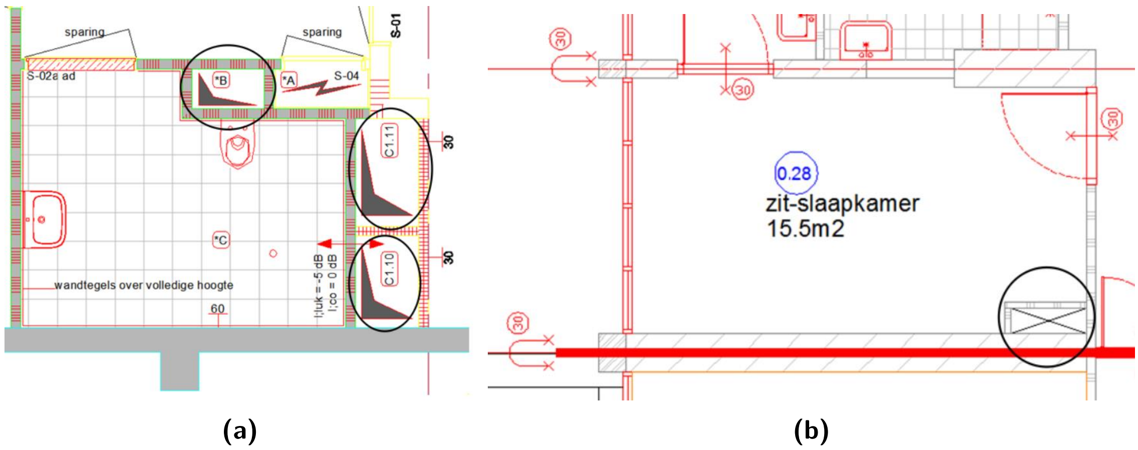


Figure 2-10: Hollow vertical shafts

2-1-3 Doors

Generally, there are four types of doors frequently used in 2D architectural floor plans. They are swing door, sliding door, pocket door and bi-fold door, among which swing door is the most common one.

Swing doors

Fig. 2-11 shows a simplified model of a swing door in both vertical and plane view. There are two main parts in the model: the door frame and the door panel. The door frame also consists of several components. A lintel is the horizontal block that spans the opening between its two doorjambs. Doorjambs are the side pieces of the door frame, which play a role of weight-bearing and connecting unit between the door and its adjacent walls. Some swing doors might also have a doorsill in the underpart of the door frame, which plays a role of threshold of the doorway and connecting unit between the door and the floor. As for door panel, it is the part of the door that actually separates two connecting spaces, on which a door knob (also called door handler) is attached and used to open or closed the door by certain mechanism.

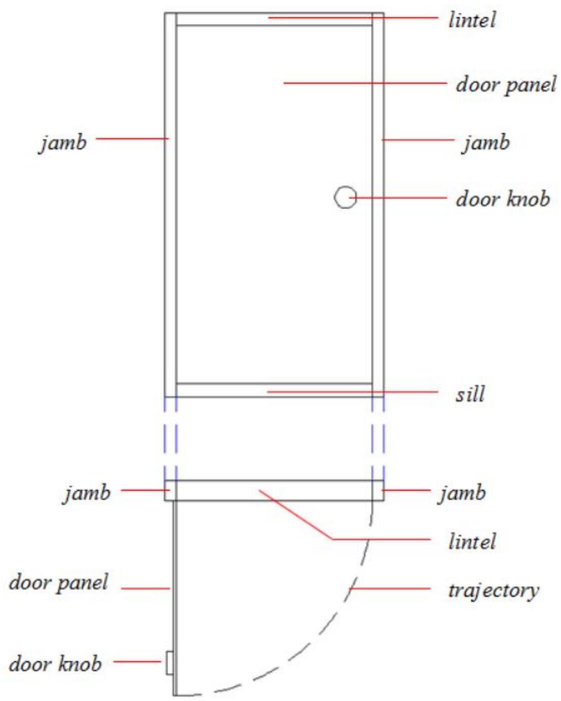


Figure 2-11: A normal swing door in both vertical and plane view

However, as simple as it is, the symbol of a single swing-door in the plane view in real-life floor plans can still vary a lot, since there are various variants in each part of the door. For the doorjamb, it might be simply represented by a rectangle, or some more detailed shapes, or even its detailed inner structure (Figs. 2-12a and 2-12b); for the door panel, it can either be represented by a rectangle indicating the thickness of the panel, or it can be simply represented by a single line (Figs. 2-12c to 2-12e); for the doorknob, it can be represented by multiple shapes, e.g. rectangle, ellipse or a single line parallel to the door panel (Fig. 2-12c); for the swing trajectory of the door panel, it can be represented by an arc (Figs. 2-12c to 2-12e and 2-12i), a single line (Figs. 2-12a and 2-12f) or broken lines (Figs. 2-12b, 2-12e and 2-12h). And the angle can either be 90 degrees or certain angle less than 90 degrees (Figs. 2-12a and 2-12f). In addition, none of these components is certain to be drawn in the symbol. In some extreme cases, a door symbol can even just be a single rectangle filling the gap between its adjacent walls (Fig. 2-12j), or a single line indicating the door panel with an arc or a line indicating the trajectory (Figs. 2-12a and 2-12f).

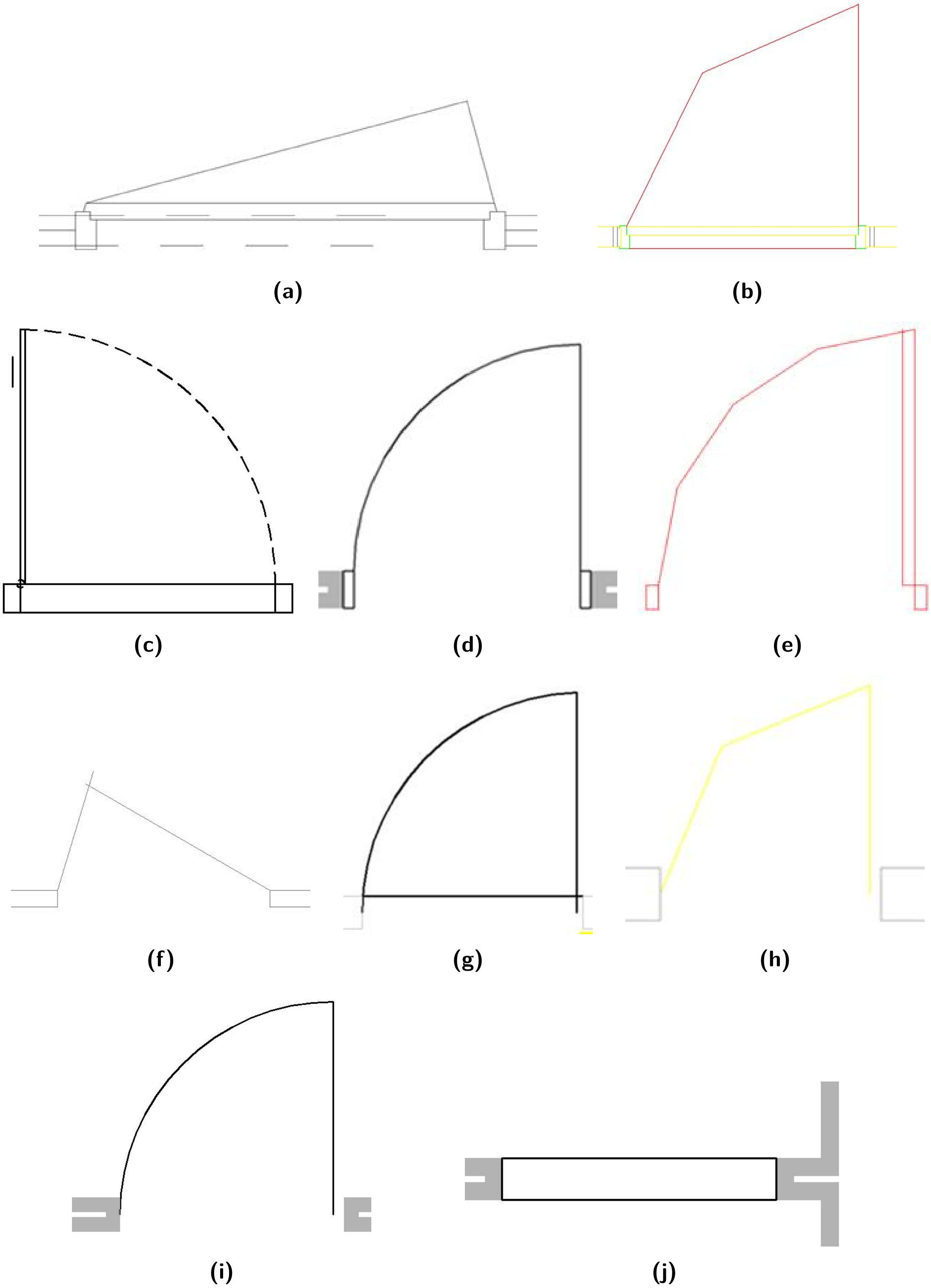
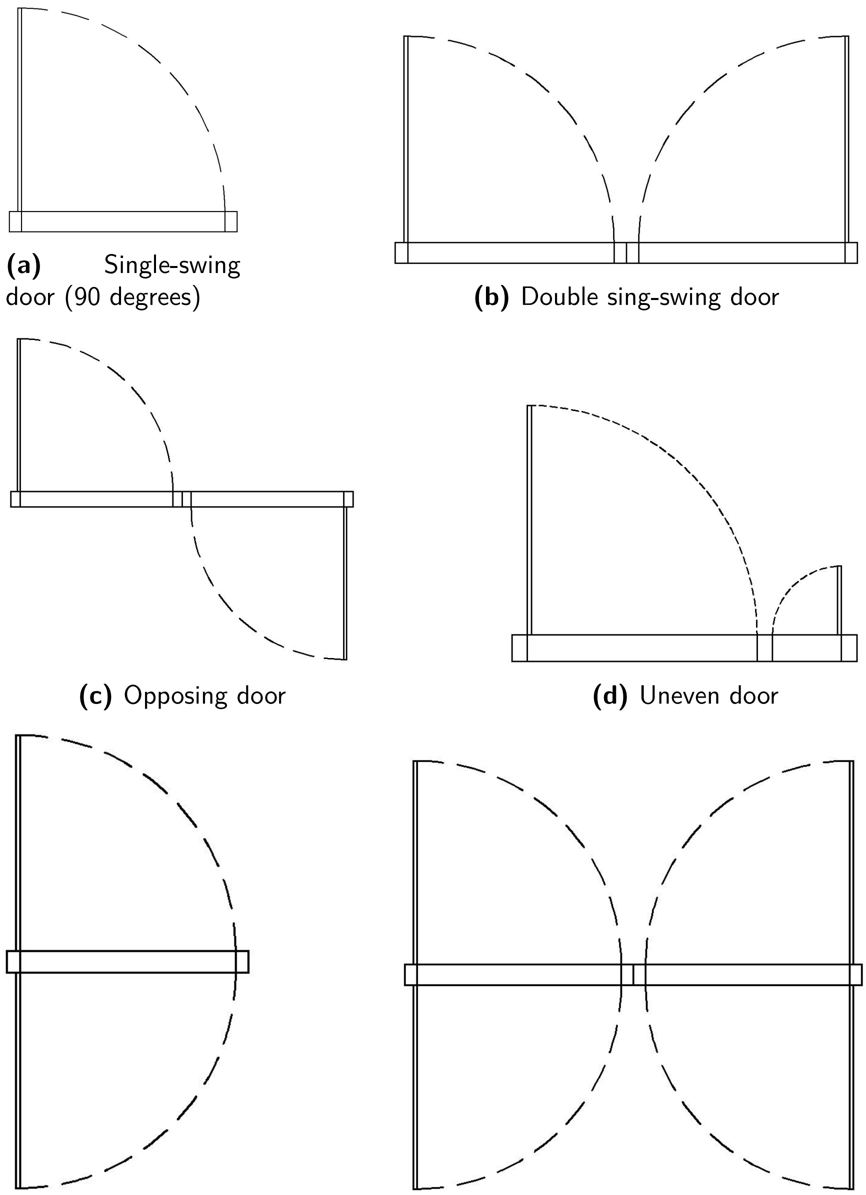


Figure 2-12: Variants of door symbols from real-life floor plans

What is described above can be summarized as the problem of level of details in the representation of the symbols. Besides that, several single-swing doors can be combined together forming other more complicated swing doors. Fig. 2-13 shows different combinations of single-swing doors in real-life floor plans.



(a) Single-swing door (90 degrees)

(b) Double sing-swing door

(c) Opposing door

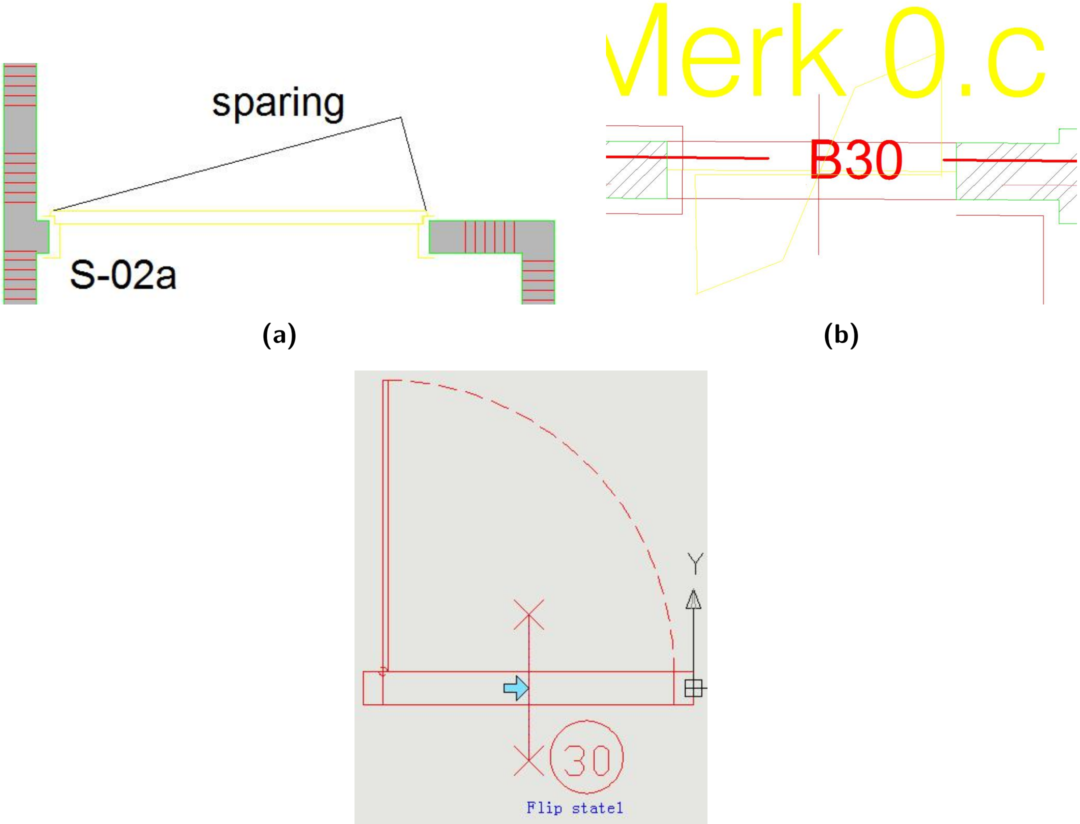
(d) Uneven door

(e) Double-acting door

(f) Double double-acting door

Figure 2-13: Different combinations of single-swing doors

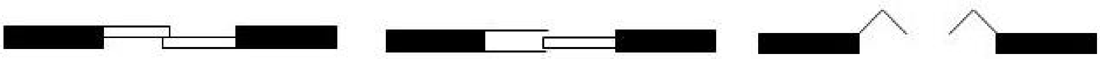
Last, in architectural floor plans, in addition to the door symbol itself, there are always some other extra primitives or annotations in the symbol expressing more detailed information about the opening. For example, in Figs. 2-14a and 2-14b, there are some texts around the doors indicating the model of the doors; in Figs. 2-14b and 2-14c, the central lines of the doors are also drawn; in Fig. 2-14c, there are two crosses on both sides of the door panel connected by its central line and a number “30” in a circle, which means this door can hold fire and smoke for thirty minutes.



(a) (b) (c)

Figure 2-14: Annotations in door symbols

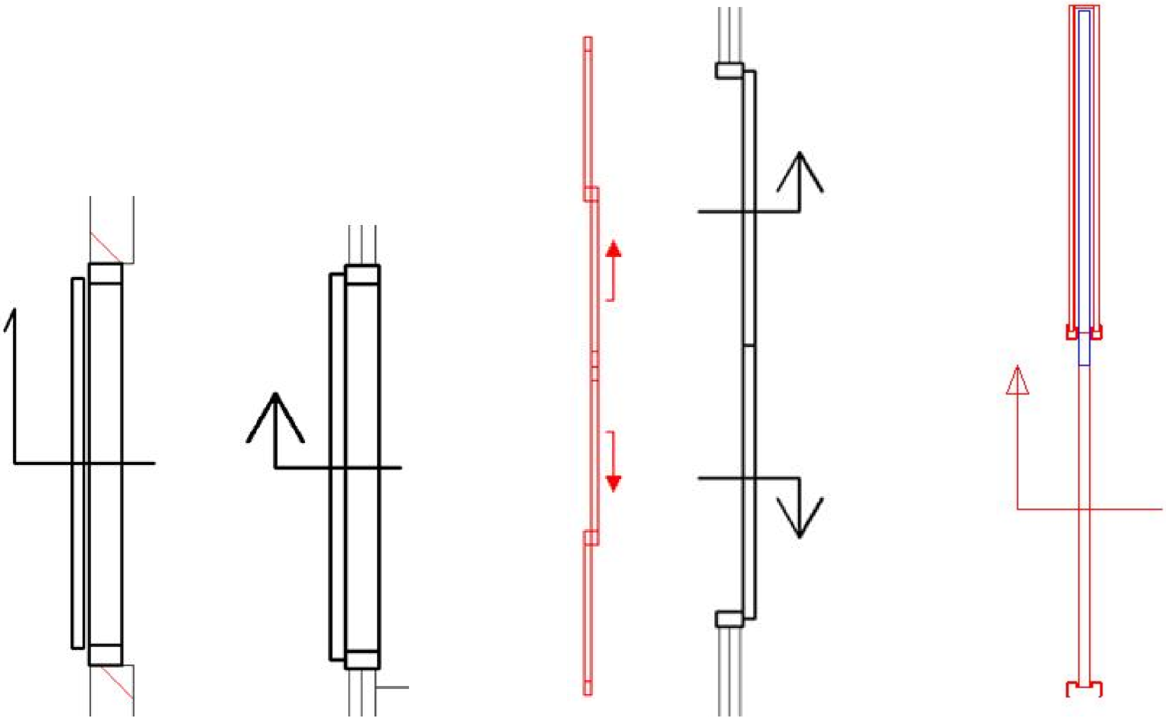
Sliding doors, pocket doors and bi-fold doors



(a) Sliding door **(b)** Pocket door **(c)** Bi-fold door

Figure 2-15: Basic models of sliding door, pocket door and bi-fold door

In addition to swing doors, occasionally there might be some sliding doors, pocket doors and bi-fold doors in real-life floor plans. Fig. 2-15 shows the basic models of sliding door, pocket door and bi-fold door. Compared to swing doors, a larger part of these doors exists in between the gap of its adjacent walls since they do not have the opened door panel with a swing trajectory. However, there usually is an arrow within the symbol indicating the direction in which the door panel moves, similar to the trajectory in swing door. Fig. 2-16 shows some variants of symbols of sliding doors and pocket doors in real-life floor plans.



(a) Single sliding door **(b)** Double sliding door **(c)** Pocket door

Figure 2-16: Sliding doors and pocket doors in real-life floor plans

2-1-4 Windows

There are mainly three types of windows normally used in architectural floor plans: fixed window, sliding window and casement window. A fixed window is a window that cannot be opened, only allowing light to go through, while sliding window and casement window belong to the category of unfixed window, whose window sashes can somehow be moved to open or close the window. Depending on the way the window sash moves, sliding window and casement window can be distinguished. A casement window is a window with a hinged sash that swings in or out like a door. Based on the location of the hinge, it can further be divided into side-hung, top-hung (also called "awning window"), and bottom-hung sash (also called "hopper window"). Fig. 2-17 shows some examples of window symbols in real-life floor plans.

Like doors, there can also be combinations of windows of different types. Fig. 2-17g shows a combined consisting of a hopper window and a single casement window. Besides, compared with doors, a special point of windows in architectural floor plans is that several windows can connect with each other to form a curtain wall, a wall of glass. Except for its transparency, this kind of wall plays a same role with normal walls of separating two adjacent spaces. Fig. 2-18 shows two examples of curtain walls.

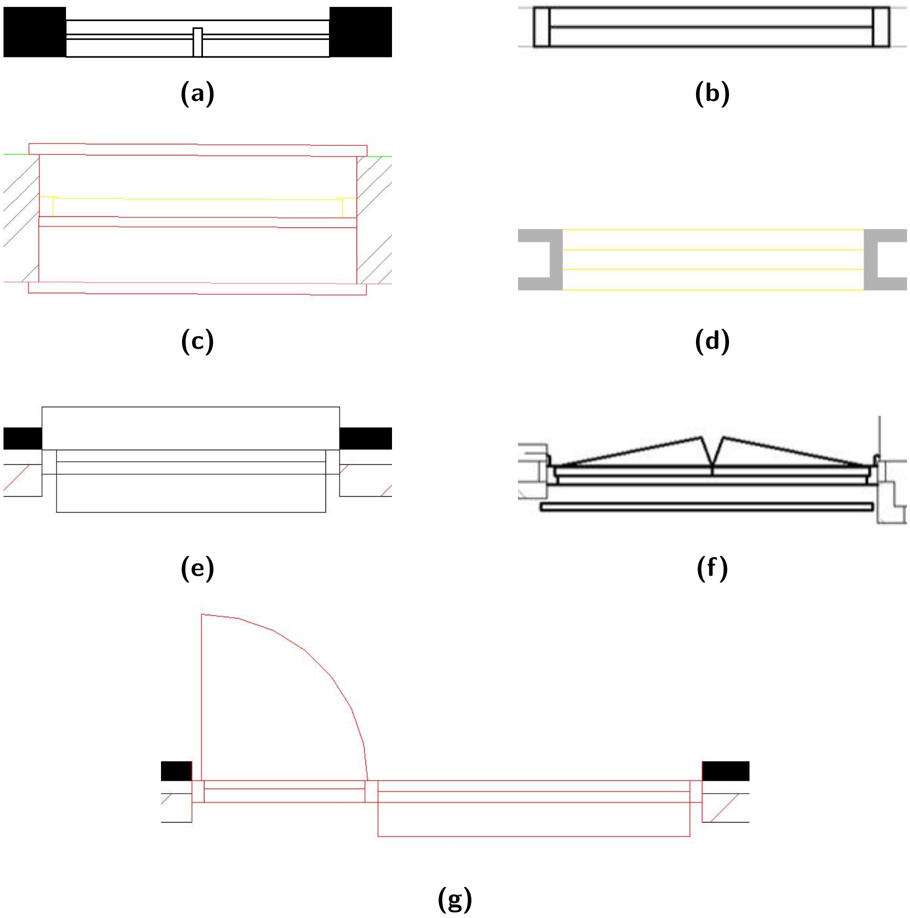


Figure 2-17: Variants of window symbols in real-life floor plans. (a) (b) Fixed window; (c) Single-hung sash; (d) Double-hung sash; (e) Hopper window; (f) Double casement window; (g) Combination of a hopper window and a single casement window

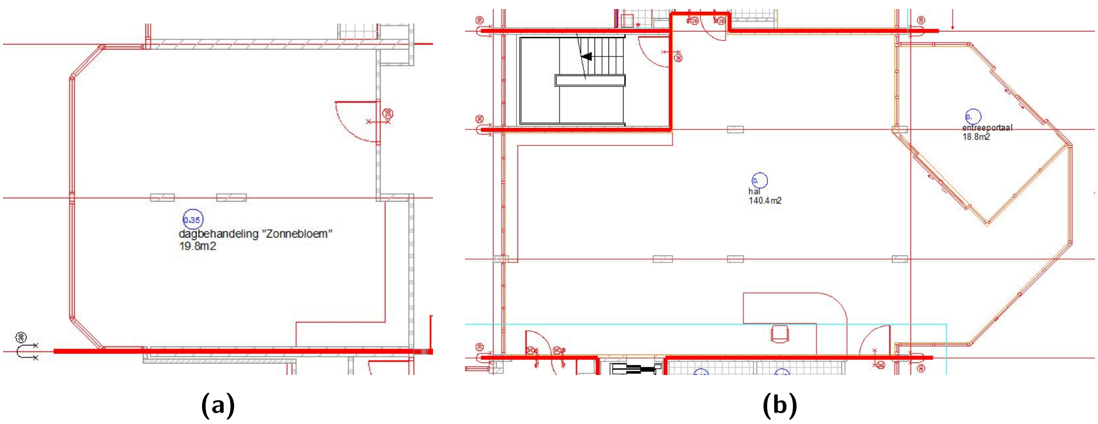


Figure 2-18: Examples of curtain walls

2-2 2D floor plan processing

There have been quite many studies done to reconstruct 3D models from 2D architectural floor plans. Based on the format of the input floor plans, they can be mainly divided into two groups: ones that use scanner images as input, and ones that use CAD-based files as input. CAD applications are widely used nowadays in the field of architecture by designers to draw graphic primitives on computers in the format of vector. They allow users to more efficiently manage graphical information by segmenting the whole drawing into different layers of related elements or grouping primitives into blocks to represent some higher level objects. In addition to CAD-based floor plans, there are still many floor plans that were drawn on paper by hand before the popularization of CAD software. These paper floor plans are usually digitally scanned and saved as raster images. Compared with CAD-based floor plans, the distinction between wall lines, graphical symbols, textual content and some other components in a raster image of architectural floor plan is much vaguer, since they are all represented by line segments of pixels in one integrated layer [20]. Fig. 2-19 shows the pipeline followed by them.

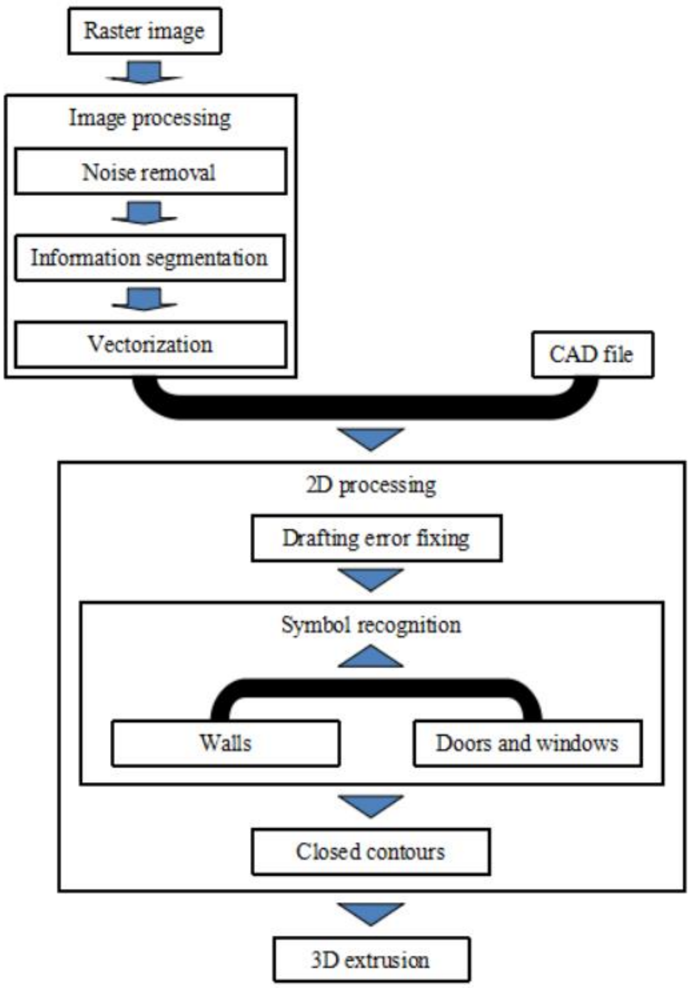


Figure 2-19: Pipeline for raster-based and CAD-based systems

Generally, researches using CAD-based floor plans as input data take advantage of CAD applicationsąŕ in-built methods, such as layers and blocks, to realize the information segmentation. Different symbol recognition methods will then be applied to detect walls, doors, windows and possibly semantics from the floor plans. Finally, closed contours of different spaces can be obtained using different loop searching algorithm and then extruded to recreate the 3D building models. In comparison, researches using raster image floor plans as input data must first segment the information contained in the floor plans since pixels representing different information all are mixed together in one single layer, before each group of information can be properly dealt with respectively. However, apart from the preprocessing of raster images, they share drafting error fixing, wall detection, opening recognition and contour reconstruction in the 2D processing phase. Thus, in this section, methods used in these four steps will be reviewed respectively.

2-2-1 Drafting error fixing

Manually generated input floor plans typically suffer from many drafting errors and redundancies [20]. These errors might be visually imperceptible in CAD applications for users and not really affect the use of the floor plans for a construction purpose, but they might make algorithms to be used in later phases generate some unpredictable results and thus affect the behavior of the overall (semi)automatic process. Therefore, they have to be found and corrected.

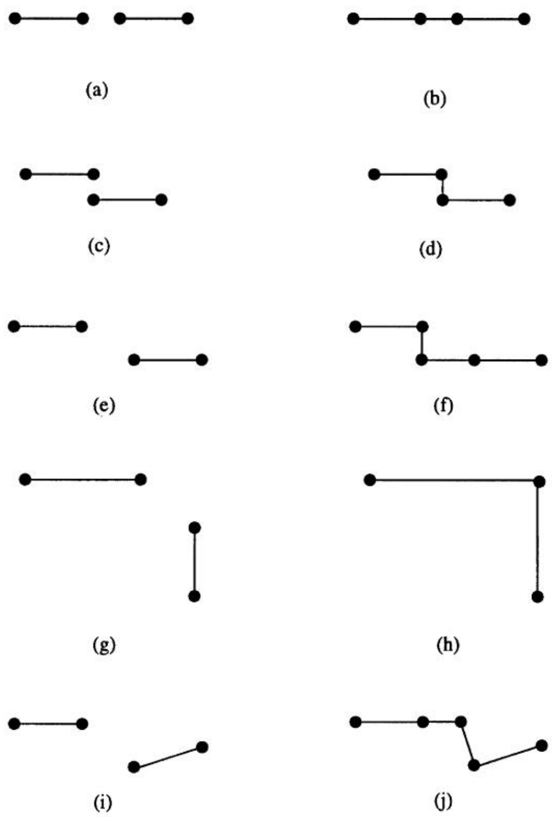


Figure 2-20: Correction of disjoint vertices [30]

However, very few researches reviewed by this thesis have considered the drafting errors. Most of their algorithms are based on the assumption that the input floor plans do not contain any drafting errors. Only Rick et al. proposed a coerce-to-grid method to clean up disjoint vertices in his prototype system called Building Model Generator (BMG) [30]. Disjoint vertices appear when two lines that are supposed to connect with each other at a same vertex disjoin or intersect. Figs. 2-20a, 2-20c, 2-20e, 2-20g and 2-20i show cases of disjoint lines. Fig. 2-21 shows a case of overlapping lines. In this method, first, every vertex is snapped to a grid of a specified resolution to fix relatively small gaps and intersections. Then lines that are still disjoint after the snapping are corrected in the way shown in Fig. 2-20. If these two lines are parallel and collinear (Fig. 2-20a), they will be simply connected (Fig. 2-20b); if they are parallel but not collinear (Figs. 2-20c and 2-20e), they will be connected perpendicularly (Figs. 2-20d and 2-20f); if they are approximately perpendicular (Fig. 2-20g), the intersection of these two lines will be computed and they will be extended to the computed intersecting point (Fig. 2-20h); if none of these conditions is fulfilled (Fig. 2-20i), a new line perpendicular to one of the two lines will be created and intersect with the extension of the other line (Fig. 2-20j). In addition, overlapping lines are also corrected by cutting each of them into two distinct lines at the intersecting point and discarding the shorter lines (Fig. 2-21).

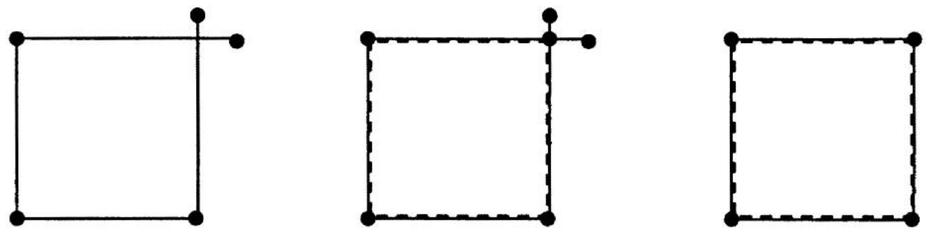


Figure 2-21: Correction of overlapping lines [30]

Nevertheless, this method only considered disjoint vertices that are caused by disjoint lines and false intersecting lines. In real-life floor plans, there also exist other drafting errors like null-length lines and duplicated lines. In this thesis, a new method is developed to fix those null-length lines and duplicated lines before disjoint vertices are fixed.

2-2-2 Opening recognition

In most cases, openings are detected by using different symbol recognition techniques. Based on the format of the input data, they can be divided into two main categories: vector-based and pixel-based. Vector-based approaches process vectorized images that contain primitives such as points, lines, arcs and circles, by checking the mutual relationship between a group of neighboring primitives. Pixel-based approaches work on raster images, trying to fit the statistical features of a symboląŕs pixels in the image. Due to this thesis focuses working on CAD-based floor plans, only main vectorized-based methods have been reviewed, which includes example-driven approach [28], graphical-knowledge-guided reasoning [31] and constraint network [32].

Guo et al. described an improved example-driven symbol recognition algorithm based on an extended relation representation mechanism with automatic knowledge acquisition capability. They also proposed a method making sure similar symbols with repeating modes can be recognized by one rule. However, they only considered limited geometrical relations that are commonly used in architectural symbols [28]. Yan et al. proposed a graphical-knowledge-guided reasoning method, which learns graphical knowledge from five types of geometric constrains (intersection, parallelism, perpendicularity, circles and arcs) from an example symbol given by user and uses the learned knowledge to recognize similar symbols. Yet, their prototype system can only learn the graphical knowledge from single example [31]. Ah-Soon and Tombre introduced a method that defines a set of constraints on geometrical features describing what architectural symbols to be recognized. Features extracted by this method from the drawing through the network of constraints can be propagated so that the network can be constantly updated whenever new symbols need to be taken in account [32]. Nevertheless, from the report on the International Symbol Recognition and Spotting Contest on 2013 [33], it can be concluded that symbol recognition still remains an open question. According to them, although several existing methods have achieved satisfying results, they all have limitation to different degrees under certain circumstances that are not originally designed for the methods.

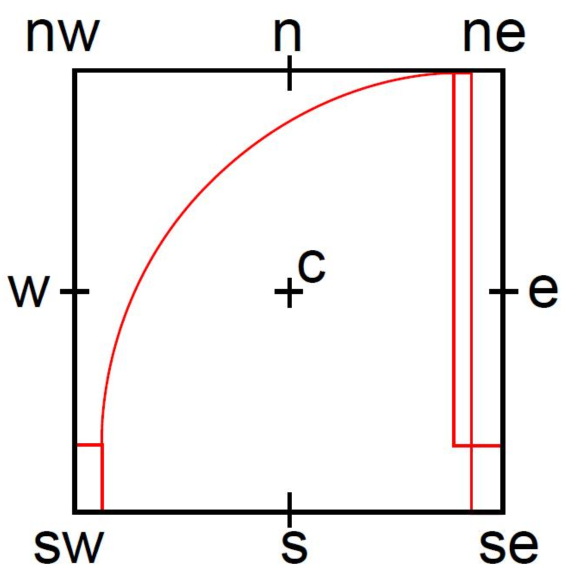


Figure 2-22: Nine-point bounding box of a door block [34]

Due to the limitation of symbol recognition, in [34] a new idea is introduced to handle the openings, which uses the bounding box of opening blocks. First, it is assumed that contents of walls and openings are properly stored in two separated layers in the CAD file and that each opening symbol is saved as an instance of block. Then the bounding box of each block is calculated among all the primitives in the block. The central point of the bounding box (point C in Fig. 2-22) is used to search for its nearest endpoints of a topology segment that represents a wall. A topology segment of the opening is created by connecting the two found endpoints. This method benefits from the use of layering and blocking supposed in CAD system to avoid the influence from the varying layout of the primitives in the symbols. It is limited to normal layout between opening and its adjacent walls, such as the one shown in Fig. 2-25. Fig. 2-23 shows a scenario where the algorithm could go wrong.

