UNIVERSIDAD DE JAÉN
ESCUELA POLITÉCNICA SUPERIOR
DE JAÉN
DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL

METHODS TO PROCESS LOW-LEVEL CAD PLANS AND CREATE BUILDING INFORMATION MODELS (BIM)

PRESENTADA POR:
BERNARDINO DOMÍNGUEZ MARTÍN

DIRIGIDA POR:
DR. D. FRANCISCO RAMÓN FEITO HIGUERUELA
DR. D. ÁNGEL LUIS GARCÍA FERNÁNDEZ

JAÉN, 17 DE DICIEMBRE DE 2014

ISBN 978-84-8439-957-5

Contents

Contents . . . iii
List of Figures . . . vii
List of Algorithms . . . xi
List of Tables . . . xiii

I Introduction 1

1 Introduction 5
1.1 Motivation . . . 6
1.2 Algorithms to obtain semantic contents . . . 7
1.3 Semantic representation of buildings . . . 10
1.4 Implementation of a user tool . . . 11

2 Previous work 13
2.1 Automatic reconstruction . . . 13
2.2 Building Information Model (BIM) . . . 16
2.2.1 Definition and history of Building Information Modeling . 16
2.2.2 Implantation of Building Information Modeling . . . 17
2.3 Geographic Information Systems . . . 17
2.4 BIM and GIS standards . . . 18
2.4.1 IFC . . . 18
2.4.2 CityGML . . . 19
2.5 Other models from the literature . . . 20
2.5.1 2D models . . . 21
2.5.2 2D models with height . . . 25
2.5.3 3D models . . . 26

II Detection of semantic elements 33

3 Architectural design. Constraints 37
3.1 Architectural drawings. File formats . . . 38
3.2 CAD standards. Floor plan structure requirements . . . 42
3.3 Local vs. global . . . 43

4 Irregularities **45**
4.1 Irregularity types . . . 46
4.2 Range comparison between real numbers . . . 47
4.3 Irregularity detection . . . 48
4.4 Irregularity fixing . . . 49

5 Local room detection **51**
5.1 Constraints on the DXF file format . . . 53
5.2 Feature extraction process . . . 53
5.2.1 Wall thickness calculation . . . 53
5.2.2 Key point, room and window extraction. Detection rules 54
5.2.3 Data storage . . . 56
5.3 3D output generation . . . 56
5.4 Results . . . 57

6 Global room detection **61**
6.1 Wall detection . . . 63
6.1.1 Wall Adjacency Graph (WAG) . . . 66
6.1.1.1 Properties . . . 73
6.1.1.2 Generalized Wall Adjacency Graph (GWAG) . . 74
6.1.1.3 Handling multiple parallel, close enough segments 75
6.1.1.4 Generalizing the wall detection process . . . 76
6.1.1.5 WAG and GWAG for curved walls . . . 78
6.2 Opening detection . . . 78
6.2.1 Opening detection from inserts . . . 79
6.2.2 Opening detection from primitives . . . 85
6.3 Clustering . . . 87
6.3.1 Line growing algorithm . . . 89
6.4 Results . . . 91
6.4.1 Wall detection . . . 92
6.4.2 Topology graph construction . . . 94

III Representation of building information **103**

7 A three-level framework for correct models **107**
7.1 Comparative analysis and discussion . . . 107
7.1.1 Geometry . . . 109
7.1.2 Topology . . . 109
7.1.3 Semantics . . . 109
7.2 Geometry module . . . 111
7.3 Semantics module . . . 113
7.4 Topology module . . . 113

8 2D topology. CAD floor plan processing 115
8.1 Construction of the topology graph 115
8.2 Linking geometric and topological information 120
8.2.1 Segment to closed space assignment 120
8.2.2 Closing inner polygons 121
8.3 Results 124

9 3D geometry. Triple extrusion 125
9.1 Building a three-view DXF 126
9.2 DXF importing. Spatial indexes 127
9.3 Contour detection of the views 133
9.4 Contour polygon triangulation 135
9.5 2D to 3D coordinate transformation 135
9.6 Extrusion of the views 136
9.7 Results 137

10 3D topology and semantics. CityGML 139
10.1 Results 141

IV Conclusions and future work 143

11 Conclusions and future work 147
11.1 Conclusions 147
11.2 Future work 149
11.2.1 Wall detection algorithm 149
11.2.2 Opening detection 149
11.2.3 Clustering. Construction of the topology graph 149
11.2.4 Three-level framework of topologically correct models . . 150
11.2.5 Triple extrusion 150
11.2.6 Other future work 150

Appendices 151

A Analysis, design and implementation of an end-user tool 153
A.1 Application features 154
A.2 Application Analysis 155
A.2.1 Task analysis 155
A.2.2 Task structure 157
A.2.3 Interaction architecture 157
A.2.4 Client-server architecture 160
A.2.5 Database structure 161
A.3 User interface design 162
A.3.1 Screen design 162
A.3.2 Model design 163
A.3.3 Design patterns 164

A.3.3.1 Model-view-controller pattern 164
A.3.3.2 Observer pattern 165
A.3.3.3 Strategy pattern 165
A.3.3.4 Command pattern 165
A.3.3.5 State pattern 166
A.3.4 UML class diagram 166
A.3.4.1 MVC and observer pattern diagrams 166
A.3.4.2 Command pattern diagram 168
A.3.4.3 Model architecture diagram 169
A.3.4.4 State pattern diagram 169
A.4 Implementation 170

B Evolution of clustering algorithms 171
B.1 K-means 171
B.2 DBSCAN algorithm 172
B.3 Brute force algorithm 174
B.4 Variation on brute force algorithm 176
B.5 Smart centroid detection 177
B.6 Clustering avoiding same edge 178
B.7 Clustering comparing edges 182
B.8 Clustering comparing sequences 182
B.9 Variation on clustering comparing sequences 185

C Algorithms from computational geometry used in this dissertation 187
C.1 Intersection between primitives 187
C.2 Distance between segments 190
C.3 Triangulation 194

D Publications related to this work 199
D.1 Conference proceedings 199
D.2 Book chapters 199
D.3 Journals 200
D.3.1 Published 200
D.3.2 Submitted 200

Bibliography 201

List of Figures

2.1 IFC Architecture 19
2.2 Graph representing accesibility between convex cells 22
2.3 Route analysis using grid graphs 24
2.4 Building data model for the structured floor plan 27
2.5 Thematic and routing model 29
2.6 Graph models for BPM 29
2.7 Room connectivity graph 31

3.1 DXF file structure 41
3.2 Walls without/with thickness 43

4.1 Irregularity cases 46
4.2 Simplified irregularity cases 47

5.1 Types of key points considered in the system 52
5.2 Thickness calculation situations for walls 54
5.3 Situations where the detection rules are applied (1) 55
5.4 Situations where the detection rules are applied (2) 55
5.5 A floor plan used in our tests 58
5.6 Floor plan and corresponding 3D model 59

6.1 Mapping between pairs of segments and walls 63
6.2 Iteration of the wall detection algorithm 65
6.3 Set of line segments and its associated WAG 68
6.4 Layouts of wall-prone pairs of line segments 70
6.5 Segment splitting for each layout 70
6.6 Example of one step of the wall detection algorithm 72
6.7 Behavior of the wall detection algorithm for three parallel segments 74
6.8 Behavior of the wall detection algorithm for narrow spaces 76
6.9 Processing three parallel segments using GWAG 77
6.10 Detection of curved walls using WAG 79
6.11 Bounding box of a door 80
6.12 Example of an oriented bounding box 80
6.13 Example of the opening detection algorithm 81

6.14 Anchorage correctness using a distance criterion 82
6.15 Longer direction of a door 83
6.16 Example of the line growing algorithm 90
6.17 Example of distance matrices in the line growing algorithm . . . 90
6.18 Line growing algorithm applied to generic cases 92
6.19 Real floor plans used to test our algorithms, together with the results of applying the wall detection algorithm. Detected walls are drawn in blue. The layers containing windows and doors are shown, but have not been processed 93
6.20 Topology representation from a portion of a CAD vector floor plan 94
6.21 Topology representation on top of the original CAD vector floor plan 95
6.22 Result of processing the floor plan of the A3 building 96
6.23 Result of processing the floor plan of the C5 building 97
6.24 Floor plan and CityGML model of *Basic lift* 98
6.25 Floor plan and CityGML model of *Planta tipo* 99
6.26 Floor plan and CityGML model of *Vilches* 100
6.27 Floor plan and CityGML model of *Bayenga* 101
6.28 Floor plan and CityGML model of *Heating* 102

7.1 Three-level architecture for building models 112

8.1 Intersecting walls 116
8.2 Existence of dangling edges in the topology graph 118
8.3 MAFL behavior with dangling edges and bridges 119
8.4 Segment to closed space assignment 121
8.5 Segments assigned to two adjacent rooms 122
8.6 Example of the grid algorithm 123
8.7 Detection of exterior and interior polygons 124

9.1 Example of the triple extrusion 126
9.2 Grid spatial index 130
9.3 Correspondence between 2D and 3D axes 136
9.4 Extrusion of the views 137
9.5 Result of the triple extrusion 137

10.1 UML of the linking between our model and CityGML 141
10.2 CityGML model of the building 142

A.1 State diagram 161
A.2 Task flow of the proposed system 162
A.3 Simplified view of the structure of the database 163
A.4 Application main screen 164
A.5 MVC. Observer pattern 167
A.6 Command pattern 168
A.7 Model 169

A.8 Application state diagram 170

B.1 Misassignment of two end points to the same cluster 177
B.2 Incorrect calculation of cluster centroids 177
B.3 Wall edges non aligned in the topology graph 181

C.1 Cases for the calculation of the distance between two segments . 195

List of Algorithms

6.1 Wall detection algorithm 69
6.2 Anchor opening to topology graph 83
6.3 Compute the OBB of an insert 84
6.4 Find the anchorage points of an OBB in a topology graph 85
6.5 Detect openings from a set of entities 86
6.6 Clustering using the end points of sequences 89
6.7 Line growing algorithm 91

8.1 Remove intersecting walls 116
8.2 Simplified version of the MAFL algorithm 117

9.1 Compute the cells traversed by a point 129
9.2 Basic insertion of a segment in a cell 131
9.3 Insert of a segment in the grid spatial index 132
9.4 Calculation of the parameter λ of a point in a segment 133
9.5 Remove a segment from the grid 134

B.1 K-means algorithm 171
B.2 DBSCAN clustering 173
B.3 Region query for DBSCAN clustering algorithm 173
B.4 Expand cluster for DBSCAN clustering algorithm 174
B.5 Brute force clustering algorithm 175
B.6 Second version of brute force clustering algorithm 176
B.7 Clustering algorithm avoiding points from the same edge to be in the same cluster 179
B.8 Computation of the smart centroid in a clustering algorithm . . . 180
B.9 Clustering algorithm comparing edges 183
B.10 Clustering algorithm comparing connected components 184
B.11 Second version of clustering algorithm comparing connected components 186

C.1 Detection of the intersection between two primitives 188
C.2 Detection of the intersection between two segments 189
C.3 Detection of the intersection between two intervals 190

C.4 Detection of the intersection between a segment and an arc 190
C.5 Detection of the intersection between two arcs 191
C.6 Triangulate a polygon defined by a sorted set of vertices 196
C.7 Check if two vertices of a polygon make up a diagonal 197
C.8 Check if a segment is contained in the cone made up by its two adjacent segments 197

List of Tables

3.1 List of primitives in 2D CAD design and their attributes. 39

4.1 Comparison between two intervals of real numbers 48
4.2 Comparison of numbers and intervals for arcs 50

6.1 WAG modifications for each layout 71
6.2 Comparison of the criteria to form wall-prone pairs of straight and circular arcs 79
6.3 Numerical data from the wall detection tests with real floor plans 92
6.4 Results of tests on real buildings 96

7.1 Comparison between the overviewed works. 110

A.1 Summary of the application states 160

B.1 Intersection cases and solutions. 178

Part I

Introduction

In this part we introduce the work of this dissertation. We will introduce the motivation of the research, and give a summary of the different parts and chapters the dissertation is divided into. Furthermore, a study of the most important previous works will be given.

This part is structured as follows: Chapter 1 contains the motivation and the summary of the rest of the document, while Chapter 2 analyzes the previous works on automatic reconstruction and building models.

Chapter 1

Introduction

It is remarkable the increasing interest of the computer users on tools related to geographic information, both in the user scope and the developer scope.

On one side, in the user scope, there are many technologies that have become very popular recently: the use of websites to access to maps, plan travel routes or view pictures of remote places, or the use of GPS navigators to guide travelers. More recently, both technologies, together with social networks, are combined into apps for smartphones or tablets that can be used to communicate the current position to other users, access maps from mobile devices or use the GPS feature to reach a destination by car or walking.

On the other side, in the developer scope, the most widespread map services like Google Maps or Microsoft Bing allow websites to include cartographic information by means of embedding frames or creating custom made web applications with their provided API's. In this sense, it is very common that websites including information about the placement of a shop, the itinerary of a sport race or the indications to reach a destination can take advantage of these services instead of implementing their own maps.

Furthermore, for urban environments, the map servers can be completed with 3D building models. Companies like Google or Microsoft (before November 2010[1]) offer to the users the tools and resources needed to populate the map systems with 3D building models. Apart from the impact of this facility as a strategic business model, it points out the high interest of users in the urban model visualization.

Some conclusions can be drawn from this fact: on one side, the importance for generated models not to be always of the highest level of detail (the above mentioned collaborative tools usually limit the size of models uploaded by users, therefore the users are required to use techniques like texture mapping or geometry simplification). On the other side, we have to remark that the existing tools are not conceived for simulating pedestrian walks, but for aerial views.

[1] In November 2010, Bing removed 3D buildings in order to focus on Bird's eye (*http://www.gearthblog.com/blog/archives/*2010/11/*bing_maps_is_dropping_their_3d_vers.html*. Retrieved 2014-07-30)

Moreover, we do not have evidence that they include the feature of entering inside the buildings.

This trend reveals the user interest on 3D urban information and involves the creation of specialized technologies for urban modeling, such as GML or CityGML. Thus, the creation of virtual urban contents is a key task for developing these technologies. The content creation can be manual or automatic. The latter case is one of the main topics that this dissertation deals with.

Geographic information gathers diverse disciplines that have evolved individually. Cartography, which is responsible for the map realization and management, is combined with Computer Science, which results into Geographic Information Systems (GIS). Furthermore, if GIS are combined with Architecture and Civil Engineering disciplines, we can provide GIS with information about buildings and roads. Finally, we can consider the use of databases and information systems of other nature to give additional content to these systems. Thus, we are dealing with a research field that needs to gather works coming from different areas into a common discipline, Computer Science, which allows to manage all the mentioned information.

1.1 Motivation

The research work contained in this PhD dissertation is part of a research project about 3D urban information management granted to the University of Jaen in 2008. The goals of this project were the following:

1. the development of software prototypes allowing the interaction with 3D urban information;

2. the design and maintenance of a map server including 3D urban information, accessible via the software developed according to the above goal;

3. the study and design of applications that, from different social sectors, allow an efficient exploitation of the tools obtained in the previous stages, developed in order to help the local development.

One of the scheduled tasks was the development of automatic 3D reconstruction technologies to be incorporated to the developed prototypes. In order to compile basis information for the project, there were two main information sources available: on one hand, public information from the Spanish Cadastre[2] served as the basis for the creation of outdoor urban navigation contents[ROOC+13, ROOF13]; on the other side, we had architectural plans from public buildings available to be used for the 3D indoor content creation.

Regarding the second information source, two research lines were started in parallel: the creation of a client/server architecture with cartographic data and the implementation of a website to navigate across the University campus using COLLADA models. Both are explained below:

[2] Available in http://www.sedecatastro.gob.es/OVCInicio.aspx (*in Spanish*)

Client/server cartography One of these lines dealt with the implementation of an urban cartography system based on a client/server database. The 3D models were manually generated from architectural plans as follows: the plans were manually processed using MapInfo in order to remove irregularities. From the resulting plans, a process was carried out to (1) detect rooms and (2) realize an automatic drafting of the floor plan geometry in order to generate a 3D building model. The information about buildings and their geometry was stored in a database at the server side and could be queried by the client in order to build X3D models.

3D contents in COLLADA Other line from the same project dealt with the creation of a website using the Google Earth web plugin. It contained 3D models from all the buildings in the Campus of the University of Jaén.

In order to shortly introduce some important concepts: Google Earth allows to load KML and KMZ files. KML (Keyhole Markup Language) is an XML schema that allows to add georeferenced annotations to Google Earth maps (place marks, pictures, polygons, 3D models and text notes). KMZ is a zipped format containing KML files, overlays, images, icons and COLLADA 3D models.

The University Campus buildings were manually modeled using SketchUp and exported to COLLADA files. The COLLADA models were georeferenced and included into KMZ files which are accessible in a website containing information about the University and offering the possibility to navigate through the 3D buildings[3].

Our work was conceived as a complement of these two lines: From the first work we considered to keep the philosophy of using databases to store information about the geometry of rooms. From the second, the richness of the created 3D models. Nevertheless, in the first work, the used plans represented the walls as single lines (walls without thickness). One of our goals was to detect rooms from plans that contained thick walls and avoid the manual processing as much as possible.

The initial goals explained above, about the creation of information databases, have been complemented during the research with other trends on urban information representation. Consequently, we dealt with a research work from two different points of view: first, part of the research has focused on the design of geometric algorithms to obtain semantic contents from geometric information; second, we dealt with the semantic representation models in order to propose a unified representation model complementing the existing ones. That model also links the semantic information with the geometric information from the source plans, as we will detail later.

1.2 Algorithms to obtain semantic contents

One of the key challenges in the project was the automation of content creation and the use of these contents to populate a spatial database. The initial ap-

[3] http://www.jaen3d.org/?q=node/2 (in Spanish, retrieved 2014-07-30)

proach was to generate a spatial database from the 2D floor plans, as in the first project described in Section 1.1. From the database it should be possible to extract the information needed to reconstruct the 3D building interior.

Therefore, the initial tasks of the work were: (1) the design and implementation of a spatial database including semantic information about whole buildings, correctly georeferenced to be included in a GIS; and (2) to populate the database with real information. In order to this, we started to study the buildings of the University of Jaén campus. The challenge was to automate the floor plan processing and design a number of algorithms able to carry out the processing with a minimum intervention from users.

The main difficulties in the research appeared at an early stage. We were facing a problem difficult to characterize due to several reasons:

- Like in other disciplines from automatic recognition, the recognition of semantic elements from a floor plan is a straightforward task for (specialized) humans but difficult to translate to the automatic scope.

- Floor plans, from a traditional architectural design point of view, are thought up for the visualization. Despite the existence of drawing standards, this fact provokes a style divergence among different draftsmen. The style divergences do not necessarily affect the visual aspect of drawings, but for the processing of geometric entities, these divergences endanger the stability of geometric algorithms, often resulting into unpredictable outputs or infinite executions.

- There exists a difference between two related tasks: the 3D reconstruction from the 2D geometry and the detection of semantic elements. The former is easy to automate and its solution is relatively well known: thick walls are represented by closed polygons that can be extruded. Floor surfaces can be obtained as the difference between the whole contour and the wall footprints. The latter requires a complete processing: first, the semantic elements have to be detected from the original geometry; then these geometric elements are used to create 3D content with semantics.

- After a bibliographic study about semantic detection from architectural floor plans, we found various approaches to deal with the problem. There are many works dealing with recognition from raster bitmaps obtained after scanning handmade drawings using a computer vision approach, whereas we found less works fully dedicated to the recognition from vector CAD drawings. Furthermore, most of the considered works appear to omit tests with complex floor plans and details about the trickiest scenarios. The main differences between raster and CAD drawings are: (1) in raster drawings, the information is not structured into layers, while in CAD drawings, layers are often used to separate different concepts (e.g., a drawing could contain a layer only for walls and another one only for furniture, etc.); and (2) measurements in CAD drawings are usually more accurate than in raster drawings.

The semantic elements that we proposed to detect were initially walls and openings, in order to determine the structure of rooms and other closed spaces. There are other building structure elements to be detected such as staircases and elevators. We have dealt with these elements less deeply.

Our research methodology to develop algorithms for the semantic detection from architectural plans consisted of abstracting automatic methods in order to emulate the way a human interprets drawings. Thus, an important stage during the research was to observe and analyze the set of available floor plans in order to abstract the human reasoning to recognize semantic elements. This first analysis showed the huge variety of drawing styles and methodologies: this fact made the development of algorithms that worked out for every situation difficult.

Another initial obstacle was how to deal with the columns from buildings. Sometimes they appeared drawn as part of the wall structure, other as part of a different layer. If the column layer and the wall layer are distinct, the walls may contain holes if the column layer is not considered. All these features were an open problem; they were considered to establish minimum criteria for the floor plans to be compatible with the geometric algorithms. Therefore, some plans would need to be modified to fit the algorithm criteria, or the algorithms would need to be changed to work out with some plans. This can be viewed as a two-way problem: (1) what do we need to assume on the plans to design algorithms that process them correctly and (2) how much can we relax our efforts on improving the algorithms by demanding the draftsmen to fulfill some drawing standards.

In our first attempt, we tried to abstract situations where columns and pillars are drawn in the wall layer. Then, a first approach arose: rule-based detection of semantics. Considering floor plans which contained different layers for door/window inserts[4] and for wall lines, we implemented an iterative algorithm that started to traverse walls made up by pairs of lines: Each iteration of the traversal starts from a door. When a wall crossing is found, the rule set is checked in order to determine which rule is triggered. The rules allow to determine the cross type, and the next direction to be followed. Once the traversal arrives to the starting door, a room has been detected and a new traversal starts from other door. The algorithm finishes when there are no more doors to iterate.

The main contribution of this approach was the detection of wall crossings even when there exist close columns. This initial work was published in the Iberoamerican Symposium on Computer Graphics 2009[DGF09]. Nonetheless, some disadvantages were found in this approach: (1) the rule set is very hard to characterize and implement; (2) overtraining arises, i.e., the algorithms works fine with test cases similar to the ones used to design the algorithms, but the behavior is unpredictable or bad in other cases; (3) the approach is very influenced by the imperfections of the floor plan.

[4]As we will detail later, an insert in a CAD drawing is a set of primitives that can be repeated in the same drawing and usually has a meaning, like a door or a window

Due to the disadvantages of the rule-based algorithm, we reached two conclusions: (1) it was necessary a preprocessing of the plans consisting of detecting and fixing irregularities. Since the irregularities are similar in many drawings, this preprocessing should be automatic if possible; (2) the semantic detection had to be approached from a global scope, as a computational geometry problem, instead of a rule-based reasoning procedure.

The irregularity detection is a problem hard to fully characterize, thus we restricted it to a number of cases, in order to focus on the main goal of the research. The main problem was that the detection and fixing of irregularities could provoke the appearance of further irregularities. It was not easy to predict how many iterations are needed to remove all the irregularities.

Regarding the search for a global algorithm to detect the semantics, we considered the global analysis of the set of lines from the wall layer in order to find thick walls that could be later completed with the openings (doors and windows) and the intersections amid walls. This would allow to determine how a floor plan is structured into rooms. The set of lines was analyzed to look for pairs of wall-prone parallel and close lines. This way, the idea of the wall adjacency graph (WAG) arose, one of the main contributions of the work.

The power of this artifact also allowed us to use it for the staircase detection. For example, it could be adapted to detect those staircases made by rectangular steps. This idea is based on the fact that the staircase detection can also be solved as the detection of pairs of parallel and close lines.

With regard to the opening search, the first algorithm we tested tried to find the openings and their surrounding walls. This solution has two implications: (1) the opening search is necessarily done after the wall search, and (2) we could predict that in those areas where the wall size is similar to the opening size, the opening detection would be inaccurate.

The search for intersections amid walls was also approached from two points of view. In an early research stage, we dealt with a problem consisting of clustering those vertices obtained from the wall detection stage. The goal was to find close vertices that are candidates to belong to the same intersection. The clustering algorithm allowed to find clusters minimizing the radius of a cluster and maximizing the distance between different clusters. In a second approach, we improved the clustering algorithm with the growing lines algorithm. Once we had the clusters, we simulated the growing of the lines until they intersected.

This part of the research is covered in Part II.

1.3 Semantic representation of buildings

Among the most recurrent applications of the use of semantic information from buildings we can cite 3D cadastre management, emergency response, construction management, indoor navigation, cultural heritage and automatic creation of video game environments.

According to the application and the information source, we can need a different Level-of-Detail (LoD), or perform different tasks to obtain semantics.

For instance, the LoD necessary to calculate acoustics in an indoor environment differs from the one needed for a virtual museum tour. In the former case, we need accurate physical parameters, whilst in the latter the presentation to the users is more important.

Regarding the data sources used to obtain 3D models, we find a number of trends. On one side the 3D building can be manually modeled using CAD tools; on the other side we find the (constrained) automatic building generation (of real or fictitious buildings). On an intermediate level we find data sources as LIDAR data, photography, building floor plans scanned from manual drawings (raster models) or CAD building floor plans (vector models). As we pointed out above, this research work is focused on vector models.

The application and the data sources determine the final data representation. In the initial research, as explained above, the representation was based on spatial databases. Nevertheless, in a first stage we also studied other models such as X3D, GML, CityGML, COLLADA/KMZ, etc. As the research progressed, and specially, after a research stay at TU Delft (Netherlands), the CityGML representation acquired more weight.

Furthermore, we considered interesting to focus on our own representation model in order to combine the advantages of the existing models with the results from the semantic detection process. Therefore, a three-part data structure appeared, representing the original information from architectural floor plans, the obtained semantic information, and the way to relate both pieces of information. This part of the research is covered in Part III.

1.4 Implementation of a user tool

All the research work has been simultaneously implemented in a desktop tool with two objectives: (1) to have a software environment to incorporate the new algorithms over a basis of software design patterns, and (2) to have a graphical interface to view the floor plans, validate the algorithm results and even make selective corrections over the inaccurate ones. The latter issue has been included in order to develop a future software product used in CAD, construction or other fields mentioned in this introduction. This topic is explained in Appendix A.

The rest of the document is organized as follows: Part II deals with the detection of semantics from CAD floor plans. Part III explains our proposal of a semantic model designed in this research work. In this model we tried to join the requirements and applications of existing models into a common framework, and keep open the creation of 3D export modules through a user-friendly interface. Part IV states the conclusions and the future work.

In Appendix A we explain the design and implementation of the application used as a benchmark to validate and fix the obtained results. In Appendix B we describe in depth the evolution of the clustering algorithms. Finally, Appendix C summarizes the main geometric algorithms used in this work, which are based

on the book *Geometric Tools for Computer Graphics*, by Philip J. Schneider and David Eberly [SE02].

Chapter 2

Previous work

This chapter gives an overview of the building modeling discipline: concepts and previous works. First we cite a number of previous works on automatic reconstruction of 3D buildings (Section 2.1. These previous works are related to the concepts described in Part II. Next, sections 2.2, 2.3 and 2.4 deal with the most important issues on Building Information Model (BIM) and Geographic Information Systems (GIS) and summarize their main standards. Finally, Section 2.5 review other building models proposed in the literature. These models are relevant to introduce the work described in Part III.

2.1 Automatic reconstruction of building environments

In this section we make a review on the main contributions to the (semi)automatic reconstruction of 3D semantic buildings. Tipically, architectural 3D indoor scenes are developed from scratch, or commercial applications, like Autodesk© 3ds Max© are used to perform some kind of 2D-to-3D conversion using the floor plan of the scene as input, but the process is not straightforward, and the user may be required to modify the preliminary result in order to get the final geometry. It is therefore desirable to have tools that can take a 2D floor plan as input, and generate a 3D model in a format general enough to be used for different purposes.

There are not many significant works related with automatic 3D scene generation using 2D floor plans as input. Maybe the most interesting is the one by Yuan et al. [YZ08], who have developed a system to compute rescue routes in a building, using topological and semantic information partially obtained from 2D floor plans. This application also allows the user to navigate the interior of the building. However, it does not consider the extraction of features like rooms, and its model is not intended for storage of data different from that related to rescue routes.

Other interesting works are the ones by Mas and Besuievsky [MB06], who

focus on retrieving data from the floor plans in order to perform light simulations, and the ones by Dosch et al. [DTASM00] and Or et al. [OWYC05], who present pattern recognition algorithms to analyze raster images of the 2D floor plans and obtain enough data to create a 3D representation of the scene; as a consequence of the pattern recognition process, these works present accuracy errors due to the way distances are measured.

Yin et al. have published an interesting survey on this topic [YWR09] where they outline the architecture of a complete solution for the automatic generation of 3D floor models using floor plans as input. Another work by Horna et al. [HMDB09] proposes a formal representation of consistency constraints together with a topological model for the representation of the information from a 2D architectural plan.

The existing works can be grouped according to the kind of input and the goals of the process:

- Some works introduce methods to recognize special symbols from a floor plan. These methods typically apply Computer Vision and Pattern Recognition techniques on scanned floor plans to obtain their results. [AST01] use networks of constraints on geometrical features to find symbols that represent doors and windows, defining a formal grammar to describe what have to be recognized. On the other hand, [DTASM00] describe a complete system for symbol recognition involving Computer Vision tools such as segmentation, vectorization and feature detection; the result is used to create a 3D reconstruction of the floor (without topology information) by means of extruding the recognized 2D geometry. Other interesting works include the one from [LLKM97], who apply the Hough transform to recognize symbols from hand-drawn architectural floor plans, and the one from [LYYC07], who deal with the recognition of structural objects and the classification of wall shapes.

- CAD vector drawings are the input data considered by other researchers that propose methods to generate 3D building models. These include the work from [HMDB09], who define the generalized-maps to represent adjacency relationships amid geometric elements, and use them to extract 2D topology and 3D volumes from a floor plan, given some assumptions on the quality of the floor plan and some considerations about the structure of a building that allow them to define constraints on the geometry; occasionally, the user intervention can be necessary to provide semantic associations to the geometry, and curved walls are considered as polylines. As far as we know, no further advances on this approach have been published up to now. Other works are the one by [MB06], based on the extrusion of planar polygons from CAD vector drawings to generate 3D building models used for light simulation, and the one by [PMSV08], who use 3D reconstruction for security modeling of critical infrastructures.

- A third group of works introduce methods to retrieve topological information from CAD vector drawings, like the one from [MY00], who use graphs

to store adjacency relationships amid floors, rooms and stairs during the design stage. Other related work is the one from [ZLF03], who build a topology graph to describe the distribution of walls and openings in a floor plan, and search for a set of fundamental loops to find corridors and rooms, so that an evacuation plan can be created from that information; this work emphasizes the loop search, but gives very few details on the graph building process.

We have also analyzed a number of existing approaches to the problem of the representation of building indoor models. Initially, a set of criteria to classify the existing models have been introduced. According to these criteria, some works have been classified into three main groups: (1) 2D (2) 2.5D and (3) 3D.

The analysis of the wide range of building models has been done according to geometric, semantic and topological features. This analysis allows us to state some conclusions:

- Due to the huge variety of existing representation models for building indoors, and the wide range of fields of application, it is quite complex to achieve unified models. Thus, we consider unavoidable a thorough design of the problem to be solved, instead of the application of general approaches.

- In most of the reviewed works, there exists a lack of automation for getting building models, without regard of the field (BIM, GIS, Spatial Databases or custom models). Therefore, algorithms for the extraction of semantic information from CAD floor plans need to be developed. In this area, we have proposed some methods to recognize rooms semi-automatically from vector floor plans in AutoDesk© DXF format, obtaining promising initial results.

- The use of formal approaches is recommendable, since it allows to take advantage of demonstrated results. For example, a lot of works formulate a problem in terms of graphs. Thus, existing algorithms from graph theory can be applied without the need to demonstrate the validity of the solution. Some of our work (in progress) tries to solve the wall automatic recognition using graph theory.

Since several works dealing with different applications of architectural data can be found in the literature, there are many representation models. Each of these models focuses on a reduced number of building aspects and applies different approaches. Traditionally, two main research areas have been defined: Geographic Information Systems (GIS) and Building Information Models (BIM). On the other hand, we can distinguish among several purposes of the representation models, i.e. topology, geometry, semantics. However, there are no pure topological, semantic or geometric models, but hybrid models using various levels-of-abstraction.

2.2 Building Information Model (BIM)

In this section we give an overview of the origins and evolution of BIM, together with a description of Industry Foundation Classes (IFC) and their applications.

2.2.1 Definition and history of Building Information Modeling

Building Information Modeling (BIM) is a digital representation of physical and functional characteristics of a facility, extending the traditional two-dimensional drawings with 3D models and information about time and cost. It also covers spatial relationships, light analysis, geographic information and other properties of building components [ETSL08].

The use of BIM allows to deal with the entire life cycle of buildings, achieving a better interoperability and thus reducing costs associated with the lack of information in CAD designs. BIM also helps detecting failures and inconsistencies and use simulation processes to predict problems in the construction stage before they occur. Furthermore, it permits an improvement of the building energetic efficiency and sustainability.

While the CAD paradigm in architecture utilizes mainly geometric primitives for 2D, or B-rep and CSG models in order to visualize and edit 3D designs, BIM technology appeared as an alternative based on parametric object modeling.

Parametric object modeling is based on the use of predefined object classes which define a mixture of fixed and parametric geometry, together with sets of relationships and rules to control the parameters and allow different instances of the same class to have different features, according to the values assigned to the class parameters for each instance. To illustrate the differences among CAD and Parametric Modeling, we present and example on how the process to design a wall with an opening is carried out according to both philosophies:

- Using 2D CAD, pieces of the wall on the left and the right of the opening are separately drawn as polylines, and the opening is then placed in the resulting hole as a set of geometric primitives, or an instance of a predefined block instance. If the opening location needs to be changed, the walls need to be redrawn. In the case of 3D CAD, a relocation of the opening would require to design manually the 3D solids which represent the surrounding walls. This is due to a lack of cohesion between walls and openings.
- Using parametric object modeling, the entire wall is inserted as a 3D solid with length, width and height parameters, and optionally other parameters about material and construction process issues; then, the opening is inserted choosing its base design and parameters such as horizontal and vertical position inside the wall, height, width, etc. The underlying geometry is automatically computed from these specifications, and changes in the opening's parameters are automatically spread to its geometric representation, as well as to the wall representation.

Although the exact origin of the BIM term is not straightforward to determine, most sources affirm that the appearance of architectural software based on parametric object modeling contributes to reinforce this paradigm. In this context, the first version of Graphisoft® ArchiCAD appears in 1984 as an alternative to the CAD design, and uses the term **Virtual Building** instead of Building Information Models. Other software for BIM design includes Autodesk Revit® Architecture and Structure, Bentley Architecture and its associated set of products, Graphisoft ArchiCAD®, Gehry Technologys Digital Project™ or Nemetschek Vectorworks®.

2.2.2 Implantation of Building Information Modeling

The effective use of BIM in construction projects has some requirements: It is advisable that a legal framework exists in order to rule the BIM construction process. In practice, this legal framework is usually accompanied by the existence of committees and associations that spread the use of BIM standards and ensure its accomplishment. Finally, the involved teams could require reeducation in the way they work and collaborate: As a BIM project is intended to be used along the entire life cycle, all the participants read and write information from/to it. Thus, they need to learn with BIM technologies and, if necessary, improve their teamwork skills. These factors cause differences in the level of BIM implantation in different countries.

The buildingSMART Alliance (formerly known as International Alliance for Interoperability, IAI) promotes the use of openBIM as the main reference standard for the use of BIM in construction. The main resources of buildingSMART cover:

1. buildingSMART Processes
2. buildingSMART Data Dictionary
3. buildingSMART Data model

Up to this moment, the buildingSMART Alliance is organized into the following chapters: Australasia, Benelux, China, French speaking, German Speaking, Iberian Alliance, Italy, Japan, Korea, Middle East, Nordic, Norway, North America, Singapore, UK & Ireland.

2.3 Geographic Information Systems

Geographic Information Systems (GIS) form other important discipline whose research and development has evolved separately. Thus, the technologies used in this area, as long as the approaches to solve its common problems are usually different from those in BIM.

While BIM's main goal is to give support to all the stages and staff in the construction process, GIS are mainly intended to manage geospatial data, and its technology relies on database storage and statistical analysis for geography.

Although BIM and GIS have different scopes, they share the necessity of representing building environments. In the case of GIS, the building information has to be related to the surrounding terrain information. In this regard, an accurate representation of the contour geometry of the buildings is crucial. Furthermore, building models need to be georeferenced.

In the last years GIS systems can manage more and more information about the indoor of buildings. To make the most of this feature, CityGML [Kol] defines an information model for the storage of city models. It includes four levels of detail to represent cities. LoD-4 is used for representing building indoor information.

2.4 BIM and GIS standards

This section briefly introduces the most important BIM and GIS standards: IFC and CityGML.

2.4.1 IFC

Since the appearance in 1997 of the first version of the Industry Foundation Classes (IFC) standard, created by the International Alliance for Interoperability (IAI), it became one of the most spread standards for BIM. The IFC data model is defined using the EXPRESS modeling language. IFC defines a schema [bui14] for the exchange of building information in all the stages of the construction process, including semantic information about the structure of buildings.

The IFC data model is registered by ISO as ISO/PAS 16739, while the EXPRESS modeling language used to define IFC is in the ISO Standard for the Exchange of Product model (STEP) as ISO 10303-11. Regarding the IFC file encodings, we find:

1. IFC-SPF (ISO 10303-21), which defines a text file with one line for each single object record. Its file extension is *.ifc*
2. IFC-XML (ISO 10303-28), which defines a XML file. It is suitable for interoperability and useful for sharing parts of building models with other XML tools. Due to its huge file size, it is less used than IFC-SPF format. Its file extension is *.ifcxml*
3. Finally, IFC-ZIP is a compressed version of IFC-SPF file format whose file extension is *.ifczip*.

EXPRESS can be considered as an object oriented language since it represents entities and relationships. The main data types are:

1. Entity data type: The basic piece of information in EXPRESS. Entities can be related to each other using inheritance, aggregation or via their attributes.

2. Enumeration data type

3. Defined data type: it specializes other data types by adding constraints on them.

4. Select data type: It allows to make a selection between two data types, mainly entity types.

5. Simple data type: used for primitive types like strings, integers, reals, booleans, etc.

IFC architecture is structured as shown in Figure 2.1.

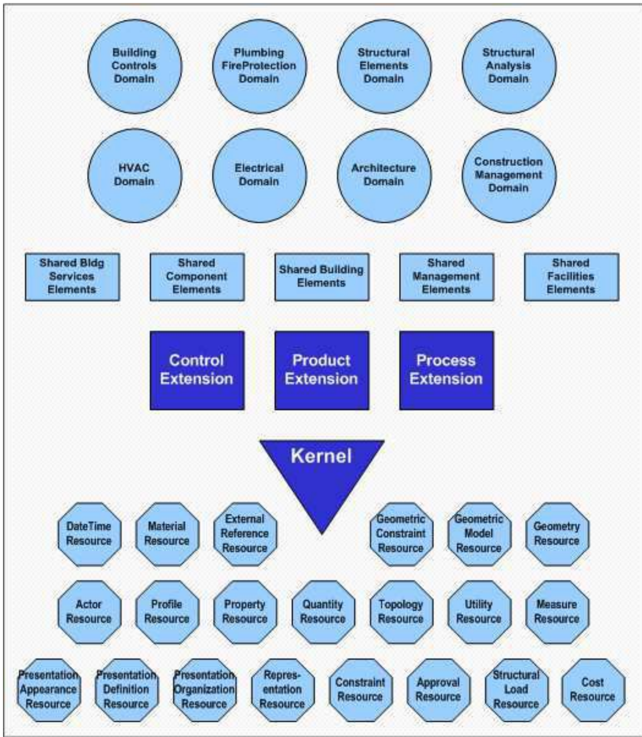


Figure 2.1: IFC Architecture

2.4.2 CityGML

The main standard for GIS considered in this work is CityGML, defined as an application independent Geospatial Information Model for virtual 3D city and landscape models [Kol07]. This model will be detailed in Section 10.

2.5 Other models from the literature

In 1992, Björk [Bjo92] laid the foundations of BIM models, introducing an object-oriented model to represent semantic data of buildings. His research is classified in the field of the Building Product Models (BPM), an initial and alternative name given to the BIM's. After studying and comparing some previous BPM's from projects like RATAS, GSD, De Waard's House Model and COMBINE IDM, his work focused on the definition of a schema including information about spaces and their enclosing entities (walls, columns, doors and windows).

Regarding the underlying geometry, BIM systems utilize CSG, Sweeping and B-rep models. The diversity of representation models is an obstacle to combine both technologies.

Isikdag et al. [IUA08] introduce a use case diagram which assumes that users with architect role create BIM models. Cerovsek makes an exhaustive research study about BIM technology [TC11]. This work offers a number of recommendations about how BIM models should evolve in order to make the development and standardization of BIM tools easier.

Building indoor information is needed for multiple applications covering diverse areas such as GIS, BIM, spatial databases or CAD. Each area uses different approaches to represent the information; the most extended standard used to manage BIM information in construction is the Industry Foundation Classes (IFC) model. Even within the same research area, a wide range of partial views of the same model can be found, depending on the specific application.

This huge combination of representation models and application areas makes it difficult to find common techniques and representation models to benefit from the results of a wide range of works. Therefore, a classification of models and application areas would be useful to have a starting point before considering the development of new approaches.

A possible criterion to classify building representation models is the spatial dimension: Building models may contain (1) 2D information (e.g. floor plans, footprints, 2D structural models of rooms and corridors, etc.); (2) 2.5D data consisting of 2D information and features concerning height of floors, relationships among contiguous floors or existence of different heights within the same floor; (3) 3D information including explicit primitives to represent 3D geometry and topology.

We will classify each reviewed model according to the above criterion. Furthermore, we will analyze which approach is used from the level-of-abstraction point of view, i.e., which level of geometric, topological and semantic information is contained in the different approaches:

- **Geometric models**: they only contain geometric information from building indoors. CAD floor plans usually belong to this category, since they consist of low-level data structured into layers [HB07]. Most of the observed CAD models represent walls as double rows of single lines with redundancy (i.e. points shared by two or more lines are defined once for

each line); openings, furniture and bathroom fittings are represented as instances [DGF09]. Although there exists some connectivity information about individual objects (instances of blocks), this representation is not considered as topological.

- **Topological/semantic models**: apart from the purely geometric representations, two levels of abstraction can be defined: (1) information about adjacency among geometric primitives is added to the model (e.g. points shared by two lines, or closed faces sharing edges) and (2) information about high-level entities (rooms, corridors, walls, columns) also appears in the model. The former will be named as *topological* and the latter as *semantic* representations of the model.

2.5.1 2D models

In this section we include works that make use of an explicit 2D geometric representation of buildings and other works that, although do not represent 2D geometry explicitly, are focused on an application area that does not need an explicit 2.5D or 3D representation.

Franz et al. [FMW05] analyze building models under two different points of view: cognitive sciences and architecture. They summarize seven existing graph-based models used in both areas, and discuss the transfer of information amid them. Regarding sthe cognitive point of view, they cite three models: (1) the occupancy grid, used in artificial intelligence for the robot navigation in a partially occupied space, (2) the place graph, used to represent connectivity between places and (3) the view graph, used to represent connected snapshots in a pictorial robot navigation. Related to the architecture field, Franz et al. cite four types of graphs: (1) the access graph between spatial regions (e.g. rooms), (2) axial maps, which represent the smallest set of lines of sight with maximum length, (3) isovist fields, representing viewshed polygons connected by edges if there exists mutual visibility, and (4) visibility graphs, derived from isovist fields.

Franz et al. study the representations from a merely topological point of view, since it does not deal with the underlying geometric representation of the cognitive and architectural models, and only the access graph includes semantic concepts as spatial regions.

Lamarche and Donikian [LD04] propose a method to represent the topology of an indoor space for the simulation of crowds of humans. They compute a set of convex cells using the constrained Delaunay triangulation of the floor plan. This set is then represented as a graph with nodes for the convex cells and edges for the neighbor convex cells (see Figure 2.2). This topological representation of the space allow them to identify passages, crossroads and dead ends in order to determine bottlenecks for pedestrians.

Regarding the underlying geometric representation, Lamarche and Donikian work with 2D convex cells from a flat indoor environment. This set of convex cells is obtained from a 3D geometric database by means of cutting 3D solids

with two parallel planes. The first plane corresponds to the floor and the second plane is such that its distance to the floor equals the height of a humanoid. The resulting lines are processed using S-MAT in order to get convex cells. Regarding the semantics, no information is included; the 2D convex cells are the result of the subdivision of spatial regions into convex subregions without keeping track of which sets of cells make up rooms, corridors, etc.

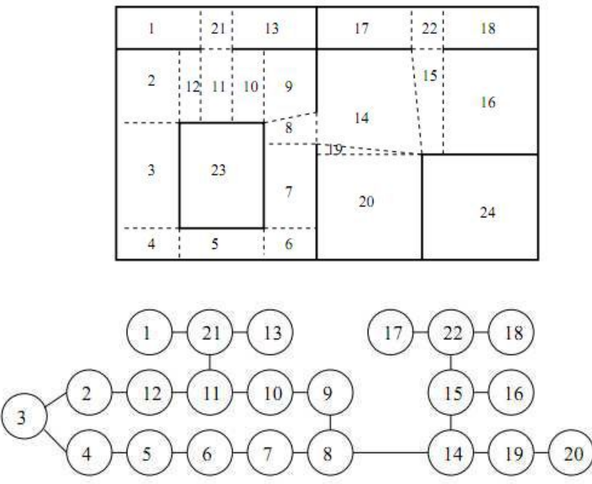


Figure 2.2: Graph representing accesibility between convex cells as defined by Lamarche and Donikian [LD04].

Plümer and Gröger [PG96] define another formal representation for the aggregation of 2D spatial objects: the nested maps. Basically, nested maps consist of constrained planar graphs whose cycles are structured hierarchically. This representation is useful to model the hierarchical structure of closed spaces. The constraints consist of seven axioms over the vertices, edges, faces and connectivity determined by planar graphs. Finally, the authors discuss about the integrity of nested maps when objects are added or deleted, and propose a generalized model in order to handle holes and unconnected areas: the complex maps.

Nested maps present a high correlation between geometry and topology: every node from the introduced planar graphs corresponds uniquely to a 2D point, and every 2D point corresponds to a node, while edges represent straight lines. The semantics of closed spaces and edges is out of the scope of that representation.

Hierarchical region graphs are used by Stoffel et al. [SLO07] to model the structure of spatial regions. Spatial regions are defined as closed and ordered sequences of corners. Some relationships over the spatial regions are defined as well, in order to model a region graph as the structure that represents several types of relationships between spatial regions. The hierarchical feature is useful to model the inclusion of spaces. The data structures include type parameters

to specify the semantics of nodes (doors, windows, etc.) and graphs (rooms, walls, stories, etc.).

In their work there exists a strong correlation among geometry, topology and semantics through the following points: (1) corners that make up the spatial regions contain a 2D position; (2) boundary nodes link adjacent spatial regions and include a type (door, window or opening); (3) child relationships establish a hierarchy among spatial regions; (4) region graphs put all the previous concepts together in order to include semantic information of the nodes including a type (floor, section, room, etc.)

Li et al. [LCR10] directly take as starting point a semantic indoor representation, including rooms, lobbies, inner or outer walls, doors and windows. This representation consists of a set of cellular units with meaning, i.e. bounded areas of the model occupied by one of the semantic elements (e.g. a piece of wall is considered a cellular unit). Cellular units can be free (rooms, lobbies, open doors) or occupied (walls, windows, closed doors). A regular decomposition of the space is then made using a grid-graph with a given level of granularity. Each node of the graph is labeled according to the underlying cellular unit, and connected to its eight neighbors. Using different algorithms from graph theory, some problems of space analysis and agent navigation can be solved (Figure 2.3).

The representation model by Li et al. is not intended to achieve an accurate geometric or topological representation. Therefore, no 2D representation model is explicitly mentioned. On the other hand, the discrete representation of the space is useful to optimize analysis problems for agents such as route planning, diffusion, centrality or topology calculation.

Zhi et al. [ZLF03] introduce a formalism to deal with the representation of architectural floor plans using the following methodology: Initially, an architectural floor plan is converted into an object graph representing the structure of walls and openings, such that loops represent closed spaces. Since rooms are obtained as minimal loops, this work uses spatial vectors to compute minimal area fundamental loops.

Zhi et al. also enumerate eight problems or difficulties which usually appear when CAD floor plans have to be processed, including the existence of drafting errors and redundant information, non-standard utilization of layer name codes, and the identification of door relationships between units, specially in multi-door units. Further constraints about the input floor plans are related with the adequate division of entities into layers and the existence of blocks for symbols and dimensions. If all the constraints are fulfilled, the selection of essential entities can be semi-automatically done.

Hahn et al. [HBW06] deal with the real-time generation of building interiors. The main characteristics of the generator are: (1) the generation of building interiors is lazy, i.e. only the areas near the view point are used to generate regions, (2) the generation schema involves the separation of the building into *temporary regions*, the creation of key points where *temporary regions* need to be converted into *built regions* and (3) a set of rules is followed to ensure the correctness and realism of the results. The generator uses pseudorandom

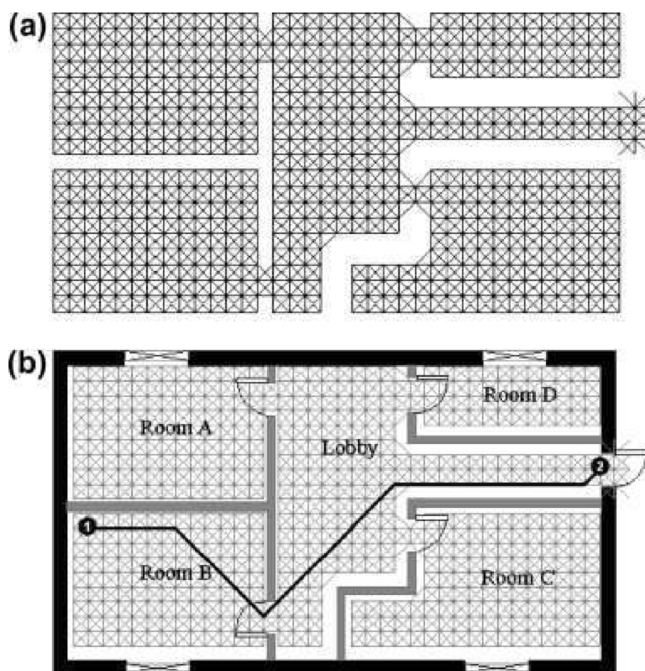


Figure 2.3: Route analysis using grid graphs [LCR10].

numbers which allow to reproduce the results if necessary by using the same seed.

Geometric and semantic aspects are fully covered by Hahn et al. since the generation process consists of making a progressive division of the temporary regions, so that the obtained subspaces correspond semantically to rooms and hallways. Regarding the topology, the authors make a partial representation by means of keeping track of the generation tree; thus, information about how subspaces come from the same space after a division step is available, and information about neighborhood of rooms or hallways can be obtained indirectly by analyzing the geometry of rooms which belong to the same parent in the generation tree.

Merrell et al. [MSK10] use an approach based on bayesian networks to solve the generation of building interiors starting from high-level requirements. An architectural program is created after training a bayesian network with real data; then the architectural program is turned into a real floor plan by applying optimization over the space of possible building layouts. Finally, this work proposes the generation of 3D models from the floor plans using customizable style templates.

In the Merrell et al. proposal, the semantic knowledge reaches a high level of detail, distinguishing between different types of rooms (bedrooms, dining rooms, kitchens, etc.) in order to design a layout which takes into account heuristics

followed by human architects (e.g. kitchens are rarely adjacent to bathrooms). Consequently, the geometry and the topological structure of the building interior appear explicitly as well.

2.5.2 2D models with height

This section summarizes works that use a 2D representation model with height information. Since some works do not describe themselves as 2.5D, it is not straightforward to decide whether a work belongs to the 2.5D category or to the 3D category. Therefore, some of the works included in this section mention 3D features, but their stress point relies on the use of 2D models with height information.

Slingsby and Raper [SR07] deal with pedestrian navigation in 3D city models. After introducing a state of the art on 3D city modeling, pedestrian navigation and pedestrian access within buildings, this work proposes a model to represent navigable spaces in cities consisting of a 2.5D representation of building floors. In order to deal with irregular morphology of floors, they propose the use of four constraint elements: ramps, stairs, breaklines and offsets.

Regarding the relationship between geometry and semantics, the authors propose the use of tags for barriers (walls and fences), openings and lifts to be associated to the geometrical elements, while higher-level semantic elements like rooms or corridors are not mentioned. The topological information remains implicit to the existence of tagged geometric elements. Thus, the search for lift or ramp tagged items would allow us to deduce partial information about the neighborhood among spatial regions.

Tutenel et al. [TBSdK09] propose a rule-based solver to generate indoor building scenarios automatically which works by using classes to represent 3D shapes (e.g. *Sofa*, *Table*, *TV*) with tags. Rules are defined in three different ways: (1) some rules are defined as constraints over the feature tags (e.g. bounding boxes from objects with the *OffLimit* tag cannot overlap, or objects like cups or plates must be placed on objects with *TableTop* tag); (2) class relationships are defined and affect to their objects; and (3) specific rules are added to the layout planner. The layout planner consists of a backtracking mechanism which defines a set of rules and executes the solver for each object to be placed. The solver computes a set of possible locations for the current object according to the previously placed ones, assigning weights to them. Then, it selects the most feasible location.

We have included the work by Tutenel et al. in the 2.5D section because although the basics for the layout solving is the architectural 2D floor plan, information about floor height has to be necessarily considered to avoid infinite stacking of objects in the same coordinates. Even though the authors do not give any details about the representation model for the geometry and semantics, it can be deduced from the paper that they are considered in the layout solver. However, no information about the topology is specified.

The arrangement of furniture is also solved by Germer and Schwarz [GS09]. However, they use a different approach in which agents are used to represent

pieces of furniture. Each agent is responsible for placing and orienting itself properly, and finding a parent object. In order to achieve this, each agent has three possible states: (1) search, when it has not been processed, its parent has been lost or the search for its position has failed; (2) arrange, when the agent has found a possible parent; and (3) rest, when the arrange is finally successful.

In the same way that Tutenel et al., Germer and Schwarz algorithms need to know the existence of rooms (from a semantical and geometric point of view) and its height, but since each room is designed independently, the topological information can be obviated.

Regarding building rendering, Van Dongen [vD08] proposes a technique to simulate building interiors viewed from the street without any storage of geometry. While buildings are modeled using single cubes, the rendering process simulate the existence of rooms, objects and people with the following algorithm: (1) a diffuse texture is applied to the exterior walls of each building using an alpha-channel: if the value is 1, the exterior texture is used, if the value is 0, the interior mapping is applied; (2) the interior mapping algorithm divides the interior of the building cube into planes which represent interior walls and ceilings; then a ray tracing algorithm is executed for each pixel in order to determine whether the first visible plane is a wall or a ceiling, and the corresponding color is applied to the pixel. Furthermore, people walking inside are simulated by adding additional billboard planes.

Van Dongen's work deals with rendering techniques and does not need to keep track of any *real* information about semantics or topology. Instead of it, ceilings and walls are artificially created by placing fictional planes within a building represented by a cube. However, this work has been classified into the 2.5D models as it contains certain information about floors with height.

The structured floor-plan proposed by Choi et al. [CKHL07] consists of a high-level semantic structure which accomplishes with nine principles about object orientation of the model, existence of information about relationships among entities, managing of spatial information and relationships, levels of detail, and automatic creation of a 3D model from the structured floor plan (Figure 2.4). This work introduces an object-oriented schema of the structured floor plan and algorithms to create this structure from the geometry of floor plans.

This model does not store an exhaustive geometric description of buildings. Instead of it, floor plans are used to compute only the relevant information for the semantics, e.g., area of slabs, ceilings, rings, or width and height of foundations, beams, openings, etc. The model is intended to be edited by CAD designers. Thus, new features added to a building design are included into the structured plan keeping the consistency. Although the underlying representation is 2.5D, the authors show some examples of creation of 3D content directly derived from the structured floor plans.

2.5.3 3D models

Finally, we present a summary of works which make a deeper use of 3D features.

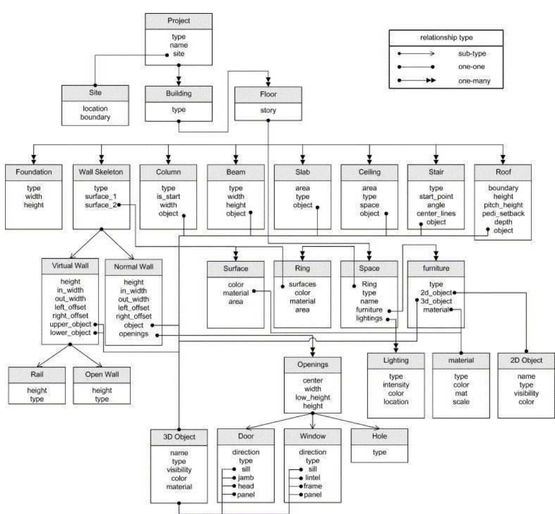


Figure 2.4: Building data model for the structured floor plan, according to Choi et al. [CKHL07].

Choi and Lee [CL09, Lee01] propose a graph structure called *3D Geometric Network* to represent connectivity between the rooms of a building. In contrast to previous works, geometric networks not only represent the structure of 2D indoors, but also model the 3D structure by means of adding edges between rooms from different floors. The methodology used to build the Geometric Network involves the use of the straight medial axis transform, in order to extract the wall structure.

The authors consider two possible inputs to build 3D geometric networks: (1) Vector CAD floor plans where walls are represented as closed polygons. The authors propose the use of the Straight Medium Axis Transform (Straight MAT), a variation of the Medium Axis Transform (MAT) algorithm [Lee82] in order to get the skeleton of the walls; (2) Raster floor plans. The authors include three methods to get a thin line from walls which are more than one pixel width: (a) Voronoi diagram-based thinning, (b) mathematical operator-based thinning and (c) boundary peeling method.

3D geometric networks represent geometry, topology and semantics: (1) B-rep models of buildings (geometric level) are used to obtain a topological model called Combinatorial Data Model (CDM); (2) 3D geometric networks (obtained from CDM's) contain information about the rooms (semantic level), about connectivity between rooms through openings and about adjacency between rooms through shared walls (topological level).

Clemen and Gielsdorf [CF08] propose a systematic way to normalize (reduce the redundancy of) geometric models. Their work uses a generalized representation for models consisting of solids made up by faces contained in planes, half edges and nodes. Geometric constraints (e.g. planarity, parallelism of planes)

can be ensured by referential integrity.

As a geometric model, this work utilizes B-rep data as input for the redundancy reduction methods. This choice is adequate as long as it is straightforward to put into correspondence both models. Regarding to the way the B-rep models are obtained, this work is focused on the management of indoor surveying data. The proposed model deals with inaccurate data obtained by measuring real environments, since the goal is the estimation of the real topology. The method assumes redundancies in the obtained data and uses a statistical approach. The topology appears explicitly as relationships between solids, faces, planes, half-edges and nodes. No high-level semantic elements are mentioned.

Van Berlo and Laat [vBdL10] collaborate with the introduction of an implementation of the conversion from IFC to CityGML. In order to achieve this, they introduce an extension for CityGML called GeoBIM. Therefore, the underlying geometric, semantical and topological models are the same in IFC and in CityGML.

Topological houses proposed by Paul and Bradley [PB03] constitute a purely mathematical abstraction to define houses. This formal definition allows to encode houses using two structures: PLA (points, lines and areas) and PLAV (PLA + volumes). The authors demonstrate that these structures can be used to represent houses into relational databases without loss of information. The mathematical model presented in this work relates geometry and topology. The only semantic elements which are taken into account are walls and ceilings.

Billen and Zlatanova [BZ03] propose the dimensional model, a topological abstraction for 3D objects which allows to analyze complex relationships between them. This model represents four *dimensional elements* (0-D, 1-D, 2-D and 3-D) for each spatial object and introduces a systematic way to analyze relationships between different dimensional elements from the same objects.

The dimensional model is tested using two data sets. The selected representation model is the Simplified Spatial Model [Zla00] due to (1) its explicit representation of objects, (2) the use of minimal elements (i.e. node and face) and (3) successful tests for large 3D models.

Since the goal of their work is to serve as a useful framework in order to answer topological/geometric queries concerning 3D cadastre, the inclusion of features which deal with the semantics of interiors is avoided; it is not necessary to include other concepts different from 3D cadastral units, buildings or pipes.

A schema with four levels of detail is proposed by Hagedorn et al. [HTGD09] to represent indoor building models. This schema has some similarities to CityGML; however the authors include features for indoor routing, not included in CityGML. Apart from the four levels of detail, this work defines three components: (1) thematic model, (2) geometry model (based on GML) and (3) routing model, with the usage of connection points between adjacent spaces (Figure 2.5). Thus, the three levels of abstraction (geometry, semantics and topology) appear related within this work.

Van Treeck and Rank [vTR07] use a graph-theory approach to represent geometric, topological and semantic data of a Building Product Model (BPM). They use data structures at several levels of detail: regarding geometry, they

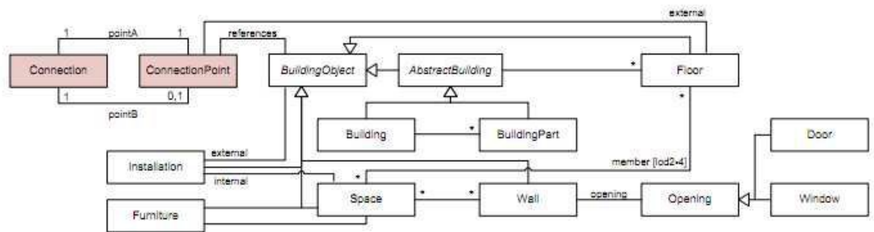


Figure 2.5: Thematic and routing model from the work by Hagedorn et al. [HTGD09].

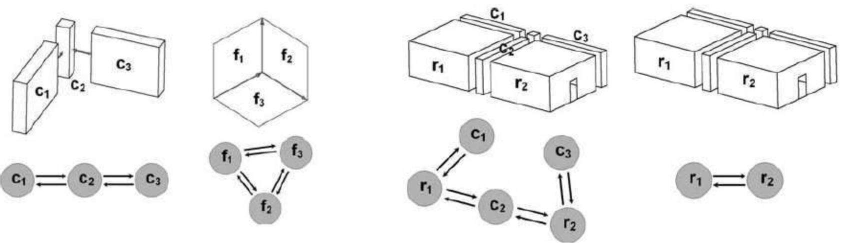


Figure 2.6: Graph models for BPM, defined by Van Treeck et al. [vTR07]. From left to right: Structural component graph; graph of room faces; relational object graph; room graph

assume rigid bodies as B-rep surfaces to model walls, windows, doors, columns, etc. To represent topology, they start from a radial-edge structure [Wei88], which represents relationships among vertices, edges, co-edges, loops, faces and bodies. From this information, this work derives four different graphs: the *structural component graph*, the *graph of room faces*, the *relational object graph* and the *room graph* (see Figure 2.6). Finally, they provide methods to map between the cited graphs and obtain some semantic information about walls, volume bodies (rooms), slabs or plates.

Borrmann and Rank [BR09] propose a set of spatial operators to determine the relative position between the bounding boxes of two 3D spatial objects which represent buildings or building parts. The set of relative positions has six elements, two for each 3D axis, namely above, below, eastOf, westOf, northOf and southOf. They introduce two approaches: (1) the projection-based model, in which an object is compared with the extrusion of other object, and (2) the halfspace-based models, in which the first object (reference) determines two halfspaces, and the second one (target) is tested to decide which halfspace it occupies. For the projection-based approach, the authors propose the *slot-tree* data structure to make the tests more efficient.