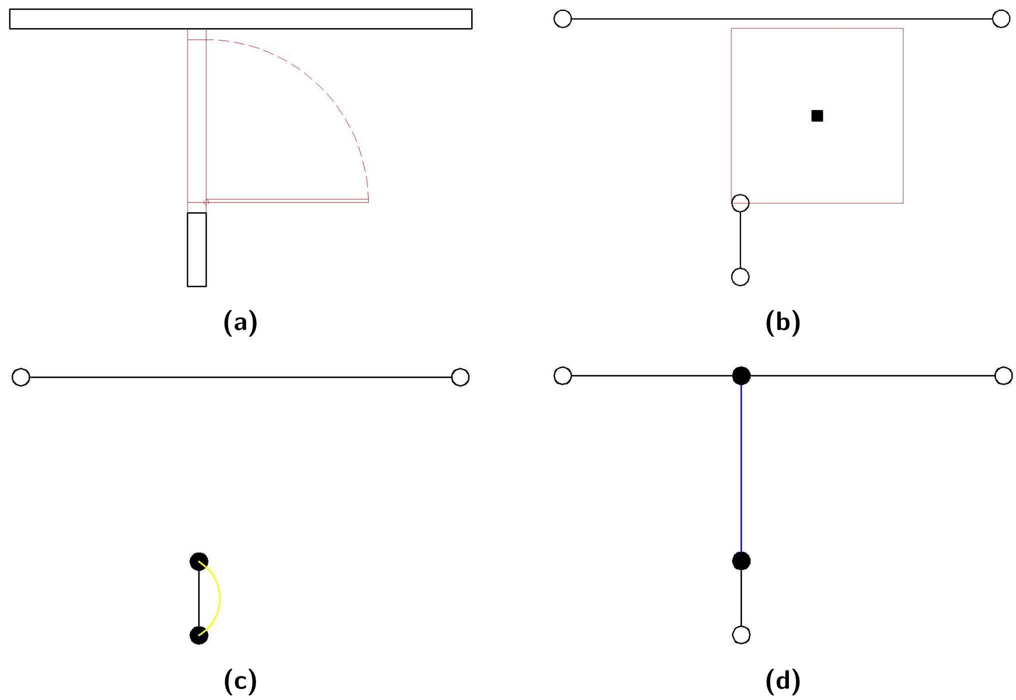


Figure 2-23: Opening topology segment: (a) a scenario with a door and its adjacent walls; (b) topology graph of the walls and the bounding box of the door; (c) false result of nearest endpoints searching; (d) expected correct result of nearest endpoints searching.

2-2-3 Wall detection and contour reconstruction

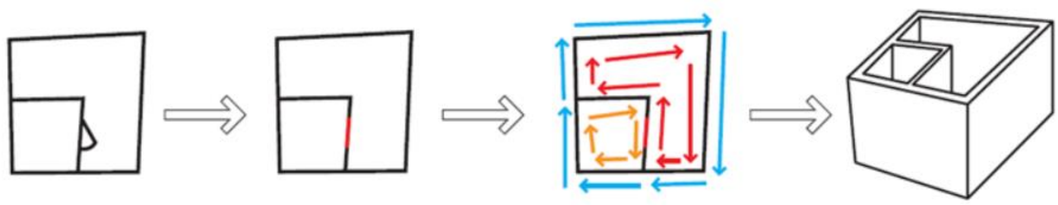


Figure 2-24: Contour searching for wall extrusion [20]

After openings have been recognized, contours of indoor spaces will be reconstructed for 3D extrusion (Fig. 2-24). Usually, equivalent lines are created for the recognized openings to replace the opening symbols in the floor plans (Fig. 2-25). Then, certain contour searching algorithm is applied to find closed contours among the opening equivalent lines and the wall lines. In some researches, to establish the topological relation between walls and openings, certain wall detection algorithm is additionally required to detect wall objects from the wall lines.

In the prototype system called the Building Model Generator (BMG) developed by Rick et al. to address the issue of creating 3D building models from existing floor plans, they replaced each such door symbol as shown in Fig. 2-25a with two parallel lines (Fig. 2-25b) as a step towards building closed room contours that can be located through a vertex-graph traversal [30]. Then with a starting vertex of a starting line and a desired orientation determined (Fig.

2-26a), the traversal chooses the leftmost turn and proceeds to next line (Fig. 2-26b). This process repeats and backtracks when dead ends are reached (Fig. 2-26c) until the starting vertex is reached (Fig. 2-26d). At this moment, a closed contour has been formed and lines in this contour are marked so that they will not be used in the next traversal.

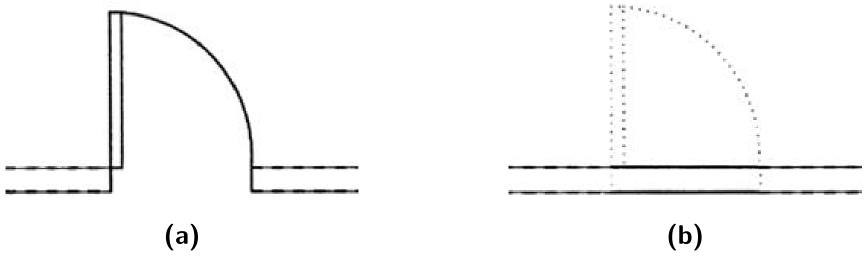


Figure 2-25: Replace door symbol with a pair of parallel lines[30]

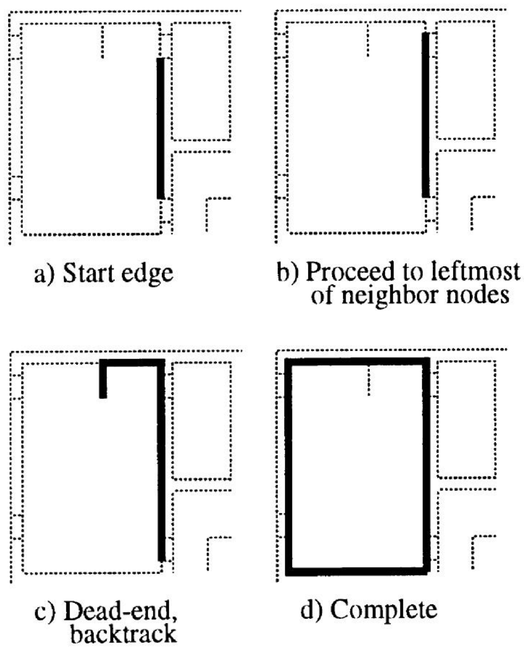


Figure 2-26: Vertex-graph traversal for interior contour [30]

In order to additionally detect floor topology from architectural floor plans, Domínguez et al. proposed a method, in which walls are extracted as single lines from a set of planar segments contained in the wall layer [34]. In this method, each pair of parallel segments that are close with each other is searched in the wall layer (Fig. 2-27 (1)). The endpoints of one segment will be projected to the other to find the common part between them (Fig. 2-27 (2)). The segments then are split at the projected point and the common part will be recognized as walls and removed from the wall layer (Fig. 2-27 (3)). This searching process repeats in the wall layer until no segments in the layer fulfill the criteria (Fig. 2-27 (4)).

Afterwards, the center line of each recognized line pairs is used to represent the walls and form a topology graph of the floor plan with opening equivalent lines recognized by certain symbol recognition techniques (Fig. 2-28). However, they also only considered the simplest

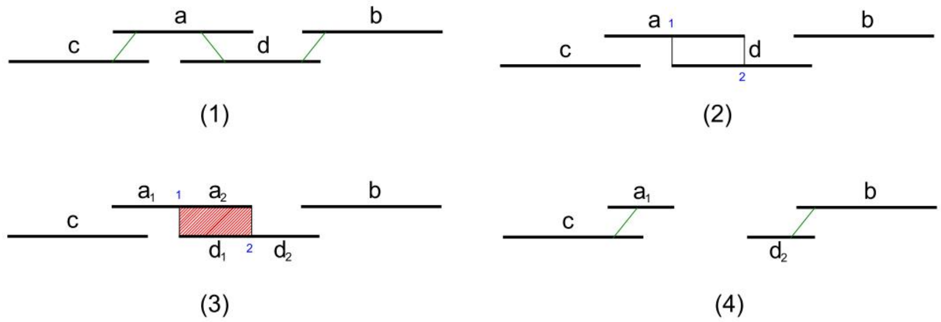


Figure 2-27: Iteration of the algorithm proposed by Domínguez et al. [34]. (1) Initial set of segments and relations. (2) Projection of the end points. (3) Segment splitting and wall extraction. (4) Updated segments and relations.

layout as shown in Fig. 2-25 between opening and its adjacent walls.

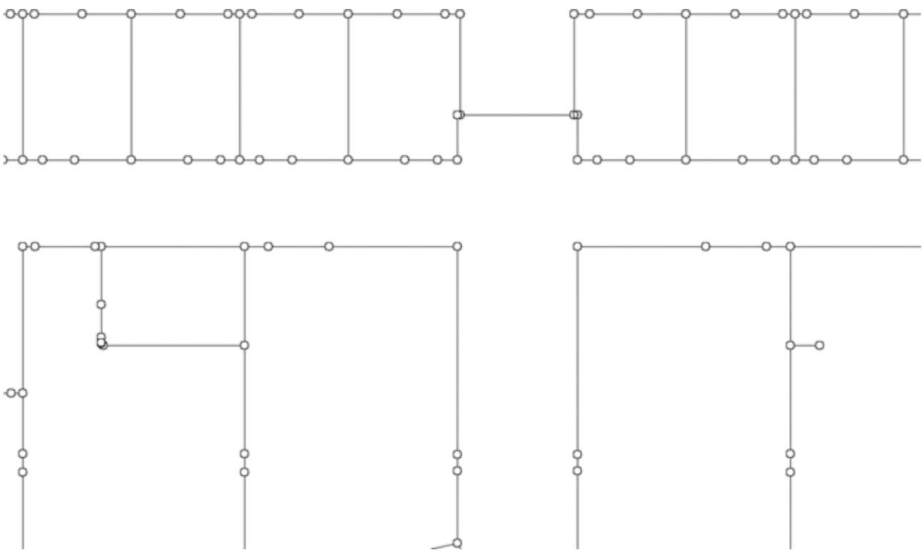


Figure 2-28: Topology representation from a portion of a CAD vector floor plan I [34]

In addition, by testing this wall detection algorithm with the floor plans used in this thesis, it is found that some line pairs that are not supposed to represent walls might also be detected in the results. Fig. 2-29 shows three examples of incorrect detection results. Figs. 2-29a, 2-29d and 2-29g are the expected results from the wall detection algorithm. In case of Fig. 2-29a, there should be three parallel line pairs being detected, while in case of Figs. 2-29d and 2-29g, there should be two and four pairs. But, in practice, more line pairs fulfilling the conditions are mistakenly detected (④ ⑤ in Fig. 2-29b, ③ in Fig. 2-29e, ② in Fig. 2-29h). This results in those extra red wall lines in Figs. 2-29c, 2-29f and 2-29i.

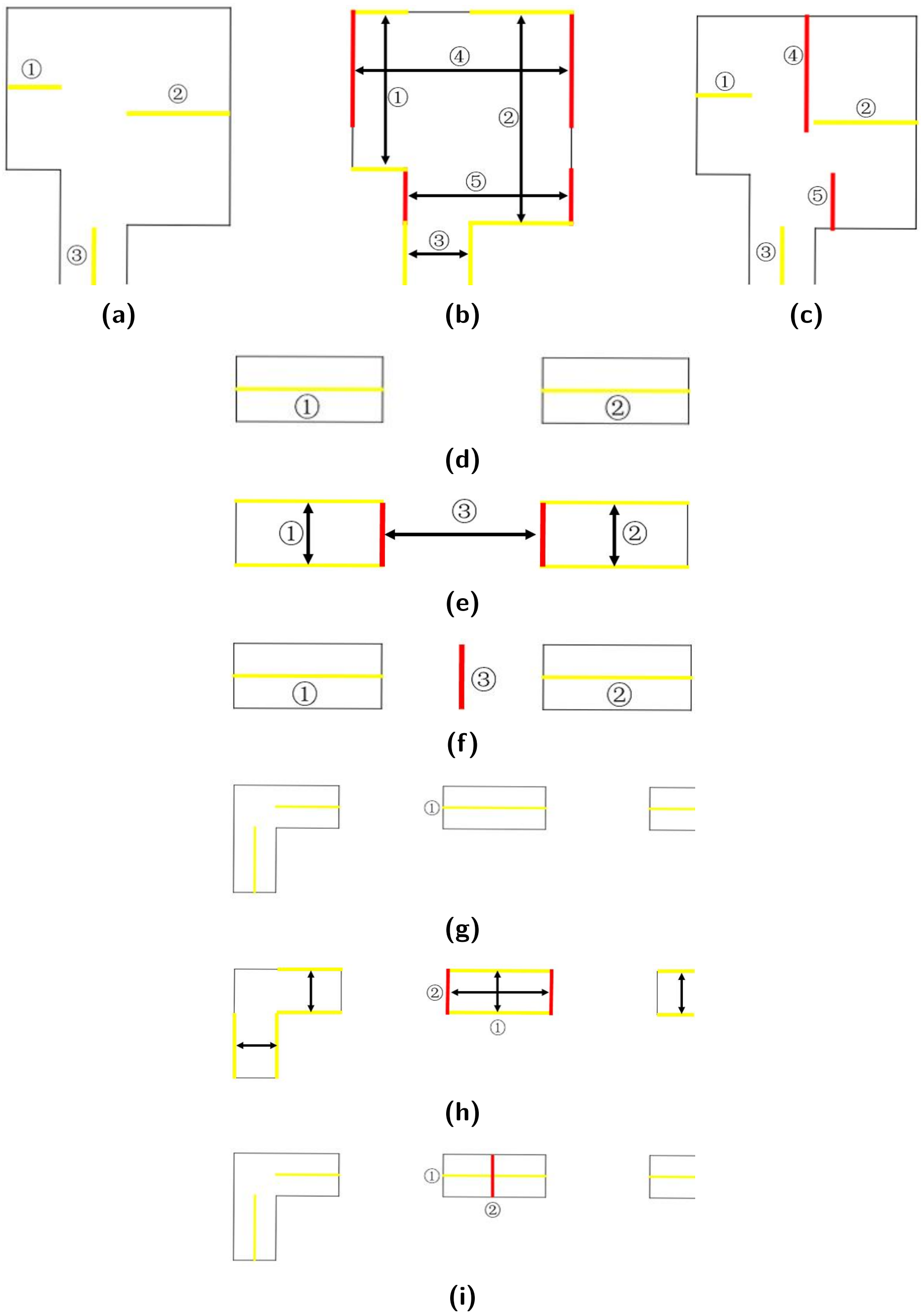


Figure 2-29: Examples of incorrect wall detection results: (a)(d)(g) expected results; (b)(e)(h) incorrectly detected parallel line pairs; (c)(f)(i) incorrectly detected wall lines.

These errors are mainly caused by the user-defined threshold. Because, to find parallel line pairs that are possibly representing walls, for each line, the algorithm only searches for its corresponding parallel line within a user-defined threshold. If this threshold is set to be too small, some thick walls might be overlooked. As a result, not all the detected wall lines can be connected to other wall lines or opening equivalent lines at its both sides. Thus, this threshold must be at least the biggest wall thickness to make sure all walls can be detected. However, problem presented above arises that some line pairs that are not meant to represent walls will be detected in turn.

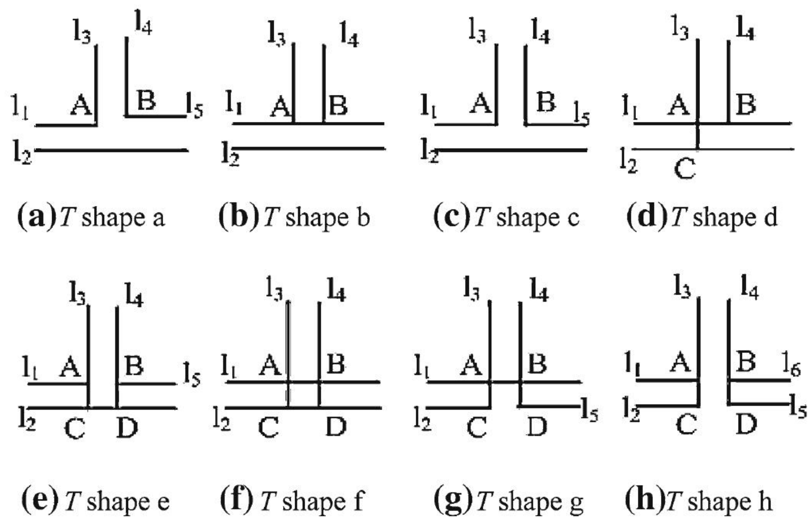


Figure 2-30: Shape T and its variations [35]

A new idea to extract walls from architectural floor plans is put forward by Lu et al. [35], who applied a shape-based recognition method for structural entities (e.g. walls and beams). In their research, they dealt with floor plans in which information is not segmented into proper layers. In this case, such simple criteria as close parallel line pairs represent structural entities cannot be trusted any more since there are a lot of disturbing lines making the analysis of parallel line pairs more complicated. Based on this, they argued that shapes of crossing regions can be used as the entrances for the recognition. They classified the most frequently occurring shapes into three types: shape T (Fig. 2-30), shape X (Fig. 2-31) and shape L (Fig. 2-32). Only after two end shapes are identified, the parallel line pairs between them can be recognized as walls.

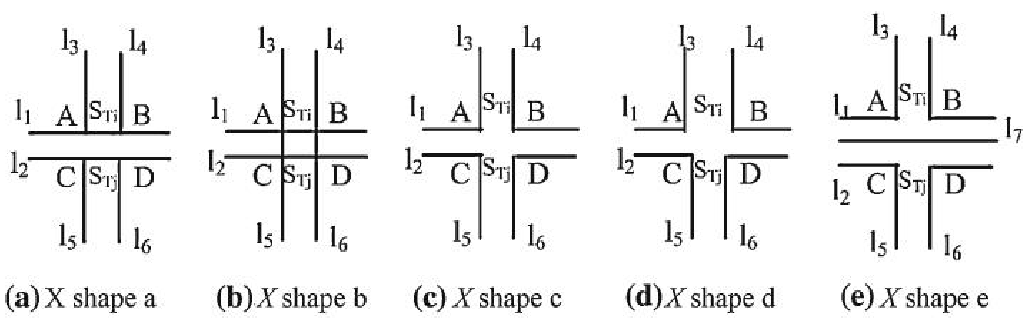
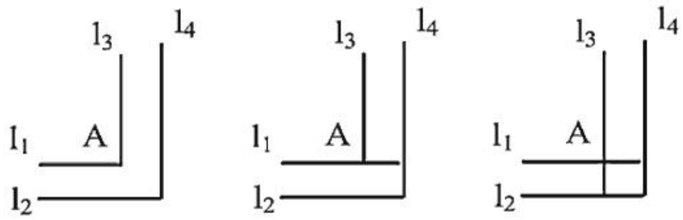


Figure 2-31: Shape X and its variations [35]

Unfortunately, although this method realized the recognition of structural entities from floor plans without being layered, it still suffered from the problem of user-defined threshold as Domínguez et al. did. The specific reason has been analyzed above. In Fig. 2-33, besides the correct recognized shape, there are also shapes that are falsely recognized or not recognized at all.



(a) L shape a **(b)** L shape b **(c)** L shape c

Figure 2-32: Shape L and its variations [35]

In a more recent research, on the basis of the work of Lu et al., Zhu et al. introduced a Shape-Opening Graph (SOG) to build the topological relationships between recognized parallel line pairs and openings, with the observation that X, L, T shapes intersect with other shapes or other openings and that an opening is adjacent to either shapes or other openings. Each time, an opening is used to search for it adjacent shapes or openings in the SOG. Then, according to the layout between the walls and openings, opening equivalent lines are created in corresponding ways (Fig. 2-35). Fig. 2-34b shows the parallel line pairs recognized from Fig. 2-34a. Fig. 2-34c shows the preprocessing result of the wall lines and the opening lines after all vertices with degree of 1 are fixed. Last, closed loops are searched by a similar vertex-graph traversal as shown in Fig. 2-26. There are two kinds of loops in the floor plan: inner loops, which represent indoor spaces, such as rooms and corridors, and outer loop, which represents the floor shell. In Fig. 2-34d, the inner loops are drawn in blue, while the outer loop is drawn in red.

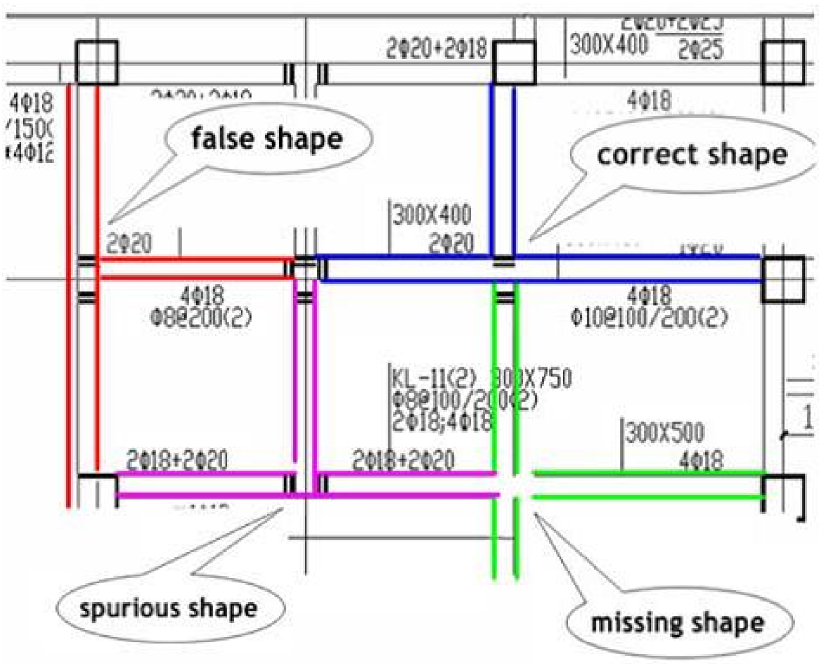


Figure 2-33: Correct, false, missing and suspicious shapes recognized by [35]

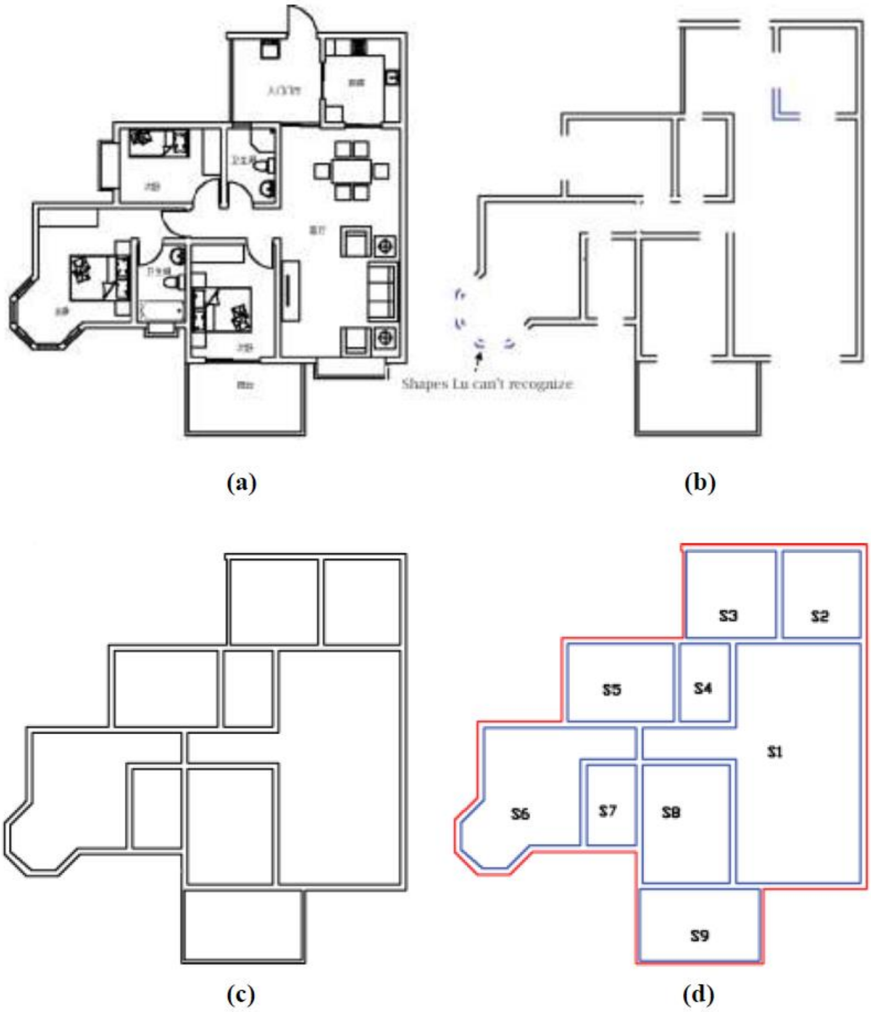


Figure 2-34: Recognized parallel line pairs of walls by Zhu et al. [36]: (a) Original floor plan; (b) recognized walls; (c) preprocessing result; (d) loop searching result.

An advancement of this method over other methods is that on the phase of creating opening equivalent lines, more possible layouts between the opening and its adjacent walls are considered. In Fig. 2-35, in addition to (a) which are the same layout as in Fig. 2-25, layouts like (b) (c) (d) (e) are also considered and corresponding solutions are provided ((g) (h) (i) (j)).

By reviewing these methods, it is found that existing wall detection algorithms which identify walls as close parallel line pairs are not reliable since the performance is influenced by the user-defined threshold for searching for nearby lines. Thus, instead of using wall detection algorithm to extract wall lines from all those disturbing lines, the layers in CAD software, an efficient information segmentation tool, can be used to separately store the wall lines. So that the contours of spaces can be obtained by just searching closed loops among the wall lines and opening equivalent lines, since the topological relationship between walls and openings is not indispensable in the reconstruction of 3D building model. In this thesis, a more general analysis of the layout between walls and openings will be carried out to help the creation of opening equivalent lines.

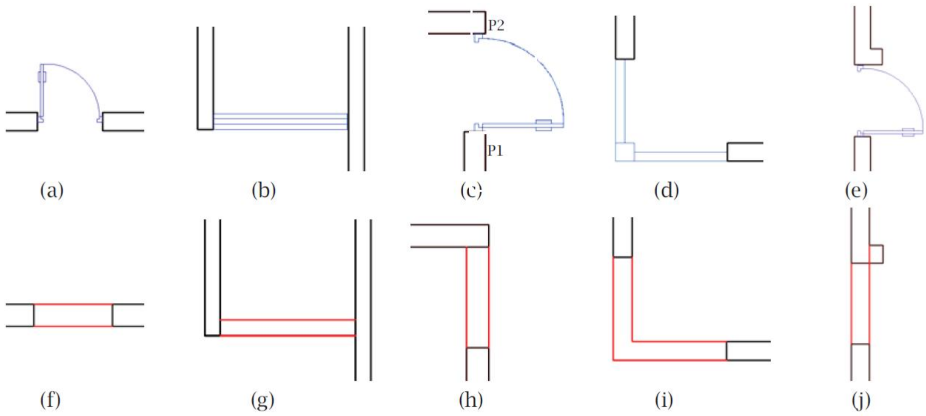


Figure 2-35: Openings and their adjacent walls analyzed in [36]

2-3 3D reconstruction

The 3D reconstruction method used for this thesis is developed by Dr. Marcus Goetz for IndoorOSM data, the extension of OpenStreetMap (OSM) in indoor field. OSM is one of the most popular examples of Volunteered Graphical Information (VGI), which is a newly evolved geodata source in recent years that rises with the idea of crowdsourcing. The aim of OSM is to use a massive amount of crowdsourced geodata collaboratively collected by individuals who can collect geodata using manual survey, GPS devices, aerial photography and other free sources, to create a free editable map of the world for everyone [37].

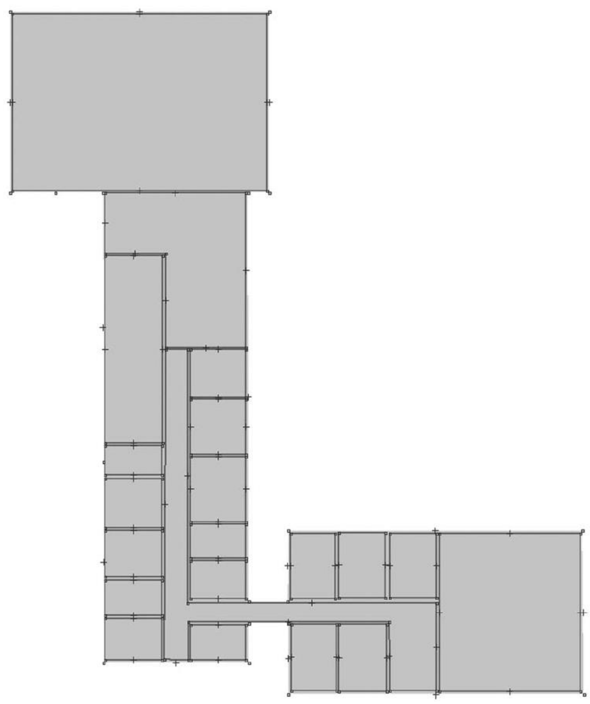


Figure 2-36: Exemplary floor plan of a building, which is mapped according to IndoorOSM in JOSM [3]

The favorable point of IndoorOSM is that it is a rich, open, free-editable and simple-formatted data source with necessary semantics for indoor applications that can be manually input by anyone who can acquire the data. There are several researches have been conducted to discover its application prospect in indoor environment. A 3D indoor routing web application purely using IndoorOSM data was developed by Marcus Goetz [38]. Not only simple route planning applications are promising, the suitability of IndoorOSM data for indoor multi-agent evacuation simulations has also been proven feasible [39]. In addition, a free and open web repository for 3D building models which can be linked to the OSM database was proposed to support the development towards 3D-VGI [40].

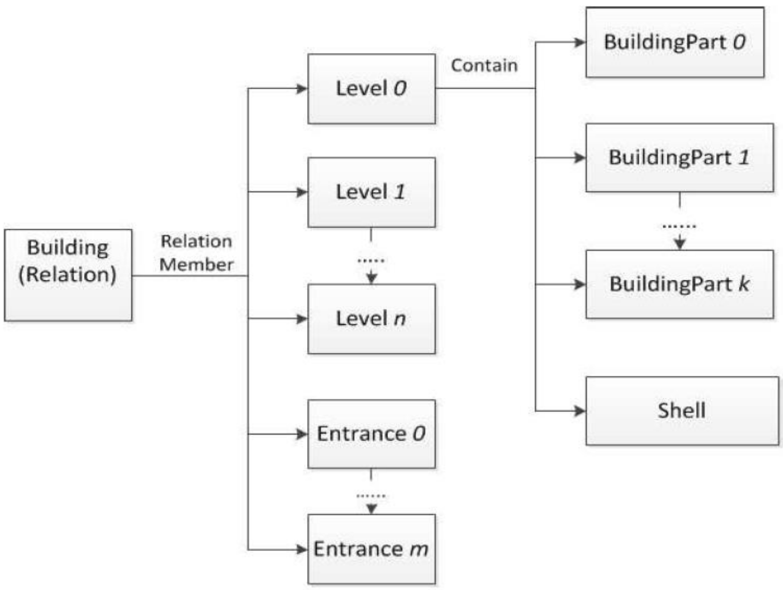


Figure 2-37: Structure of IndoorOSM building model [Liu]

Fig. 2-36 shows a floor plan of a building that is m according to IndoorOSM. There are three basic elements used to represent the floor plan in IndoorOSM: *nodes*, *ways* and *relations*. A *node* can either represent an opening (door, windows), or a corner point in a sequence of *nodes* representing a *way* or entrances of the building. A *way* is a sequence of ordered *nodes*, which can either represent a buildingpart (e.g. rooms, corridors etc.) or the shell (outline) of a level (floor). Each level is represented as a *relation*, a conceptual element, in which there might be several relation-members included. For a level relation, its relation-members are several buildingparts and a level shell. Besides, the whole building is also a *relation*, whose relation-members are a sequence of levels and its entrances. Additional semantic information (attributes) about the building, the buildingparts, the entrances and the openings, is attached as OSM key-value pairs to the OSM elements used to represent them. For example, for a building, information such as address, name, height etc. can be attached to the building *relation*; for a level, information such as name, level number, height etc. can be attached to the level *relation*; for a buildingpart, information such as type of the space, name, height etc. can attached to the *way* representing it; for a window, information such as type of the window, width, height etc. can be attached to the *node* representing it. Fig. 2-37 illustrates the structure of IndoorOSM building model as described above. Fig. 2-38 shows key-value pairs that can be attached to different objects in IndoorOSM.

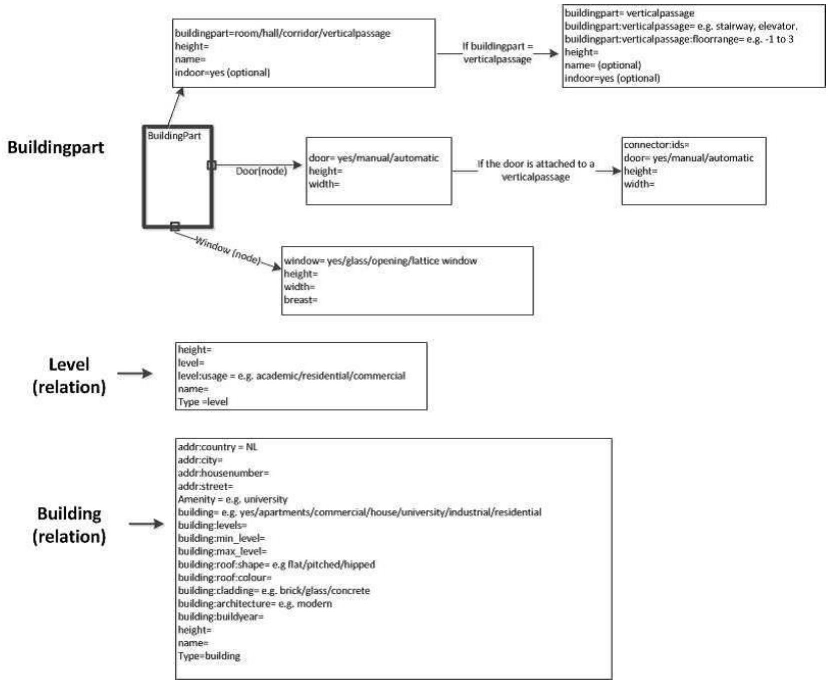


Figure 2-38: Key-values of different objects [Liu]

After a building has been mapped in the way described above, 3D building model can be reconstructed by simple extrusion of the contours of the *way* elements. The general workflow for the generation of CityGML LoD3 and LoD4 models from IndoorOSM data is shown in Fig. 2-39. In the generation of CityGML LoD3 models, only the *ways* representing the shell of each floor are extruded (Fig. 2-40). In the generation of CityGML LoD4 models, in addition to the level shells, *ways* representing buildingparts on the level are also extruded (Fig. 2-41). Fig. 2-42 shows a building model of the OTB research institution in TU Delft created from IndoorOSM data. For a clearer vision of the interior buildingparts, the front facade has been removed in the model.

However, a fatal flaw of crowdsourced geo-data is that the accuracy of the data varies a lot and sometimes can be very frustrating since anybody even non-geomatics professionals can contribute to the datasets. The result of this is the geometric distortion of the outcome 3D models. Marcus Goetz addressed these problems as follows: "some building models revealed slightly dislocated levels", "the position of windows does not fit to the provided width", "different sides of a wall are sometimes not parallel", "some four-sided rooms are obviously not quadrangular" and "many interior walls do not have a thickness" [3] (examples shown in Fig. 2-43).