Regarding the geometry, they utilize synthetic sets of 3D spatial objects designed to contain worst-case scenarios. These scenarios allow to demonstrate the

high performance of the introduced query language. The storage of topological and semantic features is out of the scope of this paper.

Isikdag et al. [IUA08] remark (1) the lack of semantic information and spatial relationships in CAD models, and (2) the inefficient representation of 3D geometry in geospatial models, as the main reasons which make the conversion between BIM and geospatial models difficult. They propose use case scenarios to implement BIM's in the geospatial environment. They propose the Output Data Model, which includes classes for structural and semantic elements such as walls, columns, stories and openings.

Boguslawski and Gold [BG10] deal with the problem of representing non-manifold CAD models using a data structure called Dual Half-Edge (DHE), consisting basically of two dual structures. On one hand, a net of half-edges which make up solids is maintained, storing for each half-edge several pointers to adjacent half-edges and faces. On the other hand, there exists a dual structure of connected solids. This data structure is tested with the use of Euler operators on real CAD buildings. The DHE data structure is tested on two linked buildings from two scanned floor plans. Then, floor plans were manually vectorized and extruded within AutoCAD. This work presents a strong correlation between geometry and topology. Regarding semantics, concepts related with high-level items like rooms, floors, etc. seem to be out of the scope of this paper.

An example of the application of BIM for computer games with indoor scenarios can be found in [YCG10]. Yan et al. propose an architecture consisting of three modules: BIM, crossover and game. Information about the buildings is managed by the BIM module, while the crossover module is used to exchange information between the BIM module and the game module. The crossover module defines a high-level graph with detailed semantic information: each node represents a room and each edge represents a door between rooms (Figure 2.7). The crossover module is interchangeable in order to fit different BIM modules.

Xu et al. [XZZ10] propose a model including geometric, semantic and topological aspects of 3D City Models. To achieve this, a 3D city model is enriched with a thematic module which contains semantic and topological information; then the items of the thematic module are mapped onto the geometric model. This work also introduces a semi-automatic integration tool for the semantic enrichment process.

This process allows to keep the geometry-semantics coherence by means of mapping the semantic information onto the geometric objects (solids and surfaces). In order to deal with the topology, the authors illustrate with an exploration case the construction of a *topological network* and a *geometric network*. The former represents adjacency, connectivity and hierarchy, while the latter is used as the underlying basis for the topological network.

More recent contributions focus on the importance of the topological correctness as well. Guo et al. [GLY$^+$13] rely on 3D GIS to accomplish with the 3D Cadastre requirements, including the search for neighboring objects and the maintenance of face-based and volume-based topological consistency. Furthermore, big efforts are made to make existing 3D complex cities to be topologically

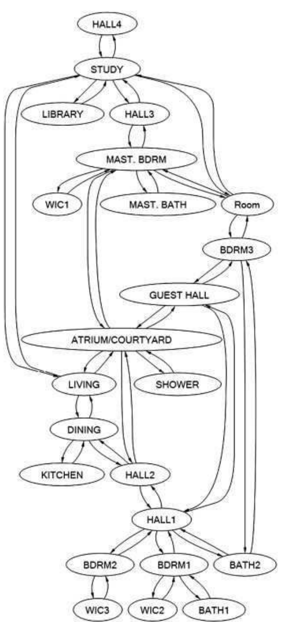


Figure 2.7: Room connectivity graph[YCG10].

consistent [XZD$^+$13] or to validate topological models of 3D cadastral plans [KTM13]. Finally, Goetz [Goe13] deals with 3D valid models automatically obtained from 2D floor plans on the basis of OpenStreetMap. The creation of the 2D topology floor plans is manually assisted.

This set of reviewed works will be compared and analyzed in Chapter 7 in order to contextualize our proposed model: a three-level framework to represent topologically correct models.

Part II

Detection of semantic elements

In this part we address those aspects of the research related to the detection of semantic elements. The basis for the creation of contents for building models is the processing of CAD architectural plans. In Chapter 3 we present some basic concepts about floor planning and DWG and DXF formats. Further information can be found in the DXF format specification document [Aut11].

Next, a discussion about which requirements are necessary on floor plans for the semantic detection algorithms is presented.

The rest of this part deals in depth with the semantic detection problem. First we will discuss the problems involved by the existence of drawing irregularities (Chapter 4). Their detection and fixing is a previous step for the subsequent algorithms.

After introducing the irregularity detection and fixing, we will discuss about local and global methods to detect the semantics. In a first research stage, we opted for local methods (Chapter 5). We will introduce them and justify why we decided to move to global methods (Chapter 6).

As global methods, we will introduce the wall detection using wall adjacency graphs (WAG) and the construction of a graph representing the topology of a building story.

Chapter 3

Architectural design. Constraints

The way architects give expression to their designs is important. Architectural drawings can be subjective, and several designs with minor differences can represent the same information. The main goal of classic designs is to be used during the construction process for visualization, or inference of acoustic, structural, thermal and other properties. The usual process the architects follow for their projects consists of three main stages: the conceptual design, the basic design and the final design (also called detailed design).

The conceptual design is a study phase in which ideas and alternatives are evaluated. This phase results in a basic process concept, a preliminary schedule and a rough cost estimate.

The basic design involves placement, description of the environment, basic definitions of floor, elevation and section models, and volume calculation. It is the elaboration of the conceptual design into a package, defining the process requirements and the associated equipment and project facilities.

The documentation obtained after the basic design is a set of plans defining the necessary aspects to know the scope of the project, i.e., terrain, placement, global floor plan, plans for different floors, elevations and sections, and budget.

The final or detailed design deals with (1) the construction, (2) the structure and (3) the installation floor plans. Its development starts after the basic design has been approved by the clients. On the basis of basic design plans, an architect or a civil engineer adds further layers including technical specifications for the building contractors, details about materials to be employed, construction techniques and strategies, etc.

The final project is usually carried on by adding layers to the existing plans, so that the information from basic and final designs is available in the same model and can be filtered depending on the project stage.

Together with the basic design plans, the final design is documented with:

- Topographical plan.

- Foundation and wall raising plan.
- Beams and columns plan.
- Installation plans: underground plumbing, electrical, water, emergency response, mechanical, voice and data, etc.
- Building inner and outer enclosure elements (walls, openings, grilles...).
- Finish plans: Painting, paving, plaster, thermic and acoustic insulation, waterproofing, etc.
- Exterior areas: sidewalks, gardens, fences.
- Descriptive and constructive reports, rule accomplishment documents.
- Work scheduling and budgets including detailed measurement.

In order to achieve the final design from the basic design, architects usually carry out an iterative process: the basic and the final designs can be modified several times during the process. Thus, if we design and implement automatic algorithms that obtain information from the basic design, we cannot rely on the fact that the basic design has been totally finished.

Therefore, the way plans are drawn should not be crucial for the subsequent processes in an architectural pipeline. If we aim to use basic design plans to infer automatically further information, we need to focus on the development of software tools and algorithms that are able to obtain high level features. Nevertheless, the plans used as input for these tools and algorithms have to fulfill some minimum style requirements to be suitable for its processing.

3.1 Architectural drawings. File formats

In this section we introduce how the concepts introduced in the previous section are applied in practice to real designs and show some examples.

First, an explanation about what architectural drawings contain is presented. Then, the concepts *layer* and *block* will be defined to analyze how they are used to structure architectural designs. The existence of CAD standards to design floor plan will be discussed, and finally we will settle the requirements that the designs used for our algorithms must fulfill.

Although the concept of CAD design is very general (including 3D elements and complex geometry), we focus on 2D designs used for architectural planning. In this sense, a 2D CAD architectural design can be informally defined as a set of primitives and blocks belonging to a 2D plane, and structured into layers.

Typically, building designs are stored as vector drawings created with CAD software applications and composed of low level graphical primitives, like lines, polylines, circular arcs, etcetera. Optionally, blocks can be created as combinations of primitives and/or other blocks; this allow the user to easily create

and instantiate representations for common elements like doors, windows or furniture.

Moreover, CAD applications allow the designer to divide the graphical information into layers, in order to group related elements together. The layer visibility can be switched on and off in order to emphasize what is really important in every situation.

A primitive is the most basic piece of information in a CAD design. Examples of primitives are lines, polylines, circles, arcs, rectangles, ellipses, elliptical arcs or hatched surfaces. Table 3.1 summarizes the most common primitives in CAD 2D design.

Table 3.1: List of primitives in 2D CAD design and their attributes.

Primitive	Attributes	Representation
Line	Four real values (two points) $x_{start}, y_{start}, x_{end}, y_{end}$	
Polyline	Sorted set of vertices. Each vertex contains a point (x, y) and a bulge value[1]	
Circle	Center point (x, y) and radius	
Circle arc	Center point (x, y), radius, start angle and end angle	
Rectangle	Two corner points	
Ellipse	Center point (x, y), major and minor axes	
Ellipse arc	Center point (x, y), major and minor axes, start angle and end angle	
Hatch	It defines a pattern for a closed surface. The closed surface is defined by a boundary path of segments an the pattern itself is defined as line data	

Besides, there are usually annotation elements (text, measurements, etc.) which help to understand the information represented in the floor plans. The annotation is independent from its annotated element, i.e., although they usually appear close to each other in the drawing, they are not necessarily linked.

[1] In a sorted sequence of vertices, the bulge measures the curvature between the current vertex and the next one. A bulge of 0 is a straight line, and a bulge of 1 is a semicircle.

Therefore, annotation information cannot be used when processing automatically the floor plan in order to know the measurements.

Architectural floor plans can also structure their primitives using two types of aggregations: layers and blocks. Though their underlying structure is similar (groups of primitives), their use is substantially different. Layers are used to logically group sets of primitives with a related field of application, while blocks are used to create objects which are commonly repeated in a design. For instance, there could exist layers to group walls, and blocks to define bathroom fittings. Another important difference is the scope of layers and blocks: while layers group instances of primitives in the drawing, blocks are used to build *abstract* groups of primitives which are then instantiated in the drawing as *inserts*.

Regarding file formats, we will focus on Document Exchange Format (DXF) due to three main reasons: (1) it is a text-based, open specified standard, (2) it is widely used by applications to exchange drawings and (3) there exist open-source libraries for loading, parsing and editing DXF files.

DXF file format was created by Autodesk® in the first version of AutoCAD® to allow exporting from the AutoCAD native file format, DWG, which is a non-publicly specified binary format. Therefore, DXF format used to include all the features of DWG models. In the most recent versions, some complex types from DWG are not correctly supported by DXF.

A DXF text file structures the information into pairs of lines. The first line of each pair (group code) is a numerical key identifier which indicates the type of the pair, while the second line is a numeric or string value. The file is structured in four sections, described below:

1. Header: This section contains the definition of variables associated with the drawing, such as AutoCAD version, measurement units and some issues related to the drawing visualization.

2. Classes: This section contains the information for a set of application-defined classes used in other sections.

3. Tables: Descriptions of the elements needed to draw the model are contained in this section. Examples of tables are (1) LTYPE, where the name, color and pattern for each line type used in the drawing are contained; (2) LAYER, which contains the name, default color and default line type for each layer in which the model is structured; or (3) VPORT, which describes the drawing's viewport. We are only interested in the LAYER table: Each layer specifies a color, line type, line style, etc. to be shown on the floor plan. Layers are used to group elements with similar semantics (walls, columns, openings, furniture, electricity, bathroom fittings, stairs and lifts, communications, plumbing...). Each layer must have a unique name.

4. Blocks: This section contains the description of all the blocks used in the drawing. Blocks are abstract definitions of drawing elements. They are described using a local coordinate system and made up by primitives or

instances of other blocks (inserts). Each block must have a unique name. For each block, this section defines some general items such as default color, default layer and then, the set of primitives or inserts that contains.

5. Entities: This section is the core part of the model. It contains a list of entities, the most basic piece of information in a DXF floor plan. The main types of entities are used to represent (1) primitives, specified according to Table 3.1; and (2) inserts, specified by their insertion point (translation), scale factors and rotation angle.

6. Objects: This section contains non-graphical items of information. Its structure is similar to the entities section.

7. Thumbnail image (optional): a preview image of the drawing.

Figure 3.1 summarizes the main elements in the structure of a DXF file. This figure is not exhaustive; we focus on three main elements: Entity, Block and Layer. Regarding the entities, the subtypes are neither exhaustive; they include only the most commonly used in our test-case building floor plans in order to detect semantic features:

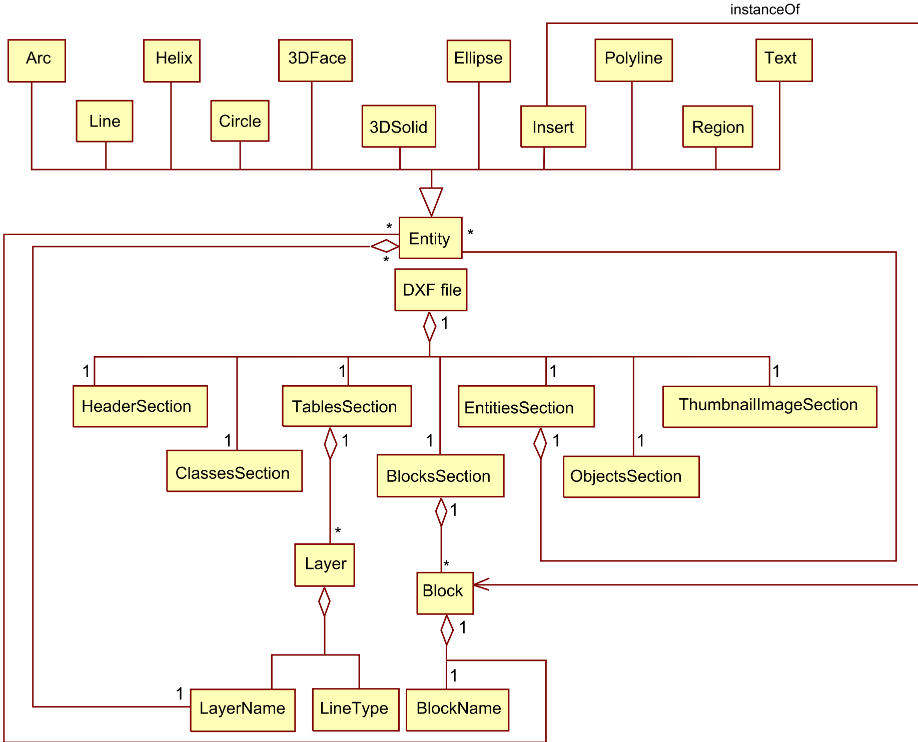


Figure 3.1: DXF file structure

- A DXF file (center of the diagram) contains the seven sections enumerated above.
- The entities section contains most of the drawing information. Thus, it is the core section for geometric processing purposes. It consists of a set of entities. An entity is an abstract type for all the drawing elements shown in the diagram (lines, arcs, circles, polylines, inserts, etc.).
- Other relevant sections are blocks and tables. In the blocks section, the abstract blocks are defined, providing them with an unique name, and a set of entities in a local coordinate system. The tables section contains global information about layers, such as name, default color and default line type.

The DXF file structure does not provide the designer with any restrictions about how a real floor plan must be organized. In other words, the DXF file structure describes the structure of a drawing, not the semantics of a floor plan. Thus, in the following sections we introduce the concept of floor plan standard, and summarize some guidelines we have observed in the test files we used. The algorithms introduced in this paper are based on these guidelines.

3.2 CAD standards. Floor plan structure requirements

Although there exist several standards on how the layers should be organized in architectural design [NCS11, ISO98, D$^+$11], CAD applications do not ensure that the users follow any of these standards, and therefore it is possible to find CAD drawings where the information regarding walls and openings is mixed with other data or divided into several layers. The second situation is easily solved by mixing the contents of the layers that keep the desired information, whereas the first situation is still an open problem, because typically there is no additional data (apart from line attributes like color or thickness) to support automatic extraction of primitives from the layers without involving the user. Here we consider that the walls are stored in one or several layers, but mixed with nothing else than (optionally) the openings, and that blocks represent the openings.

Below we describe the requirements of the floor plans in order to guarantee the correctness of the algorithms.

- There must exist at least three distinct layers: a wall layer, an opening layer and a staircase layer. The name of these layers is not relevant since they will be chosen by users as part of the semiautomatic processing. This point is needed in order to make easier to distinguish among the different algorithm inputs (wall, opening and staircase detection respectively).
- The walls can be drawn with or without explicit thickness. We say that a wall is drawn without explicit thickness (also called *paper* wall) if it is

represented by a single line (Figure 3.2.a); on the other side, it is said to be drawn with explicit thickness if each wall has been drawn extruding its line to both sides and the overlapping and missing parts have been arranged (Figure 3.2.b). The latter case is more common in the test cases we deal with. Thus, another requirement is that thick walls must be represented as parallel lines. These lines do not have information about connectivity.

- The openings (doors and windows) must be defined as blocks. Each block representing an opening is supposed to be aligned with the local coordinate axes while its instances will be rotated, translated and scaled as necessary. The name of these blocks is not relevant since users select them semiautomatically. The definition as blocks is necessary to recognize separate openings, while the requirement about the alignment of its elements with respect to a local coordinate system is useful to detect the actual orientation of the block and thus determine which walls enclose them. Nevertheless, we will discuss about how to deal with situations in which the openings are not drawn as blocks but as lines and arcs.

- The representation of columns varies amid different floor plans. Right now we consider they are part of the wall structure, leaving other possibilities as future work. Those floor plans where the columns are not drawn as part of the wall structure may contain holes. Consequences of this fact will arise for the wall detection problem, that will be explained in Section 6.1.

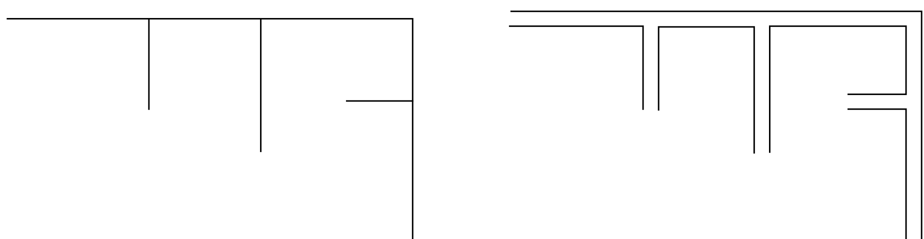


Figure 3.2: (a) walls drawn without thickness (*paper* walls); (b) walls drawn with thickness

3.3 Local methods vs. global methods

The recognition of drawing patterns is a straightforward task for humans, but it is extremely difficult to build automatic systems for the recognition of abstract items in a drawing.

There are two approaches to detect the structure: local and global.

Local methods try to reconstruct the shape of a set of walls and openings by going through the wall path to find intersection between walls, or openings.

In order to detect the closed spaces, some rules need to be followed when the method reaches a possible intersection. The main difficulty of this approach is the description of the main situations that can be found. This set of situations can be huge. Even when restricting the structure of floor plans to orthogonal-shaped walls, irregular corners containing columns can be found

Global methods try to reconstruct the floor structure analyzing the whole drawing. Their main advantages are the possibility to get information about unconnected areas in the drawing, avoiding problems that arise when local methods find an unsolvable intersection. The main disadvantage is the lack of accuracy in some areas.

The following chapters are dealing with the detection and fixing of irregularities, which is a previous step that is to be carried out before both the local and the global methods.

Chapter 4

Detection and fixing of irregularities

Architectural drawings are rarely clean sets of geometric primitives. During the drawing process, designers usually make visually imperceptible errors in the floor plans that do not affect their aspect. According to a traditional work flow in architecture, floor plans containing irregularities can be visually inspected and used by humans without drawbacks during the construction process. However, floor plans with irregularities must be avoided if their aim is to be (semi)automatically processed. The existence of irregularities makes unpredictable the result of the algorithms introduced in this work. Therefore, it is necessary to analyze and implement a previous step to detect abnormal situations.

Drawing irregularities consist of geometric, visually undetectable elements that affect the behavior of the geometry processing algorithms. The irregularities are related to the way geometric primitives are drawn. These errors can be contained in any layer. The detection and fixing process is separately applied to each layer that is taken into consideration.

Irregularities are mainly due to the three following reasons: (1) the drawing is carried out too quickly due to time requirements; (2) changes in the drawing are made without checking its consistency; or (3) visual accuracy is not always an important issue for designers, and bugs are not taken into account.

This chapter describes the most common situations found in real floor plans and how to remove them. It is structured as follows: first, the set of considered irregularities is described; then, we introduce the range comparison of numbers, a mathematical artifact needed to determine the relative positions amid geometric primitives; after that, we explain how the range comparison is applied to the irregularity detection process; finally, some issues regarding the irregularity detection and fixing tasks are discussed.

4.1 Irregularity types

The simplest irregularity type is the existence of null-length primitives, such as lines without length or arcs with the same initial and final angle. The detection of overlapping primitives includes more complex cases than the null-length primitives detection. In order to classify the cases, we consider lines in order to later generalize these cases to other primitives. Figure 4.1 shows eleven cases that can be found after analyzing a pair of segments contained in the same line.

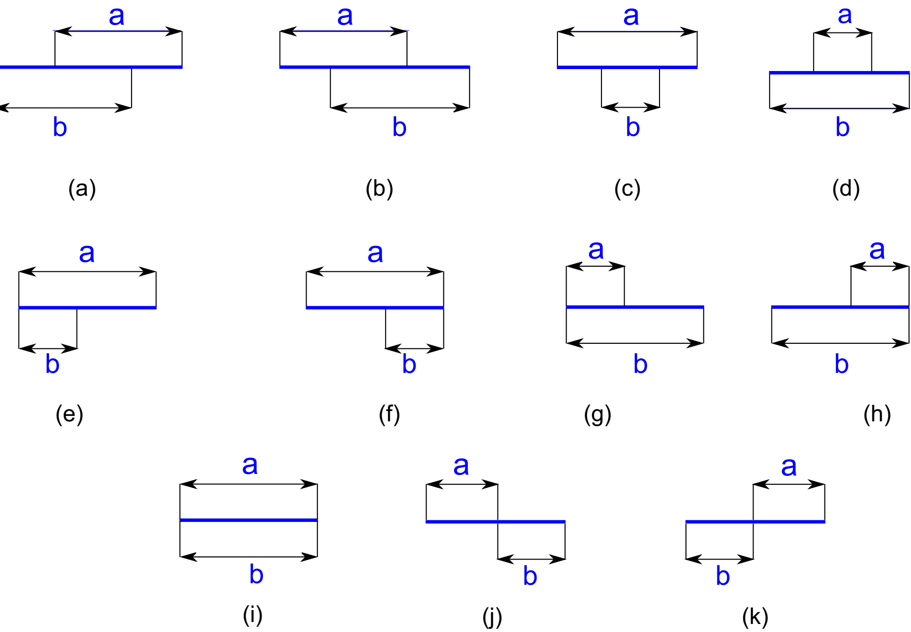


Figure 4.1: Irregularity cases: (a),(b): Overlapping lines; (c),(d): Small line contained in a bigger line; (e)-(h): Lines with common origin; (i): Duplicated lines; (j),(k): Consecutive lines.

In order to separate the above cases according to how they are fixed, we can simplify them into four by removing equivalent cases. The first two cases (a) and (b) are symmetric. Cases (c) and (d) are also symmetric, and cases (e)-(h) are particular cases of (c) and (d), thus (c)-(h) are grouped together. Finally, (j) and (k) are grouped since they are symmetric too. The resulting set of simplified cases contains these items: overlapping, duplicated, parted and contained (Figure 4.2).

Thus, the five cases are null-length, overlapping, contained, duplicated and consecutive.

The same cases, applied to concentric arcs, are obtained by comparing the start and end angles of each one, instead of the relative position of end points.

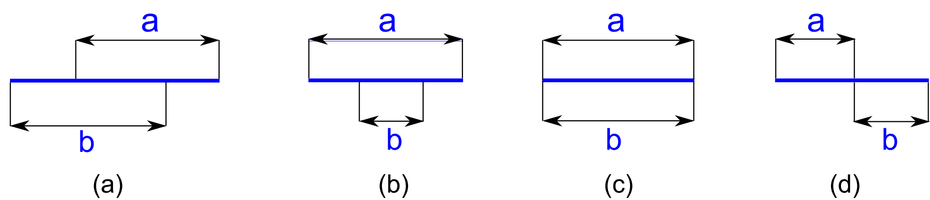


Figure 4.2: Simplified irregularity cases: (a) Overlapping; (b) Contained; (c) Duplicated; (d) Consecutive

4.2 Range comparison between real numbers

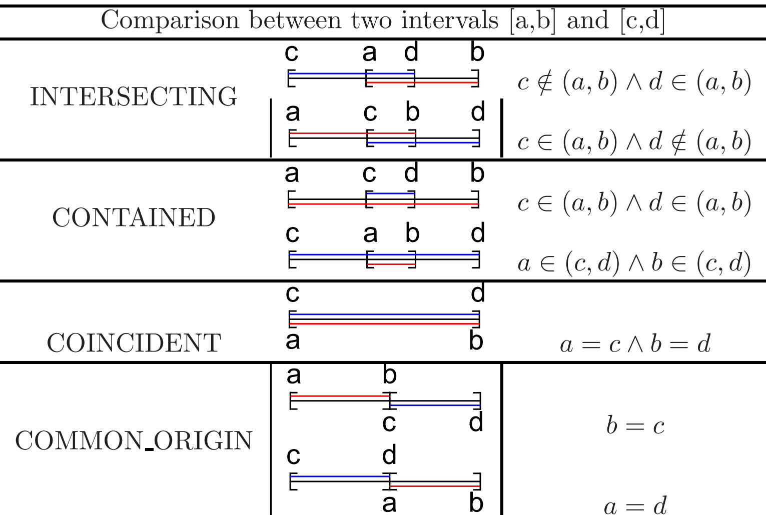
The basis for the case detection is the comparison between ranges of real numbers. Thus, in this section we explain this issue. The underlying problem to determine the relative position between two geometric primitives consists of (1) comparing a real number and an interval and (2) comparing two intervals.

The result domain of the above mentioned comparisons is a set of tags which indicate the relative situation of a number and an interval, or two intervals. These are as follows:

- Comparison between a number and an interval
  1. IN: the considered number is inside the interval
  2. OUT: the considered number is outside the interval
  3. LEFT: the considered number equals the left value of the interval
  4. RIGHT: the considered number equals the right value of the interval
- Comparison between two intervals
  1. INTERSECTING: one end of one of the intervals is IN the other interval, and the other end is OUT
  2. CONTAINED: both ends of one of the intervals are IN the other interval
  3. COINCIDENT: one end of one of the intervals is at the LEFT side of the other interval, and the other end is at the RIGHT side
  4. COMMON_ORIGIN: one end is at the LEFT side or the RIGHT side of the other interval, and the other end is OUT

Notice that the four cases between two intervals correspond to the simplified irregularity cases shown in Figure 4.2 of Section 4.1.

In order to summarize the analysis of intervals, Table 4.1 gathers all the possible inputs for numbers and intervals and their corresponding output label:



Comparison between two intervals [a,b] and [c,d]		
INTERSECTING	c a d b	$c \notin (a,b) \wedge d \in (a,b)$
	a c b d	$c \in (a,b) \wedge d \notin (a,b)$
CONTAINED	a c d b	$c \in (a,b) \wedge d \in (a,b)$
	c a b d	$a \in (c,d) \wedge b \in (c,d)$
COINCIDENT	c d / a b	$a = c \wedge b = d$
COMMON_ORIGIN	a b / c d	$b = c$
	c d / a b	$a = d$

Table 4.1: Comparison between two intervals of real numbers

4.3 Irregularity detection

In this section, range comparison between intervals is applied to the irregularity detection between segments and arcs. In order to detect all the irregularities in the drawing, all the primitives are compared each other, two by two. The three steps needed to determine the relative position between two primitives are:

1. Ensure that both primitives are comparable (collinear segments, concentric and same-radius arcs)

2. Convert the irregularity detection problem into a range comparison problem. This conversion is detailed below

3. Compute the range comparison and map the result of the range comparison into an irregularity type

Comparable primitives Not all the pairs of segments/arcs need to be compared. In the case of segments, only those that are collinear (thus parallel) are compared. In the case of arcs, they need to be concentric and have the same radius to be compared.

Irregularity detection to range comparison reduction Once we have checked that two segments are collinear, 2D segment end points can be reduced to 1D numeric values by projecting them to their containing line. Thus, each segment produces two values of an interval, and these intervals are compared according to the table 4.1.

For the projection of the segment points into their containing line, we use the parametric equation of the line. The parametric equation of a line containing points P_1 and P_2 is

$$X = \lambda P_2 + (1 - \lambda)P_1 \tag{4.1}$$

Assigning values for $\lambda \in [0, 1]$ gives us all the points in the segment bounded by P_1 and P_2. For $\lambda < 0$, the obtained points lie on the *left* side of P_1, while $\lambda > 1$ results in points lying on the *right* side of P_2.

The inverse operation is also possible: given a segment with endpoints P_1 and P_2 and a collinear point Q, the value λ such that $Q = \lambda P_2 + (1 - \lambda)P_1$ can be computed as:

$$\lambda = \frac{(Q - P_1) \cdot (P_2 - P_1)}{\|P_2 - P_1\|^2} \tag{4.2}$$

Note: If Q is not collinear to the segment $\overline{P_1P_2}$, the obtained value for λ does not hold Equation 4.1 for $X = Q$. Instead, the equation holds for the projection of Q onto the line which contains the segment $\overline{P_1P_2}$.

Given two segments $\overline{P_1P_2}$ and $\overline{P_3P_4}$, let λ_1 be the projection of P_1 onto $\overline{P_3P_4}$ and λ_2 be the projection of P_2 onto $\overline{P_3P_4}$. Hence, the intervals $[\lambda_1, \lambda_2]$ and $[0, 1]$ are the input for the range comparison algorithm.

For arcs, the used values for the interval comparison algorithm are the starting and ending angles, thus the conversion from 2D to 1D is not necessary. However, situations when an arc crosses the X^+ axis need to be considered. Given two arcs a_1, a_2 with starting and ending angles α_{s1}, α_{e1}, α_{s2} and α_{e1}, the problem is similar to the range comparison, except when either of the arcs intersects the X^+ axis. In that case, the number-to-interval test is slightly different. Suppose that angles are measured between 0 and 2π:

- If $\alpha_{s1} < \alpha_{e1}$ and $\alpha_{s2} < \alpha_{e2}$, the standard interval comparison can be applied.
- Otherwise, the comparison have to be made according to Table 4.2.

Range comparison and result mapping Once we have compared the ranges, the four possible output tags INTERSECTING, CONTAINED, COINCIDENT and COMMON_ORIGIN, correspond respectively to the irregularity types OVERLAPPING, CONTAINED, DUPLICATED and CONSECUTIVE.

4.4 Irregularity fixing

The irregularity fixing stage considers the set of tagged irregularities. In this section, we explain how to deal with the irregularities from the set of primitives and the tags.

Comparison between a number and an interval [a,b] (for arcs)	
IN	$x > a \wedge x < b$
OUT	$x \leq a \vee x \geq b$
Comparison between two intervals [a,b] and [c,d] (for arcs)	
INTERSECTING	c OUT $(a, b) \wedge d$ IN (a, b) c IN $(a, b) \wedge d$ OUT (a, b)
CONTAINED	c IN $(a, b) \wedge d$ IN (a, b) a IN $(c, d) \wedge b$ IN (c, d)
COINCIDENT	$a = c \wedge b = d$
COMMON_ORIGIN	$b = c$ $a = d$

Table 4.2: Comparison of numbers and intervals for arcs

In the simplest cases, the fixing consists of removing a primitive, while other cases require a substitution of some primitives by a new primitive not included in the original design.

Irregularities tagged as NULL-LENGTH, CONTAINED and DUPLICATED are fixed by suppressing one of the two involved primitives, respectively the null-length, contained or either of them in the duplicated case.

On the other side, irregularities tagged as OVERLAPPING and CONSECUTIVE require the creation of a new primitive from the coordinates of the irregular primitives.

Overlapping Given two overlapping segments $\overline{ab}$ and $\overline{cd}$ such that c IN (a, b), both segments and removed and the segment (a, d) is created.

Consecutive Given two segments $\overline{ab}$ and $\overline{cd}$ such that $b = c$, both segments are removed and the segment (a, d) is created.

Occasionally, these changes may produce new precision related problems; therefore, the process of checking and correcting these situations has to loop automatically over the geometry. The main issue is that the removal of some irregularities could provoke the appearance of new ones. Thus, a one-iteration algorithm which inspects the geometry and removes the irregularities is not sufficient to solve this problem, and therefore it is convenient to apply the detection and fixing of irregularities iteratively, until there are no irregularities detected.

Chapter 5

Local room detection

In our first attempt, we tried to abstract situations where columns and pillars are drawn in the wall layer. Then, a first approach arose: rule-based detection of semantics. We implemented an iterative algorithm that started to traverse walls made up by pairs of lines. Each iteration of the traversal starts from an insert of a door block. When a wall crossing is found, the set of rules is checked in order to determine which rule is triggered. The information about each rule includes the cross type, and the next direction to be followed. Once the traversal arrives to the starting door block insert, a room has been detected and a new traversal starts from other door block insert. The algorithm finishes when there are no more door block inserts to iterate.

The main contribution of this approach is the detection of wall crossings even when there exist close columns. This initial work was published in the Iberoamerican Symposium on Computer Graphics 2009[DGF09]. Nonetheless, some disadvantages were found in this approach: (1) the rule set is very hard to characterize and implement; (2) overtraining arises, i.e., the algorithms work well with test cases similar to the ones used to design the algorithms, but the behavior is unpredictable or bad in other cases; (3) the approach is very influenced by the imperfections from the floor plan.

Due to the disadvantages of the rule-based algorithm, we reached two conclusions: (1) it was necessary a preprocessing of the plans, consisting of detecting and fixing irregularities; and (2) the detection of semantics had to be approached from a global scope, as a computational geometry problem instead of a rule-based reasoning procedure.

In this chapter we summarize the rule-based detection of rooms, as it was the first approach of the detection of semantics. It was related to the goal of storing the main semantic features of a floor into a database. Thus, this work deals with the representation into databases of the detected semantic elements. Below, we briefly introduce the data types included in the designed database.

The *key point* is the fundamental database entity that allows us to organize the information of a floor plan in the database, as all the other features (doors, windows, rooms, etc.) use key points as part of their description; for instance,

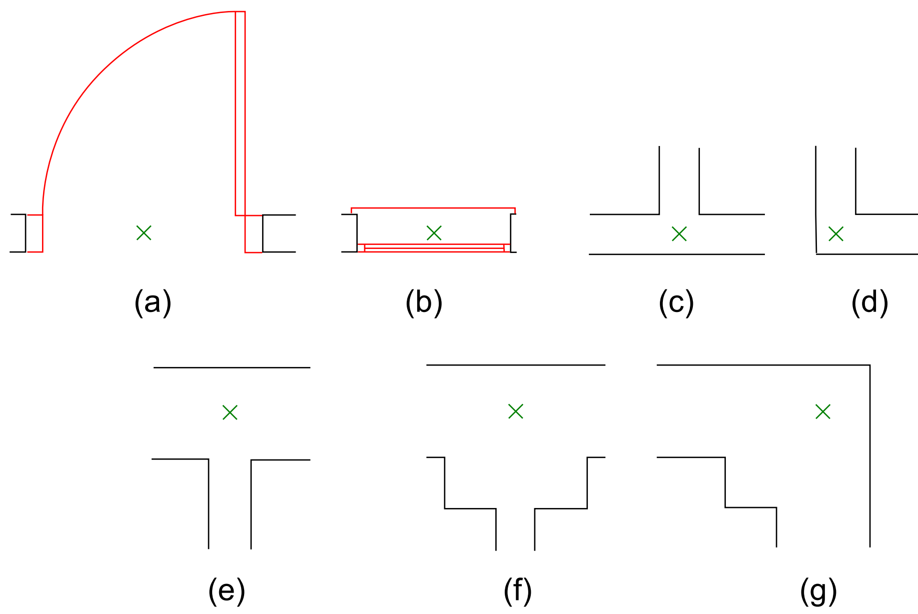


Figure 5.1: Types of key points considered in the system. The stored point is drawn in green: (a) door key point; (b) window key point; (c) T-key point between two partition walls; (d) L-key point between two partition walls; (e) T-key point between a partition wall and an outer wall; (f) irregular T-key points; (g) irregular L-key points.

a window is defined by a key point, its distance to the floor and its width and height, and a room is defined by a sequence of key points that form its perimeter.

Several types of key points are considered (see Figure 5.1): door key point, window key point, T-key point between two partition walls, L-key point between two partition walls, T-key point between a partition wall and an outer wall, and irregular variants of T-key points and L-key points; the two last types are used to model key points where walls touches pillars, as shown in Figures 5.1.f and 5.1.g. A key point is defined by both its coordinates and its type.

Additional relationships are also included in the database in order to model hierarchical behaviors among the stored features. This way, it is possible to obtain topological knowledge by means of appropriate queries.

Two special elements that need to be considered are elevators and flights of stairs. The solution that best suits our needs is to consider them once for each floor of the building; this includes some redundancy in the database, but as there are usually very few elements of this kind in a building floor, it is not significant.

In order to obtain the significant features to build the 3D scene, it is necessary to process the floor plan that contains them. Our system is able to load and process floor plans saved using the DXF file format [Aut11], and store data

related with the significant features of the scene in the database described above. In this chapter, we will focus on this process.

5.1 Constraints on the DXF file format

The basics of the DXF file format were introduced in Section 3.2 (Chapter 3). Here we show the constraints applied in order to make the local room detection algorithm work.

It is strongly recommended structuring the contents of a complex floor plan using layers to group semantically related elements. Therefore, typical architectural designs include separate layers for walls, doors and windows, furniture, plumbing, etc. Doors and windows are usually represented using blocks, while walls are represented using lines and arcs. Our system assumes that the floor plans are composed of layers in this way, and therefore, the user is required to select the layers that contain walls, doors and windows, as well as the names of the blocks used to represent doors and windows.

The DXF file format does not include any topological information other than the data implicit in the polyline entity. Therefore it is necessary to develop algorithms to detect the logical elements that make up the scene (rooms, corridors, flights of stairs, etc.).

5.2 Feature extraction process

The first step of the process is loading and parsing the DXF file. This step requires the user to select the layer and block names that are to be considered.

Once the layers that contain walls, doors and windows are identified, the system processes the walls, and initially stores (1)points, (2)lines as point indices and (3)an oriented bounding box for each block representing a door or a window. This is the starting point for the rest of the detection process.

5.2.1 Wall thickness calculation

The thickness of partition and outer walls is calculated using as a reference the blocks that represent doors and windows in the following way (see Figure 5.2):

- The door blocks are considered to calculate the thickness of partition walls (see Figure 5.2.a). For each door block, a point $\mathbf{P}$ is found such that it is the closest to (probably the same as) the location of the door block. Then, another point $\mathbf{Q}$ is found such that it is the nearest point to $\mathbf{P}$ following the direction of the normal vector $\mathbf{n}$ of the door. Finally, the wall thickness is calculated as the difference $|\mathbf{P} - \mathbf{Q}|$.

  In order to get a unique result for all the partition walls, and discard misleading values due to doors that are placed in outer walls as shown in Figure 5.2.b, the statistical mode of all the differences is considered as the new calculated thickness.

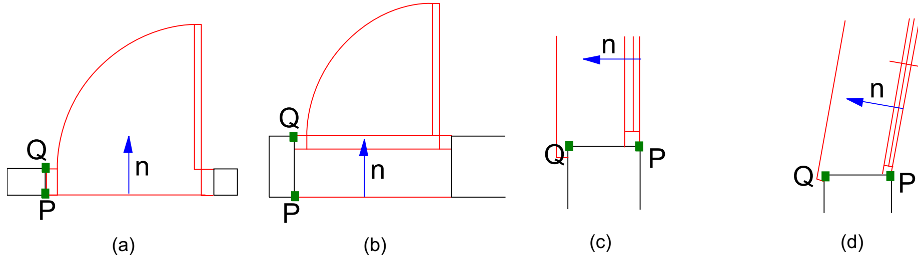


Figure 5.2: (a-b) Thickness calculation situations for partition walls; (c-d) Thickness calculation situations for outer walls (down)

- In order to calculate the thickness of outer walls, the window blocks are considered. The process is similar to the one explained before, as shown in Figure 5.2.c.

Special cases can appear when blocks are not aligned with walls, like Figure 5.2.d. In these cases, the nearest point to **P** such that the angle between vectors **n** and $\mathbf{Q} - \mathbf{P}$ is minimum is considered.

5.2.2 Key point, room and window extraction. Detection rules

Once the wall thickness is computed for both the partition and the outer walls, the next steps are the detection of key points, rooms and windows. We have developed an algorithm to perform these steps in one single process based on a set of rules that identify typical situations that correspond to each of the key point types shown in Figure 5.1.

The algorithm starts by identifying and storing door key points as the locations where door blocks are placed in the floor plan. Then, rooms are identified and stored one after another, and new key points are stored as they are identified during the process.

The system considers a room as a polygon defined by a set of ordered key points, starting from a door key point. From that point, **P** and **Q** are set to the same values used to calculate the thickness of the wall as described above. The following steps of the process consist of determining if a new key point has been found by checking the rules, and then seeking the new values for **P** and **Q**, so that **P** is following the exterior perimeter of the room, and **Q** is following the interior perimeter of the room; this is done by searching the lines that start from the old key points, and finding their opposite ends. The process continues until either the starting door key point is found, and then the room is successfully stored, or no new key point is available, and then it is considered as an unclosed room; at this stage of development, unclosed rooms are discarded by our system.

We characterized six rules to add new key points to the database. These rules consider the position of points **P** and **Q**, and the situation of lines connected

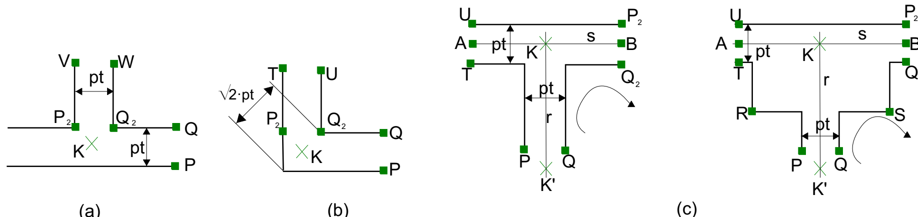


Figure 5.3: Situations where the detection rules are applied. (a) Rule 1. (b) Rule 2. (c) Left: regular case for rule 3. Right: irregular case for rule 3

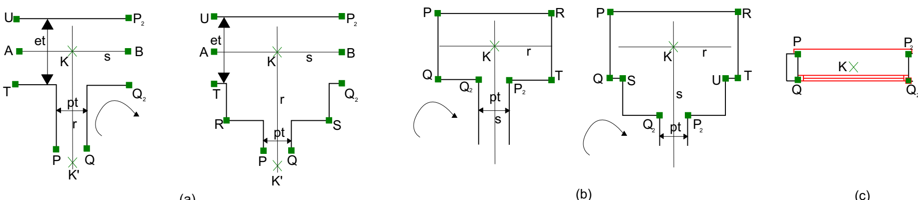


Figure 5.4: Situations where the detection rules are applied. (a) Left: regular case for rule 4. Right: irregular case for rule 4. (b) Left: regular case for rule 5. Right: irregular case for rule 5. (c) Rule 6

to them, together with the points in a certain neighbourhood of **P** and **Q**. In each step of the process, the rules are checked for applicability, given the current situation; when one applicable rule is found, it is applied, and the values of **P** and **Q** are updated. The situations that are correctly detected and handled with these rules are shown in Figures 5.3 and 5.4, and can be described as follows:

1. Rule number 1 is applied when there is a "T-like" intersection between the current wall and a new wall, provided that their thickness *pt* is the same (see Figure 5.3.a). A new T-key point, noted **K**, is added at the intersection of the medial axes of the walls, and the new values for **P** and **Q**, noted $\mathbf{P}_2$ and $\mathbf{Q}_2$ are set.

2. Rule number 2 is applied when there is an "L-like" intersection between the current wall and a new wall, as shown in Figure 5.3.b. In this case, a new L-key point, noted **K**, is added at the intersection of the medial axes of the walls, and **P** and **Q** are updated to the values of points $\mathbf{P}_2$ and $\mathbf{Q}_2$.

3. Rule number 3 is applied when the current wall intersects in a "T"-like way a new wall, provided that the thickness of the walls is the same. This case, showed in Figure 5.3.c left, is solved by adding a new T-key point, noted **K**, at the intersection of the medial axes of the walls. The new values for **P** and **Q** are selected taking into account the orientation of the

triangles $\mathbf{K'KA}$ and $\mathbf{K'KB}$, $\mathbf{K'}$ being the key point obtained in the previous step. The orientation of the triangles is computed using the method described in [FTU95].
Irregular situations, like the one shown in Figure 5.3.c right are solved by discarding intermediate points **R** and **S** after reviewing the neighbourhood of points **P** and **Q**.

4. Rule number 4 is similar to rule number 3. The main difference is that the new wall is an outer wall, and therefore, its thickness is different. See Figure 5.4.a.

5. Rule number 5 combines rules 1 and 3 (see Figure 5.4.b). The current wall (considered an outer wall) is intersected by a new wall (considered a partition wall) in a "T"-like setup. The T-key point **K** is obtained as the intersection of the medial axes of the walls, and the new direction to be followed is selected taking into account the orientation of the triangles formed by the points and the key points. Irregular situations are handled in the same way as in rule 3.

6. Rule 6 is applied to add a new window key point whenever there is no new lines starting from **P** and **Q**, and there is a window block instead. The window key point is computed as the center of the window block, and **P** and **Q** are set to the points after that block (see Figure 5.4.b).

5.2.3 Data storage

During the process explained above, information about key points, doors, windows and rooms is stored in the corresponding tables of the database. At the same time, tuples are inserted as necessary in the database in order to connect each element with their relevant key points, as well as with their ancestors in the logical hierachy. The different thickness values that have been computed before are stored as floor attributes.

As stated before, the use of a database will allow us to include additional, geometry-related data, such as shop names, phone numbers, information desk situations, etcetera. Therefore, it will be feasible to develop some kind of annotated 3D navigation system based on this work.

5.3 3D output generation

Once the floor plan has been processed, a 3D scene can be generated using the data stored in the database. The scene generation process for a given floor is as follows:

- First of all, the information about wall height and thickness is retrieved from the database.
- The next step consists of getting information about the elements in the floor and the key points related to them.
- In order to build the walls that bound each room, a set of prisms are generated: the position and length of each prism is defined by two consecutive key points, while its height and thickness is given by the floor attributes. If one of the key points for a prism is a door or window key point, the corresponding limit of the prism is displaced half the size of the door or window, in order to leave the space needed for that element. After that, new prisms are created in order to represent the pieces of wall over and under (if needed) the door or window.

In order to avoid duplicity of walls, the algorithm keeps record of the pairs of key points whose corresponding wall has already been built.

As the long-term goal of our system is to generate contents for interactive 3D indoor navigation by combining graphics with other information of interest, the scene is stored in an appropriate file format. Up to now, X3D [BD07] format is used, although the design of the application allows the inclusion of modules to store the scene using other file formats, like COLLADA [AB06] or VRML.

5.4 Results

The results obtained for this approach consisted of creating X3D models by extruding the data detected and stored in the database. The scene generation process for a given floor is as follows:

- First of all, the information about wall height and thickness is retrieved from the database.
- The next step consists of getting information about the elements in the floor and the key points related to them.
- In order to build the walls that bound each room, a set of prisms are generated: the position and length of each prism is defined by two consecutive key points. The thickness of the prisms is the same detected during the feature extraction process, while the height needs to be specified by the user.
- If one of the key points for a prism is a door or window key point, some prisms created in order to represent the pieces of wall over the door, and over and under the window. Thus, a hole represents the opening.

In order to avoid duplicity of walls, the algorithm keeps a record of the pairs of key points whose corresponding wall has already been built.

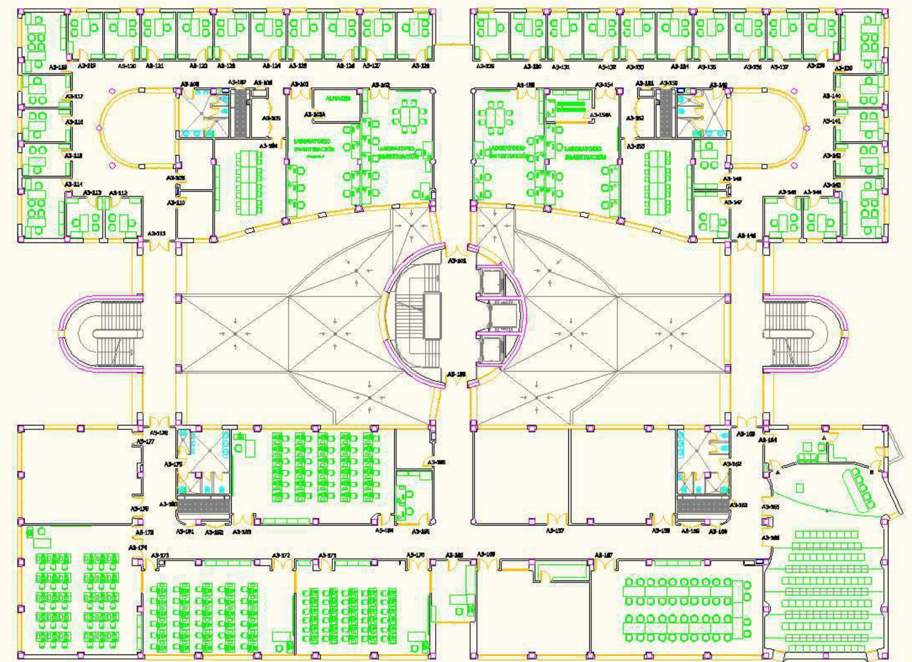


Figure 5.5: A floor plan used in our tests

X3D [BD07] format is used, although the design of the application allows the inclusion of modules to store the scene using other file formats, like COLLADA [AB06] or VRML.

The system has been tested with real architectural floor plans from the University of Jaén (Spain) in DXF format (for instance, see Figure 5.5). The application requires the user to select the layers and the names of the blocks that correspond with the data to be processed, and stores the information in the database as explained before. The database can then be queried for information about the entities that have been detected.

Using the data stored in the database, 3D scenes are built and saved using X3D format. Figure 5.6 shows a zoomed view of a portion of a floor plan, together with the resulting 3D model. The main contribution of this approach is the capacity to detect walls and corners of rooms in spite of the columns close to these corners.

In this part of the work, we implemented an algorithm able to detect rooms from a floor plan using a number of rules that fit different situations in a floor plan drawing. This algorithm worked out correctly for those floor plans that were analyzed to design it.

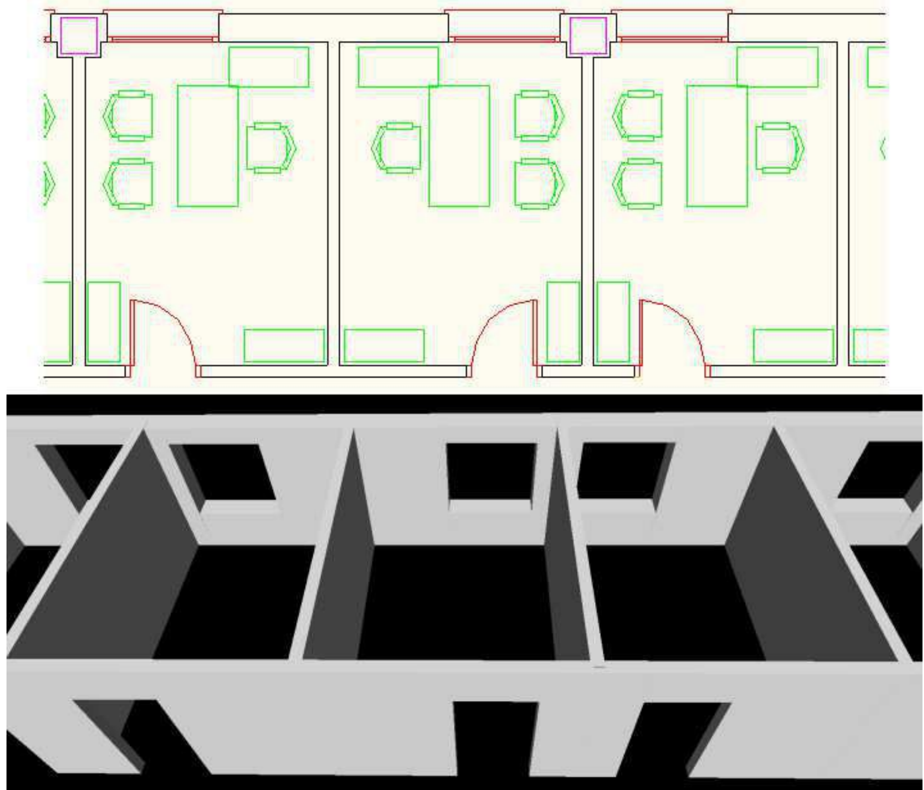


Figure 5.6: Floor plan and corresponding 3D model

Unfortunately, this approach had some limitations related to the overtraining of the algorithm and the difficulty to generalize it to a wide number of floor plans. Consequently, the following approach deals with the room detection problem by *globally* analyzing the floor plan layout. This approach is described in Chapter 6.

Chapter 6

Global room detection

Regarding the local room detection method, some disadvantages that arose are described below:

1. The method does not detect correctly those situations that do not fit any rule.

2. The rule set has been custom-made over real cases. When the method is tested with other cases not used during the rule-set design, the behavior is not fully accurate. In other words, the rule set is overtrained.

3. The implementation of the geometric constraints in the rule set is complex: It is necessary to consider all the symmetric cases. Slight differences in the analyzed situation could not match any rule due to a bad design.

4. In situations where the room shape is a non-convex polygon (with inner angles higher than 180 degrees), there might be errors to determine which is the inner part of the room. The introduced rules above where designed for rectangular rooms (therefore convex).

From all these problems we conclude that it is necessary to deal with the problem using another approach more independent of the concrete design cases. The way to reason and abstract the search for the solution implies recognizing globally the semantic elements from the floor plan. As a result of this recognition process, it is possible to obtain a topology graph.

Definition 1 (Topology graph) *A topology graph is a graph in which each node represents the 2D point on the floor plan of an intersection between walls, or between a wall and an opening (door or window), and each edge represents a wall or an opening.*

Therefore, the topology graph of a story provide us with a minimum model of the story structure. Among all the advantages of this representation, we remark that (1) it contains the minimum set of structural elements needed to

reconstruct the story, reducing the complexity of the initial geometry and (2) from this representation the rooms can be easily obtained. This cannot be done directly from the original floor plan.

The search for the topology graph is a key task in the semantic detection from floor plans. Once the topology graph has been obtained, it can be used to classify the initial geometric information. It is a two-steps process: in a first stage, the initial geometry is used to extract a simplified representation which includes the semantics; in a second stage, the obtained semantics is related with the initial geometry. This has several applications such as classifying the initial geometry into rooms or being able to validate the results.

The global recognition of elements can be decomposed into several stages:

- Wall detection: This problem consists of inspecting all the primitives that belong to the wall drawing in order to detect thick walls. In order to this, we introduce the Wall Adjacency Graph (WAG) data structure. The wall detection result is an early version of the topology graph: an unconnected graph with each node representing the start or end point of a wall, and each edge representing a wall.

- Opening detection: In this stage, the openings are found in the floor plan drawing. For each found opening, its surrounding walls are then detected. Consequently, the initial topology graph is completed with a number of edges representing the openings, although it remains unconnected.

- Search for joint points amid walls: before this stage, the unconnected graph represents walls and openings but it lacks joint points amid walls. In the last stage, the goal is to find the intersection points amid walls. In order to find them, we have followed variations of the point clustering strategy. More recently, we have designed the so-named *line growing algorithm*. It is intended to be a generalization of the former approaches. As a result of this stage, the topology graph is now finished. In an ideal scenario, free of errors, and supposed that the processed drawing does not contain unconnected parts of the same building, the obtained graph must have one connected component.

The three stages above enumerated allow us to obtain the topology graph. The stages enumerated below allow to bind the topology graph with the initial geometry:

- Search for room outer polygons: over the topology graph, we can find the graph circuits that do not contain any other circuit. The main goal of this algorithm is to recognize all the rooms and corridors that make up the story.

- Assignment of wall lines to outer polygons: the lines representing walls in the initial geometry can be assigned to the detected rooms using point-in-polygon tests.

- Search for the inner polygons: in this stage, starting from the wall lines of the original geometry that have been assigned to each room, the inner polygons are built. This problem is more complex than the previous one and consists of finding a set of lines to be added in order to get closed polygons. These polygons, when correctly built, correspond to the physical space of the rooms. In order to get the inner polygons, we use a grid. This simplifies the characterization of the problem, specially in those cases where the walls are merely orthogonal.

In the following sections we discuss each of the above summarized problems, together with some techniques and solutions proposed along the research work to solve them.

6.1 Wall detection

In this section we address the wall detection problem. We consider two types of walls: (1) walls with rectangular shape, made up in the drawing by straight and parallel line segments; and (2) curved walls, made up in the drawing by concentric circle arcs. Other types, such as variable thickness walls or those made up by other primitives different from line segments or circle arcs are omitted from the general analysis since they are less usual in real cases.

The description of the problem and its solution are explained for the rectangular walls. The solution can be trivially extrapolated to curved walls made up by concentric circle arcs.

As described before, every wall in a vector drawing is represented by primitive pairs (two parallel straight segments or two concentric arcs). Nevertheless, the relationship between drawing primitives and walls is not necessarily bijective. There can exist line segments traversing more than one contiguous rooms, and parallel to other segments from these rooms. For instance, the scenario in Figure 6.1 arises when there exists a corridor with several contiguous rooms. In this case, the segment a belongs simultaneously to several rooms.

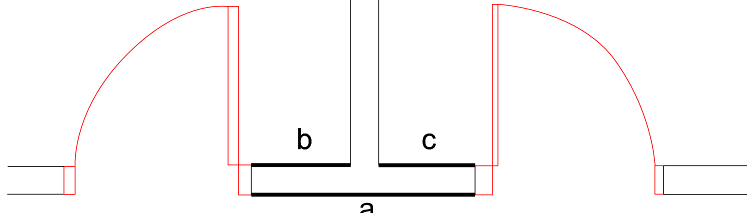


Figure 6.1: The mapping between pairs of segments and walls is not bijective, e.g. the walls ab and ac share the segment a.

Moreover, there is not any information about which segments match to make up a wall (ab or ac in Figure 6.1). In an intuitive way, the process for detecting walls from a set of segments involves three steps:

1. Select those layers from the drawing that contain the walls (input provided by the user).
2. Among all the segments from the selected layers, determine which pairs are parallel, closer than the thickness of the walls, and overlapping when projected onto each other. We name them as wall-prone pairs of segments. The thickness of walls can be automatically estimated, as shown in Chapter 5.
3. Split the segments at the projection points to obtain rectangular-shaped walls. The resulting segments which do not overlap are discarded. This process is repeated until there are no more segments to be split.

Next, we introduce some definitions to do a formal description of the wall detection process, and show some examples to illustrate it. The wall detection process involves searching for *wall-prone* pairs of segments using two thresholds min and max (minimum and maximum wall thickness, automatically estimated as shown in Chapter 5), and split them in order to obtain wall pairs. The definitions for wall-prone pair and wall-pair are as follows:

Definition 2 (Wall-prone pair of line segments) *Let a and b be line segments in $\mathbb{R}^2$. Let l and m be the lines containing a and b respectively, and a' and b' the projections of a and b onto m and l. Given two fixed thresholds min and max, the pair (a, b) is wall-prone (represented by the predicate $prone(a, b, min, max)$) if and only if all these conditions are satisfied:*

C1. a and b are parallel: $a \parallel b$

C2. a' and b (and therefore b' and a) overlap: $a' \cap b \neq \emptyset$

C3. The distance between l and m is between both thresholds: $d(l, m) \in [min, max]$

Note: In this context the line segments are considered open (their end points do not belong to them). This is done to avoid the special case where two line segments with consecutive projections hold condition C2.

Definition 3 (Wall pair of line segments) *Two line segments a and b are said to form a wall pair, given two fixed thresholds min and max (represented by the predicate $wall(a, b, min, max)$), if they satisfy the conditions to be a wall-prone pair, and their projections onto the line that contain each other match. Therefore, a new, more restrictive condition is added to C1,C2 and C3:*

C4. a' and b (and therefore b' and a) are the same: $a' = b$

Some properties of these predicates are easy to deduce:

- *prone* does not depend on the segment order:

$$prone(a, b, min, max) \Leftrightarrow prone(b, a, min, max)$$

- *wall* does not depend on the segment order:

$$wall(a, b, min, max) \Leftrightarrow wall(b, a, min, max)$$

- *wall* is a restriction of *prone*:

$$wall(a, b, min, max) \Rightarrow prone(a, b, min, max)$$

Given the above definitions, this is the outline of the algorithm to compute the walls represented by a set of line segments in a floor plan (see example in Figure 6.2):

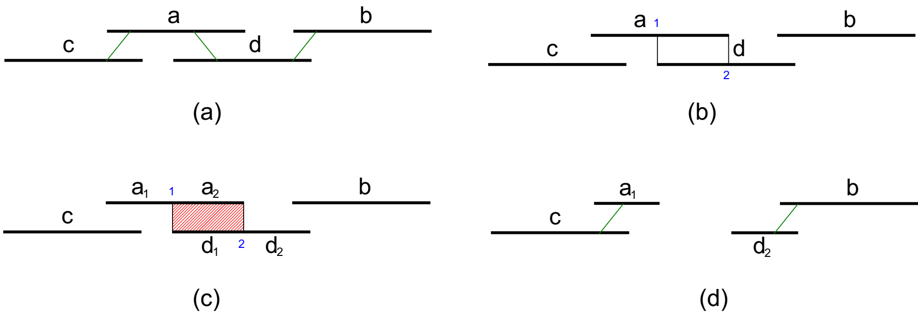


Figure 6.2: Iteration of the wall detection algorithm. (a) Initial set of segments and relations. (b) Projection of the endpoints. (c) Segment splitting and wall extraction. (d) Updated segments and relations

(1) Find all the wall-prone pairs. In Figure 6.2.1, these pairs are (a, c), (a, d) and (b, d).

(2) For each wall-prone pair, apply the following steps (in this example, we take the pair (a, d)):

(2.a) Compute the projections of the endpoints of one segment on the other segment. Applying this step to the wall-prone pair (a, d) results in points 1 and 2, as shown in Figure 6.2.2.

(2.b) Split each segment using the computed projections and check the new set of segments for wall pairs: some segments will form wall pairs while the rest will become *single* (their projections do not overlap). In Figure 6.2.3, the resulting segments are a_1, a_2, d_1 and d_2. The new detected wall pair is (a_2, d_1), and the single segments are a_1 and d_2.

(2.c) Update the set of segments by removing the old segments (a and d in this case) and adding the single ones (a_1 and d_2). Then, update the wall-prone pair set; in this example, wall-prone pairs (a_1, c) and (b, d_2) are added, as shown in Figure 6.2.4.

Each time a wall-prone pair is detected and processed, the set of segments is modified, as well as the set of wall-prone and wall relationships. This makes difficult to define how to iterate over the set of segments in an easy way. For instance, when a segment is split, the algorithm needs to check which resulting segments are assigned to wall-prone relationships, check which of the segments had already been processed, etc.