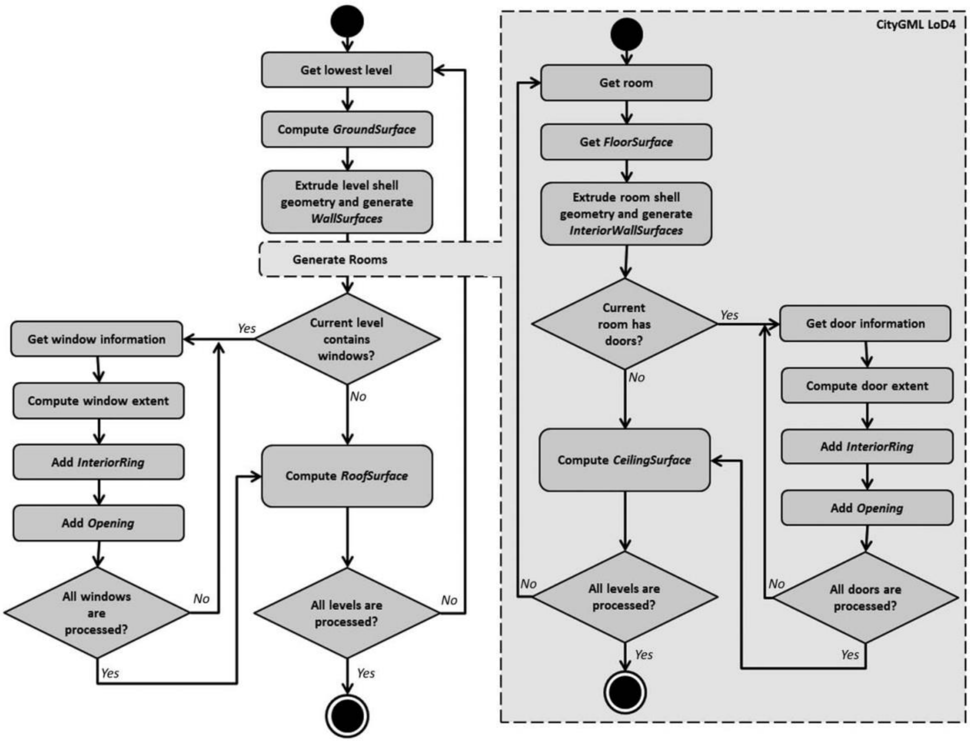


Figure 2-39: General workflow for the generation of CityGML LoD3 and LoD4 models [3]

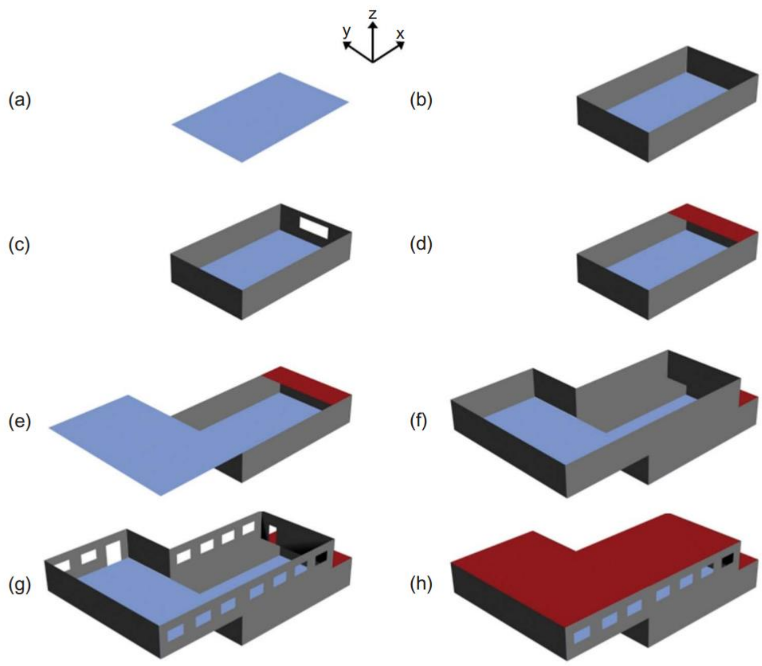


Figure 2-40: Stepwise generation of a CityGML LoD3 building model with IndoorOSM data [3]

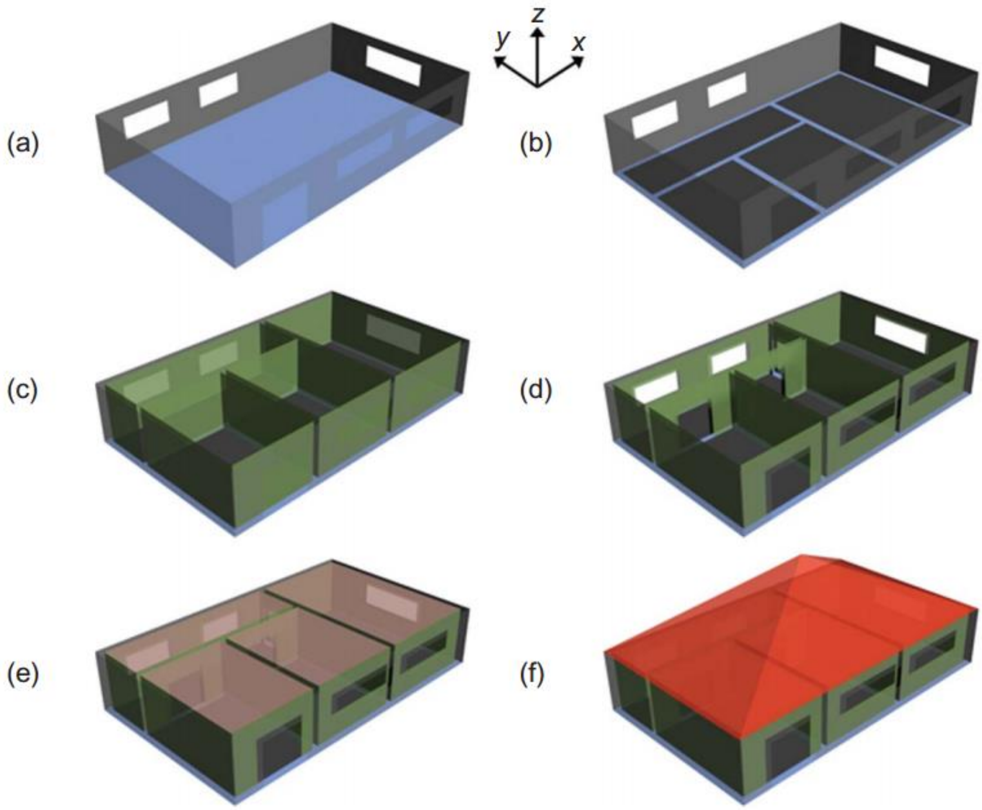


Figure 2-41: Stepwise generation of a CityGML LoD4 building model with interior structures based on IndoorOSM data [3]

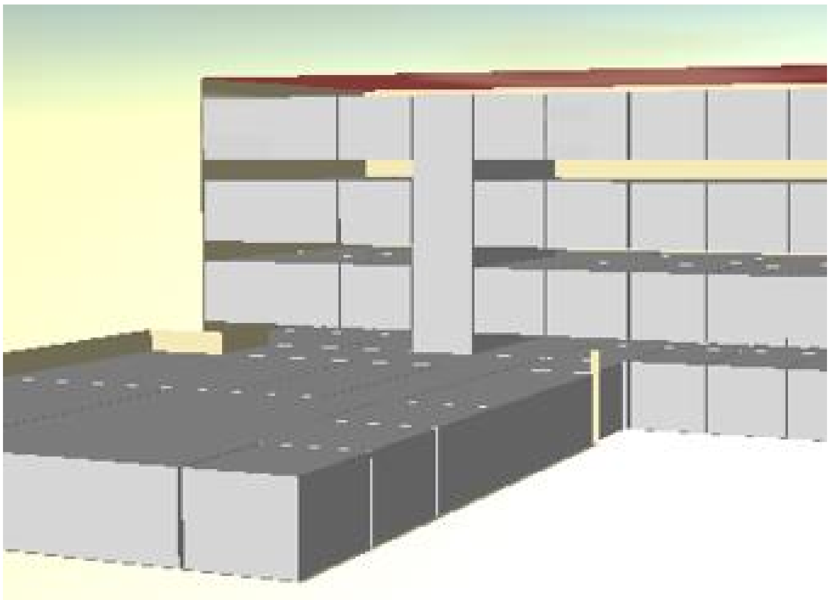


Figure 2-42: A CityGML building model created from IndoorOSM data

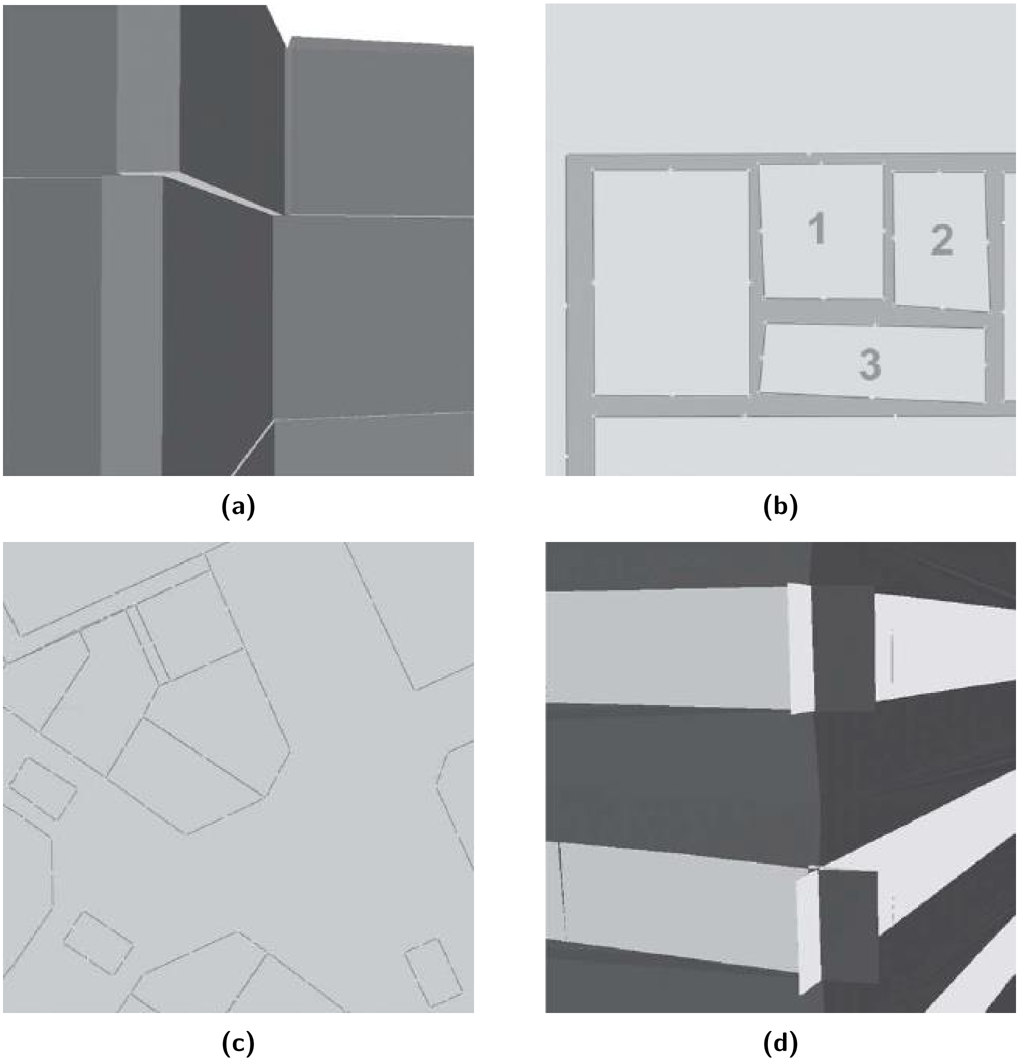


Figure 2-43: Examples of erroneous results caused by inaccurate input geo-data [3]

Chapter 3

2D floor plan processing

This chapter first illustrates how the input floor plans should be redrawn in detail, which includes the content to be kept, the specific representation of the symbols, the layering and the format. After that, each step of processing the redrawn floor plans is explained. After the information is extracted from the redrawn floor plans, it is exported to a database for 3D reconstruction.

3-1 Redrawing of floor plans

Fig. 3-1 shows the overall workflow of redrawing the floor plans.

3-1-1 Software choosing

- AutoCAD will be used as drawing software to redraw the floor plans.

  AutoCAD is a commercial software application for 2D and 3D computer-aided design (CAD) and drafting. It is used across a wide range of industries, by architects, project managers, engineers, graphic designers, and other professionals. It is supported by 750 training centers worldwide as of 1994 [41]. As Autodesk's flagship product, by March 1986 AutoCAD had become the most ubiquitous CAD program worldwide [42]. In addition to its broad spectrum of users, AutoCAD is also very easy and straightforward to use. User can create geometry by just clicking. These reasons make AutoCAD more compatible with the redrawing rules that are going to be described below.

3-1-2 Content segmentation

- Contents of structural objects (i.e. walls and columns), windows and doors should be separated from other contents by layering.

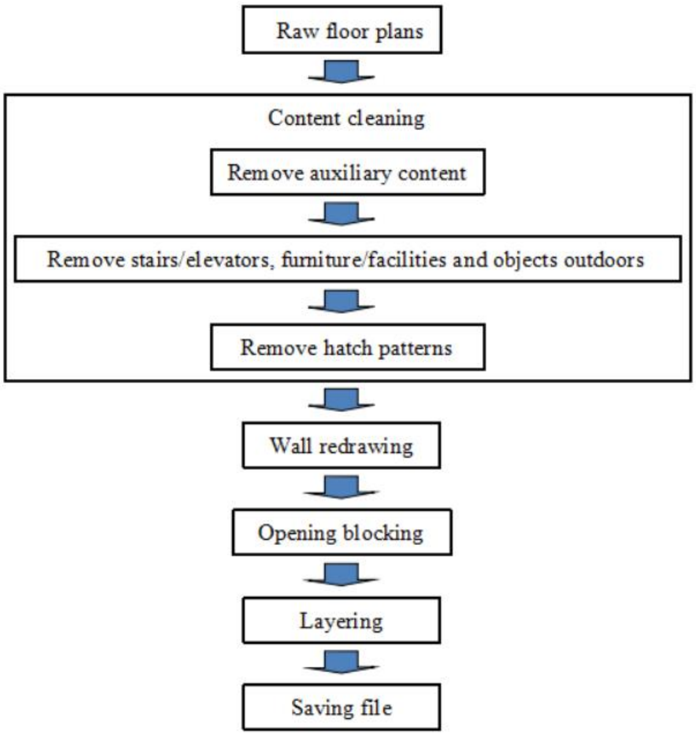
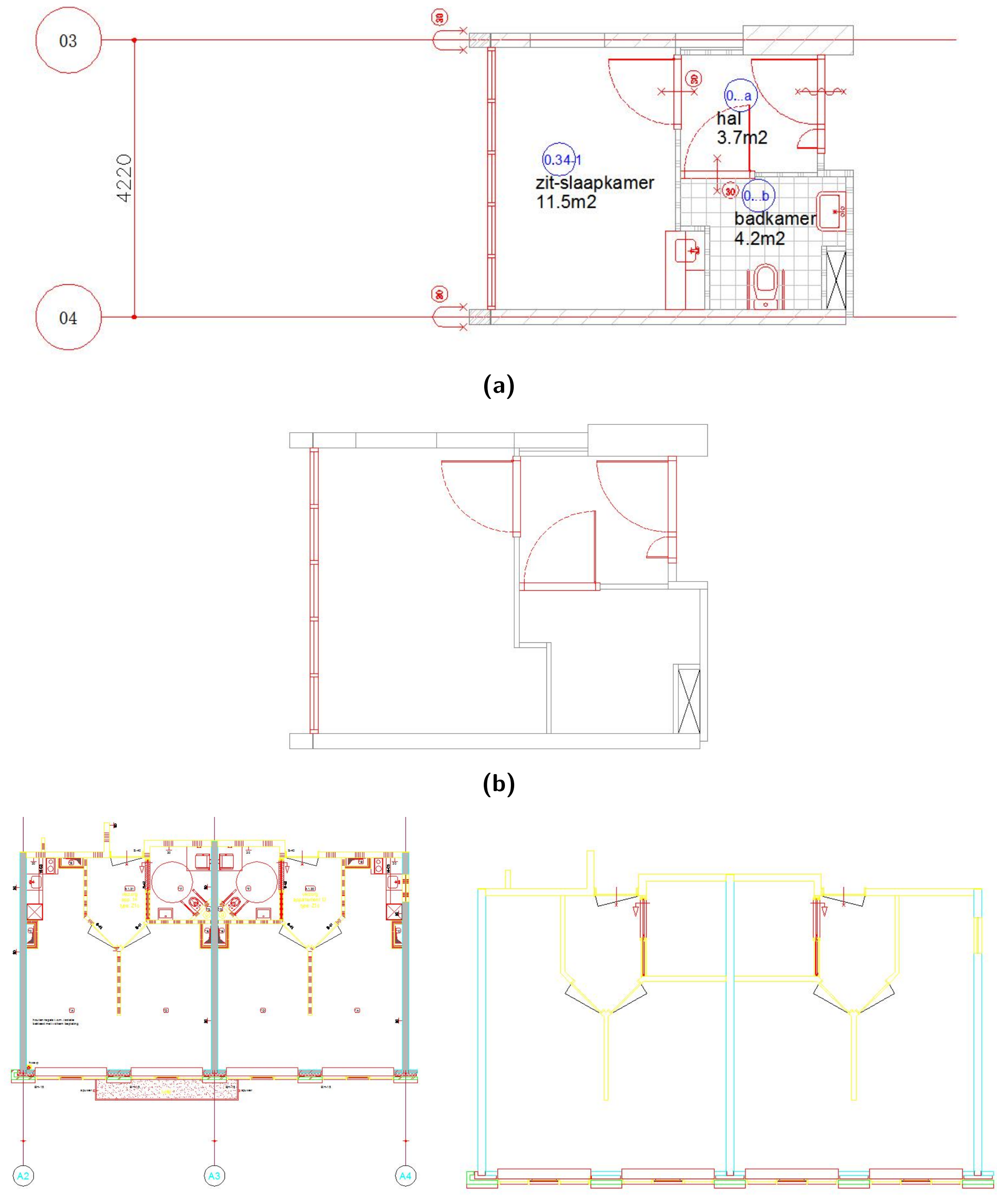


Figure 3-1: Workflow of floor plan redrawing

In last chapter, it has been mentioned that only content of structural objects (i.e. walls and columns), windows and doors will be used for 3D reconstruction. Any other non-structural objects (e.g. toilets, showers, wash-basins, ceramic tiles, kitchen ranges, furniture, stairs, elevators), objects outdoors (e.g. balconies, railings, air-conditioning brackets), indicative information and symbols (e.g. texts, dimensions, auxiliary lines) and hatch patterns within walls, need to be separated from them.

Figure 3-1 (a) and (c) are parts of two floor plans from real life. In Figure 3-1 (a), besides structural objects, windows and doorsčňthere are a toilet, a wash-basin, ceramic tiles, a kitchen range, dimensions, auxiliary lines and some other symbols and texts. In Figure 3-1 (d), in addition to these objects, there are also carpets, bathtubs and a canopy for plants outside the windows. Besides, in Figure 3-1 (a) three different patterns of parallel lines are used as hatch patterns; in Figure 3-1 (d) in addition to hatch pattern of parallel lines, there are also some walls filled by grey solid fill. These contents all need to be moved to other layers. Figure 3-1 (b) and (c) shows what the floor plans look like after removing these contents.



(a)

(b)

(c) (d)

Figure 3-2: Examples of cleaning content in real-life floor plans:(a) (c) parts of floor plans from real life; (b) (d) floor plans after removing redundant objects

3-1-3 Walls

- Data types for wall geometry: LINE, POLYLINE and LWPOLYLINE

  LINE is the most basic entity in AutoCAD, which is a straight segment specified by two endpoints; POLYLINE is a connected sequence of segments created as a single entity, which can be 2D or 3D, and has been supported since very early version of AutoCAD; LWPOLYLINE is simply "lightweight" version of a POLYLINE, which is always 2D and supported in later versions.

- Wall representation

  Walls and columns are represented by closed polygons, which can be drawn by LINE, POLYLINE and LWPOLYLINE, the three most commonly used line entities in CAD files, or any combination of these three entities. Figure 3-2 shows four possibilities of how a simple rectangular wall can be drawn by these three entities. From left to right, the wall is respectively drawn by four LINE entities, one POLYLINE entity, one LINE and one POLYLINE, one POLYLINE and one LWPOLYLINE.

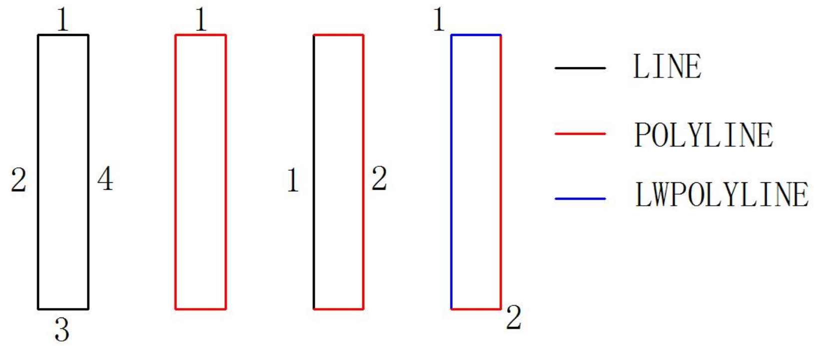


Figure 3-3: A rectangle-shape wall drawn by different entities

Only the outer boundary of each wall will be drawn. In case of that outer walls and inner walls are drawn separately (Figure 3-3 (a)), or that walls and columns are drawn separately (Figure 3-3 (b)), or that there are walls of different types intersecting with each other (Figure 3-3 (c)), intersecting polygons should be merged into one polygon. For example, the bold black polygons in Figure 3-3 (d) (e) (f) are the outer boundary of walls shown in Figure 3-3 (a) (b) (c). Therefore, the redrawing of Figure 3-3 (a) (b) (c) should look like Figure 3-3 (g) (h) (i) respectively.

Last, in case that there is a hollow vertical shaft for air canal, pipelines and electric wires (Figure 2-10), the outline of the shaft should also be drawn as an inner ring of the polygon that represents the structural object this shaft belongs to (Figure 3-4 (a)).Based on the rules described above, the final representation of the walls of Figure 3-1 (a) and (d) should be redrawn as below (bold black contours).

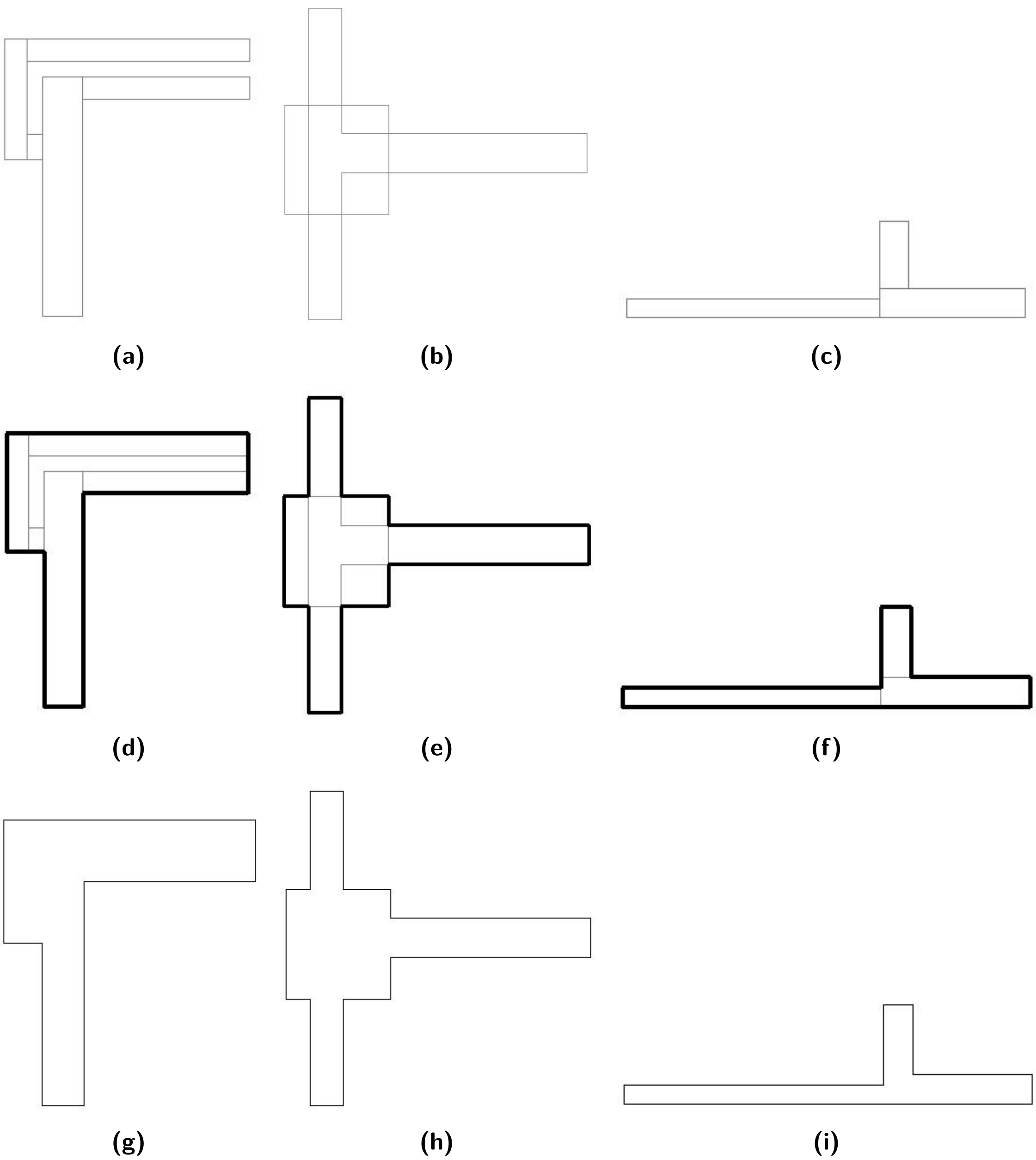


Figure 3-4: Examples of redrawing of walls: (a) (b) (c) representation of walls in real-life floor plans; (d) (e) (f) outer boundary of walls in (a) (b) (c); (g) (h) (i) redrawn representation of walls in (a) (b) (c)

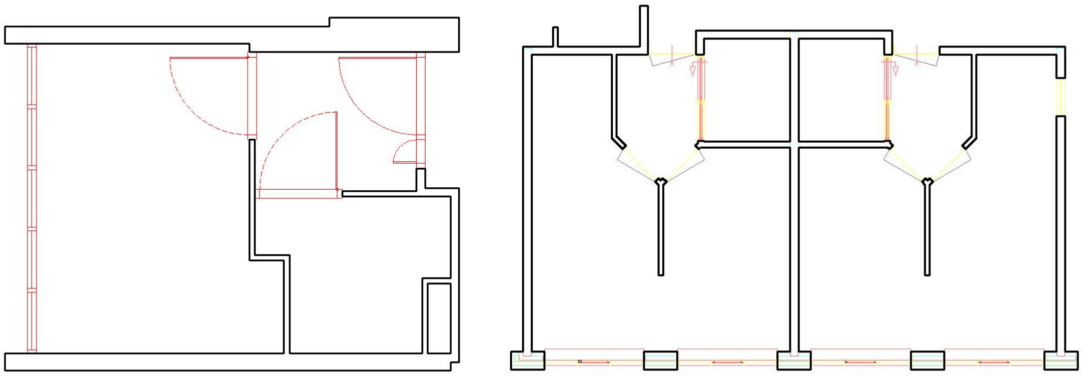

(a) (b)

Figure 3-5: Final representation of walls of Figs. 3-2a and 3-2d

3-1-4 Openings

- Data types for opening geometry: LINE, POLYLINE, LWPOLYLINE and ARC.

  In addition to LINE, POLYLINE and LWPOLYLINE that have been introduced above, ARC entity is also used in opening geometry. An ARC is a portion of a circular arc, which can be created in AutoCAD in many ways. For example, it can be created by specifying three points on it, or by specifying its start point, center and angle.

  LINE, POLYLINE, LWPOLYLINE and any other combinations of them can be used to drawn lintels, doorjambs and door panels in the same way of drawing walls. ARC entities are used to draw the swinging trajectory of doors and windows.

- Symbol representation

  For swing doors, lintel must be drawn because lintel is the most important part of the whole door that indicates the location of the door. In case that lintel is missing in the original symbol, it should be redrawn to fit the gap between its adjacent walls or the gap between its doorjambs if doorjambs exist in the original symbol. Besides lintel, other components such as doorjambs, door panel and trajectory should be kept if they exist in the original symbol. If the trajectory is going to be drawn, it should be drawn by an ARC entity. The angle of the ARC entity is not compulsory, which can range from 0 to 90 degree. Because by doing this, trajectory in the symbol can be distinguished from other components which are composed of linear primitives. Then by checking the center of the ARC entities, the location of the door hinge can be determined so that the location of the lintel can be indicated. This can help to minimize the bounding box in 2D processing phase. In addition, annotating primitives and texts shown in Figure 2-14 should be removed.

  Figure 3-5 shows the redrawing of the door symbol in Figure 2-12. The modified parts are indicated by dark blue stroke. Figure 3-5 (a) corresponds to Figure 2-12 (b), whose trajectory has redrawing of symbols shown in Figure 2-12 been redrawn by an arc; Figure 3-5 (b) (c) (d) respectively correspond to Figure 2-12 (d) (g) (i), whose missing lintel has been added; Figure 3-5 (e) (f) (g) (h) respectively correspond to Figure 2-12

(a) (e) (f) (h), whose missing lintel has been added and trajectory has been redrawn by an arc. Figure 2-12 (c) and (i) do not need to be redrawn.

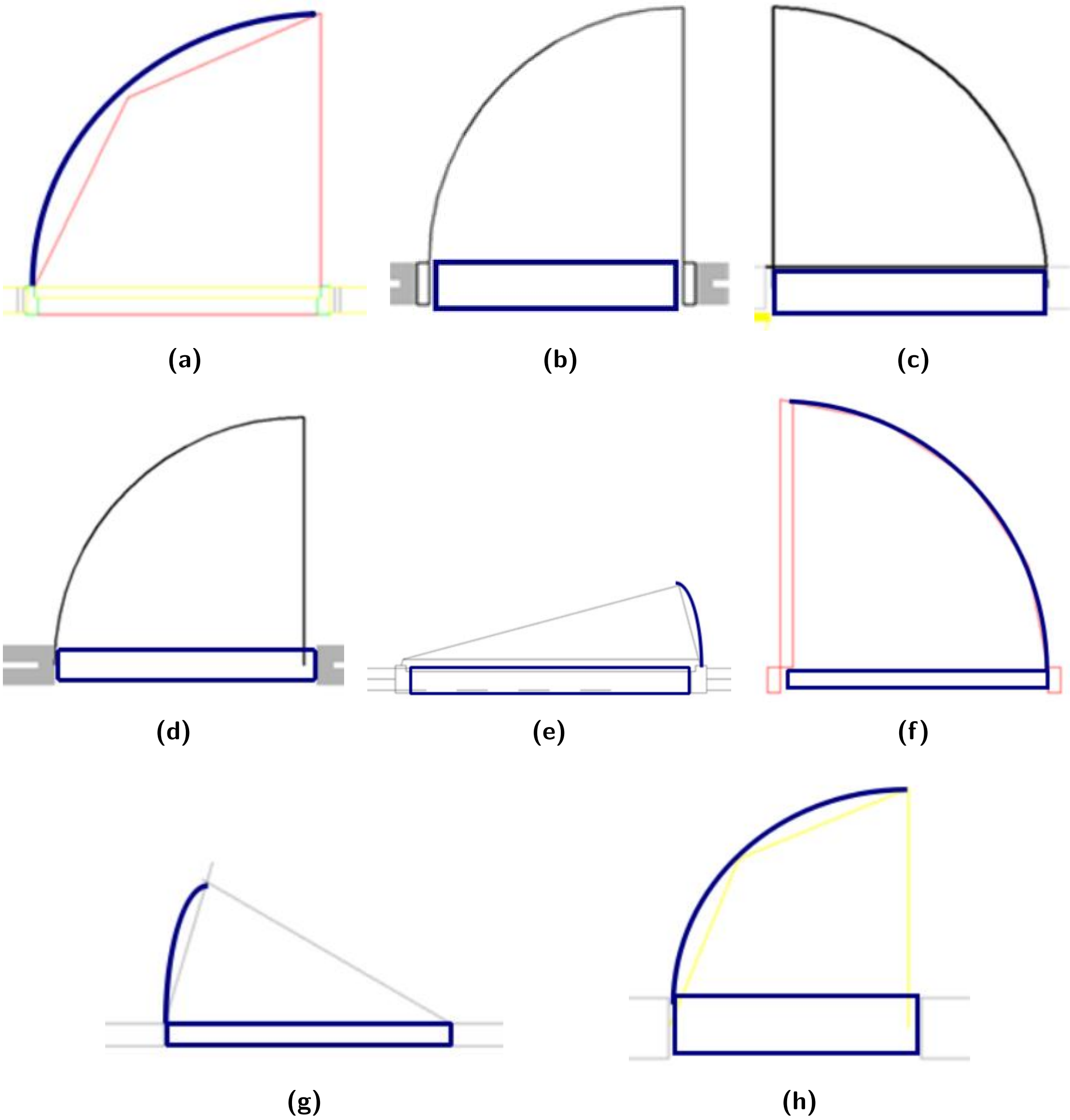


Figure 3-6: Redrawing of doors in Fig. 2-12

For sliding doors, pocket doors and bi-fold doors, only the arrows and annotating contents in the symbols should be removed. No other redrawing is required.

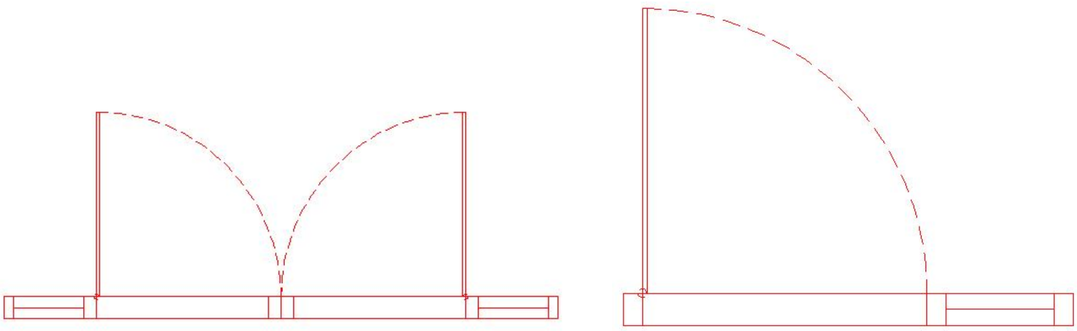
For casement windows and combined windows within which a casement window is included, they should be redrawn in same way as swing doors. For other windows, besides removing annotating contents, no extra redrawing is required, since the bounding boxes of these windows correctly indicate their locations.

- Symbol blocking

  After the symbols have been redrawn, each of them should be saved as a block entity. A block is a named group of objects that act as a single 2D or 3D object. It can be used to create repeated content such as drawing symbols, common components, and standard details. By updating a blockąŕs definition, all instances of this block in the drawing can be updated together. Blocks help designers save time, maintain consistency, and reduce file size by reusing and sharing content rather than redrawing it every time it is needed [43]. The benefit of blocking each opening symbol is that each symbol can be dealt with separately as a whole to calculate its bounding box without applying symbol recognition. Except for openings, blocking should not be used for any other purpose. A block can be copied and used multiple times for duplicated symbols.

In case of combination of openings of same type, i.e. door with door (Figure 2-13) or window with window (Figure 2-17 (g)), they should be blocked together as one single block. Within this block, each window and door can either be a block as well, or they can be directly drawn by primitives.

In case of combination of windows and doors (Figure 3-6), they should be blocked together as a single block, which counts for a door block. Within this block, each window and door can either be a block as well, or they can be directly drawn by primitives.