Therefore, in order to make easier to process the wall and wall-prone relationships amid segments, and keep record of the hierarchical relationships between each line segment and the pieces it is split into, the Wall Adjacency Graph (WAG) is proposed as a data structure to give support to this. The following sections give a detailed description of this structure and its application to the problem.

6.1.1 Wall Adjacency Graph (WAG)

The Wall Adjacency Graph (WAG) is a graph whose nodes represent the line segments from a floor plan that belong to the wall layers, and whose edges represent relationships among those segments. In order to represent the walls drawn in a floor plan, three kind of mathematical relations between segments are defined as follows:

Definition 4 (Wall-prone relation) *Given a finite set of line segments $\mathcal{A}$ and two fixed thresholds min and max, the wall-prone relation over $\mathcal{A}$ is defined as the set*

$$PR(\mathcal{A}, min, max) = \{(a, b) \in \mathcal{A} \times \mathcal{A} \mid prone(a, b, min, max)\}$$

Definition 5 (Wall relation) *Given a finite set of line segments $\mathcal{A}$ and two fixed thresholds min and max, the wall relation over $\mathcal{A}$ is defined as the set*

$$W(\mathcal{A}, min, max) = \{(a, b) \in \mathcal{A} \times \mathcal{A} \mid wall(a, b, min, max)\}$$

As the relations defined above are based on Definitions 2 and 3, the following properties hold:

- $PR(\mathcal{A}, min, max)$ is symmetric:
$$(a, b) \in PR(\mathcal{A}, min, max) \Leftrightarrow (b, a) \in PR(\mathcal{A}, min, max)$$
- $W(\mathcal{A}, min, max)$ is symmetric:
$$(a, b) \in W(\mathcal{A}, min, max) \Leftrightarrow (b, a) \in W(\mathcal{A}, min, max)$$
- $W(\mathcal{A}, min, max) \subseteq PR(\mathcal{A}, min, max)$

As the wall-prone pairs are being processed, their segments are split into pieces. For each segment a, the set of pieces it is split into form a partition $P(a)$ of that segment, because they do not overlap.

In order to store in the WAG the information about partitions, one more relation is defined as follows:

Definition 6 (Hierarchical relation) *Given a finite set of line segments $\mathcal{A}$, the hierarchical relation over $\mathcal{A}$ is defined as the set*

$$H(\mathcal{A}) = \{(a, b) \in \mathcal{A} \times \mathcal{A} \mid b \in P(a)\}$$

Unlike the wall and wall-prone relations, the hierarchical relation is obviously not symmetric.

Once all the elements that are involved in the WAG are defined, a formal definition of this structure follows:

Definition 7 (Wall Adjacency Graph (WAG)) *Given a finite set of line segments $\mathcal{A}$ and two fixed thresholds min and max, the Wall Adjacency Graph (WAG) associated with $\mathcal{A}$ is an undirected graph $G(\mathcal{A}, min, max) = (V, E)$ where the vertex set is $V = \mathcal{A}$ and the edge set is $E = PR(\mathcal{A}, min, max) \cup H(\mathcal{A})$.*

That is, the vertex set contains the segments, while the edge set contains typed edges (wall-prone and hierarchical). For a given floor plan, its WAG is therefore formed by the line segments it contains, together with the relationships amid them. A WAG is not necessarily connected, and this is not a requirement for the algorithms to work successfully.

Due to the fact that $W(\mathcal{A}, min, max) \subseteq PR(\mathcal{A}, min, max)$, it is necessary to distinguish the set of *strict* wall-prone segment pairs that do need to be processed to get wall pairs; therefore, the set $\overline{W}(\mathcal{A}, min, max)$ is defined as the complement of $W(\mathcal{A}, min, max)$ with respect to $PR(\mathcal{A}, min, max)$, i.e. $\overline{W} = PR(\mathcal{A}, min, max) \setminus W(\mathcal{A}, min, max)$.

A basic example is shown in Figure 6.3. It shows a set of segments and the WAG built from it, taking into account the *min* and *max* thickness thresholds. Let us suppose that the distance between the two rows of parallel segments (Figure 6.3.a) is a value $\varepsilon \in [min, max]$.

Example 1 (Initial WAG) *Figure 6.3 shows an example of (a) an initial set of line segments and (b) its associated WAG. Edges representing wall-prone relations are drawn with single lines, while edges representing wall relations appear as double lines. The elements defining the WAG are:*

$$\begin{aligned} \mathcal{A} &= \{a_1, a_2, a_3, a_4, a_5\} \\ PR(\mathcal{A}, min, max) &= \{(a_1, a_4), (a_2, a_4), (a_3, a_5)\} \\ H(\mathcal{A}) &= \emptyset \end{aligned}$$

The pairs in $PR(\mathcal{A}, min, max)$ can be grouped into wall and strict wall-prone sets:

$$\begin{aligned} W(\mathcal{A}, min, max) &= \{(a_3, a_5)\} \\ \overline{W}(\mathcal{A}, min, max) &= \{(a_1, a_4), (a_2, a_4)\} \end{aligned}$$

As a preliminary step, it is necessary to build the initial WAG for the segments. Its elements ($\mathcal{A}$, $PR(\mathcal{A}, min, max)$ and $H(\mathcal{A})$) are assigned as follows:

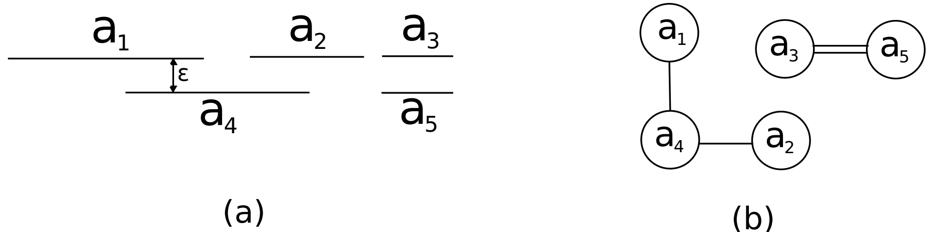


Figure 6.3: Example of (a) a set of line segments (b) its associated WAG using thresholds min and max. Single lines represent wall-prone relations, while double lines represent wall relations.

- $\mathcal{A}$ is defined as the initial set of segments.
- $H(\mathcal{A})$ is empty.
- To build the initial set of wall-prone relations $PR(\mathcal{A}, min, max)$, each possible pair of line segments (a, b) has to be analyzed to determine whether it holds the conditions to be a wall or a wall-prone pair. For the convenience of the wall detection algorithm, this set is subdivided into the subsets $W(\mathcal{A}, min, max)$ and $\overline{W}(\mathcal{A}, min, max)$, defined previously.

Once the WAG has been created, the wall-prone pairs from the subset $\overline{W}(\mathcal{A}, min, max)$ are processed in turn to obtain wall pairs. There are nine possible types of wall-prone pairs; the way they are processed will be detailed later.

The processing of each wall-prone pair results in a series of changes in the WAG:

- It is necessary to split the segments to obtain wall and wall-prone pairs; some nodes representing the new segments would be therefore added to the WAG. The set of newly added nodes will be notated as $\mathcal{A}^+$.
- The inclusion of new nodes in the WAG implies a change in the edge set in order to represent the new wall and wall-prone pairs. This involves that some wall-prone edges are deleted and other are added. The sets PR^- and PR^+ represent respectively the deleted and added edges. Moreover, one wall-prone pair is always deleted to produce a wall pair, and therefore a wall edge w has also to be added to the WAG.
- In order to keep track of the origin of each node, edges representing the hierarchical relation between the nodes corresponding to the segments that are split and the nodes representing the pieces they are split into are added to the graph. The set H^+ represents these edges.

When this process is completed, there must only exist wall pairs in the WAG, therefore $PR(\mathcal{A}, min, max) \equiv W(\mathcal{A}, min, max)$. Algorithm 6.1 formally

describes the wall detection procedure using the WAG as supporting data structure.

Algorithm 6.1: Wall detection algorithm

```
   Input: 𝒜 ← the set of edges
   min← the minimum threshold for the wall thickness
   max← the maximum threshold for the wall thickness
   Output: G = (𝒜, W(𝒜, min, max) ∪ H(𝒜)) ← the resulting WAG
 1 begin
 2     W̄(𝒜, min, max) ← {{a, b} | a, b ∈
       𝒜 ∧ prone(a, b, min, max) ∧ ¬wall(a, b, min, max)}
 3     W(𝒜, min, max) ← {{a, b} | a, b ∈ 𝒜 ∧ wall(a, b, min, max)}
 4     H(𝒜) ← ∅
 5     foreach {a, b} ∈ W̄(𝒜, min, max) do
 6         Study the layout of a and b
 7         Build 𝒜⁺, PR⁻, PR⁺, w and H⁺ depending on the layout of a
           and b
 8         𝒜 ← 𝒜 ∪ 𝒜⁺
 9         W(𝒜, min, max) ← W(𝒜, min, max) ∪ {w}
10         W̄(𝒜, min, max) ← W̄(𝒜, min, max) \ PR⁻
11         W̄(𝒜, min, max) ← W̄(𝒜, min, max) ∪ PR⁺
12         H(𝒜) ← H(𝒜) ∪ H⁺
13     end
14     return G = (𝒜, W(𝒜, min, max) ∪ H(𝒜))
15 end
```

The result of processing each wall-prone pair depends on the relative position between the segments that make up the pair. Figure 6.4 shows the nine possible cases that may appear. The changes that have to be applied to the WAG are different in each case. In order to make the description of these changes easier, the set of wall-prone edges incident to a node a will be notated as $E(a)$, while the set of nodes adjacent to a node a will be notated as $V(a)$. The formal definition of these sets follows:

$$\begin{aligned} E(a) &= \{(a,x) \mid x \in \mathcal{A} \wedge (a,x) \in \overline{W}(\mathcal{A}, min, max)\} \\ V(a) &= \{x \in \mathcal{A} \mid (a,x) \in \overline{W}(\mathcal{A}, min, max)\} \end{aligned}$$

Figure 6.5 shows how the segments from each particular layout are split to get wall pairs. The nodes representing segments in green in the figure are connected using the new wall edges that are added to the WAG, while the nodes representing segments in red in the figure are connected with the wall-prone edges that are modified. Table 6.1 shows a detailed description of the contents of $\mathcal{A}^+$, PR^-, PR^+, w and H^+ for each particular case.

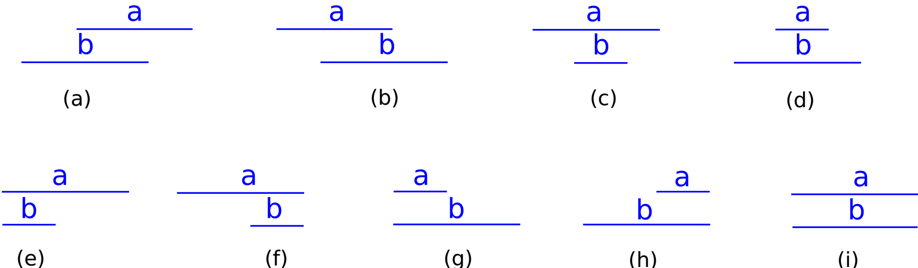


Figure 6.4: Layouts of wall-prone pairs of line segments: (a) *intersection1*, (b) *intersection2*, (c) *contained1*, (d) *contained2*, (e) *common1*, (f) *common2*, (g) *common3*, (h) *common4* and (i) *matching*

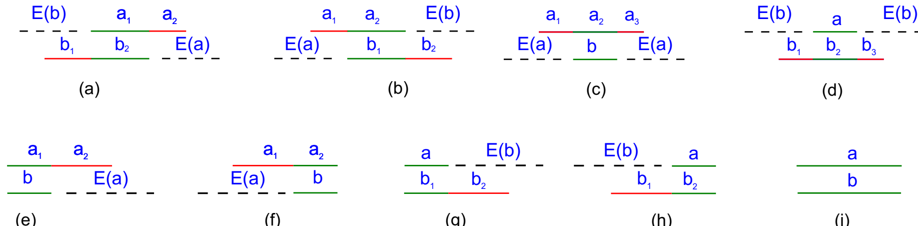


Figure 6.5: Segment splitting for each particular layout from Figure 6.4

Layout	$\mathcal{A}^+$	w	H^+	PR^-	PR^+
Intersection1 (Figure 6.5.a)	$\{a_1, a_2, b_1, b_2\}$	(a_1, b_2)	$\{(a, a_1), (a, a_2), (b, b_1), (b, b_2)\}$	$\{(a, b)\} \cup E(a) \cup E(b)$	$\{(a_2, x) \vert x \in V(a)\} \cup \{(b_1, x) \vert x \in V(b)\}$
Intersection2 (Figure 6.5.b)	$\{a_1, a_2, b_1, b_2\}$	(a_2, b_1)	$\{(a, a_1), (a, a_2), (b, b_1), (b, b_2)\}$	$\{(a, b)\} \cup E(a) \cup E(b)$	$\{(a_1, x) \vert x \in V(a)\} \cup \{(b_2, x) \vert x \in V(b)\}$
Contained1 (Figure 6.5.c)	$\{a_1, a_2, a_3\}$	(a_2, b)	$\{(a, a_1), (a, a_2), (a, a_3)\}$	$\{(a, b)\} \cup E(a)$	$\{(a_1, x) \vert x \in V(a) \wedge prone(a_1, x)\} \cup$ $\cup \{(a_3, x) \vert x \in V(a) \wedge prone(a_3, x)\}$
Contained2 (Figure 6.5.d)	$\{b_1, b_2, b_3\}$	(a, b_2)	$\{(b, b_1), (b, b_2), (b, b_3)\}$	$\{(a, b)\} \cup E(b)$	$\{(b_1, x) \vert x \in V(b) \wedge prone(b_1, x)\} \cup$ $\{(b_3, x) \vert x \in V(b) \wedge prone(b_3, x)\}$
Common1 (Figure 6.5.e)	$\{a_1, a_2\}$	(a_1, b)	$\{(a, a_1), (a, a_2)\}$	$\{(a, b)\} \cup E(a)$	$\{(a_2, x) \vert x \in V(a)\}$
Common2 (Figure 6.5.f)	$\{a_1, a_2\}$	(a_2, b)	$\{(a, a_1), (a, a_2)\}$	$\{(a, b)\} \cup E(a)$	$\{(a_1, x) \vert x \in V(a)\}$
Common3 (Figure 6.5.g)	$\{b_1, b_2\}$	(a, b_1)	$\{(b, b_1), (b, b_2)\}$	$\{(a, b)\} \cup E(b)$	$\{(b_2, x) \vert x \in V(b)\}$
Common4 (Figure 6.5.h)	$\{b_1, b_2\}$	(a, b_2)	$\{(b, b_1), (b, b_2)\}$	$\{(a, b)\} \cup E(b)$	$\{(b_1, x) \vert x \in V(b)\}$
Matching (Figure 6.5.i)	$\emptyset$	-	$\emptyset$	$\emptyset$	$\emptyset$

Table 6.1: WAG modifications for each layout in Figure 6.5.

The following example focuses on the processing of a wall-prone edge of a WAG.

Example 2 (Wall-prone edge processing) *Figure 6.6.2 shows the initial WAG for the set of line segments in Figure 6.6.1. This WAG has three wall-prone edges which must be processed sequentially to find the corresponding wall pairs.*

The segment pair represented by edge (a, d) (in red in Figure 6.6.2) corresponds to the case intersection2 *(Figure 6.5.b). Then, the endpoints of segments a and d are projected onto each other, resulting in points 1 and 4, and these points are used to split the segments into a_1, a_2, d_1 and d_2. Applying the definition of sets $\mathcal{A}^+$, PR^-, PR^+, w and H^+ from Table 6.1 results in:*

- $\mathcal{A}^+ = \{a_1, a_2, d_1, d_2\}$
- $w = (a_2, d_1)$
- $H^+ = \{(a, a_1), (a, a_2), (d, d_1), (d, d_2)\}$
- $PR^- = \{(a, d), (a, c), (b, d)\}$
- $PR^+ = \{(a_1, c), (d_2, b)\}$

Figure 6.6.3 shows the WAG after the set H^+ and the wall edge w have been added, while Figure 6.6.4 shows the WAG after the set of edges PR^- has been deleted, and the set of edges PR^+ has been added. Finally, the resulting set of segments is shown in Figure 6.6.5.

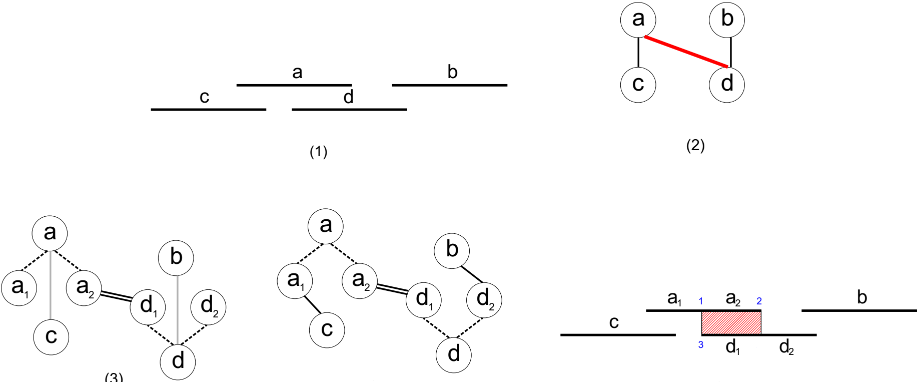


Figure 6.6: Example of one step of the wall detection algorithm. (1) Initial set of segments (2) Wall-prone edge (a, d) is processed. (3) Nodes a and d are split and the wall-prone edge is replaced by a wall edge. (4) The rest of the wall-prone edges adjacent to a and d are reassigned to their children. (5) Set of segments after processing (a, d)

6.1.1.1 Properties

After describing the WAG algorithm, we must remark some of its properties. These properties are necessary as an introduction to the study of the algorithm's execution time.

Property 1: The initial WAG does not contain hierarchical edges: This is due to the fact that the segments from the set have not been split yet

Property 2: If only the hierarchy edges of a WAG are considered, we have a set of trees. These trees will be called *hierarchy trees*, and its nodes will be called *hierarchy nodes*

Property 3: A wall-prone edge is never incident to a non-leaf node of a hierarchy tree. The reasons are the following: (1) an initial WAG does not contain any hierarchy tree as stated in Property 3, and consequently it does not contain any non-leaf hierarchy nodes. Therefore any wall-prone edge in an initial WAG is incident to a non-leaf node. (2) When a segment is split and new nodes and hierarchy edges are added to the WAG, the resulting hierarchy tree has as its root the node that represents the split segment. All the wall-prone edges incident to the root are changed for wall-prone edges incident to the leaf nodes.

Property 4: Two nodes connected with a wall edge can not be connected with any other node with a wall-prone edge, because their projections onto each other overlap (see (i) in Figure 6.4). For other segment layouts, check Section 6.1.1.2.

Property 5: As a consequence of the above points, nodes connected with wall edges are always leaf nodes, because the segments they represent will not need to be split.

Although only leaf nodes are necessary once the algorithm has processed the WAG to get the walls, non-leaf nodes will be kept for further queries about the original geometry.

The following list shows other remarks about the algorithm's running time:

1. The algorithm execution always ends, because one wall-prone edge is removed during each iteration, and the other wall-prone edges incident to the nodes that are connected by the edge that is removed are replaced by the same amount of wall-prone edges. Thus, the number of iterations equals the number of wall-prone edges, and therefore, the execution time of the algorithm is $O(n)$ with respect to the initial number of wall-prone edges. Operations executed on each iteration are $O(1)$, as they do not depend on the size of the set of wall-prone edges.

2. The initialization of $W(\mathcal{A}, min, max)$ and $\overline{W}(\mathcal{A}, min, max)$ is $O(n^2)$ with respect to the number of segments in the floor plan, since all the pairs of segments have to be studied. However, for real floor plans the number of

line segments is of the order of 10^3, and for this size of the problem, the initialization of the WAG is executed approximately in half a second, and the wall detection algorithm is executed in a few milliseconds with current CPU's.

3. The set of walls can be obtained directly from the set of wall edges. Each wall edge correspond to a wall.

6.1.1.2 Generalized Wall Adjacency Graph (GWAG)

The wall detection algorithm is based on the assumption that no more than two parallel segments are close enough to be considered as wall-prone pairs. However, there are situations where more than two parallel segments may appear in a so small enough range that the algorithm cannot handle appropriately, producing runtime inconsistencies. See Figure 6.7.a. The initial WAG for this segment layout is $G = (\mathcal{A}, PR(\mathcal{A}, min, max), H(\mathcal{A}))$, where $\mathcal{A} = \{a, b, c\}$, $PR(\mathcal{A}, min, max) = \{\{a, b\}, \{a, c\}\}$ and $H(\mathcal{A}) = \emptyset$, as shown in Figure 6.7.b. $PR(\mathcal{A}, min, max)$ is in turn divided into

$$W(\mathcal{A}, min, max) = \emptyset$$

and

$$\overline{W}(\mathcal{A}, min, max) = PR(\mathcal{A}, min, max)$$

.

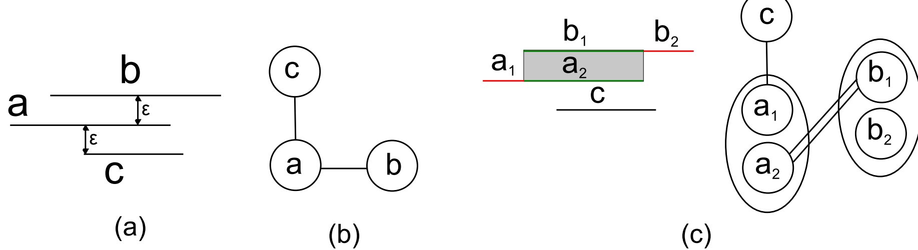


Figure 6.7: (a) Three parallel segments. (b) Initial WAG. (c) Result of applying the wall detection algorithm to the wall-prone edge $\{a, b\}$ (*Note: the hierarchical relations are represented here as ovals that group the nodes that represent the pieces a segment has been split into*)

The first loop iteration of the wall detection algorithm takes the wall-prone edge $\{a, b\}$ and applies the appropriate rule to modify the graph structure as described previously. The result of these modifications is shown in Figure 6.7.c,

and the resulting WAG elements are:

$$\begin{aligned}\mathcal{A} &= \{a, a_1, a_2, b, b_1, b_2, c\}\\ \overline{W}(\mathcal{A}, min, max) &= \{\{a_1, c\}\}\\ W(\mathcal{A}, min, max) &= \{\{a_2, b_1\}\}\\ H(\mathcal{A}) &= \{(a, a_1), (a, a_2), (b, b_1), (b, b_2)\}\end{aligned}$$

This WAG contains two semantic errors with respect to the segment layout after splitting segments a and b:

- The wall-prone edge $\{a_1, c\}$ has been added to the graph. However, these segments do not form a wall-prone pair.
- The segments a_2 and c form a wall-prone pair, but there is no wall-prone edge in the graph to represent this situation.

There are two possible strategies to appropriately handle situations like the one described above, as it is not reasonable to force the user to create floor plans taking into account this issue:

(1) Study situations with multiple segments making up wall-prone pairs, like the ones shown in Figure 6.7, that lead to a wrong initial WAG, in order to set up an automatic filter for the segments from the floor plan. This strategy is overviewed in Section 6.1.1.3

(2) Modify the WAG structure and the wall processing algorithm, so that more than one partition per segment is accepted. This strategy leads us to introduce the generalized WAG, covered in section 6.1.1.4.

6.1.1.3 Handling multiple parallel, close enough segments

It can be observed that when there exist narrow empty spaces between close walls, the corresponding WAG contains connected components whose nodes represent segments contained in more than two parallel lines. Thus, we can assume that:

1. A WAG connected component always represents an even number of parallel segments (except for irregular designs). Each pair of segments thus determine either the interior of a wall or an exterior area between close walls.
2. There is always an empty space (typically a room or a corridor) between two walls; respectively, there is always a wall between two empty spaces.
3. When a pair of segments is wrongly considered as a wall-prone pair, the initial WAG will contain an edge that has to be removed.

Therefore, we can apply a process based on the Jordan curve theorem [Sal78] to remove wrong wall-prone edges from the initial WAG (see Figure 6.8):

1. For each WAG connected component, a ray perpendicular to the segments represented by its nodes is traced from an external point. Each intersection is labeled as positive (incoming) or negative (outgoing). The first intersection is considered always as positive (Figure 6.8.b).

2. The corresponding WAG wall-prone edges are labeled according to the sign of their intersections with the ray (Figure 6.8.c).

3. Wall-prone edges labeled as negative are removed from the WAG (Figure 6.8.d).

After this process, the wall detection algorithm can be executed on the WAG without runtime inconsistencies. However, this approach has significant drawbacks:

- The WAG needs to be analyzed after its construction to detect connected components whose nodes represent segments contained in more than two parallel lines. This additional analysis can be complex, and reduces the advantages of constructing a scenario-independent WAG.

- As the starting point of the ray has to be changed for each WAG connected component, it is not always a straightforward task to place it correctly. Computing correct locations for the starting points implies additional geometrical analysis of the floor plan, therefore increasing the execution time. Additionally, this problem is difficult to characterize and solve. A more general solution is next introduced in Section 6.1.1.4.

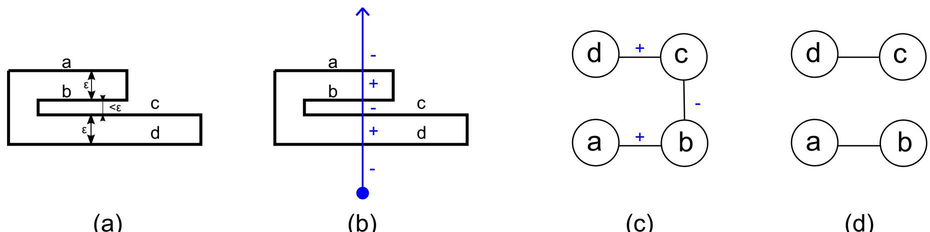


Figure 6.8: (a) Narrow spaces between close walls can lead to a wrong WAG; (b) Jordan curve theorem based solution: ray tracing and intersection labeling; (c) labeled WAG; (d) corrected WAG.

6.1.1.4 Generalizing the wall detection process

The problem described above with a segment layout like the one shown in Figure 6.7.a is due to the fact that each iteration of the wall detection algorithm removes from the WAG those wall-prone edges incident to the node that represents the segment that is split, and substitutes them by wall-prone edges connecting the

nodes that represent the pieces the segment is split into, as shown in Figure 6.7.c. In order to avoid these problems, the WAG structure and the wall detection algorithm are modified in the following way:

- The only wall-prone edge that is removed in each iteration of the algorithm will be the one that is being processed. Therefore, for each iteration of the algorithm, $PR^- = \{a, b\}$ (given the notation in Table 6.1).
- The graph allows more than one partition for the same segment.
- Wall-prone edges are allowed to connect non-leaf nodes.

Given these modifications, the resulting graph is notated as *Generalized Wall Adjacency Graph* (GWAG).

Using the GWAG and the modified wall detection algorithm, the segment layout shown in Figure 6.7.a is handled as shown in Figure 6.9. The elements of the generalized WAG after processing the wall-prone edge $\{a, b\}$ are the following (Figure 6.9.a):

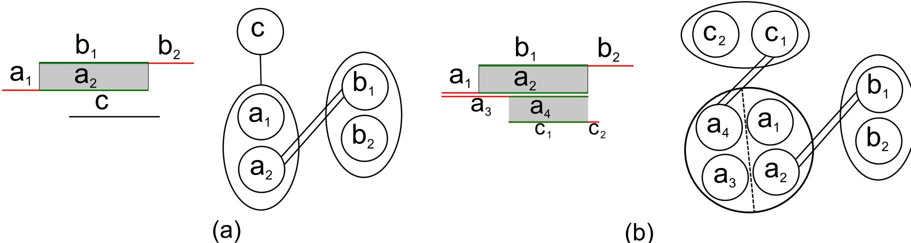


Figure 6.9: Processing the segment layout from Figure 6.7.a using a generalized WAG. (a) Result of processing the edge $\{a, b\}$. (b) Result of processing the edge $\{a, c\}$

$$
\begin{aligned}
\mathcal{A} &= \{a, a_1, a_2, b, b_1, b_2, c\} \\
\overline{W}(\mathcal{A}, min, max) &= \{\{a, c\}\} \\
W(\mathcal{A}, min, max) &= \{\{a_2, b_1\}\} \\
H(\mathcal{A}) &= \{(a, a_1), (a, a_2), (b, b_1), (b, b_2)\}
\end{aligned}
$$

Finally, the processing of the wall-prone edge $\{a, c\}$ results in the situation shown in Figure 6.9.b. The elements of the final generalized WAG are:

$$
\begin{aligned}
\mathcal{A} &= \{a, a_1, a_2, a_3, a_4, b, b_1, b_2, c, c_1, c_2\} \\
\overline{W}(\mathcal{A}, min, max) &= \emptyset \\
W(\mathcal{A}, min, max) &= \{\{a_2, b_1\}, \{a_4, c_1\}\} \\
H(\mathcal{A}) &= \{(a, a_1), (a, a_2), (a, a_3), (a, a_4), \\
&\quad (b, b_1), (b, b_2), (c, c_1), (c, c_2)\}
\end{aligned}
$$

The changes made to the wall detection algorithm do not substantially change its behavior:

1. The algorithm execution always ends, because only one wall-prone edge is removed in each iteration of the algorithm. The number of iterations of the algorithm is therefore equal to the number of wall-prone edges, and the execution time is $O(n)$. However, the time is expected to be slightly slower than the time taken to process the *old* WAG, because the amount of graph changes per iteration is smaller.

2. Each node from the initial graph can be the root of *several* subtrees, each of them representing one partition of the segment represented by the root node. Wall edges always connect leaf nodes, while wall-prone edges can connect any node.

3. The set of walls can be obtained directly from the set of wall edges.

4. Possible graph cliques would be processed without problems, as the generalization of the WAG allow the presence of more than two parallel segments in a small region of a floor plan. This could indeed result in a graph clique, but the possibility of creating more than one partition for a segment allows the algorithm to cleanly process the segments.

6.1.1.5 WAG and GWAG for curved walls

Current architectural designs often contain not only straight walls, but also curved walls in order to create more expressive, dynamic and human-friendly spaces. Typically, curved lines in CAD designs are represented as circular arcs (defined by their circle center, their radius and their angular interval) or as polylines that approximate the curved lines. The wall detection algorithms handle the second case successfully, as the final representation is a set of straight segments. Here we describe the changes in the wall detection algorithm to process walls defined using pairs of circular arcs.

Table 6.2 summarizes the criteria to form wall-prone pairs of straight segments, and shows how they are adapted to circular arcs.

Using these criteria, the wall detection algorithm has been modified to work not only with straight segments, but also with circular arcs, using angular values to split the arcs as necessary to create the corresponding wall pairs. The remaining parts of the process are analogous to the ones using straight segments. Figure 6.10 shows all the possible layouts for the processing of circular arcs.

6.2 Opening detection

The second step in the global room detection, as summarized at the beginning of this chapter, is the opening detection. In this stage, the openings are found in the floor plan drawing. For each found opening, its surrounding walls are then

Straight segments	Circular arcs
The segments must be parallel	The segments must share the same circle center
The distance between segments must be less than a threshold ε	The difference between the circle radii must be less than a threshold ε
The projection of one segment onto the line that contains the other one overlaps with it	The angular intervals that define the segments overlap

Table 6.2: Comparison of the criteria to form wall-prone pairs of straight and circular arcs

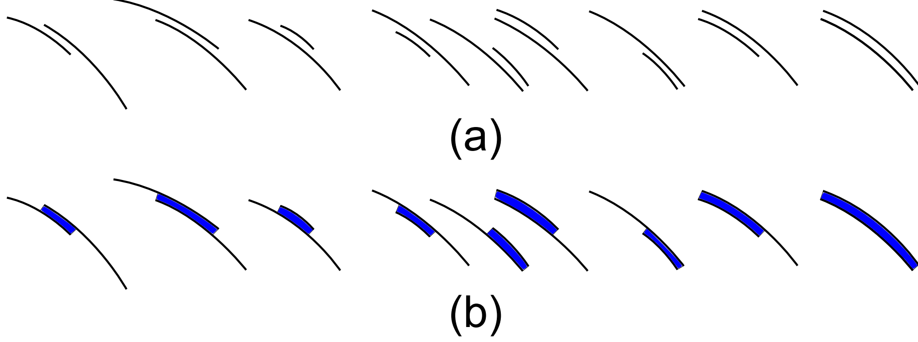


Figure 6.10: Detection of curved walls using WAG: (a) pairs of circular arcs; (b) detected walls

searched, and edges representing openings are added to the initial topology graph.

6.2.1 Opening detection from inserts

As we stated in Section 3.2 about the floor plan structure, first the user has to select the opening layer/s and blocks. From this information, each block instance (called *insert* according to the AutoCAD terminology) is analyzed separately. The goal is to obtain, for each insert, which edges from the topology graph enclose the opening represented by the insert.

In a preliminary step, the bounding box of every insert is computed. The bounding boxes we are dealing with are represented by nine points (four corners plus the center of the box and the middle point of each box side, see Figure 6.11). The center point will be notated as c, and the eight surrounding points will be notated with the cardinal directions: $sw, w, nw, s, n, se, e, ne$.

As each block is defined by a set of primitives using its own coordinate system, the corners of a block's bounding box can be obtained by analyzing the

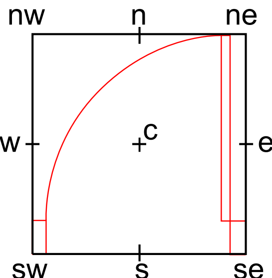
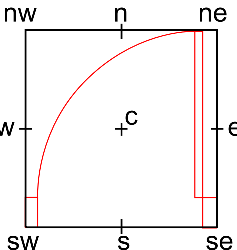


Figure 6.11: Nine-point bounding box of a door

limits of each block primitive, considering the minimum and maximum values for the x and y coordinates. In practice this task is carried out using a Java library called Kabeja[kab], that allows to read and parse DXF files. This library includes a function to compute a classic, 4-point bounding box, thus it is not necessary to implement it. The rest of the points in the bounding box are calculated as the average of the corresponding box sides, according to Figure 6.11.

Once the bounding boxes have been obtained for all the blocks, we can compute the *oriented* bounding box for each insert by applying to the block bounding box the same translation, rotation and scale transformations that have been applied to the block to get the insert. In the example in Figure 6.12, the image on the left represents a block with its bounding box; the right image is the block and its bounding box obtained after applying a scale of $x = -0.7, y = 0.7$, a rotation of 30 degrees and a translation in the x-axis both to the original block and its bounding box.



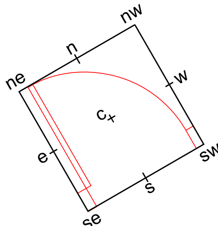


Figure 6.12: Left: Block definition and its bounding box; Right: result of applying scale, rotation and translation to the block and to the bounding box

Given an oriented bounding box (from now on, OBB) and a topology graph with the edges resulting from the wall detection algorithm, the detection of an opening consists of finding, in the topology graph, the closest vertex to the point s of the OBB at its west side, and the closest vertex to the point s of the OBB to its east side. These two vertices are considered the *anchorage* vertices of that opening.

The reason why each vertex is searched in different sides of the OBB, instead to find the two closest vertices, is to avoid incorrect anchorages if the opening is surrounded by short walls and columns, as in the situation shown in Figure 6.13.a. This scenario represents a situation where the wall at the west side of the opening is very short.

Figure 6.13.b shows the topology graph made up by three edges which represent the detected walls and the point s of the OBB of the opening. If the anchorage points of the OBB are computed as the closest points to s, the incorrect result is shown in Figure 6.13.c. On the other side, Figure 6.13.d shows the correct anchorage, using the explained algorithm.

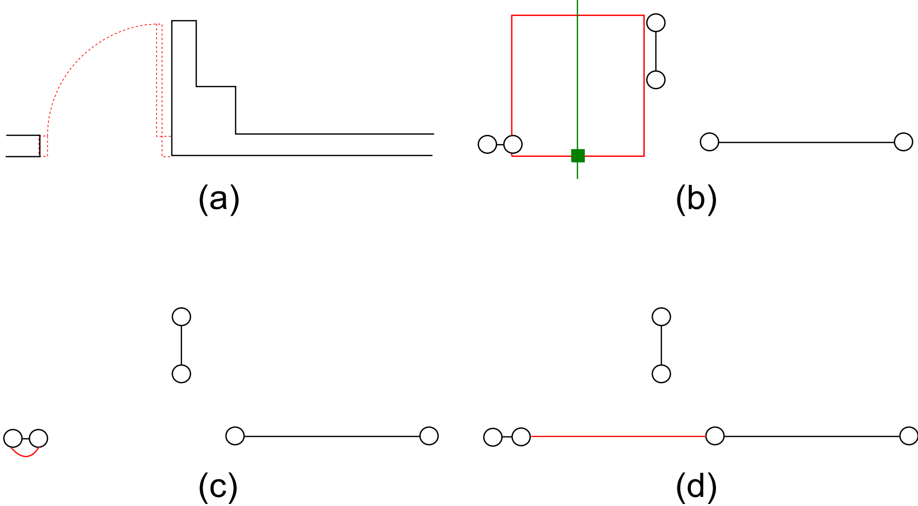


Figure 6.13: Opening detection algorithm: (a) A scenario with a door with two surrounding walls; (b) topology graph and OBB; (c) result of the anchorage searching for the closest vertices; (d) result of the anchorage searching for a vertex at each side of the OBB.

The main advantage of using this distance-based algorithm is that the layout of the primitives of the opening does not affect the result. Suppose a scenario where the opening's insert exceeds the wall limits. This produces that part of the graph lies inside the insert's OBB. With this algorithm, this is not a problem since the distance criterion ensures the correctness (Figure 6.14).

The main difficulty of this algorithm is how to determine which are the east-west and the north-south directions of the oriented bounding box. For windows, the bounding box is clearly rectangular-shaped, therefore the east-west direction is chosen as the longest one (by convention). The problem arises with door blocks, which are closely square-shaped due to the primitives that represent the opened door and its trajectory arc. In this case, it is tricky to distinguish which are the east-west and the north-south directions. In order to solve this, we consider two alternatives:

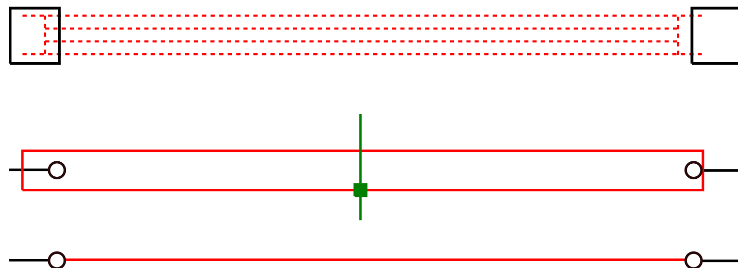


Figure 6.14: Anchorage correctness using a distance criterion: (a) window exceeding the wall limits; (b) OBB and topology graph; (c) correct edge added to the topology graph.

- Remove automatically those primitives in order to make the block rectangular-shaped. Nonetheless, this solution was discarded since it depends on the door block geometry and therefore it has the usual problems of the automatic detection algorithms.

- Study a set of test cases in order to analyze the shape of the door blocks with those primitives and decide how to determine the east-west direction according to the ratio between the length and the height of the door.

Regarding the second approach, we have studied a wide range of door block drawings from all the available floor plans and reached the following conclusions:

- The primitives representing the door and its trajectory arc make door inserts to appear drawn taller than it is wide, and consequently the east-west direction is incorrectly determined as the north-south direction. If we omit these primitives, the door block insert is actually wider than it is tall. For instance, in the door insert represented in Figure 6.15, the real east-west direction is represented by the green line. On the other hand, the primitives make the shape taller than it is wide. According to this criterion, the detected east-west direction is represented by the red line.

- This implies the need of a tolerance threshold to determine the shape of an opening. Therefore, we establish the following criterion: if the bounding box width and height differ more than a percentage, the opening is considered rectangular and the east-west direction is the longest. Otherwise, the opening is considered wider than it is tall. In other words, this benefits that the closely square-shaped openings (according to a percentage threshold) are considered wider than it is tall although they are not. The used percentage in practice, as a result of analyzing a door set, is 80%.

Algorithm 6.2 shows a pseudo-code that, from a topology graph and an opening insert, detects and adds an edge that *anchors* the opening to the topology graph. First, auxiliary function `FindBlock` searches for the block which defines

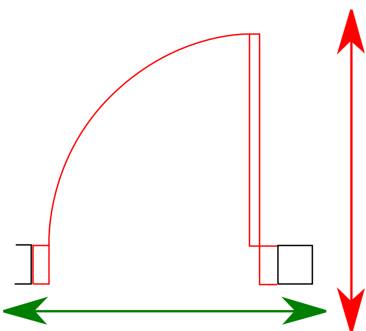


Figure 6.15: Longer direction of a door. The green arrow shows the longer direction if the arc is discarded. The red arrow shows the longer direction including the arc, which supposes the incorrect detection of the east-west direction.

the insert in the set of defined blocks (line 2) and its precalculated bounding box is obtained (line 3).

The *oriented* bounding box (OBB) is computed from the corners of the block bounding box. The OBB has nine points and is computed as shown in the function `OrientedBoundingBox` (Algorithm 6.3). The anchorage points are searched to both sides of the OBB, as explained previously. The function `OrientedBoundingBox` receives as parameters the block's bounding box, the insert data and a threshold percentage (80% by default).

From the OBB and the topology graph, two anchorage points are calculated using the function `GraphAnchorage` (Algorithm 6.4).

Finally, the graph is completed by adding a new edge which joins both anchorage points (line 6).

Algorithm 6.2: Anchor opening to topology graph

Input: insert: the opening insert; G: the current topology graph; blocks: the selected opening blocks
Output: G': the updated topology graph after adding an edge for insert

```
1 begin
2     block ← FindBlock(insert,blocks)
3     (xmin,ymin,xmax,ymax) ← BoundingBox(block)
4     OBB ← OrientedBoundingBox(xmin,ymin,xmax,ymax,insert,0.8)
5     (P1,P2)←GraphAnchorage(G,OBB)
6     G' ←InsertEdge(G,P1,P2)
7     return G'
8 end
```

In Algorithm 6.3 we show the pseudocode of the function that computes the oriented bounding box of an insert. Its input is the block definition of an insert, the transformation of the insert (translation, rotation and scale) and the threshold to consider an insert as rectangle-shaped. If the width and height of

the block bounding box (line 2) do not differ more than a ratio p (0.8 by default, as explained previously), then the flag square is toggled on. The condition for swapping east-west direction and south-north direction (line 7) is that the bounding box is wider than it is tall, or that it is taller than it is wide but the flag is toggled on. According to this, the nine points of the bounding box are assigned taking the X-axis as the east-west direction (line 8), or the north-south direction (line 11).

Finally, in lines 14 to 16 the block bounding box is transformed in order to obtain the OBB of the insert.

Algorithm 6.3: Compute the OBB of an insert

Input: (x_{min},y_{min},x_{max},y_{max}): The bounding box of the block which defines insert
insert(t_x,t_y,r,s_x,s_y): The transformation parameters of the current insert
p: The threshold to consider a block as a square
Output: OBB(sw,w,nw,s,c,n,se,e,ne): The nine points of the oriented bounding box of the insert

```
1  begin
2      if p ≤ |x_max − x_min| / |y_max − y_min| ≤ 1/p then
3          square ← true
4      else
5          square ← false
6      end
7      if |x_max − x_min| > |y_max − y_min| or square = true then
8          OBB ← Compute the OBB considering (x_min,x_max) as the
           west-east direction and (y_min,y_max) as the south-north direction
9
10     else
11         OBB ← Compute the OBB considering (x_min,x_max) as the
           south-north direction and (y_min,y_max) as the west-east direction
12
13     end
14     foreach point p in OBB(sw,w,nw,s,c,n,se,e,ne) do
15         p ← scale, rotate and translate p according to t_x,t_y,r,s_x,s_y
16     end
17     return OBB(sw,w,nw,s,c,n,se,e,ne)
18 end
```

The function to detect the edge which represents an opening and inserting it into the existing graph (Algorithm 6.4) makes two searches, one at each side of the north-south axis of the oriented bounding box (lines 3 to 13) and another at the other side of the north-south axis (lines 14 to 26). For each search, all the edges are visited; Q_1 holds the closest point to s (the south point of the OBB) in the negative half-plane from the north-south axis (this condition is checked using the line equation, lines 4 and 8). The second search, intended to find Q_2, is similar, considering the positive half-plane (condition checked using the

line equation in lines 15 and 19) and assigning the closest point to Q_2. Finally, points Q_1 and Q_2 make up the anchorage edge, and are returned as the function result.

Algorithm 6.4: Find the anchorage points of an OBB in a topology graph

```
   Input: G: Topology graph
   OBB: Nine-points oriented bounding box
   Output: Q1,Q2: Anchorage points
1  begin
2      (a,b,c) ← Line(OBB.n,OBB.s)
3      minDistance ← ∞
4      foreach edge (P1,P2) in the graph G do
5          if d(s,P1) < minDistance and a ·P1.x + b ·P1.y + c < 0 then
6              Q1 ← P1
7              minDistance ← d(s,P1)
8          end
9          if d(s,P2) < minDistance and a ·P2.x + b ·P2.y + c < 0 then
10             Q1 ← P2
11             minDistance ← d(s,P2)
12         end
13     end
14     minDistance ← ∞
15     foreach edge (P1,P2) in the graph G do
16         if d(s,P1) < minDistance and a ·P1.x + b ·P1.y + c > 0 then
17             Q2 ← P1
18             minDistance ← d(s,P1)
19         end
20         if d(s,P2) < minDistance and a ·P2.x + b ·P2.y + c > 0 then
21             Q2 ← P2
22             minDistance ← d(s,P2)
23         end
24     end
25     return (Q1,Q2)
26 end
```

6.2.2 Opening detection from primitives

The opening detection algorithms introduced above assumed that the openings were defined as block inserts. Furthermore, the blocks have to fulfill some minimal requirements, such as being aligned to the X and Y axes. However, in some test cases, the openings are not represented as blocks but as raw sets of primitives (lines and arcs).

This feature addressed us to consider alternative methods for these test cases. A possible solution, consisting of manually grouping openings into blocks (using a CAD tool) has some disadvantages:

- It is less interesting to adapt the existing floor plans to the conditions we require on them than to work on the algorithms to make them solve as many real cases as possible.
- Grouping primitives into blocks is an abstraction work which requires a detailed analysis to avoid similar inserts with different block names. This requires searching for similar openings, group them into blocks, give a different name to each block and assign to each opening its corresponding translation, rotation and scale values.

In spite of this, we did some attempts to abstract openings into inserts. Due to its complexity, an automatic alternative is necessary. This automatic process consists of grouping those primitives that intersect and consider them as a unique opening. Therefore, we need to implement algorithms to detect if two segments, a segment and an arc, or two arcs intersect. Algorithm 6.5 shows the outline of an algorithm which returns a set of openings from a set of primitives.

Algorithm 6.5: Detect openings from a set of entities

```
   Input: entities: the set of entities from the opening layer
   Output: openings: a set of openings. Each opening is a set of
           intersecting entities
 1 begin
 2     Initially, all the entities are unassigned to any opening
 3     n ← the size of entities
 4     free ← 0
 5     foreach entity i =1 to n do
 6         foreach entity j =i +1 to n do
 7             if Intersection(i,j) then
 8                 if i is not assigned and j is not assigned then
 9                     Create a new opening openings[free]
10                     Assign i and j to openings[free]
11                     free ← free + 1
12                 else if i is not assigned and j is assigned then
13                     Assign i to the opening of j
14                 else if i is assigned and j is not assigned then
15                     Assign j to the opening of i
16                 else
17                     Merge the openings of i and j
18                 end
19             end
20         end
21     end
22     return openings
23 end
```

An integer variable free is initialized to zero (line 4). This variable keeps how many sets of intersecting primitives have been initialized. For each pair of

primitives (i,j), the algorithm analyses if they intersect (function `Intersection`, line 7). If the primitives intersect, three possible cases are studied: (1) neither i nor j have been already assigned to an opening, (2) one of them has been yet assigned, or (3) both of them have been assigned. In the first case (lines 8 to 11), a new opening is initialized with i and j, and free is incremented. In the second case (lines 12 to 15), the unassigned primitive is assigned to the opening which the other primitive has been assigned. Finally, if both primitives are already assigned (lines 16 and 17), both openings are merged.

Apart from the logic of the algorithm about how the openings are detected incrementally, the most important part of the algorithm is the implementation of the function `Intersection`, which determines whether two primitives intersect. A more detailed explanation about all the computational geometry algorithms is given in Appendix C.

Once the openings have been detected, one more step would be necessary: compute the bounding boxes. A bounding box can be computed for each detected opening, but the final orientation cannot be determined. The absence of the OBB makes difficult to detect the edge that has to be inserted into the topology graph. There are some possible solutions:

- Try to determine the opening orientation from the opening primitives and the orientation of the closest walls. This could be done by inspecting the closest wall edges and studying their angles. Once the opening orientation has been estimated, the anchorage algorithm explained in Section 6.2.1 could be applied. The main disadvantage of this solution is the risk of an incorrect estimation of the angle, which would involve the detection of an incorrect graph edge.

- Assume that the opening orientation is aligned to the X and Y axes and apply the algorithm from Section 6.2.1 without information about the correct angle. In many cases the result could be correct in spite of the lack of initial information.

This problem is left as one of the open problems for the future work, together with the analysis of other special features and abnormalities that can be found in the study cases.

6.3 Clustering

The last step in the topology graph construction is to search for the joint points among walls. In order to achieve this, we have researched over a number of approaches. The features of this problem are the following:

- The problem starts from a set of edges representing the middle segment of each wall and a set of segments representing the openings.

- In a standard (ideal) floor plan, the edges would be chained, i.e., all the vertices from opening edges have degree two, and the vertices from wall

edges have degree 1 or 2, depending on whether they are anchored to a opening or not.

- If we only consider the 1-degree vertices from the wall edges, the search for joint points can be reduced to look for areas with higher density of vertices of degree one, maximizing the distance between different concentration areas.

According to this definition, we are facing a point clustering problem, and the solution is designed using this approach. The point clustering problem has been widely dealt in the bibliography.

The clustering algorithms have experimented an important evolution. In order to simplify the dissertation, we introduce in this chapter the two last algorithm: comparison of the endpoints of the connected components from the topology graph and the line growing algorithm. Further information about the evolution, advantages and disadvantages of the rest of the clustering algorithms can be found in Appendix B.

The last variation consists of considering only the end points from the polylines. This has two advantages: (1) the number of comparisons between points is lower and (2) inner point of polylines are not considered, so they keep their orthogonality. The pseudocode is shown in Algorithm 6.6.

Algorithm 6.6: Clustering using the end points of sequences

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
 1 begin
 2     Initialize a vertex cluster vertexClusters[i] for each vertex in V
 3     Initialize a sequence cluster sequenceClusters[i] for each connected
       component in E
 4     for i = 0; i < sequenceClusters.size; i + + do
 5         for j = i + 1; j < sequenceClusters.size; j + + do
 6             for k = 0; k < sequenceClusters[i].size; k + + do
 7                 for l = 0; l < sequenceClusters[j].size; l + + do
 8                     (P_min,Q_min) ← Closest points among the four pairs
                       made up taking one point from each current sequence
 9                     min ← d(P_min,Q_min)
10                 end
11             end
12             if min < ε then
13                 sequenceClusters[i] ← sequenceClusters[i] ∪
                   sequenceClusters[j]
14                 Remove sequenceClusters[j] from sequenceClusters
15                 merged ← true
16                 Merge the clusters that contain P_min and Q_min
17             end
18         end
19     end
20     foreach vertexClusters[i] in vertexClusters do
21         centroids[i] ← SmartCentroid(vertexClusters[i])
22     end
23     V' is built from V by replacing every vertex by the smart centroid of
       its vertex cluster
24     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its vertex cluster
25     return G'=(V',E')
26 end
```

6.3.1 Line growing algorithm

In the last stage of the research we designed an algorithm whose aims were:

1. Simplify the number of cases of the previous approach
2. Generalize the different situations of joint points among walls in an easy way

We have named it as *line growing algorithm*. Its basic idea is to move the endpoints of one cluster along the paths determined by the lines that contain

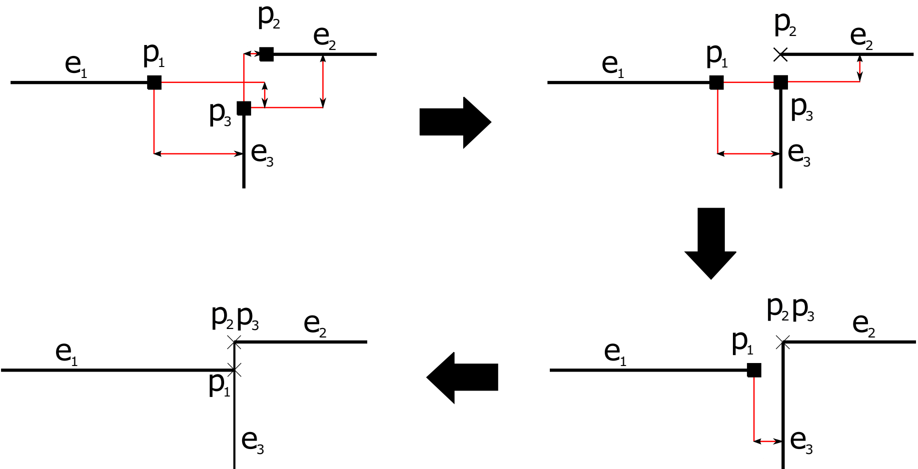


Figure 6.16: Steps of the execution of the growing algorithm on a cluster with three endpoints. Distances are drawn in red

$$\begin{pmatrix} 0 & \infty & 3 \\ \infty & 0 & 1 \\ 1 & 2 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & \infty & 2 \\ \infty & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & \infty & 1 \\ \infty & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & \infty & 0 \\ \infty & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 6.17: Distance matrices used in the example from Figure 6.16

them one step at a time. When an endpoint reaches one of the other lines, it has reached its final position, and the algorithm continues iterating with the rest of the endpoints until all of them reach their final positions (Figure 6.16). This algorithm (Algorithm 6.7) is implemented using a distance matrix which stores the distance from every endpoint of the same cluster to each of the lines that correspond to the other endpoints in the cluster.

The distance matrix is typically not symmetric and the values in the main diagonal are zero.

An endpoint p_i is said to be *locked* if all the values in the matrix row i are zero or ∞; this means that the endpoint has reached its final position. The algorithm loops through the endpoints until all of them are locked, and the process in each iteration is the following: firstly, the minimum distance is selected from the matrix and set as new displacement step, then all the endpoints that are still unlocked are displaced along their corresponding lines according to that step. The matrix is then updated by subtracting the step to each value different from zero or ∞. Figure 6.17 shows the matrices used in the example from Figure 6.16.

The intersection points amid segments may not be unique (see for instance points p_2 and p_3 in Figure 6.16). In these situations, the segments are divided

Algorithm 6.7: Line growing algorithm

```
   Input: P1,P2,...,Pn: endpoints of a cluster;
   L1,L2,...,Ln: lines containing Pi
   Output: P1,P2,...,Pn: endpoints after the line growing
 1 begin
 2     Initialize a matrix {aij}n×n such that aij is the distance from Pi to Lj
       along Li (∞ if Li and Lj are parallel)
 3     while there exists aij such that aij ≠ 0 and aij ≠ ∞ do
 4         m ← the minimum aij ≠ 0 from the matrix
 5         foreach point Pi do
 6             if row i contains values different from 0 and ∞ then
 7                 Move Pi m units towards Lj along Li
 8             end
 9         end
10         foreach aij in the matrix such that aij ≠ 0 and aij ≠ ∞ do
11             aij ← aij- m
12         end
13     end
14     return P1,P2,...,Pn
15 end
```

using the intersection points, and the resulting segments are added to the topology representation.

The algorithm solves correctly L-shaped, X-shaped and T-shaped wall intersections, and can also handle more complex cases, like lines that do not converge to a single point or lines that are not perpendicular, as shown in Figure 6.18.

The line growing algorithm is well-conditioned and robust in most of the cases. However, some clusters where segments are almost parallel result in unstable distance matrices. Thus, it is required to set an angle threshold in the parallelism test for lines.

Figure 6.18 shows two issues of the line growing algorithm. The case shown in (b) is the result of applying the algorithm to the scenario shown in (a). A loop appears, which makes necessary a post-processing stage to remove loops. The case (d) arises when some of the lines are oblique each other, as in (c). Then, the accuracy may avoid the growing lines collapse in the same point. The use of a unique threshold to collapse points for all the situations is an open problem.

6.4 Results

The algorithms have been tested in a computer with a 2.4 GHz Intel® Core™2 Quad processor with 4 GB of RAM using real floor plans of buildings of our university.

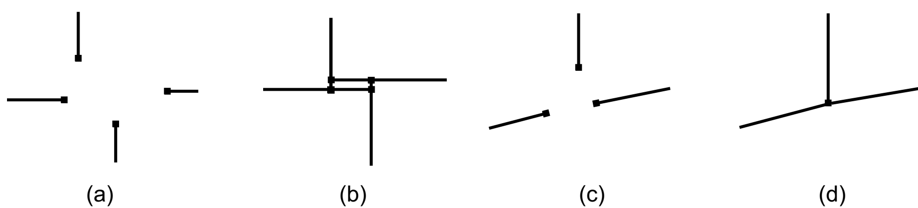


Figure 6.18: Line growing algorithm applied to generic cases: (a) lines that do not converge to a single point; (b) result of the application of the algorithm; (c) lines that are not perpendicular; (d) result of the application of the algorithm

6.4.1 Wall detection

Figure 6.19 shows three of the floor plans used and the result of applying the wall detection algorithm to them; the detected walls are drawn in blue. Although the algorithm has been only applied to the representation of walls, other layers are shown with different colors to give a better understanding of the floor plans.

Table 6.3 shows some numerical results obtained in our tests using different threshold values (the results appear in the same order the results are shown in Figure 6.19). For each test, the table shows how many straight segments and circular arcs were used as input for the algorithm, the value of the threshold ε used, the amount of wall and wall-prone edges in the initial generalized WAG that link respectively segments and arcs, the final number of wall edges at the end of the algorithm execution, the percentage of segments and arcs that do not belong to any wall in the result, the time spent to build the initial generalized WAG and the time the algorithm took to detect the walls. As can be seen, 1 millisecond is the lower limit for the execution of the wall detection algorithm. The final number of wall edges is equal to the sum of the initial number of wall edges and the number of wall-prone edges; this was the expected result, given the way the algorithm works.

TABLE LEGEND
A1-A2. Number of segments/arcs in the floor plan respectively
ε. Threshold used to form segment pairs
B1-B2. Number of initial wall-prone edges linking segments/arcs respectively
C1-C2. Number of initial wall edges linking segments/arcs respectively
D1-D2. Number of final wall edges linking segments/arcs respectively
E1-E2. Segments/arcs that do not form walls respectively (%)
F. WAG building time (milliseconds)
G. WAG processing time (milliseconds)

Plan	A1	A2	ε	B1	B2	C1	C2	D1	D2	E1	E2	F	G
1	643	0	0.4	193	0	56	0	249	0	36.10	0	118.50	1.09
			0.5	199	0	58	0	257	0	35.00	0	119.17	1.10
			0.6	203	0	60	0	263	0	33.90	0	122.03	1.16
			0.7	236	0	62	0	298	0	26.59	0	136.92	1.35
2	1148	14	0.4	482	9	159	0	641	9	32.50	11.10	612.21	3.99
			0.5	549	9	206	0	755	9	25.10	11.10	755.40	5.08
			0.6	628	9	244	0	872	9	16.00	11.10	937.65	6.38
			0.7	692	9	287	0	979	9	10.10	11.10	1161.43	8.37
3	1678	79	0.4	442	20	93	0	535	20	47.73	54.43	696.45	4.45
			0.5	455	20	104	0	559	20	45.76	54.43	785.69	4.91
			0.6	637	33	176	0	813	33	26.04	26.58	922.89	5.21
			0.7	667	33	190	0	857	33	22.76	26.58	957.27	5.56

Table 6.3: Numerical data from the wall detection tests with real floor plans

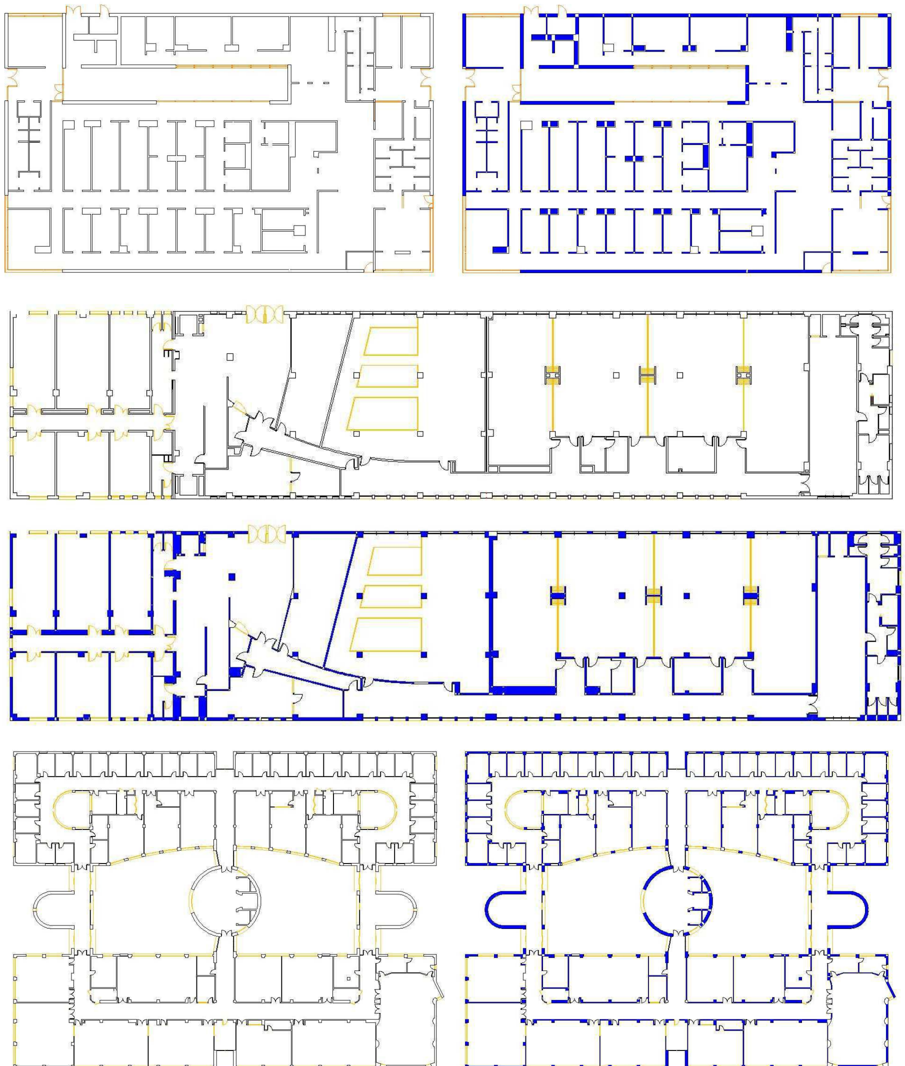


Figure 6.19: Real floor plans used to test our algorithms, together with the results of applying the wall detection algorithm. Detected walls are drawn in blue. The layers containing windows and doors are shown, but have not been processed

The time spent in the construction of the initial WAG is in general less than a second for floor plans with no more than 1700 segments, although the complexity of the drawing has a significant influence on this task.

It is interesting to remark that using different values for the threshold ε produces significant changes in the results: the bigger the threshold, the more complex is the structure of the generalized WAG, because the number of relevant pairs of segments increases, and the time spent to build and process the graph is therefore higher. On the other hand, the number of segments and arcs that are not considered as part of a wall pair at the end of the wall detection algorithm decreases as the value of the threshold increases. It could be interesting to study more carefully this issue in order to find a way to compute dynamically the optimum value of the threshold for each floor plan.

6.4.2 Topology graph construction

Figure 6.20 shows the representation of the topology of a portion of a real floor plan as obtained with the algorithms described in this paper, while Figure 6.21 shows the topology representation on top of the original floor plan.

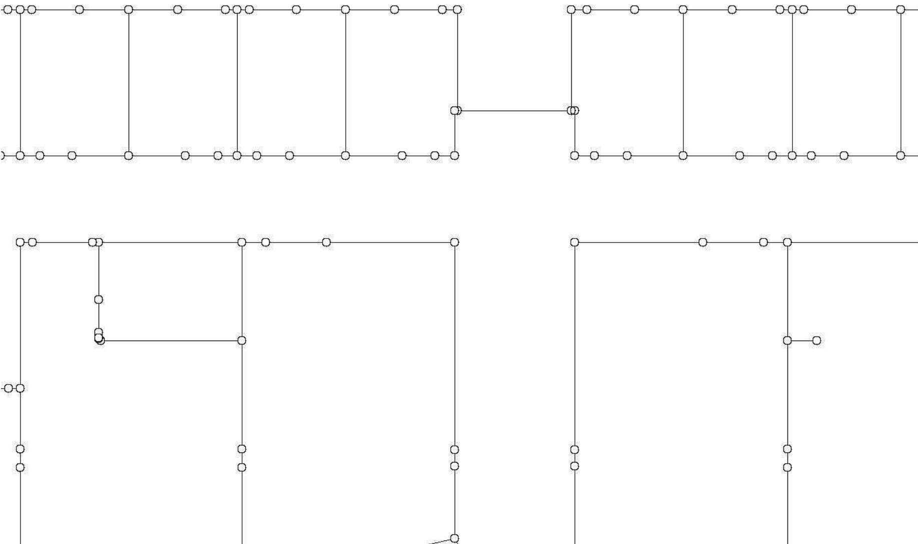


Figure 6.20: Topology representation from a portion of a CAD vector floor plan

Table 6.4 summarizes the results obtained in some tests with real floor plans. Test cases *A3* and *C5* correspond to buildings from our university campus, test case *House* corresponds to a common single-family dwelling. Other test cases are *Basic lift*, *Planta tipo*, *Vilches*, *Bayenga* and *Heating*.

The numbers represent information about the floor plans, as well as the results obtained with our software. As can be seen, simple plans are processed

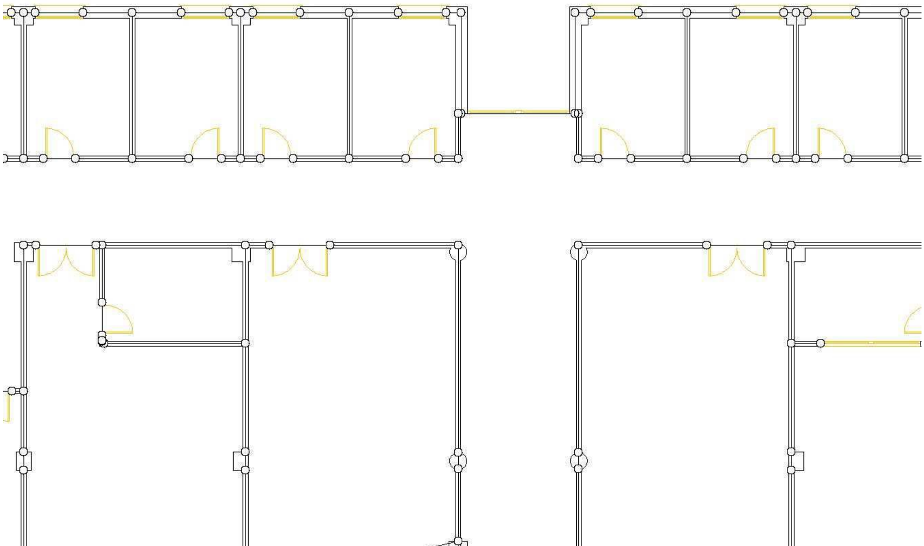


Figure 6.21: Topology representation on top of the original CAD vector floor plan

very accurately with very little user intervention, while the accuracy can be quite acceptable with more complex plans. The alternative values for cases *A3*, *C5* and *Planta tipo* show how it is possible to significantly improve the accuracy with a little user intervention to manually edit the preliminary results (from 73,81% to 95.24% in case *A3*, from 89,55% to 97,01% in case *C5*, and from 67% to 89% in case *Planta tipo*). The time necessary to edit manually the results is just a few minutes.

Figures 6.22, 6.23, 6.24, 6.25, 6.26, 6.27 and 6.28 show respectively a snapshot of the original floor plan and the resulting CityGML model after detecting the topology graph. Although the CityGML is a topic covered in Chapter 10, these figures show the result of the topology graph detection.

In Part II we have introduced a set of methods to get topology graphs of stories from architectural floor plans. In order to this, we have implemented these algorithms: wall detection, opening detection and clustering. The next part (Part III) will deal with the topological analysis of the obtained results: the topology graph will be analyzed in order to get closed spaces (generally corresponding to rooms/corridors) and the topological correctness of the methods in Part II.

	C1	C2	C3	C4	C5	C6	C7		C8	C9
							C7.1	C7.2		
House	92	15	33	4	0	4	0	3	0,00%	100,00%
A3	1769	196	594	72	10	84	0	0	13,89%	73,81%
				80	0	84	30	10	0,00%	95,24%
C5	1150	156	359	62	2	67	12	0	3,23%	89,55%
				66	1	67	12	4	1,52%	97,01%
Basic lift	162	16	62	7	0	9	0	1	0%	78%
Planta tipo	110	15	42	8	2	9	0	3	25%	67%
	110	15	42	9	1	9	0	5	11%	89%
Vilches	146	20	56	10	0	10	0	2	0%	100%
Bayenga	100	17	38	6	0	6	0	3	0%	100%
Heating	144	16	57	7	0	10	0	6	0%	70%

Table 6.4: Results of tests on real buildings. The columns represent: (C1) number of wall lines and arcs in the drawing (C2) number of openings (C3) number of detected walls (C4) number of detected rooms (C5) number of incorrectly detected rooms (C6) number of real rooms (C7) number of manual editions (C7.1) number of manually detected walls (C7.2) number of manually added edges (C8) (incorrect rooms/detected rooms) ratio (C9) success, computed as (detected rooms - incorrect rooms) / real rooms

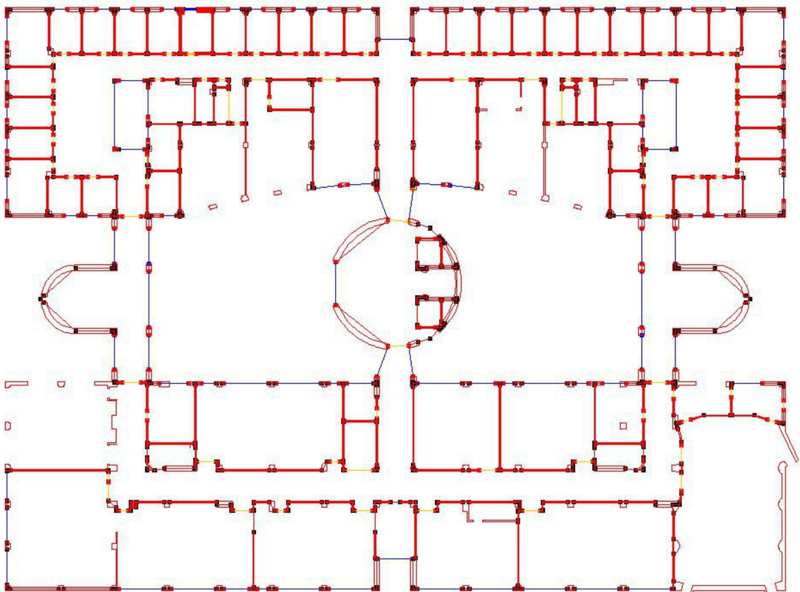


Figure 6.22: Result of processing the floor plan of the A3 building, with no additional user editing. Walls are represented with red lines, and openings are represented with blue lines

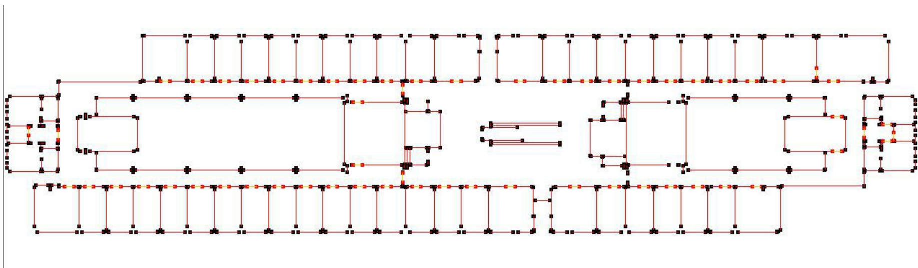
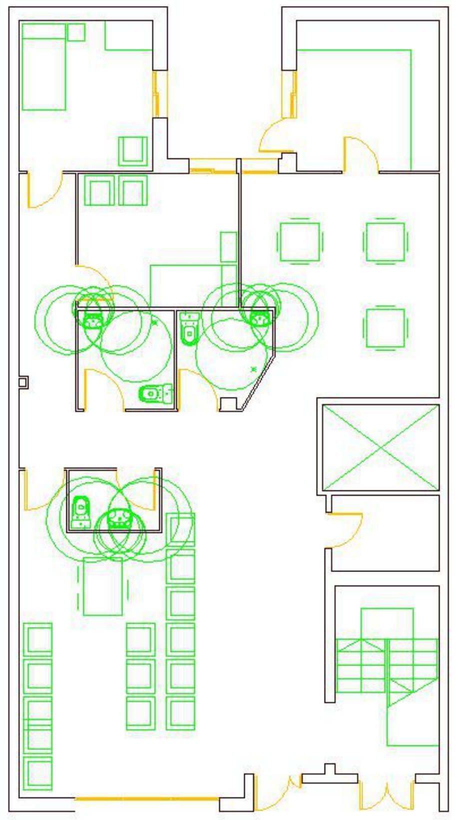


Figure 6.23: Result of processing the floor plan of the C5 building, with no additional user editing. Walls are represented with red lines, and openings are represented with blue lines



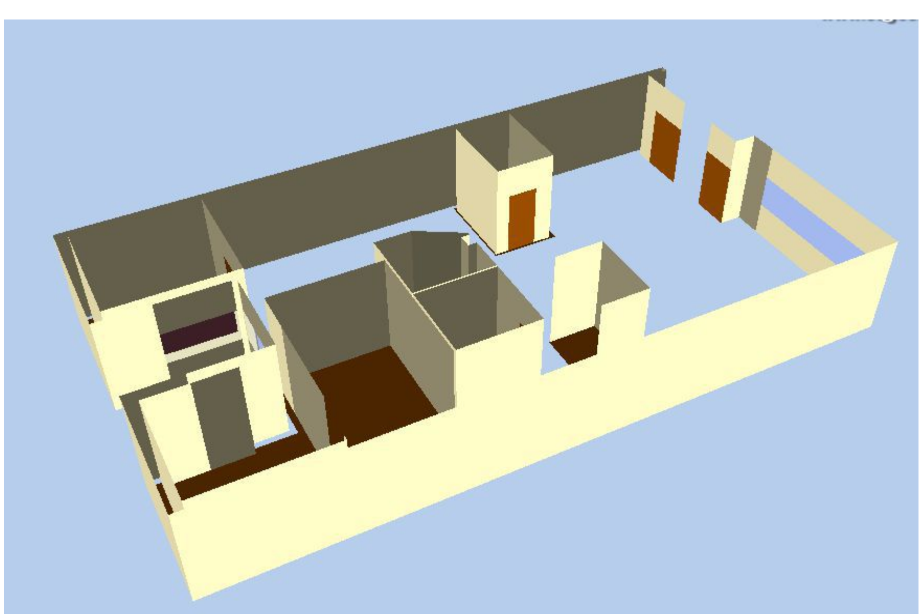
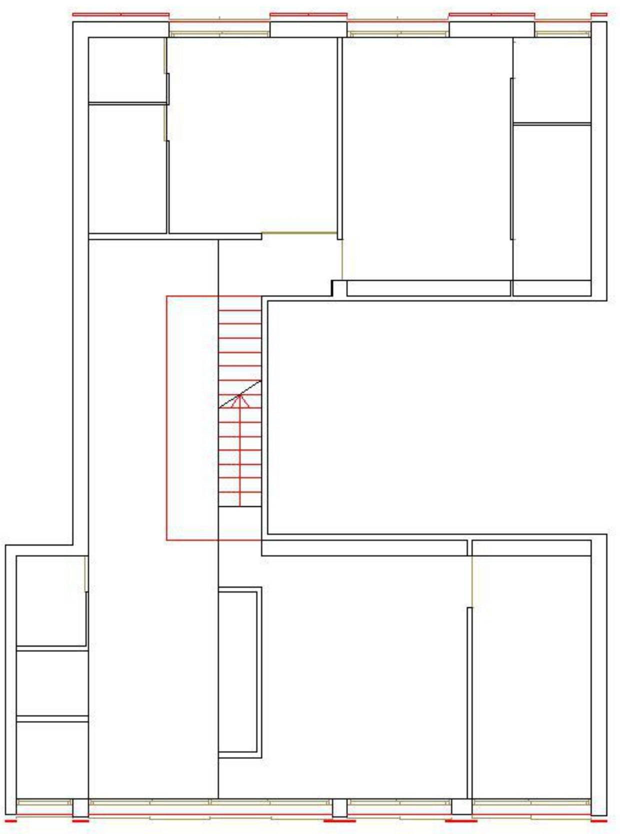


Figure 6.24: Floor plan and CityGML model of *Basic lift*



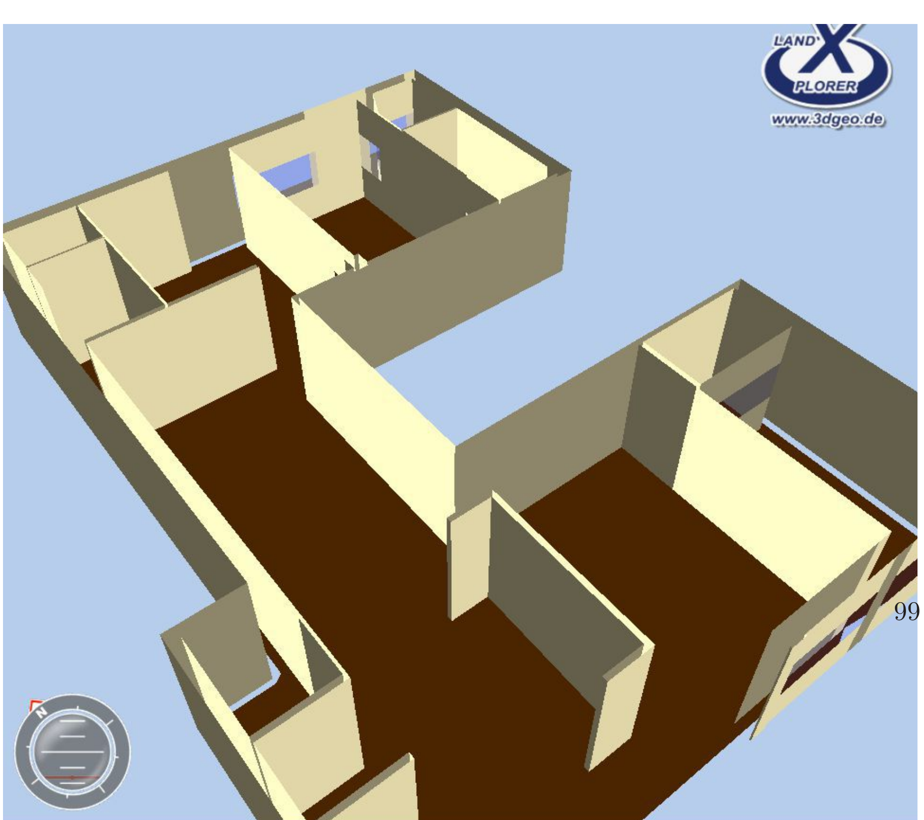


Figure 6.25: Floor plan and CityGML model of *Planta tipo*

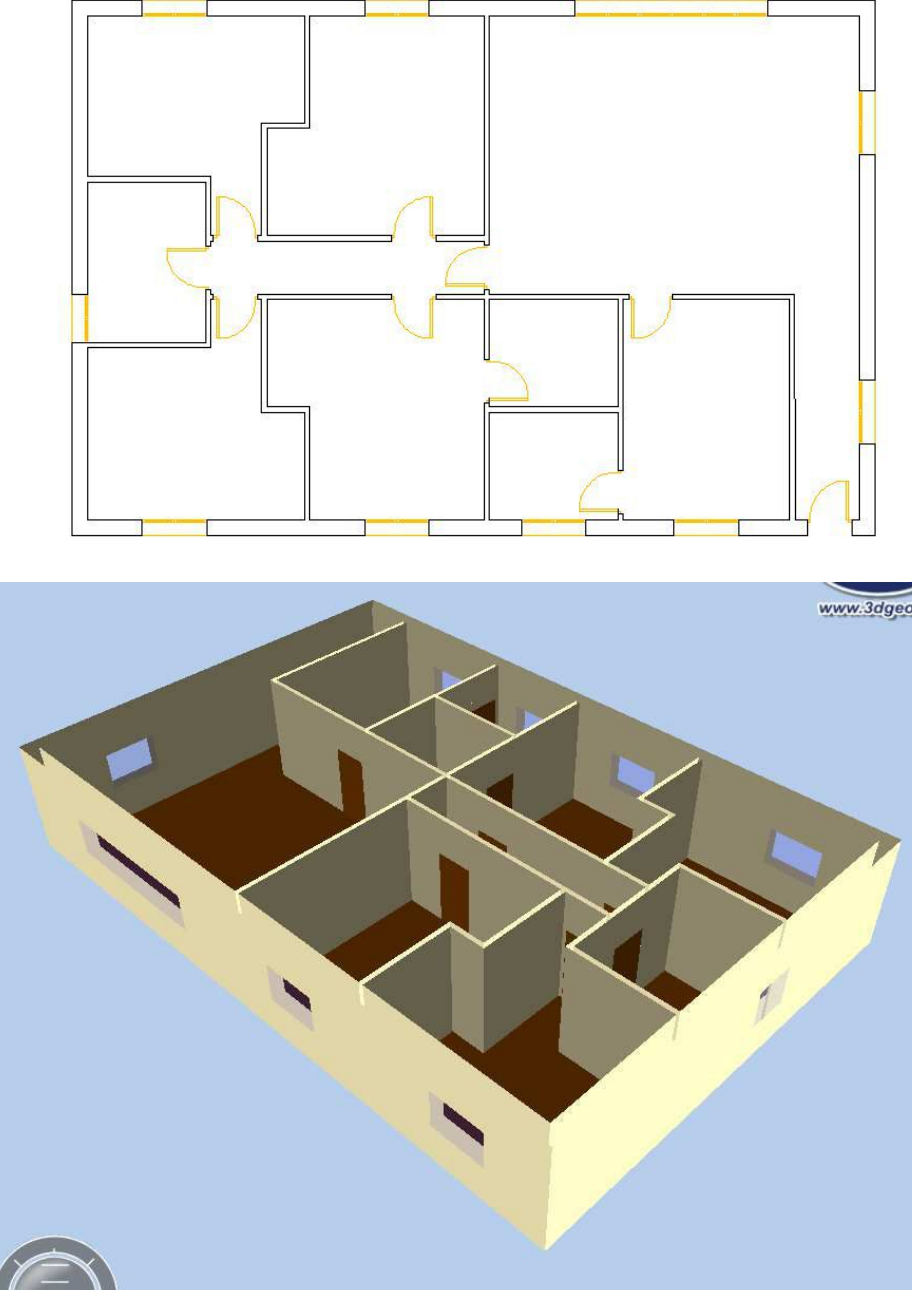


Figure 6.26: Floor plan and CityGML model of *Vilches*

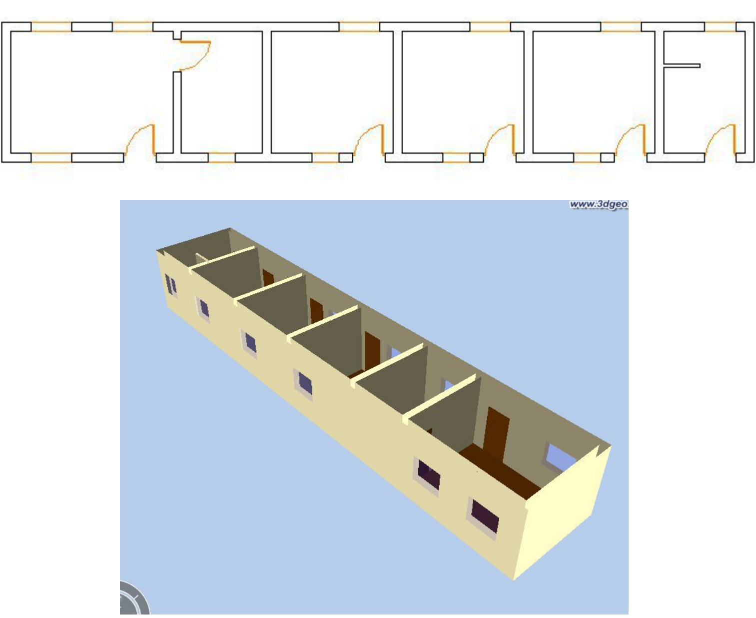


Figure 6.27: Floor plan and CityGML model of *Bayenga*

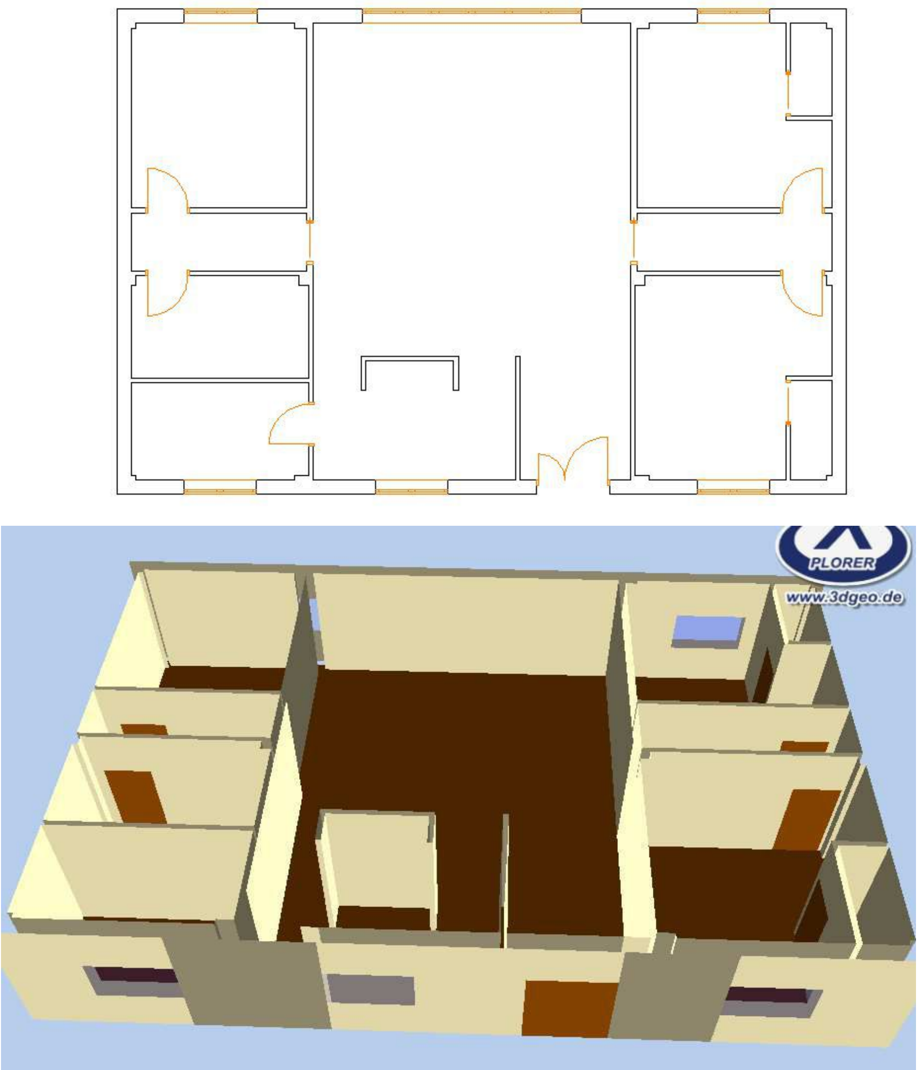


Figure 6.28: Floor plan and CityGML model of *Heating*

Part III

Representation of building information

This part of the document deals with higher level details related to the management of the information obtained in part II, such as the creation of topologically correct models in 2D and 3D. This issue is introduced according to two points of view: on one side, we will introduce a number of algorithms which create the floor topology from the information obtained in part II; on the other side, we will discuss some aspects about the topological correctness of the model, emphasizing the importance of this property.

The basic structure that will be developed in this part of the document consists of a topology graph that is composed by: (1) a set of edges, each one representing either a wall or an opening (door or window) and (2) a set of vertices, each one representing a joint among two walls, or among one wall and one opening.

This part is structured as follows: Chapter 7 proposes a unified model which accomplishes with most of the requirements found in the literature (Section 2.5) and complements the information obtained by the algorithms in Part II of the document. Chapter 8 introduces the algorithms needed to obtain topological information from the topology graph and link the topological and the geometric information. Chapter 9 deals with the triple extrusion, an algorithm which obtains a 3D geometric representation of a building from its front, side and top views. Finally, Chapter 10 describes the generation of output models from the architecture introduced in this part, such as CityGML or 3D models.

Chapter 7

A three-level framework to represent topologically correct models

Different application areas of the building indoor information have led to a huge variety of representation models. In this chapter, we propose a data model to represent building indoors with the following characteristics:

1. The model covers different levels: from the geometry of architectural drawings to the semantic and topological information about the structural distribution and the connectivity among physical spaces (e.g. adjacency between rooms, between a room and an exterior area of the building, etc.).
2. The model contains topologically correct 2D and 3D information.
3. The model has different LoD's. It includes a rough 3D representation of a building inferred by an algorithm called *triple extrusion*.
4. All the parts of the model should be related, so that information from different levels can be mapped efficiently.
5. Other information models should be easily derived from our model.

7.1 Comparative analysis and discussion

In Section 2.5, a number of papers from the literature were reviewed and classified according to the dimension used to represent data. Besides, some considerations about geometry, topology and semantics have been analyzed from each paper.

In this section we will propose a model which intends to serve as a link between low-level geometry, topology and semantics. Later, we will compare

the reviewed works and discuss about the applicability of the cited models to different areas.

Most of the works on Building Information Modeling, 3D Cadastre, model validation, etc., consider three levels of building information, i.e., geometry, topology and semantics. The topological and semantic features rely on the geometric representation. Regarding 2D geometry, most of the models are based on basic primitives (points, lines and polygons), while for 3D geometry we often find B-rep models or the use of existing tools from 3D GIS, BIM and 3D CAD. The model we introduce uses (1) CAD primitives in 2D and (2) B-rep and GML models in 3D.

The topology involves the management of the relationships among the geometric entities, e.g. adjacency among polyhedron faces or points as boundaries of lines. A key issue is the maintenance of the topological consistency against changes on the geometry. In this sense, Ledoux and Meijers [LM09] deal with the concept of topological consistency, stating three conditions for a set M of 2D objects to be consistent:

- Every line in M is formed by two points in M
- The intersection of two lines in M is either the empty set or a point in M
- The intersection of the interior of a polygon and any other primitive in M is empty

For 3D topological spaces, another additional condition must be fulfilled to ensure consistency: the intersection of the interior of a polyhedron and any other primitive is empty. We will justify that the different models we propose are topologically correct.

Isikdag et al. [IAUW07] enumerate eleven principles that BIM models must follow, among which we highlight object orientation, interoperability, comprehensiveness, and richness in semantics. Regarding the semantics, some concepts of interest for building models are the identification of walls, doors, windows and rooms. The topological structure of semantic elements can be deduced and validated from the topological relationships among geometric elements, according to the following definitions of connectivity and adjacency:

1. Two spaces in a building are considered as topologically connected if there exists a door or a window between them. Thus, models with information about openings have topological connectivity. Connectivity is *explicit* if it is represented in the model, and *implicit* if it can be deduced by analyzing the model.

2. Two spaces in a building model are topologically adjacent if they share at least one item (e.g. rooms sharing one wall). Adjacency is *explicit* if the model contains information about relationship between spaces, or *implicit* if the model does not contain this information, but it can be deduced by analyzing the geometry.

3. Connectivity and adjacency relationships can be deduced from topological relationships among polyhedrons through their shared faces. Our aim is to keep the space topology *explicit*.

The last row of the table summarizes the main features of the schema proposed in our work. The following sections describe it in depth.

Before presenting a summary of the main features of the reviewed papers (table 7.1), we introduce some previous considerations related with geometry, topological connectivity, topological adjacency and semantics.

7.1.1 Geometry

The majority of the reviewed papers include geometrical information as the basis of their models. However, they do not deal with geometry with the same level of detail. We distinguish among the following items:

1. Works that do not mention anything regarding geometry because they only focus on semantic issues. They appear in table 7.1 with a dash (-).

2. Works mentioning geometric elements without giving details of the underlying representation. They appear as *implicit*.

3. Geometric elements like vertices, edges, faces, regions, discrete cells or volumes appear abbreviated respectively as V, E, F, R, DC, VO.

4. Other works use spread models like IFC, CityGML, Geographic Markup Language (GML) or BIM's

7.1.2 Topology

We distinguish among connectivity and adjacency, when applicable.

1. Two spaces in a building model are topologically connected if there exists a door or a window between them. Thus, models with information about openings have topological connectivity. Connectivity is explicit if it appears represented in the model, and implicit if it can be deduced by analyzing the model.

2. Two spaces in a building model are topologically adjacent if they share at least one item (e.g. rooms sharing one wall). Adjacency is explicit if the model contains information about relationship between spaces, or implicit if the model does not contains this information, but it can be deduced analyzing the geometry.

7.1.3 Semantics

For each reviewed work, we specify which semantical items it contains, according to the legend: RO, O, PA, CR, S, T, W, L, C, CO represent respectively rooms, openings, passages, crossroads, stories, tags, walls, lifts, ceilings and corridors.

Table 7.1: Comparison between the overviewed works.

Group	Work	Geometry	Topology		Semantics
			Connectivity	Adjacency	
2D models	Franz et al. [FMW05]	-	Implicit	Explicit	RO, O
	Lamarche and Donikian [LD04]	Implicit	Explicit	-	PA, CR
	Plümer and Gröger [PG96]	V, E	-	Implicit	-
	Stoffel et al. [SLO07]	V, E, R	Explicit	Implicit	RO, O, S
	Li et al. [LCR10]	DC	-	-	T
	Zhi et al. [ZLF03]	V, E, R	Explicit	Implicit	R, O
	Hahn et al. [HBW06]	Implicit	Explicit	Implicit	RO, O, S
	Merrell et al. [MSK10]	Implicit	Explicit	Implicit	typed-RO, O
2.5D models	Slingsby and Raper [SR07]	Implicit	-	Implicit	W, L, O
	Tutenel et al. [TBSdK09]	Implicit	-	-	RO
	Germer and Schwarz [GS09]	Implicit	-	-	RO
	Van Dongen [vD08]	Cubes	-	-	W, C
	Choi et al. [CKHL07]	-	Explicit	Implicit	RO, O, S, W
3D models	Choi and Lee [CL09]	R	Explicit	Explicit	RO, O, CO
	Clemen and Gielsdorf [CF08]	V, E, F, P	Explicit	Explicit	-
	Van Berlo and Laat [vBdL10]	IFC, CityGML	Explicit	Implicit	RO, O, S, W
	Paul and Bradley [PB03]	V, E, F, VO	Explicit	Explicit	W, C
	Billen and Zlatanova [BZ03]	V, E, F, VO	Implicit	Explicit	Buildings
	Hagedorn et al. [HTGD09]	GML	Explicit	Explicit	RO, O, S, W
	Van Treeck and Rank [vTR07]	B-rep	Explicit	Explicit	RO, W
	Borrmann and Rank [BR09]	VO	-	-	Buildings
	Isikdag et al. [IUA08]	Implicit	Implicit	Implicit	O, S, W
	Boguslawski et al. [BG10]	V, E, F	-	Explicit	-
	Yan et al. [YCG10]	BIM	BIM	BIM	BIM
	Xu et al. [XZZ10]	F, VO	Explicit	Explicit	RO, O, S, W, C
	Our proposal	V, E	Explicit	Explicit	RO, O, PA, S, W, C

Legend: V=vertices, VO=volumes, E=edges, F=faces, P = planes, R=regions
DC=Discrete cells, RO=rooms, O=openings, PA=passages, CR=crossroads
S=stories, T=tags, W=walls, C=ceilings, L=lifts, CO=corridors

We introduce (1) a three-level model for 3D buildings representation containing information about CAD drawings and the topological structure of building interiors and (2) algorithms to generate and map information keeping the topological correctness.

Figure 7.1 presents an overview of the model. The geometry module represents the input data of the model and the rough 3D representation. It consists of CAD architectural floor plans which may include information about a variety of aspects such as structure, furnishing, plumbing, electricity, etc., together with metadata such as measurements and annotations. Due to the huge variety of possible models represented in CAD floor plans, we propose a set of constraints on the range of input data, like representing the wall geometry in one layer of the drawing and the door/window geometry as block instances in a different layer [DGF12]. The semantics module is related to the structure of a building interior, i.e., walls and openings, obtained as the result of semiautomatically filtering the information contained in the first module. In this module, the available low-level geometry elements (wall lines and opening blocks) are processed in order to obtain high-level information (walls, wall intersections and openings). The topology module includes a topology graph derived from walls and openings, and its corresponding dual graph represents the subdivision of a building into closed spaces. Finally, the room structure gathers together structural and semantic information (topology graph) and geometric information (wall lines).

Each module contains both 2D and 3D elements. Some elements like wall lines and opening blocks combine geometric and semantic information, and other elements like dual graphs can be considered as 2D/3D topological elements, as long as they include connectivity relationships among 2D or 3D spaces.

7.2 Geometry module

The geometry module contains information about CAD drawings and solids obtained from them using an algorithm called triple extrusion.

The CAD drawings contain entities, whose main types are line segments, polylines, arcs, circles and inserts.

Nonetheless, not all the layers contained in a CAD drawing are relevant to know the building structure. In addition to the building structure, it is common to find information about furniture, bathroom fittings, plumbing, electrics, telecommunications, etc. as well as fictional elements which help to interpret the floor plan (measurements, annotations, etc.). It is necessary to filter all this information. The main role of layers in a CAD drawing is to make possible the separation of elements. Unfortunately, the layer structure typically differs from one CAD file to another. The lack of uniformity in the layer structure is an obstacle for the automatic filtering of information. This leads us to propose some rules for CAD drawings:

- Walls must be drawn using straight line segments, polylines or circle arcs. A piece of wall is represented either as a pair of parallel segments (straight

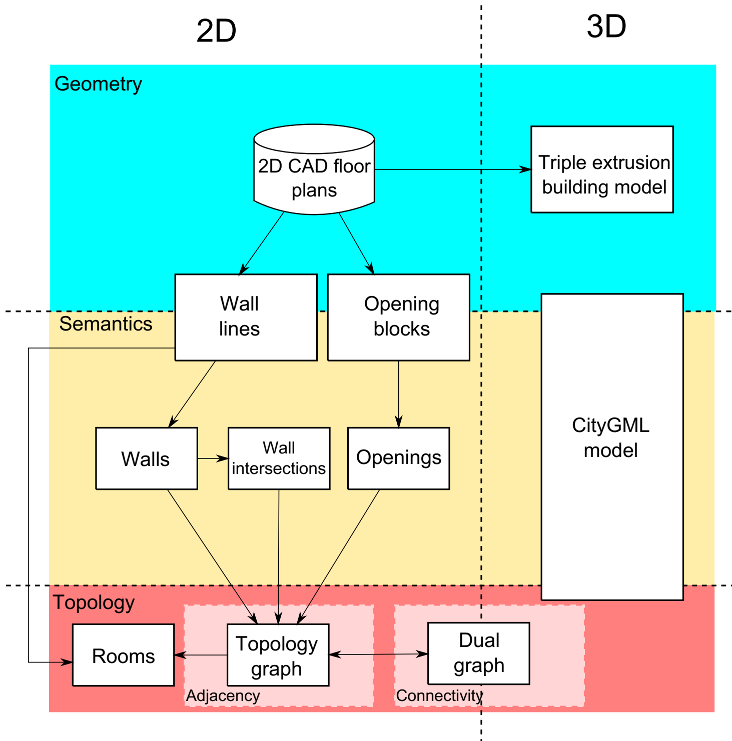


Figure 7.1: Model based in a three-level architecture to represent building information models.

wall) or as a pair of concentric circle arcs (curved wall). There are no constraints about how many layers are used for drawing walls, but every layer containing walls cannot contain segments, polylines or arcs with different semantics. This prevents other entities from being incorrectly used to extract walls.

- All the openings (windows and doors) must be drawn as inserts (instances of named blocks). Inserts representing openings must not contain any information different to the corresponding to the openings themselves. In order to determine which inserts represent openings, the names of opening blocks have to be provided by the user. There are not constraints on the opening layers, although for usability it is recommended to store them in the lowest number of layers possible.
- All the opening definitions must be aligned with the X and Y axes in block local coordinates.

Wall lines and opening blocks are obtained from a 2D floor plan after semi-automatically selecting the corresponding layers and blocks. They contain an

initial piece of semantic information.

On the other hand, the solids obtained from the CAD floor plans using triple extrusion are represented using polyhedral B-Reps.

7.3 Semantics module

The semantics module holds the result of the extraction of the relevant information needed to represent the structure of buildings (walls and openings). If the CAD drawing rules described above are fulfilled by the input floor plans, this extraction can be done semiautomatically, provided that the user selects which layers contain walls and openings, respectively, and which blocks represent openings, as proposed in [DGF12]. Otherwise, manual processing of the input is needed.

Wall lines and openings are preprocessed in the following way:

- The set of wall lines is processed to build a mesh. Points that are close enough are merged using a threshold.
- Openings can have complex designs, but for representation purposes only the bounding boxes are stored. Bounding boxes are computed in two steps: (1) for each opening block, a bounding box is computed in local coordinates; (2) the bounding boxes for the block instances are computed by translating, scaling and rotating the bounding boxes of the original blocks.

The CityGML representation contains merely semantic elements relying on a geometric description in GML. As we stated in the motivation, making explicit the topological relationships is not mandatory in the CityGML model, but we will prove that the CityGML models obtained with our system fulfill the topological correctness requirements.

7.4 Topology module

This module represents the structure and topology of building stories. Its main component is a graph-based data structure where: (1) Nodes represent end points from pieces of walls; they include information about the geometrical position of points (2) Edges represent doors, windows and pieces of walls connecting two end points. Each edge includes information about its type (*door*, *window* or *wall*).

A dual graph can be derived from the topology graph in order to represent rooms and adjacency/connectivity among them. While the topology graph edges represent walls, doors and windows, a dual graph representing closed spaces and their adjacency and connectivity can be derived. Lee and Kwan [LK05] define the adjacency graph $G = (V(G), E(G))$ and the connectivity graph $H = (V(H), E(H))$, which is a subgraph of G. As we label the edges as

door, *window* and *wall*, H will contain only those edges which connect spaces through *door* and *window* edges.

Chapter 8

2D topology. CAD floor plan processing

In this section we deal with the construction of 2D topology graphs representing the structure of building stories and the enrichment of the topology with geometric information from CAD floor plans.

8.1 Construction of the topology graph

The method to construct a graph from filtered CAD drawings consists of three steps: (1) wall medial-axis detection; (2) simplification of openings and (3) closed-area detection. As a result of this process, the topology graph is constructed using intersection points as nodes and walls/openings as edges. These steps were described in Part II of the dissertation. In this part we deal with the topological correctness of the obtained representation and propose additional steps which complete the previous algorithm and ensure the correctness.

In order to avoid topological inconsistencies in the wall detection step, we must ensure that the walls do not intersect. These situations may occur when there are walls whose length is lower than the threshold ε (see Figure 8.1). Therefore, once all the walls have been detected, we process the set to remove the intersecting rectangles as shown in Algorithm 8.1.

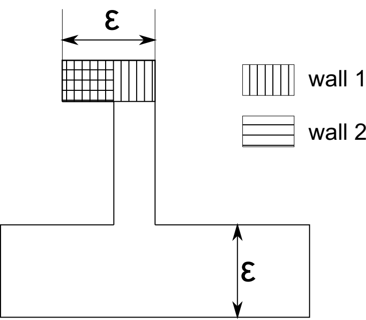


Figure 8.1: Intersecting walls: The threshold ε is higher than the length of wall 2. Therefore, a wrong wall (wall 1) appears and intersects wall 2

Algorithm 8.1: Remove intersecting walls

Input: W: the set of detected walls
Output: W: the set of walls after removing the intersecting walls

1 **begin**
2   Sort the set W of detected walls in ascendant width order
3   **foreach** *wall* w_i *in* W **do**
4   **foreach** *wall* $w_j, j \in [0, i-1]$ **do**
5   **if** w_j *intersects* w_i **then**
6   Mark w_j as removable
7   **end**
8   **end**
9   **end**
10   Remove from W all the walls marked as removable
11   **return** W
12 **end**

In Figure 8.1, wall 1 is wider than wall 2. Therefore, Algorithm 8.1 removes wall 1 and keeps wall 2.

Then, the middle line of each wall is computed and labeled as *wall* in the topology graph. On the other hand, for each opening represented by a bounding box, a segment is computed by means of searching for the closest wall axes to the bounding rectangle and joining their endpoints. The new axes are labeled as *door* or *window*, depending on the block type. Therefore, we have a set of labeled 2D segments representing walls and openings. Some segments are chained, making up groups of connected walls and openings. Once again, the topological consistency must be maintained. Two conditions are sufficient (but not necessary) to guarantee the consistency: (1) avoid intersections between bounding boxes and (2) avoid intersections between a bounding box and an edge of the topology graph (except the two *wall* edges adjacent to the opening represented by that bounding box). In practice, these conditions are only unaccomplished in degenerate cases.

Most areas with high concentration of end points coincide with intersections between walls. Thus, intersection points can be computed by means of searching for point clusters and collapsing the points which belong to the same cluster. This process gives as output the topology graph.

Rooms and corridors typically correspond to closed spaces in the topology representation. Computing a dual graph from the topology graph described above results in the detection of rooms and corridors in a floor plan.

A dual graph [LK05] of a planar graph G is a graph whose nodes are the closed spaces in G, and whose edges connect nodes that represent adjacent closed spaces. Notice that a given graph could have various non-isomorphic dual graphs, since it depends on the concrete embedding of the graph.

In our case, the embedding of the graph is unique, since each node has a 2D point associated. In order to obtain the closed spaces from the graph (i.e. the nodes of the dual graph), the MAFL algorithm [ZLF03] is applied. The MAFL algorithm extracts closed regions from an unconnected graph with nodes representing 2D points and edges representing 2D lines. A simplified version for connected graphs is outlined in Algorithm 8.2.

Algorithm 8.2: Simplified version of the MAFL algorithm [ZLF03]

Input: A connected, undirected graph
Output: A set of closed spaces

```
1  begin
2      Initialize an empty set of closed spaces
3      Assign a direction to each edge of the undirected graph
4      Initialize two flags for each edge as unvisited (one flag for each
       direction)
5      while there are unvisited directed edges do
6          Select an unvisited directed edge and mark it as visited
7          Start a new cycle
8          while the cycle is open do
9              From the set of edges adjacent to the target node of the
               current edge, select the edge with minimum clockwise angle
10             Add the selected edge to the cycle and set it as the current
               edge
11             Mark as visited the forward/reverse direction of the current
               edge, according to the direction in which it is visited
12         end
13         Add the cycle to the set of closed spaces
14     end
15     return the set of closed spaces
16 end
```

The result of the algorithm is a set of polygons representing closed spaces and a polygon representing the outside. All the vertices from polygons representing closed spaces are sorted using a clockwise orientation, while the exterior is oriented counterclockwise, and every edge is shared either by two closed spaces or

by a closed space and the exterior. The exterior polygon can be easily detected since it is the only polygon that does not fulfill the property of the sum of the angles [1] in a polygon, which must be equal to $180(n-2)$ for a polygon with n vertices.

The correctness of the algorithm can be justified as follows: (a) all the directed edges are selected (the outer while loop always ends); (b) all the cycles become closed (the inner while loop always ends); (c) for a given current edge, the next directed edge (i.e. the edge with minimum clockwise angle) has not been visited before, thus the algorithm never blocks itself when selecting the next edge.

Two special situations need to be considered: (a) graphs with dangling edges and (b) graphs with bridges (see examples in Figures 8.2 and 8.3).

A dangling edge is defined as an edge with at least one node of degree zero. On the other hand, a bridge (or cut-edge) is defined as an edge whose removal produces an increment of the number of connected components of a graph. Although it is not likely that these situations happen in topology graphs derived from real buildings, an exhaustive study is necessary in order to guarantee the correctness of the algorithms. We consider two main sources for these problems: (a) existence of walls in open spaces (e.g. buildings in ruins) and (b) columns whose lines are in the wall layer and have been collaterally detected as dangling walls.

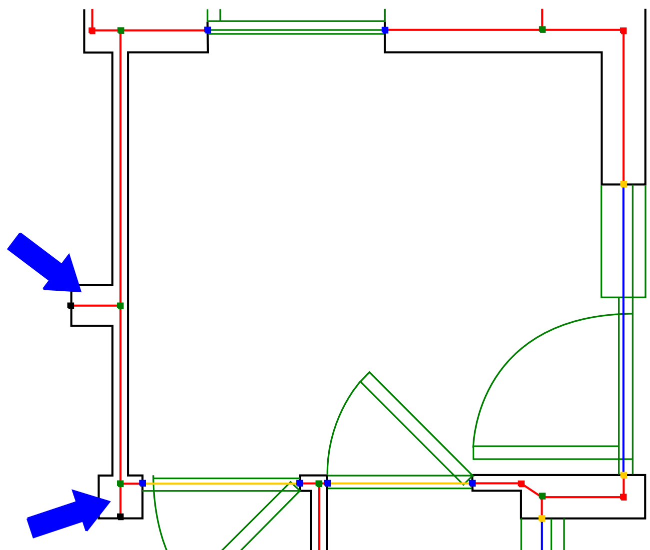


Figure 8.2: Existence of dangling edges in the topology graph

To avoid these problems, dangling edges and bridges should be removed before running the MAFL algorithm. Nevertheless, these kind of edges do not

[1] angles are measured clockwise in order to keep the coherence for non-convex polygons

produce failures in the algorithm execution. In the case of Figure 8.3a, the dangling edge e_1 is detected as part of the polygon P while the dangling edge e_2 is detected as part of the exterior. On the other hand, in the example of Figure 8.3b, which represents a connected graph with a bridge, two polygons and the exterior ring are detected.

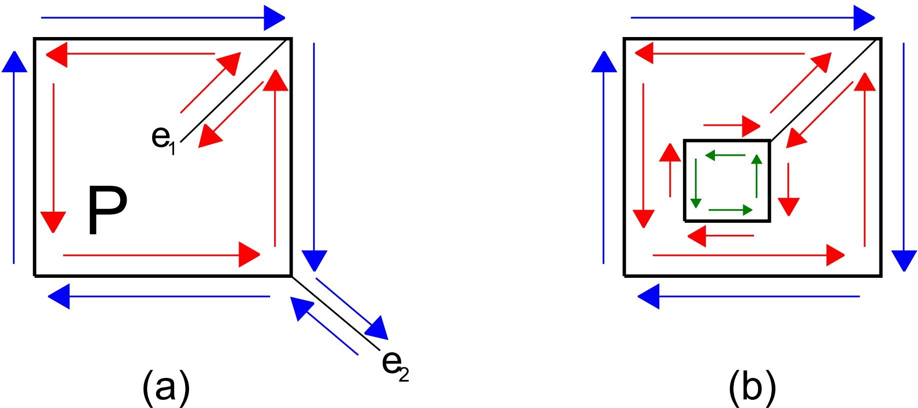


Figure 8.3: MAFL behavior with dangling edges and bridges. The arrows of the same color represent the orientation of each detected polygon.

The topological correctness of the topology graph, according to the conditions given in Section 7.1, can be proved as follows:

Proposition 1 (Topological correctness of the wall set) *If the lines from the wall layer in a CAD drawing are topologically correct, the wall edges in the computed topology graph are topologically correct.*

The two conditions for a set M of 2D points and lines to be topologically correct are that (1) each line is bounded by two points and that (2) the intersection of two lines is either the empty set or a point in M. The first condition is fulfilled since for every edge inserted in the graph, its two vertices are inserted if they were not already present. The second condition is also accomplished since the walls do not intersect (the intersecting walls were removed in Algorithm 8.1), and therefore the graph edges of type wall neither intersect.

Proposition 2 (Topological correctness of the topology graph) *If there are no intersections among the opening's bounding boxes, nor among the bounding boxes and the wall edges of the topology graph, the topology graph is topologically correct.*

This can be proved given that the set of wall edges is topologically correct; on the other hand, the door and window edges have been added by joining pairs of existing vertices, and the added edges do not intersect each other since their bounding boxes neither intersect.

8.2 Linking geometric and topological information

In the previous section we have described how topological information can be obtained to represent building floors, using graphs with 2D positioned nodes to represent the internal structure of buildings, and dual graphs representing closed spaces. The next step is the linkage between topological data and geometric data from the input floor plans. This is an important issue, as long as it allows the storage of semantic information about the geometry. The topological information is therefore an intermediate stage between the input geometry and a *labeled* geometry.

In this subsection we introduce some items of information that link the wall line set and the topology module.

- Each segment or arc from the wall line set can be assigned to a closed space using point-in-polygon tests.
- For a given closed space, its assigned segments make up a set of polylines which can be closed by adding new segments for the doors and windows. As a result, we have an *inner polygon* for each closed space, and a *outer polygon* for the exterior ring. All the segments from the inner polygons will have an assigned label (wall, window or door).

Once all the inner polygons and the outer polygon are closed, every edge of the topology graph is assigned two segments from these polygons. For the case of the walls, the segments related to a topological edge correspond to the ones detected as wall pair in the wall detection stage; for the case of the doors and windows, the two segments have been created in order to close the inner polygons.

8.2.1 Segment to closed space assignment

The assignment of segments to closed spaces is done by means of executing point-in-polygon tests for all the end points of the segments representing walls, and all the polygons representing closed spaces. Segments are assigned to polygons according to the following criteria (see Figure 8.4):

1. If both vertices of a given segment belong to the same polygon, the segment is assigned to that polygon (segment a in Figure 8.4).
2. If none of the vertices of the segment belong to any polygon, the segment belongs to the exterior (segment b in Figure 8.4).
3. If the results of the inclusion tests are different for each vertex, either they belong to different polygons (segment c_1 in Figure 8.4), or one of them belongs to a polygon and the other one to the outside (segment c_2 in Figure 8.4). In these cases, the segment is divided into two parts, and each part is assigned to the corresponding polygon, or to the outside.

4. When one vertex lies on the border between two polygons (segment d_1 in Figure 8.4) or between a polygon and the outside (segment d_2 in Figure 8.4), the segment is assigned to the polygon (or to the outside) that contains the other vertex.

All the segments assigned to closed spaces in this stage are labeled as *wall* segments.

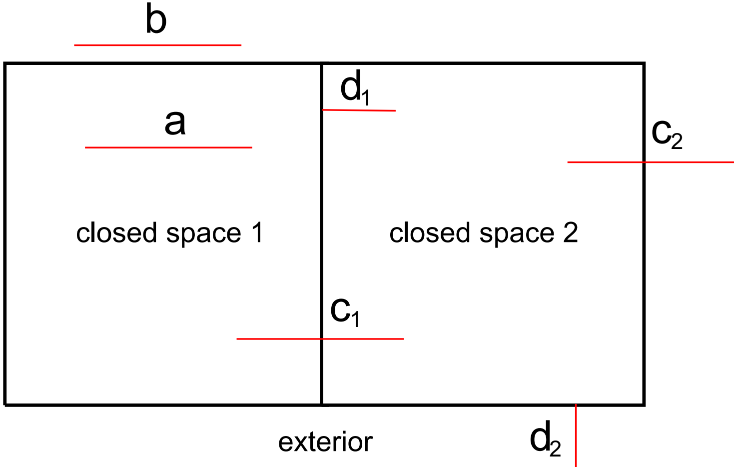


Figure 8.4: Segment to closed space assignment: a is assigned to closed space 1; b is assigned to the exterior; c_1 is split and the resulting segments are assigned to closed space 1 and 2 respectively; c_2 is split and each resulting segment is assigned to closed space 2 and the exterior respectively; d_1 is assigned to closed space 2 and d_2 is assigned to the exterior

8.2.2 Closing inner polygons

The problem of closing the inner polygon assigne to a closed space is not straightforward and requires a previous analysis of the scenario. Figure 8.5a shows two adjacent rooms connected through a door. Dotted black lines represent wall segments from other rooms; Continuous black lines represent the topological structure; finally, blue and green lines represent respectively the wall segments assigned to each one of the adjacent rooms.

A first approach to the solution consists of finding a set of segments that need to be added to join all the polylines from a closed space, so that the total length of the added segments is minimal. Nevertheless, this approach does not deal correctly with situations like the one shown in Figure 8.5.a, as it results into the solution shown in Figure 8.5.b, i.e., the inner polygons from the neighbor rooms share the red segment. A more accurate result is shown in Figure 8.5.c, where (a) inner polygons from two adjacent rooms do not share any segments and (b) the space occupied by the openings remains empty.

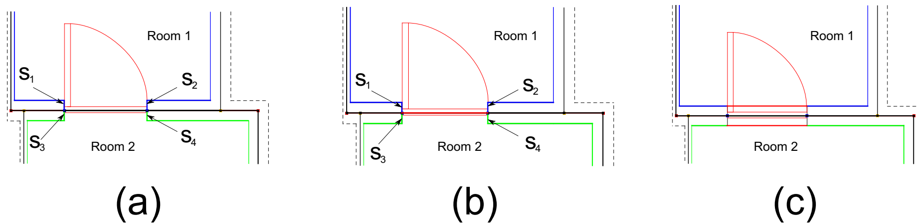


Figure 8.5: Segments assigned to two adjacent rooms: (a) Blue lines represent segments assigned to room 1 and green lines represent segments assigned to room 2; (b) segments s_1, s_2, s_3 and s_4 make the segment which closes rooms 1 and 2 be shared; (c) segments s_1, s_2, s_3 and s_4 have been removed in order to avoid that rooms 1 and 2 share the red segment.

In order to prevent these situations from happening, the set of segments assigned to a closed space needs to be simplified before computing the inner polygons using a grid (Figure 8.6). This simplification consists of removing all the *unbounded* segments. To determine whether a segment is *unbounded*, the following three steps are applied: (a) a grid made up by all the lines containing segments is computed, (b) all the intersection points between pairs of lines from the grid are determined and (c) the intersection points which do not lie inside the closed space (if it is non convex) are discarded. A segment is *unbounded* if it is not enclosed by two intersection points from the grid. Once the grid is calculated, each segment is analyzed to determine which points from the grid lie on it.

- If the segment contains less than two points, it is removed.
- If the segment contains two points, the unbounded fragment/s are removed.
- If the segment contains more than two points, only the outer points are considered to reduce this case to the above one.

Finally, the inner polygons are closed by adding new segments for doors and windows as follows: for each door or window segment from the outer polygon, the algorithm searches for the pair of polyline end points closest to them. A new segment joining the found end points is added and labeled as door or window.

Figure 8.6 shows an example of the algorithm. In (a) we show the floor plan representation of a room and its detected outer polygon; red segments represent walls, yellow segments represent doors and blue segments represent windows. In (b) the segments from the floor plan which are inside the outer polygon are highlighted in green, while gray lines and points represent the grid. In (c) the *unbounded* segments have been removed as the result of processing the set of segments and the grid. Finally, in (d) the inner polygons are closed by adding the door and window lines.

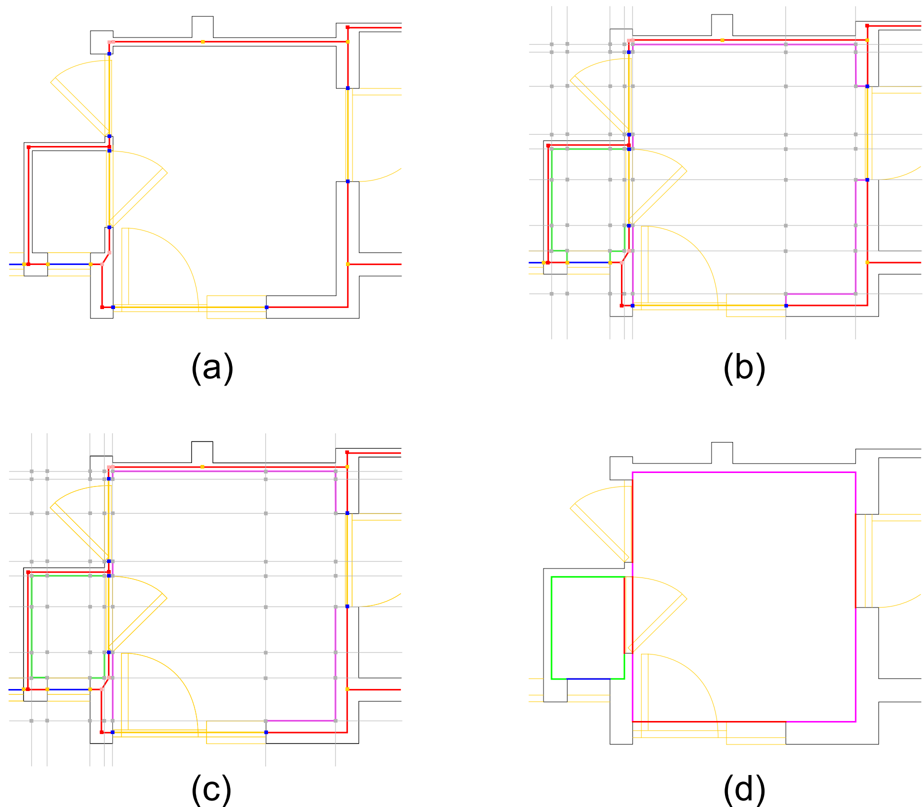


Figure 8.6: Grid algorithm: (a) Topological structure of a room. (b) Segments assigned to the room and grid. (c) The grid is used to remove *unbounded* segments. (d) Inner polygon.

The resulting data structure after linking geometry and topology is richer than the topology graph. The key points of the whole topological model are summarized as follows:

1. The topology graph contains edges representing walls, doors and windows, and its associated dual graph contains rooms and relationships of connectivity and adjacency among them.

2. Each edge from the topology graph contains information about type (wall, door, window), left and right rooms (vertices in the dual graph), and width.

3. The edges of type *wall* do not represent paper walls (they include explicitly the wall thickness).

4. Each room is associated to inner and outer polygons.

8.3 Results

Figure 8.7 shows (a) its floor plan; (b) the exterior polygons and (c) the interior polygons detected after assigning segments to closed spaces and closing the inner polygons.

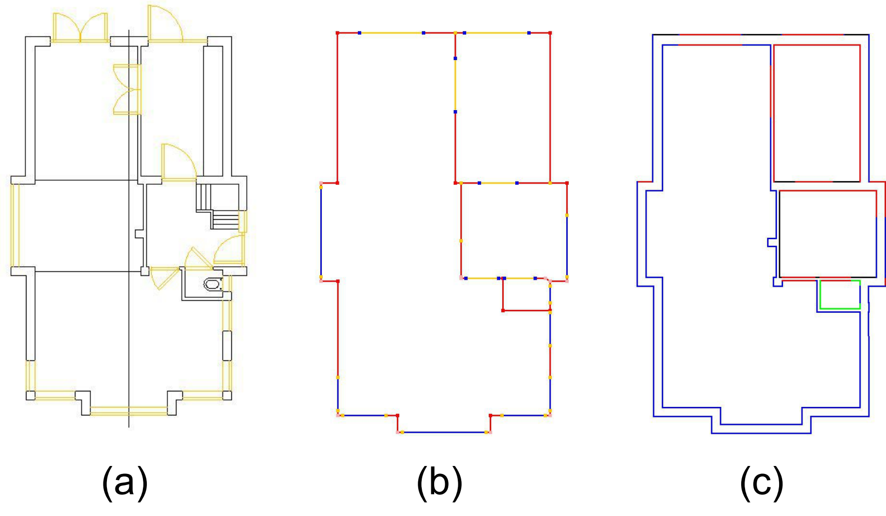


Figure 8.7: (a) Architectural floor plan of a five-room building; (b) Exterior polygons detected; (c) Interior polygons

Chapter 9

3D geometry. Triple extrusion

The framework we propose in Chapter 7 can hold so much information that it is feasible to generate a 3D model in two different LoD's. The most detailed LoD represents accurately the interior of the building. Here we are going to apply this data structure to the generation of a triple extrusion model.

The triple extrusion algorithm allows to obtain a rough 3D model of the bounding shape of a building from three CAD drawings representing the three standard views (front, side and top) only with two conditions: (1) the views must be drawn using the same scale and (2) the drawings must be correctly aligned to the drawing axes. Figure 9.1.a shows an example of the three main views of a house with sloping roofs.

Furthermore, the triple extrusion can be viewed as an intermediate step in 3D building generalization: starting with the most detailed 3D objects, triple extrusion objects can be derived. Then, convex hull models can be derived from triple extrusion objects. Finally, the most simple generalization level are 3D bounding boxes of buildings.

The triple extrusion algorithm extracts a contour polygon for each of the three views by building a closed path that begins at an external line of the drawing. In each intersection among two or more lines, the next line of the path is chosen as the one that makes up the greatest angle with the current one (Figure 9.1.b).

The three contour polygons are then extruded along the X, Y and Z axes respectively, and the intersection between the extruded solids is computed in order to get a solid that approximates the shape of the building. The extruded solids are represented as B-rep models composed of vertices, edges and faces [Män88].

In order to allow the software to process the views, the drawings must contain four separate layers for front, side and top views, and axes. The axes are necessary to compute 3D coordinates of the extruded views and ensure that the

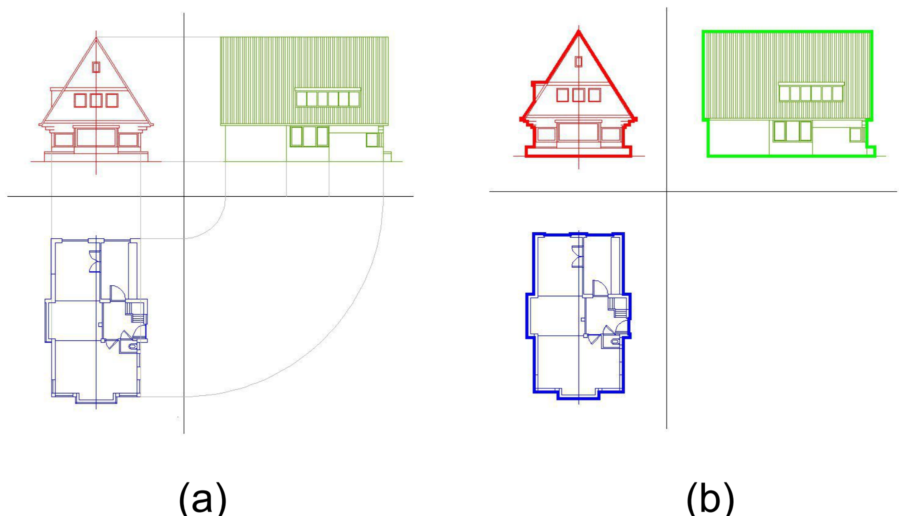


Figure 9.1: (a) Front, top and side views of a house with sloping roof; (b) Contour detection

3D solids match correctly.

The triple extrusion algorithm could therefore be divided into some tasks:

1. As a previous step, get a DXF file which contains the three views and the three axes. This file can be manually created, if necessary.
2. Import the DXF file and create a line spatial index for each view.
3. Obtain the contour polygon for each view.
4. Triangulate the contour polygons.
5. Transform the contour polygons from 2D to 3D coordinates, taking into account the correspondence points.
6. Extrude the contour polygons and compute the intersection.

Below we describe each sub-problem and the approach used to solve it.

9.1 Building a three-view DXF

In order to compute accurately the triple extrusion, the first condition is having the three views in the same DXF drawing, and using the same scale. Furthermore, each view must be in a different layer with a known name (here we use the layer names: "Top", "Front" and "Side").

An additional layer is required to contain the auxiliary axes that divide the views (we call it "Axes"). These axes will be used to put in correspondence points from different views in order to convert from 2D to 3D coordinates.

In the cases we have used to test the triple extrusion, the DXF file creation has been carried out manually, assisted by a CAD tool (AutoCAD, in our case). The drawings from different views have been respectively assigned to the layers "Top", "Front" and "Side", as well as an auxiliary layer named "Axes". The drawing from Figure 9.1.a shows an example of how a drawing must be prepared. The red, green and blue drawings represent respectively the front, side and top views, while the vertical and horizontal axes are shown in black. The gray lines joining the points in correspondence are not used in the triple extrusion algorithm, but they are shown to make the drawing more understandable.

9.2 DXF importing. Creation of spatial indexes for computing the segment intersection

Prior to computing the contour for each view, the drawings need to be previously processed in the following way:

1. Transform the primitives into straight segments. Every primitive is transformed into segments to make possible the contour detection.
2. Compute the intersections among the segments. In cases where the drawing contains crossing segments, an algorithm has to split the crossing segments to avoid errors in the contour detection.

To summarize how each kind of primitive is transformed, we recall that (1) segments do not need to be preprocessed; (2) arcs and circles are discretized into segments and (3) polylines are split into their composing segments.

Regarding the intersection computation, it is necessary to check each segment with all the other segments in the drawing, and split in case an intersection is found. This computation can be time consuming in drawings representing medium or large stories. Thus, a spatial index can be used.

In order to find a balance between implementation complexity and effectiveness of the spatial index, we just choose to subdivide each view into regular cells from a rectangular grid. The rectangle used as the basis of each spatial index is the bounding box of the corresponding view (top, front or side). Each cell contains a set of segments, and each segment is contained into one or more cells that contain it totally or partially. During the index construction, the segments that cross more than one cell are not split; despite this produces some redundancy, there are three major reasons to proceed this way: (1) this is done to avoid any additional modification on the original segment set. (2) Furthermore, the insertion and deletion of segments from the index is easier using this redundancy. (3) Finally, we need to keep track of the original segments (except those intersecting segments that are split), since the spatial index is only an auxiliary artifact.

The grid is represented by a data structure $Grid(S, C, X, Y, T(S))$ with the following fields:

- The set S of segments contained in the grid
- A matrix $C_{m \times n}$. Each cell $C_{i,j}$ is associated to a subset of S which includes the segments that traverse the cell. This subset is noted as $C_{i,j}.Segments$.
- An array $X = (x_0, x_1, ..., x_m)$ containing the divisions of the cell across the X-axis.
- An array $Y = (y_0, y_1, ..., y_n)$ containing the divisions of the cell across the Y-axis.
- A table $T(S) = \{s, \{(i,j)\}\}$ assigns to each segment s the set of cells $\{(i,j)\}$ that the segment traverses. Although this table can be obtained by inspection from the cells, it is stored in an explicit way so that it is possible to query in O(1) time the cells traversed by a segment.

The operations needed for the spatial index are: (1) compute which cell contains a 2D point; (2) compute which cells are traversed by a segment; (3) insert a segment and (4) remove a segment.

The cell that contains a 2D point can be computed in time O(1). Given a spatial index with $m \times n$ cells and divisions $x_0, x_1, ..., x_m, y_0, y_1, ..., y_n$, the cell (i,j) corresponding to a point (x, y) can be computed using the following equations:

$$i = min\left(m - 1, \left\lfloor \frac{m(x - x_0)}{x_m - x_0} \right\rfloor\right) \tag{9.1}$$

$$j = min\left(n - 1, \left\lfloor \frac{n(y - y_0)}{y_n - y_0} \right\rfloor\right) \tag{9.2}$$

Note: These equations use the min *function in order to avoid, in case $x = x_m$ or $y = y_n$, that i holds the value m or j takes the value n. These values are not valid for the cells from the spatial index.*

The cells that contain a segment whose end points are P and Q are computed in two steps: (1) the cells (i,j) and (k,l) corresponding respectively to the points P and Q are computed; (2) the cells (a,b) traversed by the segment are also computed, being $i < a < k$ and $j < b < l$. The segment is also assigned to these cells.

For the first step, we use equations 9.1 and 9.2, applied to P and Q respectively. For the second step we need to determine the path of cells traversed by the segment by computing the cells in both horizontal and vertical directions, having as inputs the coordinates of the end points and the divisions $x_0, x_1, ..., x_m, y_0, y_1, ..., y_n$. The whole process can be viewed in Algorithm 9.1. This algorithm has as input the segment end points and the grid, and as output a set $\{(i,j)\}$ of cells.

Algorithm 9.1: Compute the cells traversed by a point