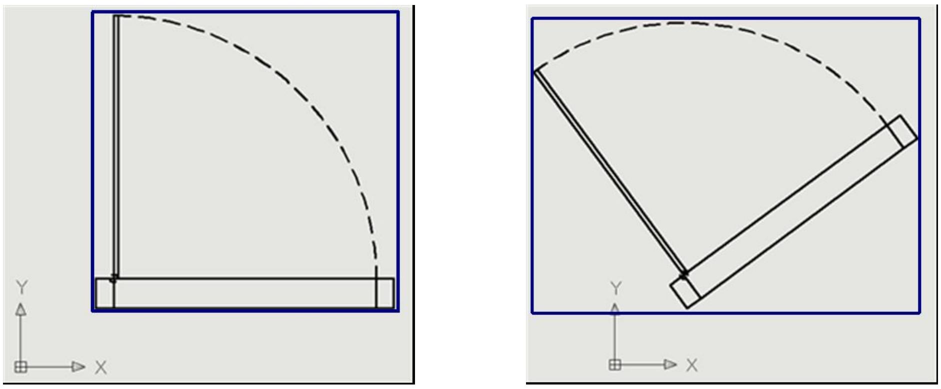
(a) **(b)**

Figure 3-7: Combination of windows and doors

- Block defining

  In AutoCAD, each block has its own coordinate system and a reference point, which can be freely located by the designer in the coordinate system. This reference point is also the origin of the system. Every primitive in the block is given coordinates with respect to the reference point in the local coordinate system. When a block instance is inserted in a drawing, all primitives in the block are transformed to the drawingąŕs coordinate system by translating, rotating and scaling. Besides, each block has a name. Whenever a repeating symbol is needed, a block instance with the same name will be inserted. The naming of the blocks is not mandatory in this thesis.

  In the following 2D processing phase, the bounding box of each block will be calculated to indicate the outline and orientation of the opening. Thus, the opening should be defined aligned with the x and y axes of the local coordinate system. Figure 3-7 shows two scenarios. In Figure 3-7 (a), the symbol is defined in the coordinate system in the expected way. It can be seen that the bounding box (in dark blue) correctly indicates the outline and orientation of the symbol. In Figure 3-7 (b), the symbol is define tilted and the bounding box is stretched and not in the same orientation as the symbol.



(a) **(b)**

Figure 3-8: Opening with bounding box in local coordinate system

3-1-5 Layering

- Walls, window blocks and door blocks should be separately stored in three different layers, respectively with the name of ąőWallsąŕ, ąőWindowsąŕ and ąőDoorsąŕ. Figure 3-8 shows how the layers should be organized in AutoCAD layer properties manager.

Figure 3-9: Layer properties manager

3-1-6 Format

- The file should be saved in DXF format.

  DXF is one of the most widely supported vector formats in the world today. It is an open standard, of which both binary and ASCII version exist. The ASCII version of DXF file can be read with a texteditor, making DXF an easy format to parse. There are several open-source libraries for manipulating DXF files (e.g. dxfgrabber, dxfwrite, ezdxf, SDXF).

- Floor plan of each floor should be saved in separated files.

3-2 Redundancy cleaning

In last chapter, it has been mentioned that manually generated input floor plans typically suffer from many drafting errors and redundancies. In spite that raw floor plans have been redrawn according to the rules described in last section, they might still contain drafting errors and redundancies as long as they are redrawn by hand. These errors and redundancies might be visually imperceptible, but they might make algorithms to be used in later phases generate some unpredictable results. Therefore, they have to be found and corrected. In the literature review, the method proposed by Rick only considered drafting errors of disjoint vertices, which are caused by disjoint lines and false intersecting lines. In this section, two types of redundancies, null-length line segments and duplicated line segments, will be introduced. Besides, there are five more specific cases of duplicated line segments. Definitions of all of them will be given, based on which the algorithm to detect and fix them will be proposed accordingly.

A null-length line segment will be created when a user incidentally assign a same point as both the start point and the end point of a line, since this kind of operations will not be recognized as illegal in most CAD applications. Although a null-length line segment is visually recognized as a point, its data type stored in the CAD file is still “LINE” and thus can still be read into later algorithms which work on content in the wall layer. In addition to line segments whose length are exactly zero, those that are shorter than a given threshold will also be regarded as null-length line segments in this thesis. They are created in a similar way that the designer incidentally puts its start point and end point extremely closed to each other. Thus, the definition of null-length line segments is given as follows, where R notates all line segments in wall layer.

Definition 1 (*NULL-LENGTH*). Let a be a line segment in R, and *L(a)* the length of a. Given a fixed threshold δ, a is considered to be *NULL-LENGTH* when $L(a) \leq \delta$.

Duplicated line segments happen when a single straight line segment in the floor plans is mistakenly represented by multiple line segments. The definition is given as below:

Definition 2 (*DUPLICATED*). Let a and b be line segments (including endpoints) in R^2. Let r and s be the lines containing a and b respectively, and *a’* and *b’* the projections of a and b onto r and s. Given a fixed threshold ε, the pair (a, b) (also the pair (b, a)) is *DUPLICATED* if and only if all the conditions below are held:

(1) Neither of a and b is *NULL-LENGTH*;

(2) $a$and b are parallel: $a \| b$;

(3) *a’*and b (and also $a$and *b’*) overlap: $a \bigcap b' \neq \oslash$;

(4) The distance between r and s is less than or equal to the threshold: $d(r, s) \neq \varepsilon$;

Based on the geometric relationship between those line segments, DUPLICATED line segments can be further divided into five specific cases: OVERLAPPING, CONTAINING, CONTAINED, IDENTICAL and CONSECUTIVE (Figure 3-9). In real floor plans, DUPLICATED line segments in these cases are so closed to each other that the errors are always visually imperceptible. But in Figure 3-9, line segments are drawn clearly separated from each other on purpose so that it can be easier to understand the geometric relationship between them: in case of Figure 3-9 (a), two line segments partially overlap with each other. The leftmost endpoint of the upper line segment connecting with the rightmost endpoint of the other line segment formed the expected line segment; in case of Figure 3-9 (b), a shorter line segment is contained by the other one. The expected line segment is just the longer one after removing the shorter one; in case of Figure 3-9 (c), the two line segments are identical. One of them should be removed; in case of Figure 3-9 (d), two line segments are consecutive sharing a common endpoint. The expected line segment is the union of them.

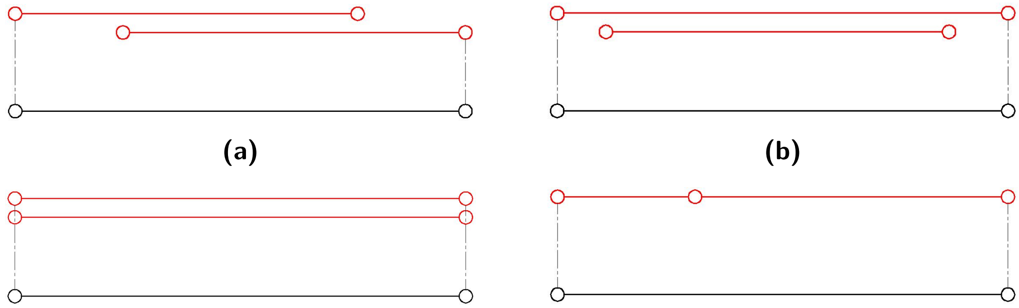


Figure 3-10: Cases of DUPLICATED line segments: (a) OVERLAPPING; (b) CONTAINING and CONTAINED; (c) IDENTICAL; (d) CONSECUTIVE.

There is one more definition needs to be given before the definitions of these specific cases can be provided.

Definition 3 (*IN and OUT*). Let a be a line segment (excluding endpoints) in R, and r a line containing a. P is a point on r. If P intersects with a, i.e. $P \bigcap a \neq \oslash$, then P is IN a; else P is OUT a.

Then, these five specific cases of DUPLICATED line segments can be defined as follows:

Definition 4 (*OVERLAPPING*). Let a and b be line segments (including endpoints) in R^2. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Let r and s be the lines containing a and b respectively. Let A_1' and A_2' be the projections of A_1 and A_2 onto s, and B_1' and B_2' be the projections of B_1 and B_2 onto r. Given a fixed threshold ε, the pair $(a,\ b)$ (also the pair $(b,\ a)$) is *OVERLAPPING* if and only if all the conditions below are held:

(1) The pair $(a,\ b)$ (also the pair $(b,\ a)$) is *DUPLICATED*

(2) A_1' is IN b and

(3) A_2' is OUT b, or A_2' is IN b and A_1' is OUT b, or B_1' is IN b and B_2' is OUT b, or B_1' is IN b and B_2' is OUT b

Definition 5 (*CONTAINING*). Let a and b be line segments (including endpoints) in R^2. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Let r and s be the lines containing a and b respectively. Let A_1' and A_2' be the projections of A_1 and

A_2 onto s, and B_1' and B_2' be the projections of B_1 and B_2 onto r. Given a fixed threshold ε, the pair $(a,\ b)$ (also the pair $(b,\ a)$) is *OVERLAPPING* if and only if all the conditions below are held:

(1) The pair $a,\ b$ is *DUPLICATED*

(2) A_1' is *OUT* b and A_2' is *OUT* b, or B_1' is *IN* a and B_2' is *IN* a

Definition 6 (*CONTAINED*). Let a and b be line segments (including endpoints) in R^2. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Let r and s be the lines containing a and b respectively. Let A_1' and A_2' be the projections of A_1 and A_2 onto s, and B_1' and B_2' be the projections of B_1 and B_2 onto r. Given a fixed threshold ε, the pair $(a,\ b)$ (also the pair $(b,\ a)$) is *OVERLAPPING* if and only if all the conditions below are held:

(1) The pair $a,\ b$ is *DUPLICATED*

(2) A_1' is *IN* b and A_2' is *IN* b, or B_1' is *OUT* a and B_2' is *OUT* a

Definition 7 (*IDENTICAL*). Let a and b be line segments (including endpoints) in R^2. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Let r and s be the lines containing a and b respectively. Let A_1' and A_2' be the projections of A_1 and A_2 onto s, and B_1' and B_2' be the projections of B_1 and B_2 onto r. Given a fixed threshold ε, the pair $(a,\ b)$ (also the pair $(b,\ a)$) is *OVERLAPPING* if and only if all the conditions below are held:

(1) The pair $a,\ b$ is *DUPLICATED*

(2) $A_1' = B_1$ and $A_2' = B_2$, or $A_1' = B_2$ and $A_2' = B_1$, or $A_1 = B_1'$ and $A_2 = B_2'$, or $A_2 = B_1'$ and $A_1 = B_2'$

Definition 8 (*CONSECUTIVE*). Let a and b be line segments (including endpoints) in R^2. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Let r and s be the lines containing a and b respectively. Let A_1' and A_2' be the projections of A_1 and A_2 onto s, and B_1' and B_2' be the projections of B_1 and B_2 onto r. Given a fixed threshold ε, the pair $(a,\ b)$ (also the pair $(b,\ a)$) is *OVERLAPPING* if and only if all the conditions below are held:

(1) The pair $a,\ b$ is *DUPLICATED*

(2) A_1' is *OUT* b and $A_2' = B_1$, or A_2' is *OUT* b and $A_1' = B_1$, or A_1' is *OUT* b and $A_2' = B_2$, or A_2' is *OUT* b and $A_1' = B_2$

To determine whether a pair of line segments is DUPLICATED, which types of DUPLICATED line segments they are, and how to fix them, the concepts of BUFFER_POLYGON and MERGED is introduced.

Definition 9 (*BUFFER POLYGON*). Let a be a line segment in $\boldsymbol{R}$, A and B be the two endpoints of a. Let r and s be the lines parallel with a on its both sides with a given offset distance ε. Let A_1 and A_2 be the projections of A onto r and s respectively, and B_1 and B_2 the projections of B onto r and s respectively. The *BUFFER POLYGON* of a (indicated by *BF(a)*) is the polygon bounded by line segments $\overline{A_1A_2}, \overline{A_2B_2}, \overline{B_2B_1}, \overline{B_1A_1}$.Line segments $\overline{A_1A_2}, \overline{A_2B_2}, \overline{B_2B_1}, \overline{B_1A_1}$ are called the boundary of *BF(a)*, indicated by *BFB(a)*.

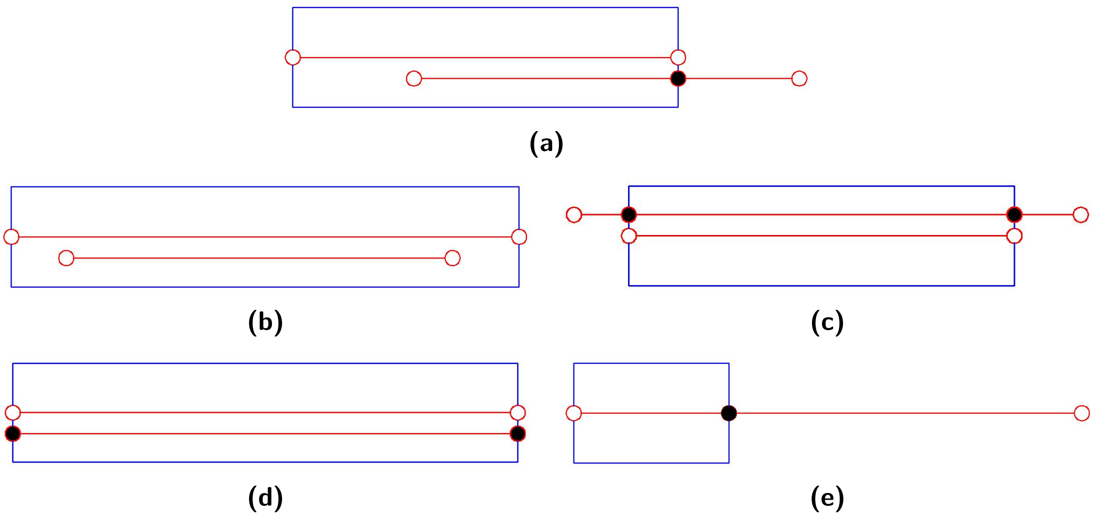


Figure 3-11: BUFFER POLYGON in cases of: (a) OVERLAPPING; (b) CONTAINING; (c) CONTAINED; (d) IDENICAL; (e) CONSECUTIVE

Definition 10 (*MERGED*). Let a and b be line segments (including endpoints) in R^2. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Line segment (A_i, B_j) $(i, j \in \{0, 1\})$ is the *MERGED* line segment of a and b (denoted by *M(a, b)*). If and only if the distance between A_i and B_j is maximum: $d(A_i, B_j) = MAX[d(A_m, B_m)], m, n \in \{0, 1\}$.

After all these terms and concepts have been clarified, the ways of identifying and fixing the redundancies are summarized in Table 3-1 .

Table 3-1: Identification and fixing of drafting errors

Drafting errors	Identification	Fixing
NULL-LENGHTH	$L(a) \leq \delta$	Delete a from R
CONTAINING	$BFB(a) \cap b =$ NULL	Delete b from R
OVERLAPPING	$BFB(a) \cap b =$ POINT	Delete a and b from R Add *M(a, b)* to R
CONSECUTIVE		
CONTAINED	$BFB(a) \cap b =$ MULTI-POINT	Delete a from R
IDENTICAL		

The pseudocode of this step is given in Function FixRedundancy. It can be seen that the whole process is iterative. Every time the first line segment in R is compared to each of the resting line segments in R. Between line 4 and line 22, the codes try to identify the pair as one of redundancy types and fix them accordingly, according to the criteria listed in Table 3-1. After a redundancy is identified and fixed, all the line segments will be taken back to line 3 to repeat the previous process. This is because a line segment might have DUPLICATED relationship with more than one other line segment, or a new line segments generated from fixing one DUPLICATED case, might in turns be DUPLICATED with another line segment. The only gate to get out of the iteration is between line 23 and line 26, when a line segment has been compared to every other line segment and no redundancy has been matched. This line segment then will be moved from R to a new group.

Algorithm 1 FixRedundancy

```
Input: R: line segments with redundancy to be fixed
Input: δ: the threshold for determining NULL-LENGTH line segments
Output: ε: the threshold for determining closed line segments
1: NR ← empty
2: while len(R) ≤ 0 do
3:     l0 ← first line in R
4:     if L(l0) ≤ δ then
5:         delete l0 from R
6:         continue
7:     end if
8:     for each li in the rest of R do
9:         if l0||li and d(l0,li)≤ ε then
10:            if BFB(l0) capli = NULL then
11:                delete li from R
12:                break
13:            else if BFB(l0) capli = POINT then
14:                nl ← M(l0, li)
15:                delete l0, li from R
16:                R ← nl
17:                break
18:            else if BFB(l0) capli = MULTI POINT then
19:                delete l0 from R
20:                break
21:            else
22:                if li is the last line in R then
23:                    NR ← l0
24:                    delete l0 from R
25:                end if
26:            end if
27:        else
28:            if li is the last line in R then
29:                NR ← l0
30:                delete l0 from R
31:            end if
32:        end if
33:    end for
34: end while
35: return NR
```

3-3 Line grouping

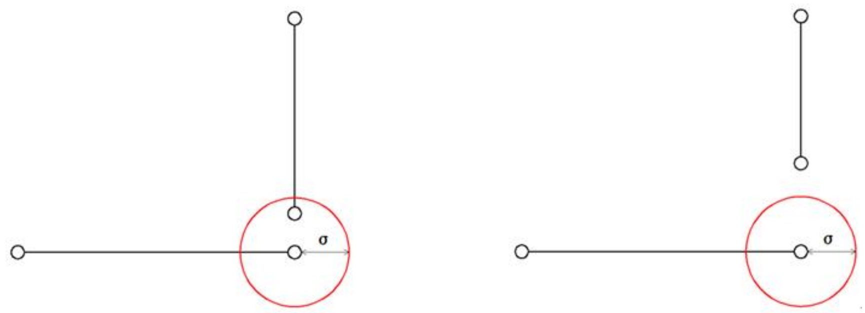
In this step, line segments within which redundancies have been cleaned will be divided into groups. Lines in each group represent a closed polygon in the floor plan. In the meantime, drafting errors of disjoint vertices caused by disjoint lines and false intersecting lines will be detected and fixed in the grouping process. In order to better explain the line grouping algorithm, two terms need to be defined first as below:

Definition 11 (*CHAIN*). A ordered sequence of points $\{P_1, P_2, \ldots, P_i\}(i \geq 2)$ is called a CHAIN, denoted by $\mathrm{C}\{P_1, P_2, \ldots, P_i\}(i \geq 2)$.

Definition 12 (*POLYGON*). A POLYGON is a closed CHAIN $\mathrm{C}\{P_1, P_2, \ldots, P_i\}(i \geq 2)$, whose first point P_1 and last point P_i are coincided, e.g. $P_1 = P_i$. To avoid the duplicate of points, the last point will not be stored. Thus, a POLYGON is denoted by $\mathrm{P}\{P_1, P_2, \ldots, P_j\}(3 \geq j \geq i-1)$.

There are two things that need to be noted. First, a line with two endpoints P_1, P_2 can also be a CHAIN according to the definition. Besides, a POLYGON just represents the exterior ring of a polygon. Any interior rings of the polygon will not be included in the POLYGON.

Definition 13 (*CONNECTED* and *UNCONNECTED*). Let a and b be line segments in $\mathbf{R^2}$. Let A_1 and A_2 be the two endpoints of a, and B_1 and B_2 the two endpoints of b. Given a fixed threshold ε, if there exist A_i and B_j $(i, j \in \{1, 2\})$, that hold the conditions that the distance between A_i and B_j is less than or equal to the threshold, i.e. $d(A_i, B_j) \leq \varepsilon$. A_i and B_j are called CONNECTED vertices, a and b are called CONNECTED line segments. Otherwise, they are UNCONNECTED.



(a) CONNECTED **(b)** UNCONNECTED

Figure 3-12: Illustration of connected and unconnected line segments

The pseudocode of this step is given in Algorithm LineGrouping. It is also a repetitive process. Every time it takes ungrouped line segments into Function FindClosedChains, which returns both closed and unclosed CHAINs among the line segments with the given threshold. The returned closed CHAINs will be added to a new group as recognized POLYGONs. This process repeats until all the line segments have been grouped into POLYGONs or the repeating times have reached certain number. In addition, the threshold taken into Function FindClosedChains to determine CONNECTED line segments is proportional to the number of times this process has been repeating.

There are three cases of disjoint vertices. As shown in Figure 3-12, (a) and (b) happen when the CONNECTED line segments are collinear; (c) and (d) happen when the CONNECTED line segments are not collinear but overlapping with each other; (e) and (f) happen when the CONNECTED line segments are not collinear and disjoint. For (a) and (b), the endpoint of the ungrouped line segment on the other side of the connecting side (the red node) will be added to the CHAIN; for (c) (d) (e) and (f), first the intersection of the CONNECTED line segments will be calculated by Function IntersectingPoint. Then the calculated point and the endpoint on the other side of the connecting side (the red nodes) will be added to the CHAIN. The yellow line segments in the figure indicate the new line segments created in this process. For each case, the connecting side could be at the start or the end of the CHAIN. In (a) (c) (e), the connecting happens at the start of the CHAINs, while in (b) (d) (f), the connecting happens at the end of the CHAINs. For (a) (c) (e), the new points will be inserted to the beginning of the CHAINs; for (b) (d) (f), the new points will be appended to the CHAINs at the end. This part is shown in Function FindClosedChains line 14, 17 and 26.

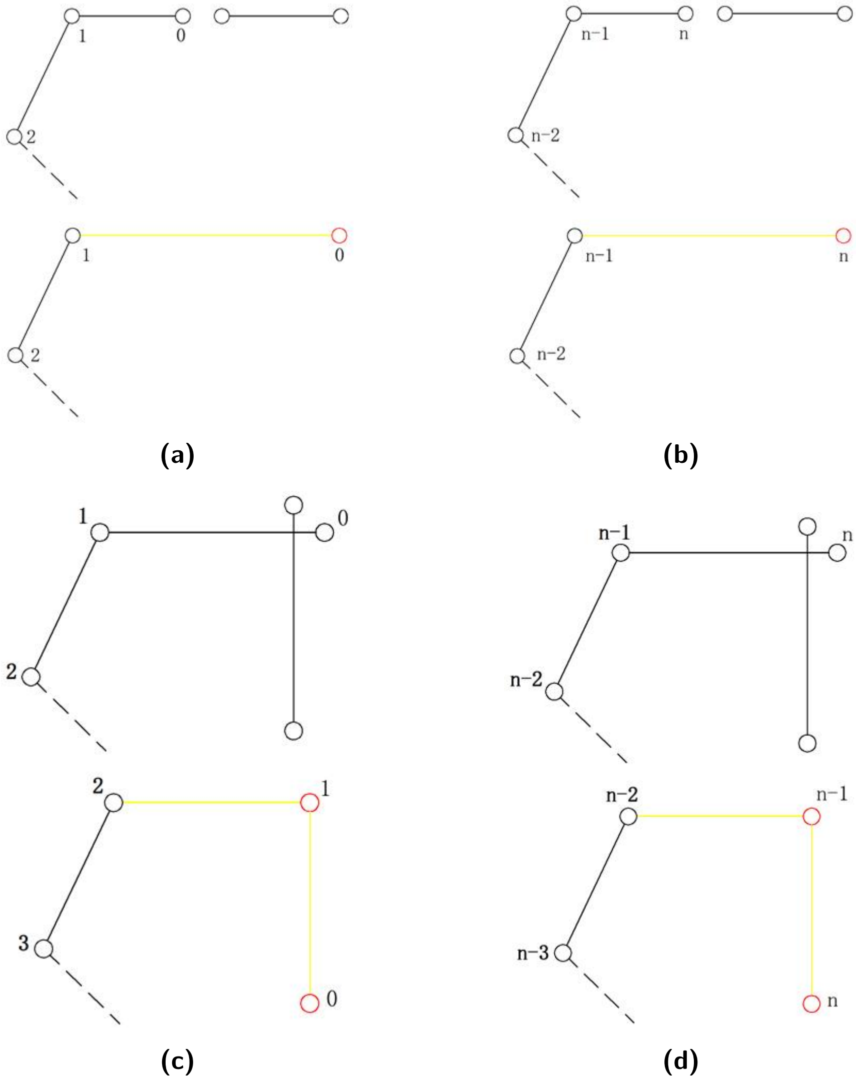


Figure 3-13: Fixing of disjoint vertices: cases when CONNECTED line segments are(a) (b) collinear; (c)(d) collinear but overlapping; (e) (f) collinear and disjoint.

Algorithm 2 LineGroupig

Input: R: line segments to be grouped
Input: σ: the threshold to consider two vertices are CONNECTED
Input: n: number of times the process is allowed to repeat
Output: P: POLYGONs that have been successfully detected

$L \leftarrow R$
$C \leftarrow empty$
$CC \leftarrow empty$
$k \leftarrow 1$
while $len(R) \leq 0$ or ($k \leq n$and C is not empty) **do**
    if $k \neq 1$ **then**
        $L \leftarrow$ break every chain in C into line segments
        $CC \leftarrow empty$
    end if
    C, CC$\leftarrow$ FindClosedChains(L, $k\delta$)
    $P \leftarrow CC$
    $CC \leftarrow empty$
    $k \leftarrow k + 1$
end while
return P

Algorithm 3 Function: FindClosedChains

Input: L: lines to be grouped
Input: d:the threshold to consider two vertices are CONNECTED
Input: C: group of detected unclosed CHAINs
Output: CC:group of detected closed CHAINs

```
C ← empty
CC ← empty
for each l ∈ L do
    if C is empty then
        c ← two endpoints of l
        C ← c
        continue
    end if
    for each c ∈ C do
        l0 ← first line segment of c
        l1 ← last line segment of c
        if l and l0 are CONNECTED within d =True then
            newpoints ← FixDisjointVertices (l, l0)
            Insert newpoints to the beginning of c
            if l and l1 are CONNECTED within d =True then
                newpoints ← FixDisjointVertices (l, l1)
                Insert newpoints to the end of c
                CC ← c
                delete c from C
                break
            else
                break
            end if
        else if l and l1 are CONNECTED within d =True then
            newpoints ← FixDisjointVertices (l, l2)
            Insert newpoints to the end of c
            break
        else
            continue
        end if
    end for
end for
return C, CC
```

Algorithm 4 Function: FixDisjointVertices

Input: l1: line segment in a chain that is CONNECTED with a ungrouped line segment
Input: l2: ungrouped line segment to be added to a chain
Output: newpoints: new points that should be added to a CHAIN

```
if C is empty then
    return the endpoint of l2 on the other side of the connecting part
else
    Pt ← IntersectingPoint(l1, l2)
    return Pt, the endpoint of l2 on the other side of the connecting part
end if
```

Algorithm 5 Function: IntersectingPoint

Input: l1: line segment in a chain that is CONNECTED with a ungrouped line segment
Input: l2: ungrouped line segment to be added to a chain
Output: newpoints: new points that should be added to a CHAIN

$P_{11}, P_{12} \leftarrow$ endpoints of l1
$P_{21}, P_{22} \leftarrow c$ endpoints of l2
$A1 \leftarrow P_{12} \cdot y - P_{11} \cdot y$
$B1 \leftarrow P_{11} \cdot x - P_{12} \cdot x$
$C1 \leftarrow P_{12} \cdot x \times P_{11} \cdot y - P_{11} \cdot x \times P_{12} \cdot y$
$A2 \leftarrow P_{22} \cdot y - P_{21} \cdot y$
$B2 \leftarrow P_{21} \cdot x - P_{22} \cdot x$
$C2 \leftarrow P_{22} \cdot x \times P_{21} \cdot y - P_{21} \cdot x \times P_{22} \cdot y$
$x0 \leftarrow -\frac{B2 \times C1 - B1 \times C2}{A1 \times B2 - A2 \times B1}$
$y0 \leftarrow -\frac{A2 \times C1 - A1 \times C2}{A2 \times B1 - A1 \times B2}$
return point(x0, y0)

3-4 Opening reconstruction

In this opening reconstruction step, BLOCK entities will be read from the DXF file for calculating the Opening Equivalent Lines (OEL), which is supposed to indicate the right location and orientation of the opening symbol for later use in the contour reconstruction phase. According to the rules that are set up for openings in the redrawing phase, the opening reconstruction algorithm must fulfill the following characters:

(1) It should be able to deal with primitives of types that are allowed to be used in the opening symbols, including LINE, POLYLINE, LWPOLYLINE and ARC.

(2) It should be able to deal with cases of combination of openings that is illustrated in Figure 2-13 and Figure 3-6. In these cases, each opening symbols in the block definition can either be drawn directly by primitives, or inserted as an INSERT entity with the type of BLOCK. Thus, besides basic primitives LINE, POLYLINE, LWPOLYLINE and ARC, INSERT entities should also be considered.

(3) Transformation between different coordinate reference systems should be applied. This is because besides the global coordinate system of a floor plan, each BLOCK entity has its own coordinate system, within which all its primitives are defined. After the bounding box of the BLOCK has been calculated in its local coordinate system, it needs to be transformed to the global coordinate system by applying translation, rotation and scaling to it. In case of BLOCK containing BLCOK, the coordinates might be transformed multiple times until the global coordinate system of the floor plan has been reached.

(4) The second and the third characters both require the algorithm to be recursive. Only if the algorithm is recursive, the primitives defined in a son BLOCK contained by a father BLOCK can be reached to calculate its bounding box. Then, the bounding box of the son BLOCK then will be recursively transformed back to the coordinate system of its father BLOCK until the global coordinate system has been reached.

(5) It should be able to give an estimation of the OEL when symbols of swing doors and casement windows are encountered. The location indicated by the bounding boxes directly calculated from these symbols is a little bit shifted from the real location. Because the extra primitives in the blocks representing door panels, window sashes and swinging trajectories stretch the bounding box. Thus, the bounding box needs to be shrunk to its real size for these symbols.

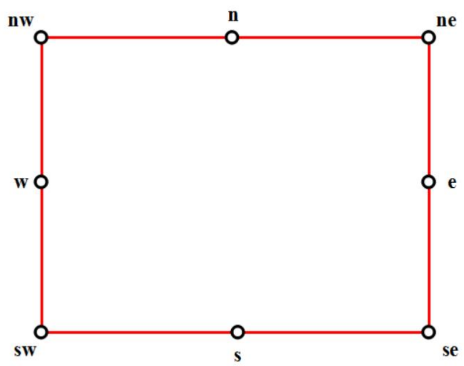


Figure 3-14: Feature points of a bounding box

An algorithm has been developed to address the problems mentioned above, the pseudocode of which is given in Algorithm CalcOpeningEquibalentLine. As shown in Fig. 3-14, there are eight feature points for each bounding box. In addition to the four corner points that are denoted respectively by *nw,ne,sw,se*, the centers of line $\overline{nw, sw}$,$\overline{sw, se}$,$\overline{nw, sw}$,$\overline{ne, se}$ are also included, denoted by *n,s,w,e* respectively. The calculation of the OELs is based on the assumption that the main direction of an opening is along the longer side of its bounding box. In addition, in order to make sure calculated OEL intersect with its adjacent wall line segments, it should be extended on its both sides with a threshold *d*. Thus, if $\overline{w, e}$ is longer than $\overline{n, s}$, the OEL of this opening is the extension of $\overline{w, e}$; otherwise, it is the extension of $\overline{n, s}$. According to the type of this opening, i.e. window or door, it will be instantiated with its thickness and width, information that will be needed in the database for later 3D reconstruction.

Algorithm 6 CalcOpeningEquibalentLine

Input: blocks: all BLOCKs in the DXF file
Input: window_layer: the layer containing window blocks
Input: door_layer: the layer containing door blocks
Input: r: the average thickness of openings
Input: d: length to extend OEL on its both sides
Output: windows: group for storing WINDOW objects
Output: doors: group for storing DOOR objects

$windows \leftarrow empty$
$doors \leftarrow empty$
for each $block_i \in block$ **do**
  $parameters0 \leftarrow$ parameters of translating, rotating and scaling of $block_i$
  $BBOX \leftarrow BBlock(block_i, parameters0, r)$
   $n, s, w, e \leftarrow BBOX$
  if $\overline{w, e}$ is longer than $\overline{n, s}$ **then**
    $OEL \leftarrow$ extend $\overline{w, e}$ with d
    $thickness \leftarrow$ length of $\overline{n, s}$
    $width \leftarrow$ length of $\overline{w, e}$
  else
    $OEL \leftarrow$ extend $\overline{n, s}$ with d
    $thickness \leftarrow$ length of $\overline{w, e}$
    $width \leftarrow$ length of $\overline{n, s}$
  end if
  if $block_i$ is in window_layer **then**
    make an instance of WINDOW object with OEL, thickness, width
    add this instance into windows
  else if $block_i$ is in door_layer **then**
    make an instance of DOOR object with OEL, thickness, width
    add this instance into doors
  end if
end for
return windows, doors

The calculation of the bounding box for openings is actually realized in Function BlockBBox, the pseudocode of which is given below. The algorithm reads an entity in the BLOCK each time. If it is of type of LINE, POLYLINE or LWPOLYLINE, the x and y coordinates of its every endpoints are read to update the bounding box (e.g. XMIN, XMAX, YMIN and YMAX); if it is an ARC entity, then its center will be stored into group W for later shrinking the bounding box; if it is of type of INSERT, which means it is a BLOCK entity, this entity will be taken into Function BlockBBox with its transformation parameters to recursively calculate its bounding box. The centers of ARCs in the son BLOCK entity are also returned together with its bounding box (Line 14) and added to W. After all the entities have been reviewed, the bounding box is made (Line 20). If there exits any ARC entity in this BLOCK or its son BLOCK, Function ShrinkBBox will be called to shrink the bounding box to an estimation of its real size based on the location of the centers (Line 22). At last, the bounding box as well as the centers of ARCs will be transformed to the coordinate system of the upper level and returned (Line 24 - 29).