```
    Input: s(P,Q): the segment
    Grid(S,C,X,Y,T(S)): The grid
    Output: cells: The set {C_{i,j}} of cells traversed by the segment
 1  begin
 2      cells ← ∅
 3      (v_x, v_y) ←(Q.x − P.x, Q.y − P.y)
 4      (i, j) ←ComputeCell(P.x,P.y,Grid)
 5      (k, l) ←ComputeCell(Q.x,Q.y,Grid)
 6      cells ←cells ∪ {C_{i,j}, C_{k,l}}
 7      if i>k then Swap(i,k)
 8      if j>l then Swap(j,l)
 9      for idx=i+1;idx≤k;idx++ do
10          c_y ← (v_y x_{idx} + v_y Q.x − v_x Q.y)/v_x
11          (i_{new}, j_{new}) ←ComputeCell(x_{idx},c_y,Grid)
12          cells ←cells ∪ {C_{idx−1,j_{new}}, C_{idx,j_{new}}}
13      end
14      for idx=j+1;idx≤l;idx++ do
15          c_x ← (v_x y_{idx} + v_y Q.x − v_x Q.y)/v_y
16          (i_{new}, j_{new}) ←ComputeCell(c_x,y_{idx},Grid)
17          cells ←cells ∪ {C_{i_{new},idx−1}, C_{i_{new},idx}}
18      end
19      return cells
20  end
```

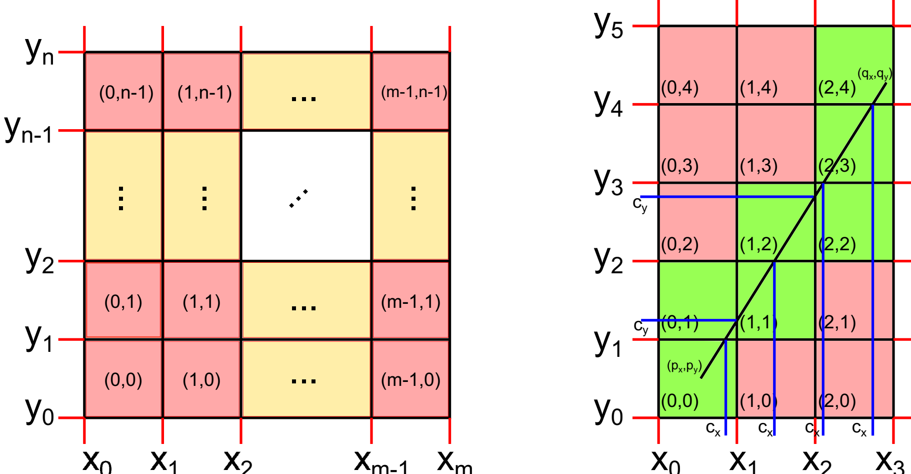


Figure 9.2: Grid spatial index

The function `ComputeCell` calculates the cells that contain a 2D point, according to equations 9.1 and 9.2. The function `Swap` is only used to ensure that the conditions $i \leq k$ and $j \leq l$ are fulfilled when the loops are executed.

This algorithm computes separately the horizontal and vertical set of cells. This may provoke that some computed cells are duplicated. To solve this, it is enough to add each cell to the output set only the first time it appears.

Figure 9.2 left shows the structure of the grid spatial index. Figure 9.2 right shows an example: how to compute the cells for a segment with end points P and Q. The computed cells are shown in green. The blue segments show the values c_x and c_y, computed as the intersections between the segment and the lines from the grid index.

The algorithm that inserts a segment into the grid spatial index has several steps: first, the cells that the segment being inserted traverses are detected; second, collisions between the segment being inserted and the existing segments are detected. In case there is a collision in the cell that is being currently checked, the colliding segments are split in their intersection point. Then they are removed from the spatial index, and the resulting new segments are inserted with the basic insert algorithm.

Therefore, we define two insertion algorithms: the basic insertion of segments, used to insert those segments that do not collide with others (for inserting the split segments), and the complete insertion, that searches for collisions and is based on the basic insertion.

The basic insertion algorithm detects the cells traversed by the segment being inserted and inserts it into every detected cell (without splitting). Besides, a set of entries is inserted into the table, associating the segment and the set of cells that it traverses. The pseudocode of this algorithm can be viewed in Algorithm 9.2. This algorithms takes as input a segment s whose endpoints are P and Q.

Algorithm 9.2: Basic insertion of a segment in a cell

```
  Input: s(P,Q): the segment
  Grid: the grid
  Output: Grid: the grid after being updated
1 begin
2     S ← S ∪ {s}
3     path ← ComputeCells(s, Grid)
4     T(s) ←path
5     forall the cells C_{i,j} in T(s) do
6         C_{i,j}.Segments ← C_{i,j}.Segments ∪ {s}
7     end
8     return Grid
9 end
```

Regarding the complete insertion, we need to check whether the segment being inserted is colliding with more than one segment. In such case, the segment needs to be split in so many segments as collisions are detected. In order to do this in one step, we need to track the position of each collision in the segment. We achieve this using an auxiliary data structure which stores the position of each collision in a sorted way.

The pseudo-code shown in Algorithm 9.3 carries out the complete insertion of the segment. Some lists need to be initialized:

- The list intersectingSegments stores the segments that intersect with the segment being inserted.
- The list intersectionPoints stores the intersection points between the segment being inserted and the segments from intersectingSegments.
- The auxiliary list sortedIntersectionPoints stores the points from intersectionPoints sorted according to their positions in the segment being inserted. The list also includes the end points P and Q from the segment. Consequently, the first point in the sorted list is P and the last is Q. The rest of points are sorted according to their relative positions between P and Q.
- The auxiliary list positions stores, for each point from sortedIntersectionPoints, its parameter value inside the segment in a sorted way. For a point P_i, this parameter consists of the value between 0 and 1 such that $P_i = (1 - \lambda)p + \lambda q$. Trivially, the first and the last values are respectively 0 and 1, corresponding to the points P and Q from the list sortedIntersectionPoints.

In order to compute the intersection between two segments we use the Algorithm C.2, that was used in Chapter 6 and is detailed in Appendix C.

The function `InsertSegment` uses some auxiliary functions that are explained below: `Parameter`, `PointInCell` and `RemoveSegment`. `Parameter` does

Algorithm 9.3: Insert of a segment in the grid spatial index

```
   Input: s(P,Q): the segment
   Grid: the grid
   Output: Grid: the grid after being updated
 1 begin
 2     sortedIntersectionPoints.Add(P)
 3     sortedIntersectionPoints.Add(Q)
 4     positions.Add(0.0)
 5     positions.Add(1.0)
 6     path ← ComputeCells(s, Grid)
 7     forall the cells C_ij in path do
 8         forall the segments (P_1, P_2) in C_ij.Segments do
 9             if lines (P,Q) and (P_1, P_2) intersect then
10                 P_int ← IntersectionSegments((P,Q),(P_1, P_2))
11                 if PointInCell(P_int, C_i,j) then
12                     intersectingSegments.Add((P_1, P_2))
13                     intersectionPoints.Add(P_int)
14                     λ ← Parameter(P_int, s)
15                     if λ ∈ [0,1] then
16                         idx ← 0
17                         while λ > positions[idx] and idx < positions.Size()
                           do
18                             idx ← idx + 1
19                         end
20                         sortedIntersectionPoints.Add(idx,P_int)
21                         positions.Add(idx,λ)
22                     end
23                 end
24             end
25         end
26     end
27     if intersectingSegments.Size() ≠ 0 then
28         for jdx = 0; jdx < sortedIntersectionPoints.Size(); jdx + + do
29             if intersectionPoints [jdx] ≠ P_1 and intersectionPoints
               [jdx] ≠ P_2 then
30                 RemoveSegment(intersectingSegments [jdx])
31                 InsertBasic(intersectionPoints [jdx],P_1)
32                 InsertBasic(intersectionPoints [jdx],P_2)
33             end
34         end
35         for kdx = 0; kdx < intersectingSegments.Size() − 1; kdx + + do
36             InsertBasic(sortedIntersectionPoints
               [kdx],sortedIntersectionPoints [kdx + 1])
37         end
38     else
39         InsertBasic(s, Grid)
40     end
41     return Grid
42 end
```

the following: given the point P_i and the segment (P, Q), it computes the parameter λ such that $P_i = (1 - \lambda)P + \lambda Q$ (see Algorithm 9.4). `PointInCell` receives a point and a cell and returns a boolean (**true** if the point belongs to the cell, **false** otherwise). `RemoveSegment` removes an existing segment from the spatial index. To achieve this, the algorithm has to remove the segment from all the cells it traverses (see Algorithm 9.5).

Algorithm 9.4: Calculation of the parameter λ of a point in a segment

Input: P_i: the analyzed point
$s(P, Q)$: the segment
Output: λ: the parameter

```
1  begin
2      if P = Q then
3          if P_i = Q then
4              return 0.0
5          end
6      end
       // horizontal line
7
8      if P.y = Q.y then
9          return (P_i.x − P.x)/(Q.x − P.x)
10     end
       // vertical line
11
12     if P.x = Q.x then
13         return (P_i.y − P.y)/(Q.y − P.y)
14     end
       // oblique line
15
16     return (P_i.x − P.x)/(Q.x − P.x)
17 end
```

In practice, we use grids with 10 horizontal and vertical divisions ($m = 10$,$n = 10$), since they provide a reasonable running time.

9.3 Contour detection of the views

The next step of the triple extrusion algorithm is the contour detection for each view. For each view, the contour detection has several steps:

1. Build a graph from the segments of each view, as obtained from the previous step (Section 9.2). This graph may contain more than one connected component.

Algorithm 9.5: Remove a segment from the grid

Input: $s(P, Q)$: the segment
Grid: the grid
Output: *Grid* : the grid after removing the segment s

```
1 begin
2     S ← S \ {s}
3     path ← T(s)
4     forall the cell C_{i,j} in path do
5         C_{i,j}.Segments ← C_{i,j}.Segments \ {s}
6     end
7     T(s) ← ∅
8     return Grid
9 end
```

2. Remove dangling edges from the graph, i.e., those edges that have any of their vertices of degree zero. The removing process is iterative: after removing dangling edges, there may appear new dangling edges.

3. Find one external edge in the resulting graph. If a ray is thrown from an arbitrary point in a direction, we say that an edge of the graph is external if it is the furthest one intersected by the ray with respect to its initial point.

4. If the graph has more than one connected component, the component containing the detected external edge is selected.

5. Detect the contour of the selected connected component.

The implementation of the steps 1 to 3 can be easily deduced from the previous description.

The fourth step consists of detecting the graph connected components and select the unique[1] component that contains the found external edge.

The last step consists of detecting the contour itself. Starting from the external edge, the selected connected component is traversed following a path of adjacent edges. In each step, the next edge to be chosen is the one that forms the greatest angle with respect to the current one. The MAFL algorithm (Algorithm 8.2 in Section 8.1) is used to detect the contour.

As explained in the room detection section, there are always two polygons sharing one edge. One of them is interior (the sum of angles is coherent) and the other one is exterior (the sum of angles is not coherent to the polygon definition). In this case, we are interested in choosing the polygon without a coherent sum of angles. For each view (front, top and side), this polygon is the detected contour.

[1] By definition, a graph cannot have more than one connected component containing the same edge.

9.4 Contour polygon triangulation

Before extruding the polygons and intersecting the solids obtained by extrusion, we compute a triangulation of the contour polygons. As no special condition is required on the triangulation, we opted for a triangulation algorithm called ear clipping [SE02]. Intuitively, it consists of traversing all the possible sets of three consecutive vertices from the polygon sequentially. For each set of three vertices, the algorithm checks if the segment connecting the first and the third point from the set make up a diagonal of the polygon[2]. In affirmative case, the triangle formed by the three considered points is added to the triangle set. The second point (the one which does not belong to the found diagonal) is removed from the initial polygon and the triangulation algorithm is recursively launched for the resulting polygon. The recursive algorithm finishes once the polygon to be triangulated has only three vertices (which are included as a triangle in the triangle set).

The pseudo code for the triangulation algorithm can be found in page 773 of the Schneider and Eberly book[SE02].

9.5 2D to 3D coordinate transformation

Once the contours have been triangulated for all the views, it is necessary to transform from the 2D coordinates of the drawing of the three views into 3D coordinates, so that the contours from each view are translated to the 3D space in a coherent way: each view must be correctly oriented and aligned in 3D with respect to the other views.

To do this, we make two assumptions, as stated in the introduction of this section:

- The drawing with the three views must contain a layer named "Axes" with two lines: a horizontal line which separates the front and the top views, and a vertical line which separates the front and the side views. Furthermore, the intersection of these lines will be considered as the point (x_0, y_0) for obtaining the 2D-to-3D transform.
- Each view must be drawn correctly, aligned to each other, and using the same scale.

The three views are drawn in the same 2D coordinate system, and the intersection of the axes is the point (x_0, y_0) (Figure 9.3.a). In order to correctly orientate the 3D extruded object, we use the following correspondences between the 2D and 3D axes (Figure 9.3.b):

- Half-axes X^+ and Y^+ from the front view correspond respectively to the half-axes Z^- and Y^+ from the 3D space.

[2] The segment which connects two polygon vertices is a diagonal (or, equivalently, that two polygon vertices are visible to each other), if that segment is not intersected by any edge of the polygon.

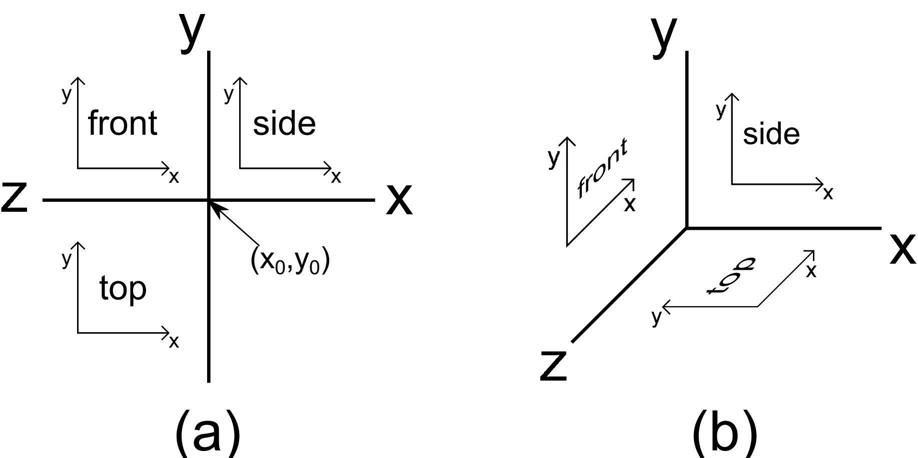


Figure 9.3: Correspondence between 2D and 3D axes

- Half-axes X^+ and Y^+ from the side view correspond respectively to the half-axes X^+ and Y^+ from the 3D space.
- Half-axes X^+ and Y^+ from the top view correspond respectively to the half-axes Z^- and X^- from the 3D space.

These correspondences are used to transform the coordinates from 2D to 3D before the extrusion. For each point from any view, two 3D points are generated in the two half-spaces defined by the plane that corresponds to the view (XZ for the top view, XY for the side view, and YZ for the front view, according to Figure 9.3.b). The distance from those points to the plane are respectively $+\infty$ and $-\infty$ (in practice the values used are very high with respect to the drawing dimensions).

The 2D local coordinates of each view are measured with respect to the drawing axes. Suppose that the coordinate origin (the intersection between the axes) that is read from the drawing file is located at (x_0, y_0). If the front, side and top view points are noted respectively as (x_f, y_f), (x_s, y_s) and (x_t, y_t), the transform is as follows:

$$(X, Y, Z) = \begin{cases} (\pm\infty, y_f - y_0, x_0 - x_f) : front \\ (x_s - x_0, y_s - y_0, \pm\infty) : side \\ (y_0 - y_t, \pm\infty, x_0 - x_t) : top \end{cases}$$

9.6 Extrusion of the views

After transforming the coordinates to 3D, the 3D triangles are generated. Each triangle is extruded to a triangular prism (6 vertices and 5 triangles), as can be viewed in Figure 9.4.

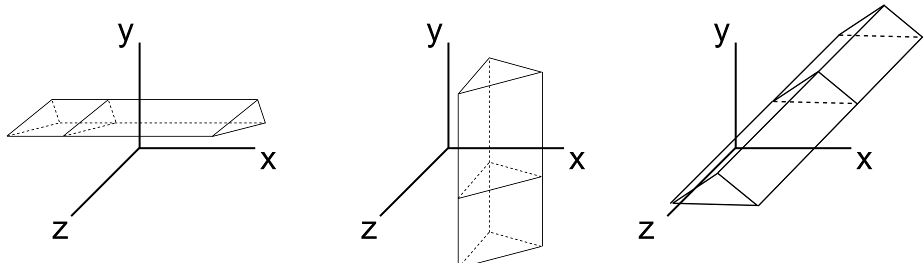


Figure 9.4: Extrusion of the views

9.7 Results

Figure 9.1.b shows the result of the contour detection on the three view drawing; Figure 9.5 shows the triple extrusion of the house (top) and some views of the solid obtained as the result of the intersection (bottom).

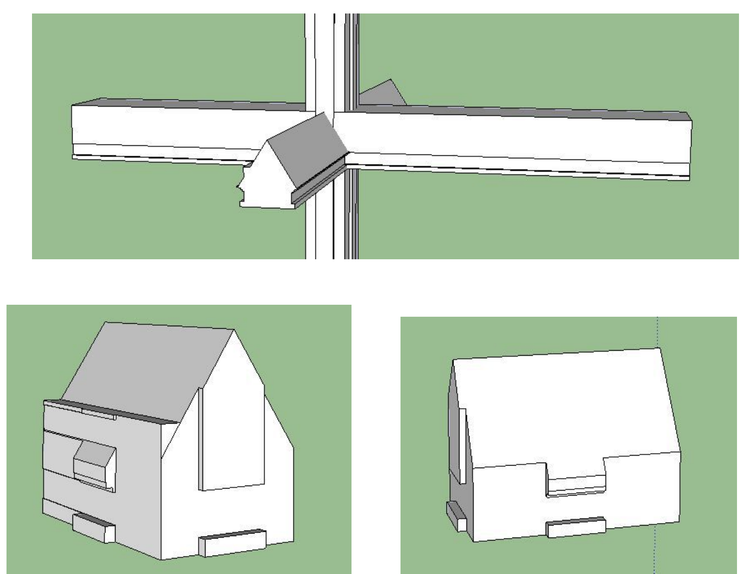


Figure 9.5: Triple extrusion (top); Intersection (bottom).

Chapter 10

3D topology and semantics. CityGML model generation

In order to generate a 3D model of a building floor from the topology and semantics information detected it is just necessary to map the semantic information from our data structure to the desired representation schema. Regarding the CityGML structure, Figure 10.1 shows a simplified UML diagram of the CityGML Building Module which contains the relevant classes from CityGML together with classes from our model.

The information from the UML model in Figure 10.1 is organized into four namespaces in order to make the structure easier to understand:

1. *bldg* contains the classes from the CityGML Building Module which are relevant to our system.

2. *gml* contains the underlying geometry for CityGML, according to the GML schema.

3. *semantics* contains the classes of the semantic module of the architecture presented in this work.

4. *geom* contains the classes of the geometric module of the architecture presented in this work.

The first two namespaces contain a subset of classes of the CityGML schema which are needed in order to link our model and the CityGML one. Since the CityGML schema contains classes and relationships which allow for high flexibility, we need to delimit the behavior of the subset of CityGML classes, according to the CityGML encoding standard [GKCN08]: (1) *building*, which is a subclass of *_AbstractBuilding*, contains zero or more interior *rooms* and is bounded by zero or more *boundary surfaces* (walls, floors, ceilings, etc.); (2) each *room* is bounded by zero or more *boundary surfaces*; (3) *boundary surfaces* are

geometrically defined as *multisurfaces* and may contain *openings*; (4) *openings* are geometrically defined as *multisurfaces* as well.

The relationship *boundedBy* that links *rooms* to *multisurfaces* in the original CityGML schema has been omitted since it may add geometric redundancy in the model.

The *semantics* namespace contains the following classes and relationships:

1. *_ClosedSpace* is an abstract class which represents closed spaces like rooms and building contours. It contains one interior polygon and one exterior polygon.

2. *OrientedPolygon* is a sorted sequence of typed vertices (see *geom* namespace explained below).

3. *Room* inherits from *_ClosedSpace* and is one-to-one related to the *bldg::Room* class. The interior polygon of a room is a counterclockwise oriented polygon made of the results of the segment to closed space assignment and the inner polygon closing algorithms (see Section 8.2.1), while the exterior polygon is the result of the closed space detection (see Section 8.2.2), i.e. a polygon containing the medial axis of the walls that enclose the room.

4. *Contour* inherits from *_ClosedSpace* and is one-to-one related to the *bldg::_BoundarySurface* class, since the contour is used to fill the *boundedBy* property of the CityGML model. The interior polygon of the contour contains the closed polygons built from the unassigned segments once the segment to closed space assignment algorithm has been run (see Section 8.2). The exterior polygon is unused.

5. *edgeType* is an enumerated type containing tags for the edges of the oriented polygons. Currently, the three available tags are *wall*, *window* and *door*.

The *geom* namespace contains classes to represent the underlying geometry of the classes from the *semantics* namespace: *semantics::OrientedPolygon* is an aggregation of zero or more *geom::Point2D* which correspond to the counterclockwise oriented vertices of the polygon; on the other hand, each pair of *geom::Point2D* is linked by a *geom::Edge*. The latter class is tagged with its type (wall, door or window).

The steps followed to build the CityGML model from our model are described below:

1. A building has one instance of *semantics::Contour* and zero or more instances of *semantics::Room*.

2. *semantics::Contour* is indirectly associated with *bldg::Building*. since *bldg::Building* is *boundedBy* a *bldg::_BoundarySurface* constructed from *Contour*.

3. A *bldg::Opening* (door or window) is associated with a *bldg::MultiSurface* representing its enclosing rectangle.

4. For each instance of *semantics::room*, an instance of a CityGML *bldg::room* is created.

5. The interior polygon of the room is used to generate the GML geometry.

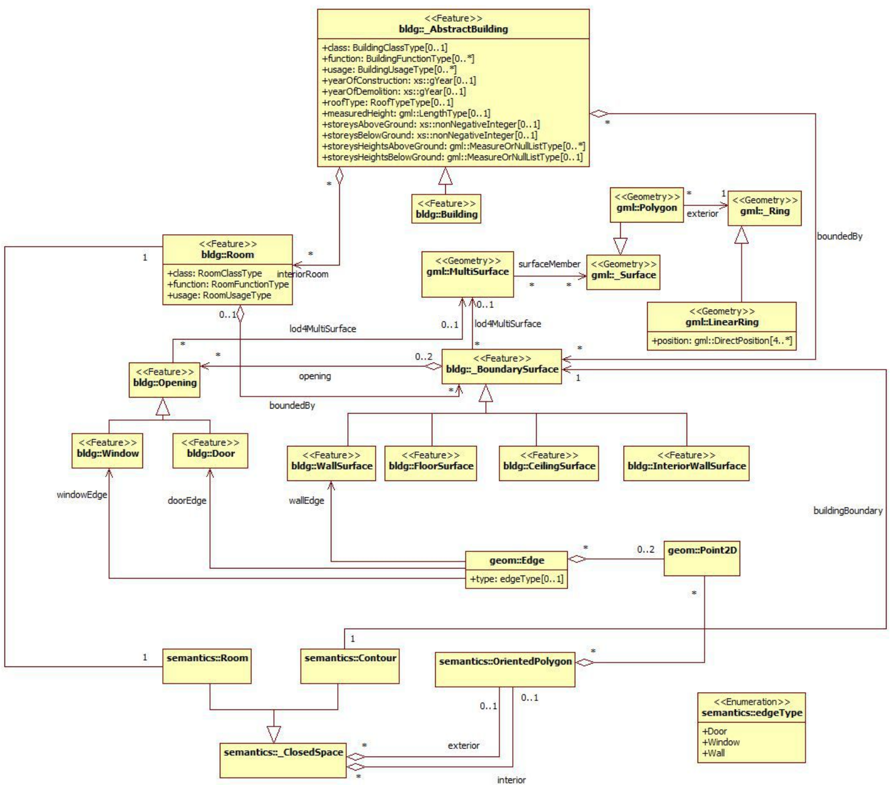


Figure 10.1: A UML diagram representing a submodel of CityGML and the linking relationships between CityGML and our model. *bldg* and *gml* namespaces are used to tag CityGML classes, while *semantics* and *geom* namespaces are used to tag classes from our model

10.1 Results

Figure 10.2 shows the generated CityGML model which includes the detected semantics.

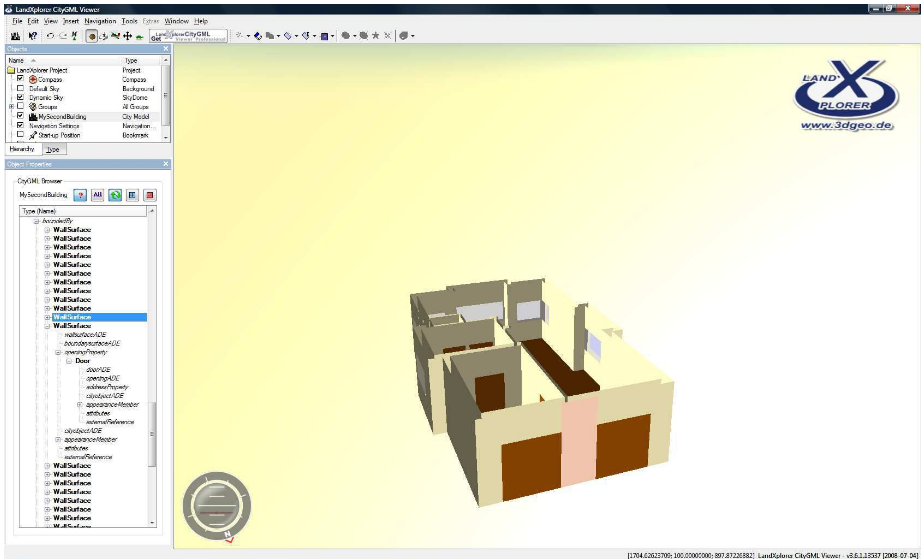


Figure 10.2: CityGML model of the building. On the left side a browser of the semantic structure is shown; on the right the 3D model is represented and the selected item in the browser appears highlighted.

Other results were already shown in Section 6.4 in order to show the output of the topology graph detection.

Part IV

Conclusions and future work

This part briefly states the conclusions and the future work of this dissertation

Chapter 11

Conclusions and future work

The last chapter of the document contains the conclusions and the future work that will be done in order to improve the results achieved during the research.

11.1 Conclusions

In this dissertation we have dealt with the representation of building information models under three scopes: geometry, topology and semantics. In order to propose a unified model we have: (1) analyzed a number of existing models, both from existing standards (e.g. CityGML) and from standards proposed in the literature; (2) studied sets of CAD architectural floor plans containing only geometric information.

This two issues have led us to:

1. Design and implement some algorithms to pre-process architectural drawings which contain common irregularities produced during the drafting of floor plans.

2. Test the algorithms with real data and get some results.

3. Propose a framework with three-levels (geometry, topology and semantics) and two scopes (2D and 3D).

The first part of the thesis introduced the motivation of the work, in the scope of a project granted to the University of Jaén, and the previous works on (1) automatic detection of geometric, semantic and topological elements from raster and CAD floor plans; (2) representation of 2D, 2.5D and 3D buildings containing geometric, topological and semantic information.

The second part described the algorithms to detect semantic elements from floor plans. The first approach was the local room detection, a rule-based algorithm to detect rooms from drawings containing low-level primitives for walls and openings. The main advantage of this algorithm was its independence of the column layouts. Nevertheless, this approach was discarded due to:

- Overtraining of the algorithm.
- The initial set of rules was designed after analyzing a reduced number of situations. It was expected that the number of rules could grow in excess.
- The situations covered were only orthogonal.

The second approach is the global room detection. The detection of semantics has been divided into some tasks: wall detection, opening detection and construction of the topology graph (using point clustering). These three tasks make up the main contribution of this dissertation. For each task, some options have been considered, providing us with families of algorithms improved after testing them and analyzing further floor plans.

Regarding the wall detection, we have proposed the Wall Adjacency Graph (WAG): a theoretical framework that detects all the possible layouts between a pair of segments and split them to detect walls. This approach has also a generalized version, the Generalized WAG (GWAG), which deals with layouts with more than two parallel segments. The result of the wall detection is a set of edges which represent the middle axis of walls. The results of this algorithms are very accurate: (1) together to the irregularity detection and fixing, the wall detection algorithm is robust; (2) the accuracy of the wall detection is very high. It only depends on two intuitive parameters: a minimum and maximum wall width thresholds given by users; (3) the algorithm is very fast, consuming only a few seconds for a large building story.

For the opening detection we have implemented an algorithm which analyze each insert that represents an opening and finds two anchorage points to the existing wall edges. This algorithm is very effective in most of common situations, which suppose a high percentage of the actual situations that can be found on real floor plans. The result of this algorithm is an initial topology graph including walls and openings, without the intersection between walls.

Finally, the clustering algorithms detect those intersections between walls in order to construct a connected topology graph that can be analyzed to search for closed spaces (rooms and corridors). This is the most complex part of the global room detection: the number of possible layouts is impossible to characterize. Thus, sometimes the accuracy of the algorithm could be better. However, we have proved that, with a reduced number of manual corrections made by users, the accuracy of the final topology graph can become higher than 90%.

The third part of the dissertation deals with the creation of a three-level framework to represent all the information obtained in the previous part, and

the proof of the topological correctness of the proposed models. Besides, a new artifact is proposed in order to achieve an initial approach to the 3D geometry of buildings: the triple extrusion algorithm. It is based on a intuitive idea: extrude each view (top, side and front) of a building and get a 3D solid after intersecting the extruded views.

The main challenges of this part are: (1) test the proposed three-level framework with more real buildings and (2) study in depth the triple extrusion algorithm: situations correctly detected and possible limitations, optimization in the calculus of the intersection and test with more buildings in order to achieve urban environments.

11.2 Future work

There are some questions that can be further studied. In this section we provide a list of the open issues after finishing this document.

11.2.1 Wall detection algorithm

- Study in depth the theoretical properties of the WAG and the GWAG. Study its use in other non-related problems.
- Generalize the WAG to other primitives different from segments and circular arcs.
- Deal with complex shapes of walls: non-rectangular walls, walls made up by segments and arcs, etc.
- Reduce the user intervention by detecting automatically the wall width thresholds.

11.2.2 Opening detection

- Implement algorithms to work with less constrained situations: openings not drawn as blocks, opening blocks whose definition is not aligned to the X and Y-axes, unstructured drawings without different layers for walls and openings, etc.
- Improve the execution time of the detection of openings from unstructured set of primitives by means of using more efficient intersection tests.

11.2.3 Clustering. Construction of the topology graph

- Improve the accuracy of the clustering algorithms. This can be done by testing the methods with further plans and achieve more general detection methods. Reduce the intervention needed by users.

- Study the numerical stability of the problem: deal with situations where the clusters are not correctly built due to inaccurate thresholds to collapse points or consider two segments as parallel. This problem is complex since the resulting topology graph often contain visually undetectable inconsistencies (loops, vertices not correctly collapsed, duplicated edges, etc.).
- Regarding the line growing algorithm: test and study the algorithm in depth and solve the problem of numerical instability to collapse points or considered two segments as parallel.

11.2.4 Three-level framework of topologically correct models

The main advances in this area can be done by testing the framework with more real cases and proving the topological correctness. This framework, together with the detection algorithms, could be used to build large environments of buildings.

11.2.5 Triple extrusion

The triple extrusion algorithm is one of the newest parts of the dissertation. It can become a powerful tool to get complex urban environments using architectural drawings, but the properties of the algorithm need to be studied:

- Study the impact of the geometric properties of the drawings in the accuracy of the resulting 3D solid.
- Improve the execution time of the boolean intersection calculus and integrate it into the user application. For this algorithm, we have used existing libraries in C++ which required to export/import data manually to obtain the 3D solids. The use of ad-hoc intersection algorithms is an open question.

11.2.6 Other future work

There are other open questions, not included in the previous sections:

- Generalize the semantic detection problem to other elements: lifts, electric installation, plumbing, furniture, open spaces.
- The detection algorithms only used the floor plan. The use of top and side views to automatically estimate some parameters that need to be provided by users (height of stories, size of openings, etc.) is an open question.
- The detection algorithms work with the information of one story. An interesting question is to adapt the algorithms and the user tool to work with several stories from the same building: take advantage of common features, join different stories after detecting staircases and lifts, detecting facades, etc.

Appendices

Appendix A

Analysis, design and implementation of an end-user tool

In this appendix we introduce the details of the work presented in this document from an implementation point of view. In any research work in Computer Graphics, the methodology of the design and implementation of solutions for the presented problems is a very important part. Nevertheless, it is rarely given the relevance that it deserves. That is the reason why we include in this appendix most of the details related to the implementation of a desktop application that implements the algorithms in this work.

The purpose of the application includes three main aspects:

- Develop a software architecture that serves as a test bench to easily include new algorithms: each algorithm has several common parts that are abstracted and generalized in the test application using software design patterns. Among the common parts for all the algorithms we can underline these: taking inputs from the canvas and/or from the outputs of other algorithms, visualization of the results on the canvas, interaction with the canvas to select inputs and highlight elements, using and synchronizing threads for different parts of the algorithm, undoing/redoing algorithms, saving to disk the result of the algorithms, etc.
- Visually and numerically validate the results of the algorithms, and modify them if necessary (e.g. by interacting with the canvas).
- Lay the foundations to build in the future an end-user application which can be interesting for engineers, architects, etc.

The aim of the appendix is not to present intensively all the functions and program code of this work, but to give a detailed overview to understand how the

software has been structured. First, we summarize the main algorithms. Then, we will introduce the application requirements and analysis, and the software design patterns. Finally we will give some details about the interface design process.

A.1 Application features

The classic software engineering process is divided into three main stages: analysis, design and implementation. The analysis stage is devoted to obtain a fully detailed description of what the user needs and which features have to be included in the software in order to cover those needs. This is a milestone in the process, as all the subsequent stages depend on its results. Here we will describe the requirements for this application.

First of all, we will enumerate the basic features that the application includes:

- The application is able to load floor plans stored in a widely accepted file format. Autodesk® DXF file format [Aut11] is a good choice, as it was originally designed for the exchange of information amid CAD applications, and it is widely used nowadays.
- As the application is going to deal with architectural drawings whose information is usually divided into layers, the software must allow the user to select the layers that contain relevant information about the architectural elements that have to be detected.
- The software must allow the user to select the blocks that contain relevant information about the architectural elements that have to be detected.
- Information about architectural elements like walls, rooms, corridors, staircases, doors or windows must be obtained as the result of the processing of the floor plan, so that corresponding virtual 3D elements could be built. This implies obtaining the semantic meaning of different sets of geometrical components (curves, lines, points, arcs, inserts...) that are included in the floor plan.
- The information about the semantic elements detected has to be stored in a database, so that 3D virtual elements that correspond to them can be created easily.

As the goal of this application is very specific, this application is not intended for common users, but for people familiar with terms and concepts from architecture and engineering. They are used to work with computers, specially with CAD/CAM related software and the way information in a drawing is organized in layers.

Other related features that will improve the user experience are:

- The application will allow the user to treat a story as a *project*. A project can be saved and loaded.

- It will be possible for the user to de/select and visualize the layers and blocks as needed. The information about the semantic elements detected could also be displayed over the CAD drawing in order to test its accuracy.
- The detection process will be semiautomatic, i.e. the user will be able to customize the way the detection algorithms are applied and select areas from the drawing to try different values of the parameters for these algorithms; moreover, the user intervention will be occasionally needed to select/correct the result for ambiguous or unexpected situations.
- Every action from the user could be undone in order to deal with user mistakes and incorrect choice of parameters.

A.2 Application Analysis

In this section we will focus on the requirements and features of the user interface for our software application, and try to give effective solutions for them.

A.2.1 Task analysis

The user interface design process requires a detailed task analysis, where the user actions that produce a change in the status of the application are listed. Notice that every action is started by the user. The parameter selection and the algorithm execution have been separated in order to make the interaction more flexible. The following tasks have been considered:

1. **Create a new project**: the application ends all the previous processes (allowing the user to save or discard the changes, if any) and create the necessary environment for a new story.
2. **Load an existing project**: the application reads an existing project from a file, and shows on a canvas all the information about the story that it represents.
3. **Close project**: the application closes the current project, allowing the user to save or discard the changes, if any.
4. **Save project**: the application saves the current project. If the project has not been saved before, the user must specify a file name.
5. **Save project as**: same as the above task, but allowing the user to specify a file name, even when the project has already been saved.
6. **Exit**: the application ends allowing the user to save or discard the unsaved changes, if any.
7. **Load DXF file**: the application loads and parses a DXF file specified by the user, and renders its content on the canvas. Two lists containing layers and blocks are shown to the user.

8. **Change layer selection**: the application allows the user to select some elements from the layer list. The drawing is refreshed to show only the entities from the selected layers.

9. **View block definition**: before running some algorithms, the user must select sets of block names which have to be detected. This task is useful as a previous step, to show the aspect of any block selected by the user.

10. **Change zoom**: the application refreshes the drawing when the user specifies a new zoom value.

11. **Pan**: the application refreshes the drawing when the user clicks and drags the mouse over the canvas.

12. **Select area**: the user selects a rectangular area from the canvas, then the application computes and stores the selected entities as a previous step to run an algorithm.

13. **Change layer selection for walls, openings and staircases**: the user selects some elements from the layer list to be used as wall, opening or staircase layers for the detection algorithms.

14. **Change block selection for openings**: the user selects some elements from the block list to be used as door/window blocks for the detection algorithms.

15. **Change threshold for wall detection**: the user specifies a new value for this threshold.

16. **Change threshold for point clustering**: similar to the above task.

17. **Apply threshold for wall detection and key point extraction**: Once the wall layers, opening layers and blocks have been selected, and the width threshold and (optionally) an area from the drawing have been specified by the user, the wall detection and key point detection algorithms are launched.

18. **Apply threshold for staircase detection**: Once all the staircase layers and a width threshold are specified by the user, the staircase detection algorithm is launched.

19. **Detect rule-based key points**: Once all the parameters cited above have been introduced by the user, the rule-based key point detection algorithm is launched.

20. **Add key point**: the application places a new key point on the floor plan. The user selects its type and its location by clicking with the mouse on the canvas or specifying its coordinates.

21. **Delete key point**: the application deletes a key point selected by the user.

22. **Change key point type**: the application changes the type of a key point selected by the user.
23. **Move key point**: the application moves a key point when the user clicks and drags it or specifies its new coordinates using the keyboard.
24. **Add room**: the application adds a new room to the model, as a sorted set of key points selected by the user.
25. **Delete room**: the application deletes a room selected by the user.
26. **Export story to the database**: the application stores the current story into a database, with all the information detected by the algorithms. The story is saved as part of a building.
27. **Delete story from the database**: the application deletes any existing story from the database.
28. **Delete building from database**: the application deletes a building and all its stories.

A.2.2 Task structure

For each task, its whole process must be described, from the moment it is launched by the user to the moment it finishes. This is quite similar to describing use cases. For each task, the interaction steps between the user and the system have been described. A task has usually the following structure: (1) the user launches the task by interacting with the application; (2) the application asks the user for the parameters needed to execute the task; (3) the task is executed and the result is shown into the interface. All the tasks fulfill this structure. For example, task 17 would have the following description:

1. The user requests the interface for the task by clicking on a button.
2. The application executes the wall detection algorithm on the selected layers and blocks, using the thresholds previously specified by the user.
3. The result of the algorithm execution is shown on the interface.

A.2.3 Interaction architecture

This stage takes as input the tasks description, and analyzes in which way they are combined. It provides us with a consistent interaction logic. The allowed pipes of the user interaction are defined after this analysis, so that some errors can be avoided. The interaction architecture analysis has two steps:

- Build a tree hierarchy with every system task (different from the user tasks analyzed before). The root contains just one node that represents the whole application. The first level of the tree represents the tasks which are part of the main execution pipeline. The lower levels represent the successive execution pipelines for each task node.

- A task does not need to be executed by invoking its children sequentially, it may have different running paths. Thus, we need to establish a set of allowed running paths for each non-leaf task from the tree. The user interface has to be designed according to the allowed running paths, enabling or disabling the controls as necessary.

Another important issue is to find the states of busy waiting. They are determined by the tasks that spend a long time of execution, and the user cannot do anything else until the task ends. These states of busy waiting have to be taken into account while designing and implementing the application. In these states, the user will need to receive some feedback and has be able to cancel the task, keeping the application in a consistent state.

The document that compiles the result of the interaction architecture analysis is called *hierarchical task analysis* (HTA) diagram[DFAB03]. The HTA diagram for this application is:

0. Building edition application
1. New project
2. Open project
    2.1. Choose project
    2.2. Wait for project loading
3. Save project
4. Save project as
5. Edit project
    5.1. Load DXF file
    5.2. Change layer selection
    5.3. Change block selection for doors and windows
    5.4. Change threshold values
    5.5. View block
    5.6. Change zoom
    5.7. Pan
    5.8. Select area
    5.9. Room detection
        5.9.1. Change layers and blocks selection for algorithms
        5.9.2. Detection of irregularities
        5.9.3. Detection of walls
        5.9.4. Detection of key points
        5.9.5. Vertex search

5.9.6. Clustering
5.9.7. Staircase detection

5.10. Semiautomatic post-process

5.10.1. Add key point
5.10.2. Delete key point
5.10.3. Change key point type
5.10.4. Move key point
5.10.5. Add room
5.10.6. Delete room

6. Close project

7. Database management

7.1. Export story into database
7.2. Delete story from database
7.3. Delete building from database

8. Exit

The last step of the study of the interaction architecture consists into specifying the allowed paths for non-leaf tasks. In order to represent a path, we will use regular expressions.

- The *Open project* task (number 2) is always sequentially executed: 2→2.1 2.2
- The *Edit project* task (number 5) must always begin with subtask *Load DXF file* (number 5.1). Then, any other subtask can be launched (5.2 to 5.10). Then, the allowed path for this task would be 5→ 5.1 (5.2 | 5.3 | 5.4 | 5.5 | 5.6 | 5.7 | 5.8 | 5.9 | 5.10)*.
- The *Room detection* task (5.9) must begin with *Change layers and blocks selection for algorithms* (5.9.1), as we assume there is no initial selection, and then any task must be launched (including 5.9.1). The allowed path is 5.9→ 5.9.1(5.9.1 | 5.9.2 | 5.9.3 | 5.9.4 | 5.9.5 | 5.9.6 | 5.9.7)*
- Every path for the *Semiautomatic post-process* task (5.8) is allowed: 5.10→ (5.10.1 | 5.10.2 | 5.10.3 | 5.10.4 | 5.10.5 | 5.10.6)*
- Every path for the *Database management* task (7) is allowed: 7→ (7.1 | 7.2 | 7.3)*
- The allowed path of execution for the whole application (task 0) is more difficult to model, since it depends on the application state. If we want to avoid situations such saving a project which has not been modified, or edit a project when there is no open project, it is difficult to make a regular expression. Instead of that, a state diagram which models the allowed path will be described.

The states are based on three parameters: (1) either a project is open or not, (2) either a project has been modified or not, and (3) either a project has a file name or not. These 3 parameters would result into 2^3 different states, though only five of them are useful: when there is not any open project, the other two parameters does not make sense. Two additional states are added: an error state, reached by every non-allowed transition to ensure completeness of the state diagram; and a final state, reached when the application terminates. Table A.1 summarizes all the states.

Open project	Project has file name	Project modified	Application state
No	Irrelevant	Irrelevant	q_1
Yes	No	No	q_2
Yes	No	Yes	q_3
Yes	Yes	No	q_4
Yes	Yes	Yes	q_5
Error state			q_E
Final state (application terminated)			q_F

Table A.1: Summary of the application states

Tasks 1 to 8 from the HTA are the transitions for the diagram. Figure A.1 describes the behavior of the state diagram. Each cell represents the new state for each state/task pair.

A.2.4 Client-server architecture

This application is a component of a complete client-server system for the generation, management and visualization of indoor information. This system allows us to store not only the geometry and topology of the scene, but also other data that can be added to the final result in order to give additional information to the user as s/he navigates the scene. Moreover, the use of a database allows us to separate the feature extraction process from the 3D model generation, and therefore it is easy to use different file formats as output, just by simply changing the module that generates the output.

Our system is intended to use a client-server approach, as Figure A.2 shows. In this system, the server processes the floor plans and extract significant features that will be used afterwards to generate the 3D model. These features are saved in a database that can optionally store additional information related with the building, like the location of business, people office numbers, etcetera.

On the other hand, the clients send queries to the server, and it replies with a full-featured 3D description of the floor of the building. This description is encoded using any appropriate standard 3D file format that can be viewed using any suitable application, either a web browser plugin or a standalone application.

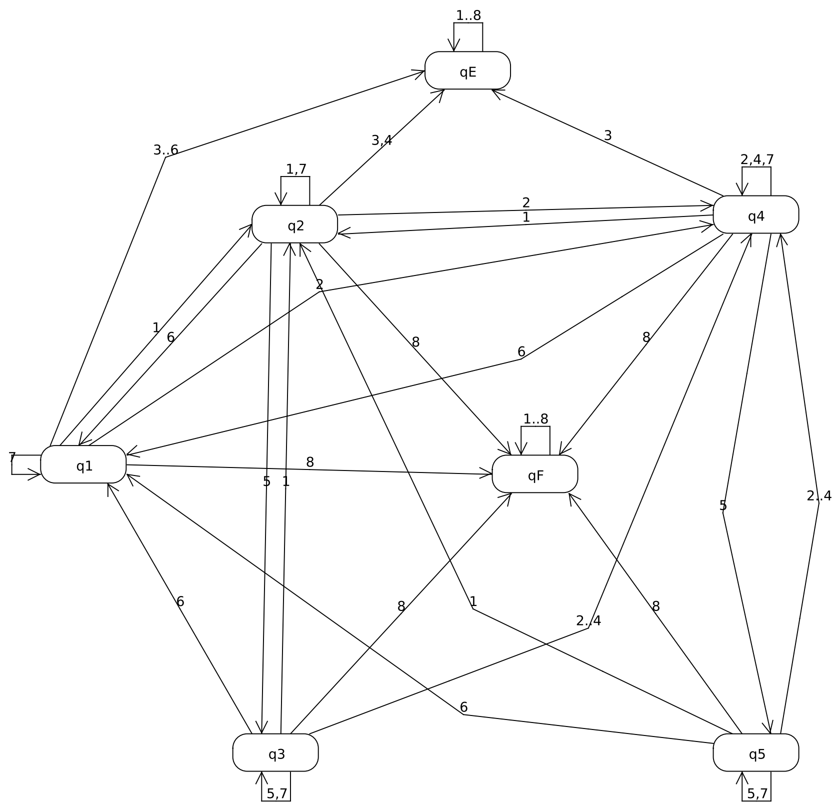


Figure A.1: State diagram

A.2.5 Database structure

In this work we have used a relational database with spatial extensions to store the significant features from the floor plans. MySQL was chosen as RDBMS to manage this database, as it includes spatial extensions that follow the specifications of the Open Geospatial Consortium (OGC) [OGC, MyS]. Using these extensions, it is possible to store not only standard data, but also geometric attributes like points, polylines or polygons, as well as use built-in spatial functions that take as input these new data types.

Figure A.3 shows a simplified view of the structure of the database. In this figure, all the attributes and some relationships have been omitted in order to enhance the legibility of the diagram.

Additional relationships are also included in the database in order to model hierarchical behaviors amid the stored features. This way, it is possible to obtain topological knowledge by means of appropriate queries.

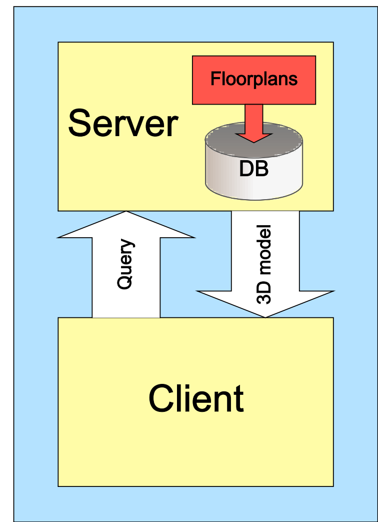


Figure A.2: Task flow of the proposed system

Two special elements that need to be considered are elevators and flights of stairs. The solution that suits best our needs is to consider them once for each floor of the building; this includes some redundancy in the database, but as there are usually very few elements of this kind in a floor plan, it is not significant.

A.3 User interface design

A.3.1 Screen design

As a result of the previous stage, all the user and application tasks have been defined, and the interaction architecture has been designed as well. The next step consists of designing the user interface itself, as we know every possible interaction path between the user and the application. The following rules may be applied to design each screen:

- In general, the main window will contain controls to launch all the tasks from the first level of the HTA. Each secondary screen, emergent dialog, internal frame or tabbed panel will contain some controls to execute subtasks.
- The non-allowed execution paths are avoided by disabling or enabling the controls from the user interface. Therefore, this responsibility does not correspond to the user.

Figure A.4 shows the main screen of the application. This screen is divided into several areas according to the HTA. The main toolbar contains buttons

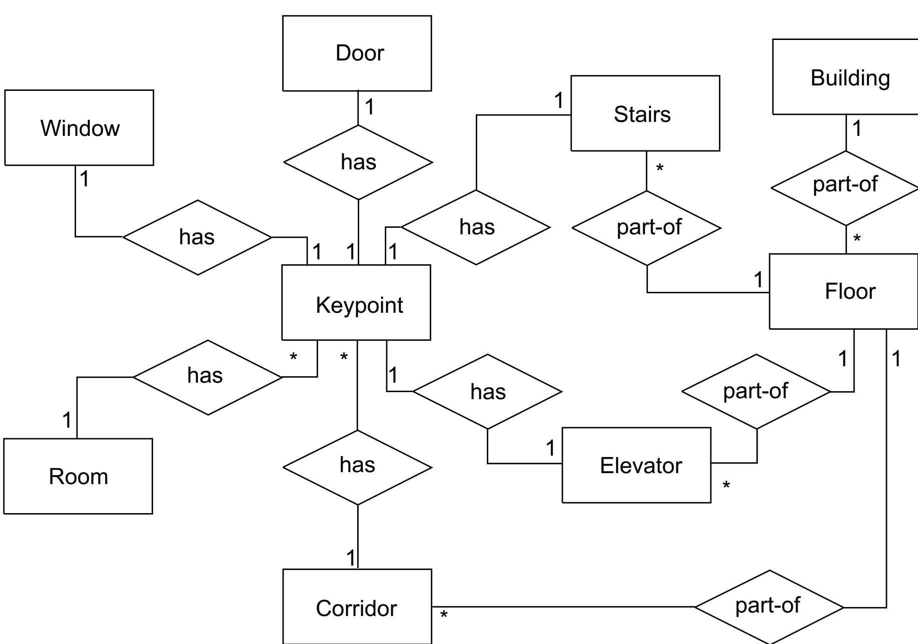


Figure A.3: Simplified view of the structure of the database

for launching most of the tasks from the first level of hierarchy, except for task 5 (edit project), which is subdivided and launched with other controls. Task 5.1 (load DXF file) can also be launched from the main toolbar. Tasks 5.2, 5.5 and 5.9 are launched using tabbed panes situated on the right side of the main screen. Tasks 5.6, 5.7 and 5.8 are not launched using controls, but interacting with the canvas using a pointing device. Task 5.10 is executed using buttons from an internal frame. Finally, there exists a standard menu bar that organizes some of the previously mentioned tasks using typical menus: file, edit, tools and view.

A.3.2 Model design

In this stage, we are going to use the Unified Modeling Language (UML)[Fow04] to represent the system model. First of all, we will discuss about the convenience of using some design patterns Then, a UML class diagram for the whole application will be introduced.

There are some design principles that must be taken into account:

- The application components must be properly delimited to avoid coupling and make the implementation process easier.
- Future modifications on the application must not cause modifications on existing components. They must be solved by adding new components

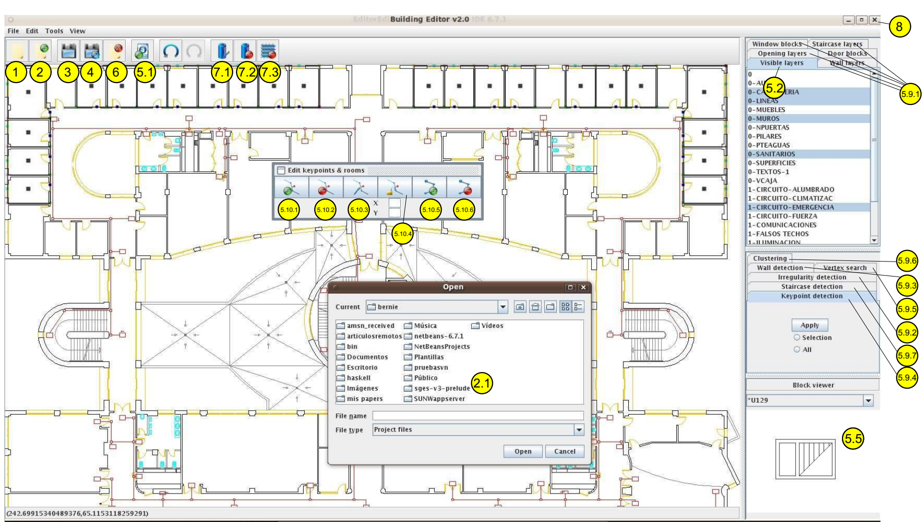


Figure A.4: Application main screen. The numbers in yellow circles show where the tasks from the HTA are launched from.

that fit into the existing architecture.

A.3.3 Design patterns

To accomplish these principles, some well-known design patterns[GHJV94, FFSB04] will be used. These are: *model-view-controller architecture* (MVC), *command pattern*, *strategy pattern*, *state pattern* and *observer pattern*. They are briefly explained below, describing how they are applied in the application.

A.3.3.1 Model-view-controller pattern

The model-view-controller architecture is used to divide the application into three parts: one of them (the model) deals with data and its processing; the second one (the view) deals with the user interface and the user-machine interaction. The controller deals with the information exchange between the model and the view. Some features about the control flow are the following:

- The view captures every user action on the interface and notifies the controller.
- The controller decides how to process the action. Then, it updates the model if necessary.
- The view has read-access to the model, so it can update itself. Any change on the model is notified to the view using the observer pattern (see below).

A.3.3.2 Observer pattern

As described at the MVC architecture section (A.3.3.1), the view must be reported about changes in the model. It would not be efficient to make the view ask asynchronously about changes in the model. On the other hand, the model must not know anything about the view existence, so it is not a good idea for the model to invoke methods from the view. Instead, we use the observer pattern.

The main roles of this pattern are (1) the observer, which waits to be notified about any kind of event; and (2) the subject, the class to be observed. Each observer registers itself in as many subjects as necessary. Each subject keeps a list of registered observers (but does not know anything about them), and notify them when a event occurs. Each observer can respond in a different way to this notification. In the MVC example, the view registers itself as an observer of the model. The model notifies its observers when something happens and the view updates itself in a proper way.

A.3.3.3 Strategy pattern

The strategy pattern allows to swap between different implementations of an algorithm or a class. For each algorithm or class for which different implementations are possible, there must be a strategy class. The specific implementations are subclasses of the strategy. There exists a *context* object, which decides which implementation has to be used. In this application, the strategy pattern deals with two different situations:

- Model, view and controller are strategies. For the model, this allows us to distinguish between behavior (set of operations supported by the model) and internal data representation. In the case of the view, this pattern allows to swap different user interfaces, no matter which model is used. The controller implementation must be coherent with the ones for model and view objects.

- The strategy pattern is very useful to test different algorithms for the same problem easily, like the wall and key point detection algorithms.

A.3.3.4 Command pattern

As described in the requirements section, the application must allow to undo/redo some user actions to make the application more flexible. To achieve this goal, the command pattern is used. Each order has to be identified, defining how it is done, undone and redone, and determining what information needs to be stored for these tasks.

The command pattern makes this issue easier:

- There is an abstract class which models the command behavior. It contains the *do* and *undo* methods.

- Each different command is encapsulated into a subclass of the above one that contains the parameters needed to redo the command.
- A command stack keeps track of the executed, undone and redone commands. It allows to undo the last executed command, and redo the last undone command.

A.3.3.5 State pattern

This pattern is used when an object behavior depends on its state. Its main goal is to avoid coupling between classes, i.e. a method of an object should not decide what to do by means of querying other objects, as it does not have to know anything about their existence. Instead of it, an object has different internal states which determine its behavior, and its current state can be changed by a context class. Therefore this object does not have to know anything about the objects that make changes to its state.

In this application, the state pattern is used for two different situations:

- As described in Section A.2.3, the architecture of operations on the projects (new, open, close, etc.) is based on the application state (project opened/not opened, project with/without name, project saved/not saved). We use the state pattern to implement this feature.
- The interaction with the canvas can be complex. When a user clicks on the canvas or drags the pointing device over it, the application response depends on its state. Clicking may be used to select an element or to insert a new one. Dragging may be used to select an area or to pan the camera.

A.3.4 UML class diagram

In this section, we introduce some class diagrams from the design stage. They are simplified versions of the whole UML diagram.

A.3.4.1 MVC and observer pattern diagrams

The first diagram (see Figure A.5) shows the MVC architecture of the application, and the observer and strategy patterns. The main ideas are:

- Model, view and controller classes inherit from model, view and controller interfaces (strategy pattern). New implementations of these classes could be added to try different possibilities. Using the strategy pattern, the rest of the system does not have to be changed to use the new implementations.
- Some observers are used to deal with asynchronous events. For each kind of observer there exists an interface, which has to be implemented by every observer class. Each observer interface defines only one method, which is implemented by every observer (there could be several observers for the

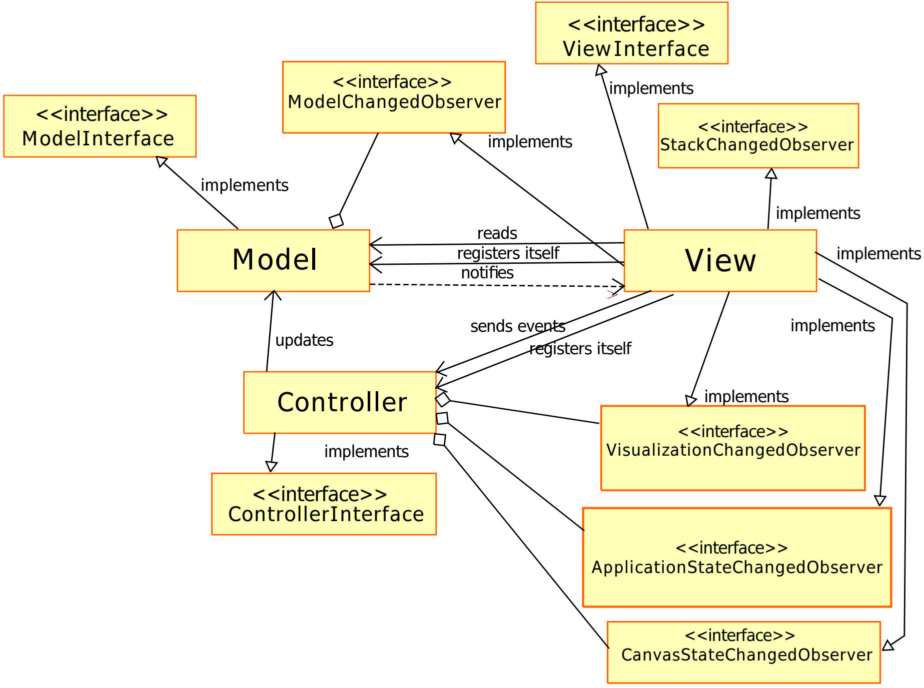


Figure A.5: MVC. Observer pattern

same kind of event, defining each of them its own behavior). The subject calls this method to notify the observers.

- *ModelChangedObserver* is used by the model (subject) to notify the view (observer) when it has been changed by the controller. Then the view reads the model and updates the interface.
- *VisualizationChangedObserver* is similar to *ModelChangedObserver*. The controller (subject) notifies the view (observer) when the user asks for some features to be shown or hidden from the canvas. Then the view updates the interface. *ModelChangedObserver* is used when the model changes (see Figure A.5), while *VisualizationChangedObserer* is used when the shown components change.
- *ApplicationStateChangedObserver* and *CanvasStateChangedObserver* are used by the controller (subject) to notify the view about the change of any of these states (state pattern). Then the controller adjusts enabled and disabled interface controls and stores the current state.
- *StackChangedObserver* is described in Section A.3.4.2.

- Regarding the MVC architecture, the view sends to the controller events received from the user through the application interface. Then, the con-

troller updates the model if neccesary, and the model notifies the change to the view (*ModelChangedObserver*). Initially, the view registers itself as *ModelChangedObserver* in the model, as *VisualizationChangedObserver*, *CanvasStateChangedObserver* and *ApplicationStateChangedObserver* in the controller, and as *StackChangedObserver* in the command stack (see Section A.3.4.2).

A.3.4.2 Command pattern diagram

The second diagram (see Figure A.6) shows the class architecture for the command pattern, using just one command (*LoadDXFCommand*) as example.

- The main class of the command pattern is *UndoStack*, a singleton (a class with a unique instance) which allows to add new commands, undo and redo commands, and query whether the stack is empty.
- Every command has an invoker (the view), a receiver (the model) and a client (the controller). The view is a *StackChangedObserver* of the stack, in order to update the interface to show which commands can be done or undone, if any.
- Every command inherits from *AbstractCommand*, an abstract class which defines a method to execute the command.

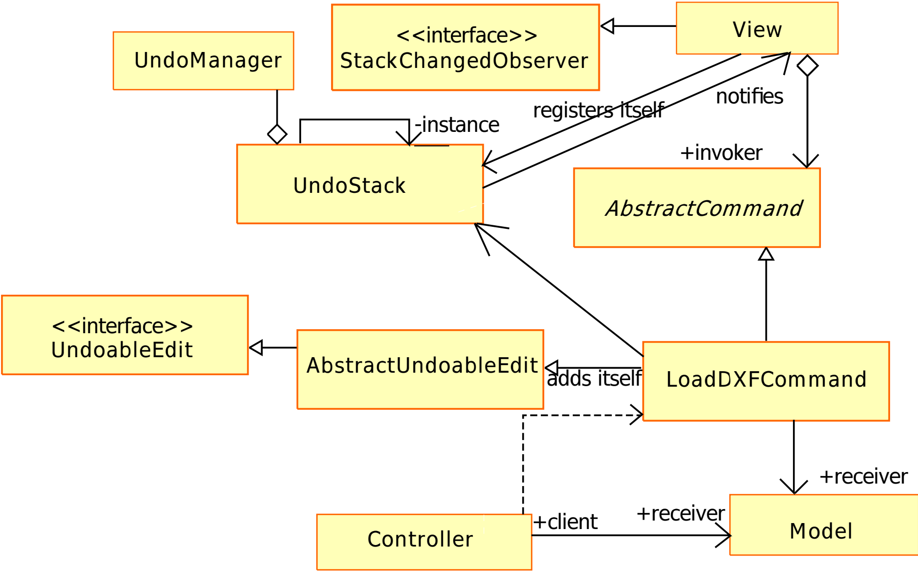


Figure A.6: Command pattern

A.3.4.3 Model architecture diagram

The third diagram (see Figure A.7) contains the model architecture, i.e. the logical representation of the data managed by the application. The class *Project* is the main class of the model. An object of the *Project* class represents an open project, editable by the user. A project has three sub-models, represented by grey rectangles:

- The *DXFModel* class represents the floor plan loaded from a DXF file. It contains lists for layers and blocks. Both lists contain the entities of the floor plan. *Line*, *Polyline*, *Arc* and *Insert* are subclasses of *Entity*. All of them contain 2D points (class *Point2D*).
- *GeometricModel* represents the floor plan in terms of colored lines. It is computed from a *DXFModel* object when it is created. To get a *GeometricModel* from a *DXFModel* these steps are followed: (1) the lines are included without modifications, (2) the arcs are discretized, (3) the polylines are split and (4) the inserts are transformed (scaled, rotated and translated), and their inner components are recursively computed.
- *SemanticModel* contains the detection results, i.e. rooms made by key points, windows, doors and staircases.

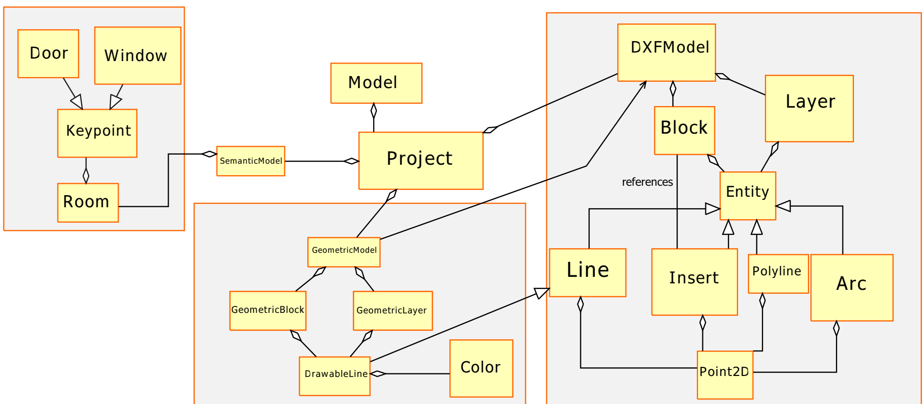


Figure A.7: Model

A.3.4.4 State pattern diagram

The last diagram (see Figure A.8) shows the implementation of the state pattern. An interface called *ApplicationState* defines methods for every possible action. Each state implements this interface, and its behavior and the following state depend on the methods implementation. The *Controller* deals with the current and previous state, and notifies the view (see Figure A.5) when the state has changed.

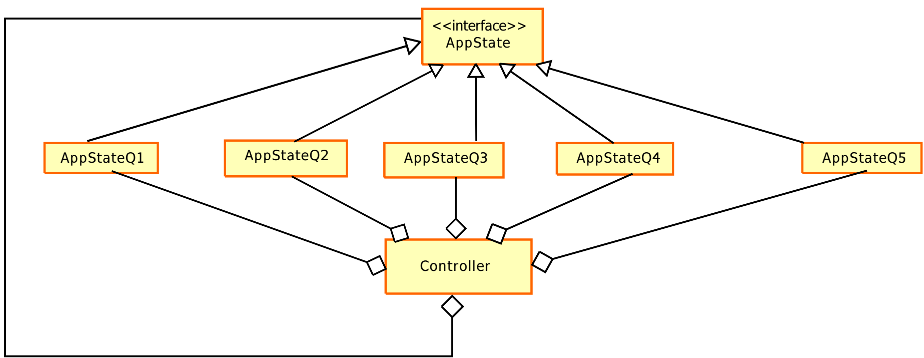


Figure A.8: Application state diagram

A.4 Implementation

We have given our software tool the name MOSES (MOdeler of building SEmanticS). It has been implemented using the Java programming language, therefore it is portable and independent from the hardware and software platforms. Additional third party libraries that were used include *Kabeja* [kab], which provides parsing of DXF files; *jGraphT* [jGr], that provides mathematical graph-theory objects and algorithms; and *citygml4j* [cit], to work with CityGML.

Finally, Carve [car13] computes CSG operations on B-rep solids. As it is implemented in C++, it is necessary to export the solids to Stanford PLY mesh file format using Java. Carve is able to load PLY solids, compute CSG operations and write the result back to a PLY file.

Now we present the main challenges found during the software implementation process and the way to deal with them:

- Regarding the detection and fixing of irregularities, the main problem is that it can provoke the appearance of new irregularities. Thus, it is not easy to predict how many iterations are needed to remove all the irregularities.
- The main difficulty of the topology detection is the numerical instability in the topology graph representation. This makes some vertices appear duplicated with slightly different position values. Some algorithms on graphs, like finding connected components or closed spaces, may fail due to the existence of these duplicated vertices. Specifically, the existence of duplicated edges associated with different vertices must be previously detected to avoid infinite loops in the closed spaces detection. The irregularities detection and fixing phase solves (most of) these problems.
- Choosing the right threshold is one of the hardest issues of the graph algorithms. Finally, we use two thresholds: maximum distance to consider two vertices the same and maximum angle to consider two edges parallel.

Appendix B

Evolution of clustering algorithms

This appendix describes the evolution of the clustering algorithms, in the chronological order that were considered for solving the problem of finding the joint points between graph edges representing walls and openings. For each algorithm, we explain its disadvantages, in order to introduce the following algorithm.

B.1 K-means

First, a family of clustering algorithms is based on the calculation of the centroid. They are better known as k-means. These algorithms start by choosing (by some criteria) k centroids from the point set. The algorithm runs while there are changes in the clusters. In each iteration, each point is assigned to the cluster with the closest centroid, and finally all the centroids are recalculated (Algorithm B.1).

Algorithm B.1: K-means algorithm

Input: A set of points S, and a value k (number of clusters)
Output: A set of clusters

```
1 begin
2     Choose k points from S as the centroids
3     while there are changes in the clusters do
4         Assign each point from S to the cluster with the closest centroid
5         Recompute all the centroids
6     end
7 end
```

The main disadvantage of this philosophy for our problem is that the k-means algorithm needs the number k of clusters prior to run. In our problem,

it is not possible to know this parameter.

Second, there is a clustering algorithm family based on the distribution of the points. In general terms, these algorithms try to fit the point set to a previously known probability distribution. To our purposes, the point set is too reduced to be fit to a concrete probability distribution. Furthermore, the points represent the spatial distribution of the wall end points, thus it is not feasible to assume that they follow probability laws. Consequently, this approach is also discarded to solve the clustering problem.

Third, we find a family of clustering algorithms based on density. Although they are not deeply explained in this work, their philosophy involves finding clusters which satisfy a density criterion. They do not require to know the number of clusters in advance. They start from the whole point set and two parameters: the minimum distance between two points to be included in the same cluster, and the minimum number of neighbor points to start a new cluster.

B.2 DBSCAN algorithm

Each point is initially tagged as *unvisited*. For each unvisited point from the set, the algorithm computes a point neighborhood formed by its closest points according to a distance measurement (usually euclidean) and a distance threshold. If the computed neighborhood has less points than the threshold of points to start a cluster, the point is tagged as *noise*. Otherwise, a new cluster is created and filled with the computed neighborhood. The same process is repeated over the unvisited points from the neighborhood until no new points can be added to the cluster.

Algorithm B.2 shows the pseudo-code of this method, called *dbscan*:

Some auxiliary functions of `dbscan` are described below (`reqionQuery`, Algorithm B.3 and `expandCluster`, Algorithm B.4).

The clustering algorithms based on density have as their main advantage that it is not necessary to know the number of clusters in advance, and they do not assume any probability distribution. They work correctly with scenarios determined by sets made of the end points of the wall edges. Nevertheless, the low size of the point set from walls makes it difficult to apply the original *dbscan* algorithm. In real cases, if the threshold is properly chosen, each cluster would not have more than five or six points (coming from five or six adjacent walls at the most). In this context it is difficult to fix a minimum number of points to create a new cluster.

The researched algorithms are based on the dbscan philosophy, though we disregard the minimum number of points needed not to consider a point as noise and start a new cluster (this is the same that applying dbscan with this parameter set to one).

Algorithm B.2: DBSCAN clustering

```
   Input: P: the point set; ε: the threshold to include two points in the
          same cluster; minPoints: the minimum number of points to start
          a new cluster
   Output: clusters: the resulting set of point clusters
 1 begin
 2     Mark all the points as not visited
 3     clusters ← ∅
 4     foreach point not visited point p in P do
 5         mark p as visited
 6         neighbors ← reqionQuery(p, ε)
 7         if length of neighbors < minPoints then
 8             mark p as noise
 9         else
10             start a new cluster C
11             C ← expandCluster(p, neighbors, ε, minPoints)
12             clusters ← clusters ∪ C
13         end
14     end
15     return clusters
16 end
```

Algorithm B.3: Region query for DBSCAN clustering algorithm

```
   Input: p: a point; ε: the threshold to include two points in the same
          cluster;
   Output: a set including the neighbors of p
 1 begin
 2     return All the points whose distance to p is less than ε
 3 end
```

Algorithm B.4: Expand cluster for DBSCAN clustering algorithm

```
   Input: p: the point; neighbors: the neighbors of point p; ε: the threshold
          to include two points in the same cluster; minPoints: the
          minimum number of points to start a new cluster
   Output: The result of expanding the cluster that contains p
 1 begin
 2     Add p to the cluster C
 3     foreach not visited point p' in neighbors do
 4         mark p' as visited
 5         neighbors' ← reqionQuery(p',ε)
 6         if length of neighbors' ≥ minPoints then
 7             add all the points from neighbors' to neighbors
 8         end
 9         if p' is not in the current cluster then
10             add p' to the cluster C
11         end
12     end
13     return C
14 end
```

B.3 Brute force algorithm

The first algorithm we introduce, called *brute force* algorithm, is based on the K-means stop criterion: while there are changes in the clusters, a new iteration starts. As input, the algorithm takes the topology graph whose points have to be clustered, and a distance threshold for merging two clusters. Initially, each point is assigned to a cluster. For each iteration, the algorithm makes a double loop on the clusters in order to compare all the possible pairs of clusters. Two clusters are merged if *all* the points from a cluster are close enough (according to the threshold) to *all* the points from the other cluster. In case two clusters are merged, the double loop is interrupted and a new iteration starts. Once all the clusters have been obtained, each point from the topology graph is replaced by a representative point, maintaining the edge coherence. In this version of the algorithm, the representative point of a cluster is computed as its centroid. Algorithm B.5 shows the pseudo code of this method.

The tests with this method provide good results but the algorithm running time is high. Each time two clusters are merged, the algorithm starts a new iteration. This makes the first iterations be interrupted after a few comparisons, since the cluster set is still unstable. While the algorithm advances, the clusters become more stable. Furthermore, estimating the running time is not straightforward since it depends on the order the points are visited and how the point set distribution is.

Additionally, for each pair of clusters, all their points are compared to each other. If any comparison returns more distance than the threshold, the two

Algorithm B.5: Brute force clustering algorithm

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
 1 begin
 2     Initialize a vertex set clusters for each vertex in V
 3     Initialize an empty vertex set toDelete
 4     merged ← true
 5     while merged = true do
 6         merged ← false
 7         for i = 0 ; i < clusters.size and merged = false; i + + do
 8             for j = i + 1; j < clusters.size and merged = false; j + + do
 9                 if all the points in clusters[i] are closer than ε to all the
                   points in clusters[j] then
10                     clusters[i] ← clusters[i] ∪ clusters[j]
11                     toDelete ← toDelete ∪ {clusters[j]}
12                     merged ← true
13                 end
14             end
15         end
16         clusters ← clusters\toDelete
17     end
18     V' is built from V by replacing every vertex by the centroid of its
       cluster
19     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its cluster
20     return G'=(V',E')
21 end
```

involved clusters are discarded for merging.

B.4 Variation on brute force algorithm

The first variation on the algorithm involves a symmetric version of the comparison between clusters. The algorithm is similar, except that two clusters are merged if *at least* one point of each cluster is closer than the threshold to *at least* one point from the other one (pseudo code in Algorithm B.6). This condition is less strict than the previous, in the sense that the point distribution of each cluster can be more disperse.

Algorithm B.6: Second version of brute force clustering algorithm

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
 1 begin
 2     Initialize a vertex set clusters for each vertex in V
 3     Initialize an empty vertex set toDelete
 4     merged ← true
 5     while merged = true do
 6         merged ← false
 7         for i = 0 ; i < clusters.size and merged = false; i++ do
 8             for j = i + 1; j < clusters.size and merged = false; j++ do
 9                 if at least one vertex in clusters[i] is closer than ε to at
                   least one vertex in clusters[j] then
10                     clusters[i] ← clusters[i] ∪ clusters[j]
11                     toDelete ← toDelete ∪ {clusters[j]}
12                     merged ← true
13                 end
14             end
15         end
16         clusters ← clusters\toDelete
17     end
18     V' is built from V by replacing every vertex by the centroid of its
       cluster
19     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its cluster
20     return G'=(V',E')
21 end
```

The running time of Algorithm B.6 does not substantially vary with respect to the first version of the *brute force* algorithm. Which algorithm is faster depends once again on the distribution of the points and it is difficult to determine the execution time prior to the execution.

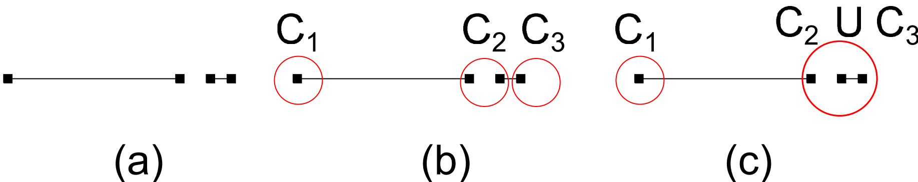


Figure B.1: Misassignment of two end points from the same wall segment to the same cluster. (a) Two edges from the topology graph representing walls; (b) Desired result of the clustering; (c) Incorrect clustering: two end points from the same wall edge have been assigned to the same cluster.

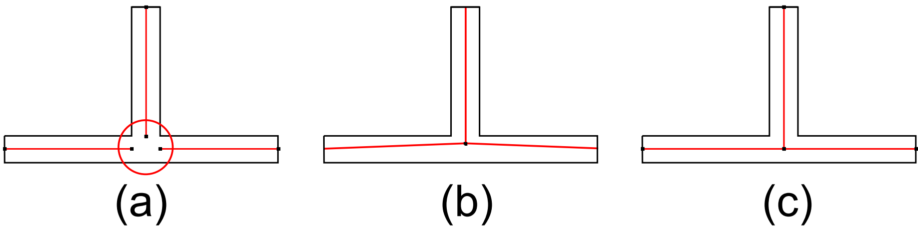


Figure B.2: (a) The three end points of the wall make up a cluster; (b) The centroid of the cluster makes the edges to lose their alignment; (c) A representative point of the centroid that does not disturb the alignment.

Apart from the high running time, we found some wrong results in special scenarios with both algorithms. If any wall segment is shorter than the used distance threshold, then both vertices of the wall are merged in the same cluster. This situation makes no sense from a topological scope, since two points from the same wall are never to be joint. Furthermore, the wall disappears, as shown in Figure B.1.

Due to this fact, preventing two points from the same wall to be merged into the same cluster is added to the next algorithm's variation. This additional condition is added to the comparison between two points.

Another flaw of the algorithm is the way the representative point of each cluster is computed. In the most usual cases, walls are orthogonal, and choosing the centroid as the representative point provokes that some edges that should be aligned become oblique, as shown in Figure B.2.

B.5 Smart centroid detection

In order to improve this, the assignment of representative points considers a set of standard cases, summarized in Table B.1. When the walls involved in the representative point computation are orthogonal, the algorithm analyzes the number of adjacent edges and its layout. If the layout matches one of the four first rows on Table B.1, the representative point is computed as appears in the

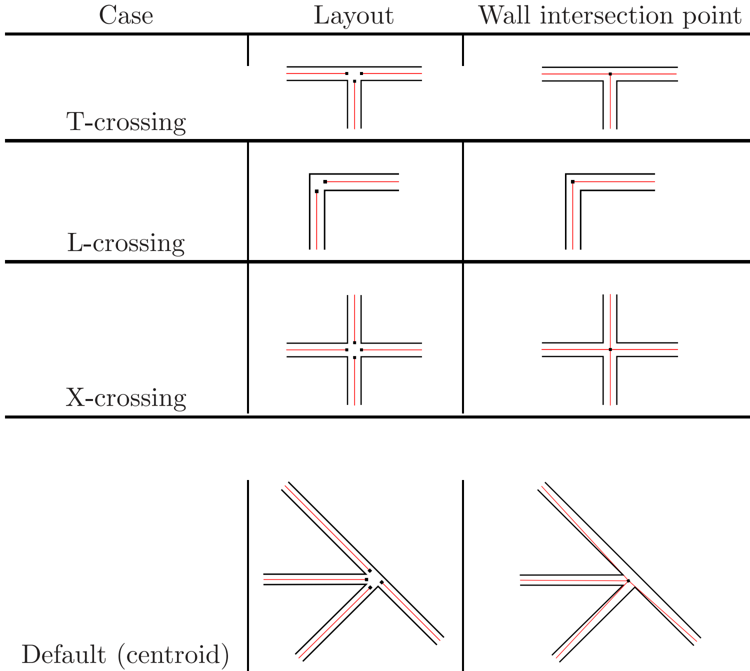


Table B.1: Intersection cases and solutions.

right column. Algorithm B.8 computes the representative point according to the cases on Table B.1. The X-crossing case does not appear explicitly in the pseudo code since this case is correctly computed as the centroid. Finally, the centroid is applied if the edge layout does not match any of the standard cases.

B.6 Clustering avoiding same edge

The pseudo code in Algorithm B.7 describes the two new issues above described. The clustering algorithm is similar to the introduced in Algorithm B.6, but adding an additional constraint: two clusters are never merged if they contain the two end points from the same wall edge. Once the clusters have been computed, the representative points are computed using the function `SmartCentroid` to assign a representative point to each cluster, instead of computing the centroids.

Algorithms B.6, B.7 and B.8 form the first approach designed to solve the search for wall joint points. Although we have cited some advantages and disadvantages of each algorithm, they are summarized as follows. Among the advantages:

- The clustering algorithms are based on point density. As a consequence, they accomplish the initial goal: they detect the areas where the walls are

Algorithm B.7: Clustering algorithm avoiding points from the same edge to be in the same cluster

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
 1 begin
 2     Initialize a vertex set clusters for each vertex in V
 3     merged ← true
 4     while merged = true do
 5         merged ← false
 6         for i = 0 ; i < clusters.size and merged = false; i + + do
 7             for j = i + 1; j < clusters.size and merged = false; j + + do
 8                 if at least one vertex in clusters[i] is closer than ε to at
                   least one vertex in clusters[j], and they do not belong to the
                   same edge then
 9                     clusters[i] ← clusters[i] ∪ clusters[j]
10                     Remove clusters[j] from clusters
11                     merged ← true
12                 end
13             end
14         end
15     end
       // smart centroid
16     foreach clusters[i] in clusters do
17         centroids[i] ← SmartCentroid(clusters[i])
18     end
19     V' is built from V by replacing every vertex by the smart centroid of
       its cluster
20     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its cluster
21     return G'=(V',E')
22 end
```

Algorithm B.8: Computation of the smart centroid in a clustering algorithm

```
   Input: cluster: the cluster of vertices
   Output: the coordinates of the smart centroid
 1 begin
 2     n ← cluster.size
 3     for i = 0 to n-1 do
 4         P_i ← cluster[i]
 5         Q_i ← Vertex adjacent to P_i in the graph
 6         s_i ← Line(P_i,Q_i)
 7         v_i ← P_i − Q_i
 8     end
 9     if n = 3 then
10         if Aligned(s_0,s_1) and v_2 · v_0 = 0 then return Project(P_2,s_0)
11         if Aligned(s_2,s_0) and v_1 · v_2 = 0 then return Project(P_1,s_2)
12         if Aligned(s_1,s_2) and v_0 · v_1 = 0 then return Project(P_0,s_1)
13         return (P_0 + P_1 + P_2)/3
14     else if n = 2 then
15         if v_0 · v_1 = 0 then return Project(P_1,s_0)
16         return (P_0 + P_1)/2
17     return (Σ_{i=0}^{n−1} P_i)/n
18 end
```

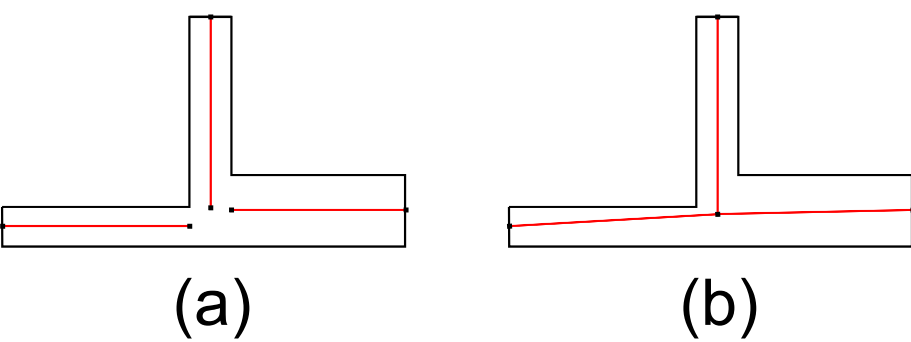


Figure B.3: (a) A layout with three walls. The two horizontal walls have different thickness. This situation does not match any standard case; (b) as a result, the representative point is computed as the centroid, thus making the edges to be oblique

supposed to meet.

- In floor plans (or parts of them) where the following conditions are fulfilled, the result is free of errors: (1) The distance among the wall end points is always lower than the wall length, in order to avoid the incorrect merging of clusters; (2) there are not many different values for the wall thickness, in order to avoid the existence of non aligned walls (as shown in Figure B.3) and (3) the walls are orthogonal, to match the T-shape, L-shape and X-shape cases.

There are features of these algorithms that need to be improved:

- It is difficult to find floor plans that fulfill the three conditions. Therefore, we need to research on strategies to deal with these not so exceptional situations.

- The running time can be reduced. On one side, the *brute force* approach does not consider any spatial optimization of the point set. Many comparison operations can be avoided if the points are filtered by proximity by using any kind of spatial index. On the other side, as explained above, the stop criterion needs many iterations to be reached, given that any new cluster merge makes the algorithm to start a new iteration.

- The information about edges is not fully considered (except the constraint of avoiding to merge both end points from the same edge). Only the position of the point is considered. The characteristics of the edges may provide the algorithms with useful information

After this analysis, the first step was to use more information about the edges instead of consider only the points as the input. Functions `ClusteringCompareEdges`,

`ClusteringCompareSequences` and `ClusteringCompareSequencesSimple` introduce three new approaches based on this. Their main features as are follows:

- The stop criterion is the same: the algorithm stops when there are not newer changes in the clusters.
- Now, two different cluster sets are kept: point clusters and edge clusters. Each point cluster still has the same function: grouping all the points that are going to be replaced by a representative point, while each edge cluster is used to cluster the edges whose points have been merged in the same point cluster. The edge cluster set is used to control the iterations and which points can be merged in the same cluster.

B.7 Clustering comparing edges

The pseudo code of the first algorithm of this approach can be viewed in Function `ClusteringCompareEdges` (B.9). The algorithm runs while there are changes in the edge cluster set. For each pair of edge clusters, all the pairs of edges made by taking one edge of each cluster are analyzed in order to search for a pair of edges having some of their end points closer than a threshold. If this condition is accomplished, then both edge clusters are merged, and both point clusters containing the closest points from these edges are merged as well.

Once the edge cluster set does not have further changes, a representative point is computed for each point cluster by using the function `SmartCentroid` already explained. Two edges from the same cluster can never be merged: once two points are merged in a cluster, the two edges containing them are no longer compared during the algorithm. This dramatically reduces the number of point comparisons with respect to the previous algorithms. In addition, the fact that two points from the same edge are never compared automatically fulfills the constraint set for Algorithm `ClusteringSameEdge` (B.7).

B.8 Clustering comparing sequences

A similar idea, but this time applied to polylines is introduced in algorithm B.10. Therefore, the algorithm requires a preprocessing for finding polylines. This is solved by searching for connected components from the topology graph (Algorithm B.10).

After testing Algorithm B.10, we observed that the stop criterion was almost superfluous, given that when two polylines from a pair are compared and some of their end points are closer than a threshold, those polylines are merged in the same cluster and are no longer compared during the algorithm. In practice, this is similar to visiting each pair of polylines only once. Consequently, the stop criterion about changes in the cluster can be removed. The new stop criterion is finishing the analysis of every pair of polylines, i.e. use a double loop over the polyline set.

Algorithm B.9: Clustering algorithm comparing edges

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
 1 begin
 2     Initialize a vertex cluster vertexClusters[i] for each vertex in V
 3     Initialize an edge cluster edgeClusters[i] for each edge in E
 4     merged ← true
 5     while merged = true do
 6         merged ← false
 7         for i = 0; i < edgeClusters.size and merged = false; i++ do
 8             for j = i+1; j < edgeClusters.size and merged = false; j++ do
 9                 close ← false
10                 for k = 0; k < edgeClusters[i].size and close = false; k++
                   do
11                     for l = 0; l < edgeClusters[j].size and close = false; l++
                       do
12                         (P_min,Q_min) ← Closest points among all the pairs
                           {(P_i, Q_i), P_i ∈ edgeClusters[i][k], Q_i ∈
                           edgeClusters[j][l]}
13                         min ← d(P_min,Q_min)
14                         if min < ε then
15                             close ← true
16                         end
17                     end
18                 end
19                 if close = true then
20                     edgeClusters[i] ← edgeClusters[i] ∪ edgeClusters[j]
21                     Remove edgeClusters[j] from edgeClusters
22                     merged ← true
23                     Merge the vertex clusters that contain P_min and Q_min
24                 end
25             end
26         end
27     end
       // smart centroid
28     foreach vertexClusters[i] in vertexClusters do
29         centroids[i] ← SmartCentroid(vertexClusters[i])
30     end
31     V' is built from V by replacing every vertex by the smart centroid of
       its cluster
32     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its vertex cluster
33     return G'=(V',E')
34 end
```

Algorithm B.10: Clustering algorithm comparing connected components

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
 1 begin
 2     Initialize a vertex cluster vertexClusters[i] for each vertex in V
 3     Initialize a sequence cluster sequenceClusters[i] for each connected
       component in E
 4     merged ← true
 5     while merged = true do
 6         merged ← false
 7         for i = 0; i < sequenceClusters.size and merged = false; i + + do
 8             for j = i + 1; j < sequenceClusters.size and
               merged = false; j + + do
 9                 close ← false
10                 for k = 0; k < sequenceClusters[i].size and
                   close = false; k + + do
11                     for l = 0; l < sequenceClusters[j].size and
                       close = false; l + + do
12                         (P_min,Q_min) ← Closest points among all the pairs
                           {(P_i, Q_i), P_i ∈ sequenceClusters[i][k], Q_i ∈
                           sequenceClusters[j][l]}
13                         min ← d(P_min,Q_min)
14                         if min < ε then
15                             close ← true
16                         end
17                     end
18                 end
19                 if close = true then
20                     sequenceClusters[i] ← sequenceClusters[i] ∪
                       sequenceClusters[j]
21                     Remove sequenceClusters[j] from sequenceClusters
22                     merged ← true
23                     Merge the vertex clusters that contain P_min and Q_min
24                 end
25             end
26         end
27     end
       // smart centroid
28     foreach vertexClusters[i] in vertexClusters do
29         centroids[i] ← SmartCentroid(vertexClusters[i])
30     end
31     V' is built from V by replacing every vertex by the smart centroid of
       its vertex cluster
32     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its vertex cluster
33     return G'=(V',E')
34 end
```

Another change on Algorithm B.10 is not stop analyzing a pair of clusters when the first pair of close polylines has been found. Two clusters can be close enough in more than one location. Therefore, both cluster should be merged using the closest pair of polylines. The next improvement consists of comparing all the pair of polylines for each pair of clusters and selecting the closest ones.

B.9 Variation on clustering comparing sequences

The pseudo code of the function `ClusteringCompareSequencesSimple`, which includes these two improvements, can be viewed in Algorithm B.11.

Algorithm B.11: Second version of clustering algorithm comparing connected components

```
   Input: G=(V,E): the graph containing the wall and opening edges
   Output: G'=(V',E'): the graph G=(V,E) after the vertex clustering
1  begin
2      Initialize a vertex cluster vertexClusters[i] for each vertex in V
3      Initialize a sequence cluster sequenceClusters[i] for each connected
       component in E
4      for i = 0; i < sequenceClusters.size; i + + do
5          for j = i + 1; j < sequenceClusters.size; j + + do
6              for k = 0; k < sequenceClusters[i].size; k + + do
7                  for l = 0; l < sequenceClusters[j].size; l + + do
8                      (P_min,Q_min) ← Closest points among all the pairs
                       {(P_i, Q_i), P_i ∈ sequenceClusters[i][k], Q_i ∈
                       sequenceClusters[j][l]}
9                      min ← d(P_min,Q_min)
10                 end
11             end
12             if min < ε then
13                 sequenceClusters[i] ← sequenceClusters[i] ∪
                   sequenceClusters[j]
14                 Remove sequenceClusters[j] from sequenceClusters
15                 merged ← true
16                 Merge the vertex clusters that contain P_min and Q_min
17             end
18         end
19     end
       // smart centroid
20     foreach vertexClusters[i] in vertexClusters do
21         centroids[i] ← SmartCentroid(vertexClusters[i])
22     end
23     V' is built from V by replacing every vertex by the smart centroid of
       its vertex cluster
24     E' is built from E by assigning the edges adjacent to every vertex to
       the centroid of its vertex cluster
25     return G'=(V',E')
26 end
```

Appendix C

Algorithms from computational geometry used in this dissertation

This appendix describes the computational geometry algorithms used in this document. They are based on the Schneider and Eberly's book entitled *Geometric Tools for Computer Graphics* [SE02]. Most of the algorithms have been used as they appear in the book. Others have been modified to adapt them to the needs of our algorithms. Finally, some of them have been modified after realizing some erratas in the book. This erratas were successfully submitted to the authors and kindly considered for its publication in the corrections website[1].

C.1 Intersection between primitives

The first family of algorithms deals with the detection of whether two primitives (2D segments or arcs) intersect each other. Algorithm C.1 is the general description of this detection.

At the same time, this function calls `IntersectionSegments`, `IntersectionSegmentArc` or `IntersectionArcs` depending on which primitives have to be analyzed (two segments, a segment and an arc or two arcs).

The function `IntersectionSegments` detects if two segments intersect (Algorithm C.2) and is based on the function FindIntersection from the Schneider/Eberly's book [SE02], p.244, which detects the point of intersection among two segments.

`IntersectionSegments` calls the function `FindIntersection` (Algorithm C.3) in order two detect whether there are common points in two real intervals. This function is also based on [SE02], p.245. The only difference between the functions from the book and the function we introduce here is that the book

[1] http://www.geometrictools.com/Books/GeometricTools/BookCorrections.html

Algorithm C.1: Detection of the intersection between two primitives

```
   Input: i,j: the entities to be tested
   Output: true if i and j intersect, false otherwise
 1 begin
 2     if i and j are segments then return IntersectionSegments(i,j)
 3
 4     else if i is a segment and j is an arc then return
       IntersectionSegmentArc(i,j)
 5
 6     else if i is a segment and j is an arc then return
       IntersectionSegmentArc(j,i)
 7
 8     else return IntersectionArcs(i,j)
 9
10 end
```

version returns arrays containing the intersection points, while we are only interested in the intersection itself. Therefore our function returns a boolean.

On the other side, we do not follow the analytic approach proposed in [SE02] for detecting the intersection between a segment and an arc, or two arcs. As the book proposes, computing the intersection among a segment and an arc involves three steps ([SE02], pp.247-248):

1. Given the line containing the segment and the circle containing the arc, compute the solution/s after replacing the line equation into the circle equation.

2. Determine whether those solutions belong to the segment (the circle may intersect the line but not the segment)

3. Determine whether those solutions belong to the arc using a point-in-arc test (the segment may intersect the circle but not the arc).

Determining geometrically whether two arcs intersect consists of calculating the intersection/s between their containing circles ([SE02], pp.257-258) and make up to two point-in-arc tests like in the third step of the intersection segment-arc([SE02], p.248).

As it is only necessary to know whether there is an intersection, we have converted the arc-arc intersection calculation into a multiple segment-segment intersection calculation, by discretizing the circles, and working with the resulting segments.

Algorithm C.4 computes the intersection among a segment and an arc. The segment is represented by two points and the arc is represented by its center, radius, and start and end angles. The arc is discretized into 10 segments. For each segment the distance to the analyzed segments is computed, and the minimum

Algorithm C.2: Detection of the intersection between two segments

```
   Input: L0(P0,Q0),L1(P1,Q1): the segments to be tested
   Output: true if L0(P0,Q0) and L1(P1,Q1) intersect, false otherwise
 1 begin
 2     D0 ← Q0 − P0
 3     D1 ← Q1 − P1
 4     E ← P1 − P0
 5     kross = D0.x · D1.y + D0.y · D1.x
 6     sqrKross ← kross · kross
 7     sqrLen0 ← D0.x · D0.x + D0.y · D0.y
 8     sqrLen1 ← D1.x · D1.x + D1.y · D1.y
 9     if sqrKross > sqrEpsilon · sqrLen0 · sqrLen1 then
           // lines of segments are not parallel
10         s = E.x · D1.y − E.y · D1.x/kross
11         if s < 0 or s > 1 then
               // intersection of lines is not a point on
               // segment P0 + s · D0
12             return false
13         end
14         t = E.x · D0.y − E.y · D0.x/kross
15         if t < 0 or t > 1 then
               // intersection of lines is not a point on
               // segment P1 + t · D1
16             return false
17         end
           // intersection of lines is a point on each segment
18         return true
19     end
       // lines of the segments are parallel
20     sqrLenE ← E.x · E.x + E.y · E.y
21     kross ← E.x · D0.y − E.y · D0.x
22     sqrKross ← kross · kross
23     if sqrKross > sqrEpsilon · sqrLen0 · sqrLenE then
           // lines of the segments are different
24         return false
25     end
       // lines of the segments are the same
26     s0 ← Dot(D0,E)/sqrLen0
27     s1 ← s0 + Dot(D0,D1)
28     smin ← min(s0,s1)
29     smax ← max s0, s1
30     return FindIntersection(0.0,1.0,smin,smax)
31 end
```

Algorithm C.3: Detection of the intersection between two intervals

```
  Input: u0,u1,v0,v1: the limits of the two intervals
  Output: true if the intervals intersect, false otherwise
1 begin
2     if u1 < v0 or u0 > v1 then return false
3     return true
4 end
```

value is stored (line 11). The function d that computes the distance among segments will be explained later. Finally, the function returns **true** if the minimum distance among the segments is under a threshold ε. This threshold decides if a number equals zero.

Algorithm C.4: Detection of the intersection between a segment and an arc

```
   Input: L(P,Q),A(C,r,α,β): the segment and the arc to be compared
   ε: the threshold value to consider null the distance
   Output: true if L and A intersect, false otherwise
 1 begin
 2     min ← ∞
 3     n ← 10
 4     for i = 0 to n − 1 do
 5         γ1 ← (α · (n − i) + β · i)/n
 6         γ2 ← (α · (n − i − 1) + β · (i + 1))/n
 7         P1 ← (Cx + r · cos(α), Cy + r · sin(α))
 8         P2 ← (Cx + r · cos(β), Cy + r · sin(β))
 9         L2 ← Line(P1,P2)
10         if d(L,L2)<min then
11             min ← d(L,L2)
12         end
13     end
14     return min < ε
15 end
```

Similarly, for the arc-arc intersection test, both arcs are discretized using n =10 segments per arc. The 10 segments of one arc are compared with the 10 segments of the other arc. The function returns **true** if the resulting minimum distance is lower than ε, as can be shown in Algorithm C.5.

C.2 Distance between segments

The solution to this problem merely depends on the relative position among the segments and it is not straightforward to characterize. The used approach is

Algorithm C.5: Detection of the intersection between two arcs

Input: $A_1(C_1, r_1, \alpha_1, \beta_1), A_2(C_2, r_2, \alpha_2, \beta_2)$: the arcs to be compared
ε: the threshold value to consider null the distance
Output: **true** if A_1 and A_2 are intersect, **false** otherwise

1 **begin**
2 $\quad \mathsf{min} \leftarrow \infty$
3 $\quad \mathsf{n} \leftarrow 10$
4 $\quad$ **for** $\mathsf{i} = 0$ *to* $\mathsf{n} - 1$ **do**
5 $\quad\quad \gamma_{11} \leftarrow (\alpha_1 \cdot (\mathsf{n} - \mathsf{i}) + \beta_1 \cdot \mathsf{i})/\mathsf{n}$
6 $\quad\quad \gamma_{12} \leftarrow (\alpha_1 \cdot (\mathsf{n} - \mathsf{i} - 1) + \beta_1 \cdot (\mathsf{i} + 1))/\mathsf{n}$
7 $\quad\quad P_{11} \leftarrow (C_{1x} + r_1 \cdot \cos(\alpha_1), C_{1y} + r_1 \cdot \sin(\alpha_1))$
8 $\quad\quad P_{12} \leftarrow (C_{1x} + r_1 \cdot \cos(\beta_1), C_{1y} + r_1 \cdot \sin(\beta_1))$
9 $\quad\quad L_1 \leftarrow \texttt{Line}(P_{11}, P_{12})$
10 $\quad\quad$ **for** $\mathsf{j} = 0$ *to* $\mathsf{n} - 1$ **do**
11 $\quad\quad\quad \gamma_{21} \leftarrow (\alpha_2 \cdot (\mathsf{n} - \mathsf{j}) + \beta_2 \cdot \mathsf{j})/\mathsf{n}$
12 $\quad\quad\quad \gamma_{22} \leftarrow (\alpha_2 \cdot (\mathsf{n} - \mathsf{j} - 1) + \beta_2 \cdot (\mathsf{j} + 1))/\mathsf{n}$
13 $\quad\quad\quad P_{21} \leftarrow (C_{2x} + r_2 \cdot \cos(\alpha_2), C_{2y} + r_2 \cdot \sin(\alpha_2))$
14 $\quad\quad\quad P_{22} \leftarrow (C_{2x} + r_2 \cdot \cos(\beta_2), C_{2y} + r_2 \cdot \sin(\beta_2))$
15 $\quad\quad\quad L_2 \leftarrow \texttt{Line}(P_{21}, P_{22})$
16 $\quad\quad\quad$ **if** $\texttt{d}(L_1, L_2) < \mathsf{min}$ **then**
17 $\quad\quad\quad\quad \mathsf{min} \leftarrow \texttt{d}(L_1, L_2)$
18 $\quad\quad\quad$ **end**
19 $\quad\quad$ **end**
20 $\quad$ **end**
21 $\quad$ **return** $\mathsf{min} < \varepsilon$
22 **end**

based again on the Schneider/Eberly's book [SE02], pp.228-229.

Firstly, the implemented algorithm from the book did not get the expected results. Thus, the development of the formulas from the book was revised. After this revision, we found two erratas. One of them was communicated to the authors in January 2013 and its correction was published in the web of book corrections on March 1, 2013. The other errata was notified to the authors in October 2013 and published in the web of book corrections on October 7, 2013.

For simplicity, the development carried out to get the distance formulas are not fully described here, but we provide a summary having as goal to make the formulas easier to understand and explain the found erratas.

For the study of the distance among segments, we start from their parametric equation.

Let the segments be $P_i + t_i \overrightarrow{d}_i$ for $i = 0, 1$ and $t_i \in [0, 1]$. Define $\overrightarrow{\Delta} = P_0 - P_1$. The squared distance between any points $P_0 + t_0 \overrightarrow{d}_0$ and $P_1 + t_1 \overrightarrow{d}_1$ can be expressed as the two-variable function F:

$$F(t_0, t_1) = ||t_0 \overrightarrow{d}_0 - t_1 \overrightarrow{d}_1 + \overrightarrow{\Delta}||^2 \tag{C.1}$$

[SE02] proves that the function F reaches its global minimum at $(\bar{t}_0, \bar{t}_1)$ if

$$\bar{t}_0 \overrightarrow{d}_0 - \bar{t}_1 \overrightarrow{d}_1 + \overrightarrow{\Delta} = \overrightarrow{0} \tag{C.2}$$

Let $\overrightarrow{d}_0^{\perp}$ and $\overrightarrow{d}_1^{\perp}$ two vectors perpendicular to $\overrightarrow{d}_0$ and $\overrightarrow{d}_1$ respectively. According to [SE02], the solution is:

$$(\bar{t}_0, \bar{t}_1) = \frac{(\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{\Delta}, \overrightarrow{d}_0^{\perp} \cdot \overrightarrow{\Delta})}{\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{d}_0}$$

However, the denominator, as it appears in [SE02], is incorrect. The book's approach is correct but one of the steps is wrong. Dotting the equation (C.2) with $\overrightarrow{d}_1^{\perp}$ leads to the solution for $\bar{t}_0$ as follows:

$$\overrightarrow{d}_1^{\perp} \cdot (\bar{t}_0 \overrightarrow{d}_0 - \bar{t}_1 \overrightarrow{d}_1 + \overrightarrow{\Delta}) = \bar{t}_0 (\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{d}_0) + \overrightarrow{d}_1^{\perp} \cdot \overrightarrow{\Delta} = \overrightarrow{0}$$

Solving for $\bar{t}_0$, we get:

$$\bar{t}_0 = \frac{-\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{\Delta}}{\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{d}_0} \tag{C.3}$$

On the other hand, it can be proved that $\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{d}_0 = -\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{d}_1$ as follows. Let $\overrightarrow{d}_i = (u_i, v_i), i = 0, 1$ and $\overrightarrow{d}_i^{\perp} = (-v_i, u_i), i = 0, 1$

$$\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{d}_0 = (-v_1, u_1) \cdot (u_0, v_0) = -u_0 v_1 + v_0 u_1 =$$

$$= -(-v_0 u_1 + u_0 v_1) = -(-v_0, u_0) \cdot (u_1, v_1) = -\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{d}_1$$

Replacing this in the equation (C.3) we get:

$$\bar{t}_0 = \frac{\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{\Delta}}{\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{d}_1}$$

In the same way, if we dot the equation (C.2) with $\overrightarrow{d}_0^{\perp}$, we get the solution for $\bar{t}_0$ as follows:

$$\overrightarrow{d}_0^{\perp} \cdot (\bar{t}_0 \overrightarrow{d}_0 - \bar{t}_1 \overrightarrow{d}_1 + \overrightarrow{\Delta}) = -\bar{t}_1(\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{d}_1) + \overrightarrow{d}_0^{\perp} \cdot \overrightarrow{\Delta} = \overrightarrow{0}$$

Solving for $\bar{t}_1$, we get:

$$\bar{t}_1 = \frac{\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{\Delta}}{\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{d}_1} \tag{C.4}$$

Thus, the final solution with the right denominator is:

$$(\bar{t}_0, \bar{t}_1) = \frac{(\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{\Delta}, \overrightarrow{d}_0^{\perp} \cdot \overrightarrow{\Delta})}{\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{d}_1}$$

As stated before, this correction has been accepted by the authors and published in the corrections list.

We will distinguish between parallel and non parallel segments in order to calculate the distance.

If the segments are not parallel and $\bar{t}_0 \in [0,1]$ y $\bar{t}_1 \in [0,1]$, the intersection lies on the interior of the segments and the distance is zero. Otherwise, the relative position of the segments needs to be analyzed, in order to determine which values for the parameters in the equation make the distance minimum. If we define:

$$\begin{aligned}
\hat{t}_0 &= -(\overrightarrow{d}_0 \cdot \overrightarrow{\Delta})/||\overrightarrow{d}_0||^2 \\
\hat{t}_1 &= \overrightarrow{d}_1 \cdot \overrightarrow{\Delta}/||\overrightarrow{d}_1||^2 \\
\tilde{t}_0 &= (\overrightarrow{d}_0 \cdot \overrightarrow{d}_1 - \overrightarrow{d}_0 \cdot \overrightarrow{\Delta})/||\overrightarrow{d}_0||^2 \\
\tilde{t}_1 &= (\overrightarrow{d}_0 \cdot \overrightarrow{d}_1 + \overrightarrow{d}_1 \cdot \overrightarrow{\Delta})/||\overrightarrow{d}_1||^2
\end{aligned}$$

then, the distance among two non-parallel segments is as follows (after fixing another errata found in the corrections website, posted on February 5, 2007):

$$\mathtt{d}(S_1,S_2) = \begin{cases}
0, & \text{if } \bar{t}_0 \in (0,1) \text{ and } \bar{t}_1 \in (0,1) \\
|\overrightarrow{d}_0^{\perp} \cdot \overrightarrow{\Delta}|/||\overrightarrow{d}_0||, & \text{if } \hat{t}_0 \in (0,1) \text{ and } \bar{t}_1 \le 0 \\
|\overrightarrow{d}_0^{\perp} \cdot (\overrightarrow{\Delta} - \overrightarrow{d}_1)|/||\overrightarrow{d}_0||, & \text{if } \tilde{t}_0 \in (0,1) \text{ and } \bar{t}_1 \ge 1 \\
|\overrightarrow{d}_1^{\perp} \cdot \overrightarrow{\Delta}|/||\overrightarrow{d}_1||, & \text{if } \hat{t}_1 \in (0,1) \text{ and } \bar{t}_0 \le 0 \\
|\overrightarrow{d}_1^{\perp} \cdot (\overrightarrow{\Delta} + \overrightarrow{d}_0)|/||\overrightarrow{d}_1||, & \text{if } \tilde{t}_1 \in (0,1) \text{ and } \bar{t}_0 \ge 1 \\
||\overrightarrow{\Delta}||, & \text{if } \hat{t}_0 \le 0 \text{ and } \hat{t}_1 \le 0 \\
||\overrightarrow{\Delta} + \overrightarrow{d}_0||, & \text{if } \hat{t}_0 \ge 1 \text{ and } \tilde{t}_1 \le 0 \\
||\overrightarrow{\Delta} - \overrightarrow{d}_1||, & \text{if } \tilde{t}_0 \le 0 \text{ and } \hat{t}_1 \ge 1 \\
||\overrightarrow{\Delta} + \overrightarrow{d}_0 - \overrightarrow{d}_1||, & \text{if } \tilde{t}_0 \ge 1 \text{ and } \tilde{t}_1 \ge 1
\end{cases}$$

The book [SE02] contains some mistakes when dealing with parallel segments. The formulas to compute the distance are the following, according to [SE02]:

$$\mathtt{d}(S_1,S_2) = \begin{cases} ||\overrightarrow{\Delta}||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 < 0 \text{ and } \overrightarrow{d}_0 \cdot \overrightarrow{\Delta} \geq 0 \\ ||\overrightarrow{\Delta} + \overrightarrow{d}_0||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 > 0 \text{ and } \overrightarrow{d}_0 \cdot (\overrightarrow{\Delta} + \overrightarrow{d}_0) \geq 0 \\ ||\overrightarrow{\Delta} - \overrightarrow{d}_1||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 > 0 \text{ and } \overrightarrow{d}_0 \cdot (\overrightarrow{\Delta} - \overrightarrow{d}_1) \geq 0 \\ ||\overrightarrow{\Delta} + \overrightarrow{d}_0 - \overrightarrow{d}_1||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 < 0 \text{ and } \overrightarrow{d}_0 \cdot (\overrightarrow{\Delta} + \overrightarrow{d}_0 - \overrightarrow{d}_1) \geq 0 \\ |\overrightarrow{d}_0 \cdot \overrightarrow{\Delta}|/||\overrightarrow{d}_0||, & \text{otherwise} \end{cases} \tag{C.5}$$

We realized that the direct application of these formulas did not produce the expected results. The formulas can be deduced from the relative position among all the vectors involved in the calculation. Therefore, a graphical representation helps to understand the formulas and to detect possible errors.

First, note that $\overrightarrow{d}_0$ and $\overrightarrow{d}_1$ have either the same or opposite directions, since the studied segments are parallel. Second, if the dot product between $\overrightarrow{d}_0$ and $\overrightarrow{d}_1$ is positive, both vectors have the same direction; otherwise they have opposite directions.

Figure C.1 shows a graphical representation of the vectors involved in the calculation of the segment distance. Cases 1 to 4 represent respectively each case from Equation C.5. As can be see in the figure, the second and the fourth conditions from the equation do not match with the graphical description of the condition. Therefore, Equation C.6 shows the right conditions (highlighted in green).

$$\mathtt{d}(S_1,S_2) = \begin{cases} ||\overrightarrow{\Delta}||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 < 0 \text{ and } \overrightarrow{d}_0 \cdot \overrightarrow{\Delta} \geq 0 \\ ||\overrightarrow{\Delta} + \overrightarrow{d}_0||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 > 0 \text{ and } \overrightarrow{d}_0 \cdot (\overrightarrow{\Delta} + \overrightarrow{d}_0) \leq 0 \\ ||\overrightarrow{\Delta} - \overrightarrow{d}_1||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 > 0 \text{ and } \overrightarrow{d}_0 \cdot (\overrightarrow{\Delta} - \overrightarrow{d}_1) \geq 0 \\ ||\overrightarrow{\Delta} + \overrightarrow{d}_0 - \overrightarrow{d}_1||, & \text{if } \overrightarrow{d}_0 \cdot \overrightarrow{d}_1 < 0 \text{ and } \overrightarrow{d}_0 \cdot (\overrightarrow{\Delta} + \overrightarrow{d}_0 - \overrightarrow{d}_1) \leq 0 \\ |\overrightarrow{d}_0 \cdot \overrightarrow{\Delta}|/||\overrightarrow{d}_0||, & \text{otherwise} \end{cases} \tag{C.6}$$

C.3 Triangulation

This algorithm is based in the ear clipping triangulation algorithm [SE02]. We have implemented it in an iterative way, as shown in the pseudo code of Algorithm C.6. The main reason to implement an iterative version of the algorithm is to increase the efficiency by reducing the number of function recursive calls. The input of the function is a sorted list of vertices, while the output is a list of triangles.

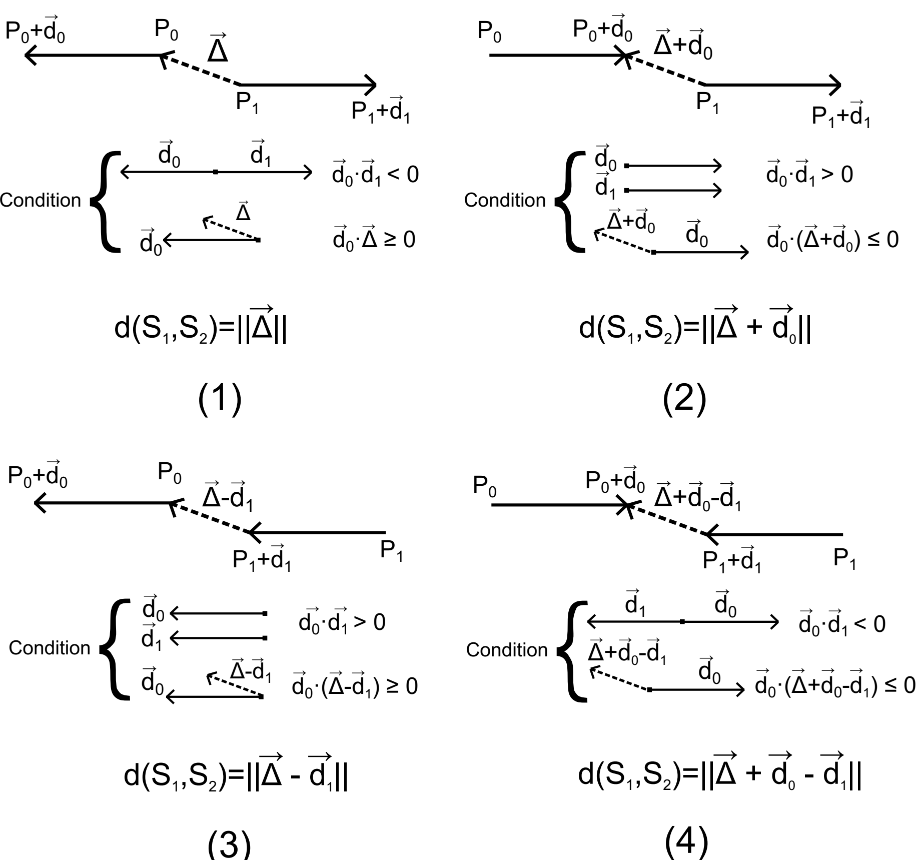


Figure C.1: Cases for the calculation of the distance between two segments

Algorithm C.6: Triangulate a polygon defined by a sorted set of vertices

```
   Input: vlist: List of vertices of the polygon to be triangulated
   Output: tlist: List of triangles
 1 begin
 2     Start an empty list of triangle indices tlist
 3     while vlist.size > 3 do
 4         diagonalFound ← false
 5         n ← vlist.size
 6         i0 ← 0, i1 ← 1, i2 ← 2
 7         while i0 < n and diagonalFound = false do
 8             if isDiagonal(vlist,i0,i2) then
 9                 diagonalFound ← true
10                 tlist.Add(i0,i1,i2)
11                 vlist.Remove(i1)
12             end
13             i0 ← i0 + 1, i1 ← (i1 + 1)%n, i2 ← (i2 + 1)%n
14         end
15         tlist.Add(vlist[0], vlist[1], vlist[2])
16     end
17     return tlist
18 end
```

Algorithm C.7: Check if two vertices of a polygon make up a diagonal

```
   Input: vlist: sorted list of vertices that define a polygon
   i,j: indices of the vertices to be checked
   Output: true if indices i,j make up a diagonal
   false otherwise
 1 begin
 2     n ← vlist.size
 3     a ← (i + n − 1)%n
 4     b ← (i + 1)%n
 5     if segmentInCone(vlist[i], vlist[j], vlist[a], vlist[b]) = false then
 6         return false
 7     end
 8     j0 ← 0,j1 ← 1
 9     while j0 < n do
10         if j0 ≠ i and j0 ≠ j and j1 ≠ i and j1 ≠ j then
11             L0 ← (vlist[i], vlist[j])
12             L1 ← (vlist[j0], vlist[j1])
13             if IntersectionSegments(L0,L1).Size() ≠ 0 then
14                 return false
15             end
16         end
17         j1 ← j0
18         j0 ← j0 + 1
19     end
20     return true
21 end
```

Algorithm C.8: Check if a segment is contained in the cone made up by its two adjacent segments

```
   Input: v0,v1,v2,v3: vertices to be checked
   Output: true if the segment is in the cone
   false otherwise
 1 begin
 2     diff ← v1 − v0
 3     edgeL ← v2 − v0
 4     edgeR ← v3 − v0
 5     if kross(edgeR,edgeL)> 0 then
 6         return kross(diff,edgeL) > 0and kross(diff,edgeR)< 0
 7     end
 8     return kross(diff,edgeR) < 0 or kross(diff,edgeL) > 0
 9 end
```

Appendix D

Publications related to this work

This appendix summarizes the papers published during the development of the PhD.

D.1 Conference proceedings

B. Domínguez, Á.L. García-Fernández, F. Feito: An Open Source Approach to Semiautomatic 3D Scene Generation for Interactive Indoor Navigation Environments. *In: Proceedings of the IV Iberoamerican Symposium on Computer Graphics.* Isla de Margarita (Venezuela), July 2009.

B. Domínguez, F. Conde, Á.L. García-Fernández, F. Feito: Desarrollo de una Herramienta Semiautomática para la Detección de Elementos Semánticos en un Plano Arquitectónico. *In: Proceedings of the Seminario de Realidad Virtual (SEREVI).* Baeza (Spain), October 2009.

B. Domínguez, Á.L. García-Fernández, F. Feito: Detección semiautomática de paredes, habitaciones y escaleras a partir de planos arquitectónicos CAD. *In: Proceedings of the Congreso Español de Informática Gráfica.* Valencia (Spain), September 2010.

B. Domínguez, Á.L. García-Fernández, F. Feito: Semantic and topological representation of building indoors: an overview. *In: Proceedings of the Joint ISPRS Workshop on 3D City Modelling and Applications.* Wuhan (China), July 2011.

D.2 Book chapters

B. Domínguez, F. Conde, Á.L. García-Fernández, F. Feito: Análisis, diseño e implementación de una herramienta para la reconstrucción semiautomática

de modelos 3D de edificios a partir de planos arquitectónicos. *In: Aplicación de Herramientas CAD a Realidad Virtual: Representaciones Jerárquicas y Luces Virtuales*. Eds: Lidia Ortega Alvarado, Alejandro J. León Salas. ISSN: 978-84-15026-96-9. 2010.

D.3 Journals

D.3.1 Published

B. Domínguez, Á.L. García-Fernández, F. Feito: Semiautomatic detection of floor topology from CAD architectural drawings. *In: Computer-Aided Design, 44(5)*. ISSN: 0010-4485. Elsevier, 2012.

D.3.2 Submitted

B. Domínguez, S. Zlatanova, Á.L. García-Fernández, P. van Oosterom: On the generation of 2D and 3D topologically correct models from 2D architectural plans. *Submitted to: Geoinformatica*. Springer, May 2014.

B. Domínguez, F. Conde, Á.L. García-Fernández, F. Feito: Detection of Semantic Elements in a CAD Floor Plan Using a Semiautomatic Tool for Processing Vector Drawings. *Submitted to: Advanced Engineering Informatics*. Elsevier, July 2014.

Bibliography

[AB06] R. Arnaud and M.C. Barnes. *COLLADA: sailing the gulf of 3D digital content creation*. Ak Peters Series. A K Peters, 2006.

[AST01] C. Ah-Soon and K. Tombre. Architectural symbol recognition using a network of constraints. *Pattern Recognition Letters*, 22(2):231–248, 2001.