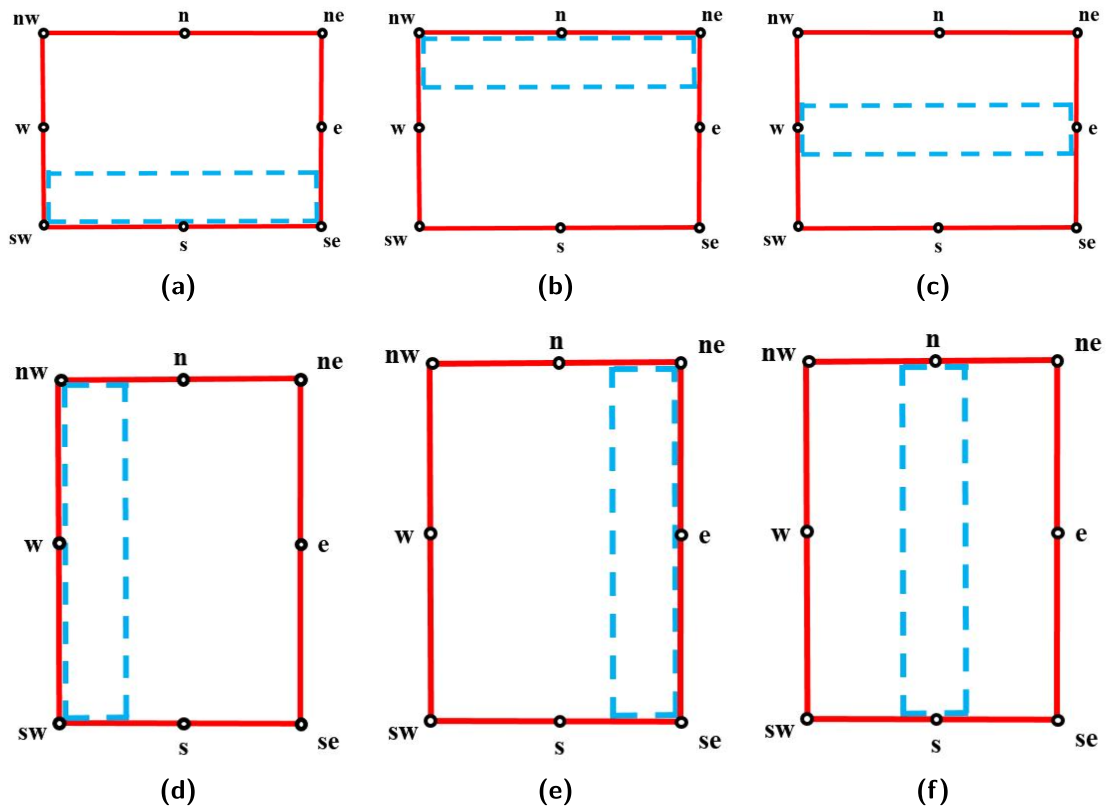


Figure 3-15: Calculation of minimized bounding box

Algorithm 7 Function: BlockBBox

Input: block: a block entity of opening read from DXF file
Input: parameters0: parameters of translating, rotating and scaling
Input: r: the average thickness of openings
Output: BBOX: bounding box of block
Output: CC: group for storing center of ARCs

$XMIN \leftarrow inf$
$XMAN \leftarrow -inf$
$YMIN \leftarrow inf$
$YMAN \leftarrow -inf$
$C \leftarrow empty$
for each $entity_i \in block$ **do**
    if $entity_i$ is of any type in {LINE, POLYLINE, LWPOLYLINE} **then**
        $xmin, xmax, ymin, ymax \leftarrow$ endpoints of $entity_i$
        updeate XMIN, XMAX, YMIN, YMAX with xmin, xmax, ymin, ymax
    else if $entity_i$ is of type of ARC **then**
        $C \leftarrow$ the center of ARC
    else if $entity_i$ is of type of INSERT **then**
        $parameters \leftarrow$ parameters of translating, rotating and scaling of $entity_i$
        $bbox0, C0 \leftarrow$ BlockBBox($entity_i$,parameters, r)
        $C \leftarrow C0$
        $xmin, xmax, ymin, ymax \leftarrow bbox0$
        updeate XMIN, XMAX, YMIN, YMAX with xmin, xmax, ymin, ymax
    end if
end for
$bbox \leftarrow$ [Point(XMIN, YMIN), Point(XMAX, YMIN), Point(XMAX, YMAX), Point(XMIN, YMAX)]
if C is not empty **then**
    $bbox \leftarrow ShrinkBBox(bbox, C, r)$
end if
$CC \leftarrow empty$
for each $c_i \in C$ **do**
    $CC \leftarrow$ CoordTransformation(c_i, parameters0)
end for
$BBOX \leftarrow CoordTransformation(bbox, parameters0)$
return BBOX, CC

To be more specific about how to shrink the bounding box, first three base lines should be established once its main direction is determined. These base lines actually indicate three possible locations of the opening frame. Based on the observation that the shafts of windows and doors are always attached to the frame, the real location of the frame can be estimated by checking the distance from the base lines to the points representing the shafts, which are indicated by the centers of ARC entities in the block definition. In addition, if there are multiple ARC entities in a BLOCK, only one of them needs to be checked. This is because even in case of the combination of openings shown in Fig. 2-13, all the arc centers are closed to a same base line since the multiple door panels share a common frame that they are attached to. Let $x_min, x_max, y_min, y_madxx$ be respectively the minimal and maximal x and y coordinates. Let dx be the width of the bounding box and dy the height of the bounding box. $dx = x_max - x_min, dy = y_max - y_min$. If dx is longer than dy, the main direction is west-east and the three base lines are $\overline{nw, ne}$,$\overline{sw, se}$,$\overline{w, e}$; if dx is shorter than dy, the main direction is west-east and the three base lines are $\overline{nw, sw}$,$\overline{ne, se}$,$\overline{n, s}$. For an opening block in which there exists any ARC entity, given the average thickness of openings r, the bounding box is shrunk based on the location of the centers of ARCs with respect to the base lines in the following way: under the circumstance that the main direction is west-east, if the center is closer to $\overline{sw, se}$ then y_{max} is adjusted to $y_{min} + r$ (Fig. 3-15a); if the center is closer to $\overline{nw, ne}$ then y_{min} is adjusted to $y_{max} - r$ (Fig. 3-15b); if the center is closer to $\overline{w, e}$ then y_{min} and y_{max} is adjusted to $fracy_{min} + y_{max} - r2$ and $fracy_{min} + y_{max} + r2$ (Fig. 3-15c); under the circumstance that the main direction is north-south, if the center is closer to $\overline{nw, sw}$ then x_{max} is adjusted to $x_{min} + r$ (Fig. 3-15d); if the center is closer to $\overline{ne, se}$ then x_{min} is adjusted to $x_{max} - r$ (Fig. 3-15e); if the center is closer to $\overline{n, s}$ then x_{min} and x_{max} is adjusted to $fracx_{min} + x_{max} - r2$ and $fracx_{min} + x_{max} + r2$ (Fig. 3-15f); If there is no ARC entity in a block, the bounding box is assumed to already represent the correct boundary of the opening and no shrinking needs to be performed. The pseudocode of this part is given in Function ShrinkBBox.

Algorithm 8 Function: CoordTransformation

Input: P0: point to be transformed
Input: angle: rotating angle
Input: dX: translation on X
Input: dY: translation on Y
Input: xs: scaling factor on X
Input: ys: scaling factor on Y
Output: Pt: point after transforming

1: $X \leftarrow xs \times P0 \cdot x \times cos(angle) - ys \times P0 \cdot y \times sin(angle) + dX$
2: $Y \leftarrow xs \times P0 \cdot x \times sin(angle) + ys \times P0 \cdot y \times cos(angle) + dY$
3: **return** Pt(X,Y)

Algorithm 9 Function: ShrinkBBox

Input: bbox: bounding box of opening
Input: C: group of center of ARCs
Input: r: the average thickness of openings
Output: BBOX: new bounding box after shrinking

$nw, ne, sw, se, n, s, w, e \leftarrow bbox$
$x_m in, x_m ax, y_m in, y_m ax \leftarrow bbox$
$dx \leftarrow x_{max} - x_{min}$
$dy \leftarrow y_{max} - y_{min}$
$c_0 \leftarrow$ the first center of ARC in C
if $dx \geq dy$ **then**
    $d1 \leftarrow$ distance from c_0 to $\overline{sw, se}$
    $d2 \leftarrow$ distance from c_0 to $\overline{nw, ne}$
    $d3 \leftarrow$ distance from c_0 to $\overline{w, e}$
    if $d1 = MIN(d1, d2, d3)$ **then**
        $y_{max} \leftarrow y_{min} + r$
    else if $d2 = MIN(d1, d2, d3)$ **then**
        $y_{min} \leftarrow y_{max} - r$
    else
        $y_{min} \leftarrow \frac{y_{min}+y_{max}-r}{2}$
        $y_{max} \leftarrow \frac{y_{min}+y_{max}+r}{2}$
    end if
else
    $d1 \leftarrow$ distance from c_0 to $\overline{nw, sw}$
    $d2 \leftarrow$ distance from c_0 to $\overline{ne, se}$
    $d3 \leftarrow$ distance from c_0 to $\overline{n, s}$
    if $d1 = MIN(d1, d2, d3)$ **then**
        $x_{max} \leftarrow x_{min} + r$
    else if $d2 = MIN(d1, d2, d3)$ **then**
        $x_{min} \leftarrow x_{max} - r$
    else
        $x_{min} \leftarrow \frac{x_{min}+x_{max}-r}{2}$
        $x_{max} \leftarrow \frac{x_{min}+x_{max}+r}{2}$
    end if
end if
$BBOX \leftarrow [Point(x_{min}, y_{min}), Point(x_{max}, y_{min}), Point(x_{max}, y_{max}), Point(x_{min}, y_{max})]$
return BBOX

3-5 Contour reconstruction

In this thesis, the contour reconstruction algorithm is developed based on the extensive study of the layout between openings and their adjacent walls. A classification of the opening-wall-layouts needs to be first presented before introducing the specific algorithm.

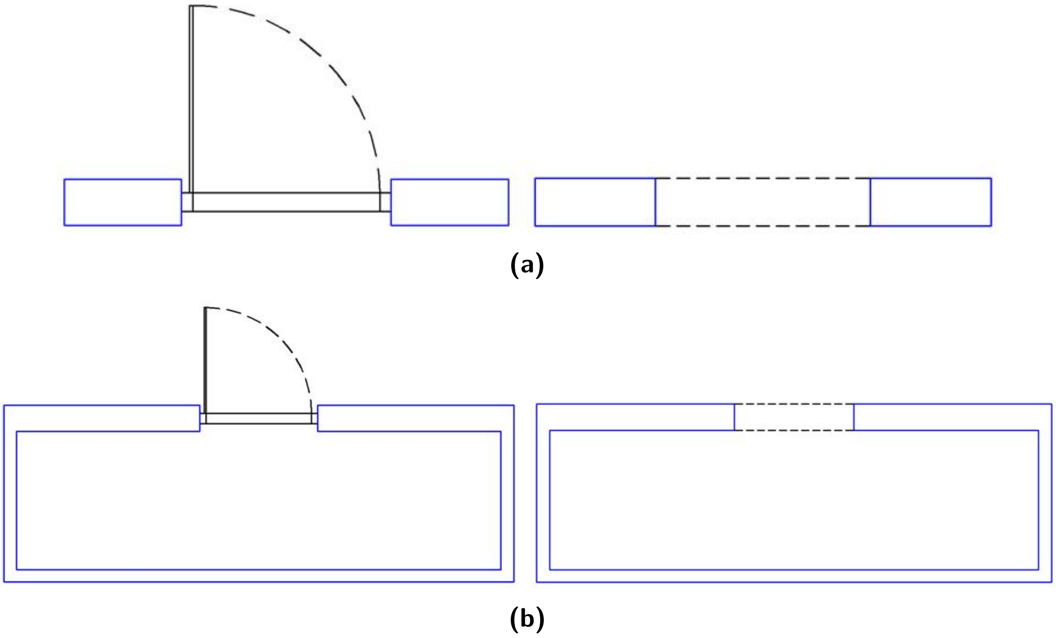


Figure 3-16: Two types of openings based on whether a space is closured by the opening: Connector Opening; (b) Closure Opening.

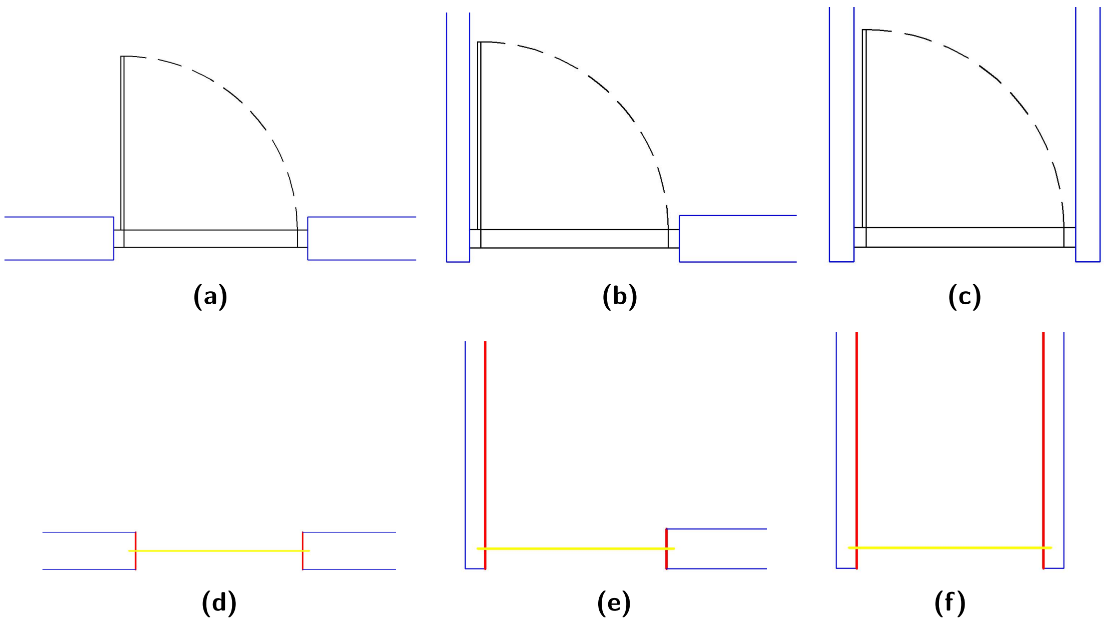


Figure 3-17: Three main layouts between an opening and its adjacent walls

For each opening, it has two adjacent walls on its both sides. Thus, for each OEL, it intersects with two wall line segments. Based on whether an opening closures a space, the openings can be divided into two types. If an opening is just connecting two different POLYGONs (Figure 3-15 (a) right), it is called Connector Opening; if an opening is connecting a POLYGON with itself making the space surrounded by this POLYGON closured (Figure 3-15 (b) right), then it is called Closure Opening. In the case of Connector Openings, after the OEL is created, the two POLYONs will be merged into a new POLYGON (Figure 3-15 (a) left); while in this case of Closure Openings, after the OEL is created, the previous POLYGON will be made into two new POLYGONs, with one containing the other (Figure 3-15 (b) left).

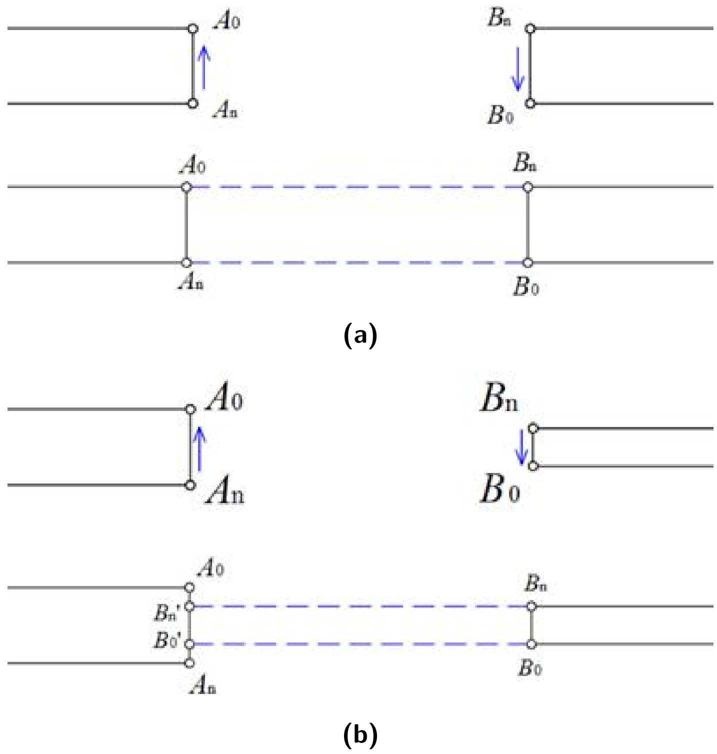


Figure 3-18: Two cases of the first layout

In addition to the types of openings, there are three main layouts between an opening and its adjacent walls as shown in Figure 3-16: (1) the adjacent walls are parallel and in a same line (Figure 3-16 (a)); (2) the adjacent walls are perpendicular (Figure 3-16 (b)); (3) the adjacent walls are parallel but not in a same line (Figure 3-16 (c)). Given a value δ indicating the average wall thickness in the floor plan, the characters of each layout can be summarized as follows: in the first layout, the two line segments intersected by OEL are both shorter than or equal to δ (Figure 3-16 (d)); in the second layout, one of the line segments intersected by OEL is shorter than or equal to δ, while the other one is longer than δ (Figure 3-16 (e)); in the third layout, the two line segments intersected by OEL are both longer than δ (Figure 3-16 (f)). In Figure 3-16 (d) (e) (f), the yellow line segments are the OELs, while the red line segments are the wall line segments intersected by OEL.

To be more specific, based on the difference of the length of the two line segments intersected by OEL, the first layout can be further divided into two cases. Given a ratio τ, let the

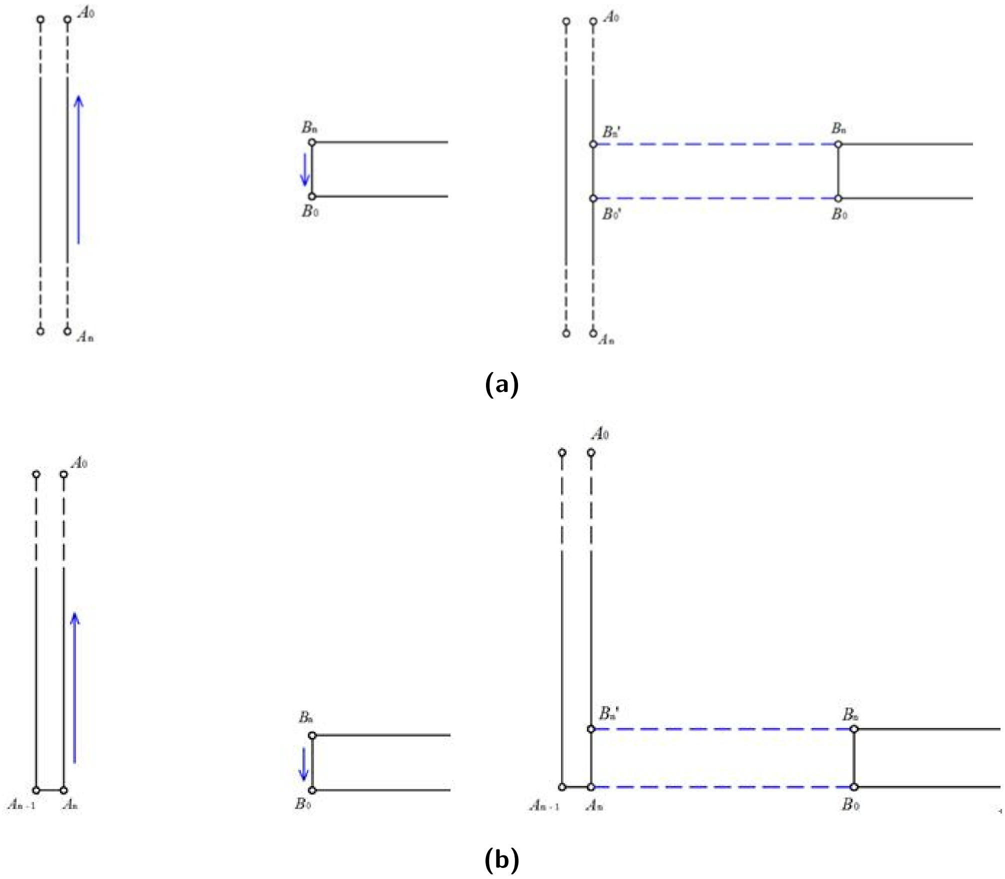


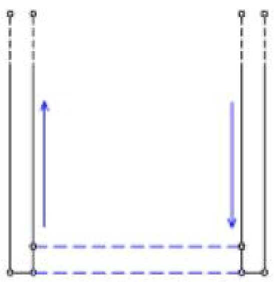
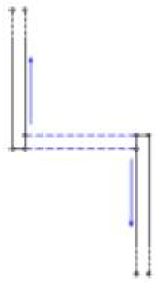
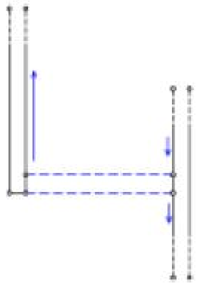
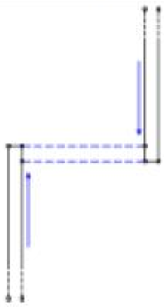
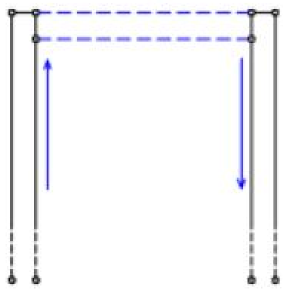
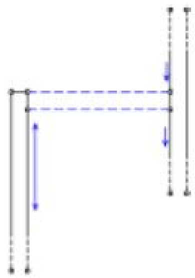
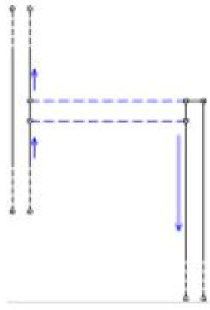
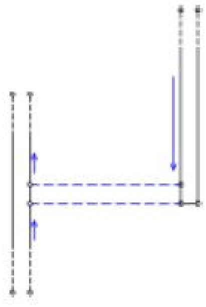
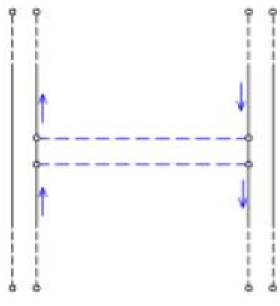
Figure 3-19: Two cases of the second layout

shorter line segment be S, and the longer one L. If $|length(S) - length(L)| \leq \tau \times length(S)$, then the endpoints of L and S are directly connected to merge the POLYGONs; else, the endpoints of S need to be first projected to L. To merge the POLYGONs, first they have to be made into anti-clockwise starting from the end of the intersected line segment. For the first case as shown in Fig. 3-18a, let $\{A_0, A_1, \ldots, A_{n-1}, A_n\}$ be the POLYGON that S belongs to, and $\{B_0, B_1, \ldots, B_{m-1}, B_m\}$ the POLYGON that L belongs to. The merged POLYGON is $\{A_0, A_1, \ldots, A_{n-1}, A_n\} + \{B_0, B_1, \ldots, B_{m-1}, B_m\}$; for the second case as shown in Fig. 3-18b, additionally let B'_0 and B'_n be the projections of B_0 and B_m respectively. The merged POLYGON is $\{A_0, A_1, \ldots, A_{n-1}, A_n\} + B'_0 + \{B_0, B_1, \ldots, B_{m-1}, B_m\}$. Both of these two cases can be called I-shape.

There are also two cases of the second layout, which is shown in Fig. 3-19. In the first case which is called T-shape, the projections of the endpoints of the shorter line segment are both IN the longer line segment; in the second case which is called L-shape, one of the projection of the endpoints of the shorter line segment coincides with one the endpoints of the longer line segment. Similarly, the two POLYGONs are made into anti-clockwise with the first point being the end of the intersected line segment. Let $\{A_0, A_1, \ldots, A_{n-1}, A_n\}$ be the POLYGON that L belongs to, and $\{B_0, B_1, \ldots, B_{m-1}, B_m\}$ the POLYGON that S belongs to. Let B'_0 and B'_n be the projections of B_0 and B_m respectively. For the first case as shown in Fig. 3-19a, the merged POLYGON is $\{A_0, A_1, \ldots, A_{n-1}, A_n\} + B'_0 + \{B_0, B_1, \ldots, B_{m-1}, B_m\} + B'_n$; for the second case as shown in Fig. 3-19b, B_0 coincides with A_n. The merged POLYGON is $\{A_0, A_1, \ldots, A_{n-1}, A_n\} + \{B_0, B_1, \ldots, B_{m-1}, B_m\} + B'_n$.

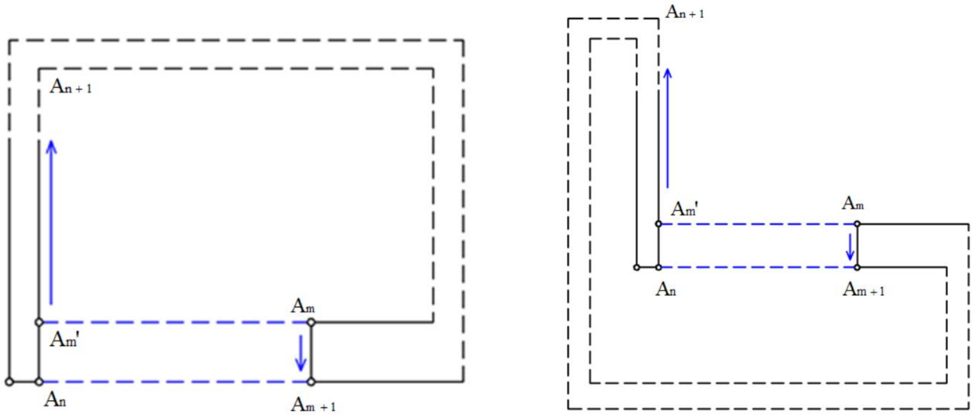
The third layout is more complicated since for both the shorter and longer line segments, there are three possible locations of the intersecting point with the OEL on each of them. Thus, the total number of the possible cases for the third layout is nine, which are listed in Table 3-2. These nine cases can be generalized into four kinds of shapes: U-shape, Z-shape, h-shape and H-shape. They are all deal with in a similar way as the other shapes.

Table 3-2: Different cases of the third layout

Location of the intersecting point	Near start of L	Near end of L	In the middle of L
Near start of S	 U-shape	 Z-shape	 h-shape
Near end of S	 Z-shape	 U-shape	 Z-shape
In the middle of S	 Z-shape	 h-shape	 H-shape

The method described above is under the circumstance that the opening is a Connector Opening. In case of Closure Openings, in addition to making the adjacent POLYGONs anti-clockwise, points belonging to the outer and the inner rings need to be separated. Similar with cases of Connector Openings, there also exist all the shapes mentioned above. The only difference is that for a certain point it is not sure if it is in the outer ring or the inner ring.

Taking L-shape as an example, Fig. 3-20 shows two possible scenarios for a self-closed L-shape. Let the POLYGON be $\{A_0, A_1, \ldots, A_{i-1}, A_i\}$. The longer line segment is $\overline{A_n, A_{n+1}}$ and the shorter one is $\overline{A_m, A_{m+1}}$. A'_m is the projection of A_m on $\overline{A_n, A_{n+1}}$. After the contour is reconstructed, the POLYGON is separated into two CHAINs: $\{A_{m+1}, A_{m+2}, \ldots, A_{n-1}, A_n\}$ and $\{A_{n+1}, A_{n+2}, \ldots, A_{m-1}, A_m, A'_m\}$. In case of (a), $\{A_{m+1}, A_{m+2}, \ldots, A_{n-1}, A_n\}$ is the outer ring, while $\{A_{n+1}, A_{n+2}, \ldots, A_{m-1}, A_m, A'_m\}$ is the inner ring. However, it is the opposite situation in case of (b). Thus, in addition to breaking a POLYGON into two CHAINs, it also has to be determined which one is the outer CHAIN and which one is the inner CHAIN. This is achieved by calculating the area of the space bounded by the CHAIN. The area of the outer CHAIN is always bigger than the area of the inner one. The pseudocode of this part is given in Function SeparateOutandIn.



(a) CONNECTED **(b)** UNCONNECTED

Figure 3-20: Two possible cases for a T-shape when the opening is Closure Opening

Algorithm 10 Function: SeparateOutandIn

Input: I1: index of the start point of one of the intersecting line segments in the POLYGON
Input: I2: index of the start point of the other intersecting line segments in the POLYGON
Input: P: the POLYGON to be separated
Output: outRing: the outer CHAIN P is divided into
Output: inRing: the inner CHAIN P is divided into

```
if I1 > I2 then
    b ← I1
    s ← I2
else
    b ← I2
    s ← I1
end if
if b + 1 < len(P) then
    a1 ← P[s + 1, b + 1]
    a2 ← P[b +1:] + P[0: s + 1]
else
    a1 ← P[b + 1 ÍC len(P), s + 1]
    a2 ← P[s +1:] + P[0: b + 1 ÍC len(P) ]
end if
S1 ← calculate the area bounded by a1
S2 ← calculate the area bounded by a2
if S1 > S2 then
    outRing ← a1
    inRing ← a2
else
    outRing ← a2
    inRing ← a1
end if
return outRing, inRing
```

Chapter 4

Implementation and testing

This chapter presents the results of each step in the proposed process with responding analysis. The floor plans from three different buildings are tested. Among them, there are floor plans that are originally drawn by CAD software, digitized floor plans and image floor plans. Besides, the structure and complexity of these three buildings are also different with each other.

4-1 Tested buildings

The algorithms introduced above are implemented and tested on three buildings floor plans. Among them, the floor plans of building "EB_alle_niveaus" and "Binnenvest" are offered by company MoreForYou. The other one is the floor plan of the ground floor of the architecture faculty of TU Delft. The specific case of each of them is explained below.

Building 1: Architecture faculty of TU Delft

Floor plan of this building is not originally drawn in CAD software but the scanned copy of paper floor plan and saved in CAD format. Although it has been post-processed by architects in CAD software and related information has generally segmented into several layers, certain degrees of geometric distortion exist in the floor plan. This means some line segments are not correctly connected, or perfectly parallel or perpendicular. More drafting errors might be contained in it than floor plans that are originally created by CAD software. In addition, not all windows and doors are properly saved as block entities, and some are still left as primitives. In reality, this building has three floors and the layout of each floor is almost the same. Thus, this thesis only processes the floor plan of its ground floor and the final building model is generated by extruding the extracted contours three times. The original floor plan of this building is shown in Fig. 4-1.

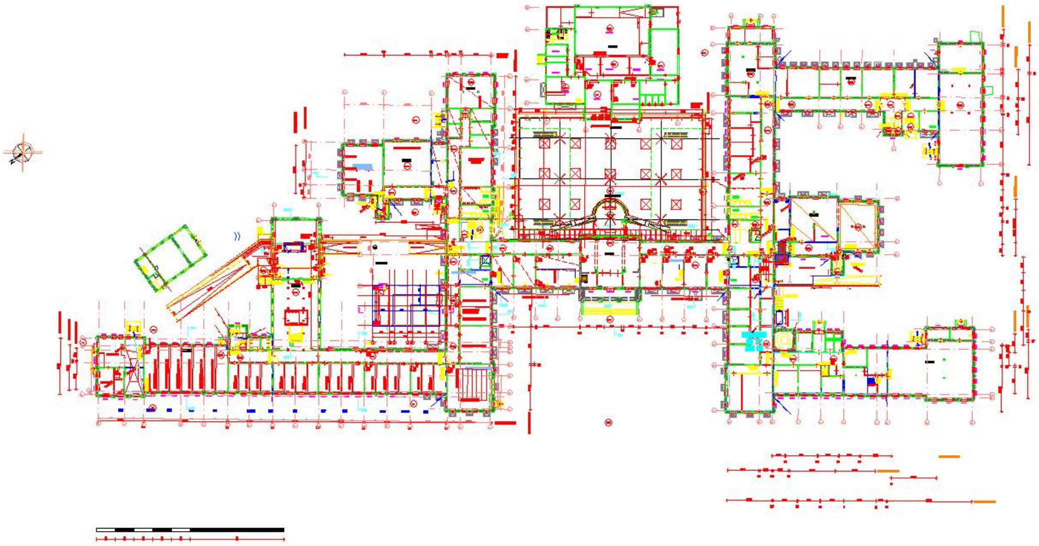


Figure 4-1: Ground floor of architecture faculty of TU Delft

Building 2: EB_alle_niveaus

This building has four floors. The layouts of the ground, the first and the second floor are all different from each other. The third floor is eaxactly the same as the second floor. Since the situation of duplicated floors will already be tested by the first building, here for this building only the floor plans of its first three floors will be processed. Since all the floor plans of this building are originally drawn by CAD software, the conditions in terms of geometric correctness and information segmentation of this building are better than the other two tested building. However, this building also is the most complicated one with the largest amount of line segments and blocks. The original floor plans of this building are shown in Fig. 4-2.

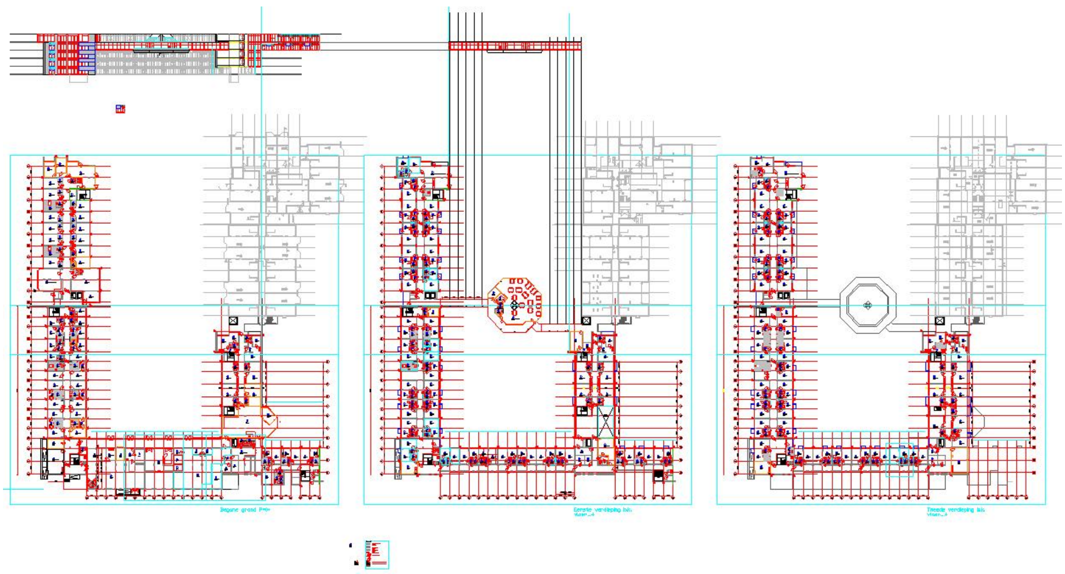
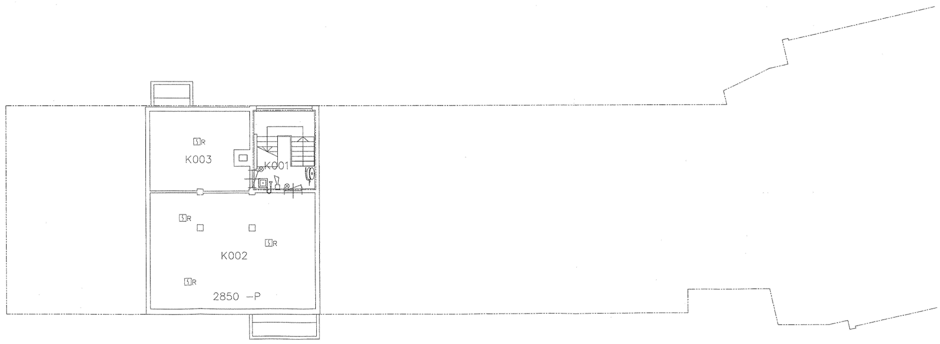


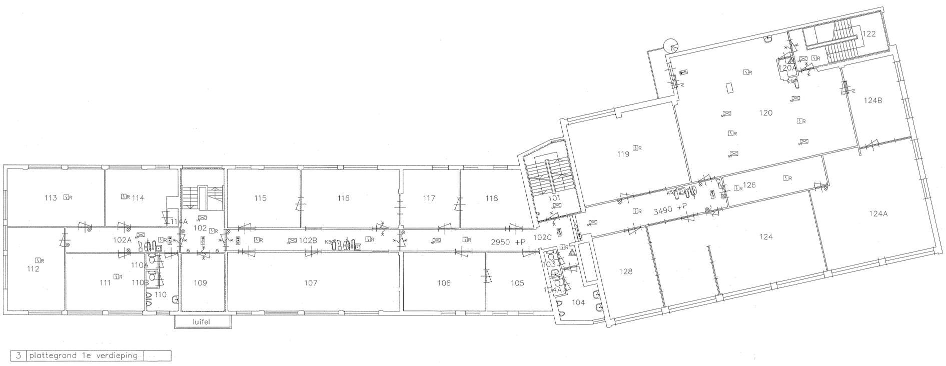
Figure 4-2: Building EB_alle_niveaus

Building 3: Binnenvest

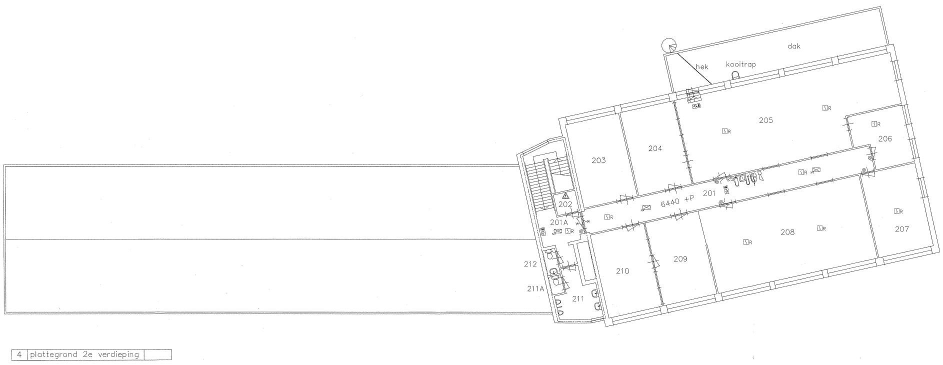
The retrieved floor plans of this building are saved as raster images, which means the floor plans of this building have to be completely redrawn based on the images according to the proposed rules in this thesis. This will also result in certain inaccuracy like the first building. Besides, this building has a small basement and its first floor only occupies a part of the area of its ground floor. The original floor plans of this building are shown in Fig. 4-3.



(a) Basement



(b) Ground floor



(c) First floor

Figure 4-3: Building Binnenvest

4-2 Redrawing

These floor plans are first redrawn according to the rules proposed in Chapeter 3. Fig. 4-4 shows the floor plans after redrawing. The number of line segments in the wall layer, the number of windows and doors of them are shown in Table 4-1. As for the time efficiency of the proposed redrawing rules, according to Bart Kroesbergen from company MoreForYou, it only took a professional architect from his team about two hours to redraw the whole set of floor plans of building EB_alle_niveaus completely from the original paper floor plans, even including the objects and information that are not to be used. He said compared to the time to manually make a 3D model of such a building, the time to redraw the floor plans is much less.