[Aut11] Autodesk, Inc. *AutoCAD 2011 DXF Reference*, 2011.

[BD07] Don Brutzman and Leonard Daly. *X3D. Extensible 3D graphics for web authors*. Morgan Kaufmann, 2007.

[BG10] Pavel Boguslawski and Christopher Gold. Rapid modelling of complex building interiors. In *Proceedings of the 5th International 3D GeoInfo Conference*, 2010.

[Bjo92] Bo-Christer Bjork. A conceptual model of spaces, space boundaries and enclosing structures. *Automation in Construction*, 1(3):193–214, 1992.

[BR09] André Borrmann and Ernst Rank. Specification and implementation of directional operators in a 3d spatial query language for building information models. *Adv. Eng. Inform.*, 23(1):32–44, 2009.

[bui14] buildingSMART. *IFC Technical Specifications*, 2014. http://buildingsmart-tech.org.

[BZ03] Roland Billen and Siyka Zlatanova. 3d spatial relationships model: a useful concept for 3d cadastre? *Computers, Environment and Urban Systems*, 27(4):411–425, 2003.

[car13] *Carve*, 2013. http://carve-csg.com/.

[CF08] Christian Clemen and Gielsdorf Frank. Architectural indoor surveying. an information model for 3d data capture and adjustment. In *Proceedings of the American Congress on Surveying and Mapping*, 2008.

[cit] *citygml4j.* http://opportunity.bv.tu-berlin.de/software/projects/citygml4j.

[CKHL07] Jin Won Choi, Doo Young Kwon, Jie Eun Hwang, and Jumphon Lertlakkhanakul. Real-time management of spatial information of design: A space-based floor plan representation of buildings. *Automation in Construction*, 16(4):449–459, 2007.

[CL09] Jinmu Choi and Jiyeong Lee. 3d geo-network for agent-based building evacuation simulation. In Jiyeong Lee and Sisi Zlatanova, editors, *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, pages 283–299. Springer Berlin Heidelberg, 2009.

[D+11] Nigel Davies et al. *AEC (UK) CAD Standard for basic layer naming, v.3.0*, 2011. http://aecuk.files.wordpress.com/.

[DFAB03] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall, 3rd edition, 2003.

[DGF09] Bernardino Domínguez, Ángel Luis García, and Francisco R. Feito. An Open Source Approach to Semiautomatic 3D Scene Generation for Interactive Indoor Navigation Environments. In *Proceedings of IV Ibero-American Symposium on Computer Graphics*, pages 131–138, 2009.

[DGF12] B. Domínguez, Á.L. García, and F.R. Feito. Semiautomatic detection of floor topology from CAD architectural drawings. *Computer-Aided Design*, 44(5):367–378, 2012.

[DTASM00] Philippe Dosch, Karl Tombre, Christian Ah-Soon, and Gérald Masini. A complete system for the analysis of architectural drawings. *International Journal on Document Analysis and Recognition*, 3:102–116, 2000. 10.1007/PL00010901.

[ETSL08] Chuck Eastman, Paul Teicholz, Rafael Sacks, and Kathleen Liston. *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors.* John Wiley and Sons, 1st edition, 2008.

[FFSB04] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Head First Design Patterns.* O'Reilly, 2004.

[FMW05] Gerald Franz, Hanspeter A. Mallot, and Jan M. Wiener. Graph-based models of space in architecture and cognitive science - a comparative analysis. In *Proceedings of the 17th International Conference on Systems Research, Informatics and Cybernetics*, 2005.

[Fow04] Martin Fowler. *UML Distilled.* Addison-Wesley, Boston, 2004.

[FTU95] Francisco R. Feito, Juan Carlos Torres, and A. Ureña. Orientation, simplicity, and inclusion test for planar polygons. *Computers & Graphics*, 19(4):595–600, 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[GKCN08] Gerhard Gröger, Thomas H. Kolbe, Angela Czerwinski, and Claus Nagel. OpenGIS City Geography Markup Language (CityGML) Encoding Standard (OGC 08-007r1). Technical report, Open Geospatial Consortium, Inc., 2008.

[GLY+13] Renzhong Guo, Lin Li, Shen Ying, Ping Luo, Biao He, and Renrong Jiang. Developing a 3D cadastre for the administration of urban land use: A case study of Shenzhen, China. *Computers, Environment and Urban Systems*, 40:46–55, 2013.

[Goe13] Marcus Goetz. Towards generating highly detailed 3d citygml models from openstreetmap. *International Journal of Geographical Information Science*, 27(5):845–865, 2013.

[GS09] T. Germer and M. Schwarz. Procedural arrangement of furniture for real-time walkthroughs. *Computer Graphics Forum*, 28(8):2068–2078, 2009.

[HB07] Rob Howard and Bo-Christer Bjork. Use of standards for cad layers in building. *Automation in Construction*, 16(3):290 – 297, 2007.

[HBW06] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, pages 179–186, New York, NY, USA, 2006. ACM.

[HMDB09] S. Horna, D. Meneveaux, G. Damiand, and Y. Bertrand. Consistency constraints and 3D building reconstruction. *Computer Aided Design*, 41(1):13–27, 2009.

[HTGD09] Benjamin Hagedorn, Matthias Trapp, Tassilo Glander, and Jurgen Dollner. Towards an indoor level-of-detail model for route visualization. In *Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, MDM '09, pages 692–697, Washington, DC, USA, 2009. IEEE Computer Society.

[IAUW07] Umit Isikdag, Ghassan Aouad, Jason Underwood, and Song Wu. Building information models: a review on storage and exchange mechanisms. In Daniel Rebolj, editor, *Proceedings of CIB W78*, Maribor (Slovenia), 2007.

[ISO98] ISO (International Organization for Standardization). *ISO 13567: Organization and naming of layers for CAD*, 1998. http://www.iso.org.

[IUA08] Umit Isikdag, Jason Underwood, and Ghassan Aouad. An investigation into the applicability of building information models in geospatial environment in support of site selection and fire response management processes. *Advanced Engineering Informatics*, 22(4):504 – 519, 2008. PLM Challenges.

[jGr] *JGraphT*. http://www.jgrapht.org/.

[kab] *Kabeja*. http://kabeja.sourceforge.net.

[Kol] Thomas H. Kolbe. *CityGML: Exchange and storage of Virtual 3D City Models*. Technische Universitaet Berlin. http://www.citygml.org.

[Kol07] Thomas H. Kolbe. Citygml 3d geospatial and semantic modelling of urban structures. In *Proceedings of the GITA/OGC Emerging Technology Summit 4*, 2007.

[KTM13] Sudarshan Karki, Rod Thompson, and Kevin McDougall. Development of validation rules to support digital lodgement of 3D cadastral plans. *Computers, Environment and Urban Systems*, 40:34–45, 2013.

[LCR10] Xiang Li, Christophe Claramunt, and Cyril Ray. A grid graph-based model for the analysis of 2d indoor spaces. *Computers, Environment and Urban Systems*, 34(6):532 – 540, 2010. GeoVisualization and the Digital City - Special issue of the International Cartographic Association Commission on GeoVisualization.

[LD04] Fabrice Lamarche and Stphane Donikian. Crowd of virtual humans: a new approach for real time navigation in complex and structured environments. *Computer Graphics Forum*, 23:509–518, 2004.

[Lee82] D.T. Lee. Medial axis transformation of a planar shape. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):363–369, 1982.

[Lee01] Jiyeong Lee. 3d data model for representing topological relations of urban features. In *Proceedings of 21st ESRI International User Conference*, 2001.

[LK05] Jiyeong Lee and M-P Kwan. A combinatorial data model for representing topological relations among 3D geographical features in micro-spatial environments. *International Journal of Geographical Information Science*, 19(10):1039–1056, 2005.

[LLKM97] Josep Lladós, Jaime López-Krahe, and Enric Martí. A system to understand hand-drawn floor plans using subgraph isomorphism and hough transform. *Mach. Vision Appl.*, 10(3):150–158, 1997.

[LM09] Hugo Ledoux and Martijn Meijers. Extruding building footprints to create topologically consistent 3D city models. In A.Krek, M.Rumor, S.Zlatanova, and E.M.Fendel, editors, *Urban and Regional Data Management, UDMS Annuals*, pages 39–48. Taylor & Francis Group, 2009.

[LYYC07] Tong Lu, Huafei Yang, Ruoyu Yang, and Shijie Cai. Automatic analysis and integration of architectural drawings. *Int. J. Doc. Anal. Recognit.*, 9(1):31–47, 2007.

[Män88] Martti Mäntylä. *An Introduction to Solid Modeling*. W.H. Freeman & Company, 1988.

[MB06] A. Mas and G. Besuievsky. Automatic architectural 3D model generation with sunlight simulation. In *Ibero-American Symposium on Computer Graphics*, pages 37–44, 2006.

[MSK10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics*, 29(6):181:1–181:12, December 2010.

[MY00] B. Medjdoub and B. Yannou. Separating topology and geometry in space planning. *Computer-Aided Design*, 32(1):39 – 61, 2000.

[MyS] MySQL AB. *MySQL 5.1 Reference Manual*. http://dev.mysql.com/doc.

[NCS11] *National CAD Standard, v.5.0*, 2011. http://www.nationalcadstandard.org.

[OGC] Inc. Open Geospatial Consortium. http://www.opengeospatial.org.

[OWYC05] Siu Hang Or, Kin Hong Wong, Ying Kin Yu, and Michael Ming Yuan Chang. Highly automatic approach to architectural floorplan image understanding & model generation. In *Proceedings of 10th Fall Workshop Vision, Modeling, and Visualization*, pages 25–32, 2005.

[PB03] Norbert Paul and Patrick Erik Bradley. Topological houses. In *Proceedings of the 16th International Conference of Computer Science and Mathematics in Architecture and Civil Engineering*, 2003.

[PG96] Lutz Pluemer and Gerhard Groeger. Nested maps - a formal, provably correct object model for spatial aggregates. In *Proceedings of the 4th ACM international workshop on Advances in geographic*

information systems, GIS '96, pages 76–83, New York, NY, USA, 1996. ACM.

[PMSV08] Alberto Paoluzzi, Franco Milicchio, Giorgio Scorzelli, and Michele Vicentino. From 2d plans to 3d building models for security modeling of critical infrastructures. *International Journal of Shape Modeling*, 14(1):61–78, 2008.

[ROOC+13] María-Dolores Robles-Ortega, Lidia Ortega, Antonio Coelho, Francisco Feito, and Augusto de Sousa. Automatic street surface modeling for web-based urban information systems. *Journal of Urban Planning and Development*, 139(1):40–48, 2013.

[ROOF13] María-Dolores Robles-Ortega, Lidia Ortega, and Francisco Feito. A new approach to create textured urban models through genetic algorithms. *Information Sciences*, 243(0):1 – 19, 2013.

[Sal78] Kenneth B. Salomon. An efficient point-in-polygon algorithm. *Computers & Geosciences*, 4(2):173–178, 1978.

[SE02] Philip J. Schneider and David Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[SLO07] Edgar-Philipp Stoffel, Bernhard Lorenz, and Hans Ohlbach. Towards a semantic spatial model for pedestrian indoor navigation. In Jean-Luc Hainaut, Elke Rundensteiner, Markus Kirchberg, Michela Bertolotto, Mathias Brochhausen, Yi-Ping Chen, Samira Cherfi, Martin Doerr, Hyoil Han, Sven Hartmann, Jeffrey Parsons, Geert Poels, Colette Rolland, Juan Trujillo, Eric Yu, and Esteban Zimányie, editors, *Advances in Conceptual Modeling Foundations and Applications*, volume 4802 of *Lecture Notes in Computer Science*, pages 328–337. Springer Berlin / Heidelberg, 2007.

[SR07] Aidan Slingsby and Jonathan Raper. Navigable space in 3d city models for pedestrians, 2007.

[TBSdK09] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. Rule-based layout solving and its application to procedural interior generation. In *Proceedings of the CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)*, 2009.

[TC11] Tomo and Cerovsek. A review and outlook for a building information model (bim): A multi-standpoint framework for technological development. *Advanced Engineering Informatics*, 25(2):224 – 244, 2011. ¡ce:title¿Information mining and retrieval in design¡/ce:title¿.

[vBdL10] Léon van Berlo and Ruben de Laat. Integration of bim and gis: The development of the citygml geobim extension. In *Proceedings of the 5th International 3D GeoInfo Conference*, 2010.

[vD08] Joost van Dongen. Interior mapping. In *CGI 2008 Conference Proceedings*, 2008.

[vTR07] Christoph van Treeck and Ernst Rank. Dimensional reduction of 3d building models using graph theory and its application in building energy simulation. *Eng. with Comput.*, 23:109–122, April 2007.

[Wei88] K. Weiler. The radial-edge structure: a topological representation for non-manifold geometric boundary modeling. In *Geometric Modeling for CAD Applications*, pages 3–36. Elsevier, 1988.

[XZD+13] Xiao Xie, Qing Zhu, Zhiqiang Du, Weiping Xu, and Yeting Zhang. A semantics-constrained profiling approach to complex 3D city models. *Computers, Environment and Urban Systems*, 41:309–317, 2013.

[XZZ10] Weiping Xu, Qing Zhu, and Yeting Zhang. Semantic modeling approach of 3d city models and applications in visual exploration. *International Jorunal of Virtual Reality*, 9(3):67–74, 2010.

[YCG10] Wei Yan, Charles Culp, and Robert Graf. Integrating bim and gaming for real-time interactive architectural visualization. *Automation in Construction*, In Press, Corrected Proof:–, 2010.

[YWR09] Xuetao Yin, Peter Wonka, and Anshuman Razdan. Generating 3D building models from architectural drawings: a survey. *IEEE Computer Graphics and Applications*, 29(1):20–30, 2009.

[YZ08] Li Yuan and He Zizhang. 3D indoor navigation: a framework of combining bim with 3D GIS. In *44th ISOCARP Congress*, 2008.

[Zla00] S. Zlatanova. 3D GIS for urban development, 2000.

[ZLF03] G.S. Zhi, S.M. 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*, 35(1):1–14, 2003.

Anexo

Resumen en castellano

Métodos de procesamiento de planos CAD de bajo nivel para la creación de Modelos de Información de Edificios (BIM)

Bernardino Domínguez

Agosto de 2014

Índice general

Índice general III
Índice de figuras VII
Índice de algoritmos IX
Índice de cuadros XI

I Introducción **1**

1. Introducción **5**
1.1. Motivación 6
1.2. Algoritmos para la obtención de información semántica 8
1.3. Representación semántica de edificios 11
1.4. Implementación de una herramienta de usuario 12

2. Trabajo previo **13**

II Detección de elementos semánticos **15**

3. Diseño arquitectónico. Restricciones **17**
3.1. Dibujos arquitectónicos. Formatos de fichero 17
3.2. Estándares CAD. Requisitos de la estructura de los planos 17
3.3. Local vs. global 18

4. Irregularidades **21**
4.1. Tipos de irregularidades 21
4.2. Detección de irregularidades 22
4.3. Eliminación de irregularidades 22

5. Detección local de habitaciones **23**
5.1. Proceso de extracción de características 23
5.1.1. Cálculo del grosor de paredes 23
5.1.2. Extracción de puntos clave, habitaciones y aberturas. Reglas de detección 24
5.1.3. Almacenamiento de datos 24

5.2. Generación de salida 3D 24
5.3. Resultados 24

6. Detección global de habitaciones **27**
6.1. Detección de paredes 28
6.1.1. Grafo de Adyacencia de Paredes (GAP) 29
6.1.1.1. Grafo de Adyacencia de Paredes Generalizado (GAPG) 30
6.1.1.2. GAP Y GAPG para paredes curvas 30
6.2. Detección de aberturas 30
6.2.1. Detección de aberturas a partir de instancias 30
6.2.2. Detección de aberturas a partir de primitivas 31
6.3. Clustering 31
6.3.1. Algoritmo de crecimiento de líneas 32
6.4. Resultados 33
6.4.1. Detección de paredes 33
6.4.2. Construcción del grafo de topología 34

III Representación de información de edificios **39**

7. Estructura de tres niveles para modelos correctos **43**
7.1. Análisis comparativo y discusión 43
7.2. Módulo de geometría 46
7.3. Módulo de semántica 47
7.4. Módulo de topología 47

8. Topología 2D. Procesamiento del plano CAD **49**
8.1. Construcción el grafo de topología 49
8.2. Enlazando información geométrica y topológica 50
8.3. Resultados 50

9. Geometría 3D. Triple extrusión **53**
9.1. Construir un DXF de tres vistas 54
9.2. Importación del DXF. Índices espaciales 54
9.3. Detección de contorno de las vistas 54
9.4. Triangulación del polígono de contorno 55
9.5. Transformación de coordenadas 2D a 3D 55
9.6. Extrusión de las vistas 55
9.7. Resultados 56

10. Topología y semántica 3D. CityGML **59**
10.1. Resultados 59

IV Conclusiones y trabajo futuro **63**

11. Conclusiones y trabajo futuro **67**
11.1. Conclusiones 67
11.2. Trabajo futuro 69
11.2.1. Algoritmo de detección de paredes 69
11.2.2. Detección de aberturas 70
11.2.3. Clustering. Construcción del grafo de topología 70
11.2.4. Esquema de tres niveles para la representación de modelos topológicamente correctos 70
11.2.5. Triple extrusión 70
11.2.6. Otro trabajo futuro 71

Bibliografía **73**

Índice de figuras

4.1. Casos de irregularidades 21

5.1. Uno de los planos utilizado en las pruebas 25
5.2. Plano y su correspondiente modelo 3D 26

6.1. Ejemplo del algoritmo de crecimiento de líneas 33
6.2. Planos reales utilizados para probar nuestros algoritmos, junto con los resultados de aplicar el algoritmo de detección de paredes. Las paredes detectadas están dibujadas en azul. 35
6.3. Representación de la topología de parte de un plano CAD vectorial 37
6.4. Representación de la topología superpuesta al plano CAD vectorial 37

7.1. Arquitectura de tres niveles para modelos de edificios 46

8.1. Ejemplo del algoritmo de rejilla 51
8.2. Detección de los polígonos exteriores e interiores 52

9.1. Índice espacial de rejilla 55
9.2. Extrusión de las vistas 56
9.3. Ejemplo de la triple extrusión 56
9.4. Resultado de la triple extrusión 57

10.1. UML del enlace entre nuestro modelo y CityGML 60
10.2. Modelo CityGML del edificio 61

Índice de algoritmos

6.1. Clustering utilizando los extremos de las secuencias 32
6.2. Algoritmo de crecimiento de líneas 34

Índice de cuadros

3.1. Lista de primitivas en diseño CAD 2D y sus atributos. 18

6.1. Comparación de los criterios para formar pares candidatos a pared entre arcos rectos y circulares 30
6.2. Datos numéricos de las pruebas de la detección de paredes con planos reales 36
6.3. Resultados de las pruebas en edificios reales 36

7.1. Comparación entre los trabajos revisados. 45

Parte I

Introducción

En esta parte se introduce el presente trabajo. Se incluye la motivación del mismo, y se presenta un breve resumen de cada parte y cada capítulo en que está estructurado. Además, se incluye un estudio de los trabajos previos más importantes.

Esta parte se estructura como sigue: el capítulo 1 contiene la motivación y el resumen del resto del documento, mientras que el capítulo 2 analiza los trabajos previos sobre la reconstrucción automática y los modelos de edificios.

Capítulo 1

Introducción

El interés creciente de los usuarios y desarrolladores en herramientas informáticas relacionadas con la información geográfica es remarcable en la actualidad.

Por una parte, en el ámbito de los usuarios, hay muchas tecnologías que se han hecho rápidamente populares: el uso de páginas web para consultar mapas, planificar rutas de viaje o ver imágenes de lugares remotos, o el uso de navegadores GPS para guiar a los viajeros. Más recientemente, estas tecnologías, combinadas con las redes sociales, han dado lugar a aplicaciones para teléfonos inteligentes o tabletas que pueden utilizarse para comunicar la ubicación actual a otros usuarios, acceder a mapas desde dispositivos móviles, o utilizar el GPS integrado para llegar a un destino concreto caminando o utilizando algún otro medio de transporte.

Por otra parte, en el ámbito de los desarrolladores, los servicios de mapas más populares, como Google Maps o Bing Maps permiten incluir información cartográfica en páginas web mediante marcos (frames) embebidos, o bien utilizando las APIs de programación que estos servicios ofrecen. Hoy en día es habitual que páginas web que incluyan información sobre la ubicación de un establecimiento, la ruta de una competición deportiva o las indicaciones para llegar a un destino concreto, utilicen estos servicios en lugar de elaborar sus propios mapas.

En entornos urbanos, estos mapas pueden completarse con modelos 3D de edificios. Empresas como Google o Microsoft (con anterioridad a 2010[1]) ofrecen a los usuarios las herramientas y recursos necesarios para poblar el mapa con modelos 3D de edificios. Aparte del impacto de esta característica como modelo estratégico de negocio, resalta el interés de los usuarios en la visualización de modelos urbanos.

De este hecho, podemos extraer algunas conclusiones: por un lado, es im-

[1]En noviembre de 2010, Bing eliminó los edificios en 3D, para centrarse en la ”vista de pájaro”(*http://www.gearthblog.com/blog/archives/*2010/11/*bing_maps_is_dropping_their_3d_vers.html*. última visita: 2014-07-30)

portante que los modelos no estén siempre al máximo nivel de detalle (las herramientas colaborativas antes mencionadas normalmente limitan el tamaño de los modelos que pueden subir los usuarios, de forma que éstos han de recurrir a técnicas como el uso de texturas o la simplificación de la geometría). Por otro lado, es relevante que las herramientas existentes no están ideadas para la simulación de navegación a nivel peatonal, sino a nivel aéreo. Además, no tenemos constancia de que incluyan la capacidad de navegar por el interior de los edificios.

Esta tendencia muestra el interés de los usuarios en la información urbana tridimensional, e implica la creación de tecnologías especializadas para el modelado urbano, como GML o CityGML. Así, la creación de contenido urbano virtual de forma manual o automática es una tarea clave para el desarrollo de estas tecnologías. En este trabajo, se va a tratar la generación automática de contenido urbano virtual.

La información geográfica combina disciplinas que han evolucionado por separado, tales como la Cartografía, encargada de la generación y gestión de mapas, y la Informática, dando como resultado los Sistemas de Información Geográfica (SIG o GIS). La combinación de los SIG con la Arquitectura y la Ingeniería Civil da lugar a modelos enriquecidos con información sobre edificios y carreteras. Finalmente, estos sistemas se combinan con bases de datos y sistemas de información de otros tipos para incluir contenidos adicionales en estos sistemas. En todo este proceso, la Informática actúa como nexo de unión y herramienta para la gestión de toda esta información.

1.1. Motivación

El trabajo contenido en este documento es parte de un proyecto de investigación sobre gestión de información urbana tridimensional concedido al Grupo de Geomática e Informática Gráfica de la Universidad de Jaén en 2008. Los objetivos de dicho proyecto eran los siguientes:

1. El desarrollo de prototipos de software para la interacción con información urbana tridimensional.

2. El diseño y mantenimiento de un servidor de mapas que incluya información urbana 3D, accesible a través del software mencionado anteriormente.

3. El estudio y diseño de aplicaciones que permitan una explotación eficiente de las herramientas anteriores, para así colaborar en el desarrollo local.

Una de las tareas previstas era el desarrollo de tecnologías de reconstrucción 3D automática, y su inclusión en los prototipos desarrollados. Como fuentes de datos para este trabajo, se utilizaron la Sede Electrónica del Catastro de España[2] para la generación de contenidos para la navegación urbana en

[2]Disponible en http://www.sedecatastro.gob.es/OVCInicio.aspx

exteriores[ROOC+13, ROOF13], y planos arquitectónicos de edificios públicos para la generación de contenidos 3D para la navegación en interiores.

Respecto a la segunda fuente de información mencionada, se trabajó en dos líneas de investigación en paralelo: la creación de una arquitectura cliente/servidor con datos cartográficos, y la implementación de una página web para navegar a través del campus de la Universidad de Jaén, utilizando para ello modelos en COLLADA. Ambas líneas se explican a continuación:

Cartografía cliente/servidor La primera línea de investigación trataba la implementación de un sistema de cartografía urbana basado en una base de datos cliente/servidor. Los modelos 3D se generaban automáticamente a partir de planos arquitectónicos de la siguiente forma: los planos se procesaban manualmente utilizando MapInfo para eliminar las irregularidades; estos planos eran posteriormente procesados para (1) detectar habitaciones, y (2) extractar un esbozo de la geometría de la planta para generar un modelo 3D del edificio. La información sobre los edificios y su geometría se almacenaba en una base de datos en el servidor, y se consultaba por el cliente para generar modelos X3D.

Contenidos 3D en COLLADA Otra línea de investigación dentro del mismo proyecto trataba la creación de una página web utilizando el plugin de Google Earth para visualización 3D. En esta página, se contenían modelos 3D de todos los edificios del Campus de Las Lagunillas de la Universidad de Jaén.

En pocas palabras: Google Earth permite cargar archivos KML y KMZ. KML (Keyhole Markup Language) es un esquema XML que permite añadir anotaciones georreferenciadas a los mapas de Google Earth (puntos singulares, imágenes, polígonos, modelos 3D y anotaciones). KMZ es un archivo comprimido con el algoritmo ZIP que contiene archivos KML, así como archivos de capas, imágenes y modelos 3D en formato COLLADA.

Los edificios del Campus de la Universidad se modelaron utilizando SketchUp y se exportaron a archivos COLLADA. Los modelos COLLADA se georreferenciaron y se incluyeron en archivos KMZ accesibles a través de una página web con información acerca de la universidad, y ofreciendo la posibilidad de navegar entre de los edificios 3D[3].

Este trabajo se planteó como un complemento de esas dos líneas: de la primera se mantuvo la filosofía de utilizar una base de datos para almacenar la información acerca de la geometría de las habitaciones; de la segunda, la riqueza de los modelos 3D generados. Sin embargo, como en el trabajo realizado respecto de la primera línea se partía de planos en los que las paredes se representaban sin grosor, uno de los objetivos planteados fue detectar, de la manera más automática posible, habitaciones en planos donde las paredes sí se representaban con grosor.

Los objetivos iniciales ya explicados, se reformularon incluyendo nuevas tendencias en la representación de información urbana. Así, se ha trabajado desde dos perspectivas: en primer lugar, parte de la investigación se ha centrado en el

[3]http://www.jaen3d.org/?q=node/2 (última visita: 2014-07-30)

diseño de algoritmos geométricos para extraer contenido semántico de la información geométrica; en segundo lugar, se ha trabajado sobre los modelos semánticos para elaborar una propuesta de un modelo unificado para la representación de edificios, que complemente a los ya existentes. Este modelo incluye también información geométrica extraída de los planos utilizados, como se detallará más adelante.

1.2. Algoritmos para la obtención de información semántica

Uno de los retos más importantes del proyecto era la creación automática de contenido, y su almacenamiento en una base de datos espacial. La primera aproximación consistió en la generación de una base de datos espacial a partir de planos 2D, a semejanza del primer proyecto descrito en la Sección 1.1. A partir de la base de datos, sería posible extraer la información necesaria para reconstruir el 3D del interior del edificio.

Por tanto, las tareas iniciales realizadas fueron: (1) el diseño y la implementación de una base de datos espacial que incluía información semántica correctamente georreferenciada sobre edificios completos, de forma que pudiera integrarse en un SIG, y (2) poblar la base de datos con información real. Para conseguir esto, se estudiaron los planos de los edificios del Campus de Las Lagunillas de la Universidad de Jaén. El reto era automatizar el procesamiento de los planos de las plantas, diseñando una serie de algoritmos para realizar este procesamiento con una intervención mínima del usuario.

Ya en las primeras etapas del trabajo aparecieron problemas, debidos a la dificultad en la caracterización del problema:

- Del mismo modo que en otros ámbitos de aplicación del reconocimiento automático, para un humano (especializado) es sencillo reconocer los elementos semánticos en un plano, pero esto es difícil de automatizar.

  Los planos arquitectónicos están pensados para la visualización. A pesar de la existencia de estándares para el dibujado de planos, existen grandes divergencias en el estilo de unos delineantes y otros. Estas divergencias no afectan el aspecto visual de los planos, pero desde el punto de vista del procesamiento de las entidades geométricas contenidas en éstos, sí hacen peligrar la estabilidad numérica de los algoritmos geométricos, provocando así resultados erróneos o ciclos infinitos en la ejecución.

- Hay una diferencia entre dos tareas relacionadas: la reconstrucción 3D a partir de geometría 2D, y la detección de elementos semánticos. La primera es fácil de automatizar, y su solución es relativamente conocida: las paredes con grosor se representan mediante polígonos cerrados que pueden ser extruidos, mientras que los suelos de las habitaciones se obtienen como la diferencia entre el contorno de la planta y los polígonos que representan las paredes. La segunda requiere un procesamiento completo: en primer

lugar se detectan los elementos semánticos a partir de la geometría original; luego, la geometría asociada a estos elementos semánticos es utilizada para crear los modelos 3D.

En nuestro estudio de la bibliografía sobre detección de semántica en planos arquitectónicos, hemos encontrado varias propuestas para tratar este problema. Hay muchos trabajos que toman como punto de partida imágenes ráster, obtenidas a partir del escaneado de planos dibujados a mano, y aplicando técnicas de visión por computador, mientras que el número de trabajos que parten de planos en formato vectorial es más reducido. Además, la mayoría de los trabajos estudiados no incluyen pruebas con planos complejos ni dan información sobre el tratamiento de las situaciones más peliagudas. Las principales diferencias entre planos ráster y vectoriales son: (1) en planos ráster, la información no está estructurada en capas, mientras que en los planos vectoriales es frecuente utilizar capas para separar diferentes conceptos (por ejemplo: una capa para la representación de paredes, otra para mobiliario, etcétera); además (2) las medidas en los planos vectoriales son normalmente más precisas que en los planos ráster.

Los elementos semánticos que inicialmente se detectaban eran paredes y aberturas, con el fin de determinar la estructura de habitaciones y otros espacios cerrados. Hay otros elementos detectables en la estructura de los edificios, tales como cajas de escaleras y ascensores, pero con estos elementos se ha trabajado con menor profundidad.

La metodología seguida para el desarrollo de algoritmos para la detección de elementos semánticos a partir de planos arquitectónicos consistió en la abstracción de métodos automáticos emulando la forma en que un ser humano interpreta los planos. Así, una fase importante en el plan de trabajo fue el estudio y análisis del conjunto de planos disponibles, para así abstraer el proceso humano de razonamiento para el reconocimiento de elementos semánticos. Este primer análisis mostró la gran variedad de estilos y metodologías para el dibujado de planos, hecho que hizo difícil el desarrollo de algoritmos que funcionaran en todas las situaciones.

Otro obstáculo de partida fue el manejar los datos de las columnas, puesto que en algunos casos éstas aparecían como parte de la estructura de paredes, mientras que en otros estaban dispuestas en una capa aparte. En este último caso, las paredes pueden tener huecos si no se tienen en cuenta las columnas. Todas estas situaciones suponían un problema abierto, y se tuvieron en cuenta para establecer un conjunto mínimo de criterios a cumplir por los planos, de forma que fueran aplicables los algoritmos diseñados. De este modo, algunos planos necesitan pequeñas modificaciones para ajustarse a estos criterios. Esto se puede considerar un problema con dos vertientes: (1) qué es necesario asumir en lo que respecta a la estructura de los planos a la hora de diseñar los algoritmos, y (2) dónde centrar el esfuerzo, confiando en que los delineantes cumplan con los estándares sobre el dibujado de planos.

En una primera aproximación al problema, se consideraron situaciones donde las columnas estaban incluidas en la capa de paredes. Como resultado, se obtuvo el algoritmo de detección de elementos semánticos basado en reglas. Dado un plano con capas separadas para las paredes y las inserciones de bloques[4] para las aberturas (puertas y ventanas), se implementó un algoritmo iterativo que recorría las paredes, entendidas como pares de segmentos: en cada iteración se partía de una puerta, y cuando se llegaba a una intersección de paredes, se aplicaba una serie de reglas para determinar el tipo de intersección, así como la dirección a seguir en el recorrido, hasta llegar al punto de partida, momento en el que se co nsideraba una habitación detectada, y se iniciaba un nuevo recorrido en otra puerta. El algoritmo finalizaba cuando no hubiera más puertas que procesar.

La principal contribución del algoritmo descrito fue la detección de intersecciones de paredes incluso en los casos en que hay una columna implicada. Esta primera aproximación fue publicada en el Simposio Iberoamericano de Informática Gráfica de 2009[DGF09]. Sin embargo, este método tiene importantes desventajas: (1) el conjunto de reglas es complicado de caracterizar e implementar; (2) aparece sobreentrenamiento, es decir, el algoritmo funciona bien con casos de prueba similares a los utilizados para la elaboración de las reglas, pero el comportamiento es impredecible en otros casos; (3) el funcionamiento del algoritmo está muy influido por las imperfecciones en el plano.

De los problemas en el algoritmo basado en reglas se extraen las siguientes conclusiones: (1) es necesario un preprocesamiento de los planos para detectar y corregir las irregularidades; puesto que estas irregularidades son similares en la mayoría de los planos, es posible hacerlo automáticamente; (2) la detección de elementos semánticos ha de hacerse desde un punto de vista global, como un problema de geometría computacional en lugar de un procedimiento basado en reglas.

La detección de irregularidades se redujo al conjunto de casos más comunes, con el objetivo de centrar los esfuerzos en los algoritmos de detección. El algoritmo detecta y corrige irregularidades de manera iterativa, puesto que la corrección de una irregularidad puede dar lugar a otras nuevas, siendo difícil predecir cuántas iteraciones son necesarias para eliminar todas las irregularidades de un plano.

En lo que respecta al desarrollo de un algoritmo para la detección global de elementos semánticos, se analiza globalmente el conjunto de segmentos de la capa de paredes de un plano para encontrar representaciones de paredes con grosor que luego puedan completarse con las aberturas (puertas y ventanas) y las intersecciones entre paredes. Esto permite determinar cómo un plano está estructurado en habitaciones. Así, se buscan pares de segmentos paralelos y suficientemente cercanos, susceptibles de representar paredes. Para la representación de esta información, se desarrolló el concepto de grafo de adyacencia de paredes (wall adjacency graph, WAG), una de las principales contribuciones

[4] Como se verá más adelante, un bloque en CAD es un conjunto de primitivas que se pueden insertar cuantas veces se quiera en un dibujo, y que normalmente tiene un significado semántico, como la representación de una puerta o una ventana

de este trabajo.

La potencia de esta estructura también posibilita su uso para la detección de cajas de escaleras. Por ejemplo, puede adaptarse a la detección de cajas de escaleras con escalones rectangulares. Esta idea se basa en el hecho de que la detección de cajas de escaleras se puede caracterizar como la detección de pares de segmentos paralelos y suficientemente próximos.

Respecto a la búsqueda de aberturas, el primer algoritmo que se probó trataba de determinar las aberturas y sus paredes circundantes. Esta solución implica que: (1) la detección de aberturas ha de realizarse necesariamente después de la detección de paredes, y (2) en las zonas del plano donde el tamaño de las paredes es similar al tamaño de las aberturas, la detección de aberturas será inexacta.

La búsqueda de intersecciones entre paredes se trató con dos estrategias diferentes. Inicialmente, se planteó una solución basada en la aplicación de un algoritmo de clustering sobre los vértices obtenidos en la fase de detección de paredes, con la idea de encontrar los vértices suficientemente cercanos y susceptibles de formar una intersección. Posteriormente, este algoritmo se complementó con el algoritmo de crecimiento de segmentos, de forma que una vez que se han creado los clusters de puntos, se hace crecer los segmentos hasta que intersectan entre ellos.

Esta parte del trabajo está descrita en profundidad en la parte II de este documento.

1.3. Representación semántica de edificios

Entre las aplicaciones más comunes que utilizan información semántica de edificios se incluyen el catastro, la gestión de emergencias, la gestión del proceso de construcción, la navegación por interiores, la gestión del patrimonio cultural o la creación de entornos virtuales para videojuegos.

Dependiendo de la aplicación y de la fuente de datos utilizada, el nivel de detalle (LoD) y la forma de conseguir la información semántica pueden variar. Por ejemplo, el LoD necesario para el cálculo de las propiedades acústicas en una sala difiere del necesario para un recorrido por un museo virtual; en el primer caso se requieren datos físicos exactos, mientras que en el segundo prevalece la presentación de la información a los usuarios.

Respecto a las estructuras de datos para la representación de modelos 3D de edificios, hay distintas opciones. Por un lado, se puede modelar manualmente el edificio utilizando herramientas CAD; por otro lado, hay algoritmos automáticos (restringidos) para la generación automática de modelos de edificios reales o ficticios. Otras opciones incluyen el tratamiento de datos obtenidos a partir de escáneres LIDAR, fotografías, el escaneo de planos de las plantas (ráster) o planos CAD de las plantas (vectoriales). Como ya se indicó anteriormente, este trabajo está centrado en el procesamiento de planos vectoriales.

El ámbito de aplicación y las fuentes de datos determinan la representación final de los datos. Tal y como se indicó anteriormente, en este trabajo se planteó la representación de la información utilizando bases de datos es-

paciales. Sin embargo, también se estudiaron otros modelos, tales como X3D, GML, CityGML, COLLADA/KMZ, etcétera. Posteriormente, y especialmente tras una estancia en la TU Delft (Países Bajos), se dio más importancia a la representación basada en CityGML.

Por último, también se consideró interesante trabajar en un nuevo modelo de representación que combinara las ventajas de los modelos existentes con los resultados del proceso de detección de elementos semánticos, por lo que se planteó un esquema de representación a tres niveles, que incluye no sólo la información del plano original, sino también la información semántica y las relaciones entre ambos tipos de datos. Este esquema es tratado con detalle en la parte III

1.4. Implementación de una herramienta de usuario

Todo el trabajo de investigación desarrollado ha sido implementado en una herramienta de escritorio con dos objetivos: (1) disponer de un entorno software para integrar los nuevos algoritmos utilizando patrones de diseño, y (2) disponer de una interfaz gráfica para visualizar los planos, validar los resultados de los algoritmos e incluso hacer correcciones selectivas sobre los posibles errores. La idea era iniciar el desarrollo de un producto software que pudiera utilizarse en entornos CAD relacionados con la construcción. Este trabajo se describe con detalle en el apéndice A.

El resto del documento está organizado como sigue: la parte II trata la detección de semántica a partir de planos CAD. La parte III describe la propuesta de esquema de representación diseñada durante esta investigación; en este esquema se ha tratado de unir las bondades de los modelos de representación existentes en un único esquema, posibilitando a la vez la generación de modelos 3D en diferentes formatos a través de la incorporación de módulos de exportación. Por último, la parte IV detalla las conclusiones y el trabajo futuro.

En el apéndice A se describe el diseño e implementación de la aplicación utilizada como banco de pruebas para probar los algoritmos y validar los resultados. En el apéndice B se describe detalladamente la evolución de los algoritmos de clustering diseñados. Finalmente, el apéndice C resume los principales algoritmos geométricos utilizados en este trabajo, basados en el el libro *Geometric Tools for Computer Graphics*, de Philip J. Schneider y David Eberly [SE02].

Capítulo 2

Trabajo previo

En este capítulo se hace un repaso del trabajo previo sobre:

Reconstruccin automtica de edificios a partir de planos y de dibujos raster

Modelos de Informacin de Edificios y Sistemas de Informacin Geogrfica

Modelos geomtricos, topolgicos y semnticos 2D, 2.5D y 3D de representacin de edificios

Parte II

Detección de elementos semánticos

Capítulo 3

Diseño arquitectónico. Restricciones

Los planos utilizados como entrada para los algoritmos tienen que cumplir algunos requisitos mínimos de estilo para ser aptos para su procesamiento.

3.1. Dibujos arquitectónicos. Formatos de fichero

Una primitiva es la pieza más básica de información en un dibujo CAD. Ejemplos de primitivas son líneas, polilíneas, círculos, arcos, rectángulos, elipses, arcos elípticos o superficies rellenas. La tabla 3.1 resume las primitivas más comunes en diseño CAD 2D.

Los planos arquitectónicos pueden también estructurar sus primitivas usando dos tipos de agregaciones: capas y bloques. Aunque su estructura subyacente es similar (grupos de primitivas), su uso es sustancialmente diferente. Las capas se usan para agrupar de forma lógica conjuntos de primitivas con un campo de aplicación relacionado, mientras que los bloques se usan para crear objetos que se repiten en el dibujo.

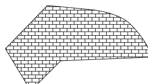
Respecto a los formatos de ficheros, nos centraremos en el Formato de Intercambio de Documentos (*Document Exchange Format*, DXF).

3.2. Estándares CAD. Requisitos de la estructura de los planos

A continuación describimos los requisitos de los planos para que se pueda garantizar la corrección de los algoritmos.

Deben existir al menos tres capas distintas: una capa de paredes, una capa de aberturas y una capa de escaleras. El nombre de estas capas no

Cuadro 3.1: Lista de primitivas en diseño CAD 2D y sus atributos.

Primitiva	Atributos	Representación
Línea	Cuatro valores reales (dos puntos) x_{start},y_{start},x_{end},y_{end}	
Polilínea	Conjunto ordenado de vértices. Cada vértice contiene un punto (x, y) y un valor de curvatura	
Círculo	Centro (x, y) y radio	
Arco circular	Centro (x, y), radio, ángulos inicial y final	
Rectángulo	Dos puntos esquina	
Elipse	Centro (x, y), ejes mayor y menor	
Arco elíptico	Centro (x, y), ejes mayor y menor, ángulos inicial y final	
Entramado	Define un patrón para una superficie cerrada.	

es relevante dado que serán seleccionadas por el usuario como parte del procesamiento semiautomático.

- Las paredes con grosor se deben representar como líneas paralelas. Estas líneas no tienen información sobre conectividad.
- Las aberturas (puertas y ventanas) se deben definir como bloques. Cada bloque que representa una abertura ha de estar alineado con los ejes de coordenadas locales mientras que sus instancias serán rotadas, trasladadas y escaladas. El nombre de los bloques no es relevante dado que el usuario los selecciona semiautomáticamente.
- Las columnas son parte de la capa de paredes.

3.3. Métodos locales vs. métodos globales

Hemos trabajado con dos enfoques para detectar la estructura de la planta de un edificio: local y global.

Los métodos locales tratan de reconstruir la estructura de un conjunto de paredes y aberturas recorriendo el camino formado por las paredes para encontrar intersecciones entre paredes o aberturas. Para detectar espacios cerrados, es necesario seguir un conjunto de reglas cada vez que el método llega a una posible intersección.

Los métodos globales tratan de reconstruir la estructura de la planta analizando el dibujo en su conjunto.

Los siguientes capítulos tratan con la detección y eliminación de irregularidades, lo que constituye un paso previo a los métodos locales y globales.

Capítulo 4

Detección y eliminación de irregularidades

Las irregularidades de un dibujo consisten en elementos geométricos visualmente indetectables que afectan al comportamiento de los algoritmos de procesamiento de geometría.

4.1. Tipos de irregularidades

Los tipos de irregularidades entre dos líneas son: solapadas, duplicadas, consecutivas y contenidas (Figura 4.1).

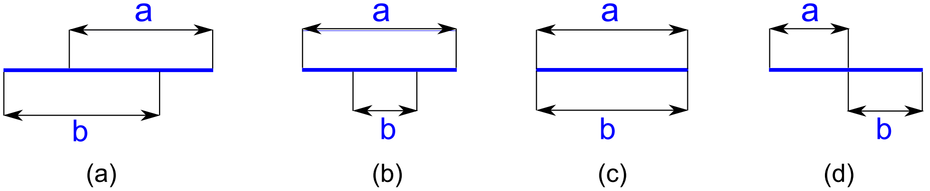


Figura 4.1: Casos de irregularidades: (a) Solapadas; (b) Contenidas; (c) Duplicadas; (d) Consecutivas

Por tanto, los cinco casos son longitud nula, solapados, contenidos, duplicados y consecutivos.

Los mismos casos, aplicados a arcos concéntricos, se obtienen comparando el ángulo final e inicial de cada arco en vez de la posición relativa de los puntos extremos.

4.2. Detección de irregularidades

Para detectar todas las irregularidades del dibujo, todas las primitivas se comparan entre ellas, dos a dos. Los tres pasos necesarios para determinar la posición relativa entre dos primitivas son:

1. Asegurarse de que ambas primitivas son comparables (segmentos colineales, o arcos concéntricos y con el mismo radio)

2. Convertir el problema de detección de irregularidades en un problema de comparación de rangos de números reales

3. Comparar los rangos de números y asignarle al par de primitivas el tipo de irregularidad encontrada

4.3. Eliminación de irregularidades

Las irregularidades etiquetadas como LONGITUD-NULA, CONTENIDAS Y DUPLICADAS se arreglan suprimiendo una de las dos primitivas involucradas.

Por otro lado, las irregularidades etiquetadas como SOLAPADAS y CONSECUTIVAS requieren la creación de una nueva primitiva a partir de las coordenadas de las primitivas que presentan la irregularidad.

Capítulo 5

Detección local de habitaciones

En este capítulo resumimos la detección de habitaciones basada en reglas, dado que fue el primer enfoque de la detección de semántica. Estaba relacionada con el objetivo de almacenar las principales características de una planta en una base de datos.

El *punto clave* es la entidad fundamental de la base de datos que permite organizar la información de un plano.

Se consideran varios tipos de puntos clave: punto clave de puerta, de ventana, con forma de T entre dos tabiques, con forma de L entre dos tabiques, con forma de T entre un tabique y un muro y variantes irregularidades de las distintas T y L.

5.1. Proceso de extracción de características

El primer paso del proceso es cargar y parsear el archivo DXF. Este paso requiere que el usuario seleccione los nombres de capas y bloques que tienen que ser considerados.

Una vez que las capas que contienen paredes, puertas y ventanas son identificadas, el sistema procesa las paredes, e inicialmente almacena (1) puntos, (2) líneas como índices a puntos y (3) una caja englobante orientada para cada bloque que representa una puerta o una ventana. Este es el punto de partida para el resto del proceso de detección.

5.1.1. Cálculo del grosor de paredes

El grosor de tabiques y muros exteriores se calcula usando como referencia los bloques que representan puertas y ventanas. Se considera como grosor de un tabique la moda de los grosores de las puertas interiores y como grosor de un muro la moda de los grosores de las ventanas exteriores.

5.1.2. Extracción de puntos clave, habitaciones y aberturas. Reglas de detección

Una vez que el grosor de los tabiques y de los muros exteriores se calcula, los siguientes pasos son la detección de puntos clave, habitaciones y ventanas. Hemos desarrollado un algoritmo para efectuar estos pasos en un proceso basado en un conjunto de reglas que identifican las situaciones típicas.

5.1.3. Almacenamiento de datos

Durante el proceso explicado anteriormente, la información sobre puntos clave, puertas, ventanas y habitaciones se almacena en las correspondientes tablas de la base de datos.

5.2. Generación de salida 3D

Una vez que el plano se ha procesado, se puede generar una escena 3D utilizando los datos almacenados en la base de datos. Hemos usado el formato X3D [BD07], aunque el diseño de la aplicación donde han incorporado los algoritmos permite la inclusión de módulos para almacenar la escena usando otros formatos de fichero como COLLADA [AB06] o VRML.

5.3. Resultados

Los resultados obtenidos para este enfoque consisten en la creación de modelos X3D mediante la extrusión de los datos detectados y almacenados en la base de datos.

El sistema ha sido probado con planos arquitectónicos reales de la Universidad de Jaén en formato DXF (por ejemplo, ver Figura 5.1).

Usando los datos almacenados en la base de datos, se generan escenas 3D y se salvan en formato X3D. La Figura 5.2 muestra una ampliación de la vista de una parte del plano, junto con el modelo 3D resultante.

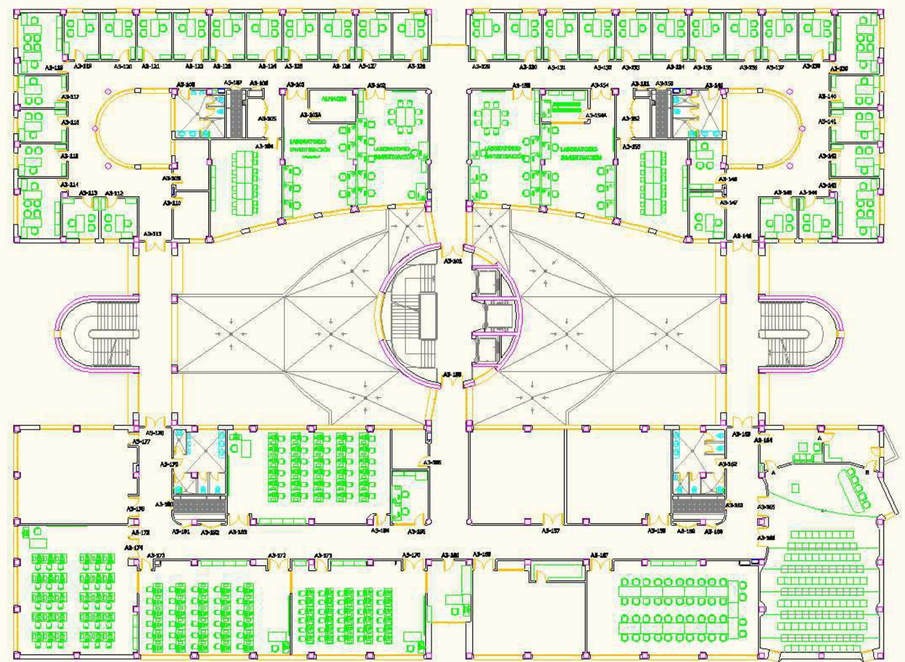


Figura 5.1: Uno de los planos utilizado en las pruebas

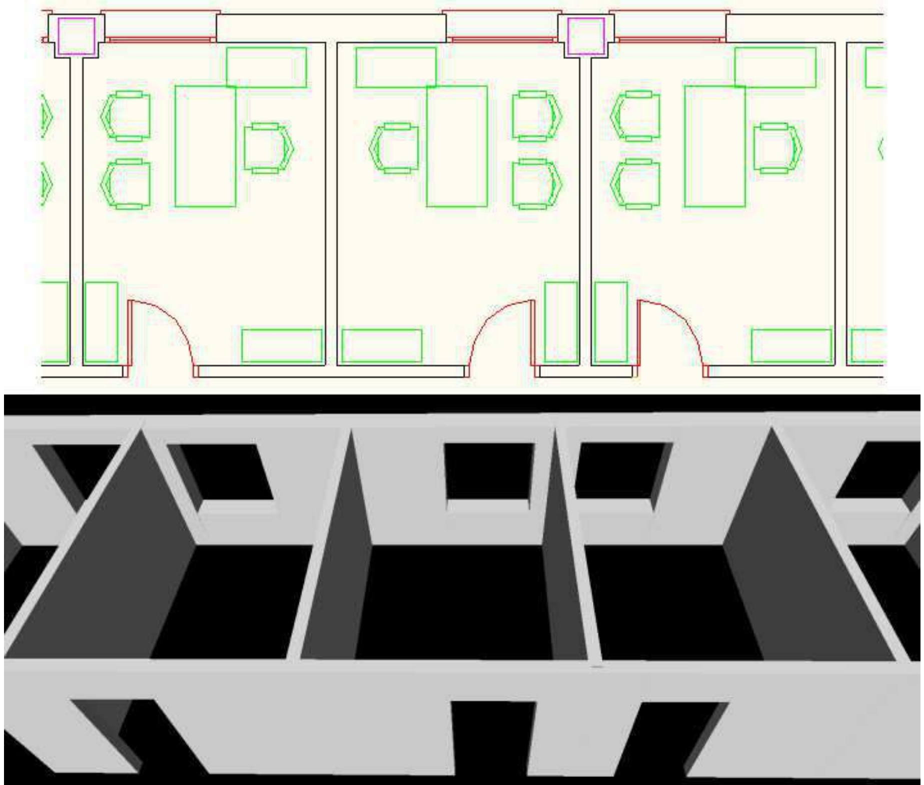


Figura 5.2: Plano y su correspondiente modelo 3D

Capítulo 6

Detección global de habitaciones

Hemos concluido que es necesario tratar con el problema de detección de habitaciones usando otro enfoque más independiente de los casos de diseño concretos. Como resultado del proceso de reconocimiento global, es posible obtener un grafo de topología de la planta.

Definición 1 (Grafo de topología) *Un grafo de topología es aquel en el que cada nodo representa el punto 2D de un plano en el que intersecan dos paredes, o una pared y una abertura, y cada lado representa una pared o una abertura.*

El reconocimiento global de elementos para construir el grafo de topología de una planta se puede descomponer en varias fases:

- Detección de paredes: Este problema consiste en inspeccionar todas las primitivas que pertenecen al dibujo de las paredes para detectar paredes con grosor.
- Detección de aberturas: En esta fase, las aberturas son buscadas en el dibujo del plano. Para cada abertura encontrada, se detectan sus paredes adyacentes.
- Buscar los puntos de unión entre paredes: antes de esta etapa, el grafo de topología no conexo representa las paredes y aberturas pero le faltan los puntos de unión entre paredes.

Las tres etapas enumeradas anteriormente nos permiten obtener el grafo de topología. Una siguiente fase consiste en relacionar el grafo de topología con la geometría inicial, para lo que se siguen estos tres pasos:

- Buscar polígonos externos: sobre el grafo de topología, podemos encontrar circuitos que no contienen a ningún otro circuito. El principal objetivo de esta búsqueda es reconocer todas las habitaciones y pasillos que forman una planta.

Asignar las líneas de paredes a los polígonos externos: las líneas que representan paredes en la geometría inicial pueden ser asignadas a las habitaciones detectadas usando el test de pertenencia de punto a polígono.

- Buscar polígonos internos: en esta fase, partiendo de las líneas de pared de la geometría original que han sido asignadas a cada habitación, los polígonos internos son construidos.

6.1. Detección de paredes

Consideramos dos tipos de paredes: (1) paredes con forma rectangular, hechas en el dibujo con segmentos paralelos de línea recta; y (2) paredes curvas, hechas en el dibujo con arcos circulares concéntricos.

El proceso de detección de paredes implica buscar pares de segmentos *candidatos a pared* usando dos umbrales min y max (grosor mínimo y máximo de las paredes, automáticamente estimado como se mostró en el Capítulo 5), y dividirlos para obtener pares *pared*. Las definiciones de par candidato a pared y par pared son las siguientes:

Definición 2 (Par de segmentos de recta candidatos a pared) *Sean a y b segmentos de recta en $\mathbb{R}^2$. Sean l y m las rectas que contienen a y b respectivamente, y a' y b' las proyecciones de a y b sobre m y l. Fijados dos umbrales min y max, el par (a, b) es candidato a pared (lo que se representa con el predicado $prone(a, b, min, max)$) si y sólo si se cumplen todas estas condiciones:*

C1. a y b son paralelos: $a \parallel b$

C2. a' y b (consecuentemente b' y a) se solapan: $a' \cap b \neq \emptyset$

C3. La distancia entre l y m está entre ambos umbrales: $d(l, m) \in [min, max]$

Nota: En este contexto los segmentos se consideran abiertos, para evitar el caso especial donde dos segmentos con proyecciones consecutivas cumplen la condición C2.

Definición 3 (Par de segmentos de recta pared) *Dos segmentos a y b se dice que forman un par pared, fijados dos umbrales min y max (lo que se representa con el predicado $wall(a, b, min, max)$), si satisfacen las condiciones para ser un par candidato a pared, y sus proyecciones en la recta que contiene al segmento contrario coinciden. Por tanto, una condición nueva y más restrictiva se añade a C1, C2 y C3:*

C4. a' y b (y por tanto b' y a) son el mismo segmento: $a' = b$

Dadas las definiciones anteriores, estos son los puntos principales del algoritmo que detecta las paredes representadas por un conjunto de segmentos en un plano:

(1) Encontrar todos los pares candidatos a pared.

(2) Para cada par candidato a pared, aplicar los siguientes pasos:

   (2.a) Calcular las proyecciones de los extremos de un segmento en el otro segmento.

   (2.b) Dividir cada segmento por las proyecciones calculadas y comprobar la existencia de pares pared en el nuevo conjunto de segmentos: algunos segmentos formarán pares pared mientras que el resto pasarán a estar desemparejados (sus proyecciones no se solapan).

   (2.c) Actualizar el conjunto de segmentos eliminando los segmentos antiguos y añadiendo los desemparejados. Después, actualizar el conjunto de pares candidatos a pared.

Para facilitar el procesamiento de las relaciones pared y candidato a pared entre segmentos, y registrar las relaciones jerárquicas entre cada segmento y los fragmentos en que se ha dividido, proponemos el Grafo de Adyacencia de Paredes (GAP) como una estructura de datos que da soporte a esto.

6.1.1. Grafo de Adyacencia de Paredes (GAP)

El Grafo de Adyacencia de Paredes (GAP) es un grafo cuyos nodos representan los segmentos de un plano que pertenecen a las capas de paredes, y cuyos lados representan las relaciones entre dichos segmentos.

Como un paso preliminar, es necesario construir el GAP inicial para los segmentos del plano. Una vez que el GAP ha sido creado, los pares candidatos a pared se procesan secuencialmente para obtener pares pared. Hay nueve posibles tipos de pares candidatos a pared, dependiendo de la situación relativa de los segmentos del par. Cuando el proceso se ha completado, solamente quedarán pares pared en el GAP. La siguiente lista muestra algunas observaciones sobre el tiempo de ejecución del algoritmo:

1. La ejecución del algoritmo siempre termina, ya que en cada iteración se elimina un lado candidato a pared, y el resto de lados candidatos a pared que inciden en los nodos que están conectados por el lado que se elimina son reemplazados por la misma cantidad de lados candidatos a pared. Por tanto, el número de iteraciones es igual al número de lados candidatos a pared, y en consecuencia, el tiempo de ejecución del algoritmo es de $O(n)$ con respecto al número inicial de lados candidatos a pared. Las operaciones ejecutadas en cada iteración son $O(1)$, dado que no dependen del tamaño del conjunto de lados candidatos a pared.

2. La inicialización del GAP es $O(n^2)$ con respecto al número de segmentos en el plano, dado que todos los pares de segmentos tienen que ser analizados.

6.1.1.1. Grafo de Adyacencia de Paredes Generalizado (GAPG)

El algoritmo de detección de paredes se basa en el supuesto de que no hay más de dos segmentos paralelos que estén lo bastante cerca como para ser considerados pares candidatos a pared. No obstante, hay situaciones donde más de dos segmentos paralelos pueden aparecer, haciendo que el algoritmo no maneje esta situación de manera adecuada y produciendo inconsistencias en la ejecución.

Por tanto, la estructura del GAP y el algoritmo fueron mejorados para permitir que para cada segmento pueda haber más de una posible partición. Esta estrategia nos lleva a introducir el GAP Generalizado (GAPG).

6.1.1.2. GAP Y GAPG para paredes curvas

En este punto describimos los cambios necesarios en el algoritmo de detección de paredes para procesar paredes definidas como pares de arcos circulares.

La Tabla 6.1 resume los criterios para formar pares de segmentos de recta candidatos a pared, y muestra cómo tienen que ser adaptados para los arcos circulares.

Segmentos de recta	Arcos circulares
Los segmentos deben ser paralelos	Los segmentos deben compartir el mismo centro de círculo
La distancia entre segmentos debe estar por debajo de un umbral ε	La diferencia entre los radios de los círculos deben estar por debajo de un umbral ε
La proyección de un segmento en la recta que contiene al otro se solapa con dicho segmento	Los intervalos angulares que definen los segmentos de arco se solapan

Cuadro 6.1: Comparación de los criterios para formar pares candidatos a pared entre arcos rectos y circulares

6.2. Detección de aberturas

El segundo paso en la detección global de habitaciones es la detección de aberturas. En esta etapa, las aberturas se localizan en el dibujo de la planta: para cada abertura encontrada, se buscan sus paredes colindantes en el grafo de topología, y nuevos lados representando las aberturas se añaden a dicho grafo de topología.

6.2.1. Detección de aberturas a partir de instancias

El objetivo es obtener, para cada instancia de abertura, qué lados del grafo de topología están a los lados de la abertura representada por la instancia. Para

ello, se calcula la caja englobante de cada abertura y se hace una búsqueda en el grafo de topología obtenido tras la detección de paredes para localizar los dos vértices más cercanos a ambos lados de dicha caja englobante. Al grafo de topología se añade un lado de tipo abertura que une dicho par de puntos.

6.2.2. Detección de aberturas a partir de primitivas

El algoritmo de detección de aberturas explicado anteriormente asume que las aberturas están definidas como instancias de bloques. Además, los bloques deben cumplir algunos requisitos, tales como estar alineados con los ejes X e Y. Sin embargo, en algunos casos de prueba, las aberturas no aparecen representadas como bloques sino como conjuntos de primitivas (líneas y arcos). Un proceso automático para detectar las aberturas en estos casos consistiría en agrupar las primitivas que intersecan y considerarlas como una abertura.

Una vez que las aberturas han sido detectadas, es necesario un paso más: calcular las cajas englobantes. A diferencia del algoritmo a partir de instancias, la orientación de las cajas englobantes no se puede determinar fácilmente.

Este problema se ha dejado como parte del trabajo futuro, junto con el análisis de otras características y anormalidades que puedan ser encontradas en los casos de estudio.

6.3. Clustering

El último paso en la construcción del grafo de topología es la búsqueda de los puntos de unión entre paredes.

Estamos hablando de un problema de clustering de puntos, y la solución se diseña teniendo en cuenta este enfoque. El problema de clustering de puntos ha sido ampliamente tratado en la bibliografía.

Durante nuestra investigación, los algoritmos de clustering han experimentado una importante evolución. Para simplificar este documento, vamos a introducir en este capítulo los dos últimos algoritmos diseñados: comparación de los extremos de las componentes conexas del grafo de topología y el algoritmo de crecimiento de líneas.

El pseudocódigo de la primera se muestra en el Algoritmo 6.1.