Table 4-1: Statistics of entities in floor plans

Building	Line segments	Window blocks	Door blocks
Architecture Faculty of TU Delft	2239	297	151
EB_alle_niveaus	7784	488	756
Binnenvest	541	40	12

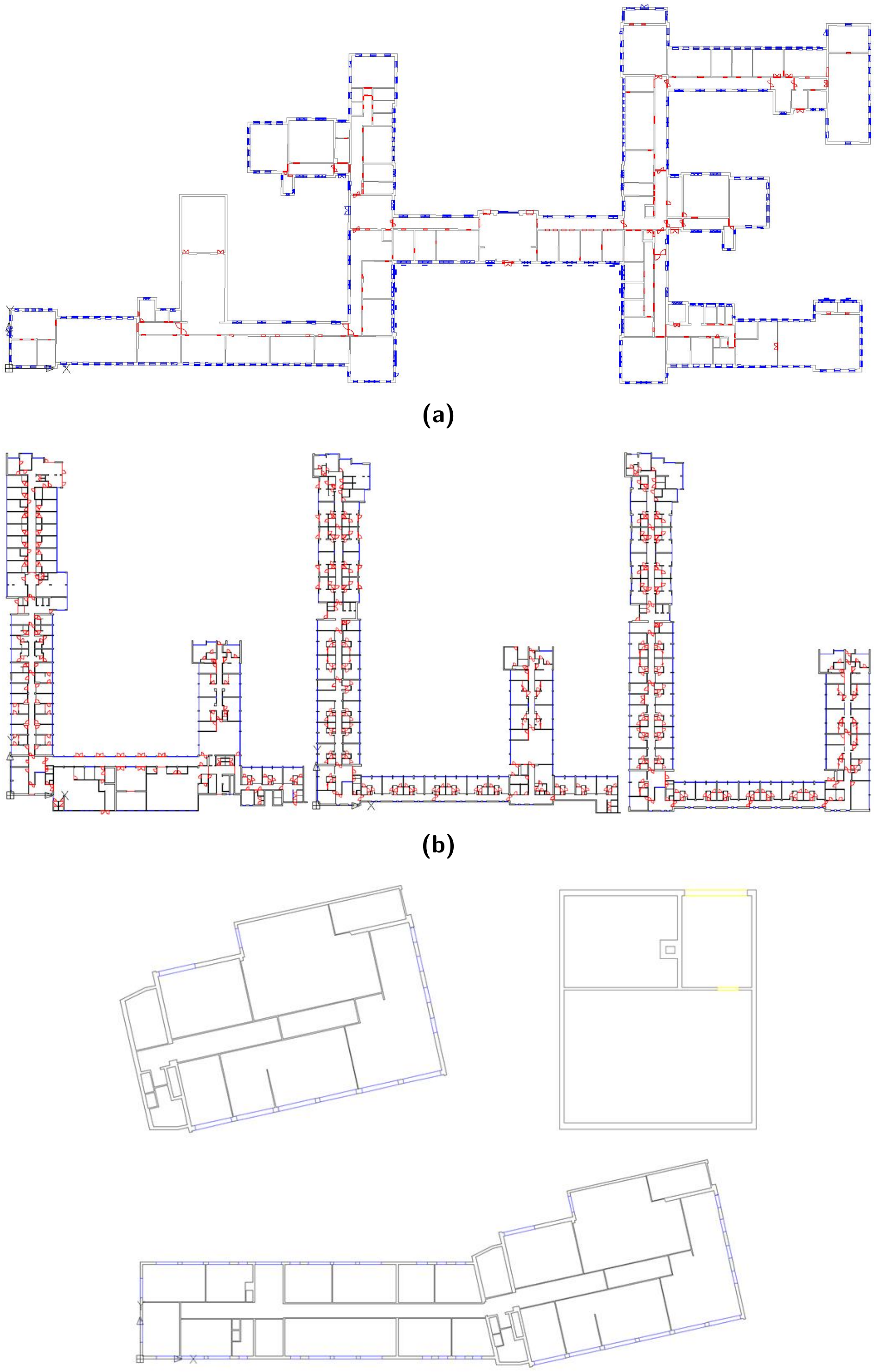
4-3 Drafting error fixing

Table 4-2 shows the results of fixing the drafting errors. Here, the threshold ε and δ for determining null-length line segments and duplicated line segments are both set to be 5 mm. This is because in real-life indoor environment, any distance smaller than 5 mm can regarded trivial and thus be discarded.

Table 4-2: Results of fixing drafting errors

Building	Null-length	Overlapped or Consecutive	Containing	Contained
Architecture Faculty of TU Delft	5	24	3	25
EB_alle_niveaus	1	4	0	3
Binnenvest	0	0	0	0

From the table we can see that the floor plan of architecture faculty contains more errors than the other floor plans. This is because in the original version of this floor plan, the representation of walls is already conformed to the proposed rules. Thus, in the redrawing phase, it is just storing the line segments of walls into a separated layer without checking and modifying them. Thus, this floor plan inevitably suffered from more drafting errors. For the other floor plans, the walls have been redrawn carefully according to the proposed rules. Thus, they contain less drafting errors.



(a)

(b)

(c)

Figure 4-4: Redrawn floor plans

4-4 Grouping

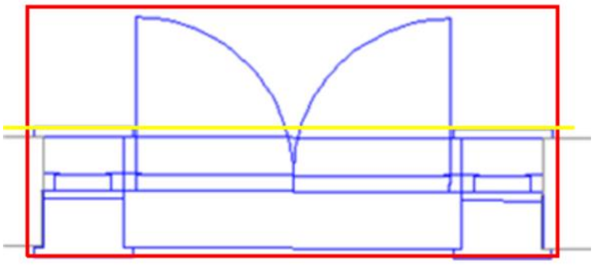
Table 4-3 shows the results of fixing the drafting errors. Here, the threshold d for searching disjoint vertices are set to be 5 mm. each time the algorithm repeats, the searching distance will be increased with a d. From the table, it can be seen the line segments in floor plan of architecture faculty took six times to be fully grouped into closed chains. While the floor plan of the second floor of building EB_alle_niveaus took the less times, indicating that the geometry in this floor plan is better than the others.

Table 4-3: Results of line grouping

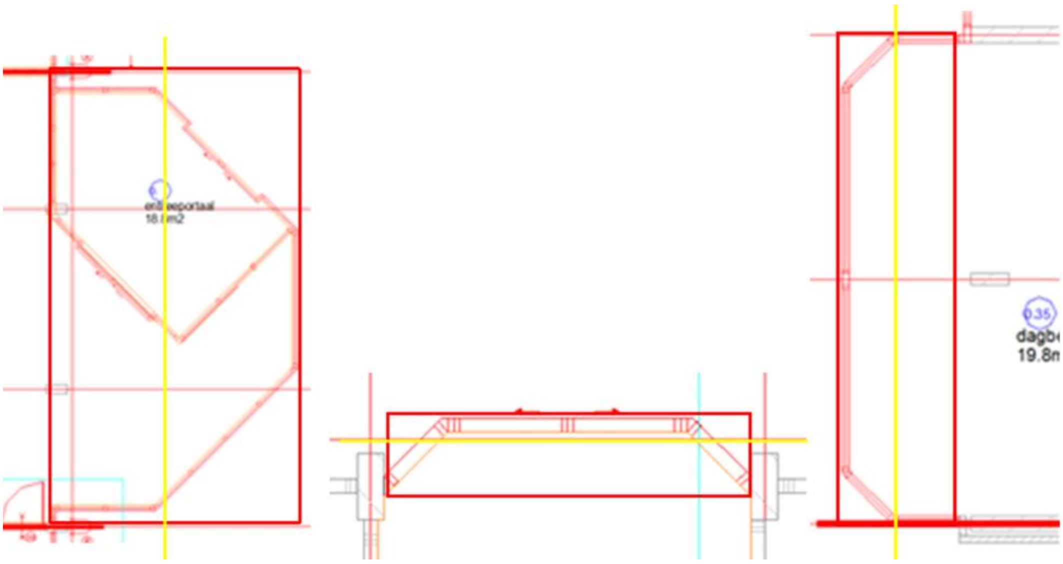
Floor plans		1	2	3	4	5	6
Input lines	BK	2188	920	376	64	38	38
Closed chains	TU	211	286	310	311	311	312
Remaining lines	Delft	920	376	64	38	38	0
Input lines	BE	1809	484	174	76	/	/
Closed chains	ground	134	147	150	152	/	/
Remaining lines	floor	484	174	76	0	/	/
Input lines	BE	2021	624	368	172	46	/
Closed chains	first	167	183	190	194	195	/
Remaining lines	floor	624	368	172	46	0	/
Input lines	BE	1909	496	42	/	/	/
Closed chains	second	180	206	208	/	/	/
Remaining lines	floor	496	42	0	/	/	/

4-5 Reconstruction of openings and contours

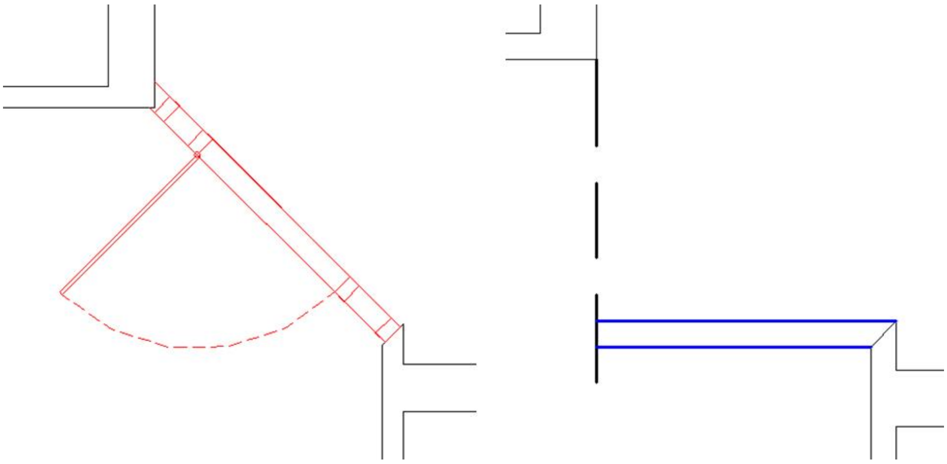
Table 4-4 shows the results of opening reconstruction. In the table, σ is the factor used in determining the main direction of bounding box. If the width of a bounding box is larger than or equal to the value of its height multiplied by σ, the main direction is assumed to be along the width when the width and height are very close; else, it is assumed to be along the height. The role of σ is to make the direction of the width has higher possibility to be the main direction since in most cases the openings are defined horizontally in its local coordinate system. Thus, for the three floor plans of building EB_alle_niveaus, the ratio of failed openings decreases as σ decrease. This is because those openings in these floor plans failed to find its two adjacent wall line segments are mostly caused by that they are vertically defined. However, for the first floor plan of, the situation is opposite. This is because those failed openings in this floor plan are not caused by the direction. After checking the created OEL, the problem is found to be like what is shown in Fig. 4-5a. The OELs of some openings (the yellow line segment in Fig. 4-5a) are mistakenly chosen as the center line of its bounding box while the correct OEL should be the bottom line. This caused the OEL shifted and not intersecting with its adjacent wall lines.



(a)



(b)



(c)

Figure 4-5: Openings cannot be reconstructed

In addition to that, Fig. 4-5b and Fig. 4-5c shows two other kinds of openings that cannot be reconstructed. Fig. 4-5b shows three cases of bay windows and glass walls found in the test floor plans. They are all combination of several consecutive single windows with turning

Table 4-4: Results of opening reconstruction

Floor plans		σ		
		0.8	0.9	1
Architecture TU Delft	failed windows	23	23	23
	ratio	7.74%	7.74%	7.74%
	failed doors	11	7	5
	ratio	7.28%	4.64%	3.31%
EB_alle_niveaus Ground Floor	failed windows	0	0	0
	ratio	0	0	0
	failed doors	0	0	0
	ratio	0	0	0
EB_alle_niveaus First Floor	failed windows	0	0	0
	ratio	0	0	0
	failed doors	1	1	12
	ratio	0.50%	0.50%	5.94%
EB_alle_niveaus Second Floor	failed windows	0	0	0
	ratio	0	0	0
	failed doors	0	0	11
	ratio	0	0	5.95%
Binnenvest	failed windows	0	0	0
	ratio	0	0	0
	failed doors	0	0	0
	ratio	0	0	

angles. Due to that in the method proposed in this thesis they are dealt with as a whole, the OELs calculated from the blocks of these openings are also shifted. Fig. 4-5c shows another case. In this case, the door intersects with a wall on its left side non-perpendicularly. However, in our algorithm, the line segments created to replace the opening are always perpendicular to the wall. Thus, the created line segments will be the blue lines in Fig. 4-5c.

Fig. 4-6 shows the contours reconstructed from the four test floor plans after some of those problematic openings on the building facades have been fixed manually to make sure the outer shell of the building can be enclosure.

4-6 Import data into database

For a building, the file names of floor plans of its every floor will be given by user. Along with the file names, the building name, the number of levels, the lowest and the highest level number will also be provided by user. Also, a record of a building relation will be generated in *table<relations>* with the provided information. Then the floor plans will be processed in order of the level (from the lowest to the highest) by the algorithms introduced in last chaper. After the contours have been reconstructed, all the contours along with the openings will be imported into database in a simplified form of IndoorOSM. In the database, four tables in total will be generated, which are *nodes, ways, relations and relation_member* respectively. The structure of these four tables is shown in Tables 4-5 to 4-8.

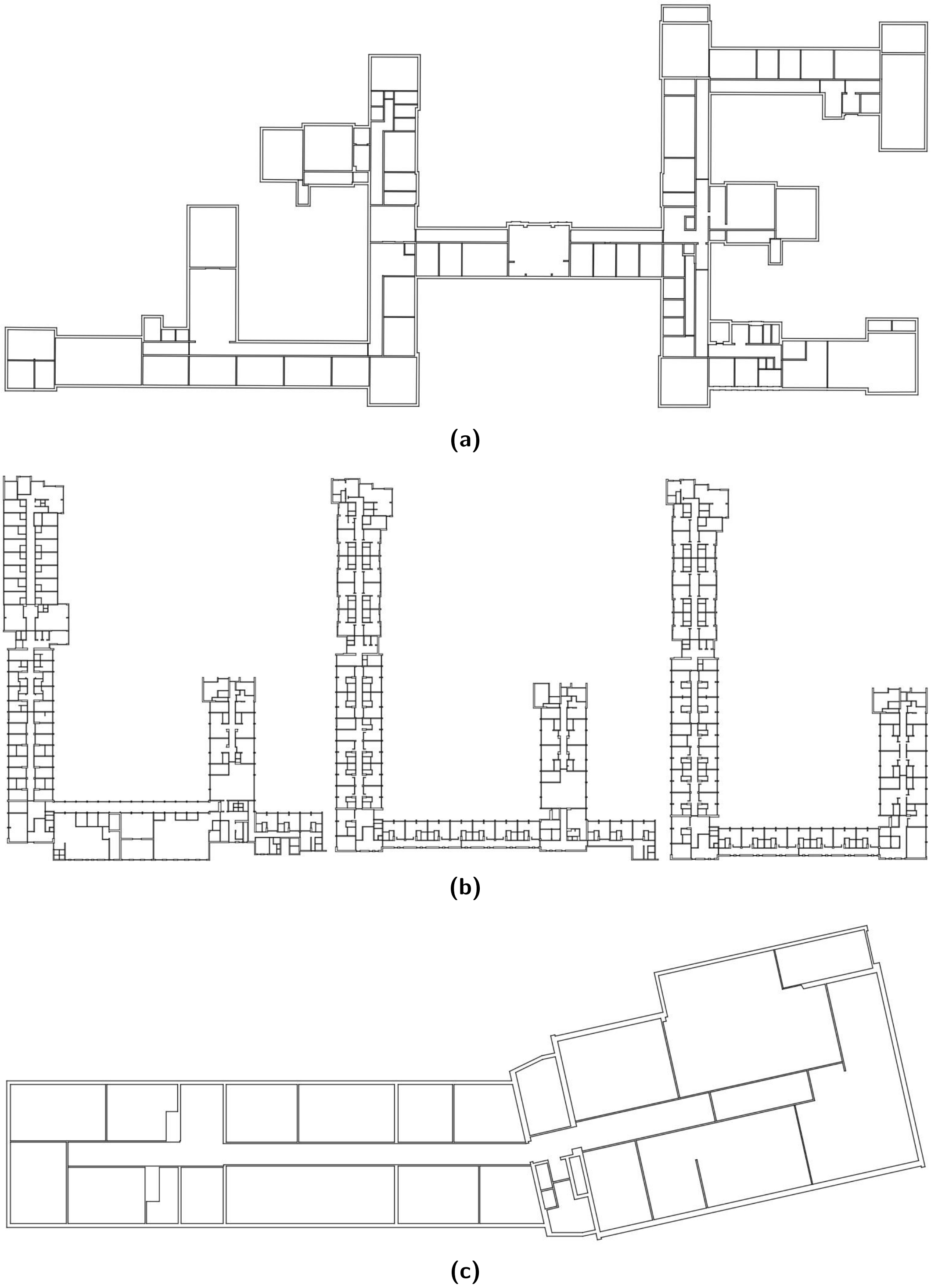


Figure 4-6: Reconstructed contours

Every time an opening is imported, a coresponding record will be generated in *table<nodes>*. There are three attributes in the table for each record: *"id"*, *"geom"* and *"tags"*. *"id"* is the identification number for this node, which will be generated automatically in order; *"geom"*

Table 4-5: nodes

Table<nodes>		
Attributes		Description
id		Identification number for this opening, automatically generated in order.
tags	window/door	"door=yes" when it is a door; "window=yes" when it is a window.
	width	width of the OEL.
	height	Provided by user.
	breast	Provided by user, only applied to windows.
	level	The level this opening belongs,to, automatically generated when the level of the floor plan is specified by,user.
geom		Central point of the OEL.

contains the central point of the OEL of this opening; *"tags"* contains the other semantic information in the form of key-value pairs. For each window, it contains five keys: *"window'*, *"width"*, *"height"*, *"breast"* and *"level"*. Among them, *"window=yes"* is used to indicate the type of this opening. *"width"* is the width of the OEL of this window. *"height"* and *"breast"* are provided by user since these two values can not be retrieved from 2D floor plans. In this thesis, for the sake of convenience, the height and breast of all windows in a building are set to be a fixed value. *"level"* is the level number this window belongs to, which is specified each time a floor plan is processed. For doors, the case is similar (see Table 4-5).

Table 4-6: ways

Table<ways>		
Attributes		Description
id		Identification number for this contour, automatically generated sequentially.
tags	name	Generated sequentially, e.g."name=room 0-0" when it is the first room on the ground floor; Generated automatically according to the level when it is the level shell, e.g. "name=FirstFloorShell".
	indoor	"indoor=yes".
	height	Provided by user.
	buildingpart	"buildingpart = room" if it is a room; skipped if it is a level shell.
	level	The level this contour belongs to, automatically generated when the level of the floor plan is specified by user.
geom		An ordered sequence of points in the contour

Every time a contour is imported, a coresponding record will be generated in table<ways>. There are also three attributes in the table for each record: *"id"*, *"geom"* and *"tags"*. *"id"* is the identification number for this way, which will be generated automatically in order; *"geom"* contains the ordered sequence of points in the contour; *"tags"* contains the other semantic information in the form of key-value pairs: *"name"*, *"indoor"*, *"height"*, *"buildingpart"* and *"level"*. *"name"* is the name of the room or level shell this contour represents. For rooms in each floor, their names will be automatically generated in sequence. For example, the first room on the ground floor will be named "room 0-0" and the first room on the first floor "room 1-0". For level shells, their names will be automatically generated according to the level, e.g. "GroundFloorShell", "FirstFloorShell". "indoor=yes" indicates this area is an indoor space. "height" is the height of the room or the height of the level, which will be provided by user. Here again, for the sake of convenience, the height of each level are set to be a same value and the height of each room are set to be the level height minus a fixed value. "buildingpart" only applies to rooms. *"level"* is the level number, which is specified each time a floor plan is processed (see Table 4-6).

Table 4-7: relations

Table<relations>		
Attributes		Description
id		Identification number for this relation, automatically generated sequentially.
tags	name	Generated automatically according to the level when it represents a level, e.g. "name=FirstFloorLevel"; If it represents a building, the name of this building is provided by user.
	type	"type = level" when it represents a level; "type = building" when it represents a building.
	height	Height of the level or the,total height of the building, provided by user.
	level	The level number when it represents a level; Skipped when it represents a building;
	building:levels	The number of levels, only applied to a building relation.
	building:min_levels	The lowest level, only applied to a building relation.
	building:max_levels	The highest level, only applied to a building relation.

Each time the processing of a floor plan finished, a coresponding record of a level relation will be generated in *table<relations>*. There are two attributes in the table for each record: *"id"* and *"tags"*. *"id"* is the identification number for this *"relation"*, which will be generated automatically in order; *"tags"* contains the other semantic information of this *"relation"* in the form of key-value pairs. For each level relaton, it contains four keys: *"name"*, *"type"*, *"height"* and *"level"*. *"name"* will be automatically generated according to the level, e.g. "GroundFloorLevel", "FirstFloorLevel". "type = level" indicates it represents a level relation; "height" is the height of level provided by user. *"level"* is the level number which will be automatically generated in sequence (see Table 4-7).

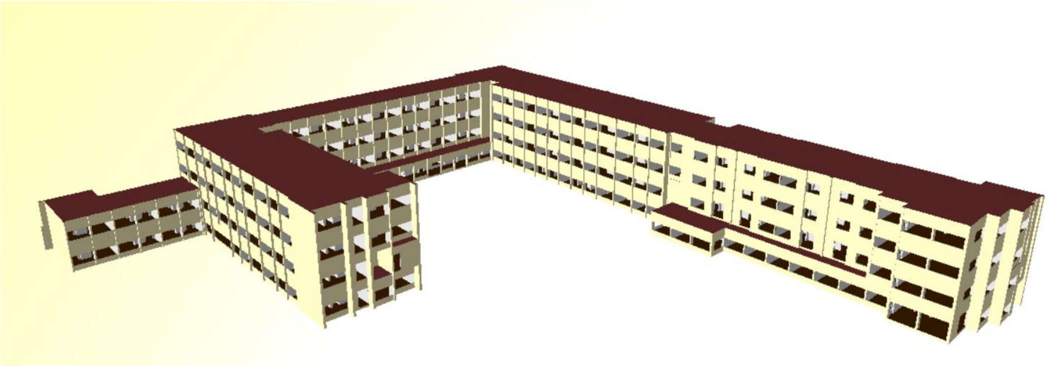
In addition, for *table<relation_members>*, each time a record of way is inserted, a coresponding record of the relation between this way and the level it belongs to will be generated; each time a floor plan is processed, a coresponding record of the relation between this level and the building will be generated (see Table 4-8).

Table 4-8: relation_members

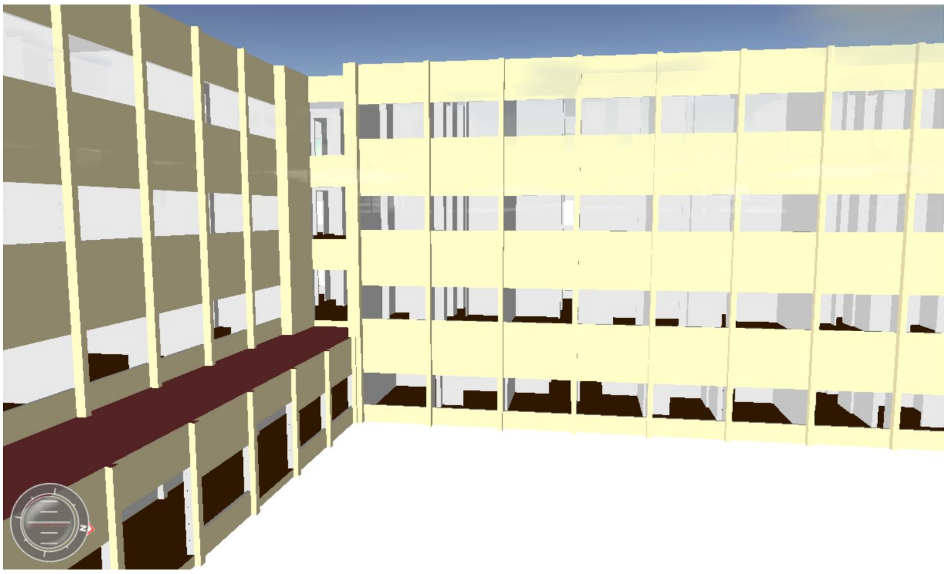
Table<relation_members>	
Attributes	Description
relation_id	Automatically generated, refers to the id of the relation in *Table<relations>*
member_id	Automatically generated: Refers to the id of the member in *Table<ways>* when it is a way; Refers to the id of the member in *Table<relations>* when it is a relation.
member_type	Automatically generated according to the type of the member: "member_type = W" when the member is a way; "member_type = R" when the member is a relation.
member_role	Automatically generated: "member_role = buildingpart" when the member is a room; "member_role = shell" when the member is a level shell; "member_role = level_{the level number}" when it is a level relation.

4-7 3D Reconstruction

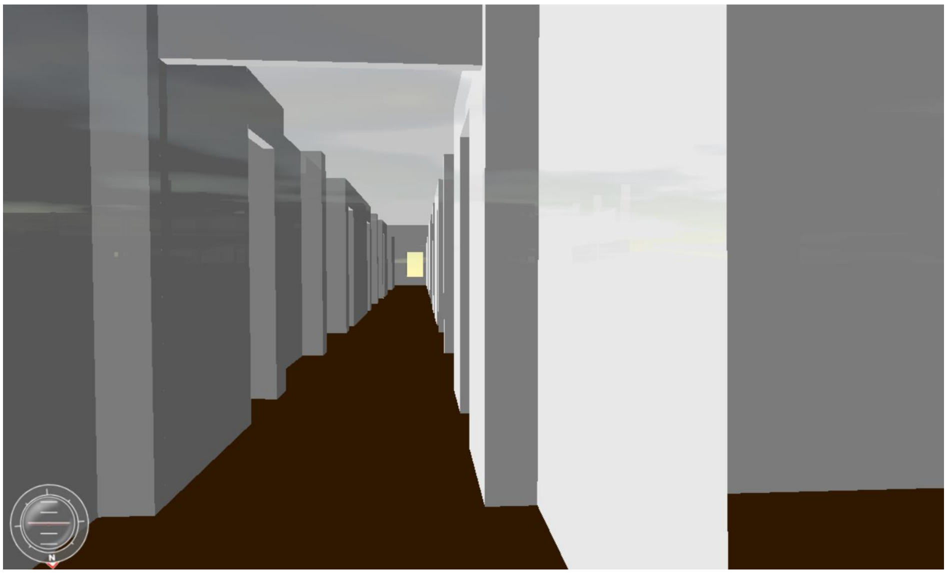
Fig. 4-7 shows the 3D building models looked from different views generated by the program developed by Dr. Marcuz by simple extrusion of the contours that have been extracted the test floor plans by our algorithms. Figs. 4-7a to 4-7c is the 3D model created from building EB_alle_niveaus. Figs. 4-7d and 4-7e is the 3D model created from the ground floor of architecture faulty of TU Delft. In Fig. 4-7e, the outer surface of the model has been removed for a clear view of the indoor space. Fig. 4-7f is the 3D model created from Binnenvest. For all the models, their indoor environment can be explored by zoom-in like Fig. 4-7c.



(a)

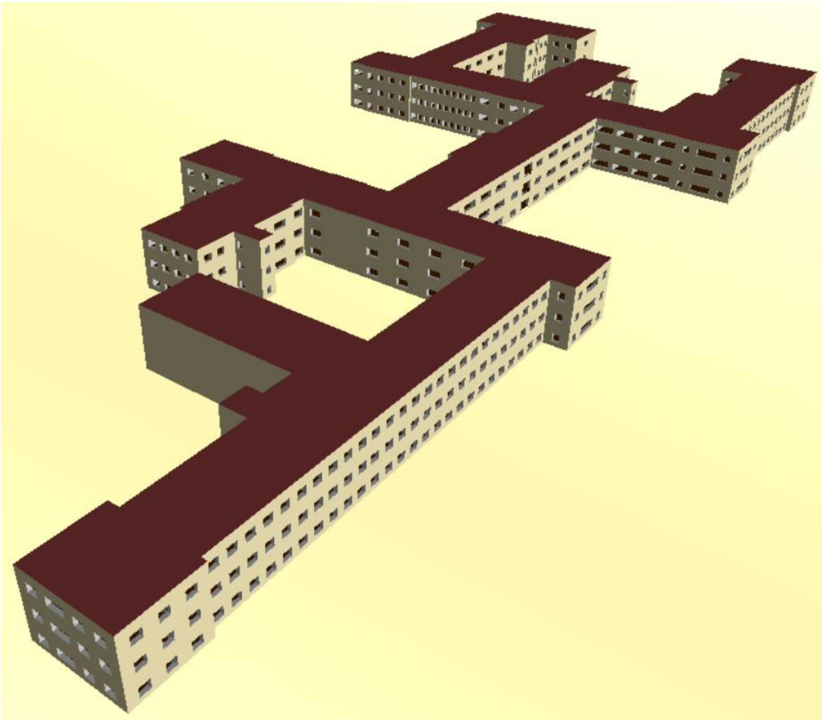


(b)

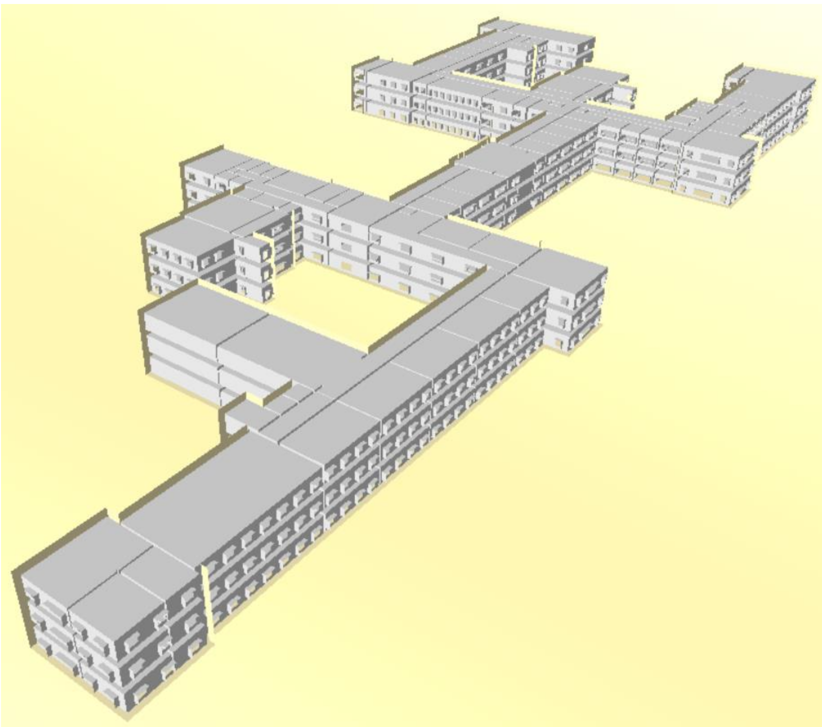


(c)

Figure 4-7: 3D models reconstructed in CityGML LOD4

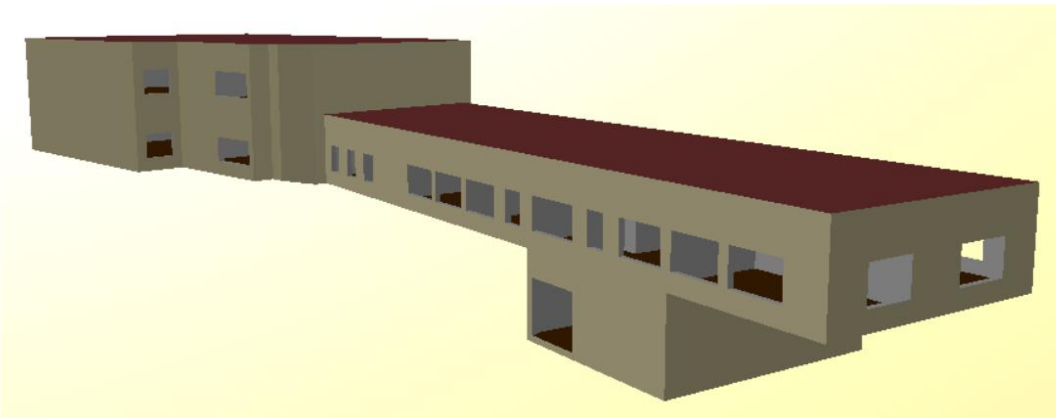


(d)



(e)

Figure 4-7: 3D models reconstructed in CityGML LOD4



(f)

Figure 4-7: 3D models reconstructed in CityGML LOD4

Chapter 5

Conclusions

In this thesis, the possibility of using 2D CAD architectural floor plans as input data for 3D reconstruction is investigated. Accordingly, a semiautomatic process is proposed and tested with several floor plans from real life. The research questions:

(1) *What information about indoor environment is contained in real-life floor plans and among them which can be exported into IndoorOSM for 3D reconstruction?*

(2) *In what way can the information to be used be extracted from the floor plans?*

(3) *How should the extracted information be reorganized in the form of IndoorOSM?*

can be answered as below:

(1) This thesis performed a throughout review of the characters of real-life floor plans. Various content and graphical representation, as well as ambiguities and inconsistencies existing in real-life floor plans are fully analyzed. Together with the literature review of other researches, it is concluded that structural objects such as walls and columns, and openings like windows and doors, are the most important content contained in the floor plans with regrads of the buildingąŕs indoor environment. Meanwhile, they are also the most basic elements in IndoorOSM. Therefore, when using architectural floor plans as input data for IndoorOSM, walls, columns, windows and doors should be extracted into the database at least.

(2) This thesis concluded that real-life floor plans should be redrawn to facilitate automatic 3D reconstruction. Some basic rules are accordingly proposed for the redrawing. The proposed rules for redrawing floor plans mainly focus on segmentation of information contained in floor plans, taking advantages of the layering and blocking supported by CAD application, with reserving the original graphical representation in the raw floor plans as much as possible. By doing this, other additional information can also be contained in the floor plans in other layers without intervening the 3D reconstruction. Thus, the redrawn floor plans are not proprietary for 3D reconstruction but can also be used for other application purposes.

(3) This thesis tested the wall detection algorithm that regards walls as parallel line pairs in the floor plans. Limitations of such algorithm have been found. A new method is thus proposed in this thesis to reconstruct the contours of indoor spaces by replacing openings with parallel line pairs in different ways in accordance with the layout between an opening and its adjacent walls. In this method, complicated symbol recognition techniques are avoided. Instead, openings are reconstructed using bounding box of blocks to estimate the location and orientation of them in the floor plans. Besides, the preprocessing for fixing drafting errors contained in floor plans is also improved, by further considering null-length and duplicated line segments, in addition to disjoint vertices.

Meanwhile, some problems still remain open and require for deeper research in future work:

(1) Algorithms proposed in this thesis have multiple thresholds, which need to be provided by the user based on the specific scenario of a given floor plan (e.g. greatest wall thickness or greatest opening width). Besides, some thresholds do not subject to floor plans but their optimal values need to be tested with multiple floor plans multiple times (e.g. the searching radius for disjoint vertices). The study of a way to automatically compute the optimum value for these thresholds needs to be conducted.

(2) Using bounding box of blocks to estimate the location and orientation of openings requires the primitives in the block are defined aligned with x and y axes in the local coordinate system and that the width of the openings is larger than its height. In some cases where these conditions are not fulfilled, the direction of the calculated OEL will be wrong, causing the OEL cannot successfully intersect with its adjacent wall lines and that some contours of indoor spaces cannot be reconstructed. Thus, this method need to be further improved to be more robust.

(3) he redrawing rules need to be further elaborated. First, the redrawing rules put forward in this thesis only focus on walls and openings. As a result, the 3D models reconstructed by this thesis lack semantic information (e.g. rooms and corridors cannot be distinguished, vertical connectors are unknown). How to set up more rules to include more information in the redrawing phase, and how should this information be processed in 2D, to enrich the semantics in the final 3D models, needs to be further studied. Second, some redrawing rules proposed in this thesis might be too strict. To make the redrawing rules easier to be conformed, the later algorithms need to be improved to be more capable of handling multiple representations in the floor plans.

(4) In this thesis, only normal-structured buildings, in which there is no room on each floor that crosses over several floors, can be reconstructed. In further research, how to extract information from floor plans of buildings with more complicated structure to fully restore the buildingsąŕ indoor spatial environment needs to be investigated. Besides, the buildingąŕs roof shape should be considered in further research, instead of simplifying every roof as flat.

To conclude, 2D architectural floor plans are a very promising data source for 3D reconstruction. In this thesis, it has been proved possible that formatted 2D architectural floor plans can be used as input data in the IndoorOSM 3D reconstruction pipeline. However, at present it is still very hard to fully automatically realize this with a raw floor plan from real life. Some trade-offs have to be made between designers of floor plans and the users of 3D models, or between the preprocessing and the reconstruction.

Appendix A

Source code(part)

This appendix shows part of the source code of the whole process. Each step of this process is chained together through *main.py*. Meanwhile, configuration for all necessary parameters and the connection between extracted information and the database are also realized in this file. Besides, the source code of some other important side functions are selected to be shown here as well. For complete source code please click here.

A-1 main.py

```
import os
import time
import math
import shapefile
import fiona
import psycopg2
import dxfgrabber

from shapely.geometry import Point
from shapely.geometry import LineString
from shapely.geometry import Polygon
from shapely.geometry import polygon

from collections import OrderedDict
from fix_drafting_errors import fix_duplicated_lines
from fix_drafting_errors import fix_disjoint_vertices

from untitled0 import blockbbox
from untitled0 import dist_p2l
from untitled0 import GetProjectivePoint
from untitled0 import separate_in_out

from extend_line import extend_line_onedir
```

```python
from extend_line import extend_line_bothdir

from Opening import Door
from Opening import Window
from LineGroupingFromSHP import LineGroupingFromSHP
from calcOpeningBoundingBox import calcOpeningBoundingBox
from ContourReconstruction00 import ContourReconstruction00



start_time = time.time()

#-------INPUT & OUTPUT SETTINGS-------
#DXF_FILENAMES=['EB_alle_niveaus_ground_floor_changed.dxf', 'EB_alle_niveaus_first_floor_changed.dxf', 'EB_alle_niveaus_second_floor_changed.dxf', 'EB_alle_niveaus_second_floor_changed.dxf']
#DXF_FILENAMES=['BK_preprocessed_changed.dxf','BK_preprocessed_changed.dxf','BK_preprocessed_changed.dxf']
DXF_FILENAMES=['Binnenvest_03.dxf','Binnenvest_01_changed.dxf','Binnenvest_02_changed.dxf']
#DXF_FILENAMES=['Binnenvest_01_changed.dxf','Binnenvest_02_changed.dxf']

WALL_LAYER_NAME='Walls'
WINDOW_LAYER_NAME='Windows'
DOOR_LAYER_NAME='Doors'
sourceCRS_EPSG=31463

#SHP_FILENAMES=['EB_alle_niveaus_ground_floor_fixed.shp', 'EB_alle_niveaus_first_floor_fixed.shp', 'EB_alle_niveaus_second_floor_fixed.shp', 'EB_alle_niveaus_second_floor_fixed.shp']
#SHP_FILENAMES=['BK_preprocessed_fixed.shp','BK_preprocessed_fixed.shp','BK_preprocessed_fixed.shp']
SHP_FILENAMES=['Binnenvest_03_fixed.shp','Binnenvest_01_fixed.shp','Binnenvest_02_fixed.shp']
#SHP_FILENAMES=[Binnenvest_01_fixed.shp','Binnenvest_02_fixed.shp']

EXPORT_DATA_INTO_DATABASE=True

#EXPORT_DATA_INTO_QGIS=True
#-----------------------------------

#-------DATABASE SETTINGS-------
#DBNAME="EB_alle_niveaus"
DBNAME="Binnenvest"
USER="postgres"
PASSWORD="lyyz064101011"
#-----------------------------------

#-------BUILDING GENERAL INFORMATION-------
BUILDINGNAME='Binnenvest'
BUILDINGHEIGHT=12
```

```
BUILDINGLEVELS=3
MINLEVEL=0
MAXLEVEL=2

#FLOORNAMES=['GroundFloor','FirstFloor','SecondFloor','ThirdFloor']
FLOORNAMES=['Basement','GroundFloor','FirstFloor']
LEVELHEIGHT=4
ROOMHEIGHT=3
DOORHEIGHT=2.5
WINDOWHEIGHT=1.5
WINDOWBREAST=0.5
#-----------------------------------

#-------2D PROCESSING SETTINGS-------
MINIMALDIST=5

#AVG_WALL_THICKNESS=230
#AVG_WALL_THICKNESS=800
AVG_WALL_THICKNESS=450

verbose=False  # provide detailed information while processing
#-----------------------------------

#-------configure the database-------
if EXPORT_DATA_INTO_DATABASE==True:

    # Connect to an existing database
    conn = psycopg2.connect("dbname=" + DBNAME + " user=" + USER + " password=" + PASSWORD + "")

    # Open a cursor to perform database operations
    cur = conn.cursor()


    # Drop all tables if they exist.
    cur.execute("""
                    DROP TABLE IF EXISTS nodes;
                    DROP TABLE IF EXISTS ways;
                    DROP TABLE IF EXISTS way_nodes;
                    DROP TABLE IF EXISTS relations;
                    DROP TABLE IF EXISTS relation_members;
                """)

    # Create a table for nodes.
    cur.execute(""" CREATE TABLE nodes (id bigint NOT NULL,
                                        tags hstore);
                """)

     # Create a table for ways.
    cur.execute("""CREATE TABLE ways (id bigint NOT NULL,
                                      tags hstore);
                """)

```

```
    # Add a postgis point column holding the location of the node.
    cur.execute("SELECT AddGeometryColumn('nodes', 'geom', " + str(sourceCRS_EPSG) + ", 'POINT', 2);")
    cur.execute("SELECT AddGeometryColumn('ways', 'linestring', " + str(sourceCRS_EPSG) + ", 'LINESTRING', 2);")

    # Create a table for relations.
    cur.execute("""CREATE TABLE relations (id bigint NOT NULL,
                                           tags hstore);""")

    # Create a table for representing relation member relationships.
    cur.execute("""CREATE TABLE relation_members (relation_id bigint NOT NULL,
                                                  member_id bigint NOT NULL,
                                                  member_type character(1) NOT NULL,
                                                  member_role text NOT NULL);""")

    # Add primary keys to tables.
    cur.execute(""" ALTER TABLE ONLY nodes ADD CONSTRAINT pk_nodes PRIMARY KEY (id);
                    ALTER TABLE ONLY ways ADD CONSTRAINT pk_ways PRIMARY KEY (id);

                    ALTER TABLE ONLY relations ADD CONSTRAINT pk_relations PRIMARY KEY (id);
                    ALTER TABLE ONLY relation_members ADD CONSTRAINT pk_relation_members PRIMARY KEY (relation_id, member_id);
                """)

    # Add indexes to tables.
    cur.execute(""" CREATE INDEX idx_nodes_geom ON nodes USING gist (geom);
                    CREATE INDEX idx_relation_members_member_id_and_type ON relation_members USING btree (member_id, member_type);
                """)

    # Set to cluster nodes by geographical location.
    cur.execute("""ALTER TABLE ONLY nodes CLUSTER ON idx_nodes_geom;""")

    # Set to cluster the tables showing relationship by parent ID and sequence
    cur.execute("ALTER TABLE ONLY relation_members CLUSTER ON pk_relation_members;")

    # Insert the building relation record into TABLE RELATION
    tag="hstore(array['type','building','height','name','building:levels','building:max_level','building:min_level'],array['building', 'yes', '" + str(BUILDINGHEIGHT) +"', '" + BUILDINGNAME + "', '" +
```

```
        str(BUILDINGLEVELS)+ "','" + str(MAXLEVEL) + "','" + str(MINLEVEL)+"']")"
    cur.execute("INSERT INTO relations (id, tags) VALUES (" + str(0) + ", " + tag + ");")

    # Make the changes to the database persistent
    conn.commit()
#----------------------------------

#-------for each floor-------
numNODES=0
numWAYS=0
levelID=1

for level in range(MINLEVEL, MAXLEVEL+1):

    print "#----------------------------"
    print "This is level: "+str(level)

    script_dir = os.path.dirname(__file__)
#-------Unconnected vertices fixing & lines grouping-------
    # groupedPoints is groups of points representing wall polygons
    abs_file_path = os.path.join(script_dir, 'INPUT_DATA/' + SHP_FILENAMES[level])
    groupedPoints = LineGroupingFromSHP(abs_file_path, MINIMALDIST)

#-------Calculate bounding box of openings and create opening objects-------
    # openings is CLASS OPENINGS to be used for contour reconstruction
    abs_file_path = os.path.join(script_dir, 'INPUT_DATA/' + DXF_FILENAMES[level])
    openings = calcOpeningBoundingBox(abs_file_path, WINDOW_LAYER_NAME, DOOR_LAYER_NAME)

#-------Reconstruct contours from wall lines and opening lines-------
    # Nodes is openings that can be successfully reconstructed and exported into TABLE NODES
    # contourPoint is the corner points of the ways to be exported into TABLE WAYS representing contours
    contourPoints, Nodes=ContourReconstruction00(groupedPoints, openings, AVG_WALL_THICKNESS, verbose)

#-------Find level shell and filter out columns-------
    maxS=0
    indx_Shell=-1
    indx_Columns=[]

    for i in range(0, len(contourPoints)):
        if len(contourPoints[i])>2:
            S=Polygon(contourPoints[i]).area
            if S<1000000: # contours with area smaller than 1 m2 are considered columns
                indx_Columns.append(i)
```

```
            elif S>maxS:
                maxS=S
                indx_Shell=i
    levelshell=contourPoints[indx_Shell]

    indx_ToDelete=indx_Columns+[indx_Shell]
    indx_ToDelete.sort()

    for i in range(0, len(indx_ToDelete)):
        contourPoints.pop(indx_ToDelete[i]-i)
#-----------------------------------

#-------Export data into database-------
    if EXPORT_DATA_INTO_DATABASE==True:


        # Insert the level relation record into TABLE RELATION
        tag="hstore(array['type','name','height','level'],array['level','" + FLOORNAMES[level] + "Level" + "', '" + str(LEVELHEIGHT) + "','" + str(level) + "'])"
        cur.execute("INSERT INTO relations (id, tags) VALUES (" + str(levelID) + ", " + tag + ");")
        conn.commit()


        # Insert the relation between this level and the building into TABLE RELATION_MEMBERS
        cur.execute("INSERT INTO relation_members (relation_id, member_id, member_type, member_role) VALUES (" + str(0) + ", " + str(levelID) + ", 'R', 'level_" + str(level) + "');")
        conn.commit()



        # Insert doors and windows into TABLE NODES
        for i in range(0, len(Nodes)):
            if Nodes[i].type==0: # doors
                tag="hstore(array['door','level', 'width','height'],array['yes','" + str(level) + "','" + str(Nodes[i].length/1000) + "','" + str(DOORHEIGHT) +"'])"
            else: # windows
                tag="hstore(array['window','level', 'width','height','breast'],array['yes','" + str(level) + "','"  + str(Nodes[i].length/1000) + "','" + str(WINDOWHEIGHT) +"','"+str(WINDOWBREAST)+"'])"

            cur.execute("INSERT INTO nodes (id, tags, geom) VALUES (" + str(numNODES+i) + ", " + tag + ", ST_GeomFromText('POINT(" + str(Nodes[i].center.y/1000) + " " + str(Nodes[i].center.x/1000) + ")', " + str(sourceCRS_EPSG) + "));")
            conn.commit()

        # Insert level shell of this floor into TABLE WAYS
```

```
        geom="'LINESTRING("
        for i in range(0, len(levelshell)):
            geom=geom+str(levelshell[i][1]/1000)+" "+str(levelshell[i][0]/1000)+","
        geom=geom+str(levelshell[0][1]/1000)+" "+str(levelshell[0][0]/1000)+")'"

        tag="hstore(array['name','height','level'],array['" + FLOORNAMES[level] + "Shell" + "','" + str(LEVELHEIGHT) +"','" + str(level) + "'])"
        cur.execute("INSERT INTO ways (id, tags, linestring) VALUES (" + str(numWAYS) + ", " + tag + ", ST_GeomFromText(" + geom + ", " + str(sourceCRS_EPSG) + "));")

        # Insert the relation between the shell and this level into TABLE RELATION_MEMBERS
        cur.execute("INSERT INTO relation_members (relation_id, member_id, member_type, member_role) VALUES (" + str(level+1) + ", " + str(numWAYS) + ", 'W', 'shell');")
        conn.commit()


        # Insert ways of rooms into TABLE WAYS
        for i in range(0, len(contourPoints)):
            if len(contourPoints[i])>0:
                geom="'LINESTRING("
                for j in range(0, len(contourPoints[i])):
                    pt=contourPoints[i][j]
                    geom=geom+str(contourPoints[i][j][1]/1000)+" "+str(contourPoints[i][j][0]/1000)+","
            geom=geom+str(contourPoints[i][0][1]/1000)+" "+str(contourPoints[i][0][0]/1000)+")'"

            tag="hstore(array['name','buildingpart','height','indoor'],array['Room "+str(level)+ "-" + str(i) + "','room','" + str(ROOMHEIGHT) +"','yes'])"
            cur.execute("INSERT INTO ways (id, tags, linestring) VALUES (" + str(numWAYS+i+1) + ", " + tag + ", ST_GeomFromText(" + geom + ", " + str(sourceCRS_EPSG) + "));")
            # Insert the relation between the way and this level into TABLE RELATION_MEMBERS
            cur.execute("INSERT INTO relation_members (relation_id, member_id, member_type, member_role) VALUES (" + str(levelID) + ", " + str(numWAYS+i+1) + ", 'W', 'buildingpart');")
            conn.commit()

        levelID=levelID+1
        numNODES=numNODES+len(Nodes)
        numWAYS=numWAYS+len(contourPoints)+1
#------------------------------------

#-------Close communication with the database-------
```

```
if EXPORT_DATA_INTO_DATABASE==True:
    cur.close()
    conn.close()
#-----------------------------------

#-----------------------------------
print '#------------------------------'
print 'finished!'
print 'executing time:', time.time() - start_time, 'seconds'
print time.strftime('%H:%M:%S', time.gmtime(time.time() - start_time))
print '#------------------------------'
```

A-2 calcOpeningBoundingBox.py

```
import math
import shapefile
import dxfgrabber

from shapely.geometry import Point
from shapely.geometry import LineString
from extend_line import extend_line_onedir
from extend_line import extend_line_bothdir
from Opening import Door
from Opening import Window


def calcOpeningBoundingBox(abs_file_path, WINDOW_LAYER_NAME, DOOR_LAYER_NAME):

    dxf = dxfgrabber.readfile(abs_file_path, {"grab_blocks":True, "assure_3d_coords":False, "resolve_text_styles":False})
    windows=[]
    doors=[]

    for i in dxf.entities:
#----------Windows----------
        if i.layer == WINDOW_LAYER_NAME and i.dxftype == 'INSERT':
            X0=i.insert[0]
            Y0=i.insert[1]
            angle=math.radians(i.rotation)
            xs=i.scale[0]
            ys=i.scale[1]
            result=blockbbox(i, dxf, X0, Y0, angle, xs, ys)
            p1=result[0]
            p2=result[1]
            p3=result[2]
            p4=result[3]
            if p1.distance(p2)>=p2.distance(p3):
                pp1,pp2=extend_line_bothdir(Point((p1.x+p4.x)/2, (p1.y+p4.y)/2), Point((p2.x+p3.x)/2, (p2.y+p3.y)/2), 20)
                width=p2.distance(p3)
```

```
                length=Point((p1.x+p4.x)/2, (p1.y+p4.y)/2).distance(Point((p2.x+p3.x)/2, (p2.y+p3.y)/2))
            else:
                pp1,pp2=extend_line_bothdir(Point((p1.x+p2.x)/2, (p1.y+p2.y)/2), Point((p3.x+p4.x)/2, (p3.y+p4.y)/2), 20)
                width=p1.distance(p2)
                length=Point((p1.x+p2.x)/2, (p1.y+p2.y)/2).distance(Point((p3.x+p4.x)/2, (p3.y+p4.y)/2))
            windows.append(Window(LineString([(pp1.x,pp1.y),(pp2.x,pp2.y)]), width, length))

#----------Doors----------
        if i.layer == DOOR_LAYER_NAME and i.dxftype == 'INSERT':

            X0=i.insert[0]
            Y0=i.insert[1]
            angle=math.radians(i.rotation)
            xs=i.scale[0]
            ys=i.scale[1]
            result=blockbbox(i, dxf, X0, Y0, angle, xs, ys)
            p1=result[0]
            p2=result[1]
            p3=result[2]
            p4=result[3]
            if p1.distance(p2)>=p2.distance(p3):
                pp1,pp2=extend_line_bothdir(Point((p1.x+p4.x)/2, (p1.y+p4.y)/2), Point((p2.x+p3.x)/2, (p2.y+p3.y)/2), 20)
                width=p2.distance(p3)
                length=Point((p1.x+p4.x)/2, (p1.y+p4.y)/2).distance(Point((p2.x+p3.x)/2, (p2.y+p3.y)/2))
            else:
                pp1,pp2=extend_line_bothdir(Point((p1.x+p2.x)/2, (p1.y+p2.y)/2), Point((p3.x+p4.x)/2, (p3.y+p4.y)/2), 20)
                width=p1.distance(p2)
                length=Point((p1.x+p2.x)/2, (p1.y+p2.y)/2).distance(Point((p3.x+p4.x)/2, (p3.y+p4.y)/2))
            doors.append(Door(LineString([(pp1.x,pp1.y),(pp2.x,pp2.y)]), width, length))

    openings=[]
    openings.extend(windows)
    openings.extend(doors)

    return openings

def blockbbox(block, dxf, X0, Y0, angle, xs, ys):
    anchors=[]
    lines=[]
    xmin=float('inf')
    xmax=float('-inf')
    ymin=float('inf')
    ymax=float('-inf')

```

```
    for j in dxf.blocks[block.name]:
        if j.dxftype=='LINE':
            if j.start[0]>xmax:
                xmax=j.start[0]
            if j.start[0]<xmin:
                xmin=j.start[0]
            if j.start[1]>ymax:
                ymax=j.start[1]
            if j.start[1]<ymin:
                ymin=j.start[1]
            if j.end[0]>xmax:
                xmax=j.end[0]
            if j.end[0]<xmin:
                xmin=j.end[0]
            if j.end[1]>ymax:
                ymax=j.end[1]
            if j.end[1]<ymin:
                ymin=j.end[1]
            lines.append(LineString([(j.start[0], j.start[1]), (j.end[0], j.end[1])]))
#----------------------

        elif j.dxftype=='POLYLINE':
            if j.is_closed==True:
                lines.append(LineString([(j.points[0][0], j.points[0][1]), (j.points[-1][0], j.points[-1][1])]))
            for k in range(0, len(j.points)):
                if j.points[k][0]>xmax:
                    xmax=j.points[k][0]
                if j.points[k][0]<xmin:
                    xmin=j.points[k][0]
                if j.points[k][1]>ymax:
                    ymax=j.points[k][1]
                if j.points[k][1]<ymin:
                    ymin=j.points[k][1]
                if k<len(j.points)-1:
                    lines.append(LineString([(j.points[k][0], j.points[k][1]), (j.points[k+1][0], j.points[k+1][1])]))
#----------------------

        elif j.dxftype=='LWPOLYLINE':
            if j.is_closed==True:
                lines.append(LineString([(j.points[0][0], j.points[0][1]), (j.points[-1][0], j.points[-1][1])]))
            for k in range(0, len(j.points)):
                if j.points[k][0]>xmax:
                    xmax=j.points[k][0]
                if j.points[k][0]<xmin:
                    xmin=j.points[k][0]
                if j.points[k][1]>ymax:
                    ymax=j.points[k][1]
                if j.points[k][1]<ymin:
                    ymin=j.points[k][1]
```

```
                if k<len(j.points)-1:
                    lines.append(LineString([(j.points[k][0], j.points[k][1]), (j.points[k+1][0], j.points[k+1][1])]))
#---------------------

        elif j.dxftype=='ARC':
            anchors.append(Point(j.center[0], j.center[1]))
            if j.center[0]>xmax:
                xmax=j.center[0]
            if j.center[0]<xmin:
                xmin=j.center[0]
            if j.center[1]>ymax:
                ymax=j.center[1]
            if j.center[1]<ymin:
                ymin=j.center[1]

            x0=j.center[0]+j.radius*math.cos(math.radians(j.startangle))
            y0=j.center[1]+j.radius*math.sin(math.radians(j.startangle))
            x1=j.center[0]+j.radius*math.cos(math.radians(j.endangle))
            y1=j.center[1]+j.radius*math.sin(math.radians(j.endangle))
            if x0>xmax:
                xmax=x0
            if x0<xmin:
                xmin=x0
            if y0>ymax:
                ymax=y0
            if y0<ymin:
                ymin=y0
            if x1>xmax:
                xmax=x1
            if x1<xmin:
                xmin=x1
            if y1>ymax:
                ymax=y1
            if y1<ymin:
                ymin=y1
            lines.append(LineString([(j.center[0], j.center[1]), (x0, y0)]))
            lines.append(LineString([(j.center[0], j.center[1]), (x1, y1)]))
            lines.append(LineString([(x0, y0), (x1, y1)]))
#---------------------

        elif j.dxftype=='INSERT':
            X0_0=j.insert[0]
            Y0_0=j.insert[1]
            angle_0=math.radians(j.rotation)
            xs_0=j.scale[0]
            ys_0=j.scale[1]
            result=blockbbox(j, dxf, X0_0, Y0_0, angle_0, xs_0, ys_0)
            for i in range(0,4):
                if result[i].x<xmin:
                    xmin=result[i].x
```

```
                if result[i].x>xmax:
                    xmax=result[i].x
                if result[i].y<ymin:
                    ymin=result[i].y
                if result[i].y>ymax:
                    ymax=result[i].y
            lines.extend(result[4])
            anchors.extend(result[5])
#----------------------

    if len(anchors)>0:
        p1=Point(xmin,ymin)
        p2=Point(xmax,ymin)
        p3=Point(xmax,ymax)
        p4=Point(xmin,ymax)
        if p1.distance(p2)>=p2.distance(p3):
            d1=dist_p2l(anchors[0], p1, p2)
            d2=dist_p2l(anchors[0], p3, p4)
            p14=Point((p1.x+p4.x)/2, (p1.y+p4.y)/2)
            p23=Point((p2.x+p3.x)/2, (p2.y+p3.y)/2)
            d3=dist_p2l(anchors[0], p14, p23)
            if d1<=d2 and d1<=d3:
                ymax=ymin+120

            elif d2<=d1 and d2<=d3:
                ymin=ymax-120
            else:
                ymax=p14.y+120/2
                ymin=p14.y-120/2
        else:
            d1=dist_p2l(anchors[0], p1, p4)
            d2=dist_p2l(anchors[0], p2, p3)
            p12=Point((p1.x+p2.x)/2, (p1.y+p2.y)/2)
            p34=Point((p3.x+p4.x)/2, (p3.y+p4.y)/2)
            d3=dist_p2l(anchors[0], p12, p34)
            if d1<=d2 and d1<=d3:
                xmax=xmin+120
            elif d2<=d1 and d2<=d3:
                xmin=xmax-120
            else:
                xmax=p12.x+120/2
                xmin=p12.x-120/2
    new_lines=[]
    for line in lines:
        p0=coordtransformation(Point(list(line.coords)[0]), angle, X0, Y0
            , xs, ys)
        p1=coordtransformation(Point(list(line.coords)[1]), angle, X0, Y0
            , xs, ys)
        new_lines.append(LineString([(p0.x, p0.y),(p1.x, p1.y)]))
    new_anchors=[]
    if len(anchors)>0:
        for anchor in anchors:
            new_anchor=coordtransformation(anchor, angle, X0, Y0, xs, ys)
```

```
                new_anchors.append(new_anchor)
    p1=coordtransformation(Point(xmin,ymin), angle, X0, Y0, xs, ys)
    p2=coordtransformation(Point(xmax,ymin), angle, X0, Y0, xs, ys)
    p3=coordtransformation(Point(xmax,ymax), angle, X0, Y0, xs, ys)
    p4=coordtransformation(Point(xmin,ymax), angle, X0, Y0, xs, ys)

    return [p1, p2, p3, p4, new_lines, new_anchors]
```