Algoritmo 6.1: Clustering utilizando los extremos de las secuencias

```
   Input: G=(V,E): El grafo que contiene los lados de paredes y aberturas
   Output: G'=(V',E'): el grafo G=(V,E) despus del clustering de vértices
 1 begin
 2     Inicializar un cluster de vértices vertexClusters[i] para cada vértice de
       V
 3     Inicializar un cluster de secuencias sequenceClusters[i] para cada
       componente conexa de E
 4     for i = 0; i < sequenceClusters.size; i++ do
 5         for j = i + 1; j < sequenceClusters.size; j++ do
 6             for k = 0; k < sequenceClusters[i].size; k++ do
 7                 for l = 0; l < sequenceClusters[j].size; l++ do
 8                     (P_min,Q_min) ← Puntos más cercanos entre los cuatro
                       pares formados tomando un punto de cada secuencia
                       actual
 9                     min ← d(P_min,Q_min)
10                 end
11             end
12             if min < ε then
13                 sequenceClusters[i] ← sequenceClusters[i] ∪
                   sequenceClusters[j]
14                 Eliminar sequenceClusters[j] de sequenceClusters
15                 merged ← true
16                 Mezclar los clusters que contienen P_min y Q_min
17             end
18         end
19     end
20     foreach vertexClusters[i] in vertexClusters do
21         centroids[i] ← SmartCentroid(vertexClusters[i])
22     end
23     V' se construye a partir de V reemplazando cada vértice por el
       centroide inteligente del cluster de vértices
24     E' se construye a partir de E reemplaando los lados adyacentes a cada
       vértice por el centroide de su cluster de vértices
25     return G'=(V',E')
26 end
```

6.3.1. Algoritmo de crecimiento de líneas

En la última etapa de la investigación hemos diseñado un algoritmo cuyos objetivos eran:

1. Simplificar el número de casos del anterior enfoque
2. Generalizar las diferentes situaciones de uniones entre paredes de una for-

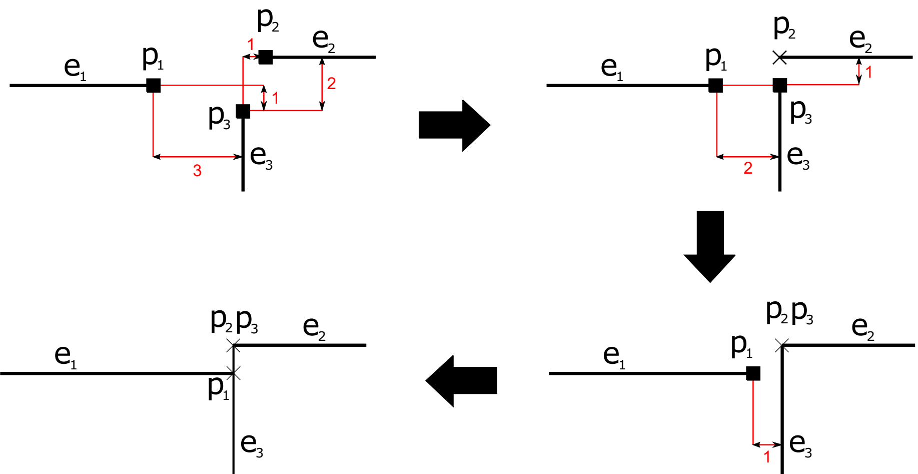


Figura 6.1: Pasos de la ejecución del algoritmo de crecimiento de líneas en un cluster con tres puntos. Las distancias aparecen dibujadas en rojo

ma simple

Dicho algoritmo lo hemos denominado *algoritmo de crecimiento de líneas*. Su idea básica es simular que los extremos de un mismo cluster avanzan a lo largo del camino determinado por las rectas que lo contienen por turnos. Cuando un extremo alcanza alguna de las otras líneas, decimos que ha alcanzado su posición final, y el algoritmo continúa interando sobre el resto de extremos, hasta que todos han alcanzado su posición final (Figura 6.1). Este algoritmo (Algoritmo 6.2) se implementa utilizando una matriz de distancias que almacena la distancia de cada extremo de un mismo cluster, a lo largo de la distancia que lo contiene, a cada una de las líneas que corresponden con los otros extremos de dicho cluster.

El algoritmo resuelve correctamente las intersecciones de paredes con forma de L, X y T, y puede manejar casos más complejos, como líneas que no convergen en un solo punto o líneas que no son perpendiculares.

6.4. Resultados

Los algoritmos de esta parte han sido probados en un ordenador con un procesador Intel® Core™2 Quad de 2.4 Ghz, y con 4 GB de RAM. Para las pruebas se han utilizado planos reales de edificios de nuestra universidad y de otros edificios privados.

6.4.1. Detección de paredes

La Figura 6.2 muestra tres de los planos usados y el resultado de aplicar la detección de paredes sobre ellos; las paredes detectadas están dibujadas en azul.

Algoritmo 6.2: Algoritmo de crecimiento de líneas

```
    Input: P1,P2,...,Pn: extremos de un cluster;
    L1,L2,...,Ln: líneas que contienen a Pi
    Output: P1,P2,...,Pn: extremos después del crecimiento de líneas
 1  begin
 2      Inicializar una matriz {aij}n×n tal que aij es la distancia de Pi a Lj a lo
        largo de Li (∞ si Li y Lj son paralelas)
 3      while exista aij tal que aij ≠ 0 and aij ≠ ∞ do
 4          m ← el mínimo aij ≠ 0 de la matriz
 5          foreach punto Pi do
 6              if la fila i contiene valores distintos de 0 and ∞ then
 7                  Mover Pi m unidades hacia Lj a lo largo de Li
 8              end
 9          end
10          foreach aij en la matriz tal que aij ≠ 0 and aij ≠ ∞ do
11              aij ← aij- m
12          end
13      end
14      return P1,P2,...,Pn
15  end
```

La Tabla 6.2 muestra resultados numéricos obtenidos en nuestras pruebas utilizando diferentes valores de umbral (los resultados aparecen en el mismo orden de los mostrados en la Figura 6.2).

El tiempo empleado en la construcción del GAP inicial está en general por debajo de un segundo para planos reales con no más de 1700 segmentos, aunque la complejidad del dibujo tiene una influencia significativa en este algoritmo.

6.4.2. Construcción del grafo de topología

La Figura 6.3 muestra la representación de la topología de una parte de un plano real tal y como se obtiene con los algoritmos descritos en este trabajo, mientras que la Figura 6.4 muestra la representación de topología superpuesta con el plano.

La Tablae 6.3 recoge los resultados obtenidos en algunos tests con planos reales. Los casos de prueba *A3* y *C5* corresponden a edificios del campus de la universidad; el caso de prueba *House* corresponde con una vivienda unifamiliar. Otros casos de prueba son *Basic lift*, *Planta tipo*, *Vilches*, *Bayenga* y *Heating*.

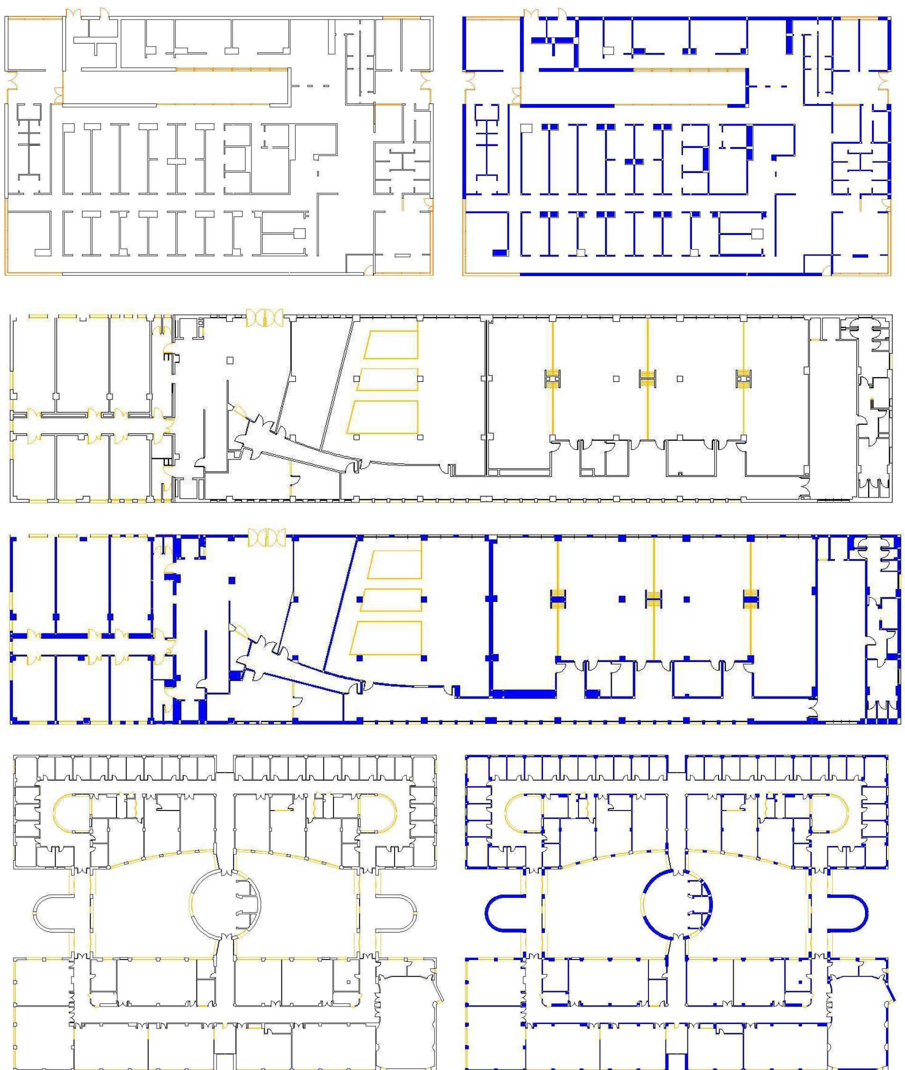


Figura 6.2: Planos reales utilizados para probar nuestros algoritmos, junto con los resultados de aplicar el algoritmo de detección de paredes. Las paredes detectadas están dibujadas en azul.

LEYENDA
A1-A2. Número de segmentos/arcos en el plano
ε. Umbral usado para formar pares de segmentos
B1-B2. Número inicial de lados candidatos a pared en segmentos y arcos respectivamente
C1-C2. Número inicial de lados pared en segmentos y arcos respectivamente
D1-D2. Número final de lados pared en segmentos y arcos respectivamente
E1-E2. Segmentos/arcos que no forman paredes (%)
F. Tiempo de construcción del GAP (milisegundos)
G. Tiempo de procesamiento del GAP (milisegundos)

Plano	A1	A2	ε	B1	B2	C1	C2	D1	D2	E1	E2	F	G
1	643	0	0.4	193	0	56	0	249	0	36.10	0	118.50	1.09
			0.5	199	0	58	0	257	0	35.00	0	119.17	1.10
			0.6	203	0	60	0	263	0	33.90	0	122.03	1.16
			0.7	236	0	62	0	298	0	26.59	0	136.92	1.35
2	1148	14	0.4	482	9	159	0	641	9	32.50	11.10	612.21	3.99
			0.5	549	9	206	0	755	9	25.10	11.10	755.40	5.08
			0.6	628	9	244	0	872	9	16.00	11.10	937.65	6.38
			0.7	692	9	287	0	979	9	10.10	11.10	1161.43	8.37
3	1678	79	0.4	442	20	93	0	535	20	47.73	54.43	696.45	4.45
			0.5	455	20	104	0	559	20	45.76	54.43	785.69	4.91
			0.6	637	33	176	0	813	33	26.04	26.58	922.89	5.21
			0.7	667	33	190	0	857	33	22.76	26.58	957.27	5.56

Cuadro 6.2: Datos numéricos de las pruebas de la detección de paredes con planos reales

	C1	C2	C3	C4	C5	C6	C7		C8	C9
							C7.1	C7.2		
House	92	15	33	4	0	4	0	3	0,00 %	100,00 %
A3	1769	196	594	72	10	84	0	0	13,89 %	73,81 %
				80	0	84	30	10	0,00 %	95,24 %
C5	1150	156	359	62	2	67	12	0	3,23 %	89,55 %
				66	1	67	12	4	1,52 %	97,01 %
Basic lift	162	16	62	7	0	9	0	1	0 %	78 %
Planta tipo	110	15	42	8	2	9	0	3	25 %	67 %
	110	15	42	9	1	9	0	5	11 %	89 %
Vilches	146	20	56	10	0	10	0	2	0 %	100 %
Bayenga	100	17	38	6	0	6	0	3	0 %	100 %
Heating	144	16	57	7	0	10	0	6	0 %	70 %

Cuadro 6.3: Resultados de las pruebas en edificios reales. Las columnas representan: (C1) número de líneas y arcos pared en el dibujo (C2) número de aberturas (C3) número de paredes detectadas (C4) número de habitaciones detectadas (C5) número de habitaciones detectadas incorrectamente (C7) número de ediciones manuales (C7.1) número de paredes detectadas manualmente (C7.2) número de lados del grafo añadidos manualmente (C8) ratio (habitaciones incorrectas/habitaciones detectadas) (C9) éxito, calculado como (habitaciones detectadas - habitaciones incorrectas) / habitaciones reales

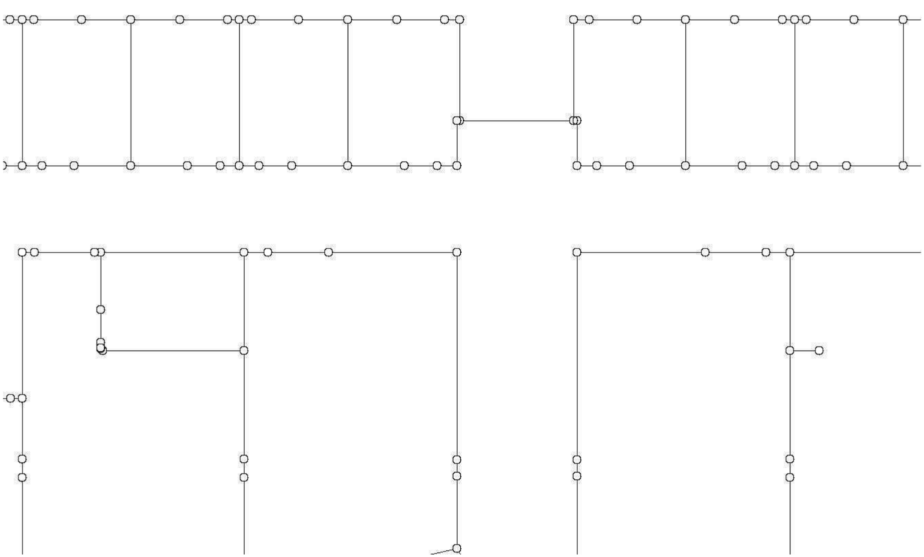


Figura 6.3: Representación de la topología de parte de un plano CAD vectorial

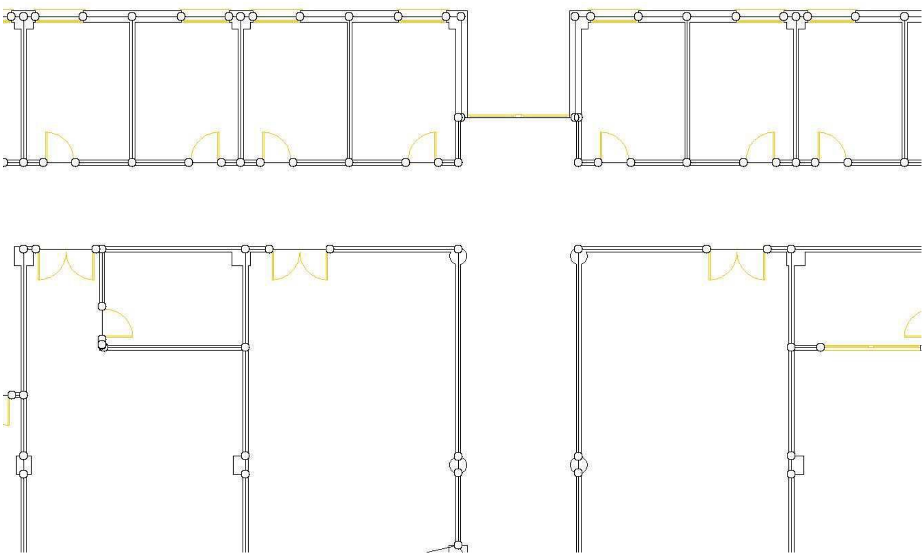


Figura 6.4: Representación de la topología superpuesta al plano CAD vectorial

Parte III

Representación de información de edificios

Esta parte del documento trata la creación de modelos topológicamente correctos 2D y 3D. Este aspecto se introduce de acuerdo con dos puntos de vista: por un lado, introduciremos una serie de algoritmos que crean la topología de la planta a partir de la información obtenida en la parte II; por otra parte, discutiremos algunos aspectos sobre la corrección topológica del modelo, poniendo énfasis en la importancia de esta propiedad.

La estructura básica que se desarrollará en esta parte del documento consiste en un grafo de topología compuesto por: (1) un conjunto de lados, cada uno representando una pared o una abertura y (2) un conjunto de vértices que representan uniones entre paredes, o entre una pared y una abertura.

Esta parte se estructura de la siguiente forma: el Capítulo 7 propone un modelo unificado que cumple gran parte de los requisitos citados en la literatura y complementa la información obtenida por los algoritmos de la Parte II del documento. El Capítulo 8 introduce los algoritmos necesarios para obtener información topológica desde el grafo de topología y enlazar la información geométrica con la topológica. El Capítulo 9 trata la triple extrusión, un algoritmo que obtiene una representación 3D de un edificio a partir de sus vistas de planta, alzado y perfil. Finalmente, el Capítulo 10 describe la generación de modelos de salida a partir de la arquitectura presentada en esta parte, tales como CityGML, entre otros modelos 3D.

Capítulo 7

Estructura de tres niveles para la representación de modelos topológicamente correctos

En este capítulo, proponemos una estructura que representa interiores de edificios con las siguientes características:

1. El modelo cubre diferentes niveles: desde la geometría de los diseños arquitectónicos hasta la información semántica y topológica sobre la distribución estructural y la conectividad entre espacios físicos.

2. El modelo contiene información 2D y 3D topológicamente correcta.

3. El modelo tiene distintos niveles de detalle (LoD). Incluye el esbozo de la representación 3D Ide un edificio inferido por un algoritmo llamado *triple extrusión*.

4. Todas las partes del modelo deben estar relacionadas, de manera que la información de os distintos niveles pueda ser mapeada eficientemente..

5. Otros modelos de información deben ser fácilmente derivados a partir de nuestro modelo.

7.1. Análisis comparativo y discusión

En la parte anterior, una serie de artículos de la literatura fueron revisados y clasificados de acuerdo con la dimensión utilizada para representar datos. Además, algunas consideraciones sobre geometría, topología y semántica han sido analizados en cada artículo.

En esta sección propondremos un modelo concebido para servir como enlace entre la geometría de bajo nivel, la topología y la semántica. Tras eso, compararemos los trabajos revisados y discutiremos sobre la aplicabilidad de los modelos citados a diferentes áreas.

La mayoría de los trabajos sobre Building Information Modeling, Catastro 3D, validación de modelos, etc., considera tres niveles de información de edificios, a saber, geometría, topología y semántica.

La topología implica la gestión de las relaciones entre las entidades geométricas, por ejemplo la adyacencia entre caras de un poliedro o los puntos como fronteras de las líneas. Un aspecto clave es mantener la consistencia topológica frente a los cambios en la geometría. En este sentido, Ledoux y Meijers [LM09] tratan el concepto de consistencia topológica, fijando tres condiciones para que un conjunto M de objetos 2D sea consistente:

- Cada línea de M está formada por dos puntos de M
- La intersección de dos líneas de M es o bien el conjunto vacío o bien un punto de M
- La intersección del interior de un polígono y cualquier otra primitiva de M es vacía.

Para espacios topológicos 3D, se debe cumplir una condición adicional para asegurar la consistencia: la intersección entre el interior de un poliedro y cualquier otra primitiva ha de ser vacía. Justificaremos que los diferentes modelos que proponemos son topológicamente correctos.

Con respecto a la semántica, algunos conceptos de interés para los modelos de edificios son la identificación de paredes, puertas, ventanas y habitaciones. La estructura topológica de los elementos semánticos se puede deducir y validar a partir de las relaciones topológicas entre elementos geométricos, de acuerdo con las definiciones de conectividad y adyacencia:

Presentamos un resumen de las principales características de los artículos revisados en la Tabla 7.1:

Cuadro 7.1: Comparación entre los trabajos revisados.

Grupo	Trabajo	Geometría	Topología		Semántica
			Conectividad	Adyacencia	
Modelos 2D	Franz et al. [FMW05]	-	Implícito	Explícito	RO, O
	Lamarche y Donikian [LD04]	Implícito	Explícito	-	PA, CR
	Plümer y Gröger [PG96]	V, E	-	Implícito	-
	Stoffel et al. [SLO07]	V, E, R	Explícito	Implícito	RO, O, S
	Li et al. [LCR10]	DC	-	-	T
	Zhi et al. [ZLF03]	V, E, R	Explícito	Implícito	R, O
	Hahn et al. [HBW06]	Implícito	Explícito	Implícito	RO, O, S
	Merrell et al. [MSK10]	Implícito	Explícito	Implícito	RO-tipado, O
Modelos 2.5D	Slingsby y Raper [SR07]	Implícito	-	Implícito	W, L, O
	Tutenel et al. [TBSdK09]	Implícito	-	-	RO
	Germer y Schwarz [GS09]	Implícito	-	-	RO
	Van Dongen [vD08]	Cubos	-	-	W, C
	Choi et al. [CKHL07]	-	Explícito	Implícito	RO, O, S, W
Modelos 3D	Choi y Lee [CL09]	R	Explícito	Explícito	RO, O, CO
	Clemen y Gielsdorf [CF08]	V, E, F, P	Explícito	Explícito	-
	Van Berlo y Laat [vBdL10]	IFC, CityGML	Explícito	Implícito	RO, O, S, W
	Paul y Bradley [PB03]	V, E, F, VO	Explícito	Explícito	W, C
	Billen y Zlatanova [BZ03]	V, E, F, VO	Implícito	Explícito	Edificios
	Hagedorn et al. [HTGD09]	GML	Explícito	Explícito	RO, O, S, W
	Van Treeck y Rank [vTR07]	B-rep	Explícito	Explícito	RO, W
	Borrmann y Rank [BR09]	VO	-	-	Edificios
	Isikdag et al. [IUA08]	Implícito	Implícito	Implícito	O, S, W
	Boguslawski et al. [BG10]	V, E, F	-	Explícito	-
	Yan et al. [YCG10]	BIM	BIM	BIM	BIM
	Xu et al. [XZZ10]	F, VO	Explícito	Explícito	RO, O, S, W, C
Nuestra propuesta		V, E	Explícito	Explícito	RO, O, PA, S, W, C

Leyenda: V=vértices, VO=volúmenes, E=lados, F=caras, P = planos, R=regiones
DC=Celdas discretas, RO=habitaciones, O=aberturas, PA=pasajes, CR=cruces
S=plantas, T=etiquetas, W=paredes, C=techos, L=ascensores, CO=pasillos

Presentamos (1) un modelo de tres niveles para la representación de edificios 3D que contiene información sobre dibujos CAD y estructura topológica de los interiores de los edificios y (2) algoritmos para generar y mapear la información manteniendo la corrección topológica.

La Figura 7.1 presenta un resumen de dicho modelo:

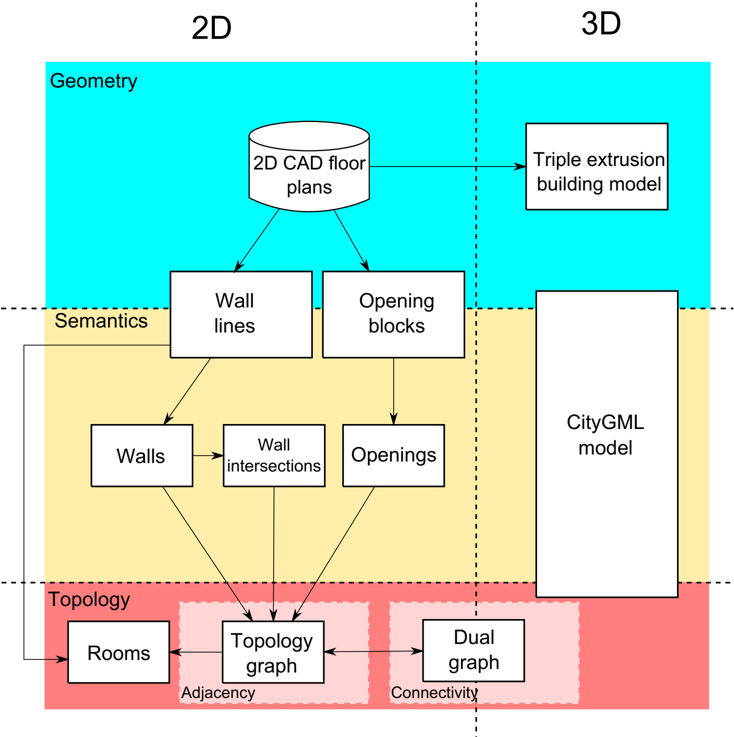


Figura 7.1: Modelo basado en una arquitectura de tres niveles para representar modelos de información de edificios.

7.2. Módulo de geometría

El módulo de geometría contiene información sobre dibujos CAD y los sólidos obtenidos a partir de ellos usando un algoritmo llamado de triple extrusión.

Las líneas de paredes y los bloques de aberturas se obtienen del plano 2D después de seleccionar por parte de un usuario las capas y bloques correspondientes, que contienen una primera componente de información semántica.

Por otra parte, los sólidos obtenidos desde los planos CAD usando la triple extrusión se representan mediante modelos B-rep de poliedros.

7.3. Módulo de semántica

El módulo de semántica alberga el resultado de la extracción de la información relevante para representar la estructura de los edificios, esto es, paredes y aberturas.

La representación CityGML contiene fundamentalmente elementos semánticos sobre la base de la descripción geométrica de GML.

7.4. Módulo de topología

Este módulo representa la estructura y topología de las plantas de un edificio. Su principal componente es una estructura de datos basada en un grafo donde: (1) Los nodos representan los extremos de los trozos de pared; (2) los lados representan aberturas y trozos de pared que conectan dos extremos. Cada lado incluye información sobre su tipo (*puerta*, *ventana* o *pared*).

Un grafo dual puede derivarse del grafo topológico para representar las habitaciones y la adyacencia/conectividad entre ellas.

Capítulo 8

Topología 2D. Procesamiento del plano CAD

En este capítulo tratamos la construcción de grafos de topología 2D que representan las plantas de un edificio y el enriquecimiento de la topología con información geométrica de los planos CAD.

8.1. Construcción el grafo de topología

El método para construir el grafo de topología a partir de diseños CAD filtrados fue descrito en la Parte II de la tesis. En esta parte tratamos la corrección topológica de la representación obtenida y proponemos pasos adicionales que completan los algoritmos previos y asegurar la consistencia.

Para evitar inconsistencias topológicas en el paso de detección de paredes, debemos asegurar que las paredes no intersecan. Para esto, una vez que todas han sido detectadas, procesamos el conjunto para eliminar los rectángulos que intersecan.

En la detección de aberturas, dos condiciones son suficientes (pero no necesarias) para garantizar la consistencia: (1) evitar intersecciones entre cajas englobantes y (2) evitar intersecciones entre una caja englobante y los lados del grafo de topología (excepto los dos lados de tipo pared adyacentes a la abertura representada por dicha caja englobante). En la práctica, estas condiciones sólo se incumplen en casos degenerados.

Las habitaciones y los pasillos normalmente corresponden con espacios cerrados en la representación topológica. El cálculo del grafo dual del grafo topológico permite la detección de habitaciones y pasillos en un plano. Para extraer regiones cerradas a partir de un grafo no conexo cuyos nodos son puntos 2D y sus lados son segmentos 2D, utilizamos el algoritmo MAFL [ZLF03].

El resultado del algoritmo es un conjunto de polígonos que representan los espacios cerrados y un polígono que representa el exterior. Todos los vértices de los polígonos que representan espacios cerrados se ordenan en sentido horario, mientras que el exterior se orienta en sentido anti-horario. Cada lado está compartido por dos espacios cerrados, o por un espacio cerrado y el exterior.

La corrección topológica del grafo, de acuerdo con las condiciones de Ledoux y Meijers, pueden ser demostradas.

8.2. Enlazando información geométrica y topológica

El siguiente paso es el enlace entre los datos topológicos y los datos geométricos de los planos de entrada. En esta subsección introducimos algunos elementos de información que enlazan el conjunto de líneas de paredes y el módulo de topología.

- Cada segmento o arco del conjunto de líneas de paredes puede ser asignado a un espacio cerrado usando tests de pertenencia de puntos a polígonos.

  Para un determinado espacio cerrado, los segmentos asignados a él forman un conjunto de polilíneas que pueden ser cerradas añadiendo nuevos segmentos para las puertas y ventanas. Como resultado, tenemos un *polígono interior* para cada espacio cerrado y un *polígono exterior* para el anillo exterior. Para esto utilizamos el llamado algoritmo de rejilla.

La Figura 8.1 muestra un ejemplo del algoritmo.

8.3. Resultados

La Figura 8.2 muestra (a) un plano; (b) los polígonos exteriores y (c) los polígonos interiores detectados después de asignar los segmentos a los espacios cerrados y de cerrarlos.

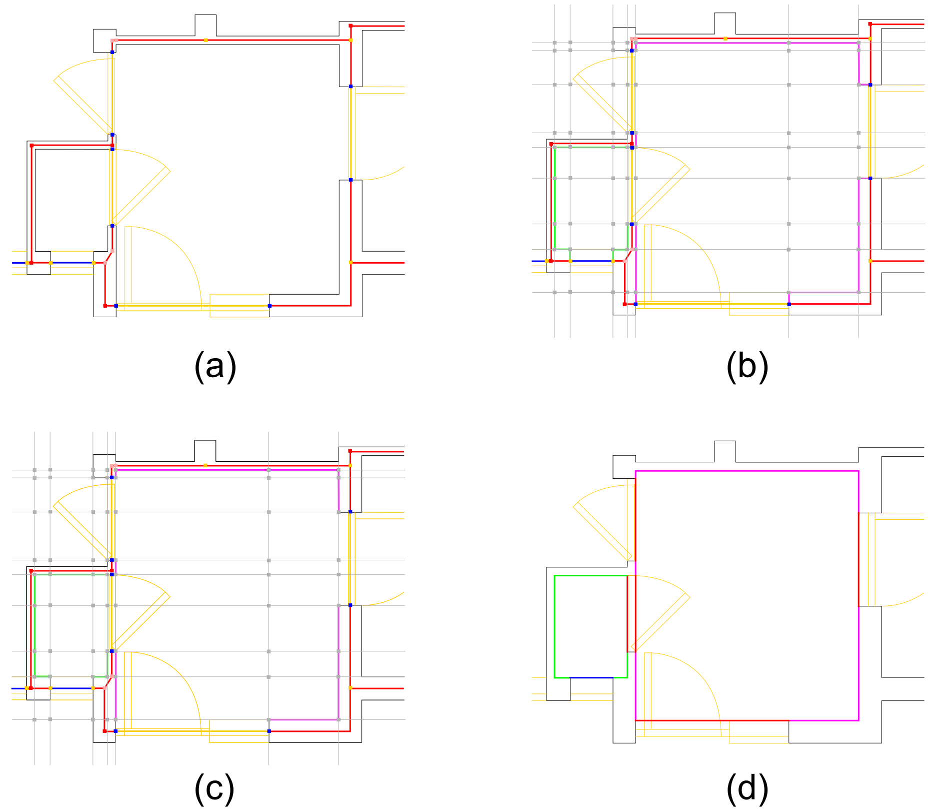


Figura 8.1: Algoritmo de rejilla: (a) Estructura topológica de una habitación. (b) Segmentos asignados a la habitación y a la rejilla. (c) La rejilla se usa para eliminar los segmentos *no acotados*. (d) Polígono interior.

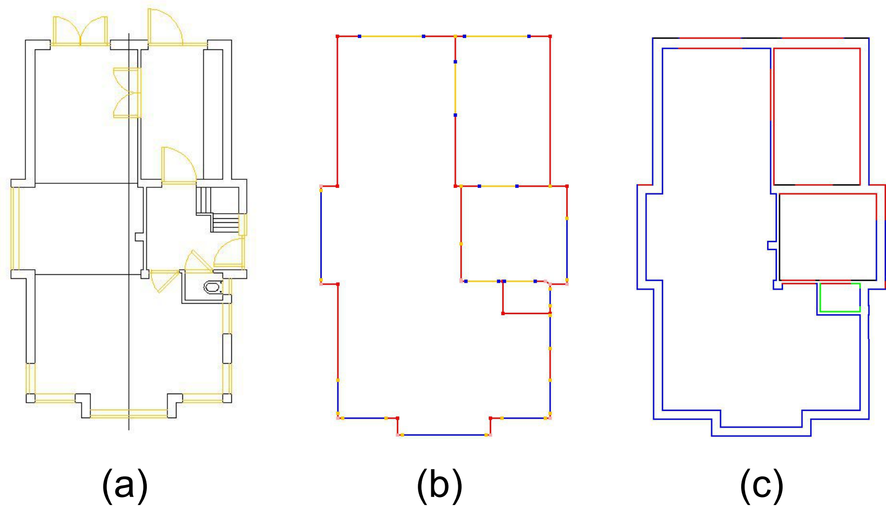


Figura 8.2: (a) Plano arquitectónico de un edificio de cinco plantas; (b) Polígonos exteriores detectados; (c) Polígonos interiores

Capítulo 9

Geometría 3D. Triple extrusión

El algoritmo de triple extrusión permite obtener un modelo 3D esbozado de la forma englobante de un edificio a partir de tres dibujos CAD que representan las tres vistas (planta, alzado y perfil) asumiendo solo dos condiciones de partida: (1) las vistas deben estar dibujadas usando la misma escala y (2) los dibujos deben estar correctamente alineados a los ejes del dibujo.

Además, la triple extrusión se puede ver como un paso intermedio en la generalización de un edificio 3D: partiendo de objetos 3D totalmente detallados, los objetos de la triple extrusión se pueden derivar. Después, modelos de envolvente convexa se pueden derivar de los objetos de triple extrusión. Finalmente, el nivel más simple de generalización son las cajas englobantes 3D de los edificios.

El algoritmo de triple extrusión extrae un polígono de contorno para cada una de las vistas construyendo un camino cerrado que empieza en una línea externa de la vista. Los tres polígonos de contorno son después extruidos a lo largo d los ejes X, Y, Z respectivamente, y la intersección entre los sólidos extruidos se calcula para obtener un sólido que aproxima la forma del edificio. Los sólidos extruidos se representan como modelos B-rep.

El algoritmo de triple extrusión se puede por tanto dividir en estas tareas:

1. Construir un DXF de tres vistas.
2. Importar el DXF y construir índices espaciales.
3. Obtener el polígono de contorno para cada vista.
4. Triangular los polígonos de contorno.
5. Transformar los polígonos de contorno de coordenadas 2D a 3D, teniendo en cuenta algunos puntos de correspondencia.
6. Extruir los polígonos de contorno y calcular la intersección booleana.

A continuación describimos cada subproblema y el enfoque utilizado para resolverlo.

9.1. Construir un DXF de tres vistas

Para calcular ocn precisión la triple extrusión, la primera condición es tener las tres vistas dibujadas a la misma escala. Además, cada vista debe estar en una capa diferente con un nombre prefijado.

9.2. Importación del DXF. Creación de índices espaciales para calcular la intersección entre segmentos

Antes de calcular el contorno para cada vista, los dibujos necesitan ser procesados como sigue:

1. Transformar las primitivas en segmentos rectilíneos. Cada primitiva se transforma en segmentos para posibilitar la detección de contornos.

2. Calcular las intersecciones entre los segmentos.

Con respecto al cálculo de intersecciones, es necesario comprobar cada segmento con todos los demás del dibujo, y dividirlo en caso de que se encuentre una intersección. Este cálculo puede ser costoso computacionalmente en dibujos que representan plantas medianas o grandes, por lo que utilizamos un índice espacial que se describe a continuación.

Para encontrar un equilibrio entre la complejidad de implementación y la efectividad del índice espacial, hemos elegido subdividir cada vista en celdas regulares de una rejilla rectangular. El rectángulo usado como base del índice espacial es la caja englobante de la vista. Cada celda contiene un conjunto de segmentos, y cada segmento está contenido en una o más celdas que lo contienen total o parcialmente.

La Figura 9.1 de la izquierda muestra la estructura del índice espacial de rejilla. La Figura 9.1 de la derecha muestra un ejemplo: cómo calcular las celdas para un segmento con extremos P y Q.

9.3. Detección de contorno de las vistas

El próximo paso del algoritmo de triple extrusión es la detección del contorno de cada vista, con los siguientes pasos:

1. Construir un grafo con los segmentos de cada vista. No tiene por qué tener una única componente conexa.

2. Eliminar los lados colgantes del grafo.

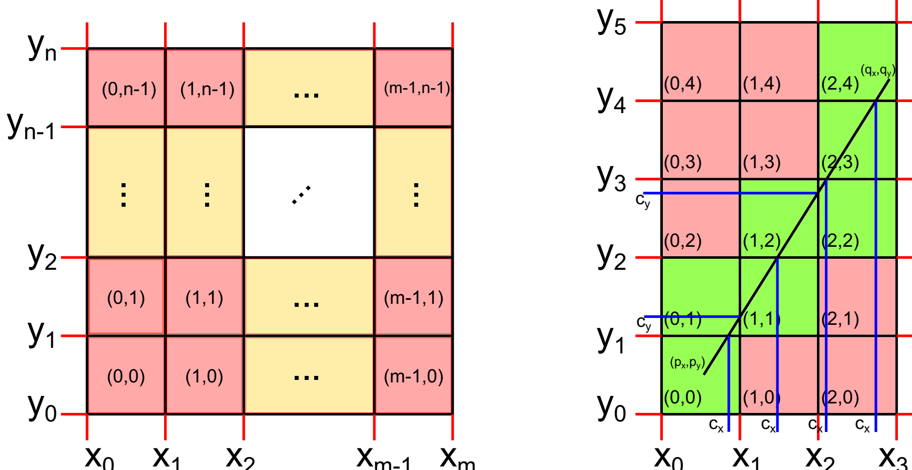


Figura 9.1: Índice espacial de rejilla

3. Encontrar un lado externo en el grafo lanzando un rayo en cualquier dirección y considerando el último lado intersecado por el rayo.
4. Si el grafo tiene más de una componente conexa, se selecciona la que contiene el lado externo detectado.
5. Detectar el contorno de la componente conexa seleccionada.

9.4. Triangulación del polígono de contorno

Antes de extruir los polígonos e intersecar los sólidos obtenidos en la extrusión, calculamos la triangulación de los polígonos de contorno. Hemos optado por un algoritmo de triangulación llamado recorte de oreja[SE02].

9.5. Transformación de coordenadas 2D a 3D

Una vez que los contornos han sido triangulados para cada vista, es necesario transformar las coordenadas 2D del dibujo a 3D, de manera que los contornos se trasladen al espacio 3D de una forma coherente: cada vista debe estar correctamente orientada y alineada en 3D con respecto a las otras vistas.

9.6. Extrusión de las vistas

Después de transformar las coordenadas a 3D, los triángulos 3D son generados. Cada triángulo se extruye a un prisma triangular, como se puede observar en la Figura 9.2.

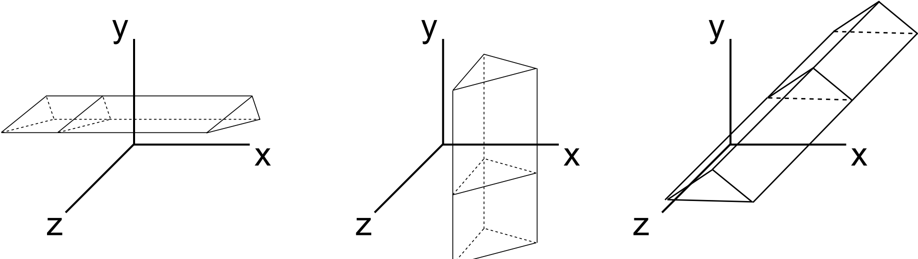


Figura 9.2: Extrusión de las vistas

9.7. Resultados

La Figura 9.3.b muestra el resultado de la detección de contornos en un dibujo de tres vistas; La Figura 9.4 mestra la triple extrusión de la casa (arriba) y algunas vistas del sólido obtenido como resultado de la intersección (abajo).

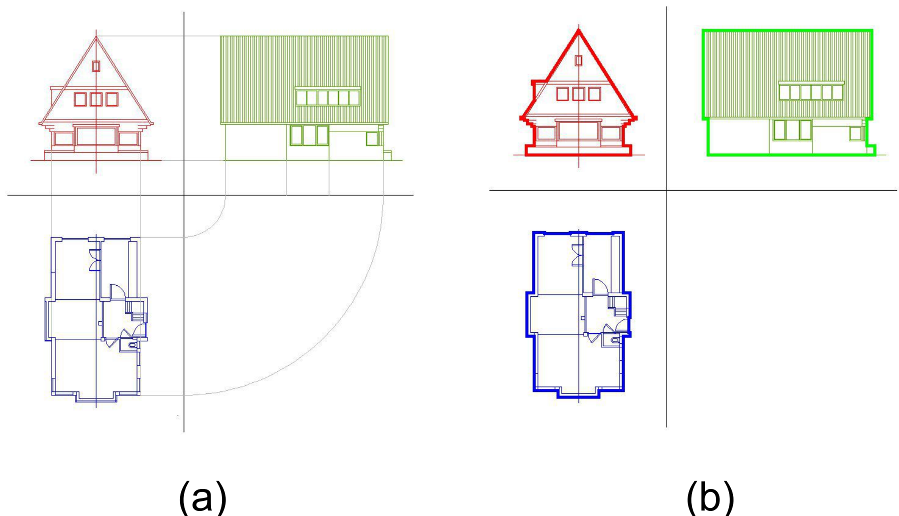


Figura 9.3: (a) Vistas de planta, alzado y perfil de una casa con tejado inclinado; (b) Detección de contorno

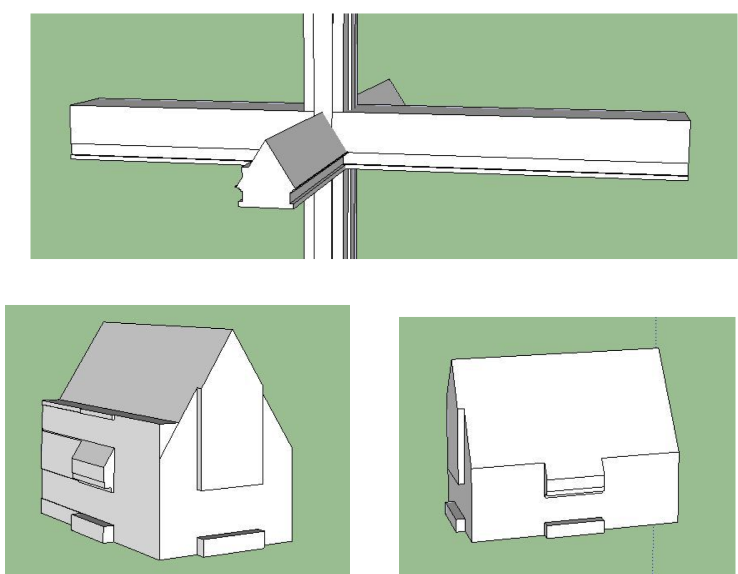


Figura 9.4: Triple extrusión (arriba); Intersección (abajo).

Capítulo 10

Topología y semántica 3D. Generación de modelos CityGML

Para generar un modelo 3D de la planta de un edificio a partir de la información topológica y semántica detectada solamente es necesario mapear la información semántica de nuestra estructura de datos al esquema de representación deseado. Respecto a la estructura CityGML, La Figura 10.1 muestra un diagrama UML simplificado del Módulo de Edificios de CityGML, que contiene las clases relevantes para CityGML junto con las clases de nuestro modelo.

La información sobre el modelo UML de la Figura 10.1 se organiza en cuatro espacios de nombres para facilitar su comprensión:

1. *bldg* contiene las clases del Módulo de Edificios de CityGML que son relevantes para nuestro sistema.
2. *gml* contiene la geometría subyacente a CityGML, de acuerdo con el modelo GML
3. *semantics* contiene las clases del módulo de semántica de la arquitectura presentada en esta tesis.
4. *geom* contiene las clases del módulo de geometría de la arquitectura presentada en esta tesis.

10.1. Resultados

Figure 10.2 muestra el modelo CityGML generado que incluye la semántica detectada.

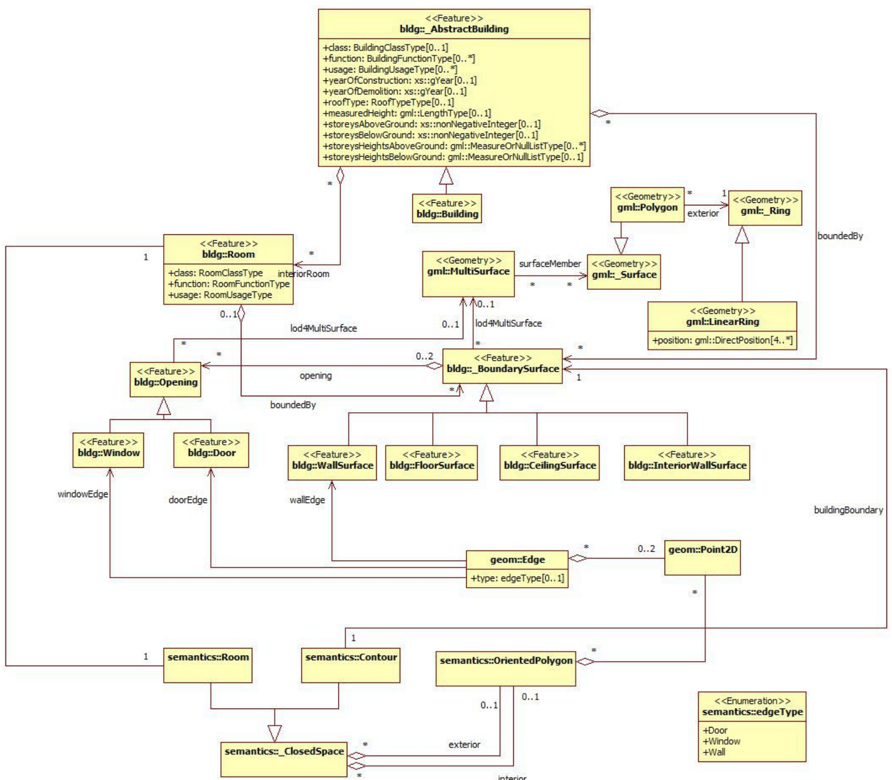


Figura 10.1: Un diagrama UML que representa un submodelo de CityGML y las relaciones de enlace entre CityGML y nuestro modelo. Los espacios de nombres *bldg* y *gml* se usan para etiquetar las clases de CityGML, mientras que los espacios de nombres *semantics* y *geom* se usan para etiquetar las clases de nuestro modelo.

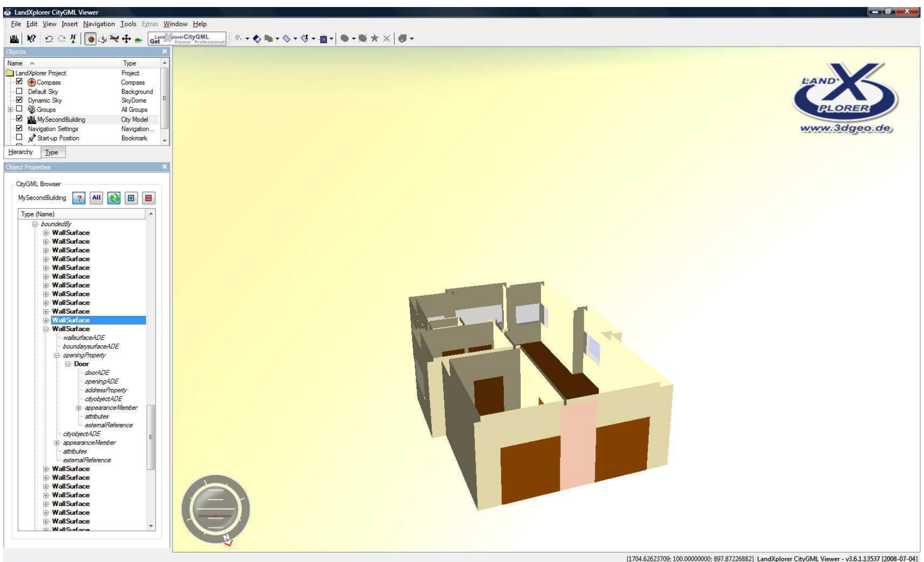


Figura 10.2: Modelo CityGML del edificio. A la izquierda, se muestra un navegador de la estructura semántica; a la derecha se representa el modelo 3D y los elementos seleccionados en el navegador aparecen resaltados.

Parte IV

Conclusiones y trabajo futuro

Esta parte explica brevemente las conclusiones y perspectivas de futuro de este trabajo

Capítulo 11

Conclusiones y trabajo futuro

El último capítulo contiene las conclusiones y el trabajo futuro a desarrollar para mejorar los resultados conseguidos durante la investigación.

11.1. Conclusiones

Este trabajo trata sobre la representación de modelos de información de edificios desde tres puntos de vista: geometría, topología y semántica. Con el fin de conseguir un modelo de representación unificado, (1) hemos analizado una serie de modelos ya existentes, tanto estándares en uso (como CityGML), como propuestas que se pueden encontrar en la literatura sobre el tema; (2) también hemos estudiado una serie de planos CAD de plantas de edificios, conteniendo únicamente información geométrica.

Fruto de estas dos tareas, hemos conseguido los siguientes resultados:

1. El diseño y la implementación de algoritmos para pre-procesar planos arquitectónicos que contienen irregularidades comunes en el proceso de elaboración de los mismos.

2. Los algoritmos antes mencionados han sido abundatemente probados con datos reales, obteniendo resultados satisfactorios.

3. La propuesta de un esquema de tres niveles (geometría, topología y semántica) para la representación, tanto en 2D como en 3D, de modelos de interiores de edificios.

La primera parte de este documento introduce las motivaciones de este trabajo, en el contexto de un proyecto desarrollado en la Universidad de Jaén, así como los trabajos previos en (1) la detección automática de elementos geométricos, topológicos y semánticos en planos CAD ráster y vectoriales; (2) la representación

de edificios en 2D, 2.5D y 3D, conteniendo información geométrica, topológica y semántica.

La segunda parte describe los algoritmos para detectar elementos semánticos en planos arquitectónicos de plantas de edificios. La primera propuesta era la detección local de habitaciones, un algoritmo basado en reglas para detectar habitaciones a partir de dibujos que contienen primitivas de bajo nivel para la representación de paredes y aberturas. La principal ventaja de este algoritmo es la independencia respecto a la disposición de las columnas. Sin embargo, esta propuesta se desechó debido a:

- Sobreentrenamiento del algoritmo.

  El conjunto inicial de reglas se diseñó después de analizar un conjunto reducido de situaciones. En las pruebas se vio que cuanto mayor era el número de planos probados, el número de reglas que se añadían a este conjunto inicial no hacía más que crecer.

- Sólo se cubrían los casos en que la disposición de las paredes era ortogonal.

La segunda propuesta es la detección global de habitaciones. La detección de información semántica se ha dividido en las siguientes tareas: detección de paredes, detección de aberturas y construcción del grafo de topología (utilizando clustering de puntos). Estas tres tareas constituyen la principal aportación de esta tesis. Se han considerado diferentes opciones para cada tarea, y como resultado se han obtenido una serie de algoritmos, mejorados a partir de las pruebas con un conjunto extendido de planos.

Respecto a la detección de paredes, se ha presentado la propuesta del Grafo de Adyacencia de Paredes (Wall Adjacency Graph, o WAG): se trata de un marco teórico que permite detectar todas las posibles situaciones entre un par de segmentos paralelos, y dividirlos para la detección de representaciones de paredes. Este modelo se ha generalizado (WAG generalizado, o GWAG) para trabajar con situaciones en las que hay más de dos segmentos implicados. El resultado de la detección de paredes utilizando estas herramientas es un conjunto de segmentos que representan los ejes centrales de las paredes. Los resultados de estos algoritmos son muy precisos: (1) junto con el preprocesado para detectar y corregir las irregularidades en el plano, el algoritmo de detección de paredes es robusto; (2) la exactitud de la detección de paredes es muy alta, dependiendo sólo de dos parámetros sencillos: las anchuras máxima y mínima de las representaciones de paredes en el plano, dadas por el usuario; (3) el algoritmo es muy rápido, consumiendo apenas unos pocos segundos para procesar el plano de una planta completa de un edificio grande.

Para la detección de aberturas, se ha implementado un algoritmo que analiza cada inserción de bloques representando aberturas, y calcula dos puntos de anclaje respecto de los segmentos que representan las paredes que han sido

obtenidos con los algoritmos anteriores. Este nuevo algoritmo es muy efectivo en un alto porcentaje de las situaciones encontradas en los planos reales con los que se ha trabajado.

El resultado final de la aplicación de estos algoritmos es un primer grafo de topología que incluye paredes y aberturas, aunque sin incluir las intersecciones entre paredes.

Finalmente, los algoritmos de clustering implementados detectan las intersecciones entre paredes para construir un grafo de topología conexo, que puede ser analizado para encontrar espacios cerrados (habitaciones y pasillos). Esta es la parte más compleja de la detección global de habitaciones: el número de casos posibles es imposible de caracterizar, y la exactitud del algoritmo se ve afectada a veces. Sin embargo, hemos comprobado que, con una pequeña participación del usuario, la exactitud del grafo de topología finalmente obtenido puede ser superior al 90

La tercera parte de este documento trata de la creación de un esquema de tres niveles para representar toda la información obtenida con los algoritmos descritos anteriormente, así como la prueba de la corrección topológica de los modelos propuestos. Además, se presenta una propuesta para la generación de una representación aproximada de la geometría 3D de un edificio, denominada triple extrusión; este método se basa en una idea intuitiva: extruir las tres vistas clásicas de un edificio (planta, alzado y perfil), y calcular un sólido 3D a partir de la intersección de las tres extrusiones.

Los principales retos pendientes en esta parte son: (1) la realización de más pruebas del modelo propuesto con planos de edificios reales, y (2) el estudio en profundidad de la triple extrusión, para determinar qué situaciones se detectan correctamente, sus posibles limitaciones, la optimización del cálculo de intersecciones y realizar pruebas con más planos de edificios reales para construir entornos urbanos virtuales.

11.2. Trabajo futuro

Hay algunos temas abiertos que requieren más estudio. En esta sección incluimos una lista de estos temas a la finalización de este trabajo.

11.2.1. Algoritmo de detección de paredes

- Estudio en profundidad de la propiedades teóricas del WAG y el GWAG. Estudio de su uso en otro tipo de problemas no relacionados con la representación de edificios.

- Generalización del WAG a otro tipo de primitivas diferentes de segmentos y arcos de circunferencia.

Tratamiento de paredes de formas complejas: paredes no rectangulares, paredes creadas combinando segmentos y arcos, etcétera.

Reducir la intervención del usuario mediante la detección automática de los anchos de las paredes.

11.2.2. Detección de aberturas

- Implementar algoritmos para tratar con situaciones menos restrictivas: aberturas que no están representadas por bloques, bloques cuya definición no está alineada con los ejes, planos que no están estructurados en capas, etcétera.

Mejorar el tiempo de ejecución del algoritmo de detección de aberturas a partir de conjuntos no estructurados de primitivas, utilizando para ello tests de intersección más eficientes.

11.2.3. Clustering. Construcción del grafo de topología

Mejorar la exactitud de los algoritmos de clustering, mediante pruebas con más planos reales, que conduzcan a la elaboración de métodos de detección más genéricos. Reducción de la intervención del usuario en el proceso.

- Estudio de la estabilidad numérica: manejo de situaciones donde los clusters no se construyen correctamente debido a la elección de umbrales equivocados, que provoca que se fusionen erróneamente puntos, o que se consideren como paralelos segmentos que no lo son. Este tema es complejo, puesto que el grafo de topología que se obtiene en estos casos a menudo contiene inconsistencias que no se pueden detectar visualmente (ciclos, vértices que no se han fusionado correctamente, lados duplicados, etcétera).
- Respecto al algoritmo de crecimiento de líneas: probar y estudiar el algoritmo en profundidad, y resolver el problema de inestabilidad numérica para fusionar puntos o considerar dos segmentos como paralelos.

11.2.4. Esquema de tres niveles para la representación de modelos topológicamente correctos

Los principales trabajos pendientes en esta área incluyen las pruebas con más casos reales, comprobando la corrección topológica. Este esquema, junto con los algoritmos de detección, podría utilizarse para crear grandes entornos virtuales de edificios.

11.2.5. Triple extrusión

El algoritmo de triple extrusión es uno de los últimos desarrollos de este trabajo. Podría llegar a ser una poderosa herramienta para la generación de

modelos urbanos complejos a partir de planos arquitectónicos, pero este algoritmo debe estudiarse con más detalle:

- Estudiar el impacto de las propiedades geométricas de los planos en la exactitud del modelo 3D generado.
- Mejorar el tiempo de ejecución del cálculo de la intersección, e integración en la aplicación. Para este cálculo, hemos utilizado librerías en C++ ya existentes, que requieren la exportación e importación manual de los datos. El uso de algoritmos de intersección específicamente diseñados para este caso es una tarea pendiente.

11.2.6. Otro trabajo futuro

Otras cuestiones abiertas, no incluidas en las secciones anteriores:

- Generalizar el problema de detección de semántica a otros elementos: ascensores, instalación eléctrica, fontanería, mobiliario, espacios abiertos...
- Los algoritmos de detección trabajan a partir del plano de la planta. El uso de las vistas de alzado y perfil para calcular automáticamente algunos parámetros que actualmente proporciona el usuario (altura de las plantas, tamaño de las ventanas, etcétera) es trabajo pendiente.
- Los algoritmos de detección trabajan con la información de una planta. Un tema interesante es la adaptación de los algoritmos y la herramienta software para trabajar con varias plantas del mismo edificio, para así aprovechar la representación de elementos comunes y unirlas a partir de éstos, una vez detectados: tramos de escaleras, ascensores, fachadas, etcétera.

Bibliografía

[AB06] R. Arnaud and M.C. Barnes. *COLLADA: sailing the gulf of 3D digital content creation*. Ak Peters Series. A K Peters, 2006.

[BD07] Don Brutzman and Leonard Daly. *X3D. Extensible 3D graphics for web authors*. Morgan Kaufmann, 2007.

[BG10] Pavel Boguslawski and Christopher Gold. Rapid modelling of complex building interiors. In *Proceedings of the 5th International 3D GeoInfo Conference*, 2010.

[BR09] André Borrmann and Ernst Rank. Specification and implementation of directional operators in a 3d spatial query language for building information models. *Adv. Eng. Inform.*, 23(1):32–44, 2009.

[BZ03] Roland Billen and Siyka Zlatanova. 3d spatial relationships model: a useful concept for 3d cadastre? *Computers, Environment and Urban Systems*, 27(4):411–425, 2003.

[CF08] Christian Clemen and Gielsdorf Frank. Architectural indoor surveying. an information model for 3d data capture and adjustment. In *Proceedings of the American Congress on Surveying and Mapping*, 2008.

[CKHL07] Jin Won Choi, Doo Young Kwon, Jie Eun Hwang, and Jumphon Lertlakkhanakul. Real-time management of spatial information of design: A space-based floor plan representation of buildings. *Automation in Construction*, 16(4):449–459, 2007.

[CL09] Jinmu Choi and Jiyeong Lee. 3d geo-network for agent-based building evacuation simulation. In Jiyeong Lee and Sisi Zlatanova, editors, *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, pages 283–299. Springer Berlin Heidelberg, 2009.

[DGF09] Bernardino Domínguez, Ángel Luis García, and Francisco R. Feito. An Open Source Approach to Semiautomatic 3D Scene Generation for Interactive Indoor Navigation Environments. In *Proceedings of*

IV Ibero-American Symposium on Computer Graphics, pages 131–138, 2009.

[FMW05] Gerald Franz, Hanspeter A. Mallot, and Jan M. Wiener. Graph-based models of space in architecture and cognitive science - a comparative analysis. In *Proceedings of the 17th International Conference on Systems Research, Informatics and Cybernetics*, 2005.

[GS09] T. Germer and M. Schwarz. Procedural arrangement of furniture for real-time walkthroughs. *Computer Graphics Forum*, 28(8):2068–2078, 2009.

[HBW06] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, pages 179–186, New York, NY, USA, 2006. ACM.

[HTGD09] Benjamin Hagedorn, Matthias Trapp, Tassilo Glander, and Jurgen Dollner. Towards an indoor level-of-detail model for route visualization. In *Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, MDM '09, pages 692–697, Washington, DC, USA, 2009. IEEE Computer Society.

[IUA08] Umit Isikdag, Jason Underwood, and Ghassan Aouad. An investigation into the applicability of building information models in geospatial environment in support of site selection and fire response management processes. *Advanced Engineering Informatics*, 22(4):504 – 519, 2008. PLM Challenges.

[LCR10] Xiang Li, Christophe Claramunt, and Cyril Ray. A grid graph-based model for the analysis of 2d indoor spaces. *Computers, Environment and Urban Systems*, 34(6):532 – 540, 2010. GeoVisualization and the Digital City - Special issue of the International Cartographic Association Commission on GeoVisualization.

[LD04] Fabrice Lamarche and Stphane Donikian. Crowd of virtual humans: a new approach for real time navigation in complex and structured environments. *Computer Graphics Forum*, 23:509–518, 2004.

[LM09] Hugo Ledoux and Martijn Meijers. Extruding building footprints to create topologically consistent 3D city models. In A.Krek, M.Rumor, S.Zlatanova, and E.M.Fendel, editors, *Urban and Regional Data Management, UDMS Annuals*, pages 39–48. Taylor & Francis Group, 2009.

[MSK10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics*, 29(6):181:1–181:12, December 2010.

[PB03] Norbert Paul and Patrick Erik Bradley. Topological houses. In *Proceedings of the 16th International Conference of Computer Science and Mathematics in Architecture and Civil Engineering*, 2003.

[PG96] Lutz Pluemer and Gerhard Groeger. Nested maps - a formal, provably correct object model for spatial aggregates. In *Proceedings of the 4th ACM international workshop on Advances in geographic information systems*, GIS '96, pages 76–83, New York, NY, USA, 1996. ACM.

[ROOC+13] María-Dolores Robles-Ortega, Lidia Ortega, Antonio Coelho, Francisco Feito, and Augusto de Sousa. Automatic street surface modeling for web-based urban information systems. *Journal of Urban Planning and Development*, 139(1):40–48, 2013.

[ROOF13] María-Dolores Robles-Ortega, Lidia Ortega, and Francisco Feito. A new approach to create textured urban models through genetic algorithms. *Information Sciences*, 243(0):1 – 19, 2013.

[SE02] Philip J. Schneider and David Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[SLO07] Edgar-Philipp Stoffel, Bernhard Lorenz, and Hans Ohlbach. Towards a semantic spatial model for pedestrian indoor navigation. In Jean-Luc Hainaut, Elke Rundensteiner, Markus Kirchberg, Michela Bertolotto, Mathias Brochhausen, Yi-Ping Chen, Samira Cherfi, Martin Doerr, Hyoil Han, Sven Hartmann, Jeffrey Parsons, Geert Poels, Colette Rolland, Juan Trujillo, Eric Yu, and Esteban Zimányie, editors, *Advances in Conceptual Modeling Foundations and Applications*, volume 4802 of *Lecture Notes in Computer Science*, pages 328–337. Springer Berlin / Heidelberg, 2007.

[SR07] Aidan Slingsby and Jonathan Raper. Navigable space in 3d city models for pedestrians, 2007.

[TBSdK09] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. Rule-based layout solving and its application to procedural interior generation. In *Proceedings of the CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)*, 2009.

[vBdL10] Léon van Berlo and Ruben de Laat. Integration of bim and gis: The development of the citygml geobim extension. In *Proceedings of the 5th International 3D GeoInfo Conference*, 2010.

[vD08] Joost van Dongen. Interior mapping. In *CGI 2008 Conference Proceedings*, 2008.

[vTR07] Christoph van Treeck and Ernst Rank. Dimensional reduction of 3d building models using graph theory and its application in building energy simulation. *Eng. with Comput.*, 23:109–122, April 2007.

[XZZ10] Weiping Xu, Qing Zhu, and Yeting Zhang. Semantic modeling approach of 3d city models and applications in visual exploration. *International Jorunal of Virtual Reality*, 9(3):67–74, 2010.

[YCG10] Wei Yan, Charles Culp, and Robert Graf. Integrating bim and gaming for real-time interactive architectural visualization. *Automation in Construction*, In Press, Corrected Proof:–, 2010.

[ZLF03] G.S. Zhi, S.M. 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*, 35(1):1–14, 2003.