A-3 ContourReconstruction.py

```
import math
import shapefile

from shapely.geometry import Point
from shapely.geometry import LineString
from shapely.geometry import Polygon
from shapely.geometry import polygon

from untitled0 import GetProjectivePoint
from untitled0 import separate_in_out

from extend_line import extend_line_onedir
from extend_line import extend_line_bothdir
from extend_line import point_on_line


def ContourReconstruction00(groupedPoints, openings, AVG_WALL_THICKNESS,
    verbose):
    Nodes=[]
    groups_P=groupedPoints
    for i in range(0, len(openings)):
        l1=openings[i].mline
        width=openings[i].width
        anchor_lines=[]

        for j in range(0, len(groups_P)):
            for k in range(0, len(groups_P[j])):
                if k==len(groups_P[j])-1:
                    l2=LineString([groups_P[j][k], groups_P[j][0]])
                else:
                    l2=LineString([groups_P[j][k], groups_P[j][k+1]])

                if l1.intersects(l2)==True:
                    anchor_lines.append([j, k, l2])

            if len(anchor_lines)==2:
                break
            else:
                continue
        if j==len(groups_P)-1 and len(anchor_lines)<2:
            print 'Opening ' + str(i) +' reconstruction failed!'
            continue
```

```
        else:
            # openings that can be successfully reconstructed
            Nodes.append(openings[i])

            if anchor_lines[0][2].length>=anchor_lines[1][2].length:
                longL=anchor_lines[0]
                shortL=anchor_lines[1]
            else:
                longL=anchor_lines[1]
                shortL=anchor_lines[0]

            if anchor_lines[0][0]==anchor_lines[1][0]:
                # self-closed
                if verbose==True:
                    print 'self-closed'
                if longL[2].length<=AVG_WALL_THICKNESS:

                    if verbose==True:
                        print 'situation111'

                    if math.fabs(longL[2].length-shortL[2].length)/shortL[2].length<=0.15:
                        outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                        groups_P.pop(longL[0])
                        groups_P.append(outRing)
                        groups_P.append(inRing)
                    else:
                        outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])

                        ptProj0=GetProjectivePoint(Point(groups_P[shortL[0]][shortL[1]][0], groups_P[shortL[0]][shortL[1]][1]), longL[2])
                        ptProj1=GetProjectivePoint(Point(groups_P[shortL[0]][shortL[1]+1][0], groups_P[shortL[0]][shortL[1]+1][1]), longL[2])

                        if groups_P[shortL[0]][shortL[1]] in outRing:
                            new_outRing=outRing+[(ptProj0.x, ptProj0.y)]
                            new_inRing=[(ptProj1.x, ptProj1.y)]+inRing
                        else:
                            new_outRing=[(ptProj1.x, ptProj1.y)]+outRing
                            new_inRing=inRing+[(ptProj0.x, ptProj0.y)]

                        groups_P.pop(longL[0])
                        groups_P.append(new_outRing)
                        groups_P.append(new_inRing)
                elif shortL[2].length>AVG_WALL_THICKNESS:
                    # self-closed
                    if verbose==True:
                        print 'self-closed222'
                    shortP=l1.intersection(shortL[2])
```

```
                    longP=l1.intersection(longL[2])
                    startS=groups_P[shortL[0]][shortL[1]]
                    if shortL[1]+1>=len(groups_P[shortL[0]]):
                        endS=groups_P[shortL[0]][shortL[1]+1-len(groups_P[shortL[0]])]
                    else:
                        endS=groups_P[shortL[0]][shortL[1]+1]

                    startL=groups_P[longL[0]][longL[1]]
                    if longL[1]+1>=len(groups_P[longL[0]]):
                        endL=groups_P[longL[0]][longL[1]+1-len(groups_P[longL[0]])]
                    else:
                        endL=groups_P[longL[0]][longL[1]+1]
                    if shortP.distance(Point(startS[0],startS[1]))<=60:
                        if longP.distance(Point(endL[0],endL[0]))<=60:
                            d1=shortP.distance(Point(startS[0],startS[1]))
                            d2=longP.distance(Point(endL[0],endL[0]))
                            if math.fabs(d1-d2)/d1<=0.15:
                                if verbose==True:
                                    print 'U shape(closed)'
                                # situation 1
                                outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                                new_pt0=point_on_line(Point(endS[0],endS[1]), shortP, (d1+d2)/2)
                                new_pt1=point_on_line(Point(startL[0],startL[1]), longP, (d1+d2)/2)
                                new_outRing=outRing
                                new_inRing=[(new_pt0.x,new_pt0.y)]+inRing+[(new_pt1.x,new_pt1.y)]
                                groups_P.pop(longL[0])
                                groups_P.append(new_outRing)
                                groups_P.append(new_inRing)
                            else:
                                if verbose==True:
                                    print 'U(Z) shape(closed)'
                                if d1<d2:
                                    ptProj=GetProjectivePoint(Point(startS[0],startS[1]), longL[2])
                                    outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                                    new_pt0=point_on_line(Point(endS[0],endS[1]), shortP, (d1+d2)/2)
                                    new_pt1=point_on_line(Point(startL[0],startL[1]), longP, (d1+d2)/2)
                                    new_outRing=outRing+[(ptProj.x,ptProj.y)]
                                    new_inRing=[(new_pt0.x,new_pt0.y)]+inRing+[(new_pt1.x,new_pt1.y)]
                                    groups_P.pop(longL[0])
                                    groups_P.append(new_outRing)
```

```
            groups_P.append(new_inRing)
        else:
            ptProj=GetProjectivePoint(Point(endL[0],endL[1]), shortL[2])
            outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
            new_pt0=point_on_line(Point(endS[0],endS[1]), shortP, (d1+d2)/2)
            new_pt1=point_on_line(Point(startL[0],startL[1]), longP, (d1+d2)/2)
            new_outRing=outRing+[(ptProj.x,ptProj.y)]
            new_inRing=[(new_pt0.x,new_pt0.y)]+inRing+[(new_pt1.x,new_pt1.y)]
            groups_P.pop(longL[0])
            groups_P.append(new_outRing)
            groups_P.append(new_inRing)
elif longP.distance(Point(startL[0],startL[0]))<=60:
    # situation 2
    if verbose==True:
        print 'Z shape(closed)'
    ptProj0=GetProjectivePoint(Point(startS[0],startS[1]), longL[2])
    ptProj1=GetProjectivePoint(Point(startL[0],startL[1]), shortL[2])
    outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
    new_outRing=outRing+[(ptProj0.x, ptProj0.y)]
    new_inRing=[(ptProj1.x, ptProj1.y)]+inRing
    groups_P.pop(longL[0])
    groups_P.append(new_outRing)
    groups_P.append(new_inRing)
else:
    # situation 3
    if verbose==True:
        print '4 shape(closed)'
    ptProj=GetProjectivePoint(Point(startS[0],startS[1]), longL[2])
    outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
    d1=shortP.distance(Point(startS[0],startS[1]))
    new_pt0=point_on_line(Point(endS[0],endS[1]), shortP, d1)
    new_pt1=point_on_line(Point(startL[0],startL[1]), longP, d1)
    if groups_P[shortL[0]][shortL[1]] in outRing:
        new_outRing=outRing+[(ptProj.x, ptProj.y)]
        new_inRing=inRing+[(new_pt1.x,new_pt1.y)]+[(new_pt0.x,new_pt0.y)]
    else:
```

```
            new_outRing=outRing+[(new_pt1.x,new_pt1.y)]+[(new_pt0.x,new_pt0.y)]
            new_inRing=inRing+[(ptProj.x, ptProj.y)]
        groups_P.pop(longL[0])
        groups_P.append(new_outRing)
        groups_P.append(new_inRing)
elif shortP.distance(Point(endS[0],endS[1]))<=60:
    if longP.distance(Point(startL[0],startL[0]))<=60:
        # situation 4
        d1=shortP.distance(Point(endS[0],endS[0]))
        d2=longP.distance(Point(startL[0],startL[1]))
        if math.fabs(d1-d2)/d1<=0.15:
            if verbose==True:
                print 'U shape(closed)'
            outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
            new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, (d1+d2)/2)
            new_pt1=point_on_line(Point(endL[0],endL[1]), longP, (d1+d2)/2)
            new_outRing=outRing
            new_inRing=[(new_pt1.x,new_pt1.y)]+inRing+[(new_pt0.x,new_pt0.y)]
            groups_P.pop(longL[0])
            groups_P.append(new_outRing)
            groups_P.append(new_inRing)
        else:
            if verbose==True:
                print 'U(Z) shape(closed)'
            if d1<d2:
                ptProj=GetProjectivePoint(Point(endS[0],endS[1]), longL[2])
                outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, (d1+d2)/2)
                new_pt1=point_on_line(Point(endL[0],endL[1]), longP, (d1+d2)/2)
                new_outRing=outRing+[(ptProj.x,ptProj.y)]
                new_inRing=[(new_pt1.x,new_pt1.y)]+inRing+[(new_pt0.x,new_pt0.y)]
                groups_P.pop(longL[0])
                groups_P.append(new_outRing)
                groups_P.append(new_inRing)
            else:
                ptProj=GetProjectivePoint(Point(startL[0],startL[1]), shortL[2])
                outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, (d1+d2)/2)
```

```
                                        new_pt1=point_on_line(Point(endL[0],endL[1]), longP, (d1+d2)/2)
                                        new_outRing=outRing+[(ptProj.x,ptProj.y)]
                                        new_inRing=[(new_pt1.x,new_pt1.y)]+inRing+[(new_pt0.x,new_pt0.y)]
                                        groups_P.pop(longL[0])
                                        groups_P.append(new_outRing)
                                        groups_P.append(new_inRing)
                            elif longP.distance(Point(endL[0],endL[0]))<=60:
                                # situation 5
                                if verbose==True:
                                    print 'Z shape(closed)'
                                ptProj0=GetProjectivePoint(Point(endL[0],endL[1]), shortL[2])
                                ptProj1=GetProjectivePoint(Point(endS[0],endS[1]), longL[2])
                                outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                                new_outRing=outRing+[(ptProj1.x, ptProj1.y)]
                                new_inRing=inRing+[(ptProj0.x, ptProj0.y)]
                                groups_P.pop(longL[0])
                                groups_P.append(new_outRing)
                                groups_P.append(new_inRing)
                            else:
                                # situation 6
                                if verbose==True:
                                    print '4 shape(closed)'
                                ptProj=GetProjectivePoint(Point(endS[0],endS[1]), longL[2])
                                outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                                d1=shortP.distance(Point(endS[0],endS[1]))
                                new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, d1)
                                new_pt1=point_on_line(Point(endL[0],endL[1]), longP, d1)
                                if groups_P[shortL[0]][shortL[1]] in outRing:
                                    new_outRing=outRing+[(new_pt0.x,new_pt0.y)]+[(new_pt1.x,new_pt1.y)]
                                    new_inRing=inRing+[(ptProj.x, ptProj.y)]
                                else:
                                    new_outRing=outRing+[(ptProj.x, ptProj.y)]
                                    new_inRing=inRing+[(new_pt0.x,new_pt0.y)]+[(new_pt1.x,new_pt1.y)]
                                groups_P.pop(longL[0])
                                groups_P.append(new_outRing)
                                groups_P.append(new_inRing)
                        else:
                            if longP.distance(Point(startL[0],startL[0]))<=60:
                                # situation 7
```

```
        if verbose==True:
            print '4 shape(closed)'
        ptProj=GetProjectivePoint(Point(startL[0], startL[1]), shortL[2])
        outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
        d1=longP.distance(Point(startL[0], startL[1]))
        new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, d1)
        new_pt1=point_on_line(Point(endL[0],endL[1]), longP, d1)
        new_outRing=outRing+[(ptProj.x, ptProj.y)]
        new_inRing=[(new_pt1.x,new_pt1.y)]+inRing+[(new_pt0.x,new_pt0.y)]
        groups_P.pop(longL[0])
        groups_P.append(new_outRing)
        groups_P.append(new_inRing)
    elif longP.distance(Point(endL[0],endL[0]))<=60:
        # situation 8
        if verbose==True:
            print '4 shape(closed)'
        ptProj=GetProjectivePoint(Point(endL[0], endL[1]), shortL[2])
        outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
        d1=longP.distance(Point(endL[0],endL[1]))
        new_pt0=point_on_line(Point(endS[0],endS[1]), shortP, d1)
        new_pt1=point_on_line(Point(startL[0],startL[1]), longP, d1)
        new_outRing=outRing+[(ptProj.x, ptProj.y)]
        new_inRing=[(new_pt0.x,new_pt0.y)]+inRing+[(new_pt1.x,new_pt1.y)]
        groups_P.pop(longL[0])
        groups_P.append(new_outRing)
        groups_P.append(new_inRing)
    else:
        # situation 9
        if verbose==True:
            print 'H shape(closed)'
        new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, width/2)
        new_pt1=point_on_line(Point(endL[0],endL[1]), longP, width/2)
        new_pt2=point_on_line(Point(startL[0],startL[1]), longP, width/2)
        new_pt3=point_on_line(Point(endS[0],endS[1]), shortP, width/2)
        outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
        if groups_P[shortL[0]][shortL[1]] in outRing:
```

```
                                    new_outRing=outRing+[(new_pt0.x,new_pt0.y)]+[(new_pt1.x,new_pt1.y)]
                                    new_inRing=inRing+[(new_pt2.x,new_pt2.y)]+[(new_pt3.x,new_pt3.y)]
                                else:
                                    new_outRing=outRing+[(new_pt2.x,new_pt2.y)]+[(new_pt3.x,new_pt3.y)]
                                    new_inRing=inRing+[(new_pt0.x,new_pt0.y)]+[(new_pt1.x,new_pt1.y)]
                                groups_P.pop(longL[0])
                                groups_P.append(new_outRing)
                                groups_P.append(new_inRing)
                    else:
                        outRing,inRing=separate_in_out(longL[1],shortL[1],groups_P[longL[0]])
                        ptProj0=GetProjectivePoint(Point(groups_P[shortL[0]][shortL[1]][0], groups_P[shortL[0]][shortL[1]][1]), longL[2])
                        ptProj1=GetProjectivePoint(Point(groups_P[shortL[0]][shortL[1]+1][0], groups_P[shortL[0]][shortL[1]+1][1]), longL[2])
                        if groups_P[shortL[0]][shortL[1]] in outRing:
                            new_outRing=outRing+[(ptProj0.x, ptProj0.y)]
                            new_inRing=[(ptProj1.x, ptProj1.y)]+inRing
                        else:
                            new_outRing=[(ptProj1.x, ptProj1.y)]+outRing
                            new_inRing=inRing+[(ptProj0.x, ptProj0.y)]
                        groups_P.pop(longL[0])
                        groups_P.append(new_outRing)
                        groups_P.append(new_inRing)
                else:
                    if longL[2].length<=AVG_WALL_THICKNESS:
                        if verbose==True:
                            print 'not self-closed','situation 1'
                        if math.fabs(longL[2].length-shortL[2].length)/shortL[2].length<=0.15:
                            longGroup=groups_P[longL[0]]
                            reL=longGroup[longL[1]+1:]+longGroup[0:longL[1]+1]
                            shortGroup=groups_P[shortL[0]]
                            reS=shortGroup[shortL[1]+1:]+shortGroup[0:shortL[1]+1]
                            if longL[0]>shortL[0]:
                                groups_P.pop(longL[0])
                                groups_P.pop(shortL[0])
                            else:
                                groups_P.pop(shortL[0])
                                groups_P.pop(longL[0])
                            groups_P.append(reL+reS)
                        else:
                            longGroup=groups_P[longL[0]]
                            reL=longGroup[longL[1]+1:]+longGroup[0:longL[1]+1]
```

```
                    shortGroup=groups_P[shortL[0]]
                    reS=shortGroup[shortL[1]+1:]+shortGroup[0:shortL[1]+1]
                    ptProj0=GetProjectivePoint(Point(shortGroup[shortL[1]][0],shortGroup[shortL[1]][1]), longL[2])
                    ptProj1=GetProjectivePoint(Point(shortGroup[shortL[1]+1][0],shortGroup[shortL[1]+1][1]),longL[2])
                    if longL[0]>shortL[0]:
                        groups_P.pop(longL[0])
                        groups_P.pop(shortL[0])
                    else:
                        groups_P.pop(shortL[0])
                        groups_P.pop(longL[0])
                    groups_P.append(reS+[(ptProj0.x, ptProj0.y)]+reL+[(ptProj1.x, ptProj1.y)])
            elif shortL[2].length>AVG_WALL_THICKNESS:
                if verbose==True:
                    print 'not self-closed','situation 2'
                longGroup=groups_P[longL[0]]
                reL=longGroup[longL[1]+1:]+longGroup[0:longL[1]+1]
                shortGroup=groups_P[shortL[0]]
                reS=shortGroup[shortL[1]+1:]+shortGroup[0:shortL[1]+1]
                shortP=l1.intersection(shortL[2])
                longP=l1.intersection(longL[2])
                startS=reS[-1]
                endS=reS[0]
                startL=reL[-1]
                endL=reL[0]
                if shortP.distance(Point(startS[0],startS[1]))<=60:
                    if longP.distance(Point(endL[0],endL[0]))<=60:
                        d1=shortP.distance(Point(startS[0],startS[1]))
                        d2=longP.distance(Point(endL[0],endL[0]))
                        if math.fabs(d1-d2)/d1<=0.15:
                            if verbose==True:
                                print 'U shape'
                            # situation 1
                            new_group=reS[0:]+reL[0:]
                            new_pt=point_on_line(Point(startL[0],startL[1]), longP, (d1+d2)/2)
                            new_group.append((new_pt.x,new_pt.y))
                            new_pt=point_on_line(Point(endS[0],endS[1]), shortP, (d1+d2)/2)
                            new_group.append((new_pt.x,new_pt.y))
                            if longL[0]>shortL[0]:
                                groups_P.pop(longL[0])
                                groups_P.pop(shortL[0])
                            else:
                                groups_P.pop(shortL[0])
                                groups_P.pop(longL[0])
```

```
        groups_P.append(new_group)
    else:
        if verbose==True:
            print 'U(Z) shape'
        if d1<d2:
            ptProj=GetProjectivePoint(Point(startS[0],startS[1]), longL[2])
            new_group=reS[0:]+[(ptProj.x, ptProj.y)]+reL[0:]
            new_pt=point_on_line(Point(startL[0],startL[1]), longP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            new_pt=point_on_line(Point(endS[0],endS[1]), shortP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            if longL[0]>shortL[0]:
                groups_P.pop(longL[0])
                groups_P.pop(shortL[0])
            else:
                groups_P.pop(shortL[0])
                groups_P.pop(longL[0])
            groups_P.append(new_group)
        else:
            ptProj=GetProjectivePoint(Point(endL[0],endL[1]), shortL[2])
            new_group=reS[0:]+[(ptProj.x, ptProj.y)]+reL[0:]
            new_pt=point_on_line(Point(startL[0],startL[1]), longP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            new_pt=point_on_line(Point(endS[0],endS[1]), shortP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            if longL[0]>shortL[0]:
                groups_P.pop(longL[0])
                groups_P.pop(shortL[0])
            else:
                groups_P.pop(shortL[0])
                groups_P.pop(longL[0])
            groups_P.append(new_group)
elif longP.distance(Point(startL[0],startL[0]))<=60:
    # situation 2
    if verbose==True:
        print 'Z shape'
    ptProj0=GetProjectivePoint(Point(startS[0],startS[1]), longL[2])
    new_group=reS[0:]+[(ptProj0.x, ptProj0.y)]+reL[0:]
    ptProj1=GetProjectivePoint(Point(startL[0],startL[1]), shortL[2])
    new_group.append((ptProj1.x, ptProj1.y))
    if longL[0]>shortL[0]:
```

```
            groups_P.pop(longL[0])
            groups_P.pop(shortL[0])
        else:
            groups_P.pop(shortL[0])
            groups_P.pop(longL[0])
        groups_P.append(new_group)
    else:
        # situation 3
        if verbose==True:
            print '4 shape 1'
        ptProj0=GetProjectivePoint(Point(startS[0], startS[1]), longL[2])
        new_group=reS[0:]+[(ptProj0.x, ptProj0.y)]+reL[0:]
        d1=shortP.distance(Point(startS[0],startS[1]))
        new_pt=point_on_line(Point(startL[0],startL[1]), longP, d1)
        new_group.append((new_pt.x,new_pt.y))
        new_pt=point_on_line(Point(endS[0],endS[1]), shortP, d1)
        new_group.append((new_pt.x,new_pt.y))
        if longL[0]>shortL[0]:
            groups_P.pop(longL[0])
            groups_P.pop(shortL[0])
        else:
            groups_P.pop(shortL[0])
            groups_P.pop(longL[0])
        groups_P.append(new_group)
elif shortP.distance(Point(endS[0],endS[1]))<=60:
    if longP.distance(Point(startL[0],startL[0]))<=60:
        # situation 4
        d1=shortP.distance(Point(endS[0],endS[0]))
        d2=longP.distance(Point(startL[0],startL[1]))
        if math.fabs(d1-d2)/d1<=0.15:
            if verbose==True:
                print 'U shape'
            # situation 1
            new_group=reL[0:]+reS[0:]
            new_pt=point_on_line(Point(startS[0], startS[1]), shortP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            new_pt=point_on_line(Point(endL[0],endL[1]), longP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            if longL[0]>shortL[0]:
                groups_P.pop(longL[0])
                groups_P.pop(shortL[0])
            else:
                groups_P.pop(shortL[0])
                groups_P.pop(longL[0])
            groups_P.append(new_group)
```

```
    else:
        if verbose==True:
            print 'U(Z) shape'
        if d1<d2:
            ptProj=GetProjectivePoint(Point(endS[0],endS[1]), longL[2])
            new_group=reL[0:]+[(ptProj.x, ptProj.y)]+reS[0:]
            new_pt=point_on_line(Point(startS[0],startS[1]), shortP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            new_pt=point_on_line(Point(endL[0],endL[1]), longP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            if longL[0]>shortL[0]:
                groups_P.pop(longL[0])
                groups_P.pop(shortL[0])
            else:
                groups_P.pop(shortL[0])
                groups_P.pop(longL[0])
            groups_P.append(new_group)
        else:
            ptProj=GetProjectivePoint(Point(startL[0],startL[1]), shortL[2])
            new_group=reL[0:]+[(ptProj.x, ptProj.y)]+reS[0:]
            new_pt=point_on_line(Point(startS[0],startS[1]), shortP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            new_pt=point_on_line(Point(endL[0],endL[1]), longP, (d1+d2)/2)
            new_group.append((new_pt.x,new_pt.y))
            if longL[0]>shortL[0]:
                groups_P.pop(longL[0])
                groups_P.pop(shortL[0])
            else:
                groups_P.pop(shortL[0])
                groups_P.pop(longL[0])
            groups_P.append(new_group)
elif longP.distance(Point(endL[0],endL[0]))<=60:
    # situation 5
    if verbose==True:
        print 'Z shape'
    ptProj0=GetProjectivePoint(Point(endL[0],endL[1]), shortL[2])
    new_group=reS[0:]+[(ptProj0.x, ptProj0.y)]+reL[0:]
    ptProj1=GetProjectivePoint(Point(endS[0],endS[1]), longL[2])
    new_group.append((ptProj1.x, ptProj1.y))
    if longL[0]>shortL[0]:
        groups_P.pop(longL[0])
        groups_P.pop(shortL[0])
```

```
        else:
            groups_P.pop(shortL[0])
            groups_P.pop(longL[0])
        groups_P.append(new_group)
    else:
        # situation 6
        if verbose==True:
            print '4 shape 2'
        d1=shortP.distance(Point(endS[0],endS[1]))
        new_pt=point_on_line(Point(startS[0],startS[1]), shortP, d1)
        new_group=reS[0:]+[(new_pt.x,new_pt.y)]
        new_pt=point_on_line(Point(endL[0],endL[1]), longP, d1)
        new_group.append((new_pt.x,new_pt.y))
        new_group.extend(reL)
        ptProj=GetProjectivePoint(Point(endS[0],endS[1]), longL[2])
        new_group.append((ptProj.x, ptProj.y))
        if longL[0]>shortL[0]:
            groups_P.pop(longL[0])
            groups_P.pop(shortL[0])
        else:
            groups_P.pop(shortL[0])
            groups_P.pop(longL[0])
        groups_P.append(new_group)
else:
    if longP.distance(Point(startL[0],startL[0]))<=60:
        # situation 7
        if verbose==True:
            print '4 shape 3'
        d1=longP.distance(Point(startL[0],startL[1]))
        new_pt0=point_on_line(Point(startS[0],startS[1]), shortP, d1)
        new_pt1=point_on_line(Point(endL[0],endL[1]), longP, d1)
        new_group=reS[0:]+[(new_pt0.x,new_pt0.y),(new_pt1.x,new_pt1.y)]+reL[0:]
        ptProj=GetProjectivePoint(Point(startL[0],startL[1]), shortL[2])
        new_group.append((ptProj.x, ptProj.y))
        if longL[0]>shortL[0]:
            groups_P.pop(longL[0])
            groups_P.pop(shortL[0])
        else:
            groups_P.pop(shortL[0])
            groups_P.pop(longL[0])
        groups_P.append(new_group)
    elif longP.distance(Point(endL[0],endL[0]))<=60:
        # situation 8
        if verbose==True:
            print '4 shape 4'
```

```
                            d1=longP.distance(Point(endL[0],endL[1]))
                            ptProj=GetProjectivePoint(Point(endL[0],endL[1]), shortL[2])
                            new_group=reS[0:]+[(ptProj.x, ptProj.y)]+reL[0:]
                            new_pt0=point_on_line(Point(startL[0],startL[1]), longP, d1)
                            new_pt1=point_on_line(Point(endS[0],endS[1]), shortP, d1)
                            new_group.append((new_pt0.x,new_pt0.y))
                            new_group.append((new_pt1.x,new_pt1.y))
                            if longL[0]>shortL[0]:
                                groups_P.pop(longL[0])
                                groups_P.pop(shortL[0])
                            else:
                                groups_P.pop(shortL[0])
                                groups_P.pop(longL[0])

                            groups_P.append(new_group)
                        else:
                            # situation 9
                            if verbose==True:
                                'H shape'
                            new_pt=point_on_line(Point(startS[0],startS[1]), shortP, width/2)
                            new_group=reS[0:]+[(new_pt.x,new_pt.y)]
                            new_pt=point_on_line(Point(endL[0],endL[1]), longP, width/2)
                            new_group.append((new_pt.x,new_pt.y))
                            new_group.extend(reL)
                            new_pt=point_on_line(Point(startL[0],startL[1]), longP, width/2)
                            new_group.append((new_pt.x,new_pt.y))
                            new_pt=point_on_line(Point(endS[0],endS[1]), shortP, width/2)
                            new_group.append((new_pt.x,new_pt.y))
                            if longL[0]>shortL[0]:
                                groups_P.pop(longL[0])
                                groups_P.pop(shortL[0])
                            else:
                                groups_P.pop(shortL[0])
                                groups_P.pop(longL[0])

                            groups_P.append(new_group)
                else:
                    if verbose==True:
                        print 'not self-closed','situation 3'
                    longGroup=groups_P[longL[0]]
                    reL=longGroup[longL[1]+1:]+longGroup[0:longL[1]+1]
                    shortGroup=groups_P[shortL[0]]
                    reS=shortGroup[shortL[1]+1:]+shortGroup[0:shortL[1]+1]
```

```
                    ptProj0=GetProjectivePoint(Point(shortGroup[shortL[1]][0], shortGroup[shortL[1]][1]), longL[2])
                    if shortL[1]+1>=len(shortGroup):
                        ptProj1=GetProjectivePoint(Point(shortGroup[shortL[1]+1-len(shortGroup)][0], shortGroup[shortL[1]+1-len(shortGroup)][1]), longL[2])
                    else:
                        ptProj1=GetProjectivePoint(Point(shortGroup[shortL[1]+1][0], shortGroup[shortL[1]+1][1]), longL[2])
                    if longL[0]>shortL[0]:
                        groups_P.pop(longL[0])
                        groups_P.pop(shortL[0])
                    else:
                        groups_P.pop(shortL[0])
                        groups_P.pop(longL[0])
                    groups_P.append(reS+[(ptProj0.x, ptProj0.y)]+reL+[(ptProj1.x, ptProj1.y)])
    new_groups_P0=[]
    for i in range(0, len(groups_P)):
        new_group=[]
        for j in range(0, len(groups_P[i])):
            if j==0:
                dv1_x = groups_P[i][-1][0] - groups_P[i][j][0]
                dv1_y = groups_P[i][-1][1] - groups_P[i][j][1]
                dv2_x = groups_P[i][j+1][0] - groups_P[i][j][0]
                dv2_y = groups_P[i][j+1][1] - groups_P[i][j][1]
            elif j==len(groups_P[i])-1:
                dv1_x = groups_P[i][j-1][0] - groups_P[i][j][0]
                dv1_y = groups_P[i][j-1][1] - groups_P[i][j][1]
                dv2_x = groups_P[i][0][0] - groups_P[i][j][0]
                dv2_y = groups_P[i][0][1] - groups_P[i][j][1]
            else:
                dv1_x = groups_P[i][j-1][0] - groups_P[i][j][0]
                dv1_y = groups_P[i][j-1][1] - groups_P[i][j][1]
                dv2_x = groups_P[i][j+1][0] - groups_P[i][j][0]
                dv2_y = groups_P[i][j+1][1] - groups_P[i][j][1]
            dv1xdv2 = dv1_x * dv2_x + dv1_y * dv2_y
            absdv1 = math.sqrt(dv1_x * dv1_x + dv1_y * dv1_y)
            absdv2 = math.sqrt(dv2_x * dv2_x + dv2_y * dv2_y)
            if absdv1==0:
                if j<=1:
                    dv1_x = groups_P[i][j-2+len(groups_P[i])][0] - groups_P[i][j][0]
                    dv1_y = groups_P[i][j-2+len(groups_P[i])][1] - groups_P[i][j][1]
                else:
                    dv1_x = groups_P[i][j-2][0] - groups_P[i][j][0]
                    dv1_y = groups_P[i][j-2][1] - groups_P[i][j][1]
                dv1xdv2 = dv1_x * dv2_x + dv1_y * dv2_y
                absdv1 = math.sqrt(dv1_x * dv1_x + dv1_y * dv1_y)
                absdv2 = math.sqrt(dv2_x * dv2_x + dv2_y * dv2_y)
```

```
                angle = math.degrees(math.acos(dv1xdv2 / (absdv1 * absdv2)))
                if math.fabs(angle)>=3 and math.fabs(angle)<=177:
                    new_group.append((groups_P[i][j][0], groups_P[i][j][1]))
            elif absdv2==0:
                continue
            else:
                if dv1xdv2 / (absdv1 * absdv2)>1:
                    angle=math.degrees(math.acos(1))
                elif dv1xdv2 / (absdv1 * absdv2)<-1:
                    angle=math.degrees(math.acos(-1))
                else:
                    angle = math.degrees(math.acos(dv1xdv2 / (absdv1 * absdv2)))
                if math.fabs(angle)>=3 and math.fabs(angle)<=177:
                    new_group.append((groups_P[i][j][0], groups_P[i][j][1]))
        new_groups_P0.append(new_group)
        new_groups_P=[]
        for i in range(0, len(new_groups_P0)):
            if len(new_groups_P0[i])>2:
                new_groups_P.append(list(Polygon(new_groups_P0[i]).buffer(0).exterior.coords)

    return new_groups_P, Nodes
```

A-4 LineGroupingFromSHP.py

```
import math
import shapefile

from shapely.geometry import Point
from shapely.geometry import LineString
from shapely.geometry import Polygon
from shapely.geometry import polygon

from fix_drafting_errors import fix_disjoint_vertices


def LineGroupingFromSHP(abs_file_path, MINIMALDIST):


    sf = shapefile.Reader(abs_file_path)

#---------read lines from shapefile---------
    chains=[]
    for geom in sf.shapeRecords():
        chain=[(geom.shape.points[0][0],geom.shape.points[0][1]),(geom.shape.points[1][0],geom.shape.points[1][1])]
        chains.append(chain)
#------------------------------------------
```

```

#---------group lines & fix unconnected vertices---------
    closed_chains=[]
    k=1
    RADIUS=MINIMALDIST

    print '#----------------------------'
    print 'k=', k, 'RADIUS=', RADIUS
    print 'len(chains)', len(chains)
    print 'len(closed_chains)', len(closed_chains)

    while (k<=5 and len(chains)>0):
        if len(chains)==1:
            pt11=Point(chains[0][0][0], chains[0][0][1])
            pt12=Point(chains[0][-1][0],chains[0][-1][1])

            if pt11.distance(pt12)<=RADIUS:
                chains.pop(0)
                l1=LineString([chain1[0], chain1[1]])
                l2=LineString([chain1[-1], chain1[-2]])
                new_pts1=fix_disjoint_vertices(l1, l2)
                new_chain=chain1[1:-1]+new_pts1
                closed_chains.append(new_chain)
                break
            else:
                k=k+1
                RADIUS=RADIUS+MINIMALDIST
                print '#----------------------------'
                print 'k=', k, 'RADIUS=', RADIUS
                print 'len(chains)', len(chains)
                print 'len(closed_chains)', len(closed_chains)
                break
        L=len(chains)
        for i in range(0, len(chains)-1):
            chain1=chains[i]
            pt11=Point(chain1[0][0], chain1[0][1])
            pt12=Point(chain1[-1][0],chain1[-1][1])
            if pt11.distance(pt12)<=RADIUS:
                chains.pop(i)
                l1=LineString([chain1[0], chain1[1]])
                l2=LineString([chain1[-1], chain1[-2]])
                new_pts1=fix_disjoint_vertices(l1, l2)
                new_chain=chain1[1:-1]+new_pts1
                closed_chains.append(new_chain)
                break
            for j in range(i+1, len(chains)+1):
                if j==len(chains):
                    break
                chain2=chains[j]
                pt21=Point(chain2[0][0], chain2[0][1])
                pt22=Point(chain2[-1][0],chain2[-1][1])
                if pt11.distance(pt21)<=RADIUS:
                    l1=LineString([chain1[0], chain1[1]])
```

```
                    l2=LineString([chain2[0], chain2[1]])
                    new_pts1=fix_disjoint_vertices(l1, l2)
                    if pt12.distance(pt22)<=RADIUS:
                        # closed
                        l1=LineString([chain1[-1], chain1[-2]])
                        l2=LineString([chain2[-1], chain2[-2]])
                        new_pts2=fix_disjoint_vertices(l1, l2)

                        chains.pop(j)
                        chains.pop(i)
                        chain1.reverse()
                        new_chain=new_pts2+chain1[1:-1]+new_pts1+chain2[1:-1]
                        closed_chains.append(new_chain)
                        break
                    else:
                        chains.pop(j)
                        chains.pop(i)
                        chain1.reverse()
                        new_chain=chain1[0:-1]+new_pts1+chain2[1:]
                        chains.append(new_chain)
                        break
                elif pt11.distance(pt22)<=RADIUS:
                    l1=LineString([chain1[0], chain1[1]])
                    l2=LineString([chain2[-1], chain2[-2]])
                    new_pts1=fix_disjoint_vertices(l1, l2)

                    if pt12.distance(pt21)<=RADIUS:
                        # closed
                        l1=LineString([chain1[-1], chain1[-2]])
                        l2=LineString([chain2[0], chain2[1]])
                        new_pts2=fix_disjoint_vertices(l1, l2)

                        chains.pop(j)
                        chains.pop(i)
                        new_chain=new_pts1+chain1[1:-1]+new_pts2+chain2[1:-1]
                        closed_chains.append(new_chain)
                        break
                    else:
                        chains.pop(j)
                        chains.pop(i)
                        new_chain=chain2[0:-1]+new_pts1+chain1[1:]
                        chains.append(new_chain)
                        break
                elif pt12.distance(pt21)<=RADIUS:
                    l1=LineString([chain1[-1], chain1[-2]])
                    l2=LineString([chain2[0], chain2[1]])
                    new_pts1=fix_disjoint_vertices(l1, l2)
                    chains.pop(j)
                    chains.pop(i)
                    new_chain=chain1[0:-1]+new_pts1+chain2[1:]
                    chains.append(new_chain)
```

```
                    break
                elif pt12.distance(pt22)<=RADIUS:
                    l1=LineString([chain1[-1], chain1[-2]])
                    l2=LineString([chain2[-1], chain2[-2]])
                    new_pts1=fix_disjoint_vertices(l1, l2)
                    chains.pop(j)
                    chains.pop(i)
                    chain2.reverse()
                    new_chain=chain1[0:-1]+new_pts1+chain2[1:]
                    chains.append(new_chain)
                    break
                else:
                    continue
            if j==L:
                continue
            else:
                break
        if i==L-2 and j==L:
            k=k+1
            RADIUS=RADIUS+MINIMALDIST
            print '#----------------------------'
            print 'k=', k, 'RADIUS=', RADIUS
            print 'len(chains)', len(chains)
            print 'len(closed_chains)', len(closed_chains)
        else:
            continue
    print '#----------------------------'
    print 'len(chains)', len(chains)
    print 'len(closed_chains)', len(closed_chains)
#-------------------------------------------

#-------------------------------------------
    groups_P=[]
    for i in range(0, len(closed_chains)):
        if len(closed_chains[i])>2:
            if Polygon(closed_chains[i]).is_valid==True:
                ply=Polygon(closed_chains[i])
                new_ply=polygon.orient(ply, sign=1.0)
                groups_P.append(list(new_ply.exterior.coords)[0:-1])
            else:
                ply=Polygon(closed_chains[i]).buffer(0).exterior.coords
                new_ply=polygon.orient(ply, sign=1.0)
                groups_P.append(list(new_ply.exterior.coords)[0:-1])
        else:
            print 'closed_chains ',i,' only has two points'
    return groups_P
#
    ------------------------------------------------------------------------------
```

A-5 FixDraftingErrors.py

```
from shapely.geometry import LineString
from shapely.geometry import Point
from angle_of_line import angle_of_line
import math
import time
import fiona
from collections import OrderedDict


def find_intersectingPoint(l1, l2):
    # find intersecting point of two unparallel lines
    x11=list(l1.coords)[0][0]
    y11=list(l1.coords)[0][1]
    x12=list(l1.coords)[1][0]
    y12=list(l1.coords)[1][1]

    x21=list(l2.coords)[0][0]
    y21=list(l2.coords)[0][1]
    x22=list(l2.coords)[1][0]
    y22=list(l2.coords)[1][1]

    A1=y12-y11
    B1=x11-x12
    C1=x12*y11-x11*y12

    A2=y22-y21
    B2=x21-x22
    C2=x22*y21-x21*y22

    x0=(-1)*(B2*C1-B1*C2)/(A1*B2-A2*B1)
    y0=(-1)*(A2*C1-A1*C2)/(A2*B1-A1*B2)

    return Point(x0, y0)

def fix_disjoint_vertices(l1, l2):
    a1=angle_of_line(l1)
    a2=angle_of_line(l2)
    if math.fabs(a1-a2)<=2:
        return []

    else:
        if l1.intersects(l2)==True:
            pt=l1.intersection(l2)
            return [(pt.x, pt.y)]

        else:
            pt=find_intersectingPoint(l1, l2)
            return [(pt.x, pt.y)]



def fix_duplicated_lines(lines, MINIMALDIST):

```

```

    new_lines=[]
    while len(lines)>0:
        print len(lines),len(new_lines)

        l1=lines[0]

        if l1.length<MINIMALDIST:
            print 'Null-length'
            lines.pop(0)
            continue

        if len(lines)==0:
            break
        elif len(lines)==1:
            new_lines.append(l1)
            lines.pop(0)
            break
        else:

            for i in range(1, len(lines)+1):

                if i==len(lines):
                    new_lines.append(l1)
                    lines.pop(0)
                    break

                l2=lines[i]
                if l2.length<MINIMALDIST:
                    print 'Null-length'
                    lines.pop(i)
                    break
                a1=angle_of_line(l1)
                a2=angle_of_line(l2)
                if math.fabs(a1-a2)<=2:
                    bff=l1.buffer(MINIMALDIST, resolution=16, cap_style
                        =2)
                    if bff.intersects(l2)==True:
                        if bff.exterior.intersection(l2).geom_type=='
                            GeometryCollection':
                            # contains
                            lines.pop(i)
                            print 'Contains'
                            break
                        elif bff.exterior.intersection(l2).geom_type=='
                            Point':
                            # overlapped or consecutive
                            pt11=Point(list(l1.coords)[0])
                            pt12=Point(list(l1.coords)[1])
                            pt21=Point(list(l2.coords)[0])
                            pt22=Point(list(l2.coords)[1])
                            if pt21.intersects(bff)==True:
```

```
                                if pt22.distance(pt11)>=pt22.distance(pt12):
                                    nl=LineString([(pt11.x, pt11.y), (pt22.x, pt22.y)])
                                else:
                                    nl=LineString([(pt12.x, pt12.y), (pt22.x, pt22.y)])
                            else:
                                if pt21.distance(pt11)>=pt21.distance(pt12):
                                    nl=LineString([(pt11.x, pt11.y), (pt21.x, pt21.y)])
                                else:
                                    nl=LineString([(pt12.x, pt12.y), (pt21.x, pt21.y)])
                            lines.pop(i)
                            lines.pop(0)
                            lines.append(nl)
                            print 'Overlapped or Consecutive'
                            break
                        elif bff.exterior.intersection(l2).geom_type=='MultiPoint':
                            # contained
                            lines.pop(0)
                            print 'Contained'
                            break


    return new_lines
```

Bibliography

[1] C. S. Jensen, K.-J. Li, and S. Winter, "The other 87%: A report on the second international workshop on indoor spatial awareness (san jose, california-november 2, 2010)," *SIGSPATIAL Special*, vol. 3, no. 1, pp. 10–12, 2011.

[2] S. Winter, "Indoor spatial information," *International Journal of 3-D Information Modeling (IJ3DIM)*, vol. 1, no. 1, pp. 25–42, 2012.

[3] M. Goetz, "Towards generating highly detailed 3d citygml models from openstreetmap," *International Journal of Geographical Information Science*, vol. 27, no. 5, pp. 845–865, 2013.

[4] Bing, "Bing maps venue maps now feature nine largest us malls, 148 total." http://www.bing.com/community/site_blogs/b/maps/archive/2011/03/22/bing-maps-venue-maps-now-feature-largest-nine-us-malls-148-total.aspx, 2011. [Online; accessed 18-Apr-2015].

[5] Google, "Go indoors with google maps 6.0 for android." http://googlemobile.blogspot.de/2011/11/go-indoors-with-googlemaps-60-for.html, 2011. [Online; accessed 18-Apr-2015].

[6] Navteq, "Navteq extends the journey beyond theąőfront doorąŕ." http://www.prnewswire.com/news-releases/navteqextends-the-journey-beyond-the-front-door-118353959.html, 2011. [Online; accessed 18-Apr-2015].

[7] J. Baus, C. Kray, and A. Krüger, "Visualization of route descriptions in a resource-adaptive navigation aid," *Cognitive Processing*, vol. 2, no. 2-3, pp. 323–345, 2001.

[8] V. Coors and A. Zipf, "Mona 3d–mobile navigation using 3d city models," *LBS and Telecartography*, 2007.

[9] R. Héno and L. Chandelier, *3D Modeling of Buildings: Outstanding Sites*. John Wiley & Sons, 2014.

[10] S. Zlatanova, "Working group iiąłacquisitionąłposition paper: Data collection and 3d reconstruction," in *Advances in 3D Geoinformation Systems*, pp. 425–428, Springer, 2008.

[11] E. M. Mikhail, J. S. Bethel, and J. C. McGlone, *Introduction to modern photogrammetry*, vol. 1. John Wiley & Sons Inc, 2001.

[12] A. Gruen, E. Baltsavias, and O. Henricsson, *Automatic extraction of man-made objects from aerial and space images (II)*. Birkhäuser, 2012.

[13] E. Schwalbe, H.-G. Maas, and F. Seidel, "3d building model generation from airborne laser scanner data using 2d gis data and orthogonal point cloud projections," *Proceedings of ISPRS WG III/3, III/4*, vol. 3, pp. 12–14, 2005.

[14] E. Schwalbe, "3d building model generation from airborne laserscanner data by straight line detection in specific orthogonal projections," *International Archives of Photogrammetry and Remote Sensing*, vol. 35, no. 3, pp. 249–254, 2004.

[15] G. Vosselman, S. Dijkman, *et al.*, "3d building model reconstruction from point clouds and ground plans," *International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences*, vol. 34, no. 3/W4, pp. 37–44, 2001.

[16] I. Suveg and G. Vosselman, "Reconstruction of 3d building models from aerial images and maps," *ISPRS Journal of Photogrammetry and remote sensing*, vol. 58, no. 3, pp. 202–224, 2004.

[17] V. Verma, R. Kumar, and S. Hsu, "3d building detection and modeling from aerial lidar data," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 2, pp. 2213–2220, IEEE, 2006.

[18] S.-H. Or, K.-H. Wong, Y.-k. Yu, M. M. Chang, and H. Kong, "Highly automatic approach to architectural floorplan image understanding & model generation," *Pattern Recognition*, 2005.

[19] L. Liu and S. Zlatanova, "Generating navigation models from existing building data," in *Acquisition and Modelling of Indoor and Enclosed Environments 2013, Cape Town, South Africa, 11-13 December 2013, ISPRS Archives Volume XL-4/W4, 2013*, ISPRS, 2013.

[20] X. Yin, P. Wonka, and A. Razdan, "Generating 3d building models from architectural drawings: A survey," *IEEE Computer Graphics and Applications*, no. 1, pp. 20–30, 2009.

[21] Wikipedia, "Computer-aided design." https://en.wikipedia.org/wiki/Computer-aided_design, 2015. [Online; accessed 23-August-2015].

[22] AutoDesk, "What is cad software?." http://www.autodesk.com/solutions/cad-software, 2015. [Online; accessed 23-August-2015].

[23] Wikipedia, "Category:cad file formats." https://en.wikipedia.org/wiki/Category:CAD_file_formats, 2015. [Online; accessed 23-August-2015].

[24] U. of Minnesota, "Interior design student handbook, part 2: Basic drafting standards and symbols." http://www.slideshare.net/supergirlanchal/interior-design-student-handbook, 2005. [Online; accessed 23-August-2015].

[25] W. P. Spence, *Architectural working drawings: Residential and commercial buildings.* John Wiley & Sons, 1993.

[26] S. A. Katherine, *Interior construction document.* Fairchild Books, 2004.

[27] R. Kilmer and W. O. Kilmer, *Construction drawings and details for interiors: Basic skills.* John Wiley & Sons, 2011.

[28] T. Guo, H. Zhang, and Y. Wen, "An improved example-driven symbol recognition approach in engineering drawings," *Computers & Graphics*, vol. 36, no. 7, pp. 835–845, 2012.

[29] G. Zhi, S. Lo, and Z. Fang, "A graph-based algorithm for extracting units and loops from architectural floor plans for a building evacuation model," *Computer-Aided Design*, vol. 35, no. 1, pp. 1–14, 2003.

[30] R. Lewis and C. Séquin, "Generation of 3d building models from 2d architectural plans," *Computer-Aided Design*, vol. 30, no. 10, pp. 765–779, 1998.

[31] L. Yan and L. Wenyin, "Engineering drawings recognition using a case-based approach," in *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*, pp. 190–194, IEEE, 2003.

[32] C. Ah-Soon and K. Tombre, "Architectural symbol recognition using a network of constraints," *Pattern Recognition Letters*, vol. 22, no. 2, pp. 231–248, 2001.

[33] E. Valveny, M. Delalandre, R. Raveaux, and B. Lamiroy, "Report on the symbol recognition and spotting contest," in *Graphics Recognition. New Trends and Challenges*, pp. 198–207, Springer, 2013.

[34] B. Domínguez, Á. García, and F. R. Feito, "Semiautomatic detection of floor topology from cad architectural drawings," *Computer-Aided Design*, vol. 44, no. 5, pp. 367–378, 2012.

[35] T. Lu, H. Yang, R. Yang, and S. Cai, "Automatic analysis and integration of architectural drawings," *International Journal of Document Analysis and Recognition (IJDAR)*, vol. 9, no. 1, pp. 31–47, 2007.

[36] J. Zhu, H. Zhang, and Y. Wen, "A new reconstruction method for 3d buildings from 2d vector floor plan," *Computer-Aided Design and Applications*, vol. 11, no. 6, pp. 704–714, 2014.

[37] M. Anderson, "Global positioning tech inspires do-it-yourself mapping project," *National Geographic News*, 2006.

[38] M. Goetz, "Using crowdsourced indoor geodata for the creation of a three-dimensional indoor routing web application," *Future Internet*, vol. 4, no. 2, pp. 575–591, 2012.

[39] M. Goetz and A. Zipf, "Using crowdsourced geodata for agent-based indoor evacuation simulations," *ISPRS International Journal of Geo-Information*, vol. 1, no. 2, pp. 186–208, 2012.

[40] M. Uden and A. Zipf, "Open building models: towards a platform for crowdsourcing virtual 3d cities," in *Progress and New Trends in 3D Geoinformation Sciences*, pp. 299–314, Springer, 2013.